Curso Básico de C

Apostila

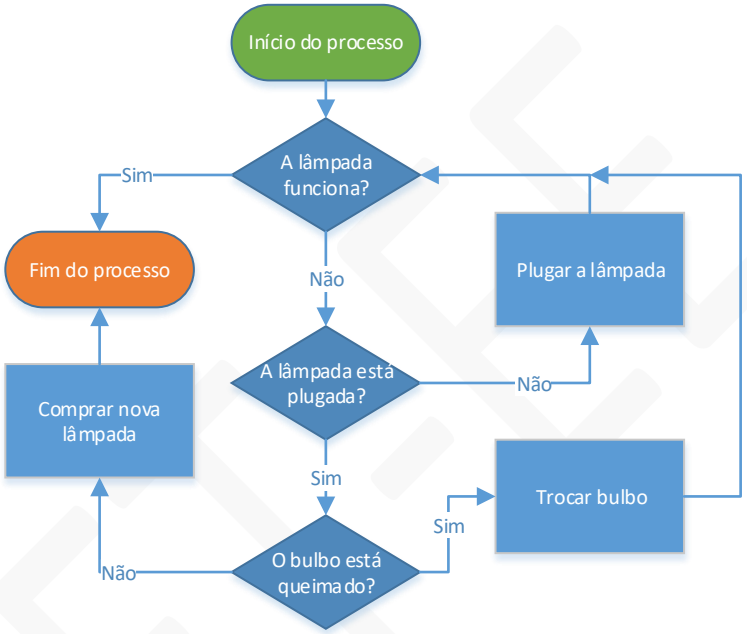
MARCOS BRESSAN

Sumário

Fluxogramas

Um fluxograma é um tipo simples de diagrama esquemático, que mostra de forma eficaz e descomplicada a transição de ações e a manipulação de informações durante a execução de um algoritmo. Cada elemento que o compõe possui desenho e função próprios, funções estas que vão desde declaração de variáveis até a tomada de decisões.

O fluxograma abaixo ilustra uma sequência de etapas a se seguir para lidar com uma lâmpada queimada:



Repare que a forma de losango dos elementos de tomada de decisão, em que se é perguntado se uma afirmativa é falsa ou verdadeira, são diferentes das formas de início/fim do processo e de execução de uma ação (retangulares).

Introdução à linguagem C

O início do C

A programação como um todo teve como início a necessidade de otimização de processos, para conseguirmos reduzir o tempo de determinada tarefa e principalmente automatizá-la, pois a máquina não possui o fator humano que influencia no processo de execução de qualquer tarefa.

Um dos primeiros programadores que se tem notícia foi uma mulher, Ada Lovelace, filha de Lord Byron, em 1842 programou algoritmos que serviriam para resolver problemas matemáticos na máquina de computar inventada por Charles Babbage. Em 1801, Joseph Marie Jacquard projetou uma máquina de tecelagem que tivesse várias opões de cor para os tecidos, ele fez isso através de cartões em que cada tipo de perfuração representava uma cor, assim podemos dizer que ela possuía um algoritmo. Através dessa mesmo princípio, só que aplicado a passagem de trens, onde o condutor observava onde era o assento de cada passageiro através das perfurações dos cartões. Com isso, Heman Hollerith projetou uma máquina que lia as perfurações dos cartões, mas para otimizar o senso americano, que precisou de bem menos tempo para averiguação dos resultados.

Em 1940 foi feito um dos primeiros computadores elétricos, muito arcaico e caro e difícil de programar, logo surgiram algumas linguagens de programação como o C-10 e o ENIAC. Anos mais tardes, foram inventados o COBOL, o LISP e o FORTAN que avançaram na busca da programação que temos hoje.

Por volta dos anos 70 surge a linguagem C, através de mudanças feitas em uma linguagem anteriormente denominada de B. Ela era fácil de programar, se comparada a outras linguagens como Assembly. Foi criada em um dos laboratórios de Gran Bell, o inventor do telefone, por Ken Thompson e Dennis Ritchie.

Níveis de linguagem

Temos vários tipos de linguagem de programação e o que define o nível de uma linguagem é a sua proximidade com a linguagem interpretada pelo processador, ou seja quanto mais próxima deste, menor o nível.

As linguagens de baixo nível são as chamadas *linguagens de máquina*, que possuem apenas sequências de bits em *strings* e tornava complicada a interpretação e a programação do computador. Por isso foi inventado o *Assembly,* que possuía alguns comandos que ajudavam na interpretação do código pelo programador e que facilitaram a programação em linguagem de máquina.

