PROGRAMAÇÃO

2019

ESCOLA TÉCNICA MUNICIPAL DE SETE LAGOAS

TÉCNICO EM ELETRÔNICA

Sumário

1. ALGORITMO

A automação é o processo em que uma tarefa deixa de ser desempenhada pelo homem e passa a ser realizada por máquinas, sejam estes dispositivos mecânicos, eletrônicos (como os computadores) ou de natureza mista.

Para que a automação de uma tarefa seja bem-sucedida é necessário que a máquina que passará a realizá-la seja capaz de desempenhar cada uma das etapas constituintes do processo a ser automatizado com eficiência, de modo a garantir a repetibilidade do mesmo.

Assim, é necessário que seja especificado com clareza e exatidão o que deve ser realizado em cada uma das fases do processo a ser automatizado, bem como a sequência em que estas fases devem ser realizadas.

À especificação da sequência ordenada de passos que deve ser seguida para a realização de uma tarefa, garantindo a sua repetibilidade, dá-se o nome de algoritmo.

Ao contrário do que se pode pensar, o conceito de algoritmo não foi criado para satisfazer às necessidades da computação. Pelo contrário, a programação de computadores é apenas um dos campos de aplicação dos algoritmos. Na verdade, há inúmeros casos que podem exemplificar o uso (involuntário ou não) de algoritmos para a padronização do exercício de tarefas rotineiras.

Para que um computador possa desempenhar uma tarefa é necessário que esta seja detalhada passo-a-passo, numa forma compreensível pela máquina, utilizando aquilo que se chama de programa. Neste sentido, um programa de computador nada mais é que um algoritmo escrito numa forma compreensível pelo computador (linguagem de programação).

Outras definições de algoritmos:

“Um conjunto finito de regras que provê uma sequência de operações para resolver um tipo de problema específico” [KNUTH].

“Sequência ordenada, e não ambígua, de passos que levam à solução de um dado problema” [TREMBLAY].

“Processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, as regras formais para a obtenção do resultado ou da solução do problema” [AURÉLIO].

“Conjunto de passos que resolveu o problema” ... “como se fôssemos ensinar uma máquina a fazer alguma tarefa específica” [LOPES].

2. REPRESENTAÇÕES DE UM ALGORITMO

Temos várias formas de representar algoritmos, desde a mais simples, representada por formas, até as mais detalhistas, contendo regras de implementação.

2.1. Descrição Narrativa

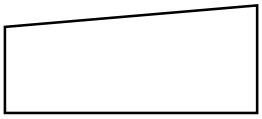
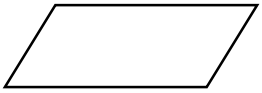
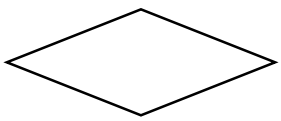
Forma em que os algoritmos são expressos em linguagem natural.

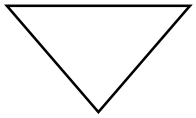
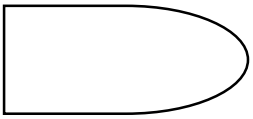
Esta representação é pouco usada na prática porque o uso da linguagem natural muitas vezes dá oportunidade a más interpretações, ambiguidades e imprecisões.

Por exemplo, a instrução "afrouxar ligeiramente as porcas" no algoritmo da troca de pneus está sujeita a interpretações diferentes por pessoas distintas. Uma instrução mais precisa seria: "afrouxar a porca, girando-a 30º no sentido anti-horário".

2.2. Fluxograma

Forma em que os algoritmos são expressos graficamente por formas geométricas diferentes que indicam ações distintas. Utilizado para organizar o raciocínio lógico a ser seguido para a execução de tarefas, resolução de problemas ou para documentar rotinas não muito extensas.

Principais símbolos usados nos fluxogramas e o que representam:	
	Terminal; início e o final do fluxograma.
	Processamento; execução de operações ou ações como cálculos, atribuição de valores, entre outras.
	Entrada de dados para o programa por meio do dispositivo padrão (teclado).
	Saída de dados ou informações por meio do dispositivo padrão (monitor de vídeo).
	Movimento ou transferência de dados entre elementos do sistema.
	Decisão; ação lógica que resultará na escolha de uma das sequencias de instruções.
	Preparação; sub-rotina; ação de preparação para o processamento, ou seja, um processo predefinido.

	Armazenamento; ação de guardar itens.
	Aguardar; Atraso, pausa, *delay*, espera
	Conector; utilizado para interligar partes do fluxograma ou para desviar o fluxo corrente para um determinado trecho do fluxograma.
	Orientação/sentido do fluxo; a sequência pode ser desenvolvida horizontalmente ou verticalmente.

2.3. Pseudocódigo

A linguagem algorítmica, também chamada de pseudocódigo é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer a sintaxe de nenhuma linguagem de programação. Esse meio termo resulta em uma linguagem que se aproxima das construções de uma linguagem de programação, sem exigir, no entanto, rigidez na definição das regras para utilização de suas instruções.

```
algoritmo Média
var
    media, nota1, nota2, nota3 : real
inicio
    leia nota1, nota2 e nota3
    media <- (nota1+nota2+nota3)/3
    se (media >= 6) então
        escreva “aluno aprovado”
    senão
        escreva “aluno reprovado”
    fimse
fimalgoritmo
```

3. Níveis de Linguagens de Programação

As linguagens de programação podem ser classificadas em níveis de linguagens, sendo que os níveis mais baixos são mais próximos da linguagem interpretada pelo processador e mais distante das linguagens naturais.

