1. Fundamentos da Linguagem C

Neste capitulo serão vistos os fundamentos da linguagem C. O conceito de linguagem de programação, linguagens de alto e baixo nível, linguagens genéricas e especificas. Será visto um pouco do histórico da criação da linguagem e a descrição das características mais importantes da linguagem C. Finalmente, será visto o aspecto geral de um código fonte escrito em C.

1.1 Linguagens de Programação

Um *programa* de computador é um conjunto *instruções* que representam um *algoritmo* para a resolução de algum problema. Estas instruções são *escritas* através de um conjunto de *códigos* (símbolos e palavras). Este conjunto de códigos possui regras de estruturação lógica e sintática própria. Dizemos que este conjunto de símbolos e regras formam uma *linguagem de programação*.

1.1.1 Exemplos de códigos.

Existem muitas linguagens de programação. Podemos escrever um algoritmo para resolução de um problema por intermédio de qualquer linguagem. A seguir mostramos alguns exemplos de trechos de códigos escritos em algumas linguagens de programação.

Exemplo: trecho de um algoritmo escrito em **Pseudo-linguagem** que recebe um número `num` e escreve a tabuada de 1 a 10 para este valor:

```
leia num
para n de 1 até 10 passo 1 faça
  tab ← num * n
  imprime tab
fim faça
```

Exemplo: trecho do mesmo programa escrito em **linguagem C**:

```
scanf(&num);
```

```
for(n = 1; n <= 10; n++){
  tab = num * n;
  printf(”\n %d”, tab);
};
```

Exemplo: trecho do mesmo programa escrito em **linguagem Basic**:

```
10 input num
20 for n = 1 to 10 step 1
30 let tab = num * n
40 print chr$ (tab)
50 next n
```

Exemplo: trecho do mesmo programa escrito em **linguagem Fortran**:

```
   read (num);
   do 1 n = 1:10
     tab = num * n
     write(tab)
10 continue
```

Exemplo: trecho do mesmo programa escrito em **linguagem Assembly** para INTEL 8088:

```
MOV CX,0
IN  AX,PORTA
MOV DX,AX
LABEL:
INC CX
MOV AX,DX
MUL CX
OUT AX, PORTA
CMP CX,10
JNE LABEL
```

1.1.2 Linguagens de baixo e alto nível.

Podemos dividir, genericamente, as linguagens de programação em dois grandes grupos: as linguagens de *baixo nível* e as de *alto nível*:

Linguagens de baixo nível: São linguagens voltadas para a máquina, isto é, são escritas usando as instruções do microprocessador do computador. São genericamente chamadas de linguagens *Assembly*.

Vantagens: Programas são executados com maior *velocidade* de processamento. Os programas ocupam menos *espaço* na memória.

Desvantagens: Em geral, programas em Assembly tem pouca *portabilidade*, isto é, um código gerado para um tipo de processador não serve para outro. Códigos Assembly não são estruturados, tornando a *programação* mais difícil.

Linguagens de alto nível: São linguagens voltadas para o ser humano. Em geral utilizam sintaxe estruturada tornando seu código mais legível. Necessitam de *compiladores* ou *interpretadores* para gerar instruções do microprocessador. Interpretadores fazem a interpretação de *cada* instrução do programa fonte executando-a dentro de um ambiente de programação, **Basic** e **AutoLISP** por exemplo. Compiladores fazem a tradução de *todas* as instruções do programa fonte gerando um programa executável. Estes programas executáveis (`*.exe`) podem ser executados fora dos ambientes de programação, **C** e **Pascal** por exemplo. As linguagens de alto nível podem se distinguir quanto a sua aplicação em *genéricas* como **C**, **Pascal** e **Basic** ou *específicas* como **Fortran** (cálculo matemático), **GPSS** (simulação), **LISP** (inteligência artificial) ou **CLIPPER** (banco de dados).

Vantagens: Por serem compiladas ou interpretadas, tem *maior portabilidade* podendo ser executados em varias plataformas com pouquíssimas modificações. Em geral, a programação torna-se mais fácil por causa do maior ou menor grau de estruturação de suas linguagens.

Desvantagens: Em geral, as rotinas geradas (em linguagem de maquina) são mais genéricas e portanto mais complexas e por isso são mais lentas e ocupam mais memória.

1.2 Linguagem C

A **linguagem C** é uma linguagem de *alto nível, genérica.* Foi desenvolvida *por* programadores *para* programadores tendo como meta características de flexibilidade e portabilidade. O C é uma linguagem que nasceu juntamente com o advento da teoria de *linguagem estruturada* e do *computador pessoal*. Assim tornou-se rapidamente uma linguagem “popular” entre os programadores. O C foi usado

para desenvolver o sistema operacional **UNIX**, e hoje esta sendo usada para desenvolver novas linguagens, entre elas a linguagem **C++** e **Java**.

1.2.1 Características do C

Entre as principais características do C, podemos citar:

- O **C** é uma linguagem de alto nível com uma sintaxe bastante estruturada e flexível tornando sua programação bastante simplificada.
- Programas em **C** são compilados, gerando programas executáveis.
- O **C** compartilha recursos tanto de alto quanto de baixo nível, pois permite acesso e programação direta do microprocessador. Com isto, rotinas cuja dependência do tempo é crítica, podem ser facilmente implementadas usando instruções em Assembly. Por esta razão o **C** é a linguagem preferida dos programadores de aplicativos.
- O **C** é uma linguagem estruturalmente simples e de grande portabilidade. O compilador C gera códigos mais enxutos e velozes do que muitas outras linguagens.
- Embora estruturalmente simples (poucas funções intrínsecas) o **C** não perde funcionalidade pois permite a inclusão de uma farta quantidade de rotinas do usuário. Os fabricantes de compiladores fornecem uma ampla variedade de rotinas pré-compiladas em bibliotecas.

1.2.2 Histórico

1970: *Denis Ritchie* desenha uma linguagem a partir do **BCPL** nos laboratórios da *Bell Telephones, Inc.* Chama a linguagem de **B**.

1978: *Brian Kerningham* junta-se a *Ritchie* para aprimorar a linguagem. A nova versão chama-se **C**. Pelas suas características de portabilidade e estruturação já se torna popular entre os programadores.

~1980: A linguagem é padronizada pelo *American National Standard Institute*: surge o **ANSI C**.

~**1990:** A *Borland International Co*, fabricante de compiladores profissionais escolhe o **C** e o **Pascal** como linguagens de trabalho para o seu *Integrated Development Enviroment* (Ambiente Integrado de Desenvolvimento): surge o **Turbo C**.

~1992: O C se torna ponto de concordância entre teóricos do desenvolvimento da teoria de *Object Oriented Programming* (programação orientada a objetos): surge o **C++**.

1.3 Estrutura de um programa em C

Um programa em **C** é constituído de:

- um cabeçalho contendo as **diretivas de compilador** onde se definem o valor de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, declaração de rotinas, etc.
- um bloco de instruções **principal** e outros blocos de **rotinas**.
- documentação do programa: comentários.

Programa Exemplo: O arquivo `e0101.cpp` contém um programa para calcular a raiz quadrada de um número real positivo:

1.3.1 Conjunto de caracteres

Um programa fonte em C é um **texto não formatado** escrito em um editor de textos usando um o conjunto padrão de caracteres ASCII. A seguir estão os caracteres utilizados em C:

Caracteres válidos:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 0
+ - * / \ = | & ! ? # % ( ) { } [ ] _ ‘ “ . , : < >
```

Caracteres não válidos:

```
@ $ ¨ á é õ ç
```

Os caracteres acima são válidos apenas em strings. **Veja seção 2.1.4.**

1.3.2 Comentários

Em C, comentários podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo. Para que o comentário seja identificado como tal, ele deve ter um `/*` antes e um `*/` depois. Observe que no exemplo `e0101.cpp` todo o cabeçalho esta dentro de um comentário.

Exemplo:

```
/* esta e´ uma linha de comentário em C */
```

Observação: O C++ permite que comentários sejam escritos de outra forma: colocando um `//` em uma linha, o compilador entenderá que tudo que estiver a direita do símbolo é um comentário. Observe no programa exemplo `e0101.cpp` as linhas de comentários colocadas a direita dos comandos.

Exemplo:

```
// este e´ um comentário valido apenas em C++
```

1.3.3 Diretivas de Compilação

Em C, existem comandos que são processados durante a **compilação** do programa. Estes comandos são genericamente chamados de *diretivas de compilação*. Estes comandos informam ao compilador do C basicamente quais são as **constantes simbólicas** usadas no programa e quais **bibliotecas** devem ser anexadas ao programa executável. A diretiva `#include` diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente estes arquivos contem bibliotecas de funções ou rotinas do usuário. Voltaremos a trabalhas esta diretiva com mais detalhe no capítulo 5. A diretiva `#define` diz ao compilador quais são as constantes simbólicas usadas no programa. Veremos sobre esta diretiva no capitulo 2.

1.3.4 Declaração de variáveis

Em C, como na maioria das linguagens, as variáveis devem ser declaradas no inicio do programa. Estas variáveis podem ser de vários tipos: `int` (inteiro), `float` (real de simples precisão) e outras que serão vistas no capitulo 2. No exemplo acima `num`, `raiz`, `inf` e `sup` são declaradas como variáveis reais, enquanto `i` é declarada como uma variável inteira.

1.3.5 Entrada e saída de dados

Em C existem varias maneiras de fazer a leitura e escrita de informações. Estas operações são chamadas de operações de entrada e sadia. Veremos no capitulo 3 algumas funções de entrada e sadia de informações via teclado e tela. Outras funções de leitura e escrita em arquivos, sadia gráfica, funções de manipulação de *mouse*, entrada e sadia de informações via portas serial e paralela serão vistas em capítulos posteriores. No exemplo acima `printf` é uma função de escrita na tela, `scanf` é uma função de leitura de teclado.

1.3.6 Estruturas de controle

A linguagem C permite uma ampla variedade de estruturas de controle de fluxo de processamento. Estas estruturas serão vistas em detalhes nos capítulos 4 e 5. Duas estruturas das estruturas básicas (decisão e repetição) são muito semelhantes as estruturas usadas nas Pseudo-linguagem algorítmicas:

Estrutura de Decisão: Permite direcionar o fluxo lógico para dois blocos distintos de instruções conforme uma condição de controle.

```
Pseudo-linguagem                    Linguagem C

se condição                         if(condição){
  então bloco 1                       bloco 1;
  senão bloco 2                     }else{
fim se                                bloco 2;
                                    };
```

Estrutura de Repetição: Permite executar repetidamente um bloco de instruções ate que uma condição de controle seja satisfeita.

```
Pseudo-linguagem                    Linguagem C

faça                                do{
  bloco                               bloco;
até condição                         }while(condição);
```

2. Constantes e Variáveis

Neste capitulo veremos como os dados constantes e variáveis são manipulados pela linguagem C. O que são constantes inteiras, reais, caracteres e strings. Quais são as regras de atribuição de nomes a variáveis e quais são os tipos de dados que O C pode manipular. Veremos também como são declaradas as variáveis e as constantes simbólicas usadas em um programa.

2.1 Constantes

O C possui quatro tipos básicos de constantes: **inteiras**, de **ponto flutuante**, **caracteres** e ***strings***. Constantes inteiras e de ponto flutuante representam números de um modo geral. Caracteres e strings representam letras e agrupamentos de letras (palavras).

2.1.1 Constantes inteiras

Uma constante inteira é um número de valor inteiro. De uma forma geral, constantes inteiras são seqüências de dígitos que representam números inteiros. Números inteiros podem ser escritos no formato **decimal** (base 10), **hexadecimal** (base 16) ou **octal** (base 8).

Uma constante inteira **decimal** é formada por uma seqüência de dígitos decimais: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`. Se a constante tiver dois ou mais dígitos, o primeiro **não** pode ser `0`. Na verdade, pode ser `0` mas o compilador considerará esta constante como **octal** e não decimal.

Exemplo: A seguir são mostradas algumas constantes inteiras decimais válidas.

```
0   3   -45   26338   -7575   1010
```

Exemplo: Algumas constantes inteiras decimais inválidas.

```
1.          (ponto)
1,2         (vírgula)
045         (primeiro dígito é 0: não é constante decimal)
212-22-33   (caracter ilegal: -)
```

Uma constante inteira **hexadecimal** é formada por uma seqüência de dígitos decimais: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F` (ou `a, b, c, d, e`). Uma constante

hexadecimal **deve** começar por `0x`. Neste caso, os dígitos hexadecimais podem ser minúsculos ou maiúsculos.

Exemplo: A seguir são mostrados algumas constantes inteiras hexadecimais **válidas**.

```
0x0   0x3   0x4f5a   0x2FFE   OxABCD   0xAaFf
```

Exemplo: Algumas constantes inteiras hexadecimais **inválidas**.

```
0x3.       (ponto)
0x1,e      (vírgula)
0x ff      (espaço)
FFEE       (não começa com 0x: não é constante hexadecimal)
0Xfg34     (caracter ilegal: g)
```

Uma constante inteira **octal** é formada por uma seqüência de dígitos octais: `0, 1, 2, 3, 4, 5, 6, 7`. A constante octal **deve** ter o primeiro digito `0` para que o compilador a identifique como tal

Exemplo: A seguir são mostrados algumas constantes octais **válidas**.