Para trazer a linguagem de programação cada vez mais para nosso cotidiano, foram criadas as linguagens de programação de alto nível que utilizam expressões do cotidiano como se e enquanto que torna mais fácil a elaboração do código. São exemplos desta a linguagem C, C++, Pascal e JAVA. As vantagens da linguagem de

baixo nível em relação as de alto nível é que um código, em Assembly por exemplo, ocupa menos espaço é mais rápido e também ajuda ter total controle do *hardware*, porém a desvantagem é o número reduzido de comandos, que dificulta sua utilização.

Inicializando um programa

Qualquer programa escrito em C possui o seguinte esqueleto:

```
#include <nome da biblioteca.h>

int main(){

// códigos e comandos;

return 0;
}
```

Código 1 - Esqueleto de qualquer programa em C

Bibliotecas (Cabeçalhos)

As bibliotecas são arquivos que possuem diversos protótipos de funções que podem ser incluídas no seu programa através da diretriz *#include*:
#include <"nome da biblioteca">

Em que *"nome da biblioteca"* representa o nome do arquivo de biblioteca que deseja-se incluir. Cada biblioteca possui funções específicas e de natureza semelhante, como manipulação de dados ou funções matemáticas, e as mais utilizadas estão listadas a seguir:

- *<stdio.h> – Standard Input/Output –* Abrange funções de manipulação, entrada e saída de dados, como leitura do teclado e escrita. As funções que mais usaremos desta biblioteca serão *scanf* e *printf*, que deverão ser melhor explanadas mais tarde.

- *<stdlib.h> – Standard Library –* Este cabeçalho define diversas funções de propósitos gerais, incluindo conversão de variáveis, geração de sequências pseudoaleatórias, alocação dinâmica de memória e comunicação com o ambiente.

- *<string.h> – String –* Contém funções específicas para o tratamento de cadeias de caracteres (*strings*).

- *<math.h> – Mathematics –* É possível fazer uso de várias funções matemáticas com a inclusão desse cabeçalho, como funções trigonométricas e exponenciais.

Main

Todo programa em C deve obrigatoriamente conter a função *main*. É a partir dela que se dá início ao programa em si. Dentro da *main* é que devemos inserir o nosso bloco de códigos e instruções, como podemos ver no **Código 1**.

A linha de código "*return 0;"*, que precede a chave de fechamento da *main*, indica ao

sistema operacional que o programa foi encerrado corretamente, mas deveremos falar de retornos de funções mais adiante.

Em qualquer trecho do código, é possível fazer a inserção de comentários – anotações livres do programador que podem ser bastante úteis para facilitar o entendimento do programa. Para comentar o final de uma linha, basta inserir duas barras (//). A partir daí, todo o texto até o fim da linha será desprezado na compilação do programa. Para comentários mais extensos, é possível inserir /* para indicar o início do comentário e */ para indicar o término.

```
/* ... */
int main () {    //  Toda essa frase será desprezada no processo de compilação :)

/* É POSSÍVEL AINDA
INSERIR COMENTÁRIOS
DE VÁRIAS LINHAS */
/* ... */
```

Código 2 - Comentários

Variáveis

Em qualquer linguagem de programação, se faz necessário o armazenamento constante de dados, seja quando precisamos guardar o resultado de uma soma ou mesmo salvar uma informação para manipulá-la e mais tarde mostrá-la ao usuário. Por exemplo, antes de efetuarmos o produto de dois números fornecidos pelo usuário, precisamos guardar os dois valores em duas variáveis distintas, para somente a seguir realizar o produto e mostrar o resultado.

No C, dispomos de diversos tipos de variáveis, que possuem naturezas distintas e ocupam diferentes quantidades de bytes na memória do computador. É muito importante conhecermos que tipo de variável vamos necessitar e utilizar o mais adequado:

- *int (inteiro):* como o próprio nome diz, é utilizada para guardar valores numéricos inteiros, tanto negativos quanto positivos. É o tipo mais comum de variável.
- *char (caractere):* indicada para guardar e representar caracteres.
- *float (ponto flutuante):* esta variável é capaz de armazenar valores de ponto flutuante (números reais).
- *double (ponto flutuante de dupla precisão):* de modo semelhante à variável de tipo *float*, esta armazena um número real com maior precisão, ao custo de ocupar mais espaço na memória.

Modificadores

A esses quatro tipos de variáveis podemos atribuir ainda **modificadores**, que basicamente alteram a maneira de armazenamento de uma variável.

Os modificadores *unsigned* e *signed* podem ser usados somente sobre variáveis do tipo *int* e não mudam o tamanho ocupado pela variável na memória. Por padrão, variáveis inteiras são *signed* (com sinal), mas podem ser alteradas para *unsigned* para limitá-las somente a

valores não negativos (sem sinal). Variáveis *int* podem ainda ser alteradas pelos modificadores *short*, *long* e *long long*, que podem reduzir ou aumentar as suas dimensões.

Variáveis *char* podem ser modificadas (embora isso não seja muito praticado para programas de fins gerais) pelos modificadores *signed* e *unsigned*.