3.1. Linguagem de Máquina

Lembrando que o computador corresponde basicamente a um conjunto de circuitos, a sua operação é controlada através de programas escritos numa forma bastante primitiva, baseada no sistema binário de numeração tanto para a representação dos dados quanto das operações. A esta forma de representação dos programas, é dado o nome de linguagem de máquina, em razão de ser a forma compreendida e executada pelo hardware do sistema.

As instruções de linguagem de máquina são representadas por códigos que correspondem palavras binárias cuja extensão pode variar de 8 a 64 bits. Dependendo da operação considerada, o código de uma instrução pode simbolizar a operação a ser executada e os dados envolvidos na operação (ou uma referência à localização dos dados).

```
1 0 0 1 1 0 1 0 1 1 0 0 1 0 1 0

0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0

1 0 0 0 0 1 0 1 1 0 1 0 0 1 1 1

0 0 0 1 0 1 1 0 0 1 1 1 0 0 1 0
```

Por uma questão de custo a nível do hardware, as operações representadas pelas instruções de linguagem de máquina são bastante elementares, como por exemplo, a transferência de dados entre memória e registro da CPU, a adição de dois valores, o teste de igualdade entre dois valores, etc... A linguagem de máquina é impraticável para escrita ou leitura. É inviável escrever ou ler um programa codificado na forma de uma *string* de bits.

3.2. Linguagem Hexadecimal

Para simplificar a compreensão e a programação de computadores, num primeiro tempo foi adotado a notação hexadecimal para representar programas em linguagem de máquina, onde a sequência de bits é representada por números hexadecimais.

```
11 1A FB AB 7F 43 27 5B 6C D5 6F 99 FF 10 11 20

39 03 30 39 73 63 F4 3A B4 74 84 AB 7D 6B 54 35

84 47 F3 37 84 50 83 BC 5F 6C 10 39 85 85 94 47

84 03 83 03 83 78 5F FF FF 00 00 00 00 00 00 74
```

Ilustração de um programa em linguagem hexadecimal

A linguagem hexadecimal é, portanto, apenas uma simplificação de notação da linguagem de máquina. Apesar disto, a programação e leitura usando a linguagem hexadecimal continua impraticável.

3.3. Linguagem Assembly

Embora seja a linguagem diretamente executável pelos processadores, a programação de aplicações diretamente em linguagem de máquina é impraticável, mesmo representada na notação hexadecimal. Por esta razão, a linguagem de máquina de cada processador é acompanhada de uma versão “legível” da linguagem de máquina que é a chamada linguagem simbólica Assembly. Simbólica pois esta linguagem não é composta de números binários ou hexadecimais como nas duas linguagens anteriores. A linguagem Assembly é na realidade uma versão legível da linguagem de máquina. Ela utiliza palavras abreviadas, chamadas de mnemônicos, indicando a operação. Abaixo são apresentados dois exemplos de instruções Assembly:

MOV R1, R2 – nesta instrução identifica-se o mnemônico MOV (abreviação de MOVE) e dois registradores como parâmetros: R1 e R2. Quando o processador executa esta instrução, ele comanda o movimento do conteúdo de R2 para R1 (equivalente a instrução Pascal R1=R2, sendo R1 e R2 equivalente a duas variáveis);

ADD R1, R2 – nesta instrução identifica-se o mnemônico ADD (abreviação de ADDITION) e dois registradores como parâmetros: R1 e R2. Quando o processador executa esta instrução, ele comanda a adição do conteúdo de R1 ao conteúdo de R2 e o resultado é armazenado em R1 (equivalente a instrução Pascal R1:=R1+R2).

Escolhendo nomes descritivos para as posições de memória, e usando mnemônicos para representar códigos de operação, a linguagem Assembly facilitou significativamente a leitura de sequências de instrução de máquina. Como exemplo, supomos a operação de dois números inteiros: A=B+C. Esta operação, em um PC, em notação hexadecimal ficaria: A1000203060202A30402. Se associarmos o nome B à posição de memória 200h, C à posição 202h e A à posição 204h, usando a técnica mnemônica, a mesma rotina poderá ser expressa da seguinte forma:

```
MOV AX,B    ; registro AX recebe o valor de memória contida na variável B
ADD AX,C    ; AX recebe a soma de AX (valor de B) com o valor de C
MOV A,AX    ; variável A recebe valor de AX
```

A maioria concorda que a segunda forma, embora ainda incompleta, é melhor que a primeira para representar a rotina. Apesar de oferecer uma representação mais próxima do que o programador está acostumado a manipular, a linguagem Assembly apresenta certas dificuldades para a realização dos programas, tais como a necessidade de definição de um conjunto relativamente grande de instruções para a realização de tarefas que seriam relativamente simples (se representadas através de outras linguagens) e a exigência do

conhecimento de detalhes do hardware do sistema (arquitetura interna do processador, endereços e modos de operação de dispositivos de hardware, etc...).