```
00   -03   045   02633   07575   -0101
```

Exemplo: Algumas constantes inteiras octais **inválidas**.

```
010.       (ponto)
01,2       (vírgula)
0 4        (espaço)
45         (primeiro digito não é 0: não é constante octal)
01784      (caracter ilegal: 8)
```

2.1.2 Constantes de ponto flutuante

Números **reais** (não inteiros) são representados em base 10, por números com um ponto decimal e (opcionalmente) um expoente. Um número ponto flutuante **deve** ter um ponto decimal que não pode ser substituído por uma vírgula. Um número de ponto flutuante pode ser escrito em notação científica. Neste caso o `x10` é substituído por `e` ou `E`. O número `1.23e4` representa `1.23 x` 10^4 ou `12300`.

Exemplo: Números de ponto flutuante válidos.

```
0.234   125.65   .93   1.23e-9   -1.e2   10.6e18   -.853E+67
```

A forma de representação de um número real em C é bastante flexível.

Exemplo: O número 314 pode ser representado por qualquer uma das seguintes formas:

```
314.  3.14e2     +3.14e+2   31.4e1     .314E+3    314e0
```

2.1.3 Constantes caracteres

Uma constante caracter é uma letra ou símbolo colocado entre **aspas simples**.

Exemplo: Abaixo estão representados algumas constantes caracteres.

```
'a'   'b'   'X'   '&'   '{'   ' '
```

Embora sejam **visualizados** como letras e símbolos as constantes caracteres são armazenadas internamente pelo computador como um número **inteiro** entre 0 e 255. O caracter `'A'` por exemplo, tem valor `65`. Os valores numéricos dos caracteres estão padronizados em uma tabela chamada de *American Standard Code for Information Interchange Table* ou simplesmente *tabela ASCII*. **Veja apêndice B.**

Certos codigos de controle da tabela ASCII (como o *line feed*) ou caracteres especiaiss (como `'`) possuem representação especial no C. Esta representacao chama-se **seqüência de escape** representada por uma *barra invertida* (\) e um *caracter*. Sequencias de escape são interpretadas como caracteres simples. Abaixo segue uma lista das principais sequencias de escape usadas no C.

Controle/Caracter	**Sequencia de escape**	**Valor ASCII**
nulo (null)	\0	00
campainha (bell)	\a	07
retrocesso (backspace)	\b	08
tabulacao horizontal	\t	09
nova linha (new line)	\n	10
tabulacao vertical	\v	11
alimentacao de folha (form feed)	\f	12
retorno de carro (carriage return)	\r	13
aspas (")	\"	34
apostrofo (')	\'	39
interrogacao (?)	\?	63
barra invertida (\)	\\	92

2.1.4 Constantes strings

Uma constante string consiste de um conjunto de caracteres colocados entre **aspas duplas**. Embora as instruções do C usem apenas os caracteres do conjunto padrão ASCII, as constantes caracter e string podem conter caracteres do conjunto estendido ASCII: é, ã, ç, ü, ...

Exemplo: Abaixo seguem algumas constantes strings válidas.

```
"Oba!"
"Caxias do Sul"
"A resposta é: "
"João Carlos da Silveira"
"a"
"isto é uma string"
```

2.2 Identificadores

Identificadores são os **nomes** das variáveis e funções usadas no programa. Por exemplo `raiz` e `MAX` são nomes de variáveis utilizadas no programa `e0101.cpp`.

2.2.1 Regras de sintaxe

Os identificadores devem seguir as seguintes regras de construção:

- Os identificadores **devem** começar por uma letra (`a - z` , `A - Z`) ou um *underscore* (`_`).
- O resto do identificador deve conter apenas letras, *underscores* ou dígitos (`0 - 9`). **Não pode** conter outros caracteres. Em C, os identificadores podem ter até 32 caracteres.
- Em C, letras maiúsculas são **diferentes** de letras minúsculas: Por exemplo: `MAX`, `max`, `Max` são nomes diferentes para o compilador. Esta propriedade é chamada de *case sensibility*.

Exemplo: os nomes abaixo são válidos:

```
abc, y24, VetorPontosMovimentoRobo, nota_1, TAM_MAX.
```

Exemplo: os nomes abaixo não são válidos:

```
3dia, vetor-1, pao&leite, iteração.
```

2.2.2 Palavras reservadas

Existem certos nomes que **não podem** ser usados como identificadores. São chamadas as *palavras reservadas* e são de uso restrito da linguagem C (comandos, estruturas, declarações, etc.). O conjunto de palavras reservadas usadas em C é o seguinte:

`asm`	`auto`	`break`	`case`	`cdecl`	`char`
`class`	`const`	`continue`	`_cs`	`default`	`delete`
`do`	`double`	`_ds`	`else`	`enum`	`_es`
`extern`	`_export`	`far`	`_fastcall`	`float`	`for`
`friend`	`goto`	`huge`	`if`	`inline`	`int`
`interrupt`	`_loadds`	`long`	`near`	`new`	`operator`
`pascal`	`private`	`protected`	`public`	`register`	`return`
`_saveregs`	`_seg`	`short`	`signed`	`sizeof`	`_ss`
`static`	`struct`	`switch`	`template`	`this`	`typedef`
`union`	`unsigned`	`virtual`	`void`	`volatile`	`while`

Exemplo: Não é possível declarar estes conjunto de variáveis:

```
do, re, mi, fa, sol, la, si
old, new
```

Dos conjuntos acima, `do` e `new` são palavras reservadas.

2.3 Tipos de dados

Em C, como na maioria das linguagens, os dados são divididos tipos: inteiro, real, caracter, etc. Esta divisão se deve basicamente ao número de *bytes* reservados para cada dado. Cada tipo de dado possui um intervalo de valores permitidos.

2.3.1 Tipos básicos

Abaixo segue uma lista dos tipos básicos de dados permitidos em C. Os tipos `char` e `int` são inteiros e os tipos `float` e `double` são de ponto flutuante.

Tipo	Tamanho	Intervalo	Uso
char	1 byte	-128 a 127	número muito pequeno e caracter ASCII
int	2 bytes	-32768 a 32767	contador, controle de laço
float	4 bytes	3.4e-38 a 3.4e38	real (precisão de 7 dígitos)
double	8 bytes	1.7e-308 a 1.7e308	científico (precisão de 15 dígitos)

2.3.2 Declaração de variáveis

Para que se possa usar uma variável em um programa, **é necessário** fazer uma *declaração de variável* antes. A declaração de variáveis simplesmente informa ao processador quais são os nomes utilizados para armazenar dados variáveis e quais são os tipos usados. Deste modo o processador pode *alocar* (reservar) o espaço necessário na memória para a manipulação destas variáveis. É possível declarar mais de uma variável ao mesmo tempo, basta separá-las por vírgulas (`,`).

Sintaxe: A sintaxe para declaração de variáveis é a seguinte:

```
tipo variavel_1 [, variavel_2, ...] ;
```

Onde *`tipo`* é o tipo de dado e *`variavel_1`* é o nome da variável a ser declarada. Se houver mais de uma variável, seus nomes são separados por vírgulas.

Exemplo: Declaração das variáveis:

```
int i;
int x,y,z;
char letra;
float nota_1,nota_2,media;
double num;
```

No exemplo acima, `i`, `x`, `y` e `z` foram declaradas variáveis inteiras. Assim elas podem armazenar valores inteiros de -32768 até 32767. Do mesmo modo `letra` foi declarada como variável caracter podendo receber valores de -128 até 127 ou caracteres do conjunto padrão ASCII. As variáveis `nota_1`, `nota_2` e `media` foram declaradas como ponto flutuante tipo float e `num` como ponto flutuante tipo double.

A declaração de variáveis é feita, em geral, **dentro** de uma rotina. Por exemplo, a rotina principal `main()`. Deste modo se diz que está se fazendo uma declaração de variáveis *locais*. Variáveis locais podem ser referenciadas apenas dentro da rotina dentro da qual foi declarada, neste caso a rotina `main()`.

Exemplo: Observe o uso da declaração de variáveis no trecho de programa abaixo:

```
void main(){
  float raio, area;     // declaracao de variaveis
  raio = 2.5;
  área = 3.14 * raio * raio;
}
```

No exemplo acima, as variáveis `area` e `raio` foram declaradas como variáveis locais tipo `float`. Assim o processador faz a alocação de dois espaços (endereços) de 4 bytes cada para armazenar as informações, um para cada variável. Na terceira linha, o processador coloca no endereço alocado para `raio` o valor `2.5`. Depois, na quarta linha, o processador coloca o resultado da conta (`19.625`) no endereço de `área`.

É possível fazer a declaração de variáveis **fora** de uma rotina. Neste caso diz-se que se fez a declaração de variáveis *globais*. O uso de variáveis globais é explicado na sessão ?.?.

2.3.3 Tipos modificados

Além dos tipos de dados citados acima existem outros tipos de dados ditos **modificados**. Em C existem dois modificadores: o modificador `long` e o modificador `unsigned`. Tipicamente o modificador `long` aumenta o número de bytes usados para o registro do número. Por conseqüência o intervalo de validade do número fica aumentado significativamente. O modificador `unsigned`, usado somente em inteiros, permite que um **bit** usado para guardar o sinal do número seja usado para guardar o valor do número. Em conseqüência disto o intervalo do número fica dobrado, porém somente permite o uso de números positivos.

Tipo	**Tamanho (bytes)**	**Intervalo**
unsigned char	1	0 a 255
unsigned int	2	0 a 65 535
long int	4	-2 147 483 648 a 2 147 483 647
unsigned long int	4	0 a 4 294 967 295
long double	10	3.4e-4932 a 1.1e4932

2.3.4 Strings.

Uma string é um conjunto ordenado de caracteres que pode ser armazenado sob forma de um vetor um ponteiro. Estas estruturas de dados serão vistas em detalhe nos capitulo ? e ?. Por enquanto, nos basta saber como declarar e armazenar um conjunto caracter em uma variável.

Sintaxe: Para declararmos uma variável para receber um conjunto caracter devemos escrever:

```
char* var;
```

Exemplo: No exemplo seguinte a variável `nome` foi declarada como conjunto caracter e em seguida recebe uma constante string.

```
char* nome;
nome = "João Carlos de Oliveira Andrade";
```

2.3.5 Inicialização de variáveis.

Quando se faz a **declaração** de uma variável está se determinando que tipo de dado ela vai receber. É possível, em C, declarar uma variável e já armazenar nela um valor inicial. Chamamos este procedimento de **inicialização** de uma variável.

Sintaxe: A sintaxe para a inicialização de variáveis é:

```
tipo var_1 = valor_1 [, var_2 = valor_2, ...] ;
```

Onde *`tipo`* é o tipo de dado, *`var_1`* é o nome da variável a ser inicializada e *`valor_1`* é o valor inicial da variável.

Exemplo: Inicialização de variáveis:

```
int i = 0, j = 100;
float num = 13.5;
char* titulo = " Programa Teste ";
```

No exemplo acima, `i` e `j` foram declaradas variáveis tipo `int`. O valor inicial de `i` é `0` e o de `j` é `100`. Do mesmo modo `num` foi declarada como variável `float` com valor inicial de `13.5`. Tambem a variavel `titulo` foi declarada como um conjunto caracter e recebeu como conteudo inicial a string `" Programa Teste "`.

2.3.6 Conversão de tipo (*Casting*)

Algumas vezes queremos, momentaneamente, modificar o tipo de dado representado por uma variável, isto é, queremos que o dado seja apresentado em um tipo diferente do qual a variável foi inicialmente declarada. Por exemplo: declaramos uma variável como `int` e queremos, momentaneamente, que seu conteúdo seja apresentado como `float`. Este procedimento é chamado de **conversão de tipo** ou *casting* (moldagem, em inglês).

Sintaxe: A sintaxe da instrução de conversão de tipo é:

```
(tipo) variável
```

onde `tipo` é o nome do tipo ao qual queremos converter o dado armazenado em `variável`.

Exemplo: observe a conversão de tipo feita no exemplo abaixo:

```
int num;
float valor = 13.0;
num = (int)valor % 2;
```

No exemplo acima a variável `valor` foi declarada inicialmente como sendo do tipo `float` recebendo o valor inicial `13.0`. Logo em seguida o conteúdo de valor é convertido para o tipo `int` para realizar a operação módulo (%) com o inteiro 2. Aqui a conversão é necessária pois a operação módulo somente pode ser feita com inteiros. É importante salientar que a conversão de tipo é feita com o **dado** armazenado em uma variável mas a **variável** continua tendo o seu tipo original. No exemplo acima a variável `valor` e os dados nela armazenados continuam sendo do tipo `float` após a conversão.

Veremos na **seção 3.1** uma explicação mais detalhada do uso da conversão de tipos.

2.4 Constantes Simbólicas

Muitas vezes identificamos uma constante numérica por um símbolo: Pi = 3,14159 por exemplo. Podemos definir um nome simbólico para esta constante, isto é, podemos definir uma **constante simbólica** que represente valor.

2.4.1 Constantes definidas pelo programador

O programador pode definir constantes simbólicas em qualquer programa.

Sintaxe: A sintaxe da instrução de definição de uma constante simbólica é:

`#define` *`nome valor`*

Onde `#define` é uma diretiva de compilação que diz ao compilador para trocar as ocorrências do texto *`nome`* por *`valor`*. Observe que **não há `;`** no final da instrução pois trata-se de um comando para o compilador e não para o processador. A instrução `#define` **deve** ser escrita **antes** da instrução de declaração da rotina principal.

Exemplo: a seguir definimos algumas constantes simbólicas.