Às variáveis *float* não é possível aplicar nenhum modificador e, às do tipo *double*, somente o modificador *long*.

Para cada máquina em que se trabalha, uma mesma variável pode ocupar diferentes tamanhos na memória. Ademais, não há garantia de que, por exemplo, uma variável declarada como *long int* seja maior do que uma do tipo *int*; tudo o que se sabe é que uma variável *short int* NÃO é maior que uma somente *int*, que por sua vez NÃO é maior que uma *long int*, que por fim NÃO é maior que uma *long long int.*

Modificador	**Variável à qual pode ser aplicado**
signed	*int*, *char*
unsigned	*Int*, *char*
short	*int*
long	*int*, *double*
long long	*int*

Tabela 1 - Modificadores

Declaração de Variáveis

Para atribuirmos valores a variáveis, é preciso que antes estas sejam declaradas. É nesse momento que o programa informa ao sistema para reservar um espaço de memória para a variável, e o tamanho deste espaço vai depender do tipo e se há algum modificador associado a ela.

A sintaxe de declaração de uma variável segue o seguinte esquema:

```
/* ... */
// <modificadores> <tipo_da_variável> <nome_da_variavel>;
long unsigned int x;
double variavel;
char caractere;
/* ... */
```

Código 3 - Declaração de variáveis

O Código 3 mostra diferentes variáveis sendo declaradas. Entretanto, podemos notar que nenhum valor foi atribuído a elas ainda. Isso pode ser feito simplesmente igualando a variável ao valor desejado:

```
/* ... */
long unsigned int x;   //   declara uma variável do tipo long unsigned int de nome “x”
double variavel;
char caractere;

caractere = ‘M’;
x = 255;
variavel = 98.9;
/* ... */
```

Código 4 - Declaração de variáveis seguida de atribuição.

Note que para passar um caractere como o valor de uma variável *char* é preciso colocá-lo entre apóstrofes (‘ ‘), e que o separador decimal utilizado para números reais em variáveis *float* e *double* é o ponto (.) em vez de vírgula (,).

Há uma maneira simplificada de, ao passo em se declara uma variável, se atribuir a seguir um valor a ela, como mostrado no Código 5.

```
/* ... */
float numero = 5.9;    //  declara “numero” e atribui a ele 5.9
char letra = ‘A’;    //   “letra” é um char que armazena a letra A
/* ... */
```

Código 5 - Declaração de variáveis

Entrada e Saída de dados

Além de proporcionar uma interação forte entre o uso do potencial de computação da máquina e o programador, é vital que o programa ofereça meios de interação com o usuário leigo, e que ele possa de maneira fácil usufruir dessas ferramentas sem se ver preso ao entendimento da própria linguagem de programação. Para isso, qualquer *software* desenvolvido buscar construir um ambiente de fácil comunicação com o usuário.

As funções *printf* e *scanf* possuem basicamente esse objetivo: a primeira serve para mostrar ao usuário alguma informação, e a segunda para recolher algum valor digitado pelo usuário no teclado. Ambas podem ser utilizadas no programa quando incluída a biblioteca *<stdio.h>*.

A função *printf* (*print* formatted) recebe os seguintes parâmetros: uma *string* (sequência de caracteres; será explicada mais adiante) de controle e as variáveis que serão incorporadas nessa *string*.

Acompanhe o seguinte exemplo:

```
#include <stdio.h>
int main(){
        int x = 25;
        printf("Tenho menos de %d anos", x);
        return 0;
}
```

Código 6 – A mensagem mostrada é Tenho menos de 25 anos

```
#include <stdio.h>
int main(){
        char carac = 'W';
        float altura = 2.98;
        printf("Hello %corld, tenho %f cm de altura!", carac, altura);
        return 0;
}
```

Código 7 – A mensagem mostrada é Hello World, tenho 2.98 cm de altura!

Perceba que o código *%d* na *string* de controle do **Código 6** será substituído pelo valor gravado na variável *x* (o inteiro 25). Cada tipo de variável possui um código diferente a ser substituído na *string* de controle da função *printf*. A tabela a seguir ilustra os principais códigos:

Código no *printf*	**Tipo de Variável**
%d ou *%i*	(*signed*) *int*
%u	*unsigned int*
%f	*float* ou *double*
%c	*char*
%p	Ponteiro*
%s	Sequência de caracteres* (*string)*
%%	Insere um %

Tabela 2 - Códigos de variáveis na função printf

De modo semelhante trabalha a função *scanf*. Porém, ao ser chamada, ela solicita ao usuário uma entrada que será gravada na variável indicada assim que a tecla *Enter* for pressionada. Diferentemente do *printf*, é preciso colocar um "E comercial" (&) antes da variável a ser modificada.

```
#include <stdio.h>
int main(){
        unsigned idade;
        printf("Digite sua idade, pessoa:\n");
        scanf("%u", &idade);    // o valor digitado pelo usuário será gravado em idade
        printf("Verdade que possui %u anos? Uau!", idade);
        return 0;
}
```