Por outro lado, a utilização da linguagem Assembly proporciona um maior controle sobre os recursos do computador, permitindo também obter-se bons resultados em termos de otimização de código.

Como a linguagem Assembly é apenas uma versão legível da linguagem de máquina, a passagem de um programa escrito em Assembly para a linguagem de máquina é quase sempre direta, não envolvendo muito processamento. Esta passagem de um programa Assembly para linguagem de máquina é chamada de Montagem, e o programa que realiza esta operação é chamado de montador (Assembler).

A linguagem Assembly é orientada para máquina (ou melhor, para processador), é necessário conhecer a estrutura do processador para poder programar em Assembly. A linguagem Assembly utiliza instruções de baixo nível que operam com registros e memórias diretamente. Assim ela é muito orientada às instruções que são diretamente executadas pelo processador. Na sequência da evolução das linguagens de programação, procurou-se aproximar mais a linguagem de programação à linguagem natural que utilizamos no dia-a-dia: surgiram então as linguagens de alto nível, tipo Pascal, C, C++, etc.

3.4. Linguagem de Alto Nível

As linguagens de alto nível são assim denominadas por apresentarem uma sintaxe mais próxima da linguagem natural, fazendo uso de palavras reservadas extraídas do vocabulário corrente (como *READ*, *WRITE*, *TYPE*, etc.) e permitirem a manipulação dos dados nas mais diversas formas (números inteiros, reais, vetores, listas, etc.); enquanto a linguagem Assembly trabalha com bits, bytes, palavras, armazenados em memória.

As linguagens de alto nível ou de segunda geração surgiram entre o final da década de 50 e início dos anos 60. Linguagens como Fortran, Cobol, Algol e Basic, com todas as deficiências que se pode apontar atualmente, foram linguagens que marcaram presença no desenvolvimento de programas, sendo que algumas delas têm resistido ao tempo e às críticas, como por exemplo Fortran que ainda é visto como uma linguagem de implementação para muitas aplicações de engenharia. Cobol é um outro exemplo de linguagem bastante utilizada no desenvolvimento de aplicações comerciais.

Em comparação com a linguagem Assembly, a passagem de um programa escrito em linguagem de alto nível para o programa em linguagem de máquina é bem mais complexa. Para esta passagem são utilizados compiladores e *linkers*.

Com o desenvolvimento das linguagens de alto nível, o objetivo da independência de máquina foi amplamente alcançado. Dado que os comandos das linguagens de alto nível não referenciam os atributos de uma dada máquina, eles podem ser facilmente compilados tanto em uma máquina como em outra. Assim, um programa escrito em linguagem de alto nível poderia, teoricamente, ser usado em qualquer máquina, bastando escolher o compilador correspondente.

Em realidade, no entanto, provou não ser tão simples. Quando um compilador é projetado, certas restrições impostas pela máquina subjacente são, em última instância, refletidas como características da linguagem a ser traduzida. Por exemplo, o tamanho do registrador e as células de memória de uma máquina limitam o tamanho máximo dos inteiros que nela podem ser convenientemente manipulados. Disso resulta o fato de que, em diferentes máquinas, uma "mesma" linguagem pode apresentar diferentes características, ou dialetos. Consequentemente, em geral é necessário fazer ao menos pequenas modificações no programa antes de move-lo de uma máquina para outra.

A causa deste problema de portabilidade é, em alguns casos, a falta de concordância em relação à correta composição da definição de uma linguagem em particular. Para auxiliar nesta questão, o American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) adotaram e publicaram padrões para muitas das linguagens mais populares. Em outros casos, surgiram padrões informais, devido à popularidade de um dado dialeto de uma linguagem e ao desejo, por parte de alguns autores de compiladores, de oferecerem produtos compatíveis.

3.5. Linguagens estruturadas

Nesta classe, encaixam-se as chamadas linguagens de programação de alto nível surgidas em meados dos anos 60. As linguagens concebidas neste período foram resultado da necessidade da produção de código de programa de forma clara, aparecendo o conceito de estruturação do código (endentação, utilização de letras maiúsculas e minúsculas nos identificadores, eliminação de instruções "problemáticas" como o "*go to*", etc.).

O período compreendido entre a década de 60 e a de 80 foi bastante produtivo no que diz respeito ao surgimento de linguagens de programação, o que permitiu o aparecimento de uma grande quantidade de linguagens as quais podem ser organizadas da seguinte forma:

- as linguagens de uso geral, as quais podem ser utilizadas para implementação de programas com as mais diversas características e independente da área de aplicação considerada; encaixam-se nesta categoria linguagens como Pascal, Modula-2 e C;