```
#define PI  3.14159
#define ON  1
#define OFF 0
#define ENDERECO 0x378
void main(){
...
```

No exemplo acima, definimos `PI` como `3.14159`. Isto significa que todas as ocorrências do texto `PI` será trocado por `3.14159`. Assim se escrevemos uma instrução:

```
área = PI * raio * raio;
```

o compilador vai interpretar esta instrução como se fosse escrita assim:

```
área = 3.14159 * raio * raio;
```

Poderíamos escrever estas instruções assim:

```
float pi = 3.14159;
área = pi * área * área;
```

porém este tipo de instrução tem duas **desvantagens**: Primeiro, reserva 4 bytes de memória desnecessariamente. Segundo, esta instrução é executada mais lentamente pois o processador precisa acessar a memória para verificar qual é o valor de `pi`.

Observe também que no exemplo definimos os nomes simbólicos com letras maiúsculas. Isto **não é necessário**, podemos perfeitamente definir nomes simbólicos usando letras minúsculas, porém faz parte do jargão dos programadores C usar letras maiúsculas para definir constantes simbólicas.

O uso da diretiva `#define` não se restringe apenas ao apresentado acima, podemos usá-la para definir **macro instruções**. Não veremos o uso de macros neste texto, procure mais detalhes na bibliografia recomendada.

2.4.2 Constantes pré-definidas

Em alguns compiladores C, algumas constantes simbólicas já estão pré-definidas. Estas constantes em geral definam alguns valores matemáticos (π, π/2, e, etc.), limites de tipos etc. A seguir segue uma tabela contendo algumas (existem muitas outras) constantes simbólicas pré-definidas no compilador Turbo C++ da Borland.

Biblioteca	Constante	Valor	Significado
math.h	M_PI	3.14159...	π
math.h	M_PI_2	1.57079...	π/2
math.h	M_PI_4	0,78539...	π/4
math.h	M_1_PI	0,31830...	1/π
math.h	M_SQRT2	1,41421...	√2
conio.h	BLACK	0	valor da cor (preto)
conio.h	BLUE	1	valor da cor (azul)
conio.h	GREEN	2	valor da cor (verde)
conio.h	CYAN	3	valor da cor (cyan)
conio.h	RED	4	valor da cor (vermelho)
conio.h	MAGENTA	5	valor da cor (magenta)
limits.h	INT_MAX	32767	limite superior do tipo int
limits.h	INT_MIN	-32768	limite inferior do tipo int

Cada uma das constantes acima esta definida em uma biblioteca. Uma biblioteca, em C, é um arquivo pré-compilado chamado arquivo *header* (cabeçalho, em inglês). Em cada biblioteca estão agrupadas constantes e funções semelhantes (veja seção 3.7.2). Por exemplo, constantes e funções matemáticas estão guardadas na biblioteca `math.h` (*mathematical functions*), constantes e funções de manipulação teclado e monitor estão guardadas na biblioteca `conio.h` (*console input and output*). Para

que se possa usar a constante simbólica em um programa é preciso **incluir** a biblioteca na compilação do programa.

Sintaxe: A sintaxe de inclusão de bibliotecas é a seguinte:

```
#include <nome_bib>
```

onde *nome_bib* é o nome da biblioteca que se deseja incluir. Esta instrução deve ser escrita antes do programa principal.

Exemplo: O programa abaixo usa a constante predefinida M_PI para calcular a área de um disco circular.

```
#include <math.h>
void main(){
  float area, raio = 5.0;
  área = M_PI * raio * raio;
}
```

3. Operadores, Expressões e Funções

Um programa tem como característica fundamental a capacidade de processar dados. Processar dados significa realizar operações com estes dados. As operações a serem realizadas com os dados podem ser determinadas por **operadores** ou **funções**. Os operadores podem ser de atribuição, aritméticos, de atribuição aritmética, incrementais, relacionais, lógicos e condicionais.

Exemplo: o símbolo `+` é um *operador* que representa a operação aritmética de adição. O identificador `sqrt()` é uma *função* que representa a operação de extrair a raiz quadrada de um número.

Uma **expressão** é um arranjo de operadores e operandos. A cada expressão válida é atribuído um valor numérico.

Exemplo: `4 + 6` é uma expressão cujo valor é `10`. A expressão `sqrt(9.0)` tem valor `3.0`.

3.1 Operador de Atribuição

A operação de atribuição é a operação mais simples do C. Consiste de atribuir valor de uma **expressão** a uma **variável**.

Sintaxe: A sintaxe da operação de atribuição é a seguinte:

identificador `=` *expressão*`;`

onde *`identificador`* é o nome de uma variável e *`expressão`* é uma expressão válida (ou outro identificador).

Exemplo: A seguir são mostradas algumas atribuições válidas:

```
a = 1;
delta = b * b - 4. * a * c;
i = j;
```

Observe que o símbolo de atribuição (`=`) **não tem** o mesmo significado que o usual da matemática que representa a igualdade de valores. Este símbolo, em C, representa a atribuição do valor calculado em *`expressão`* a variável *`identificador`*. Em pseudo-linguagem o operador de

atribuição é representado como ←. Também não se pode confundir o operador de atribuição (`=`) com o operador relacional de igualdade (`==`) que será visto na seção 3.5.1.

Observe-se também que o operando esquerdo **deve ser** um identificador de variável, isto é, **não pode** ser uma constante ou expressão.

Exemplo: A seguir são mostradas algumas atribuições inválidas:

```
1 = a;          // constante!
b + 1 = a;      // expressão!
```

3.1.1 Conversão de tipo.

Se os dois operandos de uma atribuição **não são** do mesmo tipo, o valor da expressão ou operador da direita **será convertido** para o tipo do identificador da esquerda.

Exemplo: Algumas atribuições com conversão de tipo:

```
int i;
float r;
i = 5;        // valor de i: 5
r = i ;       // valor de r: 5.0
```

A variável `i` foi inicializada com o valor `5`. Ao final da terceira instrução, `r` recebe o valor `5.0`.

Nestas conversões podem ocorrer alterações dos valores convertidos se o operando da esquerda for de um tipo que utilize **menor numero** de bytes que o operando da direita.

Exemplo: Algumas atribuições com conversão de tipo e perda de informação:

```
int i;
float r = 654.321;
i = r;                // truncamento!
```

Após a execução deste trecho de programa o valor da variável `i` será `654` pois seu valor foi truncado durante a conversão.

Pode-se dizer que as conversões potencialmente perigosas (onde há possibilidade de perda de informação) são:

`char ← int ← float ← double`

Observe que o compilador C ao encontrar esta operação **não gera** nenhum aviso de atenção para o programador. Assim este detalhe pode gerar um erro de programação (*bug*) que passe desapercebido ao programador inexperiente. É possível dizer que a linguagem C possui tipos “macios” (*soft types*) pois a operação com variáveis de tipos diferentes é perfeitamente possível. Esta característica do C se contrapõe a algumas linguagens em que isto não é possível (Fortran, por exemplo). Estas linguagens possuem tipos “duros” (*hard types*).

3.1.2 Limites do intervalo do tipo de dado.

Também é importante observar que os tipos em C tem intervalos bem definidos e os resultados das operações devem respeitar estes intervalos. Se a uma variável for atribuído um valor que esteja fora dos seus limites então este valor será alterado.

Exemplo: Observe as expressões abaixo, assuma que `i` seja uma variável do tipo `int`.

```
i = 4999;           // o valor de i e'   4999
i = 4999 + 1;       // o valor de i e'   5000
i = 5000 + 30000;   // o valor de i e' -30536
```

O valor de `35000` ultrapassou o limite superior do tipo `int` (`32767`).

É importante observar que em C, ao contrário de outras linguagens, a ultrapassagem do limite de um tipo **não é interpretado como erro.** Isto pode acarretar resultados inesperados para o programador desatento.

3.1.3 Atribuição múltipla.

E possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de **atribuição múltipla**.

Sintaxe: A sintaxe da atribuição múltipla é seguinte:

var_1 = [*var_2* = ...] *expressão*;

onde *var_1*, *var_2*, ... são os identificadores de variáveis e *expressão* é uma expressão válida.

Observe que na atribuição múltipla as operações ocorrem da **direita** para a **esquerda**, isto é, inicialmente o valor de *expressão* é atribuído a *var_2* e depois o valor de *var_2* é atribuído a *var_1*. Deve-se tomar cuidado com as conversões de tipo e limites de intervalo para atribuições de tipos diferentes.

Exemplo: Observe a instrução de atribuição múltipla abaixo: as variáveis inteiras `i, j` e `k` são todas inicializadas com o valor `1`. E as variáveis de dupla precisão `max` e `min` são inicializadas com o valor `0.0`:

```
int i, j, k;
double max, min;
i = j = k = 1;
max = min = 0.0;
```

Programa Exemplo: O arquivo `e0301.cpp` traz um programa para visualizar alguns aspectos relacionados com o operador de atribuição. Execute o programa passo-a-passo e observe o valor das variáveis.

3.2 Operadores Aritméticos

Existem cinco operadores aritméticos em C. Cada operador aritméticos está relacionado ao uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (`%`) chamado operador de **módulo** cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Operador	**Operação**
`+`	adição.
`-`	subtração.
`*`	multiplicação
`/`	divisão
`%`	módulo (resto da divisão inteira)

Sintaxe: A sintaxe de uma expressão aritmética é:

operando operador operando

onde *operador* é um dos **caracteres** mostrados acima e *operando* é uma **constante** ou um identificador de **variável**.

Exemplo: Algumas expressões aritméticas:

```
1+2   a-4.0   b*c   valor_1/taxa   num%2
```

Não existe em C, como existe em outras linguagens, um operador específico para a operação de potenciação (a^b). Existe, porém, uma função de biblioteca (**`pow()`**) que realiza esta operação. Veja a seção 3.7 adiante. Embora as operações do C sejam semelhantes as operações aritméticas usuais da matemática, alguns detalhes são específicos da linguagem, e devem ser observados.

3.2.1 Restrições de operandos.

Os operandos dos operadores aritméticos devem ser constantes numéricas ou identificadores de variáveis numéricas. Os operadores `+, -, *, /` podem operar números de todos os tipos (inteiros ou reais) porém o operador **`%`** somente aceita operandos **inteiros**.

Exemplo: Expressões válidas

Expressão	**Valor**
`6.4 + 2.1`	`8.5`
`7 - 2`	`5`
`2.0 * 2.0`	`4.0`
`6 / 3`	`2`
`47 % 2`	`1`

Uma restrição ao operador de divisão (`/`) é que o denominador **deve** ser diferente de zero. Se alguma operação de divisão por zero for realizada o correra um **erro de execução** do programa (*run-time error*), o programa será abortado e a mensagem `divide error` será exibida.

Exemplo: A expressão abaixo é inválida pois o primeiro operando não é um numero inteiro.

Expressão	**Valor**
`6.4 % 3`	`invalido!`

Podemos contornar o problema do operador inteiro da operação modulo usando o artifício da conversão de tipo (*casting*) mostrada na seção 2.3.4:

Exemplo: Observe o trecho de programa abaixo:

```
int num;
float valor = 13.0;
num = valor % 2;       // inválido!
num = (int)valor % 2;  // válido!
```

Observe que usamos a conversão de tipo para que o dado armazenado em `valor` fosse transformado no tipo `int` assim a operação módulo pode ser efetuada.

3.2.2 Conversão de tipo.

O resultado de uma operação aritmética **depende** dos tipos dos operandos. Se os operandos são do **mesmo** tipo o resultado será do **mesmo** tipo. Se os operando forem de tipos **diferentes** então haverá uma **conversão** de tipo tal que o tipo que ocupa **menos** espaço de memória será convertido para o tipo que ocupa **mais** espaço de memória e o resultado será deste tipo. Em geral:

`char` $\rightarrow$ `int` $\rightarrow$ `float` $\rightarrow$ `double`

Esta é uma regra geral, alguns compiladores podem ter outras regras de conversão.

Exemplo: Observe as conversões de tipo abaixo:

Expressão	**Valor**	**Conversão**
`6 + 2.0`	`8.0`	`int` $\rightarrow$ `float`
`7.000000 - 2.0000000000000`	`5.000000000000000`	`float` $\rightarrow$ `double`
`2 * 3.000000000000000`	`6.000000000000000`	`int` $\rightarrow$ `double`

Observe que estas conversões podem gerar resultados surpreendentes para o programador desatento.

Exemplo: Observe as expressões abaixo. Assuma que as variáveis `num_i, num_f, den_i` e `den_f` são inicializadas como:

```
int   num_i = 7  , den_i = 2  ;
float num_f = 7.0, den_f = 2.0;
```

Expressão	**Valor**	**Operandos**	**Resultado**
`num_f / den_f`	`3.5`	`float / float`	`float`
`num_f / den_i`	`3.5`	`float / int`	`float`
`num_i / den_f`	`3.5`	`int / float`	`float`
`num_i / den_i`	`3`	`int / int`	`int`

Observe que no exemplo acima o valor da última expressão é **`3`** e não **`3.5`**. Isto ocorre porque como os dois operandos são tipo `int` o resultado e convertido para o tipo `int` e ocorre o truncamento. O truncamento da divisão inteira é feito de modo a obter o menor valor absoluto.

Em C caracteres são armazenados na memória como números inteiros e por isso operações aritméticas são permitidas com tipos `char`. Os valores usados são os correspondentes da tabela ASCII.

Exemplo: Observe as expressões abaixo:

Expressão	**Valor**	**Conversão**
'A' + 1	'B'	65 + 1 → 66
'A' + 'B'	'â'	65 + 66 → 131
'A' + 32	'a'	65 + 32 → 97

3.2.4 Precedência de operadores.

Quando mais de um operador se encontram em uma expressão aritmética as operações são efetuadas uma de cada vez respeitando algumas regras de precedência: Estas regras de precedência são as mesmas da matemática elementar.

Os operadores de multiplicação (*), divisão (/) e módulo (%) tem precedência sobre os operadores de adição (+) e subtração (-). Entre operadores de mesma precedência as operações são efetuadas da **esquerda** para a **direita**. Veja a tabela 3.1.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	**Valor**	**Ordem**
1 + 2 - 3	0	+ -
24 - 3 * 5	9	* -
4 - 2 * 6 / 4 + 1	2	* / - +
6 / 2 + 11 % 3 * 4	11	/ % * +

A ordem de precedência dos operadores pode ser quebrada usando-se parênteses: ( ). Os parênteses são, na verdade, operadores de mais alta precedência e são executados primeiro. Parênteses internos são executados primeiro que parênteses externos.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	**Valor**	**Ordem**
1 + (2 - 3)	0	- +
(24 - 3) * 5	105	- *

```
(4 - 2 * 6) / 4 + 1      -1   * - / +
6 / ((2 + 11) % 3) * 4   24   + % / *
```

Observe que os operadores e os operandos deste exemplo são os mesmos do exemplo anterior. Os valores, porém, são diferentes pois a ordem de execução das operações foi modificada pelo uso dos parênteses.

Programa Exemplo: O arquivo `e0302.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores aritméticos. Execute o programa passo-a-passo e observe o valor das variáveis.

3.3 Operadores de Atribuição Aritmética

Muitas vezes queremos alterar o valor de uma variável realizando alguma operação aritmética com ela. Como por exemplo: `i = i + 1` ou `val = val * 2`. Embora seja perfeitamente possível escrever estas instruções, foi desenvolvido na linguagem C uma instruções **otimizadas** com o uso de operadores ditos **operadores de atribuição aritmética**. Os símbolos usado são (`+=, -=, *=, /= , %=`). Deste modo as instruções acima podem ser rescritas como: `i += 1` e `val *= 2`, respectivamente.

Sintaxe: A sintaxe da atribuição aritmética é a seguinte:

```
var += exp;
var -= exp;
var *= exp;
var /= exp;
var %= exp;
```

onde *var* é o identificador da variável e *exp* é uma expressão válida. Estas instruções são equivalentes as seguintes:

```
var = var + exp;
var = var - exp;
var = var * exp;
var = var / exp;
var = var % exp;
```

Exemplo: Observe as atribuições aritméticas abaixo e suas instruções equivalentes:

Atribuição aritmética	**Instrução equivalente**
`i += 1;`	`i = i + 1;`

```
j -= val;                    j = j - val;
num *= 1 + k;                num = num * (1 + k);
troco /= 10;                 troco = troco / 10;
resto %= 2;                  resto = resto % 2;
```

O operador de atribuição aritmética tem precedência menor que os outros operadores até aqui discutidos. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0303.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores de atribuição aritmética. Execute o programa passo-a-passo e observe o valor das variáveis.

3.4 Operadores Incrementais

Em programação existem instruções muito comuns chamadas de **incremento** e **decremento**. Uma instrução de incremento **adiciona** uma unidade ao conteúdo de uma variável. Uma instrução de decremento **subtrai** uma unidade do conteúdo de uma variável.

Existem, em C, operadores específicos para realizar as operações de incremento (`++`) e decremento (`--`). Eles são genericamente chamados de **operadores incrementais**.

Sintaxe: A sintaxe dos operadores incrementais é a seguinte:

	instrução equivalente
`++ var`	`var = var + 1`
`var ++`	`var = var + 1`
`-- var`	`var = var - 1`
`var --`	`var = var - 1`

onde *`var`* é o nome da variável da qual se quer incrementar ou decrementar um unidade.

Observe que existe duas sintaxes possíveis para os operadores: pode-se colocar o operador **à esquerda** ou **á direita** da variável. Nos dois casos o valor da variável será incrementado (ou decrementado) de uma unidade. Porém se o operador for colocado **á esquerda** da variável, o valor da variável será incrementado (ou decrementado) **antes** que a variável seja usada em alguma outra operação. Caso o operador seja colocado **à direita** da variável, o valor da variável será incrementado (ou decrementado) **depois** que a variável for usada em alguma outra operação.

Exemplo: Observe o fragmento de código abaixo e note o valor que as variáveis recebem **após** a execução da instrução:

```
                          valor das variáveis
int a, b, c, i = 3;       // a: ?   b: ?   c: ?   i: 3
a = i++;                  // a: 3   b: ?   c: ?   i: 4
b = ++i;                  // a: 3   b: 5   c: ?   i: 5
c = --i;                  // a: 3   b: 5   c: 4   i: 4
```

Os operadores incrementais são bastante usados para o controle de laços de repetição, que serão vistos na seção ??. É importante que se conheça exatamente o efeito sutil da colocação do operador, pois isto pode enganar o programador inexperiente.

Os operadores incrementais tem a mais alta precedência entre todos, sendo superados apenas pelos parênteses que tem precedência ainda maior. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0304.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores incrementais. Execute o programa passo-a-passo e observe o valor das variáveis.

3.5 Operadores Relacionais e Lógicos

A chave para a flexibilidade de um algoritmo é a tomada de decisões através da avaliação de condições de controle. Uma condições de controle é uma **expressão lógica** que é avaliadas como **verdadeira** ou **falsa**. Uma expressão lógica é construída com **operadores relacionais** e **lógicos**.

3.5.1 Operadores relacionais

Operadores relacionais verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais em C:

Operador	Significado
>	maior que
<	menor que
>=	maior ou igual a (não menor que)
<=	menor ou igual a (não maior que)
==	igual a
!=	não igual a (diferente de)

Sintaxe: A sintaxe das expressões lógicas é:

expressão_1 operador expressão_2

onde *expressão_1* e *expressão_2* são duas expressões numéricas quaisquer, e *operador* é um dos operadores relacionais.

Ao contrário de outras linguagens, em C **não existem** tipos lógicos, portanto o **resultado** de uma expressão lógica é um **valor numérico**: uma expressão avaliada **verdadeira** recebe o valor `1`, uma expressão lógica avaliada **falsa** recebe o valor `0`.

Se os operandos forem de **tipos diferentes** haverá uma conversão de tipo **antes** da avaliação da expressão. Esta conversão de tipo é feita de acordo com a regra mostrada na seção 3.2.2.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `i` e `j` são variáveis `int` cujos valores são `5` e `-3`, respectivamente. As variáveis `r` e `s` são `float` com valores `7.3` e `1.7`, respectivamente.

Expressão	Valor
`i == 7`	`0`
`r != s`	`1`
`i > r`	`0`
`6 >= i`	`1`
`i < j`	`0`
`s <= 5.9`	`1`

Os operadores relacionais de igualdade (`==` e `!=`) tem precedência **menor** que os de magnitude (`>`, `<`, `>=` e `<=`). Estes, por sua vez, tem precedência **menor** que os operadores aritméticos. Operadores relacionais de mesma precedência são avaliados da esquerda para a direita. Veja a tabela 3.1.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `m` e `n` são variáveis tipo `int` com valores `4` e `1`, respectivamente.

Expressão	Valor	Ordem de Operação
`m + n == 5`	`1`	`+ ==`
`m != 2 * n > m`	`1`	`* > !=`
`6 >= n < 3 - m`	`0`	`- >= <`
`m == n <= m > m`	`0`	`<= > !=`

3.5.2 Operadores lógicos

São três os operadores lógicos de C: `&&`, `||` e `!`. Estes operadores tem a mesma significação dos operadores lógicos Booleanos AND, OR e NOT.

Sintaxe: A sintaxe de uso dos operadores lógicos:

```
expr_1 && expr_2
expr_1 || expr_2
!expr
```

onde *`expr_1`*, *`expr_2`* e *`expr`* são expressões quaisquer.

Observe que os operadores lógicos atuam sobre expressões de quaisquer valores. Para estes operadores todo valor numérico diferente de `0` é considerado `1`.

Exemplo: A seguir é mostrado o valor lógico de uma expressão qualquer:

Expressão	***Valor lógico***
`0`	`0`
`1`	`1`
`1.0`	`1`
`0.4`	`1`
`-5.2`	`1`

onde *`expr_1`*, *`expr_2`* e *`expr`* são expressões quaisquer.

O resultado da operação lógica `&&` será `1` somente se os dois operandos forem `1`, caso contrário o resultado é `0`. O resultado da operação lógica `||` será `0` somente se os dois operandos forem `0`, caso contrário o resultado é `1`. O resultado da operação lógica `!` será `0` se o operandos for `1`, e `1` se o operando for `0`. Abaixo mostra-se o resultado das possíveis combinações entre os operandos para cada operador lógico:

Operador **&&**:	**op_1**	**op_2**	**Res**
op_1 && op_2	1	1	1
	1	0	0
	0	1	0
	0	0	0

Operador **||**:	**op_1**	**op_2**	**Res**
op_1 || op_2	1	1	1

1	0	1
0	1	1
0	0	0

Operador !:	op	Res
!op	1	0
	0	1

O Operador && tem precedência sobre o operador ||. Estes dois têm precedência menor que os operadores relacionais. O operador ! tem a mesma precedência que os operadores incrementais. Veja a tabela 3.1.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que a, b e c são variáveis tipo int com valores 0, 1 e 2, respectivamente.

Expressão	Valor	Ordem de Operação
a && b	0	
c > b || a < c	1	> < ||
a + b && !c - b	1	! + - &&
!b && c || a	0	! && ||

Programa Exemplo: O arquivo e0305.cpp traz um programa para visualizar alguns aspectos relacionados com os operadores lógicos e relacionais. Execute o programa passo-a-passo e observe o valor das variáveis.

3.6 Operador Condicional

O operador condicional (?:) é usado em expressões condicionais. Uma expressão condicional pode ter dois valores diferentes dependendo de uma condição de controle.

Sintaxe: A sintaxe de uma expressão condicional é:

condição ? *expressão_1* : *expressão_2*

onde *expressão_1* e *expressão_2* são duas expressões quaisquer, e *condição* é uma expressão lógica que será avaliada primeiro. Se o valor de *condição* for 1, isto é, verdadeiro, então a expressão condicional assumirá o valor de *expressão_1*. Caso contrario assumirá o valor de *expressão_2*. Uma expressão condicional é equivalente a uma estrutura de decisão simples:

```
se condição
  então expressao_1
  senão expressao_2
fim se
```

Exemplo: Observe as expressões condicionais abaixo e verifique o resultado de sua avaliação. Admita que `i`, `j` e `k` são variáveis tipo `int` com valores `1`, `2` e `3`, respectivamente.

Expressão	Valor
`i ? j : k`	2
`j > i ? ++k : --k`	4
`k == i && k != j ? i + j : i - j`	-1
`i > k ? i : k`	3

O operador condicional tem baixa precedência, precedendo, apenas, aos operadores de atribuição. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0306.cpp` traz um programa para visualizar alguns aspectos relacionados com o operador condicional. Execute o programa passo-a-passo e observe o valor das variáveis.

3.7 Funções de biblioteca

Uma função é um **sub-programa** (também chamado de rotina). Esta função *recebe* informações, *as processa* e *retorna* outra informação. Por exemplo, podemos ter uma função que receba um valor numérico, calcule seu logaritmo decimal e retorne o valor obtido. Existem dois tipos de funções: *funções de biblioteca* e *funções de usuário*. Funções de biblioteca são funções escritas pelos fabricantes do compilador e já estão pré-compiladas, isto é, já estão escritas em código de máquina. Funções de usuário são funções escritas pelo programador. Nesta seção trataremos somente das funções de biblioteca, funções de usuário serão vistas no capítulo ?.

3.7.1 O uso de funções

Antes de usar uma função é preciso saber como a função esta declarada, isto é, quais são os parâmetros que a função recebe e quais são os para metros que a função retorna. Estas informações estão contidas no manual do usuário do compilador ou em sua documentação *on-line*.

Sintaxe: A sintaxe de declaração de uma função é:

```
tipo_ret nome(tipo_1, tipo_2, ...)
```

onde `nome` é o nome da função, `tipo_1`, `tipo_2`, ... são os tipos (e quantidade) de parâmetros de entrada da função e `tipo_ret` é o tipo de dado de retorno da função. Além dos tipos usuais vistos na seção 2.3, existe ainda o tipo `void` (vazio, em inglês) que significa que aquele parâmetro é inexistente.

Exemplo: A função `cos()` da biblioteca `math.h` é declarada como:

```
double cos(double);
```

Isto significa que a função tem um parâmetro de entrada e um parâmetro de saída, ambos são do tipo `double`.

Exemplo: A função `getch()` da biblioteca `conio.h` é declarada como:

```
int getch(void);
```

Isto significa que a função não tem parâmetros de entrada e tem um parâmetro `int` de saída.

Para podermos usar um função de biblioteca devemos **incluir** a biblioteca na compilação do programa. Esta inclusão é feita com o uso da diretiva `#include` colocada antes do programa principal, como visto na secao 2.4.2.

Exemplo: Assim podemos usar a função no seguinte trecho de programa:

```
#include <math.h>            // inclusão de biblioteca
void main(){                 // inicio do programa principal
  double h = 5.0;            // hipotenusa
  double co;                 // cateto oposto
  double alfa = M_PI_4;      // angulo: π/4
  co = h * cos(alfa);        // calculo: uso da funcao cos()
}                            // fim do programa
```

As funções tem alta precedência, sendo mais baixa apenas que os parênteses. A tabela 3.1 mostra as precedências de todos os operadores estudados neste capítulo.

3.7.2 As bibliotecas disponíveis e algumas funções interessantes

A seguir segue uma lista de todas as bibliotecas disponíveis no compilador *Turbo C++ 3.0 Borland*: Ao longo do texto veremos o uso de muitas funções cobrindo uma boa parte destas bibliotecas, porém o leitor que desejar tornar-se "fluente" na linguagem C pode (e deve) estudá-las com profundidade.

