**Apostila – Algoritimo e Lógica**

Sumário

[Sumário 1](#_Toc167195218)

[1. Introdução à programação 3](#_Toc167195219)

[1.1 Como funciona um computador 3](#_Toc167195220)

[1.2 O que é um programa? 4](#_Toc167195221)

[2. Lógica de Programação 4](#_Toc167195222)

[3. Algoritmos 5](#_Toc167195223)

[3.1 Conceitos básicos 5](#_Toc167195224)

[3.2 Tipos de Algoritmos 6](#_Toc167195225)

[3.2.1 Descrição Narrativa 6](#_Toc167195226)

[3.2.2 Pseudocódigo 7](#_Toc167195227)

[3.2.3 Fluxograma 7](#_Toc167195228)

[4. Codificação/Compilação 10](#_Toc167195229)

[4.1 Linguagem de Programação 10](#_Toc167195230)

[4.1.1 Sintaxe 11](#_Toc167195231)

[4.1.2 Semântica 11](#_Toc167195232)

[4.2 Compilação 12](#_Toc167195233)

[5. Terminologia 12](#_Toc167195234)

[6. Linguagem C 13](#_Toc167195235)

[6.1 Estrutura de um programa em C 13](#_Toc167195236)

[6.1.1 Bibliotecas 14](#_Toc167195237)

[6.1.2 main 15](#_Toc167195238)

[6.1.3 Comentários 15](#_Toc167195239)

[6.1.4 Indentação 15](#_Toc167195240)

[6.1.5 Defina e Ifdef 16](#_Toc167195241)

[6.2 Variáveis simples 16](#_Toc167195242)

[6.2.1 Tipos de dados 16](#_Toc167195243)

[6.2.2 Identificadores 18](#_Toc167195244)

[6.2.3 Declaração de uma variável 19](#_Toc167195245)

[6.2.4 Inicializando uma variável 19](#_Toc167195246)

[6.2.5 Classe de armazenamento 20](#_Toc167195247)

[6.3 Operadores 20](#_Toc167195248)

[6.3.1 Operadores aritméticos 20](#_Toc167195249)

[6.3.2 Operador de Atribuição 21](#_Toc167195250)

[6.3.3 Operadores relacionais 21](#_Toc167195251)

[6.3.4 Operadores lógicos 22](#_Toc167195252)

[6.3.5 Tabela de preferência 22](#_Toc167195253)

[6.4 Funções de Biblioteca 23](#_Toc167195254)

[6.4.1 Funções de entrada e saída de dados 23](#_Toc167195255)

[6.5 Comandos 30](#_Toc167195256)

[6.5.1 Comandos de seleção 30](#_Toc167195257)

[6.5.2 Comandos de repetição 37](#_Toc167195258)

[6.5.3 Teste de mesa 41](#_Toc167195259)

[6.6 Vetores 42](#_Toc167195260)

[6.6.1 Vetores unidimensionais 43](#_Toc167195261)

[6.6.2 Vetores de char (strings) 47](#_Toc167195262)

[6.6.3 Vetores multidimensionais 50](#_Toc167195263)

[6.6.4 Preenchendo e imprimindo um vetor 54](#_Toc167195264)

[6.7 Funções 54](#_Toc167195265)

[6.7.1 Nome da função 55](#_Toc167195266)

[6.7.2 Parâmetros e argumentos 55](#_Toc167195267)

[6.7.3 Bloco de comandos 56](#_Toc167195268)

[6.7.4 Retorno da função 56](#_Toc167195269)

[7. Bibliografia 58](#_Toc167195270)

1. Introdução à programação

A programação inicia-se com a escrita do programa (código-fonte) e encerra com a geração de um programa executável.

* 1. Como funciona um computador

O computador é um sistema complexo, uma máquina composta de vários circuitos e componentes eletrônicos que recebe e manipula dados para obter e fornecer o resultado com um ou mais objetivos.

Para realizar essa função, o computador responde a um bloco de instruções especificar de uma maneira bem definida e pode executar uma lista pré-gravada dessas instruções. Esse trabalho realizado pelo computador se chama processamento e essas listas pré-gravadas são chamadas de programas, por isso pode-se dizer que o computador é programável (2019, Marçula).

O sistema de um computador é composto basicamente de dois elementos: hardware, a parte física, e o software, a parte lógica.

Em resumo, o hardware de um computador possui os seguintes componentes[[1]](#footnote-1):

1. Unidade de Processamento Central (CPU): é o componente crucial de um sistema de computação, responsável por todas as operações do computador e é responsável pelo processamento direto ou indireto dos dados, executando instruções internas de acordo com as instruções externas que recebe dos programas, possui duas unidades chamadas: Unidade de Controle (UC) e Unidade Lógica e Aritmética (ULA).
2. Memória: armazena dados e informações que serão utilizados no processamento, o computador possui dois tipos de memórias: Memória secundária, que armazenam os dados por um longo período, e memória principal, que é onde a CPU armazeno os dados e instruções que estão sendo manipulados e o seu conteúdo é perdido quando o computador é desligado, diferente da memória secundária (Nesta apostila, quando falar em memória está se referindo a memória principal)
3. Unidade de Entrada: é um dispositivo que permite que o usuário interaja com o computador, fornecendo-lhe dados e informações que serão processadas (EVARISTO, Jaime), por exemplo: teclado, mouse, *touch screen*.
4. Unidade de Saida: é um dispositivo que serve para que sejam fornecidos ao usuário do computador os resultados do processamento realizado (EVARISTO, Jaime), por exemplo: monitor, impressora, caixa de som, etc.

Já o software, também chamados de programas, é o conjunto de instruções que o processador irá processar, manipulando os dados recebidos através do desempenho desejado e produzindo resultados.

* 1. O que é um programa?

Para que um computador possa desempenhar uma tarefa ele deve executar um programa, escrito numa forma compreensível pela máquina, permitindo que realize tarefas complexas, automatize processos e resolva problemas de forma eficiente

Programar é basicamente estruturação de dados e construção de algoritmos e, em resumo, suas principais etapas são:

1. Análise: estudar o problema a ser resolvido pelo programa e definir, através da *lógica de programação*, os dados de entrada, o conjunto de instruções e os dados de saída;
2. Criar o *Algoritmo*: ferramentas do tipo descrição narrativa, fluxograma ou português estruturado utilizadas para descrever o problema com suas soluções, aplicando a lógica de programação da 1ª etapa;
3. *Codificação/Compilação*: algoritmo é transformado em códigos da linguagem de programação escolhida para se trabalhar.
4. Lógica de Programação

A lógica está presente no nosso dia a dia, para realizar qualquer atividade é seguida uma sequência lógica, o que normalmente é feito de maneira tão natural que passa despercebido, mas ao analisar bem é possível enxergar os passos que levaram ao resultado.

Para realizar atividades diárias, cada pessoa define a sua própria sequência de, podendo incluir ou remover alguns já definidos.

Quando o assunto é programar, definir as etapas do que deve ser feito assume uma grande importância, já que para desenvolver um programa ou rotina a ser executada pelo computador, é necessário deixar bem claro a sequência que deve ser seguida para atingir o resultado esperado. Esse encadeamento lógico na programação é chamado de Lógica de Programação.

A lógica de programação está presente logo na primeira etapa para começar a criar um programa, é a base de todo o desenvolvimento de software e envolve a capacidade de resolver problemas de forma estruturada e eficiente, identificando padrões, organizando informações e criando soluções elegantes para desafios específicos

É a habilidade de pensar de forma lógica e estruturada, decompondo um problema complexo em etapas mais simples, com o objetivo de criar algoritmos claros e eficientes, que possam ser traduzidos em código de programação.

A lógica de programação desempenha um papel fundamental no desenvolvimento de software e na resolução de problemas, pois é essencial para: resolução de problemas, abordando-os de forma sistemática; eficiência, consumindo menos recursos computacionais; facilidade de manutenção e atualização de código, economizando tempo e recursos.; e adaptação a novas linguagens de programação, permitindo que os programadores aprendam mais rapidamente.

Os algoritmos e a lógica de programação estão intrinsicamente ligados, uma vez que aqueles são a implementação prática desta

Ao praticar a lógica de programação, aprende-se a criar algoritmos sólidos, envolvendo a identificação de variáveis, estruturas de controle e o uso de estruturas de dados apropriadas.

Um programador habilidoso não é apenas alguém que sabe escrever código, mas também alguém que sabe pensar de forma lógica, criando algoritmos elegantes que economizam tempo e recursos.

1. Algoritmos
   1. Conceitos básicos

No dia a dia, toda e qualquer tarefa a ser realizada segue uma sequência de instruções a serem executadas para que se consiga atingir o objetivo predeterminado, essa sequência obedece a uma lógica.

Por exemplo:

- Objetivo predeterminado: trabalhar pela manhã

- Sequência de instruções:

1. Acordar.

2. Tomar banho

3.Vestir-se

4. Tomar café

5. Tirar o carro da garagem

Essa sequência de instruções bem definidas e não ambíguas que levam à solução de um problema ou tarefa é chamada de algoritmo, estando presente em toda a vida moderna, desde a ciência da computação até a vida cotidiana.

Um algoritmo deve ter:

* + Precisão: As instruções devem ser claras e inequívocas, sem espaço para interpretações dúbias.
  + Não ambiguidade: Cada passo deve ter um único significado e não levar a interpretações diferentes.
  + Finitude: O algoritmo deve ter um número finito de instruções e terminar em um tempo finito
  + Entrada de dados (INICIO): atribui valores ao algoritmo, a partir de especificações de variáveis e funções antes de sua iniciação.
  + Saída de informações (FIM): representa os valores após o processamento, ou seja, a execução das ações.
  + Eficiência: O algoritmo deve ser eficiente no uso de recursos, como tempo e memória.
  + Generalidade: O algoritmo deve ser capaz de resolver um problema genérico, não apenas um caso específico.

Para especificar um algoritmo, primeiro é preciso conhecer e identificar o problema a ser resolvido e, depois, terá que encontrar uma sequência lógica de instruções, que, por fim, permitirá chegar a uma solução que atenda às convenções para a sua elaboração.

As fases de um bom algoritmo para a criação de um programa são:

1. Inicialização: são atribuídas as informações/dados iniciais
2. Processamento: aquisição dos valores de dados de entrada e é feita a manipulação dos dados recebidos
3. Finalização: fornecimento/impressão dos resultados.
   1. Tipos de Algoritmos

Os algoritmos podem ser expressos de várias maneiras, sendo os mais comuns: descrição narrativa, pseudocódigo e fluxograma, estes dois últimos os mais utilizados na programação.

* + 1. Descrição Narrativa

Após a análise do problema, descreve as instruções utilizando uma linguagem natural, por exemplo a língua portuguesa, os passos a serem seguidos para a resolução do problema.

Um exemplo de algoritmo projetado com descrição narrativa:

Tarefa: trocar a lâmpada

INICIO

1º passo: pegar a lâmpada

2º passo: pegar a escada

3º passo: posicionar a escada embaixo da lâmpada

4º passo: subir na escada com a lâmpada

5º passo: retirar a lâmpada queimada

6º passo: colocar a lâmpada nova

7º passo: descer da escada

8º passo: ligar o interruptor

9º passo: guardar a escada

10º passo: jogar a lâmpada velha no lixo

FIM

* + 1. Pseudocódigo

É uma forma de representação de algoritmos é rica em detalhes e assemelhar-se bastante à forma em que os programas são escritos, é considerada uma linguagem de programação falsa que usa a linguagem natural para representar cada passo

Muito útil para pensar no problema sem os detalhes da linguagem de programação.