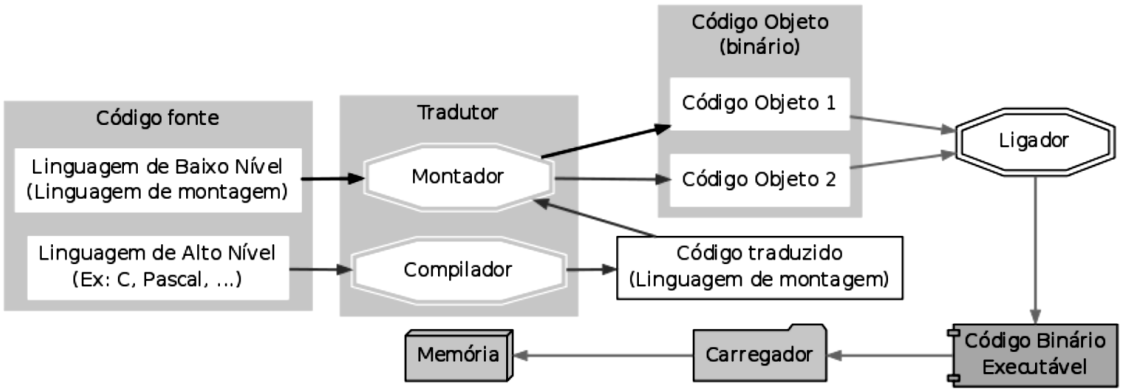
- as linguagens especializadas, as quais são orientadas ao desenvolvimento de aplicações específicas; algumas das linguagens que ilustram esta categoria são Prolog, Lisp e Forth;
- as linguagens orientadas a objeto, que oferecem mecanismos sintáticos e semânticos de suporte aos conceitos da programação orientada a objetos; alguns exemplos destas linguagens são Smalltalk, Eiffel, C++ e Delphi.

4. TRADUTORES E INTERPRETADORES

Do mesmo modo que você precisaria de um tradutor para poder lidar com uma linguagem que não consegue entender, os computadores também necessitam de um tradutor para traduzir um programa escrito em linguagem de programação para um programa correspondente em linguagem de máquina.

Dois softwares básicos são responsáveis por realizar a tradução em questão: os tradutores e os interpretadores.

Os tradutores podem ser classificados como montadores e compiladores. Quando o processo de tradução converte um programa que se encontra no nível de linguagem de montagem (representação simbólica da linguagem de máquina, ex.: linguagem Assembly) para a linguagem de máquina, o tradutor utilizado é o montador. Já na tradução de programas em linguagem de alto nível para a linguagem de montagem, o software responsável é o compilador. Perceba que não há tradução direta da linguagem de alto nível para a linguagem de máquina. Para que esta seja alcançada, são necessários vários passos intermediários, sendo um deles a tradução para a linguagem de montagem.



No processo de compilação, cada parte de um programa (módulo) escrito em linguagem de alto nível é traduzido para um módulo objeto diferente, que consiste em sua representação em linguagem de montagem. Antes de serem traduzidos para linguagem de máquina pelo montador, é necessário que os vários módulos objetos sejam integrados de modo a formarem

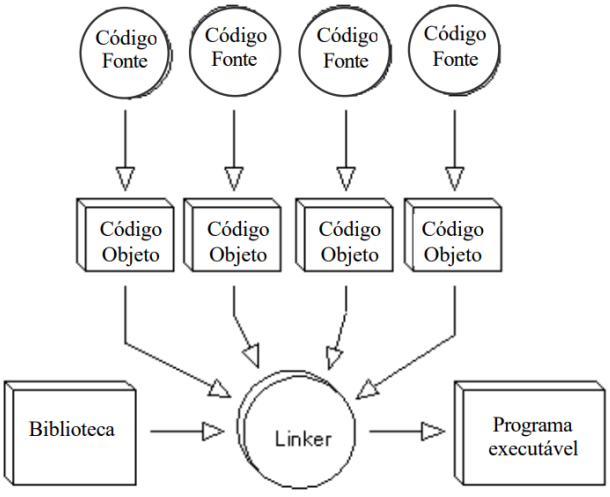
um único código. Por fim, carrega o programa na memória, a fim de tornar suas instruções prontas para serem executadas pelo processador.

Os interpretadores, além de realizar a tradução de um programa para a linguagem de máquina, ainda executam suas instruções. Assim que traduz uma instrução, ela é imediatamente executada, gerando assim um ciclo de tradução e execução que prossegue de instrução a instrução até o fim do programa

Por não traduzir um programa escrito em linguagem de alto nível diretamente para linguagem de máquina, o processo de compilação tende a ser mais rápido que o processo de interpretação. Além disso, uma vez compilado, um programa pode ser executado várias vezes sem a necessidade de haver uma nova compilação. Já na interpretação, cada vez que um programa tiver que ser reexecutado, todo o processo de interpretação deverá ser refeito, independentemente de ter ocorrido modificações no código fonte do programa desde sua última execução. A vantagem da interpretação fica por conta da possibilidade de testar os programas ao mesmo tempo em que são desenvolvidos.

4.1. Editores de ligação

A tarefa realizada pelo editor de ligações, ou *linker* como é mais conhecido é rearranjar o código do programa, incorporando a ele todas as partes referenciadas no código original, resultando num código executável pelo processador. Esta tarefa pode ser feita também pelos chamados carregadores.



4.2. Depuradores ou *debuggers*

Os *debuggers* são assim chamados devido à sua função essencial que é de auxiliar o programador a eliminar (ou reduzir) a quantidade de “bugs” (erros) de execução no seu programa. Eles executam o programa gerado através de uma interface apropriada que possibilita uma análise efetiva do código do programa graças à:

- execução passo-a-passo (ou instrução por instrução) de partes do programa;
- visualização do “estado” do programa através das variáveis e eventualmente dos conteúdos dos registros internos do processador;
- alteração em tempo de execução de conteúdos de memória ou de variáveis ou de instruções do programa;
- etc...