```
alloc.h      assert.h     bcd.h        bios.h       complex.h
conio.h      ctype.h      dir.h        dirent.h     dos.h
errno.h      fcntl.h      float.h      fstream.h    generic.h
graphics.h   io.h         iomanip.h    iostream.h   limits.h
locale.h     malloc.h     math.h       mem.h        process.h
setjmp.h     share.h      signal.h     stdarg.h     stddef.h
stdio.h      stdiostr.h   stdlib.h     stream.h     string.h
strstrea.h   sys\stat.h   sys\timeb.h  sys\types.h  time.h
values.h
```

Vejamos algumas funcoes disponiveis nas bibliotecas C.

Biblioteca math.h

```
int abs(int i);
double fabs(double d);
```

Calcula o valor absoluto do inteiro `i` e do real `d`, respectivamente.

```
double sin(double arco);
double cos(double arco);
double tan(double arco);
double asin(double arco);
double acos(double arco);
double atan(double arco);
```

Funções trigonometricas do ângulo `arco`, em radianos.

```
double ceil(double num);
double floor(double num);
```

Funcoes de arredondamento para inteiro.

`ceil()` arredonda para cima. Ex. `ceil(3.2) == 3.0;`

`floor()` arredonda para baixo. Ex. `floor(3.2) == 4.0;`

```
double log(double num);
double log10(double num);
```

Funcoes logaritmicas: `log()` é logaritmo natural (base *e*), `log10()` é logaritmo decimal (base 10).

```
double pow(double base, double exp);
```

Potenciacao: `pow(3.2,5.6)` = $3.2^{5.6}$.

```
double sqrt(double num);
```

Raiz quadrada: `sqrt(9.0) = 3.0.`

Biblioteca stdlib.h

```
int random(int num);
```

Gera um número inteiro aleatório entre `0` e *num* `- 1`.

Programa Exemplo: O arquivo `e0307.cpp` traz um programa para visualizar alguns aspectos relacionados com funções de biblioteca. Execute o programa passo-a-passo e observe o valor das variáveis.

3.8 Precedência entre os operadores do C

A tabela 3.1 mostra a ordem de precedência de todos os operadores estudados neste capítulo. Os operadores de maior precedência são os **parênteses** e as chamadas de **funções**. Os operadores de menor precedência são os o operadores de **atribuição**.

Categoria	Operadores	Prioridade
parênteses	`( )`	interno → externo
função	`nome()`	E → D
incremental, lógico	`++ -- !`	E ← D
aritmético	`* / %`	E → D
aritmético	`+ -`	E → D

relacional	`< > <= >=`	E → D
relacional	`== !=`	E → D
lógico	`&&`	E → D
lógico	`||`	E → D
condicional	`?:`	E ← D
atribuição	`= += -= *= /= %=`	E ← D

***Tabela 3.1**: Precedência dos operadores. Maior precedência no topo, menor precedência na base.*

4. Entrada e Saída

Para que um programa torne-se minimamente funcional é preciso que ele receba dados do meio externo (teclado, mouse, portas de comunicação, drives de disco, etc.) e emita o resultado de seu processamento de volta para o meio externo (monitor, impressora, alto-falante, portas de comunicação, drives de disco, etc.). De outro modo: um programa deve trocar informações com o meio externo. Em C, existem muitas funções pré-definidas que tratam desta troca de informações. São as funções de **entrada e saída** do C. Nos exemplos mostrados nos capítulos anteriores foram vistas algumas funções de entrada (`scanf(), getch()`) e algumas funções de saída (`printf()`). Neste capítulo veremos, em detalhe, estas e outras funções de modo a permitir escrever um programa completo em C.

Mostraremos, nas duas seções iniciais as mais importantes funções de entrada e saída de dados em C: as funções `printf()` e `scanf()`. A partir do estudo destas funções é possível escrever um programa propriamente dito com entrada, processamento e saída de dados.

4.1 Saída formatada: `printf()`

Biblioteca: `stdio.h`

Declaração: `int printf (const char* st_contr [, lista_arg]);`

Propósito: A função `printf()` (*print formated*) imprime dados da lista de argumentos `lista_arg` formatados de acordo com a string de controle `st_contr`. Esta função retorna um valor inteiro representando o número de caracteres impressos.

Esta função imprime dados numéricos, caracteres e *strings*. Esta função é dita de saída formatada pois os dados de saída podem ser formatados (alinhados, com número de dígitos variáveis, etc.).

Sintaxe: A *string de controle* é uma máscara que especifica (formata) o que será impresso e de que maneira será impresso.

Exemplo: Observe no exemplo abaixo as *instruções* de saída formatada e os respectivos resultados.

Instrução	**Saída**
`printf(“Ola’, Mundo!“);`	`Ola’, Mundo!`
`printf(“linha 1 \nlinha 2 “);`	`linha 1` `linha 2`

Observe que na primeira instrução, a saída é exatamente igual a string de controle. Já na segunda instrução a impressão se deu em duas linhas. Isto se deve ao `\n` que representa o código ASCII para quebra de linha (veja seção 2.1.3).

Nesta mascara é possível reservar espaço para o **valor** de alguma variável usando *especificadores de formato*. Um especificador de formato marca o **lugar** e o **formato** de impressão das variáveis contidas na **lista variáveis**. Deve haver um especificador de formato para cada variável a ser impressa. Todos os especificadores de formato começam com um %.

Exemplo: Observe no exemplo abaixo as *instruções* de saída formatada e os respectivos resultados. Admita que `idade` seja uma variável `int` contendo o valor `29` e que `tot` e `din` sejam variáveis `float` cujo valores são, respectivamente, `12.3` e `15.0`.

Instrução:
```
printf(“Tenho %d anos de vida”,idade);
```
Saída:
```
Tenho 29 anos de vida
```

Instrução:
```
printf(“Total: %f.2 \nDinheiro: %f.2 \nTroco: %f.2“,tot,din,din-tot);
```
Saída:
```
Total: 12.30
Dinheiro: 15.00
Troco: 2.70
```

Depois do sinal %, seguem-se alguns modificadores, cuja sintaxe é a seguinte:

`% [flag] [tamanho] [.precisão] tipo`

[flag]	**justificação de saída: (Opcional)**
-	justificação à esquerda.
+	conversão de sinal (saída sempre com sinal: + ou -)
<espaço>	conversão de sinal (saídas negativas com sinal, positivas sem sinal)

[tamanho]	**especificação de tamanho (Opcional)**
n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por brancos).
0n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por zeros).

[.precisão]	**especificador de precisão, dígitos a direita do ponto decimal. (Opcional)**
(nada)	padrão: 6 dígitos para reais.
.0	nenhum digito decimal.
.n	são impressos n dígitos decimais.

Tipo	**caracter de conversão de tipo (Requerido)**
d	inteiro decimal
o	inteiro octal
x	inteiro hexadecimal
f	ponto flutuante: [-]dddd.dddd.
e	ponto flutuante com expoente: [-]d.dddde[+/-]ddd
c	caracter simples
s	string

Programa Exemplo: O arquivo `e0401.cpp` contém um programa que ilustra o uso da função `printf()` usando várias combinações de *strings de controle* e *especificadores de formato*. Execute o programa passo-a-passo e verifique a saída dos dados.

4.2 Leitura formatada: `scanf()`

Biblioteca: `stdio.h`

Declaração: `int scanf(const char* st_contr [, end_var, ...]);`

Propósito: A função `scanf()` (*scan formated*) permite a entrada de dados numéricos, caracteres e 'strings' e sua respectiva atribuição a variáveis cujos endereços são *`end_var`*. Esta função é

dita de entrada formatada pois os dados de entrada são formatados pela *string de controle* `st_contr`. a um determinado tipo de variável (`int`, `float`, `char`, ...).

Sintaxe: O uso da função `scanf()` é semelhante ao da função `printf()`. A função lê da entrada padrão (em geral, teclado) uma lista de valores que serão formatados pela string de controle e armazenados nos endereços das variáveis da lista. A string de controle é formada por um conjunto de especificadores de formato, cuja sintaxe é a seguinte:

`% [*] [`*`tamanho`*`] `*`tipo`*

`*`	**indicador de supressão (Opcional)**
`<presente>`	Se o indicador de supressão estiver presente o campo não é lido. Este supressor é útil quando não queremos ler um campo de dado armazenado em arquivo.
`<ausente>`	O campo é lido normalmente.

`Tamanho`	**especificador de tamanho(Opcional)**
`n`	Especifica `n` como o numero máximo de caracteres para leitura do campo.
`<ausente>`	Campo de qualquer tamanho.

`Tipo`	**define o tipo de dado a ser lido (Requerido)**
`d`	inteiro decimal (`int`)
`f`	ponto flutuante (`float`)
`o`	inteiro octal (`int`)
`x`	inteiro hexadecimal (`int`)
`i`	inteiro decimal de qualquer formato(`int`)
`u`	inteiro decimal sem sinal (`unsigned int`)
`s`	string (`char*`)
`c`	caracter (`char`)

A lista de variáveis é o conjunto de (endereços) de variáveis para os quais serão passados os dados lidos. Variáveis simples devem ser precedidos pelo caracter `&.` Veja mais sobre endereços na seção ?.? Variáveis string e vetores não são precedidos pelo caracter `&.` Veja mais sobre strings e vetores na seção ?.?

Programa exemplo: O arquivo `e0402.cpp` contém um programa que ilustra o uso da função `scanf()` na leitura de dados. Execute o programa passo-a-passo e verifique como os especificadores de formato agem sobre os dados digitados.

4.3 Entrada de caracter individual: `getchar()`

Biblioteca: `stdio.h`
Declaração: `int getchar(void);`

Propósito: A função `getchar()` (*get character*) lê um caracter individual da entrada padrão (em geral, o teclado). Se ocorrer um erro ou uma condição de '*fim-de-arquivo*' durante a leitura, a função retorna o valor da constante simbólica `EOF` (*end of file*)definida na biblioteca `stdio.h`. Esta função permite uma forma eficiente de detecção de finais de arquivos.

Esta função é dita *line buffered*, isto é, não retorna valores até que o caracter de controle *line feed* (`\n`) seja lido. Este caracter, normalmente, é enviado pelo teclado quando a tecla [*enter*] é pressionada. Se forem digitados vários caracteres, estes ficarão armazenados no *buffer* de entrada até que a tecla [*enter*] seja pressionada. Então, cada chamada da função `getchar()` lerá um caracter armazenado no *buffer*.

4.4 Saída de caracter individual: `putchar()`

Biblioteca: `stdio.h`
Declaração: `int putchar(int c);`

Propósito: Esta função `putchar()` (*put character*) imprime um caracter individual *c* na saída padrão (em geral o monitor de vídeo).

Programa Exemplo: O programa `e0403.cpp` mostra o uso das funções `getchar()` e `putchar()` em um programa que lê caracteres do teclado e os reimprime convertidos para maiúsculos.

4.5 Leitura de teclado: `getch(), getche()`

Biblioteca: `conio.h`
Declaração:
```
int getch(void);
int getche(void);
```

Propósito: Estas funções fazem a leitura dos códigos de teclado. Estes códigos podem representar tecla s de caracteres (`A`, `y`, `*`, `8`, etc.), teclas de comandos (`[enter]`, `[delete]`, `[Page Up]`, `[F1]`, etc.) ou combinação de teclas (`[Alt]` + `[A]`, `[Shift]` + `[F1]`, `[Ctrl]` + `[Page Down]`, etc.).

Ao ser executada, a função `getch()` (*get character*) aguarda que uma tecla (ou combinação de teclas) seja pressionada, recebe do teclado o código correspondente e retorna este valor. A função `getche()`(*get character and echoe*) também escreve na tela, quando possível, o caracter correspondente.

Código ASCII: ao ser pressionada uma tecla correspondente a um caracter ASCII, o teclado envia um código ao '*buffer*' de entrada do computador e este código é lido. Por exemplo, se a tecla `A` for pressionada o código `65` será armazenado no *buffer* e lido pela função.

Código Especial: ao serem pressionadas certas teclas (ou combinação de teclas) que não correspondem a um caracter ASCII, o teclado envia ao '*buffer*' do computador **dois** códigos, sendo o primeiro **sempre** 0. Por exemplo, se a tecla `[F1]` for pressionada os valores `0` e `59` serão armazenados e a função deve ser chamada **duas vezes** para ler os dois códigos.

Programa exemplo: O arquivo `e0404.cpp` mostra um programa para a leitura de teclado. Este programa usa a função `getch()` para reconhecer teclas e combinação de teclas.

Programa exemplo: O arquivo `e0405.cpp` mostra um programa para a leitura de teclado usando a função `getche()`.

4.6 Escrita formatada em cores: `cprintf()`

Biblioteca: `conio.h`
Declaração: `int cprintf (const char* st_contr [, lista_arg]);`

Propósito: Esta função `cprintf()` (*color print formated*) permite a saída de dados numéricos, caracteres e strings usando cores. O uso da função `cprintf()`é semelhante a `printf()`porém permite que a saída seja a cores. Para que a saída seja colorida é necessário definir as cores de fundo e de letra para a impressão antes do uso da função.

Cores (Modo Texto)

Cor	Constante	Valor	Fundo	Letra
Preto	BLACK	0	ok	ok

Azul	BLUE	1	ok	ok
Verde	GREEN	2	ok	ok
Cian	CYAN	3	ok	ok
Vermelho	RED	4	ok	ok
Magenta	MAGENTA	5	ok	ok
Marrom	BROWN	6	ok	ok
Cinza Claro	LIGHTGRAY	7	ok	ok
Cinza Escuro	DARKGRAY	8	--	ok
Azul Claro	LIGHTBLUE	9	--	ok
Verde Claro	LIGHTGREEN	10	--	ok
Cian Claro	LIGHTCYAN	11	--	ok
Vermelho Claro	LIGHTRED	12	--	ok
Magenta Claro	LIGHTMAGENTA	13	--	ok
Amarelo	YELLOW	14	--	ok
Branco	WHITE	15	--	ok
Piscante	BLINK	128	--	ok

Estas definições são feitas pelas funções `texcolor()` e `textbackground()` cuja sintaxe é:

```
textcolor(cor_de_letra);
textbackground(cor_de_fundo);
```

onde *`cor_de_letra`* e *`cor_de_fundo`* são números inteiros referentes as cores da palheta padrão (16 cores, modo texto). Estes valores de cor são representadas por constantes simbólicas definidas na biblioteca `conio.h`. Para se usar uma letra piscante deve-se adicionar o valor `128` ao valor da cor de letra. Alguns valores de cor não podem ser usados como cor de fundo. A relação acima mostra as cores, suas constantes simbólicas e onde podem ser usadas:

Exemplo: O trecho de programa abaixo imprime uma mensagem de alerta em amarelo piscante sobre fundo vermelho.

```
#include <conio.h>
...
textbackground(RED);
textcolor(YELLOW + BLINK);
cprintif(“ Alerta: Vírus Detectado! ”);
...
```

Programa Exemplo: O programa do arquivo `e0406.cpp` mostra todas as combinações possíveis de impressão colorida em modo texto.

4.7 Saída sonora: `sound(), delay(), nosound()`

Biblioteca: `dos.h`

Declarações:
```
void sound(unsigned freq);
void delay(unsigned tempo);
void nosound(void);
```

Propósito: A função `sound()` ativa o alto-falante do PC com uma freqüência *`freq`* (Hz). A função `delay()` realiza uma pausa (aguarda intervalo de tempo) de duração *`tempo`* (milisegundos). A função `nosound()` desativa o alto-falante.

Programa Exemplo: O uso destas funções é muito simples mas produz resultados interessantes. No arquivo `e0407.cpp` temos um exemplo do uso de sons em programas.

4.8 Limpeza de tela: `clrscr(), clreol()`

Biblioteca: `conio.h`

Declarações:
```
void clrscr(void);
void clreol(void);
```

Propósito: A função `clrscr()` (*clear screen*) limpa a janela de tela e posiciona o cursor na primeira linha e primeira coluna da janela (canto superior esquerdo da janela). A função `clreol()` (*clear to end of line*) limpa uma linha desde a posição do cursor até o final da linha mas não modifica a posição do cursor. Ambas funções preenchem a tela com a cor de fundo definida pela função `textbacground()`.

4.9 Posicionamento do cursor: `gotoxy()`

Biblioteca: `conio.h`

Declarações: `void gotoxy(int pos_x, int pos_y);`

Propósito: Em modo texto padrão, a tela é dividida em uma janela de 25 linhas e 80 colunas. A função `gotoxy()` permite posicionarmos o cursor em qualquer posição `(pos_x,pos_y)` da tela. Sendo que a posição (1,1) corresponde ao canto superior esquerdo da tela e a posição (80,25) corresponde ao canto inferior direito. Como as funções `printf()` e `cprintf()` escrevem a partir da posição do cursor, podemos escrever em qualquer posição da tela.

4.10 Redimencionamento de janela: `window()`

Biblioteca: `conio.h`
Declarações: `void window(int esq, int sup, int dir, int inf);`

Propósito: Esta função permite redefinir a janela de texto. As coordenadas `esq` e `sup` definem o canto superior esquerdo da nova janela, enquanto as coordenadas `inf` e `dir` definem o canto inferior direito da nova janela. Para reativar a janela padrão escreve-se a instrução `window(1,1,80,25)`. Quando uma janela é definida, o texto que ficar fora da janela fica congelado até que se redefina a janela original.

4.11 Monitoração de teclado: `kbhit()`

Biblioteca: `conio.h`
Declarações: `int kbhit(void);`

Propósito: Esta função (*keyboard hitting*) permite verificar se uma tecla foi pressionada ou não. Esta função verifica se existe algum código no *buffer* de teclado. Se houver algum valor, ela retorna um número não nulo e o valor armazenado no *buffer* pode ser lido com as funções `getch()` ou `getche()`. Caso nenhuma tecla seja pressionada a função retorna 0. Observe que, ao contrário de `getch()`, esta função **não aguarda** que uma tecla seja pressionada.

Programa Exemplo: O arquivo e0408.cpp contém um programa para exemplificar o uso das funções `clrscr(), clreol(), gotoxy(), window(), kbhit()`.

5. Estruturas de Controle

Estruturas de controle permitem controlar a seqüência das ações lógicas de um programa. Basicamente, existem dois tipos de estruturas de controle: estruturas de **repetição** e estruturas de **decisão**. A estrutura de repetição permite que um bloco de instruções seja executado repetidamente uma quantidade controlada de vezes. A estrutura de decisão permite executar um entre dois ou mais blocos de instruções. Neste capítulo estudaremos em detalhe as instruções do C que permitem implementar estas estruturas.

5.1 Condição de controle

Em todas as estruturas, existe pelo menos uma expressão que faz o controle de **qual** bloco de instruções será executado ou **quantas vezes** ele será executado: é o que chamamos de **condição de controle**. Uma condição de controle é uma expressão lógica ou aritmética cujo resultado pode ser considerado verdadeiro ou falso. Conforme vimos na seção 3.5, a linguagem C não possui, entretanto, variáveis ou constantes lógicas, possui somente expressões numéricas, assim quando uma **expressão numérica** se encontra em uma **condição de controle**, ela será considerada **falsa** se seu valor for **igual a zero**, e **verdadeira** se seu valor for **diferente de zero**.

Exemplo: Observe nas condições abaixo, seu valor numérico e seu significado lógico. Considere as variáveis `int i = 0, j = 3;`

condição	**valor numérico**	**significado lógico**
`(i == 0)`	`1`	`verdadeiro`
`(i > j)`	`0`	`falso`
`(i)`	`0`	`falso`
`(j)`	`3`	`verdadeiro`

Este fato deve ficar claro pois, nas estruturas que estudaremos neste capítulo, quando for dito que uma condição é **falsa** ou **verdadeira** quer se dizer que seu valor e **igual** a zero ou **diferente** de zero.

5.2 Estrutura `do...while`

Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. Sua sintaxe é a seguinte:

Sintaxe:

```
do{
  bloco
}while(condição);
```

onde: *condição* é uma expressão lógica ou numérica.

bloco é um conjunto de instruções.

Esta estrutura faz com que o bloco de instruções seja executado pelo menos uma vez. Após a execução do bloco, a condição é avaliada. Se a condição é **verdadeira** o bloco é executado outra vez, caso contrário a repetição é terminada. Ofluxograma desta estrutura é mostrada na figura 5.1:

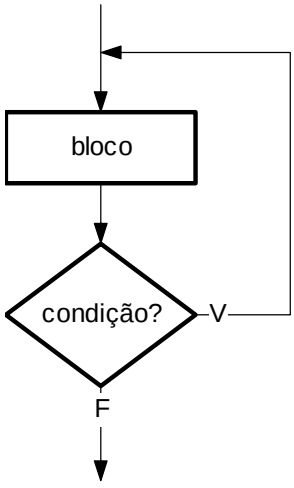

Figura 5.1: *Fluxograma da estrutura `do...while`.*

Exemplo: No trecho abaixo, a leitura de um número é feita dentro de um laço de repetição condicional. A leitura é repetida caso o número lido seja negativo.

```
do{
  puts("Digite um número positivo:");
  scanf("%f",&num);
}while(num <= 0.0);
```

Programa exemplo: No arquivo `e0501.cpp` existe um programa que calcula o fatorial de um número. Este programa ilustra o uso da estrutura `do...while`.

5.3 Estrutura `while`

A estrutura de repetição condicional `while` é semelhante a estrutura `do...while`. Sua sintaxe é a seguinte:

Sintaxe:

```
while(condição){
  bloco
}
```

onde: *`condição`* é uma expressão lógica ou numérica.

`bloco` é um conjunto de instruções.

Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição é **verdadeira** o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja **falsa** a repetição é terminada sem a execução do bloco. Observe que nesta estrutura, ao contrário da estrutura `do...while`, o bloco de instruções pode não ser executado nenhuma vez, basta que a condição seja inicialmente falsa. O fluxograma desta estrutura é mostrada na figura 5.2:

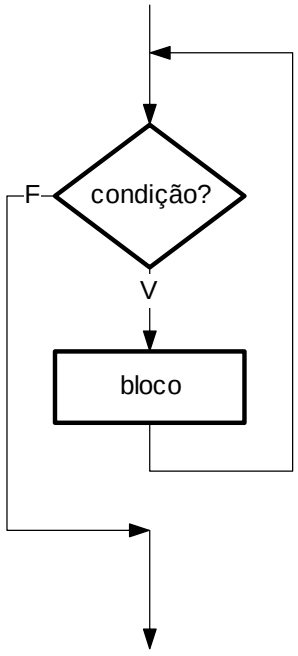

Figura 5.2: *Fluxograma da estrutura `while`.*

Exemplo: No trecho abaixo, calcula-se a precisão (ε) do processador aritmético do PC. A variável `eps` tem seu valor dividido por 2 enquanto o processador conseguir distinguir entre 1 e $1+\varepsilon$. Após a execução do laço, o valor de `eps` contém a precisão da máquina.

```
eps = 1.0;
while(1.0 + eps > 1.0){
  eps /= 2.0;
}
```

Programa exemplo: No arquivo `e0502.cpp` existe um programa que calcula a raiz quadrada de um número real positivo usando o método de Newton. Este programa ilustra o uso da estrutura `while`.

5.4 Estrutura `for`

A estrutura `for` é muito semelhante as estruturas de repetição vistas anteriormente, entretanto costuma ser utilizada quando se quer um número determinado de ciclos. A contagem dos ciclos é feita por uma variável chamada de **contador**. A estrutura `for` é, as vezes, chamada de estrutura de **repetição com contador**. Sua sintaxe é a seguinte:

Sintaxe:

```
for(inicialização; condição; incremento){
  bloco
}
```

onde: *`inicialização`* é uma expressão de inicialização do contador.
`condição` é uma expressão lógica de controle de repetição.
`incremento` é uma expressão de incremento do contador.
`bloco` é um conjunto de instruções a ser executado.

Esta estrutura executa um número determinado de repetições usando um contador de iterações. O contador é inicializado na expressão de *`inicialização`* **antes** da primeira iteração. Por exemplo: `i = 0;` ou `cont = 20;`. Então o bloco é executado e **depois** de cada iteração, o contador é incrementado de acordo com a expressão de *`incremento`*. Por exemplo: `i++` ou `cont -= 2`. Então a expressão de condição é avaliada: se a condição for verdadeira, o *`bloco`* é executado novamente e o ciclo recomeça, se a condição é falsa

termina-se o laço. Esta condição é, em geral, uma expressão lógica que determina o ultimo valor do contador. Por exemplo: `i <= 100` ou `cont > 0`.

Exemplo: No trecho abaixo, o contador `i` é inicializado com o valor `1`. O bloco é repetido enquanto a condição `i <= 10` for verdadeira. O contador é incrementado com a instrução `i++`. Esta estrutura, deste modo, imprime os números `1, 2, ..., 9, 10`.