Código 8 – Função scanf

A tabela de códigos da *string* de controle da função *scanf* difere um pouco da anterior:

Código no *scanf*	**Tipo de Variável**
%d ou *%i*	(*signed*) *int*
%u	*unsigned int*
%f	*float*
%lf	*double*
%c	*char*
%s	Sequência de caracteres* (*string)*

Tabela 3 - Códigos de variáveis na função scanf

Tomada de Decisão

Frequentemente em nossos programas necessitamos testar afirmativas, avaliar informações, coletar dados, comparar valores e, posteriormente, tomar decisões com base nesses elementos. A linguagem C é dotada de comandos que permitem testar parâmetros e executar determinados blocos de código dependendo se o valor avaliado foi verdadeiro ou falso.

Verdadeiro ou Falso

Os comandos *if* e *else* funcionam de maneira semelhante a como faríamos em nosso cotidiano com a sentença "**se** isso é verdade, então devo fazer aquilo; **senão**, devo tomar essa ação". Acompanhe o Código 6:

```
/* ... */
if (1 > 2) {            //  SE 1 é maior que 2...
        /*  códigos a serem executados em caso verdadeiro */
} else {                //   SENÃO
        /*  códigos a serem executados em caso falso */
}
/* ... */
```

Código 9 – If/else

Perceba a presença de um comparador, ou operador relacional: o símbolo da desigualdade *maior que*, que testa a veracidade da "afirmação" 1>2. A tabela a seguir lista os operadores relacionais:

Símbolo em C	Operador Relacional
==	Igualdade. Retorna *verdadeiro* caso os dois valores sejam iguais.
!=	Desigualdade. Retorna *verdadeiro* caso os dois valores sejam diferentes.
<	Menor que. Caso particular de desigualdade que retorna *verdadeiro* se o valor à esquerda for menor que o valor à direita.
>	Maior que. Análogo ao comparador acima, retorna *verdadeiro* somente quando o valor à esquerda é maior que o da direita.
<=	Menor ou igual a. Pode ser entendido como uma união do primeiro e do terceiro comparador.
>=	Maior ou igual a. União do primeiro e do quarto comparador.

Tabela 4 - Operadores Relacionais

```
/* ... */
int x = 25;
int y = 40;
if ( x != y ) {    //    Testa se o valor de x é diferente do de y
x = y-2;     //   Como o teste retorna verdadeiro, esse bloco será executado
} else {
y = x+2;    //   Já esse bloco não será.
}
/* ... */
```

Código 10 – Exemplo de uso de tomada de decisão com variáveis

Para unir diversas proposições, podemos utilizar operadores lógicos. São eles:

Símbolo em C	Operador Lógico
&&	Operador AND. Retorna *verdadeiro* somente quando as duas proposições são *verdadeiras*. *Falso* caso ao menos uma seja *falsa*.
||	Retorna *verdadeiro* quando ao menos uma proposição é *verdadeira*. Retorna *falso* apenas quando todas são falsas.
!	Inverte o nível lógico de uma proposição, retornando *verdadeiro* quando esta for *falsa* e *falso* quando for *verdadeira*.

Tabela 5 - Operadores Lógicos

```
float x=2, y=x+2, z=y+x;
if( (x>=0) && (x<=100) )
        printf("x esta no intervalo real [0, 100]");
if ( !(!(z<2) || (y>x) ) )     //  se y<=x e z<2
        printf("Deu verdadeiro!");
```

Código 11 – Exemplo de uso de tomada de decisão com variáveis

Ainda é possível escrevermos uma sequencia de *if's* e *else's* aninhados um ao outro ou empregar *else if* para uma nova avaliação condicional dentro da exceção da anterior. Acompanhe:

```
#include <stdio.h>

int main(){
	int x;
	printf(“Digite um numero legal\n”);
	scanf(“%d”, &x);
	if (x<0){
		printf("Eu não trabalho com negativos!");
	} else if (x<10){
		printf("Parece um valor pequeno");
	} else if (x<100) {
		printf("Olha as coisas melhorando...");
	} else if (x<1000) {
		printf("Apelou");
	} else {
		printf("Tudo na vida tem limites, colega!!!\n");
		if(x>100000)
			printf(“Exceto voce”);
	}
	return 0;
}
```

Código 12 – Condições aninhadas

Note que cada avaliação depois da primeira só ocorre se a anterior for falsa.