4.3. Compilador Versus Interpretador

Um programa compilado geralmente executa mais rapidamente que um programa interpretado. A vantagem do interpretador é que ele não necessita passar por um estágio de compilação durante a qual as instruções de máquina são geradas. Este processo pode consumir muito tempo se o programa é longo. O interpretador, por outro lado, pode executar imediatamente os programas de alto-nível. Por esta razão, os interpretadores são algumas vezes usados durante o desenvolvimento de um programa, quando um programador deseja testar rapidamente seu programa. Além disso, os interpretadores são com frequência usados na educação, pois eles permitem que o estudante programe interativamente.

Tanto os interpretadores como os compiladores são disponíveis para muitas linguagens de alto nível. Mas, Java, Basic e LISP são especialmente projetadas para serem executadas por um interpretador.

Embora se obtenha um programa mais lento, algumas linguagens interpretadas têm outra vantagem: a portabilidade. Como não é gerado um código de máquina, e sim um código intermediário que será interpretado por uma máquina virtual, pode-se obter a portabilidade do código se esta máquina virtual for desenvolvida para várias plataformas (computadores diferentes). Este é o caso da linguagem Java.

5. Paradigma de Programação

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa. Os paradigmas representam abordagens fundamentalmente diferentes para a construção de soluções para os problemas, portanto afetam todo o processo de desenvolvimento de software. A seguir serão descritos os paradigmas de programação clássicos:

5.1. Paradigma imperativo

Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa. Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens.

5.2. Paradigma declarativo

Descreve propriedades da solução desejada, não especificando como o algoritmo em si deve agir. Muito popular em linguagens de marcação, sendo utilizado na programação das páginas web (linguagem HTML) e descrição de documentos multimídia como a linguagem *Nested Context Language* – NCL, adota pelo padrão brasileiro de TV Digital.

5.3. Paradigma funcional

Trata a computação como uma avaliação de funções matemáticas. Ela enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa. Neste paradigma ao se pensar em uma função simples de cálculo de médias de notas, usamos o auxílio de funções mais primitivas, podendo a função Média (Números) ser representada pela expressão:

(Divide (Soma Números) (Conta Números))

Logo a função Divide opera com os resultados das funções Soma e Conta.

5.4. Paradigma orientado a objeto

Neste paradigma, diferente do paradigma imperativo, os dados passam a ter um papel principal na concepção do algoritmo. No paradigma imperativo, rotinas de controle manipulam os dados que são elementos passivos. Já na orientação a objetos, os dados são considerados objetos auto gerenciáveis formados pelos próprios dados e pelas rotinas responsáveis pela manipulação destes dados.

6. PROGRAMA EM C

É estruturada em um conjuntos blocos de códigos e instruções, delimitados com chaves ({ ... }). Um bloco também pode conter outros blocos.

Uma instrução corresponde a uma ação executada, e deve sempre terminar com ponto e vírgula (;).

O compilador ignora espaços, tabulações e quebras de linha no meio do código; esses caracteres são chamados de **espaço em branco** (*whitespace*) mas, com o objetivo indicar a hierarquia dos elementos, usamos regras de escrita (**Indentação do código)** que padroniza a estética do programa, auxiliando a leitura e interpretação.

A linguagem **é sensível** à utilização de maiúsculas e minúsculas. Por exemplo, se você escrevesse Printf no lugar de printf, ocorreria um erro, pois o nome da função é totalmente em minúsculas.

A linguagem C possui um total de 32 palavras conforme definido pelo padrão ANSI.

Palavras reservadas são escritas em minúsculo e não podem ser utilizadas para outro propósito. Alguns compiladores incluem outras palavras reservadas como, asm, cdecl, far, fortran, huge, interrupt, near, pascal, typeof.

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

Tipos de dados

Para realizar tarefas nos algoritmos computacionais manipulamos valores. Tipo de dado é como pode ser representado computacionalmente característica ou estado atual de algo do mundo real.

Cada linguagem de programação define sua forma de representar e manipular esses dados, que podem ser classificados em dois grandes grupos: os tipos de dados primitivos (números inteiros e reais, lógicos ou booleanos e caracteres) e os tipos de dados não primitivos (strings, vetores, matrizes, classes, estruturas, etc.).

O tipo de dado determina a natureza, o tamanho e a faixa de representação de um valor, como pode ser visto na tabela a seguir:

ESPECIFICADOR DE TIPO	NATUREZA	TAMANHO	IMAGEM
void	Vazio	0	
char	Caractere	1	-128 a 127
signed char	Caractere com sinal	1	-128 a 127
unsigned char	Caractere sem sinal	1	0 a 255
int	Inteiro	2	-32.768 a 32.767
signed int	Inteiro com sinal	2	-32.768 a 32.767
unsigned int	Inteiro sem sinal	2	0 a 65.535
short int	Inteiro curto	2	-32.768 a 32 767
signed short int	Inteiro curto com sinal	2	-32.768 a 32.767
unsigned short int	Inteiro curto sem sinal	2	0 a 65.535
long int	Inteiro long	4	-2.147.483.648 a 2.147.483.647
signed long int	Inteiro longo com sinal	4	-2.147.483.648 a 2.147.483.647
unsigned long int	Inteiro longo sem sinal	4	0 a 4.294.967.295
float	Ponto flutuante com precisão simples	4	3.4 E-38 a 3.4E+38
double	Ponto flutuante com precisão simples	8	1.7 E-308 a 1.7E+308
long double	Ponto flutuante com precisão dupla longo	16	3.4E-4932 a 1.1E+4932

Bibliotecas

Uma biblioteca é um arquivo contendo um conjunto de funções (pedaços de código) já implementados e que podem ser utilizados pelo programador em seu programa. O comando **#include** é utilizado para declarar as bibliotecas que serão utilizadas pelo programa. Esse comando diz ao pré-processador para tratar o conteúdo de um arquivo especificado como se o seu conteúdo houvesse sido digitado no programa no ponto em que o comando **#include** aparece.