```
for(i=1; i<=10; i++){
  printf(" %d",i);
}
```

É interessante notar que a mesma estrutura lógica pode ser implementada usando as estruturas `for` ou `do...while`:

Exemplo: As seguintes instruções são plenamente equivalentes:

```
i = 0;                  for(i = 0; i <= 100; i++){
do{                       bloco
  bloco                 }
  i++;
}while(i <= 100);
```

Podem existir mais de uma expressão de *`inicialização`* e de *`incremento`* na estrutura `for`. Estas expressões devem ser separadas por vírgula (`,`). Mas **não pode** haver mais de uma expressão de *`condição`*. Por exemplo: `for(i=0, j=10; i<10; i++, j--){...}`

Programa exemplo: No arquivo `e0503.cpp` existe um programa que calcula a amplitude de um conjunto de valores. Este programa exemplifica o uso da estrutura `for...`

5.5 Estrutura de decisão `if...else`

A estrutura `if...else` é a mais simples estrutura de controle do C. Esta estrutura permite executar um entre vários blocos de instruções. O controle de qual bloco será executado será dado por uma *`condição`* (expressão lógica ou numérica). Esta estrutura pode se apresentar de modos ligeiramente diferentes. Nesta seção vamos apresentar separadamente cada uma das possibilidades de sintaxe.

5.5.1 Decisão de um bloco (`if...`)

A estrutura de decisão de um bloco permite que se execute (ou não) um bloco de instruções conforme o valor de uma condição seja verdadeiro ou falso. O fluxograma desta estrutura é mostrada na figura 5.3.

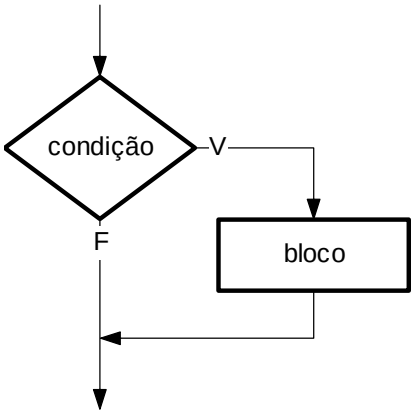

Figura 5.3: *Fluxograma da estrutura de decisão* `if...`

Sintaxe: Decisão com um bloco:

```
if(condição){
  bloco
}
```

onde: *`condição`* é uma expressão lógica ou numérica.

`bloco` é um conjunto de instruções.

Se a condição **verdadeira**, o *`bloco`* é executado. Caso contrário, o bloco não é executado.

Exemplo: No trecho abaixo, se o valor lido for maior que 10, então o seu valor é redefinido como 10. Observe que o bloco constitui-se de um única instrução.

```
printf("Digite o número de repetições: (máximo 10)";
scanf("%d",&iter);
if(iter > 10){
  iter = 10;
}
```

Programa Exemplo: O arquivo `e0504.cpp` mostra um programa que utiliza a estrutura `if...` para emitir um sinal sonoro ao imprimir um número múltiplo de 4.

5.5.2 Decisão de dois blocos (`if...else`)

Também é possível escrever uma estrutura que execute um entre dois blocos de instruções. A figura 5.4 mostra o fluxograma correspondente a esta estrutura de decisão.

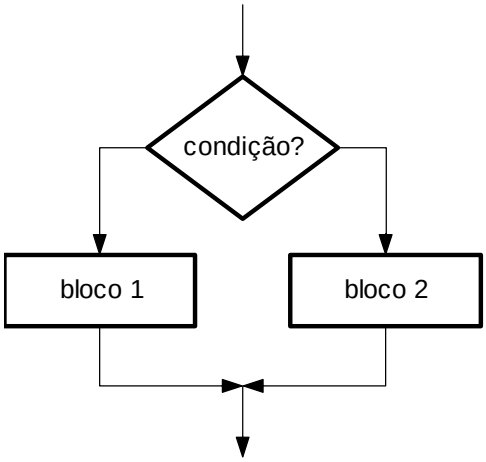

Figura 5.4: *Fluxograma da estrutura de decisão* `if...else`

Sintaxe: Decisão de dois blocos:

```
if(condição){
  bloco 1;
}else{
  bloco 2;
}
```

onde: *`condição`* é uma expressão lógica ou numérica.

`bloco 1` e *`bloco 2`* são conjuntos de instruções.

Se a `condição` for **verdadeira** o *`bloco 1`* é executado. Caso contrário, o *`bloco 2`* é executado.

Exemplo: No trecho abaixo, se o valor de `raiz*raiz` for maior que `num` o valor de `raiz` será atribuído a `max`, caso contrario, será atribuído a `min`.

```
if(raiz*raiz > num){
  max = raiz;
}else{
  min = raiz;
}
```

Programa Exemplo: O arquivo `e0505.cpp` mostra um programa que utiliza a estrutura `if...else` para determinar o tipo de raízes de uma equação de segundo grau.

5.5.3 Decisão de múltiplos blocos (`if...else if...`)

Também é possível escrever uma estrutura que execute um entre múltiplos blocos de instruções. A figura 5.5 mostra o fluxograma correspondente a esta estrutura de decisão.

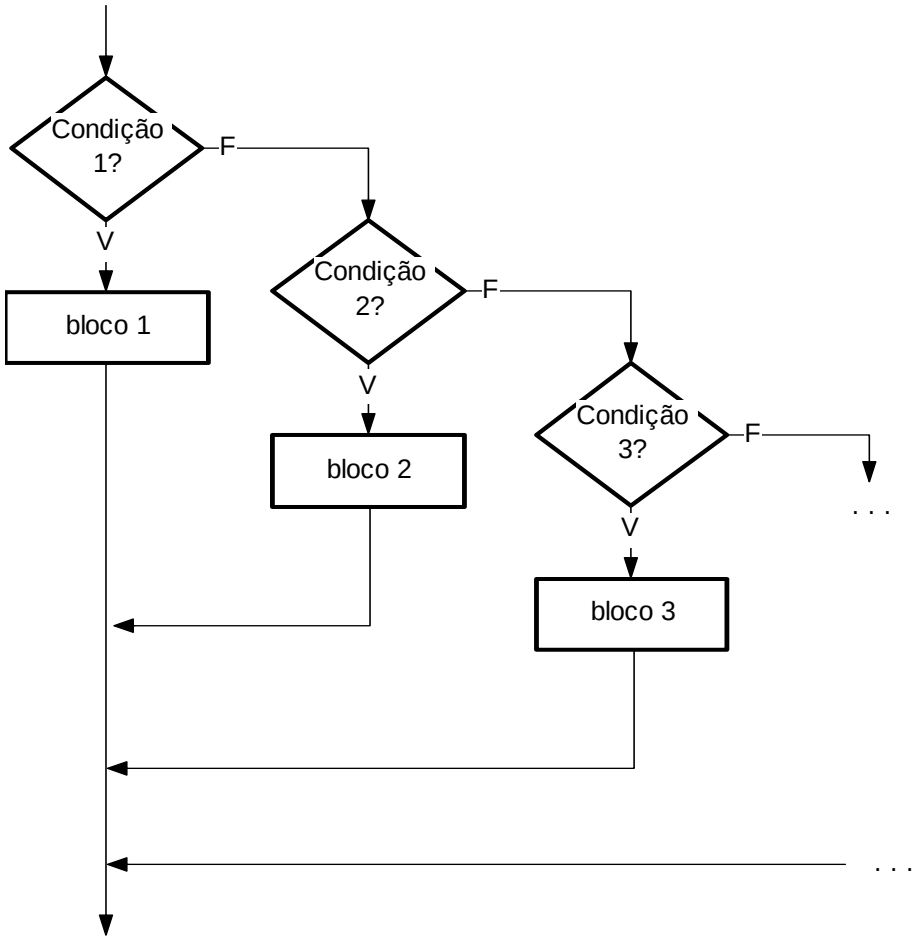

Figura 5.5: *Fluxograma da estrutura de decisão* `if...else if.`

Sintaxe: Decisão de múltiplos blocos:

```
if(condição 1){
  bloco 1;
...
}else if(condição N){
  bloco N;
}else{
  bloco P
}
```

onde: *`condição 1, condição 2, ...`* são expressões lógicas ou numéricas.

`bloco 1 , bloco 2,...` são conjuntos de instruções.

Se a condição 1 for **verdadeira** o *bloco 1* é executado. Caso contrario, a condição 2 é avaliada. Se a condição 2 for **verdadeira** o *bloco 2* é executado. Caso contrario, a condição 3 é avaliada e assim sucessivamente. Se nenhuma condição é **verdadeira** *bloco P* é executado. Observe que apenas um dos blocos é executado.

Exemplo: No trecho abaixo, uma determinada ação é executada se o valor de num for positivo, negativo ou nulo.

```
if(num > 0){
  a = b;
}else if(num < 0){
  a = b + 1;
}else{
  a = b - 1;
}
```

Programa Exemplo: O arquivo e0506.cpp mostra um programa que utiliza a estrutura if...else if para determinar se um número é maior, menor ou igual a outro.

5.6 Estrutura switch...case

A estrutura switch...case é uma estrutura de decisão que permite a execução de um conjunto de instruções a partir pontos diferentes conforme o resultado de uma expressão inteira de controle. O resultado deste expressão é comparado ao valor de cada um dos rótulos, e as instruções são executadas a partir desde rótulo. A figura 5.6 mostra o fluxograma lógico desta estrutura.

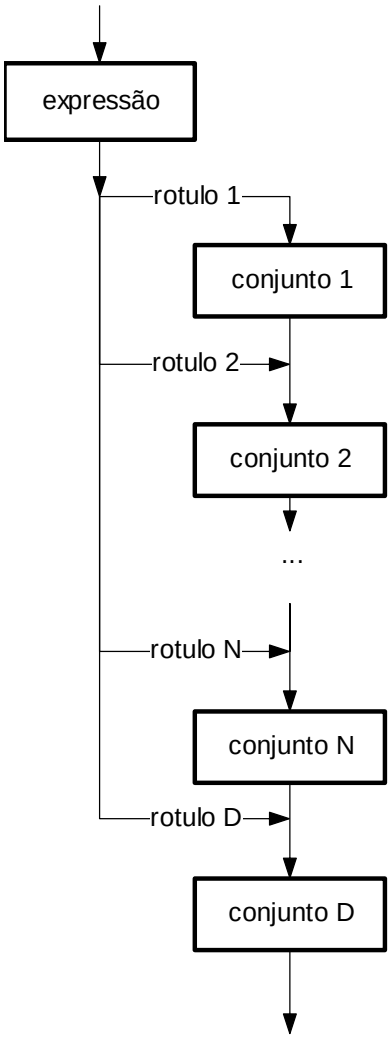

Figura 5.6: *Fluxograma da estrutura* `switch...case`.

Sintaxe: Esta estrutura possui a seguinte sintaxe:

```
switch(expressão){
case rótulo_1:
  conjunto_1
case rótulo_2:
  conjunto_2
...
case rótulo_n:
  conjunto n
[default:
  conjunto d]
}
```

onde:

expressão é uma expressão inteira.

rótulo_1,rótulo_2,...rótulo_n e rótulo_d são constantes inteiras.

conjunto 1, conjunto 2, ..., conjunto n e conjunto d são conjuntos de instruções.

O valor de *expressão* é avaliado e o fluxo lógico será desviado para o conjunto cujo *rótulo* é igual ao resultado da expressão e todas as instruções **abaixo** deste rótulo serão executadas. Caso o resultado da expressão for diferente de todos os valores dos rótulos então *conjunto d* é executado. Os rótulos devem ser expressões constantes inteiras **diferentes** entre si. O rótulo `default` é opcional.

Esta estrutura é particularmente útil quando se tem um conjunto de instruções que se deve executar em ordem, porém se pode começar em pontos diferentes.

Exemplo: O trecho abaixo ilustra o uso da instrução `switch` em um menu de seleção. Neste exemplo, o programa iniciará o processo de usinagem de uma peça em um ponto qualquer dependendo do valor lido.