Switch

É um tipo particular de condicional discreto, que avalia o valor de uma variável *int* ou *char* e salta para cada *case* correspondente. A leitura dos códigos dentro do *switch* prossegue normalmente até que todo o código seja lido ou um comando *break* seja encontrado. O código a seguir ilustra o seu funcionamento:

```
char x = 'a';
switch(x) {
case 'a':
        printf("Essa eh a primeira vogal");
        break;
case 'e':   /*  como esse caso não termina com break, as linhas abaixo continuarão a
ser executadas */
case 'i':
printf("Caso e ou i levaram a este texto");
        break;   /*   se essa linha for executada, o programa sai do bloco do switch */
case 'o':
printf("Ooooooooh ou ... ");
/* Não possui break. O case abaixo também será executado */
case 'u':
printf("Uuuuuuuuuh");
        break;
default:   //  caso padrão, quando nenhum case acima é acionado
        printf("É maiúscula ou consoante!");
        break;  //  é opcional aqui, o código ia acabar mesmo :B
}
```

Código 13 – Uso de switch

Deixar o *default* por último não é obrigatório, mas facilita o entendimento do código. Atenção ao uso consciente dos comandos *break* aqui.

Estruturas de Repetição

Até aqui aprendemos algumas formas de se avaliar uma expressão (ou mais) e executar determinado bloco de códigos quando ela for verdadeira.

Contudo, às vezes se torna necessário fazer mais que isso: repetir aquele bloco *enquanto* certa expressão for verdadeira. Os três comandos que veremos a seguir funcionam dessa maneira: eles avaliam uma expressão constantemente, e em cada iteração executam um bloco de código, não necessariamente nessa ordem.

Como nosso objetivo, na maioria das vezes, não é entrarmos em um caso em que uma expressão é sempre verdadeira (como *1==1*), o que resultaria num *loop infinito*, seria prudente construirmos uma expressão que seja verdadeira durante algumas iterações mas que em algum momento se torne falsa.

A sintaxe do *while* é bem intuitiva, e se assemelha bastante com a do *if*. Acompanhe:

```
int x = 0;   //  valor inicial de x
while(x<100){              // Enquanto x for menor que 100 (0 a 99)
        printf("%d- Nao devo falar mal do meu professor\n", x);
        x++;   //   equivalente a x=x+1;
        /*  quando x assumir o valor 100, o bloco não será mais executado  */
}
```

Código 14 – Uso de repetição com o comando While

O fato de a variável *x* ser incrementada a cada iteração faz o "atual" valor de *x* ser maior até que *x<100* se torne falso.

O comando *do-while* segue a mesma lógica do *while*, diferindo apenas porque o código é executado antes da avaliação. Em outras palavras, mesmo que a expressão seja falsa, há a garantia de que o bloco será executado ao menos uma vez.

```
int x = 0;
do{
        printf("x vale %d\n",x);
        x++;
} while (x<3);
```

Código 15 – Uso de repetição com o comando Do-While

Por fim, o comando *for* trata-se de uma maneira mais compacta de dar um valor inicial à variável de iteração, avaliá-la e incrementá-la.

```
int x;
for(x=0; x<100; x++)    ///  (inicialização; avaliação; passo final)
        printf("Voce ja elogiou seu professor hoje?\n");

/*                  EXEMPLO EQUIVALENTE
x = 0;
while(x<100){
        printf("Voce ja elogiou seu professor hoje?");
        x++;
}
*/
```

Código 16 – Uso do comando for

Funções

É muito provável que, durante a execução de nosso programa, necessitemos realizar um mesmo algoritmo diversas vezes, mas, além de ser cansativo, escrever um mesmo código repetidamente pode tornar a leitura muito ineficiente. Funções, logo, são utilizadas para esse fim: elas podem receber variáveis como parâmetros e retornar informações específicas para o ponto de onde foram chamadas.

Tome como exemplo a função *sin* (seno trigonométrico), que pode ser incluída com a biblioteca matemática *<math.h>*: ela recebe um parâmetro (uma variável ou valor do tipo *double*) e retorna um valor de mesmo tipo.

```
#include <math.h>
#include <stdio.h>
/* ... */
int main(){
        double x=0;
        double y = sin(x);    /*   aqui, a função sin é chamada com o parâmetro x=0; A
função devolve o valor 0, correspondente ao seno de x, e isso é armazenado em y */
        printf("O seno de %f eh %f", x, y);
        return 0;
}
```

Código 17 – Função sin da biblioteca <math.h>

Os próprios comandos *printf* e *scanf*, que temos utilizado frequentemente, são funções da biblioteca <stdio.h>. O *main*, como já havíamos citados, é uma função "especial" que caracteriza-se por ser a primeira a ser chamada quando executamos o programa. Nos nossos exemplos, ela não precisa de nenhum parâmetro, mas deve retornar um inteiro.