Biblioteca	Descrição
<assert.h>	Macro para ajudar na detecção de erros lógicos e outros tipos de erros em versões de depuração de um programa.
<complex.h>	Conjunto de funções para manipular números complexos.
<ctype.h>	Funções usadas para classificar caracteres pelo tipo ou para converter entre caixa alta e baixa independentemente da codificação.
<errno.h>	Teste de códigos de erro reportados pelas funções de bibliotecas.
<fenv.h>	Controle de ponto flutuante.
<float.h>	Constantes de propriedades específicas de implementação da biblioteca de ponto flutuante, como a menor diferença entre dois números de ponto flutuante distintos (_EPSILON), a quantidade máxima de dígitos de acurácia (_DIG) e a faixa de números que pode ser representada (_MIN, _MAX).
<inttypes.h>	Conversão precisa entre tipos inteiros.
<iso646.h>	Programação na codificação de caracteres ISO 646.
<limits.h>	Constantes de propriedades específicas de implementação da biblioteca de tipos inteiros, como a faixa de números que pode ser representada (_MIN, _MAX).
<locale.h>	Constantes para setlocale() e assuntos relacionados.
<math.h>	Funções matemáticas comuns em computação.
<setjmp.h>	Macros setjmp e longjmp, para saídas não locais.
<signal.h>	Tratamento de sinais.
<stdarg.h>	Acesso dos argumentos passados para funções com parâmetro variável.
<stdbool.h>	Definição do tipo de dado booleano.
<stdint.h>	Definição de tipos de dados inteiros.
<stddef.h>	Diversos tipos e macros úteis.
<stdio.h>	Manipulação de entrada/saída.
<stdlib.h>	Diversas operações, incluindo conversão, geração de números pseudo-aleatórios, alocação de memória, controle de processo, sinais, busca e ordenação.
<string.h>	Tratamento de cadeia de caracteres.
<tgmath.h>	Funções matemáticas.
<time.h>	Conversão de tipos de dado de data e horário.
<wchar.h>	Manipulação de caractere wide, usado para suportar diversas línguas.
<wctype.h>	Classificação de caracteres wide.

O comando **#include** permite duas sintaxes:

- **#include <nome_da_biblioteca>**: o pré-processador procurará pela biblioteca nos caminhos de procura pré-especificados do compilador. Usa-se essa sintaxe quando estamos incluindo uma biblioteca que é própria do sistema, como as bibliotecas **stdio.h** e **stdlib.h**;
- **#include "nome_da_biblioteca"**: o pré-processador procurará pela biblioteca no mesmo diretório onde se encontra o nosso programa. Podemos ainda optar por informar o nome do arquivo com o caminho completo, ou seja, em qual diretório ele se encontra e como chegar até lá.

Arquivos de bibliotecas tem extensão **.h**.

Exemplos do uso do comando **#include**:

```
#include <stdio.h>
#include "D:\Programas\soma.h"
```

Comentários

Muitas vezes é bastante útil colocar comentários no código, por exemplo para esclarecer o que uma função faz, ou qual a utilidade de um argumento, etc. A maioria das linguagens de programação permite comentários; em C, eles podem aparecer de duas maneiras:

```
/* Comentários que podem
   ocupar várias
   linhas.
*/
```

e

```
// Comentários de uma linha só, que englobam

// tudo desde as duas barras até o final da linha.
```

Tudo que estiver entre as marcas /* e */ ou entre // será ignorado pelo compilador

Escopo

É o nível em que um dado pode ser acessado; em C há dois níveis: **local** e **global**. Uma variável *global* pode ser acessada por qualquer parte do programa; variáveis *locais* podem ser acessadas apenas dentro do bloco onde foram declaradas (ou nos seus sub-blocos), mas não fora dele (ou nos blocos que o contêm). Isso possibilita que você declare várias variáveis com o mesmo nome mas em blocos diferentes.

Identificador

Nomeia constantes, variáveis e funções. É composto por um ou mais caracteres alfanuméricos (letras e números), o primeiro caractere deve ser obrigatoriamente uma letra ou um sublinhado. Não pode haver espaço entre os caracteres.

Constante

Valor fixo que não se modifica ao longo do tempo durante a execução do programa. Uma constante pode ser de qualquer um dos tipos primitivos de dados.

Declaração:

```
#define identificador valor;
const tipo_de_dado identificador = valor;
```

Exemplos:

```
#define PI 3.14159
#define MAIORIDADE 18
const int aposentadoria = 65;
const int codigo = 12440;
```

Variáveis

Codinome de determinada posição de memória onde se pode guardar qualquer valor de um tipo associado.