Exemplo:

Algoritmo: Calculo\_Media

INICIO

Variáveis n1,n2,media

LEIA n1,n2

media ← (N1 + N2) / 2

SE media >= 7 ENTAO

Escreva “Aprovado”

SENAO

Escreva “Reprovado”

FIMSE

FIM

* + 1. Fluxograma

Um fluxograma é um tipo de diagrama, uma representação gráfica onde formas geométricas diferentes implicam ações (instruções, comandos) distintos, ilustra de forma descomplicada a sequência de execução dos elementos.

Um fluxograma deve ser claro, simples e de fácil leitura, a direção do fluxo é de cima para baixo ou da esquerda para a direita.

Os principais símbolos usados em um fluxograma são:

|  |  |
| --- | --- |
|  | **Fluxo de dados**  Indica o sentido do fluxo de dados  Conecta os demais símbolos |
|  | **Terminal**  Indica o INICIO ou o FIM de um algoritmo |
|  | **Processamento**  Indica processamentos em geral  Ex.: calculo de dois números  Admite apenas uma saída de fluxo |
|  | **Entrada/Saída** (genérica)  Operação de entrada e saída de dados  Ex: leitura ou gravação de arquivos |
|  | **Desvio** (conector)  Permite o desvio para um ponto qualquer do programa |
|  | **Entrada manual**  Indica entrada de dados via teclado |
| Diagrama  Descrição gerada automaticamente | **Exibir/Saída**  Mostra informações ou resultados |
|  | **Decisão**  Permite elaborar processos de decisão  Admite apenas uma entrada de fluxo |

Há um único símbolo inicial por onde a execução do algoritmo começa, e um ou mais símbolos finais, que são pontos onde a execução do algoritmo se encerra.

Diagrama

Descrição gerada automaticamente

Os fluxogramas para algoritmos de programação estruturada são escritos considerando apenas três estruturas básicas:

1. Sequência:O fluxo de execução das instruções de modo linear, uma após a outra. Deve haver apenas um caminho possível no conjunto de instruções

Exemplo:

Diagrama

Descrição gerada automaticamente

1. Seleção:Execução seletiva de um grupo de instruções baseada em alguma condição, representada por uma expressão lógica ou relacional. Criam alternativas no fluxo de execução das instruções do algoritmo. Em tempo de execução, apenas das alternativas será escolhida

Exemplo:

Diagrama

Descrição gerada automaticamente

1. Repetição: Execução seletiva de um grupo instruções até que alguma condição seja satisfeita. O fluxo da execução pode realizar várias repetições de um mesmo conjunto de comandos, antes de prosseguir para a próxima etapa.

Exemplo:

Diagrama

Descrição gerada automaticamente

1. Codificação/Compilação

Um programa de computador é a codificação de um algoritmo, ou seja, a conversão de suas declarações em um comando ou instrução específico de uma certa linguagem de programação. Um programa completo, escrito em uma linguagem de programação, é chamado de código.

Após a codificação, é necessário gerar um programa executável pelo computador, esse processo é chamado de compilação.

* 1. Linguagem de Programação

A linguagem de programação é uma linguagem criada para instruir um computador, ou seja, é um conjunto de palavras e um conjunto de regras gramaticais usados para instruir o sistema de computação a realizar tarefas específicas.

A primeira e mais primitiva linguagem de programação criada é chamada de linguagem de máquina, que é a linguagem própria do computador e utiliza apenas dois símbolos (0 e 1)[[2]](#footnote-2), denominados *bit*, na qual todo programa deve ser convertido antes de ser executado.

Apesar de ser a linguagem que o hardware entende de forma direta, é muito mais complicada e difícil para o programador, em razão disso foram criadas linguagens de programação utilizando símbolos alfabéticos, ou seja, mais próximas a linguagem natural do ser humano.

As linguagens de programação que necessitam de comandos escritos mais próximos da linguagem de máquina são chamadas de linguagens de baixo nível, a mais conhecida é a linguagem de montagem (*assembly*).

No *assembly* cada instrução produz exatamente uma instrução de máquina, ou seja, há uma correspondência um-para-um. Para traduzir um programa escrito em *assembly* para a linguagem de máquina é realizada a montagem por um programa chamado montador, é a tradução mais rápida e simples que existe.

Já as linguagens mais próximas da linguagem natural, estruturadas de acordo com a compreensão e a intenção do programador, e que refletem mais os procedimentos utilizados na solução de problemas, são chamadas de linguagens de alto nível.

Uma linguagem de alto nível permite que o programador especifique a realização de ações do computador com muito menos instruções.

Exemplo:

Algoritmo:

1- Carregar o valor de A

2- Somar B

2- Subtrair C

3- Guardar o resultado em X

Quadro comparativo:

|  |  |  |
| --- | --- | --- |
| **Linguagem de máquina** | **Assembly** | **C** |
| 0001 0010 0011  0011 0010 0100  0100 0010 0101  0010 0010 0110 | LOAD A  ADD B  SUB C  STR X | X=A+B-C |

Para passar um algoritmo de linguagem natural para uma linguagem de programação estruturada de alto nível, como o “C”, é preciso, além de estabelecer o passo a passo, seguir a forma como essa linguagem interpretará essas instruções de maneira que sejam compreendidas pelo computador.

Cada linguagem tem o seu conjunto de palavras-chave e sintaxes.

* + 1. Sintaxe

Para que seja traduzida uma instrução escrita com caracteres de algum idioma para instruções escritas na linguagem de máquina, é necessário que cada instrução seja escrita de acordo com regras preestabelecidas, chamadas sintaxe da instrução e quando não são obedecidas ocorre erro de sintaxe.

* + 1. Semântica

Cada instrução tem uma finalidade específica, ou seja, a execução de uma instrução resulta na realização de alguma ação, esta é chamada semântica da instrução.

Infelizmente, um programa pode não conter erros de sintaxe, mas a sua execução pode não fornecer como saída o resultado esperado para alguma entrada. Neste caso, dizemos que o programa contém erros de lógica.

* 1. Compilação

Compilação é o processo de conversão do programa escrito em linguagem de programação (código-fonte) para a linguagem de máquina (código-objeto). Esse processo é realizado por um programa chamado compilador.

Diagrama

Descrição gerada automaticamente

Outro aspecto importante é que um programa pode utilizar códigos ou dados criados previamente e armazenados em arquivos conhecidos como biblioteca ([Capítulo 3, i, 1](#biblioteca)).

O processo de compilação deve agrupar todas essas partes, essa parte do processo é realizada por um módulo da linguagem de programação conhecido como lincador, que reúne o programa objeto com as bibliotecas referenciadas e gera o código executável.

Diagrama

Descrição gerada automaticamente

1. Terminologia

Alguns termos que são muito usados ao estudar uma linguagem de programação podem ser desconhecidos e gerar confusão, a seguir estão listados alguns

* Declaração/Instrução: unidade de código que tem um efeito, como criar uma variável ou exibir um valor, em C, toda instrução deve ser encerrada com ponto e vírgula ‘;’.
* Comando: Um comando é uma estrutura de programação, representa uma instrução ou ação, podem ser sequenciais, de decisão ou de repetição.
* Expressão: Expressão uma combinação de variáveis, constantes e operadores, e que, uma vez avaliada, resulta num valor.
* Função: Função é um conjunto de instruções e/ou declarações desenhadas para cumprir uma determinada tarefa e referenciá-la por um nome. Existem funções de biblioteca previamente definidas e funções definidas pelo programador. Os parênteses após o nome\_da\_funcao indica que ela é uma função.

1. Linguagem C

Surgiu na década de 70, criada por Dennis Ritchie, quando havia a necessidade de uma linguagem de alto nível que fosse multiplataforma e linguagem mais simples que a *assembly*, já que cada processador tinha sua própria linguagem *assembly*.

Atualmente é relevante em sistemas operacionais, sistemas desktop, sistemas embarcados, telecomunicação etc.

Uma das suas vantagens é possuir características tanto de “alto nível” quanto de “baixo nível”.

As principais características da linguagem C são:

* 1. Multiplataforma: fosse possível programar em um único código que servisse para multiplataformas tanto do ponto de vista de arquitetura de computadores, hardware e processador distintos.
  2. Desempenho: apresenta um alto desempenho, é compilada.
  3. Proximidade aos recursos de hardware: consegue trabalhar de maneiras mais específica e poderosa
  4. Sintaxe rebuscada: negativo por causa de curva de aprendizagem, positivo na migração para outras linguagens.
  5. Baixa aplicabilidade em web e mobile

É *case sensitive*, o quer dizer que diferencia letras maiúsculas e minúsculas, tanto para o nome de funções, variáveis e comandos de linguagem, se os comandos foram escritos em maiúscula o compilador não poderá interpretá-los corretamente

* 1. Estrutura de um programa em C

Geralmente, o primeiro passo ao se iniciar a elaboração de um programa é a inclusão de bibliotecas cujas funções serão usadas no programa ([Capítulo 3, i, 1](#biblioteca)).

Todo programa em C deve conter uma função identificada por main, com lista de parâmetros vazia e tipo de dado não obrigatório. Dentro dessa função deve estar as instruções do programa ([Capítulo 3, i, 2](#main)).

Por fim, apesar de não ser obrigatório, para finalizar o programa basta escrever *return 0*, o que será explicado mais tarde.

Exemplo de um programa em C:

**Código**

#include<stdio.h>

int main ()

{

printf(“hello, word”);

return 0;

}

**Tela**



* + 1. Bibliotecas

Biblioteca, também chamada arquivo de cabeçalho (header file), é uma coleção de funções, onde armazena o nome de cada função, o código-objeto da função e mais informações necessárias para o processo de *linkedição*

A extensão padrão das bibliotecas é *.h*

Sintaxe para inclusão de biblioteca: *#include<biblioteca>*

* + - 1. stdio.h

O *stdio* vem de *standard input output*, entrada e saída padrão, nessa biblioteca estão as funções e comandos para entrada e saída de dados da linguagem C. Portanto, não pode esquecer de inserir esta linha nos programas.

* + - 1. math.h

Fornece um conjunto de funções para operações matemáticas, tais como funções trigonométricas, hiperbólicas, logaritmos, potência e arredondamentos.

Todas as funções da biblioteca *math.h* retornam um valor do tipo double.

Na tabela abaixo apresentamos algumas funções presentes na biblioteca *math.h*

|  |  |
| --- | --- |
| **Função** | **Descrição do comando** |
| floor( ) | arredonda para baixo |
| ceil( ) | arredonda para cima |
| sqrt( ) | Calcula raiz quadrada |
| pow(variável, expoente) | Potenciação |
| sin( ) | Seno |
| cos( ) | Cosseno |
| tan( ) | Tangente |
| log( ) | logaritmo natural |
| log10( ) | logaritmo base 10 |

* + - 1. locale.h

Serve para fazer a adaptação de um programa às características de um determinado idioma ou de uma região.