Para criar nossa própria função, devemos saber antes de escrevê-la:

- O tipo de variável que ela retorna (*int*, *char*, *float*, *double*) - permitindo ainda o uso de modificadores - ou se ela não retorna valor algum (*void*);
- Os parâmetros que ela deverá receber (qualquer um dos tipos, com modificadores ou não, ou sem parâmetros)
- O nome da função.

Como exemplo, o seguinte código expressa o quadrado de um número inteiro: ele recebe um parâmetro *int* e devolve um outro *unsigned int*.

```
#include <stdio.h>
unsigned int quadrado (int a){    /*   função de nome "quadrado",
        recebe inteiro com sinal, devolve o quadrado (sem sinal)  */
        return a*a;
}
int main(){
        int x;
        scanf(“%d”, &x);
        printf("O quadrado de %d resulta em %u", x, quadrado(x));
        return 0;
}
```

Código 18 – Função implementada “quadrado”

A estrutura do C não permite a criação de funções dentro de outras; as funções devem ser independentes entre si. Isso implica que não é possível criar funções dentro da própria *main*, mas não quer dizer que não possamos chamar, a partir de uma função, outras.

Para chamar uma função a partir de outra, é necessária que a primeira tenha sido declarada anteriormente, caso contrário ocorrerá um erro na compilação: seria como usarmos uma variável antes de declará-la. Entretanto, caso ainda queiramos colocar funções personalizadas depois da *main*, por exemplo, basta que escrevemos logo de início *protótipos*

de funções. O código abaixo é equivalente ao anterior, mas, empregando o protótipo da função quadrado antes da *main*, é possível definir logo após a função:

```
#include <stdio.h>

unsigned int quadrado (int a);  //  esse é o protótipo da função quadrado

int main(){
        printf("O quadrado de %d resulta em %u", x, quadrado(x));
        return 0;
}

unsigned int quadrado (int a){     // agora é possível defini-la depois da main
        return a*a;
}
```

Código 19 – Protótipo de função

Vetores

Um vetor corresponde a uma sequência finita de variáveis de mesmo tipo. A sintaxe que caracteriza a declaração de um vetor pode ser vista no seguinte código:

```
int x[2];
```

Em que x é o nome do vetor, e o número 2 identifica que nesse exemplo foi reservado um espaço de memória para duas variáveis do tipo *int*. A maneira que referenciamos estas variáveis é por meio de índices: podemos ter acesso à primeira posição do vetor x com o índice 0, e à segunda com o índice 1.

Perceba que, na declaração, escrevemos a quantidade *n* de elementos que o vetor deverá conter, mas os elementos desse vetor serão indexados de *0* a *n-1*.

```
int x[2];
x[0] = 5;
x[1] = 17;
/*   Atenção para não atribuir valores à posição x[2]: como esse espaço não foi
reservado, teríamos um problema de overflow. */
```

Código 20 – Atribuição de valores a posições do vetor

Caso soubéssemos de antemão os valores dos elementos do vetor no momento de sua declaração, poderíamos declarar o vetor e atribuirmos esses valores com a seguinte sintaxe:

```
float vetor[] = {1.6, 6.5, 9.0, 4.65};
/* vetor de 4 elementos, em que x[0] = 1.6, x[1] = 6.5, x[2] = 9.0, x[3] = 4.65 */
char caracteres[] = {'L', 'e', 't', 'r', 'a', 's'};
```

Código 21 – Atribuição de valores a posições do vetor

Perceba que não é necessário explicitar a quantidade de elementos do vetor entre os colchetes; o compilador automaticamente conta quantos valores foram escritos e os atribui aos índices de forma sequencial.

Strings

Strings nada mais são que vetores do tipo *char*, em que cada elemento é um caractere que compõe um texto. Entretanto, há um artifício usado em C para identificar o fim de uma string: colocar um caractere nulo /0 na posição final do vetor. Desse modo, podemos percorrer uma string por meio de laços (*for*, *while*, ...) e saber a posição de memória em que ela termina ao reconhecer o caractere nulo na última posição do vetor.

Há diferentes maneiras de se declarar uma *string*:

```
char string[] = "Podemos declarar dessa maneira; o compilador automaticamente
conta a quantidade de caracteres e posiciona o caractere nulo no final."
char outra[] = {'C', 'o', 'm', 'o', ' ', 'u', 'm', ' ', 'v', 'e', 't', 'o', 'r', '/0'};
// Tivemos de colocar manualmente o caractere nulo para identificar o fim da string
```

Código 22 – Declaração de strings

Dispomos da biblioteca padrão *<string.h>* com funções destinadas ao tratamento de strings que pode ser incluída no programa. Seguem algumas das funções mais usadas:

- *int strlen(char string[]):* retorna um inteiro correspondente à quantidade de elementos em uma string. Não conta o caractere nulo.
- *int strcmp(char string1[], char string2[]):* compara duas strings, estabelecendo um critério de ordem alfabética: retorna -1 caso a primeira anteceda a segunda, 1 caso a segunda anteceda a primeira e 0 caso as duas strings sejam idênticas.
- *int strcpy(char string1[], char string2[]):* copia o conteúdo da string2 para string1. É necessário cuidado por parte do programador com o espaço alocado para a string1, pois este não deve ser menor que o tamanho da string2.
- *int strcat(char string1[], char string2[]):* concatena (une) a *string2* à *string1*. É necessário cuidado por parte do programador com o espaço alocado para a string1, pois este não deve ser menor que os tamanhos da string1 e da string2 somados.

```
#include <string.h>
#include <stdio.h>
int main(){
	unsigned int x, y;
	char vetor[] = "Curso Basico de C";
	char outro[] = "Professor bonito, o seu";
	x = strlen(vetor);   //   aqui x recebe 17
	y = strlen(outro) + strlen(vetor);  //  y recebe 40
	switch(strcmp(vetor, outro)){
	case -1:
		printf("A frase '%s' antecede '%s'.", vetor, outro);
		break;
	case 1:
		printf("A frase '%s' antecede '%s'.", outro, vetor);
		break;
	case 0:
		printf("As duas frases sao iguais!");
		break;
	}
	return 0;
}
```

Código 23 – Exemplo de uso das funções strlen e strcmp

Matrizes

O conceiro de vetor pode ser estendido para dimensões maiores, tornando-se elementos que chamamos de matrizes. A declaração de uma matriz é análoga à de um vetor.

```
double matriz[2][2]; //  matriz de dimensão 2x2. Cada dimensão é indexada de 0 a 1
matriz[1][0] = 3.3;  //  o elemento da segunda linha (índ. 1) e primeira coluna (0) vale 3.3
long long int x[][2] = {123, 413, 524, 643};
/* os valores são atribuídos sequencialmente aos elementos x[0][0], x[0][1], x[1][0] e x[1][1] */
int dimensao[4][5][4][5];   //   podemos estender a qualquer dimensão, nesse caso, 4
```

Código 24 – Declaração de matrizes.

Ponteiros

Diferente de outras linguagens de programação, a linguagem C permite um controle muito maior dos elementos com os quais trabalhamos. Uma ferramenta muito poderosa são os ponteiros. Um ponteiro é um tipo especial de variável que armazena o endereço de memória. Este pode ser o endereço de uma variável como aquelas já estudadas ou de outro ponteiro.

Ao declarar um ponteiro, precisamos especificar o tipo de variável para a qual ele vai apontar. O operador & é o responsável por fornecer o endereço de uma variável, ou seja, dada uma variável, ele nos fornece o endereço dela na memória. Já o operador * é responsável por ler o valor que está contido em um endereço. Vejamos o código a seguir:

```
int x = 2;
int* pont;    //   declara um ponteiro pont para inteiro
pont = &x;    //    aqui, o endereço de x é armazenado em pont

*pont = 78;
/*  aqui acessamos o valor endereçado pela variável pont e o modificamos. A variável
x é indiretamente alterada para 78 */

*(&x) = 13;
/* nessa ordem, os operadores * e & podem ser tratados como inversos um do outro.
Contudo, não faria sentido escrever &(*x) porque x não é ponteiro */
```

Código 25 – Exemplo de uso de ponteiro

*OBS.: É crucial saber distinguir entre as diversas aplicações do símbolo *. Até aqui, já o aplicamos em comentários multilinhas, produto aritmético, declaração de ponteiros e acesso ao valor de um endereço.*

Aritmética de Ponteiros

Suponha que nossa máquina reserve 4 bytes para variáveis do tipo *int*. Então digamos que declaramos uma variável *x* desse tipo, atribuímos a ela um valor e a seguir lemos esse valor. O computador já reconhece o tipo da variável e quantos bytes de memória sequenciais deve ler até que toda a informação contida em *x* seja capturada.

É por isso que precisamos informar o tipo de variável que um ponteiro endereça: ao usar o operador *, já será sabida a quantidade de bytes a ser lida.