O valor atribuído a uma variável pode ser alterado ao longo do tempo durante a execução do programa, entretanto ela só pode armazenar um único valor a cada instante.

A escolha do tipo de variável define o conjunto de possíveis valores de serem nesse espaço de memória, além do conjunto de operadores que podem agir sobre este dado.

Para utilizar uma variável, é necessário solicitar previamente a reserva do espaço necessário para armazenar seus dados, isso é através da declaração da variável. Nesse momento já é possível atribuir algum valor à mesma.

- É possível ter identificador igual ao de outra variável já existente em escopo superior, porém é recomendado não usar variáveis iguais, por dificultar entendimento do programa principalmente em caso de manutenção ou correção;
- A declaração e uso de variáveis é sensível à utilização de maiúsculas e minúsculas (case sensitive).
- Podemos declarar um conjunto de variáveis de mesmo tipo numa única instrução.

Declaração:

```
tipo_de_dado identificador;
tipo_de_dado identificador = valor;
```

Funções

É um bloco de código que realiza uma tarefa. Quando queremos realizar essa tarefa, simplesmente fazemos uma *chamada de função* para a função correspondente.

Uma função pode precisar que o programador dê certos dados para realizar a tarefa; esses dados são chamados **argumentos**. A função também pode retornar um valor, que pode indicar se a tarefa foi realizada com sucesso, por exemplo; esse valor é o **valor de retorno**.

Em C, para chamar uma função, devemos escrever o seu nome, seguido da lista de argumentos (separados por vírgula) entre parênteses, mesmo que não haja nenhum argumento. Lembre que a chamada de função também é uma instrução, portanto devemos escrever o ponto-e-vírgula no final. Alguns exemplos de chamadas de funções:

```
funcao(arg1, arg2, arg3);

funcao();
```

Se quisermos saber o valor de retorno de uma função, podemos armazená-lo numa variável. Variáveis serão introduzidas logo adiante, mas a sintaxe é muito fácil de aprender:

```
valor_de_retorno = funcao(arg1, arg2);
```

Todo o código (exceto as declarações de variáveis e funções) deve estar dentro de funções. Todo programa deve ter pelo menos uma função, a função **main**.

Operadores

Operadores Aritméticos

Tabela: Operadores aritméticos

Operador	Finalidade	Exemplo	Resultado
+	Adição	5 + 2	7
-	Subtração	5 – 2	3
*	Multiplicação	5 * 2	10
/	Divisão (Quociente)	5 / 2	2
%	Divisão Euclidiana (Resto)	30 % 7	2

Notar o último operador. Notar que são operadores que operam apenas com 2 operandos (operadores binários).

Na divisão euclidiana temos 30 dividido 7 tem por quociente 4 e como resto 2.

Operadores relacionais

Permite fazer comparações lógicas de ordenação de números, e ainda de letras (mas não strings)

Operador	Significado
>	Maior que
<	Menor que
>=	Maior ou igual à
<=	Menor ou igual à
==	Igual a
!=	Diferente de

Operadores lógicos

Operador	Nome
&&	And
||	Or
!	Not

Operadores Lógicos Bit a Bit

Operador	Ação lógica
&	AND (E)
¦	OR (OU)
^	XOR (OR exclusivo/OU exclusivo)
~	NOT
>>	Deslocamento de bits à direita
<<	Deslocamento de bits à esquerda

Estrutura de Seleção

Frequentemente são expostos problemas onde é necessário selecionar uma opção de fluxo do sistema, dentre duas ou mais possíveis. Para estas situações são usados os comandos que formam a Estrutura de Seleção (SE-SENÃO, SE-SENÃO ANINHADO e ESCOLHA).

Um programador iniciante pode ficar em dúvida de qual a melhor Estrutura de Seleção escolher para cada caso. Aqui vão algumas dicas básicas para ajudar a definir a escolha.

Primeiro deve-se considerar a seguinte regra da Lógica de Programação: “Sempre que uma condição é verdadeira, a linha de código imediatamente abaixo é executada”. Isso facilita a montagem da condição a ser testada pela Estrutura de Seleção Escolhida. Por exemplo:

```
SE (condição a ser testada) ENTÃO
    entra aqui se for a condição for verdadeira
SENÃO
    entra aqui se a condição for falsa
FIM SE
```

Dessa forma, coloca-se no bloco SE os comandos que serão executados caso a condição seja verdadeira. Uma vez executado o bloco SE, seu respectivo bloco SENÃO é ignorado. Isso melhora o desempenho do sistema, uma vez que não será gasto tempo de processamento.

Quando não se conhece os valores exatos para atender a uma condição verdadeira (maior que, menor que, maior ou igual a, etc.), a Estrutura de Seleção SE-SENÃO deve ser utilizada. Ela permite definir o intervalo de teste. Isto vale para todas as linguagens de programação. Por exemplo:

```
SE (idade >= 18) ENTÃO
    entra aqui se for a condição for verdadeira
SENÃO
    entra aqui se a condição for falsa
FIM SE
```

Quando se tem condições binárias, onde existem apenas duas opções de continuação do fluxo, o recomendado é o uso da Estrutura de Seleção SE-SENÃO simples, como visto anteriormente. Porém, existem podem existir mais de duas opções de continuação do fluxo do sistema. Neste cenário podem ser utilizadas duas outras estruturas de seleção SE-SENÃO ANINHADO ou ESCOLHA.