A linguagem C utiliza o arquivo de cabeçalho *locale.h* para implementar a localização de programas.

A utilização do arquivo *locale.h* e da função setlocale() configurada adequadamente vai garantir que caracteres como “ç” e acentuação sejam exibidos normalmente em nosso programa.

Sintaxe: *setlocale(LC\_ALL,"Portuguese");*

* + - 1. windows.h

Nesta biblioteca tem as funções e comandos que servem apenas o sistema operacional Windows. A função mais utilizada é a de pausar o programa:

Sintaxe: *system("pause);*

* + 1. main

Obrigatoriamente, todo programa deve possuir uma função *main*, a qual é a primeira função a ser executada, essa função marca o ponto de início da execução do programa.

Assim como qualquer função, o corpo da *main* é sempre definido entre chaves "{ }", chamados de delimitadores, e o conteúdo entre as chaves será executado sequencialmente quando o programa for executado.

Sintaxe:

*int main ()*

*{*

*Declarações*

*}*

* + 1. Comentários

Os comentários são blocos de textos que são ignorados pelos compiladores e que podem ser inseridos com o objetivo de documentar um programa, eles auxiliam a entender o funcionamento dos programas.

No C há duas formas de fazer comentários:

1. Comentários de apenas uma linha, o compilador vai ignorar apenas aquela linha

Sintaxe:

*// comentário*

1. Comentários em blocos, com mais de uma linha, o compilador vai ignorar tudo o que estiver entre /\* e \*/

Sintaxe:

*/\**

*Comentário*

*\*/*

* + 1. Indentação

Indentar um código nada mais é que separa os códigos em blocos através da tabulação, buscando escrever cada instrução em uma linha e a sequência vinculada a uma estrutura com uma tabulação diferente da tabulação diferente.

Considerando que os compiladores da linguagem C não consideram espaços nem mudanças de linha na hora de compilar um programa, esta forma de se editar um programa facilita sobremaneira a legibilidade dos programas.

Seja um mesmo programa fosse digitado das seguintes formas:



#include <stdio.h>

int main (){printf("Hello, world!\n");}



#include <stdio.h>

int main ()

{

printf("Hello, world!\n");

}

Os dois programas seriam executados executado da mesma, porém a legibilidade do primeiro estaria prejudicada.

* + 1. Defina e Ifdef
  1. Variáveis simples

Uma variável é uma posição na memória cujo conteúdo pode ser modificado durante a execução de um programa, o tamanho do espaço reservado para aquela posição depende do tipo de dado que vai ser armazenada.

* + 1. Tipos de dados

O tipo de dado associado a uma variável é o conjunto dos valores que podem ser nela armazenados.

A linguagem C possuí cinco tipos básicos de dados:

1. Char

O tipo char é composto por caracteres simples, é o mais simples na linguagem C, uma vez que reserva um único byte na memória do computador e guarda apenas o código em binário da tabela ASCII/Unicode.

Importante lembrar que na memória do computador não existem caracteres e sim números guardados. A tabela ASCII mostra esta equivalência.

Na linguagem C não há um tipo especial para armazenar cadeia de caracteres (strings).

Código de formatação: *%c*

1. Int

O tipo int são dados numéricos que não possuem componentes decimais ou fracionários, ou seja, guarda apenas números inteiros

A variável "int" reserva 4 bytes na memória do computador e guarda valores entre -2147483648 e 2147483647.

Código de formatação: %d

1. Float

O tipo float são números que têm componente decimal ou fracionário, ou seja, números reais.

A variável "float" reserva 4 bytes na memória do computador, e guarda valores entre -3.4E-38 a 3.4E+38. Esta variável também é chamada na literatura como "números reais em ponto flutuante".

O número depois do E é a potência da base 10, exemplo: 3.4E+38 = 3,4\*1038.

Código de formatação: %f

1. Double

O tipo double são valores em ponto flutuante de precisão dupla, que apresentam alcance mais extenso, com capacidade para armazenar um faixa de valores maior que o "float". Para esta variável utiliza-se a palavra reservada "double" para declará-la. Ela guarda valores entre -1.7E-307 a 1.7E+308.

Código de formatação: %lf

1. Void

O tipo void significa valores que ocupam 0 bits e não possuem valor algum.

* + - 1. Tabela dos tipos de daos

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Guarda** | **Memoria** | **Faixa de valores** |
| Char | Apenas um caractere | 1 byte | conjunto de caracteres ASCII ou valores inteiros de -128 a +127. |
| Int | Apenas um número inteiro | 4 bytes | -2147483648 a 2147483647. |
| Float | Apenas um número real em ponto flutuante | 4 bytes | -3.4E-38 a 3.4E+38. |
| double | Apenas um número real em ponto flutuante | 8 bytes | -1.7E-307 a 1.7E+308. |

* + - 1. Modificadores/qualificadores

Os qualificadores são utilizados para as variáveis do tipo inteira e caractere.

 São eles:

Short: valores curtos. Os valores inteiros são guardados nos mesmos 2 bytes do inteiro normal.

Long: valores longos. Os valores são guardados em 4 bytes de memória o que amplia a faixa de valores.

Unsigned: valores sem sinal. Transforma os valores negativos em positivos dobrando assim a faixa de valores guardados em memória.

Aplicando os qualificadores nas variáveis acima:

|  |  |  |
| --- | --- | --- |
| Inteiro curto – hd | short int ou short | -32768 a +32767 |
| Inteiro longo – ld | long int ou long | -2147483648 a 2147483647 |
| caractere Sem sinal - uc | unsigned char | 0 a 255 |
| Inteiro curto sem sinal - hu | unsigned short int ou unsigned short | 0 a 65535 |
| Inteiro Sem sinal – ud | unsigned int ou unsigned | 0 a 4294967295 |
| Inteiro longo sem sinal - lu | unsigned long int ou unsigned long | 0 a 4294967295 |

Obs.: Na função printf para imprimir um valor "long" utiliza-se a letra "l" associada a letra "d" para um valor "short' e a letra "h" junto com a letra "d". Para os valores sem sinal utiliza-se a letra "u" para o "int" a letra "l" associada a letra "u" para o "long" sem sinal e a letra "h" junto com a letra "u" para o "short' sem sinal.

* + 1. Identificadores

Os identificadores consistem nos nomes que utilizamos para representar variáveis, constantes, tipos, funções e rótulos do programa.

Um identificador é uma sequência de uma ou mais letras, dígitos ou sublinhas (\_), que começa com uma letra ou sublinha.

Em geral, evita-se iniciar um identificador com sublinhas, pois, este tipo de notação é reservado para o compilador.

Importante lembrar que a linguagem C é *case sensitive* e diferencia letras maiúsculas de minúsculas.

Existem 3 regras básicas para se dar um nome a uma variável, são elas:

1. Todo nome de variável deve começar por uma letra ou o caractere sublinha(\_).
2. Os caracteres seguintes podem ser letras, números ou o caractere sublinha. É usual em C utilizar letras minúsculas para nome de variáveis e letras maiúsculas para constantes simbólicas
3. Existem certas palavras reservadas que formam a linguagem C que não podem ser utilizadas.
   * + 1. Palavras reservadas

As palavras reservadas, também conhecidas como palavras-chave, são identificadores que possuem uso específico para a linguagem C, ou seja, têm significados especiais e não podem ser utilizados para a nomeação de variáveis, funções etc.

São elas: auto, break, case, char, continue, default, do, double, else, entry, extern, float, for, goto, if, int, long, register, return, short, sizeof, static, struct, switch, typedef, union, unsigned, while.

* + 1. Declaração de uma variável

Antes de usar uma variável, ela deve ser declarada, ou seja, é preciso informar ao computador que é preciso separar uma posição na memória e, para ele saber tamanho do espaço que deve ser reservado, também é preciso informar o tipo de dado que será armazenado naquela posição.

Portanto, para declarar uma variável precisa primeiro especificar o tipo do dado que será armazenado, apenas depois disso deve colocar o nome da variável <identificador>.

Sintaxe geral:

<tipo\_de\_dado> <identificador>;

Exemplo:

|  |  |
| --- | --- |
| int abacaxi; | //declaração de uma variável do tipo int com o nome “abacaxi” |
| float uva; | //declaração de uma variável do tipo float com o nome “uva” |

É possível declarar mais de uma variável especificando apenas uma vez o tipo, desde que todas as variáveis da lista sejam do mesmo tipo, para isso, deverão ser separadas por virgulas:

<tipo\_de\_dado> <identificador\_var1>, <identificador\_var2>;

Exemplo:

|  |  |
| --- | --- |
| int abacaxi,banana; | //declaração de uma lista com duas variáveis do tipo int, com os seguintes nomes: “abacaxi” e “banana |
| float uva,manga,pera; | //declaração de uma lista com três variáveis do tipo float, com os seguintes nomes: “uva”, “manga” e “pera” |

* + 1. Inicializando uma variável

Quando uma variável é declarada, o computador apenas reserva um espaço na memória para aquela variável, ou seja, não atribui qualquer valor.

É importante lembrar que todo espaço da memória do computado já foi usando antes por outros programas que deixaram vestígios e o C não limpa esses espaços, por isso nunca se deve presumir que quando uma variável é declarada seu valor é zero ou qualquer outro valor.

Desse modo, até que nova variável receba um primeiro valor, ela terá um valor indefinido, que não pode ser utilizado para nada.

Atribuir um valor inicial a uma variável é chamado de inicialização da variável. Esse comando é feito através de uma expressão que contêm: o nome da variável, o [operador de atribuição (=)](#opAtrib) e o valor inicial da variável.

Sintaxe básica de atribuição de valor a uma variável:

<nome\_da\_variável> = <valor>;

Os compiladores da linguagem C permitem que inicializações de variáveis sejam feitas em dois momentos:

* 1. Quando a variável é declaração, seguindo a seguinte sintaxe:

<tipo\_de\_dados> <nome\_da\_variável> = <valor>;

Exemplo:

int a=2;

* 1. Em qualquer lugar do programa, após a variável ter sido declarada

Exemplo:

int a;

a=2;

* + 1. Classe de armazenamento
  1. Operadores

Operador é um símbolo que representa uma certa operação que pode ser executada em um certo dado. O valor do dado, que pode ser uma variável ou uma constante é chamado de operando.

Realizam funções aritméticas, de atribuição, relacionais e lógicas.

Expressões combinam as variáveis com os operadores e em uma expressão podemos ter mais de um operador, neste último caso é necessária a avaliação da expressão passo a passo, ou seja, um operador por vez, de acordo com as regras de precedência

* + 1. Operadores aritméticos

Os operadores aritméticos são aqueles que operam sobre números e/ou expressões numéricas, o resultado consiste em um valor numérico.

Desta forma, apenas operadores aritméticos e variáveis numéricas (inteiro e real) podem ser utilizadas em expressão desse tipo.

Os operadores aritméticos são:

|  |  |
| --- | --- |
| **OPERAÇÃO** | **OPERADOR** |
| Soma | + |
| Subtração | - |
| Multiplicação | \* |
| Divisão | / |
| Resto | % |
| Incrementa | ++ |
| Decrementa | -- |

O operador resto é usado quando se deseja encontrar o resto da divisão entre duas variáveis A e B. A apresentação é dada por A % B. Supondo A=3 e B=2, temos que A%B=1.