Como um ponteiro guarda um endereço em formato numérico, podemos somar ou subtrair valores inteiros dele para acessar endereços anteriores ou posteriores na memória. É dessa maneira que funcionam os vetores: os elementos estão armazenados de forma sequencial na memória e o nome do vetor é um ponteiro para a sua primeira posição (de índice 0); os endereços subsequentes são acessados somando números inteiros positivos ao endereço da primeira posição.

```
array[2];   //   declaração de um vetor de inteiros com 2 elementos

array[0] = 56;
/* Seria equivalente escrevermos *array = 56, pois, como foi dito antes, o nome do
vetor já é um ponteiro para a sua primeira posição */

*(array+1) = 76;  /* Equivalente a escrever array[1]=76 , pois capturamos o valor
contido no endereço subsequente ao índice 0 (o índice 1) */
//   de maneira geral, array[n] é equivalente a *(array+n), onde n é um inteiro.

int* ponteiro = array+1;
/*   declaramos um ponteiro que endereça a segunda posição do vetor array */

ponteiro[2] = 13;   //   É atribuído 13 a array[3]

*(ponteiro+3) = 0;   //  É atribuído 0 a array[4]
```

Código 26 – Uso de ponteiros com vetores

Algumas das diferenças entre ponteiros e vetores são:

- A declaração de vetores já reserva um espaço pré-definido de elementos, enquanto para ponteiros nenhum espaço é reservado;
- É possível alterar o valor de um ponteiro, mas o endereço de um vetor declarado é constante e não pode ser modificado pelo programador;
- O uso de ponteiros permite criar vetores de tamanhos flexíveis quando associado a funções de alocação dinâmica de memória. Esse conteúdo foge do escopo deste curso por tratar de um assunto mais avançado de C.

Qualquer ponteiro declarado, independente do tipo de variável que ele endereça, ocupa o mesmo valor de bytes na memória. Todavia, informar esse tipo auxilia também ao programa reconhecer que, ao ser somado 2 a um ponteiro que endereça um inteiro, ele deve saltar 4x2 = 8 bytes a partir do endereço armazenado no ponteiro para iniciar a leitura (Se ainda válida aquela suposição de que em nossa máquina um inteiro ocupa 4 bytes).

Funções += Ponteiros

Vamos analisar o seguinte protótipo função:

```
void funcao(int i, char c, float r);
```

Os nossos estudos até aqui nos permitem reconhecer que, embora não saibamos o que de fato esta função realiza, sabemos que ela recebe três parâmetros e não retorna nenhum valor (*void*).

Quando uma função como essa é invocada pela *main* (ou por outra função, ou por ela mesma), os valores dos parâmetros no momento da chamada são COPIADOS para as variáveis *i*, *c* e *r*. Logo, qualquer alteração desses valores dentro da *funcao* não serão aplicados quando

retornarmos para *main*. Vejamos:

```
#include <stdio.h>
void SomaDez(int x);
int main(){
        int x = 20;
        SomaDez(x);    //  o VALOR de x (20) é mandado para a função SomaDez
        printf("x = %d", x);   //  mas aqui, x ainda vale 20
        return 0;
}
void SomaDez(int x){   //   essa variável x (independente da outra x) vai receber 20
        x += 10;   //    e agora vai valer 30
}
```

Código 27 – Função

O que passa é que copiamos o valor de *x* para outra variável *x*, de outra função, mas as duas NÃO OCUPAM O MESMO ESPAÇO NA MEMÓRIA. Caso quiséssemos de fato MODIFICAR a variável *x* contida na *main*, devemos passar o seu endereço na memória e modificá-la indiretamente! A função a seguir funciona como gostaríamos (alteramos o nome das variáveis para não causar mais confusão):

```
#include <stdio.h>
void SomaDez(int* y);   //   agora y vai armazenar um Endereço
int main(){
        int x = 20;
        SomaDez(&x);    //  o ENDEREÇO de x é mandado para a função SomaDez
        printf("x = %d", x);   //  finalmente aqui x vale 30!
        return 0;
}
void SomaDez(int* y){    //   esse ponteiro  vai receber o endereço de x
        (*y) += 10;        //    e agora vai acessá-lo e incrementá-lo de 10 unidades
}
```

Código 28 – Função implementada "quadrado"

A essa técnica de passar um ponteiro para a variável em questão chama-se *passar parâmetro por referência*.

Vimos que vetores podem ser referenciados como ponteiros; portanto, acessar as posições de um vetor em uma função são bem intuitivas. A seguinte função percorre um vetor de caracteres (string) trocando todas letras E por 3 e A por 4.

```
#include <stdio.h>
void estilo(char* string);   //   protótipo
int main(){
        char texto[] = "Esse aqui pode ser considerado meu texto, caro colega";
        puts(texto);
        estilo(texto);   //   "texto" já é o endereço da posição 0 da string
        puts(texto);
        return 0;
}
void estilo(char* string){   //   recebe um ponteiro para a posição 0 da string
        int i;   //   variável iteradora
        for(i=0; string[i] != '\0'; i++)   //   enquanto não encontrar o caractere nulo
        switch(string[i]) {   //   ou *(string + i)
                case 'A': case 'a':
                        string[i] = '4';   //  ' ' trata-se do CARACTERE, não do INTEIRO
                        break;
                case 'e': case 'E':
                        string[i] = '3';
                        break;
        }
}
```

Código 29 – Função com ponteiros que modifica variável.