A regra dos valores exatos se aplica aqui também. Quando não se conhece os valores exatos da condição verdadeira a Estrutura de Seleção SE-SENÃO ANHINADO deve ser utilizado. Vale sempre lembrar que quando um bloco SE é executado, seu SENÃO correspondente é ignorado. Por exemplo:

```
SE (idade >=18) ENTÃO
    SE (consumo <= 200) ENTÃO
        comandos...
    SENÃO
        comandos...
    FIM SE
SENÃO
    comandos...
FIM SE
```

Quando todos os possíveis valores de uma determinada condição são conhecidos, pode-se usar a estrutura ESCOLHA. Essa estrutura melhora consideravelmente o desempenho do sistema, pois não testa todas as condições. A estrutura ESCOLHA ainda permite definir um fluxo padrão caso nenhuma das condições estabelecidas sejam verdadeiras. Por exemplo:

```
ESCOLHA gols
    0 : comandos do fluxo 1
    1 : comandos do fluxo 2
    2 : comandos do fluxo 3
SENÃO
    comandos do fluxo alternativo
FIM ESCOLHA
```

É evidente que variações e combinações podem ser montadas conforme a necessidade. A forma incorreta de montar uma Estrutura de Seleção pode impactar o fluxo do sistema, modificando inclusive a regra de negócios. O desempenho também pode ser impactado quando muitas condições são testadas sem necessidade.

Estruturas de Repetição

Independentemente da linguagem de programação escolhida para o desenvolvimento do sistema, existem três tipos básicos Estrutura de Repetição: PARA-FAÇA, ENQUANTO-FAÇA e REPITA. A questão básica para quem está iniciando no mundo da programação de computadores é definir qual a diferença entre elas e quando utilizá-las. Aqui segue algumas dicas de quando e por que utilizar cada uma delas.

Sempre vale lembrar a seguinte regra da lógica de programação: “Sempre que uma condição é verdadeira, a linha de código imediatamente abaixo é executada”. Isso fará diferença na hora de decidir a melhor Estrutura de Repetição.

Quando sabemos exatamente quantas vezes um determinado conjunto de instruções deve ser executado, o ideal é utilizar a estrutura PARA-FAÇA. Nessa estrutura é possível definir o começo e o fim da repetição, ou seja, definir exatamente quantas vezes será executada. A Estrutura de Repetição PARA-FAÇA utiliza-se de um “contador” que controla o número de execuções. No exemplo abaixo o contador “mês” começa em 1 e termina em 12, fazendo com que o bloco de comandos em seu interior seja executado doze vezes:

```
PARA mês de 1 até 12 FAÇA
    comandos...
FIM PARA
```

O contador da estrutura PARA-FAÇA pode ser tanto incremental (quando seu valor é crescente) quanto decremental (quando seu valor é decrescente). A própria estrutura é

responsável por manipular o valor do contador. Ele testa o valor do contador antes da execução. Se estiver dentro do limite definido o bloco é executado, caso contrário, todo o bloco é ignorado e o ponteiro de execução do fluxo é colocado na instrução imediatamente seguinte. Este princípio é o mesmo utilizado na validação das Estruturas de Seleção.

Existe situações que não é possível determinar quantas vezes uma determinada instrução será executada. Neste caso pode-se utilizar a estrutura ENQUANTO-FAÇA ou REPITA. Estas estruturas diferem apenas na obrigatoriedade de execução, de pelo menos uma vez, do bloco de comandos em seu interior.

Se não é possível determinar o número de repetições e o bloco de comandos não tem obrigatoriedade de execução então a melhor Estrutura de Repetição a ser utilizada é ENQUANTO-FAÇA. Essa estrutura valida a condição antes da execução e, se for verdadeira, executa o bloco de comandos em seu interior enquanto tal condição for verdadeira. Ao final da execução do bloco o ponteiro de execução do fluxo é colocado novamente no início do bloco e a condição é testada outra vez. Por exemplo:

```
ENQUANTO consumo < 200 FAÇA
    comandos...
FIM ENQUANTO
```

É importante assegurar que a condição testada possa ser modificada dentro bloco, passando de verdadeira para falsa, caso contrário o sistema entra em LOOP infinito, “travando” seu sistema.

Se o número de repetições é indefinido, mas a execução de pelo menos uma vez é necessária então a melhor Estrutura de Repetição a ser utilizada é REPITA. Essa estrutura executará o bloco de comandos uma vez e ao final fará o teste condicional. Se a condição testada for verdadeira o ponteiro de fluxo de execução será colocado na instrução seguinte, saindo da Estrutura de Repetição (regra básica da lógica de programação). Nessa estrutura a repetição se dá enquanto a condição for FALSA, pois somente assim o ponteiro de fluxo poderá retornar ao início do bloco de repetição para uma nova execução. Por exemplo:

```
REPITA
    comandos...
ATÉ estoque = 1
```

Da mesma forma do ENQUANTO-FAÇA, é importante permitir a modificação da condição testada dentro do bloco de repetição como forma de evitar repetições infinitas.