Os operadores de incremento e de decremento são equivalentes as seguintes operações:

x = x + 1 ---> ++x ou x++

x = x - 1 ---> --x ou x--

Embora as expressões ++x e x++ pareçam equivalentes, elas agem de forma diferente quando utilizadas em expressões com o operador de atribuição simples "=".

Quando o operador incrementa está "antes" da variável, primeiro se incrementa a variável para depois atribuir o resultado.

x = 10;

y = ++x;

incrementa x e atribui a y -> x = 11 e y = 11

Porém, se o operador incrementa está "depois" da variável, primeiro se atribui o valor a variável para depois incrementar o conteúdo da variável atribuída.

x = 10;

y = x++;

atribui x a y e incrementa x -> x = 11 e y = 10

* + 1. Operador de Atribuição

O operador de atribuição simples na linguagem C é usado para armazenar um valor em uma variável. Ele é composto por dois operandos:

* 1. Operando à esquerda: A variável que receberá o valor.
  2. Operando à direita: O valor que será armazenado na variável.

A linguagem C trata o sinal de igual "=" como sendo um operador e não como sinal de igual "=" da matemática. Ele atribui um valor ou resultado de uma expressão, que está a sua direita, para a variável especificada a sua esquerda.

Sintaxe básica:

*<nome\_da\_variável> = <valor>;*

1. Operadores de atribuição composta
   * 1. Operadores relacionais

Os operadores relacionais referem-se à comparação entre dois valores de um tipo básico.

Quando se compara 2 valores utilizando-se os operadores relacionais, obtém-se os seguintes resultados: falso ou verdadeiro.

|  |  |
| --- | --- |
| **OPERAÇÃO** | **OPERADOR** |
| Igual | == |
| Diferente | != |
| Maior | > |
| Menor que | < |
| Maior ou igual a | >= |
| Menor ou igual a | <= |

* + 1. Operadores lógicos

Os operadores lógicos são aqueles cujo resultado consiste em um valor lógico verdadeiro ou falso.

Estes operadores são muito utilizados para combinar logicamente os resultados (0 ou 1) obtidos nas relações utilizando-se os operadores relacionais.

|  |  |  |
| --- | --- | --- |
| **OPERADOR** | **SÍMBOLO** | **EXPLICAÇÃO** |
| Conjunção | && | Conjunção entre duas variáveis resulta em um valor verdadeiro somente quando as duas variáveis são verdadeiras. |
| Disjunção | || | A disjunção entre duas variáveis resulta em um valor verdadeiro quando pelo menos uma das variáveis é verdadeira. |
| Negação | ! | A negação inverte o valor de uma variável. Se a variável A é verdadeira, a negação de A, torna o valor da variável falso. |

* + 1. Tabela de preferência

|  |  |
| --- | --- |
| **MAIOR PRIORIDADE** | |
|  | ( ) operador parênteses: determina a prioridade das operações  [ ] operador colchetes: indexação de vetores |
| ! operador logico NOT  ++ incremento de 1  -- decremento de 1  - menos unário (transforma um n. positivo em negativo e vice-versa)  (tipo ou casts) expressão --> converte um tipo dado em outro |
| \* multiplicação  / divisão  % devolve o resto da divisão inteiro (modulo) |
| + adição  - subtração |
| < menor que  <= menor ou igual a  > maior que  >= maior ou igual a |
| == igual  != diferente |
| && operação logica AND |
| || operação logica OR |
| **MENOR PRIORIDADE** | |

* 1. Funções de Biblioteca
     1. Funções de entrada e saída de dados

Na linguagem C, as principais funções de entrada e saída encontram-se na biblioteca stdio.h e usam, em regra, o teclado e a tela como padrões para entrada e saída, respectivamente.

Antes de passar para as funções de entrada e saída, é importante explicar o que é um buffer e como ele funciona:

* + - 1. Saída

A saída de dados permite mostrar exibir resultados do processamento e mensagens aos usuários.

1. printf

Essa função permite escrever dados formatados na saída padrão, em forma de caracteres. Essa função encontra-se na biblioteca *stdio.h*.

Sintaxe:

*printf("<serie de controle>", <lista de argumentos>);*

*<serie de controle>*:

Contém uma descrição de tudo que a função vai colocar na tela, ou seja, a mensagem que pretende exibir.

Além dos caracteres que devem ser exibidos, também mostra o conteúdo de uma variável na sua respectiva posição

Para determinar a posição em que o valor da variável deve ser exibido basta colocar o código de formatação, que indica como o conteúdo de uma variável deve ser exibido, na posição desejada.

Os códigos de formação são:

|  |  |
| --- | --- |
| **Código** | **Descrição** |
| %c | exibe um único caractere |
| %d | exibe um inteiro |
| %f | exibe um número real em ponto flutuante |
| %e | exibe um número real em ponto flutuante na notação cientifica |
| %g | utiliza o mais curto de %e ou %f, |
| %x | exibe um número em notação hexadecimal |
| %o | exibe um número em notação octal |
| %hd | exibe um número inteiro curto |
| %ld | exibe um numero inteiro longo |
| %c | exibe um caractere Sem sinal |
| %hu | exibe um número inteiro curto sem sinal |
| %u | exibe um número inteiro sem sinal |
| %lu | exibe um número inteiro longo sem sinal |

*Obs.: no código de formatação de float (%f), em regra aparecem 6 casas depois da virgula, mas é possível decidir o número de casas que irá aparecer depois da virgula, basta colocar um ponto e o número desejado antes do f, exemplo: %.2f, irá aparecer apenas duas casas.*

Códigos especiais para a exibição de alguns caracteres especiais e realização de ações que permitam formatar a saída do programa.

|  |  |
| --- | --- |
| **Código** | **Descrição** |
| \n | nova linha |
| \t | tabulação horizontal |
| \b | retrocesso (backspace) |
| \f | mudança de página (form feed) |
| \r | início de linha (carriage return) |
| \\ | exibe a barra a invertida |
| \' | exibe o apostrofe |
| %% | exibe o caractere % |

*<lista de argumentos>*:

Informa o valor que vai substituir a posição do código de formatação

Pode conter identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes

Não é obrigatória, já que o *printf* pode conter apenas uma cadeia de caracteres

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char c = 'A';

int n = 41;

float m = 5.5;

printf("exemplos dos codigos de formatacao e variaveis simples\n");

//print sem código de formatação ou lista de argumentos

printf("Char: %c\n\n", c);

printf("int n: %d float m: %f\n\n", n, m);

printf("n + m= %.2f\n\n", m+n);

return 0;

}

**Tela de saída**

Texto

Descrição gerada automaticamente

1. putchar

Imprime apenas um caractere na saída padrão.

Sintaxe:

*putchar(<nome\_da\_variável);*

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char c=’G’;

printf("o caracter c e': ");

putchar(c);

return 0;

}

**Tela**



* + - 1. Entrada

A entrada de dados permite receber os dados digitados pelo usuário, esses dados recebidos são armazenados em variáveis.

Na linguagem C existem diversas funções para entrada de dados, algumas delas são scanf e getchar.

1. scanf

A função scanf é a mais utilizada a leitura de dados formatados da entrada padrão, normalmente, um valor digitado pelo usuário através do teclado.

Sintaxe:

*scanf("<serie de controle>", <lista de argumentos>);*

*<serie de controle>*

Contém apenas o código de formatação que especifica o tipo de informação definhada na lista de argumentos (tipo da variável).

|  |  |
| --- | --- |
| **Código** | **Descrição** |
| %c | exibe um único caractere |
| %d | exibe um inteiro |
| %f | exibe um número real em ponto flutuante |
| %e | exibe um número real em ponto flutuante na notação cientifica |
| %g | utiliza o mais curto de %e ou %f, |
| %x | exibe um número em notação hexadecimal |
| %o | exibe um número em notação octal |
| %hd | exibe um número inteiro curto |
| %ld | exibe um numero inteiro longo |
| %uc | exibe um caractere sem sinal |
| %hu | exibe um número inteiro curto sem sinal |
| %ud | exibe um número inteiro sem sinal |
| %lu | exibe um número inteiro longo sem sinal |

*<lista de argumentos>*

Contém o endereço da variável, mostra onde o dado deve ser armazenado na memória que o computador alocou para a variável declarada. Por isso o operador de endereço (&) é utilizado esta função, fazendo com que o valor digitado pelo usuário seja colocado no endereço correto de memória onde a variável foi criada pelo computador.

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char char1,char2;

float f;

printf("digite o char1: ");

char1=getchar();

printf("\n\ndigite o char2: ");

scanf("%c",&char2);

printf("\n\nchar1: %c\nchar2: %c",char1,char2);

return 0;

}

**Tela**

Texto

Descrição gerada automaticamente

1. getchar

Lê apenas um caractere da entrada padrão + o ENTER.

Sintaxe:

*<identificador\_variavel> = getchar();*

A função irá retornar apenas o primeiro caractere do buffer de entrada (ou buffer do teclado)

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char c;

printf("digite um caracter : ");

c = getchar();

printf("\n\ncaracter digitado: ");

putchar(c);

printf("\nFIM DO PROGRAMA");

return 0;

}

**Tela**

Texto

Descrição gerada automaticamente

* + - 1. Buffer

*Buffer* é de uma área na memória principal que armazena temporariamente dados enquanto aguardam ser processados (lidos ou escritos) ou transferidos entre dois dispositivos ou entre dispositivos e uma aplicação, atuando como um intermediário.

Assim, enquanto o processador manipula o dado localizado no buffer, o dispositivo realiza outra operação de leitura no mesmo instante.[[3]](#footnote-3)

Em geral, um buffer é associado a transferência de dados entre um dispositivo de entrada e a memória principal ou o processador.

Diagrama

Descrição gerada automaticamente

Para a programação, é necessário entender como as funções de entrada utilizam buffer e que o modo como manipulam o buffer pode trazer alguns problemas caso seja feita a leitura de mais de uma entrada, vejamos:

1. Todos os dados que estão sendo digitados são armazenados no buffer, incluído o “ENTER”.
2. A execução das funções de entrada armazena os dados que estão no buffer no local reservado da variável pretendida.

Ocorre que normalmente é deixado um resquício no buffer, ou seja, extraem o conteúdo e deixam o ENTER no buffer, ou no caso de inserir um dado maior que o tamanho da variável reservada.

Assim, se não estiver vazio, o conteúdo do buffer será armazenado por uma próxima ativação de uma destas funções

Isso ocorre muito quando as funções de entrada são utilizadas para o recebimento de uma variável do tipo char.

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char char1,char2;

float f;

printf("digite o char1: ");

char1=getchar();

printf("\n\ndigite o char2: ");

scanf("%c",&char2);

printf("\n\nchar1: %c\nchar2: %c",char1,char2);

return 0;

}

**Tela**

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

O que aconteceu:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Linha do programa** | **Dado entrada** | **Buffer** | **char1** | **char2** |
| Inicio | - | - | - | - |
| printf("digite o char1: "); | - | - | - | - |
| char1=getchar(); | a | |  |  |  | | --- | --- | --- | | a | “enter” |  | | - | - |
| *\*transferência dado buffer-variável* | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | - |
| printf("\n\ndigite o char2: "); | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | - |
| scanf("%c",&char2); | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | - |
| *\*transferência dado buffer-variável* | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | “enter” |

Portanto é prudente limpar o buffer antes de uma chamada de uma segunda chamada, para isso pode-se usar a função *fflush(<dispositivo>)*, que "descarrega" todo o buffer dos dados associado ao dispositivo de saída especificado.

No caso de funções de entrada, bastar colocar o dispositivo de entrada padrão (teclado):

*fflush(stdin);*

O ideal é chamar essa função antes da segunda entrada de dados, para garantir que o buffer vai estar vazio:

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char char1,char2;

printf("digite o char1: ");

char1=getchar();

printf("\n\ndigite o char2: ");

fflush(stdin);

scanf("%c",&char2);

printf("\n\nchar1: %c\nchar2: %c",char1,char2);

return 0;

}

**Tela**

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente com confiança média

O que aconteceu:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Linha do programa** | **Dado entrada** | **Buffer** | **char1** | **char2** |
| Inicio | - | - | - | - |
| printf("digite o char1: "); | - | - | - | - |
| char1=getchar(); | a | |  |  |  | | --- | --- | --- | | a | “enter” |  | | - | - |
| *\*transferência dado buffer-variável* | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | - |
| printf("\n\ndigite o char2: "); | - | |  |  |  | | --- | --- | --- | | “enter” |  |  | | a | - |
| fflush(stdin) | - | |  |  |  | | --- | --- | --- | |  |  |  | | a | - |
| scanf("%c",&char2); | b | |  |  |  | | --- | --- | --- | | b |  |  | | a | - |
| *\*transferência dado buffer-variável* | - | |  |  |  | | --- | --- | --- | | b |  |  | | a | b |

* 1. Comandos

Como anteriormente esclarecido, comando é uma estrutura de programação, representa uma instrução ou ação, servem para controlar o fluxo do programa ou a repetição de blocos.

Os principais comandos são os chamados de comandos de seleção e os comandos de repetição (laços).

* + 1. Comandos de seleção

Os comandos de seleção são aqueles que permitem ao programador alterar a sequência de execução do programa, controlando o fluxo, avaliando uma condição e executando um bloco de declarações se a condição testada for verdadeira ou executando um outro bloco se for falsa.

Os comandos são: if e switch.

* + - 1. If

O comando *if* representa uma tomada de decisão do tipo "SE isto ENTAO aquilo".

Avalia-se uma expressão para tomada de decisão e se a expressão for verdadeira executa a declaração, se falsa não executa a declaração.

O corpo do *if* pode ter uma única declaração ou várias declarações entre um abre e fecha chave “{ }”.

O formato geral quando se tem uma única declaração é:

if (expressão)

declaração;

O formato geral quando se tem mais de uma declaração tem que se usar o abre e fecha chave "{ }" para agrupar as declarações é:

if (expressão)

{

declaração;

declaração;

}

Exemplo:

**Código**

#include<stdio.h>

int main()

{

char ch;

printf("digite uma letra: ");

ch=getchar();

if(ch == 'r')

printf("\nVoce digitou a tecla r");

printf("\n\nfim");

return 0;

}

**Tela**

falso

Texto

Descrição gerada automaticamente

verdadeiro

Texto

Descrição gerada automaticamente

1. if-else

O comando *if-else* é uma expansão do comando *if*.

No comando *if* só se executa a declaração ou declarações se a expressão for verdadeira. Acrescentando a cláusula *else* permite executar outra declaração ou declarações se a expressão for falsa.

Resumindo, se a expressão for verdadeira executa-se as declarações antes do *else*, se a expressão for falsa executa-se as declarações depois do *else*.

O formato geral quando se tem uma única declaração é:

if (expressão)

declaracao1;

else

declaracao2;

O formato geral quando se tem mais de uma declaração é:

if (expressão)

{

declaracao1;

declaracao1;

}

else

{

declaracao2;

declaracao2;

}

Exemplo

**Código**

#include<stdio.h>

int main()

{

char ch;

printf("digite uma letra: ");

ch=getchar();

if(ch == 'r')

{

printf("\nVoce digitou a tecla r");

printf("\nparabens voce acertou");

}

else

{

printf("\nVoce nao digitou a tecla r");

printf("\nque azar voce errou");

}

printf("\n\nfim");

return 0;

}

**Tela**

falsa

Texto

Descrição gerada automaticamente com confiança média

verdadeira

Texto

Descrição gerada automaticamente

Uma construção comum em programação é o encadeamento ou aninhamento de *if-else*. As expressões condicionais serão avaliadas de cima para baixo.

O formato pode ser:

if (expressão 1)

{

if(expressão 2)

{

declaracao1;

declaracao1;

}

else

declaracao2;

}

else

{

if(expressão 3)

{

declaracao3;

declaracao3;

}

}

Exemplo:

**Código**

#include<stdio.h>

int main()

{

int opcao;

int valor;

printf ("Converte um numero de: \n");

printf ("1: decimal para hexadecimal\n");

printf ("2: hexadecimal para decimal\n");

printf ("\nInforme sua opcao: ");

scanf ("%d", &opcao);

if (opcao == 1)

{

printf ("\nInforme o valor em decimal: ");

scanf ("%d", &valor);

printf ("%d em hexadecimal e': %x", valor, valor);

}

else

}

if (opcao == 2) /\* if aninhado \*/

{

printf ("\nInforme o valor em hexadecimal: ");

scanf ("%x", &valor);

printf ("%x em decimal e': %d", valor, valor);

}

else

printf ("\nA opcao escolhida e' invalida.");

}

printf("\n\nfim");

return 0;

}

**Tela**

opção 1

Texto

Descrição gerada automaticamente

opção 2

Texto

Descrição gerada automaticamente

* + - 1. switch

No comando *switch*, a variável é sucessivamente testada contra uma lista de constantes. Quando uma associação é encontrada, o conjunto de declarações associada à constante é executado.

Há três coisas importantes a saber sobre o comando *switch*:

* 1. O comando *switch* difere do comando *if* porque *switch* só pode testar igualdade, enquanto *if* pode avaliar uma expressão logica e/ou relacional;
  2. Duas constantes *case* no mesmo *switch* não podem ter valores iguais. Porém, um comando *switch* incluído em outro comando *switch* mais externo pode ter as mesmas constantes no *case*.
  3. Se constantes do tipo caractere são usadas no comando *switch*, elas são automaticamente convertidas para seus valores inteiros.

O formato geral é:

switch (variável)

{

case constante\_1:

declaração\_1;

break;

case constante\_2:

declaração\_2;

break;

.

.

.

case constante\_n:

declaração\_n;

break;

default:

declaração\_default;

}

A declaração *default* é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.

Na estrutura do comando *switch* usa-se o comando *break*, que é um comando de desvio, onde ele realiza um desvio incondicional. O *case* será executado até que o comando *break* ou o fim do *switch* seja alcançado.

Quando o comando *break* é alcançado o programa salta para a declaração seguinte ao final do *switch* (o fecha chave "}"). Caso não houvesse inserido o *break* o programa continuaria testando os demais “cases” e poderia causar um erro de logica do programa.

Exemplo:

**Código**

#include<stdio.h>

int main()

{

int opcao;

int valor;

printf ("Converte um numero de: \n");

printf ("1: decimal para hexadecimal\n");

printf ("2: hexadecimal para decimal\n");

printf ("\nInforme sua opcao: ");

scanf ("%d", &opcao);

switch(opcao)

{

case 1:

{

printf ("\nInforme o valor em decimal: ");

scanf ("%d", &valor);

printf ("%d em hexadecimal e': %x", valor, valor);

break;

}

case 2:

{

printf ("\nInforme o valor em hexadecimal: ");

scanf ("%x", &valor);

printf ("%x em decimal e': %d", valor, valor);

break;

}

default:

printf ("\nA opcao escolhida e' invalida.");

}

printf("\n\nfim");

return 0;

}

**Tela**

opção 1

Texto

Descrição gerada automaticamente

opção 2

Texto

Descrição gerada automaticamente

* + 1. Comandos de repetição

Em muitos casos, os programas precisam repetir uma ou mais declarações até que o laço atenda a uma condição específica que não envolva necessariamente uma contagem.

* + - 1. while

Este é um dos comandos mais utilizados.

Usa-se este comando quando queremos que uma tarefa seja executada enquanto a condição logica for verdadeira. Quando esta é falsa o programa pula para a declaração seguinte ao laço.

Neste comando, o teste da condição logica é executado antes de entrar no laço. Portanto se a condição for falsa no primeiro teste a declaração ou declarações que estão dentro do laço não serão executadas nenhuma vez.

O formato geral quando se tem uma única declaração é:

while (condição)

declaração;

O formato geral quando se tem mais de uma declaração é:

while (condição)

{

declaração;

declaração;

}

Exemplo

**Código**

#include<stdio.h>

int main()

{

int x,y=1,fat=1;

printf("digite um numero para calcular a fatorial: ");

scanf("%d",&x);

while (y < x)

{

y = y + 1;

fat = fat \* y;

}

printf("O valor da fatorial e' = %.d\n",fat);

return 0;

}

**Tela**

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

[**Teste de mesa**](#testemesa)

|  |  |  |  |
| --- | --- | --- | --- |
| **Linha** | **y** | **fat** | **x** |
| INICIO | 1 | 1 | - |
| Scanf | 1 | 1 | 3 |
| while r1 - l1 | 1+1 | 1 | 3 |
| while r1– l2 | 2 | 1\*2 | 3 |
| while r2– l1 | 2+1 | 2 | 3 |
| while r2– l2 | 3 | 2\*3 | 3 |
| Saí do while | 3 | 6 | 3 |
| FIM | 3 | 6 | 3 |

* + - 1. do-while

Este comando é semelhante ao anterior, a diferença está na avaliação da condição logica que acontece após a execução da declaração ou declarações pelo menos uma vez, pois a condição está no final do loop

Em outras palavras, a declaração ou declarações são executadas pelo menos uma vez e depois é avaliada a condição logica. Se a condição for verdadeira a declaração ou declarações são executadas novamente. Se a condição logica for falsa a execução do programa continua na declaração seguinte ao loop.

Para este comando é necessário sempre se utilizar o abre e fecha chave "{ }" entre o *do* e o *while* e o ponto e vírgula “;” após o while

O formato geral que é igual tanto para uma como para mais declarações é:

do

{

declaração;

declaração;

}while (condição);

Exemplo

**Código**

#include<stdio.h>

int main()

{

int x,y,fat;

char tecla;

do

{

printf("\ndigite um numero para calcular a fatorial: ");

scanf("%d",&x);

y=1;

fat=1;

while (y < x)

{

y = y + 1;

fat = fat \* y;

}

printf("O valor da fatorial e' = %d\n",fat);

printf("\ndeseja continuar?(s/n) ");

fflush(stdin);

tecla = getchar();

}while ((tecla != 'n') && (tecla != 'N'));

printf("\n\nfim");

return 0;

}

**Tela**

Texto

Descrição gerada automaticamente

[**Teste de mesa**](#testemesa)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Linha** | **y** | **fat** | **x** | **Tecla** |
| INICIO | - | - | - | - |
| do r1 - scanf | - | - | 3 | - |
| do r1 – y=1 | 1 | - | 3 |  |
| do r1 – fat=1 | 1 | 1 | 3 |  |
| do r1 - while r1 - l1 | 1+1 | 1 | 3 | - |
| do r1 - while r1– l2 | 2 | 1\*2 | 3 | - |
| do r1 - while r2– l1 | 2+1 | 2 | 3 | - |
| do r1 - while r2– l2 | 3 | 2\*3 | 3 | - |
| do r1 – sai do while | 3 | 6 | 3 | - |
| do r1 – getchar | 3 | 6 | 3 | S |
| do r2 - scanf | 3 | 6 | 4 | S |
| do r2 – y=1 | 1 | 6 | 4 | S |
| do r2 – fat=1 | 1 | 1 | 4 | S |
| do r2 - while r1 - l1 | 1+1 | 1 | 4 | S |
| do r2 - while r1– l2 | 2 | 1\*2 | 4 | S |
| do r2 - while r2– l1 | 2+1 | 2 | 4 | S |
| do r2 - while r2– l2 | 3 | 2\*3 | 4 | S |
| do r2 - while r3 – l1 | 3+1 | 6 | 4 | S |
| do r2 - while r3 – l2 | 4 | 6\*4 | 4 | S |
| do r2 – sai do while | 4 | 24 | 4 | S |
| do r1 – getchar | 4 | 24 | 4 | N |
| Sai do do-while | 4 | 24 | 4 | N |
| FIM | 4 | 24 | 4 | N |

* + - 1. for

O comando *for* repete uma declaração ou declarações, controlado por uma variável e executado por um número finito de vezes.

A sua sintaxe é declarada toda na mesma linha, o que facilita sua compreensão.

O formato geral quando se tem uma única declaração é:

for (expressão 1 ; expressão 2 ; expressão 3)

declaração;

O formato geral quando se tem mais de uma declaração é:

for (expressão 1 ; expressão 2 ; expressão 3)

{

declaração;

declaração;

}

Na *expressão 1* inicializa-se a variável de controle o loop.

Na *expressão 2* testa-se a variável de controle, que pode ser um teste lógico ou um contador. Enquanto o teste da variável de controle for verdade o loop continua a ser executado. Quando o teste se torna falso, o controle do programa passa para a declaração seguinte ao loop após o fecha chave "}" quando este existir.

Na *expressão 3* controla-se o incremento ou decremento da variável de controle do loop.

O funcionamento do comando *for* começa pela *expressão 1* onde a variável de controle do loop é inicializada. A *expressão 1* é executada uma única vez no início do loop. Então executa-se as declarações. Em seguida a *expressão 3*, incrementa ou decrementa da variável de controle. Após a execução da *expressão 3 é* testada a variável de controle na *expressão 2*.

Exemplo:

**Código**

#include<stdio.h>

int main()

{

int i;

for (i = 0; i <= 5; i++)

{

printf ("Este e' o valor da variavel de controle i: %d ", i);

printf (" e i ao quadrado e': %d\n", i \* i);

}

printf("\n\nfim");

return 0;

}

**Tela**

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

[**Teste de mesa**](#testemesa)

|  |  |  |  |
| --- | --- | --- | --- |
| **Linha** | **i** | **i \* i** | **i++** |
| INICIO | - | - | - |
| for r1 – exp2 | 0 | - | - |
| for r1 – l1 | 0 | 0\*0 | - |
| for r1 – exp3 | 0 | - | 1+1 |
| for r2 – exp2 | 1 | - | - |
| for r2 – l1 | 1 | 1\*1 | - |
| for r2 – exp3 | 1 | - | 1+1 |
| for r3 – exp2 | 2 | - | - |
| for r3 – l1 | 2 | 2\*2 | - |
| for r3 – exp3 | 2 | - | 2+1 |
| for r4 – exp2 | 3 | - | - |
| for r4 – l1 | 3 | 3\*3 | - |
| for r4 – exp3 | 3 | - | 3+1 |
| for r5 – exp2 | 4 | - | - |
| for r5 – l1 | 4 | 4\*4 | - |
| for r5 – exp3 | 4 | - | 4-1 |
| for r6 – exp2 | 5 | - | - |
| for r6 – l1 | 5 | 5\*5 | - |
| for r6 – exp3 | 5 | - | 5-1 |
| Sai do for | 6 | - | - |
| FIM | 6 | - | - |

* + 1. Teste de mesa

O Teste de Mesa é um processo manual que é utilizado para validar a lógica de um determinado algoritmo.

Esse teste consiste em acompanhar os valores das variáveis do programa e verificar se os resultados são os esperados.

É uma simulação de um programa, de forma manual, geralmente feito com papel e caneta.

Exemplo:

#include<stdio.h>

int main()

{

int num=0,cont,k;

printf("Digite a quantidade de termos desejada\n");

scanf("%i",&k);

for(cont=1;cont<=k;cont++)

{

num=num+4;

printf("1/%d \n",num);

}

return 0;

}

**Teste de mesa**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Linha** | **cont** | **k** | **num** | **cont++** |
| INICIO | - | - | 0 | - |
| Scanf | - | 3 | 0 | - |
| for r1 – exp2 | 1 | 3 | 0 | - |
| for r1 – l1 | 1 | 3 | 0+4 |  |
| for r1 – exp3 | 1 | 3 | 4 | 1+1 |
| for r2 – exp2 | 2 | 3 | 4 | - |
| for r2 – l1 | 2 | 3 | 4+4 | - |
| for r2 – exp3 | 2 | 3 | 8 | 2+1 |
| for r3 – exp2 | 3 | 3 | 8 | - |
| for r3 – l1 | 3 | 3 | 8+4 | - |
| for r3 – exp3 | 3 | 3 | 12 | 3+1 |
| Sai do for | 4 | 3 | 12 | - |
| FIM | 4 | 3 | 12 | - |

Nota-se que na linha “for – 4” o cont não é mais menos ou igual ao k, por isso que as declarações do for não são mais executadas e o num fica com o valor igual ao da última rodada

* 1. Vetores

Vetor é uma variável composta e homogênea que agrupa dados do mesmo tipo, por isso pode ser dizer que os vetores são uma estrutura de dados indexada, que pode armazenar uma quantidade, determinada pelo programador, de dados do mesmo tipo. Em C, são identificados pela existência de colchetes após o nome da variável no momento da declaração.

Cada dado armazenado em um vetor pode ser chamado de item ou elemento, sendo que o tamanho de um vetor é definido pela quantidade/número de elementos que serão armazenados.

Quando um vetor é declarado, os espaços para armazenar cada um dos itens são reservados na memória sequencialmente, de acordo com o tipo de dados que serão armazenados e com o tamanho do vetor.

Os vetores podem ser unidimensionais ou multidimensionais, estes também são chamados de matrizes. As únicas diferenças estão na quantidade de dimensões (fileiras) envolvidas e, consequentemente, na declaração e manipulação dos itens.

* + 1. Vetores unidimensionais

Um vetor unidimensional possui apenas uma dimensão, ou seja, possui apenas uma fileira/linha de dados.

Como na linguagem C a numeração começa sempre em zero, um vetor é indexado a partir do 0 até um número antes do tamanho do vetor. Por exemplo, um vetor unidimensional de dados do tipo int de tamanho 5, os dados serão indexados da seguinte forma:



Quando é declarado um vetor unidimensional, reserva os espaços para armazenar cada um dos itens, utilizando o exemplo anterior será reservado cinco espaços de 4 bytes ([tipo de dado int](#Int)), ou seja, serão reservados sequencialmente 5x4=20 bytes para o vetor:

Gráfico

Descrição gerada automaticamente

* + - 1. Declarando um vetor unidimensional

Para declarar um vetor, primeiro é preciso informar o [tipo dos dados](#TipoDado) que serão armazenados, o identificador da variável e, por fim, o tamanho/número de elemento do vetor, para isso utiliza-se o operador "[]". Sintaxe:

<[tipo\_de\_dados](#TipoDado)> <[identificador](#identificadores)>[<[tamanho\_do\_vetor](#tamVet)>];

Exemplo:

int A[3];

Gráfico

Descrição gerada automaticamente com confiança média

* + - 1. Referenciando um elemento do vetor unidimensional

Quando se deseja acessar ou alterar o conteúdo de um dado armazenado, através de comandos, funções, atribuições ou expressões, é preciso referenciar a variável atribuída.

Cada um dos elementos de um vetor é referenciado individualmente por meio do índice a ele associado entre colchetes "[]" após o nome do vetor (identificador).

Importante relembrar que o índice especifica a posição do elemento no vetor e que este é indexado sequencialmente, portanto, os índices começam a partir do 0 até tamanho\_do\_vetor -1.

Tabela

Descrição gerada automaticamente com confiança média

Quando tem um número entre colchetes pode ter dois significados diferentes:

* 1. Tamanho do vetor/número de elementos, quando o vetor é declarado

Exemplo:

int A[3];

Tabela

Descrição gerada automaticamente

* 1. Índice do elemento, para acessar/referenciar um elemento no vetor, apenas após a devida declaração do vetor.

Exemplo:

int A[3];

|  |  |
| --- | --- |
| scanf(“%d”,&A[2]); | //A[2] esta referenciando a posição 2 do vetor A para ser armazenador A |
|  |  |
| printf(“%d”,A[0]); | //A[0] esta referenciando elemento armazenado na posição 0 do vetor A |
|  |  |

O índice de um elemento pode ser indicado diretamente, por meio de um número inteiro, ou por meio de uma expressão que resulte em um valor inteiro.

Exemplo:

int A[3],i=0;

|  |  |
| --- | --- |
| scanf(“%d”,&A[i+2]); | //a expressão i+2 é igual a 0+2 =2, então A[i+2] é igual A[2] e esta referenciando elemento armazenado na posição 2 do vetor A |
|  | Tabela  Descrição gerada automaticamente |
| printf(“%d”,A[i]); | // i é igual a 0, então A[i] é igual A[0] e esta referenciando elemento armazenado na posição 0 do vetor A |
|  | Tabela  Descrição gerada automaticamente |

Na linguagem C NÃO há verificação de limites nos vetores, ou seja, é possível ultrapassar o fim de um vetor, escrevendo os dados em espaços não reservados para o vetor, podendo sobrescrever em outras variáveis ou mesmo em trechos de código de programa e correndo o risco de o computador travar, surgindo bugs.

Exemplo:

**Código**

#include<stdio.h>

int main()

{

int A[3],i;

for(i=0;i<5;i++) //deve rodar 5 vezes e guardar 5 itens até o índice 4 do vetor A

{

printf(“digite um número inteiro:”);

scanf(“%d”,&A[i]);

}

for(i=0;i<5;i++) //deve rodar 5 vezes e imprimir os 5 itens até o índice 4 do vetor A

printf(“\nA[%d]: %d”,i,A[i]);

return 0;

}

**Tela**

**Texto

Descrição gerada automaticamente**

Obs.: olhar os \* após o teste de mesa

**Teste de mesa**

|  |  |  |
| --- | --- | --- |
| **Linha** | **i** | **A[i]** |
| INICIO | - | - |
| for r1 – exp1 | 0 | - |
| for r1 – exp2 | 0 | - |
| for r1 – l1 – scanf – A[0] | 0 | 1 |
| for r1 – exp3 | 0+1 | 1 |
| for r2 – exp2 | 1 | - |
| for r2 – l1 – scanf – A[1] | 1 | 2 |
| for r2 – exp3 | 1+1 | 2 |
| for r3 – exp2 | 2 | - |
| for r3 – l1 – scanf – A[2] | 2 | 3 |
| for r3 – exp3 | 2+1 | 3 |
| for r4 – exp2 | 3 | - |
| for r4 – l1 – scanf – A[3]\*\* | 3 | 4 |
| for r4 – exp3 | 3+1 | 4 |
| for r5 – deu bug\* | 4 | - |
| Sai do for | 4 | - |
| for r1 – exp1 | 0 | - |
| for r1 – exp2 | 1 | - |
| for r1 – l1 – printf – A[0] | 1 | 1 |
| for r1 – exp3 | 0+1 | - |
| for r2 – exp2 | 1 | - |
| for r2 – l1 – printf – A[1] | 1 | 2 |
| for r2 – exp3 | 1+1 | - |
| for r3 – exp2 | 2 | - |
| for r3 – l1 – printf – A[2] | 2 | 3 |
| for r3 – exp3 | 2+1 | 3 |
| for r4 – exp2 | 3 | - |
| for r4 – l1 – printf – A[3] | 3 | 3\*\*\* |
| for r4 – exp3 | 3+1 | 3\*\*\* |
| for r5 – exp2 | 4 | - |
| for r5 – l1 – printf– A[0] | 4 | 7869408\*\*\* |
| for r5 – exp3 | 4+1 | 7869408\*\*\* |
| Sai do for | 5 | - |
| FIM |  |  |

\*é possível ver que o primeiro for rodou apenas 4 vezes, ou seja, deu algum bug que impediu que o for rodasse mais uma vez como deveria

\*\*mesmo assim, o primeiro for permitiu a entrada de dados no índice 4, indo além do limite do tamanho do vetor

Uma imagem contendo Gráfico

Descrição gerada automaticamente

\*\*\*já no segundo for, rodou as 5 vezes previstas e imprimiu até o índice 4, indo além do limite do tamanho, mas quando ele imprimiu o índice 3 não foi o valor inserido no primeiro for e no índice 4 imprimiu lixo de memória

Gráfico

Descrição gerada automaticamente

Por isso, quando estiver referenciando um vetor, é tarefa do programador tomar cuidado para que os índices dos vetores estejam sempre dentro dos limites estabelecidos pela declaração do tamanho do vetor.

* + - 1. Inicializando um vetor

Assim como acontece com as variáveis simples, também possível inicializar os elementos de um vetor junto com a sua declaração. Neste caso, após o operador de atribuição, os elementos devem ser inicializados sequencialmente entre chaves ‘{ }’ e separados por virgulas, da seguinte forma:

<tipo\_dados> <identificador>[<tamanho\_do\_vetor]={<elemento1>,<elemento2 >,...}

Lembrando que os elementos devem ser colocados na ordem em que serão colocados no vetor, para inicializar os elementos um vetor há as seguintes formas:

1. Declarando o tamanho do vetor explicitamente e iniciando todos os elementos

exemplo:

int A [4] = {1,2,3,4};

1. Declarando o tamanho do vetor explicitamente e inicializando apenas alguns elementos. Neste caso, quando o número de itens inicializados é menor que o número total de itens do vetor, os itens não inicializados são automaticamente zerados.

Exemplo:

|  |  |
| --- | --- |
| int A[4]={1,2} | equivalente a: int A[4]={1,2,0,0} |

1. Não declarando o tamanho do vetor e inicializando todos os elementos. Neste caso, os colchetes são declarados vazios e serão preenchidos pelo pré-compilador que contará o número de elementos da inicialização.

Exemplo:

|  |  |
| --- | --- |
| int A[ ] = {1,2,3,4}; | equivalente a: int A[4]={1,2,3,4} |

Um vetor somente pode ser inicializado dessas formas no momento da declaração e se for uma variável global ou static local.

Porém, se um vetor for declarado como variável local ou não for inicializado quando foi declarado, então ele pode ser inicializado elemento por elemento.

Exemplo:

int A[4];

A[0]=1;

A[1]=2;

A[2]=3;

A[3]=4;

* + 1. Vetores de char (strings)

Importante lembrar que um vetor do tipo char armazena mais de um caractere, que ocupa um byte na memória, um ao lado do outro em sequência, constituindo uma cadeia de caracteres, portanto, cada elemento do vetor é um caractere.

Ressalta-se que em C não há um tipo de dados chamado string, mas sim um vetor tipo char que possui um terminador, se não o tiver ao fim do vetor é apenas um vetor de char.

Em C, o compilador reconhece uma string quando a cadeia de caracteres aparece entre aspas, enquanto um único caractere aparece entre apóstrofe, por exemplo:

“C” //string

‘C’ //caractere

Para um vetor de char ser considerado uma string deve conter um delimitador que indica o final da string, que é um caractere especial que representa o 0 do código ASCII, também conhecido como caractere nulo ou terminador, que é escrito em C da seguinte forma: \0.

Exemplo:

“C” //string



‘C’ //caractere



Os compiladores C contêm recursos especiais para tratamento de uma string como na inicialização, que pode ser feita de forma diferente do que um vetor comum, e possui um código de formatação especifíco (%s) para as funções de entrada e saída.

* + - 1. Declarando um vetor de char (string)

A declaração de um vetor de char é praticamente igual à declaração de um vetor unidimensional, entretanto para armazenar uma string é necessário lembra que que, ao final da cadeia de caracteres, o último elemento deve ser sempre o '\0' (terminador nulo).

Portanto, o tamanho de uma string, também conhecido como cumprimento da string, deve ser, no mínimo, número de caracteres desejados mais o terminador nulo ‘\0’. Por exemplo, o programador quer declarar uma cadeia com 7 caracteres, então deve declarar o vetor com tamanho 8:

char C[4];

Note que o tamanho deve ser sempre maior, mas é possível armazenar uma string menor que o tamanho declara, usando o exemplo anterior, é inserido no vetor C[8] a palavra “texto”, nesse caso apenas as 6 primeiras posições do vetor serão utilizadas e as duas últimas terão valores indeterminados, já que o C não inicializa variáveis

Uma imagem contendo relógio, desenho

Descrição gerada automaticamente

Entretanto, se entrar com uma string de comprimento maior que o declarado, o programa irá aceitar, mas os resultados podem ser desastrosos, como anteriormente explicando, podendo sobrescrever em outras variáveis ou mesmo em trechos de código de programa e correndo o risco de o computador travar, surgindo bugs.

* + - 1. Referenciando uma string

Uma string pode ser acessada de 2 formas, caractere por caractere através do "índice" de cada um dos elementos do vetor, ou como uma string acessando todos os caracteres até encontrar o finalizador.

Como as strings são vetores de caracteres, para se acessar um determinado caractere de uma string, basta usar um índice para acessar o caráter desejado, como todo vetor é indexado a partir do 0, portanto, a primeira letra fica no índice 0, a segunda no índice 1 e assim por diante.

Lembrando que para acessar um único caractere, deve ser usado o código de formatação %c, para as funções de entrada e saída de dados.

Usando o exemplo anterior usado, no vetor char C[8], que contém a palavra “texto” e quer-se imprimir a letra ‘X’ (terceira letra), que para ser acessada basta usar o índice 2:

|  |  |
| --- | --- |
| printf(“%c”,C[2]); | //C[2] esta referenciando terceira letra armazenada no índice 2 do vetor C |
|  |  |

Já é possível referenciar uma string inteira, ou seja, todos os caracteres do vetor de char, basta usar o nome da variável sem nenhum índice [ ], somente é possível porque o sistema coloca no final da cadeia o finalizador '\0'.

Para isso, o C possui um código de formatação específico para as strings %s, quando é utilizado o vetor é percorrido até encontrar o finalizador \0, no caso de saída de dados, e preenche com a cadeia de caracteres e inclui automaticamente o finalizador:

|  |  |
| --- | --- |
| printf(“digite:\n”);  scanf(“%s”,&C); | //&C está referenciando o vetor C e que deve ser preenchido com a cadeia de caracteres digitada, no caso “TEXTO”, e que deve colocar o \0 ao final |
|  | Uma imagem contendo relógio, desenho  Descrição gerada automaticamente |
| printf(“C: %s”,C); | //C está referenciando o vetor C e que deve ser lido e impresso até encontrar o \0 |
|  | Uma imagem contendo relógio, desenho  Descrição gerada automaticamente |

* + - 1. Inicializando uma string em um vetor de char

Assim como nos vetores comuns, que só podem ser inicializados se for declarado como uma variável global ou static local, o que continua valendo para um vetor de char.

Há diversas formas para inicializar um vetor de char com uma string um:

1. Inicializando caractere por caractere como nos vetores comuns, lembrando que cada caractere deve estar entre apóstrofes e devem ser colocados na ordem em que serão colocados no vetor:

char <identificador>[<tamanho\_do\_vetor]={‘<elemento1>’,’<elemento2 >’,...};

Neste caso, para que seja tratado como uma string o finalizador deve ser colocado explicitamente, se não for colocado não será possível acessar o conteúdo como string apenas como caracteres, ou seja, um por um através do índice.

Exemplo:

char C[8] = {‘T',’E','X','T','O',’\0’};

1. Inicializando como string, lembrando que as strings são reconhecidas apenas quando estão entre aspas:

char <identificador>[<tamanho\_do\_vetor]={“<cadeia de caracteres>” };

Neste caso o finalizador é colocado automaticamente.

Exemplo:

char C[8] = {"TEXTO"};

1. Não declarando o tamanho do vetor e inicializando todos os elementos. Neste caso, os colchetes são declarados vazios e serão preenchidos pelo pré-compilador que contará o número de elementos da inicialização.

Exemplos:

|  |  |
| --- | --- |
| int C[ ] = {‘T',’E','X','T','O',’\0’}; | equivalente a: char C[8] = {‘T',’E','X','T','O',’\0’}; |
| int C[ ] = {“TEXTO”}; | equivalente a: char C[8] = {"TEXTO"}; |

Em todos os exemplos apresentados, o resultado seria:

Uma imagem contendo relógio, objeto

Descrição gerada automaticamente

* + 1. Vetores multidimensionais

Também conhecido como matriz, o vetor multidimensional possui duas ou mais dimensões, sendo que para cada dimensão deve ter seu próprio tamanho e deve ser indexada sequencialmente.

Para melhor entender é possível comparar as dimensões com páginas, linhas, colunas e assim por diante, por exemplo um vetor com 3 dimensões, o 1º índice tem tamanho 2, o 2º índice tem o tamanho 15 e o 3º índice tem tamanho 30, na comparação anterior, então o vetor terá 2 páginas com 15 linhas e 30 colunas.

Para saber quantos elementos o vetor terá, basta multiplicar os tamanhos de cada dimensão, no exemplo anterior o vetor teria 900 elementos (2x15x30).

Assim como no vetor unidimensional, cada dimensão é indexada a partir do 0 até um número antes do tamanho da dimensão, por exemplo, um vetor bidimensional com a dimensão 1 de tamanho 3 e a dimensão 2 de tamanho 5

Tabela, Linha do tempo

Descrição gerada automaticamente

A reserva os espaços para armazenar cada um dos itens funciona da mesma forma que no vetor unidimensional, utilizando o exemplo anterior será reservado 15 (3x15) espaços de 4 bytes ([tipo de dado int](#Int)), ou seja, serão reservados sequencialmente 15x4=60 bytes para o vetor:

Linha do tempo

Descrição gerada automaticamente

* + - 1. Declarando um vetor multidimensional

A declaração de um vetor multidimensional é uma extensão natural da declaração de um vetor unidimensional, primeiro declara o [tipo dos dados](#TipoDado) que serão armazenados, depois o identificador da variável e, por fim, o tamanho de cada dimensão do vetor, utilizando o operador "[]".

<tipo\_dados> <identificador>[<tamanho1>]...[<tamanhoX>]...[<tamanhoN>]

*<tamanhoX>*: representa o tamanho da dimensão X do vetor

*<tamanhoN>*: representa o tamanho da dimensão da última dimensão do vetor, onde o N é igual ao número de dimensões do vetor.

Exemplo:

int A[2][3][5];

É importante ressaltar que a dimensão mais à direita varia mais rapidamente que a dimensão mais à esquerda, no exemplo anterior, a dimensão [2] representa o número de páginas, dimensão [3] representa o número de linhas e a dimensão [5] representa o número de colunas, o vetor terá 30 elementos.

Página [0]

Tabela, Linha do tempo

Descrição gerada automaticamente

Página [1]

Tabela, Linha do tempo

Descrição gerada automaticamente

* + - 1. Referenciando um vetor multidimensional

Cada um dos elementos de um vetor é referenciado **individualmente** por meio do índice a ele associado entre colchetes "[]".

Importante relembrar que o índice especifica a posição do elemento no vetor e que este é indexado sequencialmente, portanto, os índices de cada dimensão começam a partir do 0 até tamanhoX -1.

Para acessar/referenciar um elemento em um vetor multidimensional, apenas após a devida declaração do vetor, basta colocar os índices correspondentes em cada dimensão

Exemplo:

int A[3][5];

|  |  |
| --- | --- |
| scanf(“%d”,&A[2][4); | //A[2][4] esta referenciando elemento armazenado na posição (2,4) do vetor A |
| Tabela  Descrição gerada automaticamente | |
| printf(“%d”,A[0][2]); | //A[0][2] esta referenciando elemento armazenado na posição (0,2) do vetor A |
| Tabela  Descrição gerada automaticamente | |

O índice de um elemento pode ser indicado diretamente, por meio de um número inteiro, ou por meio de uma expressão que resulte em um valor inteiro.

Exemplo:

int A[3][5],i=0;

|  |  |
| --- | --- |
| scanf(“%d”,&A[2][i]); | // i é igual a 0, então A[2][i] é igual A[2][0] e esta referenciando elemento armazenado na posição (2,0) do vetor A |
| Tabela  Descrição gerada automaticamente | |
| printf(“%d”,A[i+1][4]); | // a expressão i+1 é igual a 0+1 = 1, então A[i+1] é igual A[1][4] e esta referenciando elemento armazenado na posição (1,4) do vetor A |
| Tabela  Descrição gerada automaticamente | |

* + - 1. Inicializando um vetor multidimensional

Os vetores de mais de uma dimensão são inicializados na declaração da mesma forma que os vetores de uma única dimensão:

<tipo\_dados> <identificador>[<tamanho1>]...[<tamanhoN>]={<lista de elementos>}

A lista de elementos é composta pelos dados do mesmo tipo, que irão compor o vetor, separados por vírgula, lembrando que os elementos devem ser listados sequencialmente, ou seja, na ordem em que serão colocados no vetor.

Há duas formas de listar os elementos:

1. Os elementos das dimensões mais à esquerda entre chaves e separadas por vírgulas:

Exemplos:

int B[2][3]={{6,5,4},{3,2,1}}

Tabela

Descrição gerada automaticamente

int A[2][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}}

Tabela

Descrição gerada automaticamente

Tabela

Descrição gerada automaticamente

1. Outra foram é usar apenas com um abre chave "{" e um fecha chave "}" simplesmente, elencando os elementos sequencialmente

Exemplo:

int B[2][3]={6,5,4,3,2,1}

Tabela

Descrição gerada automaticamente

int A[2][2][3]={1,2,3,4,5,6,7,8,9,10,11,12}

Tabela

Descrição gerada automaticamente

Tabela

Descrição gerada automaticamente

* + 1. Preenchendo e imprimindo um vetor
  1. Funções

Neste capítulo, serão tratadas as funções feitas pelo programador.

Uma função é um bloco de código que pode ser usado diversas vezes na execução de um programa, é conjunto de instruções e/ou declarações desenhadas para cumprir uma determinada tarefa, é escrita no programa uma única vez, mas pode ser executado muitas vezes durante a execução do programa.

A principal razão para usar funções é dividir uma tarefa em subtarefas específicas, simplificando e organizando o programa como um todo.

Outra razão para escrever funções é a de reduzir o tamanho do programa, uma vez que qualquer sequência de instruções ou declarações que apareça no programa mais de uma vez é candidata a se transformar em uma função, portanto, pode-se dizer que as funções servem como um atalho a um bloco de código.

A declaração geral de uma função em C é:

<tipo\_de\_retorno> <nome\_da\_funcao> (<declaracao\_parametros>)

{

<bloco\_de\_comandos>

}

* + 1. Nome da função

*O <nome\_da\_funcao>* atribuído a ela, que deve seguir as mesmas regras dos [identificadores](#identificadores), atribuído a função, é como ela será referenciada pelo programa principal (*main()*) ou outras funções, é como será chamada.

* + - 1. Chamada da função

Uma vez que a função já foi devidamente declarada e definida, ela pode ser usada sem se preocupar como foi escrita sendo chamada.

Chamar uma função é desviar o fluxo do programa para que uma bloco de comandos seja executado, ao fim dessa execução, o fluxo do programa retorna para a declaração seguinte a da chamada da função.

A chamada da função acontece dentro da função main() ou até mesmo em outra função, é feita pela referência ao *<nome\_da\_funcao>* da função seguido dos argumentos em relação aos quais se pretende executar a função, a sintaxe geral da chamada de uma função é:

<nome\_da\_funcao>(<argumentos>);

Os argumentos são os valores que serão atribuídos a cada um dos parâmetros declarados na função e devem ser compatíveis em quantidade e tipo de dados.

* + 1. Parâmetros e argumentos

Os parâmetros são um conjunto de variáveis onde serão armazenadas as cópias das informações/valores que a função irá receber para que possa executar a tarefa para a qual foi desenvolvida, constituem os dados de entrada da função.

Quando a função é declarada, após o *<nome\_da\_funcao>*, vem a *<declaracao\_parametros>* entre parênteses, que informa ao compilador quais serão as entradas de dados da função

A *<declaracao\_parametros>* é composta pela lista de variáveis separadas por virgulas e a especificação do tipo de dados e identificador para cada uma das N variáveis, com a seguinte sintaxe:

(<tipo><nome\_var1>,<tipo><nome\_var2>,...,<tipo><nome\_varN>)

Estas variáveis serão tratadas como variáveis locais da função e receberão uma cópia dos valores dos argumentos passados quando a função for chamada, portanto os valores originais passados não podem ser alterados, pois a função chamada manipula apenas as copias dos mesmos, e, quando a execução da função se encerrar, o conteúdo dessas variáveis é perdido, tendo em vista que são locais.

Exemplo:

void exemplo(int x,int z)

{

<bloco\_de\_comandos>

}

main()

{

int num1=2,num2=3;

exemplo(num1,num2);

return 0;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Linha** | **X** | **z** | **num1** | **num2** |
| INICIO | - | - | 2 | 3 |
| l1- chama função exemplo | 2 | 3 | 2 | 3 |
| l2-termina exc a função exemplo | - | - | 2 | 3 |
| FIM |  |  |  |  |

Neste exemplo, os valores das variáveis num1 e num2, passadas como argumentos no momento da chamada da função, foram copiados nos locais reservados pelas variáveis passadas na declaração de parâmetros e, quando terminou a execução da função, o conteúdo dessas variáveis foi perdido. Nota-se que o conteúdo das variáveis num1 e num2 não foram alterados em qualquer momento

Quando uma função não tiver argumentos de entrada, a *<declaracao\_parametros>* será vazia, entretanto, os parênteses da declaração da função são obrigatórios.

* + 1. Bloco de comandos
    2. Retorno da função

Uma função pode retornar um ou nenhum valor. Para retornar um valor a função que chamou utiliza-se o comando "return".

O formato geral é:

return(informação); retorna o valor

return(); não retorna valor

return; não retorna valor

A utilização do comando "return" é opcional quando não há retorno de valor.

IMPORTANTE: O comando "return" retorna UMA UNICA INFORMACAO.

Os valores devolvidos pelo comando "return" são inteiros por default como no exemplo anterior. Quando não existe valor devolvido, se a função que chamou estiver esperando algum valor, recebera lixo e ocorrera um erro de logica não detectável pelo compilador.

As funções podem devolver valores não inteiros. Para isto devemos especificar claramente o "tipo\_de\_retorno" na função chamada e declarar também este mesmo tipo na função de chamada.

O formato geral para chamar uma função que retorna uma valor diferente de inteiro é:

<nome\_da\_função> (parametros ou informacoes);

Deve-se tambem declarar o tipo da informação retornada na função que chamou junto com as variáveis locais desta.

O formato geral é:

<tipo> <nome\_da\_funcao>();

Exemplo:

float soma();

Lembrando que o formato para se declarar uma variável simples é:

<tipo> <nome\_da\_variavel>;

Exemplo:

float numero;

Note que a declaração do tipo de "retorno da função" é semelhante a declaração de uma variável simples.

O formato geral para declarar uma função que retorna uma valor diferente de inteiro é:

<tipo\_de\_retorno> <nome\_da\_funcao>(parametros ou informacoes);

Exemplo:

float soma(a,b) Atenção: aqui não tem ";" pois é a declaração da função

float a,b;

{

}

1. Bibliografia

<https://www.alura.com.br/artigos/algoritmos-e-logica-de-programacao>

<https://blog.rocketseat.com.br/logica-de-programacao-para-iniciantes-em-programacao/#:~:text=A%20l%C3%B3gica%20de%20programa%C3%A7%C3%A3o%20%C3%A9%20o%20conjunto%20de%20regras%20e,complexo%20em%20etapas%20mais%20simples>.

1. Para um estudo detalhado do funcionamento de um computador acessar a [Apostila de Arquitetura de Computadores](file:///H:\Meu%20Drive\FATEC\2º%20Semestre\1º%20Semestre\Arquitetura%20de%20Computadores\Apostila%20P2\Apostila%20P2.pdf) [↑](#footnote-ref-1)
2. o computador é composto por vários circuitos e componentes eletrônicos e a comunicação entre as unidades do computador é obtida através de fenômenos físicos, que fornecem apenas dois estados distintos: ligado - passa corrente elétrica - e desligado - não passar corrente elétrica. [↑](#footnote-ref-2)
3. Para mais informações olhar a [Apostila de Arquitetura](file:///H:\Meu%20Drive\FATEC\2º%20Semestre\1º%20Semestre\Arquitetura%20de%20Computadores\Apostila%20P2\Apostila%20P2.pdf) [↑](#footnote-ref-3)