```
int seleção;
puts("Digite estagio de usinagem:");
scanf("%d",&selecao);
switch(seleção){
case 1:
  // desbaste grosso...
case 2:
  // desbaste fino...
case 3:
  // acabamento...
case 4:
  // polimento...
}
```

Programa Exemplo: O arquivo `e0507.cpp` mostra um programa que utiliza a estrutura `switch` para determinar o valor de um lanche.

5.7 Interrupção e desvio: `break, continue, goto, exit()`

As instruções vistas anteriormente podem sofrer **desvios** e **interrupções** em sua seqüência lógica normal através do uso certas instruções. As instruções que veremos a seguir

devem ser usadas com muita parcimônia, pois fogem da lógica estruturada tem a tendência de tornar um programa incompreensível.

5.7.1 A instrução `break`.

Esta instrução serve para terminar a execução das instruções de um laço de repetição (`for, do...while, while`) ou para terminar um conjunto `switch...case`.

Quando em um laço de repetição, esta instrução força a interrupção do laço independentemente da condição de controle.

Exemplo: No trecho abaixo um laço de repetição lê valores para o cálculo de uma média. O laço possui uma condição de controle sempre verdadeira o que, a principio, é um erro: laço infinito. Porém, a saída do laço se dá pela instrução `break` que é executada quando um valor negativo é lido.

```
puts("digite valores:");
do{
  puts("valor:");
  scanf("%f",&val);
  if(val < 0.0){
    break;     // saída do laço
  }
  num++;
  soma += val;
}while(1);     // sempre verdadeiro
printf("média: %f",soma/num);
```

Exemplo: No exemplo acima, o uso da instrução `break` poderia ter sido evitado, como segue:

```
puts("digite valores:");
do{
  puts("valor:");
  scanf("%f",&val);
  if(val >= 0.0){
    num++;
    soma += val;
  }
```

```
}while(val >= 0.0);
printf("média: %f",soma/num);
```

O outro uso da instrução `break`, em estruturas `switch...case`, serve para separar os conjuntos de instruções em cada `case`.

Exemplo: Estrutura `switch...case` com a instrução `break`:

```
int tipo;
puts("Selecione o sabor de sua pizza:");
puts("Muzzarela Calabreza Alho&Oleo:");
tipo = getch();
switch(tipo){
case ´M´:
  // prepara pizza muzzarela...
case ´C´:
  // prepara pizza calabreza...
case ´A´:
  // prepara pizza Alho&Oleo...
default:
  puts("Opcao incorreta");
}
```

Programa Exemplo: O arquivo `e0508.cpp` mostra um programa que utiliza a estrutura `switch` com a instrução `break` para simular um piano no teclado do computador.

5.7.2 A instrução `continue`.

Esta instrução opera de modo semelhante a instrução `break` dentro de um laço de repetição. Quando executada, ela pula as instruções de um laço de repetição sem sair do laço. Isto é, a instrução força a avaliação da condição de conttrole do laço.

Exemplo: No trecho abaixo revemos um laço de repetição lê valores para o cálculo de uma média. Se `(val < 0.0)` então o programa salta diretamente para a condição de controle, sem executar o resto das instruções.

```
puts("digite valores:");
do{
```

```
  puts("valor:");
  scanf("%f",&val);
  if(val < 0.0){      // se val é negativo...
    continue;         // ...salta para...
  }
  num++;              // se (val < 0.0) estas instruções
  soma += val;        // não são executadas!
}while(val >= 0.0);   // ...fim do laço
printf("média: %f",soma/num);
```

5.7.3 A instrução `goto`.

Esta instrução é chamada de desvio de fluxo. A instrução desvia o programa para um rótulo (posição identificada) no programa. São raros os casos onde a instrução goto é necessária, no entanto, há certas circunstâncias, onde usada com prudência, ela pode ser útil.

Sintaxe: A sintaxe da instrução goto é a seguinte:

```
goto rótulo;
...
rótulo:
...
```

onde *`rótulo`* é um identificador válido.

Exemplo: No trecho abaixo revemos um laço de repetição lê valores para o cálculo de uma média. Foram usadas duas instruções `goto`.

```
puts("digite valores:");
inicio:               // rótulo
  puts("valor:");
  scanf("%f",&val);
  if(val < 0.0){      // se val é negativo...
    goto fim;         // ...salta para fim
  }
  num++;              // se (val < 0.0) estas instruções
  soma += val;        // não são executadas!
goto inicio;          // salta para inicio
fim:                  // rótulo
```

```
printf("média: %f",soma/num);
```

5.7.4 A função `exit()`.

Esta função (não instrução) `exit()` , da biblioteca `stdlib.h`, é uma função que termina a execução de um programa. Normalmente um programa é terminado quando se executa a última sua instrução, porém pode-se terminar a execução do programa a qualquer momento com o uso desta função.

A função `exit()` tem a seguinte declaração: `void exit(int status)`. Onde o argumento da função é um valor inteiro que será passado para o Sistema Operacional: (variável de sistema `errorlevel` no DOS).

Exemplo: No trecho abaixo revemos um laço de repetição lê valores para o cálculo de uma média. Foi usado a função exit para terminar a execução do programa.

```
puts("digite valores:");
do{
  puts("valor:");
  scanf("%f",&val);
  if(val < 0.0){                    // se val é negativo...
    printf("média: %f",soma/num);   // imprime resultado
    exit(0);                        // termina programa
  }
  num++;  soma += val;
}while(1);
```

6. Funções

Funções (também chamadas de **rotinas**, ou **sub-programas**) são a essência da programação estruturada. Funções são segmentos de programa que executam uma determinada tarefa específica. Já vimos o uso de funções nos capítulos anteriores: funções já providenciadas pelas bibliotecas-padrão do C (como `sqrt()`, `toupper()`, `getch()` ou `putchar()`).

É possível ao programador, alem disso, escrever suas próprias rotinas. São as chamadas de **funções de usuário** ou rotinas de usuário. Deste modo pode-se segmentar um programa grande em vários programas menores. Esta segmentação é chamada de **modularização** e permite que cada segmento seja escrito, testado e revisado individualmente sem alterar o funcionamento do programa como um todo. Permite ainda que um programa seja escrito por vários programadores ao mesmo tempo, cada um escrevendo um segmento separado. Neste capítulo, veremos como escrever funções de usuário em C.

6.1 Estrutura das funções de usuário

A estrutura de uma função de usuário é muito semelhante a estrutura dos programas que escrevemos até agora. Uma função de usuário constitui-se de um **bloco de instruções** que definem os procedimentos efetuados pela função, um **nome** pelo qual a chamamos e uma **lista de argumentos** passados a função. Chamamos este conjunto de elementos de **definição da função**.

Exemplo: o código mostrado abaixo é uma função definida pelo usuário para calcular a média aritmética de dois números reais:

```
float media2(float a, float b){
  float med;
  med = (a + b) / 2.0;
  return(med);
}
```

No exemplo acima definimos uma função chamada `media2` que recebe dois argumentos tipo `float`: `a` e `b`. A média destes dois valores é calculada e armazenada na

variável `med` declarada internamente. A função retorna, para o programa que a chamou, um valor também do tipo `float`: o valor da variável `med`. Este retorno de valor é feito pela função `return()`que termina a execução da função e retorna o valor de `med` para o programa que a chamou.

Depois de definimos um função, podemos usá-la dentro de um programa qualquer. Dizemos que estamos fazendo uma **chamada** a função.

Exemplo: No exemplo abaixo chamamos a função `media2()` dentro de um programa principal:

```
void main(){
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);   // chamada a função
  printf(”\nA media destes números e´ %f”, med);
}
```

6.2 Definição de funções

De modo formal, a sintaxe de uma função é a seguinte:

```
tipo_de_retorno nome_da_função(tipo_1 arg_1, tipo_2 arg_2, ...){
  [bloco de instruções da função]
}
```

A primeira linha da função contém a **declaração** da função. Na declaração de uma função se define o **nome** da função, seu **tipo de retorno** e a **lista de argumentos** que recebe. Em seguida, dentro de chaves `{}`, definimos o bloco de instruções da função.

O **tipo de retorno** da função especifica qual o tipo de dado retornado pela função, podendo ser qualquer tipo de dado mostrado na seção 2.3: `int`, `float`, etc. Se a função não retorna nenhum valor para o programa que a chamou devemos definir o retorno como `void`, ou seja um retorno ausente. Se nenhum tipo de retorno for especificado o compilador entenderá que o retorno será tipo `int`.

Vale notar que existe apenas **um** valor de retorno para funções em C. **Não** podemos

fazer o retorno de **dois ou mais** valores como em algumas linguagens (no MatLab: `[media,desvio] = estat(a, b, c, d, e)`). Porém isto não é um limitação séria pois o uso de ponteiros (cap. ?) contorna o problema.

Por ser um identificador, o **nome da função** segue as mesmas regras de definição de identificadores (veja seção 2.2).

A **lista de argumentos** da função especifica quais são os valores que a função recebe. As variáveis da lista de argumentos são manipuladas normalmente no corpo da função (veja seção 6.5 adiante).

A chamada de uma função termina com a instrução `return()` que transfere o controle para o programa chamador da função. Esta instrução tem duas finalidades: determina o **fim lógico** da rotina e o **valor de retorno** da função. O argumento de `return()` será retornado como valor da função.

6.3 Localização das funções

Existem basicamente duas posições possíveis para escrevermos o corpo de uma função: ou **antes** ou **depois** do programa principal. Podemos ainda escrever uma função no **mesmo arquivo** do programa principal ou em **arquivo separado**.

6.3.1 Corpo da função *antes* do programa principal (no mesmo arquivo)

Quando escrevemos a definição de uma função **antes** do programa principal e no mesmo arquivo deste, nenhuma outra instrução é necessária. A sintaxe geral para isto é a seguinte:

Sintaxe: Uma função escrita antes do programa principal:

```
tipo nomef(...){            // definição da função
  [corpo de função]
}
void main(){                // programa principal
  ...
  var = nomef(...)          // chamada da função
```

```
  ...
}
```

Exemplo: Função `media2()` escrita antes do programa principal.

```
float media2(float a, float b){  // função
  float med;
  med = (a + b) / 2.0;
  return(med);
}
void main(){                    // programa principal
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada da função
  printf(”\nA media destes números e´ %f”, med);
}
```

Programa exemplo: No arquivo `e0601.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário.

6.3.2 Corpo da função *depois* do programa principal (no mesmo arquivo)

Quando escrevemos a definição de uma função **depois** do programa principal e no mesmo arquivo deste, devemos incluir um **protótipo** da função chamada. Um protótipo é uma instrução que define o nome da função, seu tipo de retorno e a quantidade e o tipo dos argumentos da função. O protótipo de uma função indica ao compilador quais são as funções usadas no programa principal os tipo. A sintaxe geral para isto é a seguinte:

Sintaxe: Uma função escrita depois do programa principal:

```
void main(){                // programa principal
  tipo nomef(...);          // protótipo da função
  ...
  var = nomef(...)          // chamada a função
  ...
}
tipo nomef(...){            // definição da função
```

```
  [corpo de função]
}
```

Exemplo: Função `media2()` escrita depois do programa principal.

```
void main(){                    // programa principal
  float media2(float,float);    // protótipo de media2()
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada a função
  printf(”\nA media destes números e´ %f”, med);
}
float media2(float a, float b){  // função media2()
  float med;
  med = (a + b) / 2.0;
  return(med);
}
```

Observe que o protótipo de uma função nada mais é que a **declaração** da função sem o seu corpo. Observe ainda que na lista de argumentos do protótipo podem ser escritos apenas os **tipos** dos argumentos.

Programa exemplo: No arquivo `e0602.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário.

6.3.3 Corpo da função escrito em *arquivo separado*

Em C, como em muitas outras linguagens, é permitido que o usuário crie uma função em **um arquivo** e um programa que a chame em outro **arquivo distinto**. Esta facilidade permite a criação de **bibliotecas de usuário**: um conjunto de arquivos contendo funções escritas pelo usuário. Esta possibilidade é uma grande vantagem utilizada em larga escala por programadores profissionais.

Quando escrevemos a definição de uma função em **arquivo separado** do programa principal devemos **incluir** este arquivo no conjunto de arquivos de **compilação** do programa principal. Esta inclusão é feita com a diretiva `#include`. Esta diretiva, vista nas seções 2.4.2 e

3.7.1, instrui o compilador para incluir na compilação do programa outros arquivos que contem a definição das funções de usuário e de biblioteca.

Sintaxe: A sintaxe de inclusão de funções de usuário é a seguinte:

```
#include "path"          // inclusão da função
void main(){             // programa principal
...
  var = nomef(...)       // chamada a função
...
}
```

Na diretiva `#include`, indicamos entre aspas duplas o caminho de localização do arquivo onde está definida a função chamada.

Exemplo: A função `media2()` está escrita em um arquivo separado.

```
#include "c:\tc\userbib\stat.h"     // inclusão da função
void main(){                        // programa principal
  float num_1, num_2, med;
  puts("Digite dois números:");
  scanf("%f %f", &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada a função
  printf("\nA media destes números e´ %f", med);
}
```

Programa exemplo: No arquivo `e0603.cpp` existe um programa que calcula o maior valor entre dois números digitados. Este programa faz uso da função `max()`escrita pelo usuário no arquivo `e0604.cpp`.

Observação: Um arquivo pode conter a definição de uma ou mais funções. Em geral, quando o arquivo possui apenas uma função ele é nomeado com o mesmo nome da função e extensão `*.cpp` ou `*.c`. Por exemplo, poderíamos definir a função `media()` no arquivo `media.cpp`. Quando um arquivo possui a definição de mais de uma função, ele é nomeado com a extensão `*.h` ou `*.lib`. Por exemplo: poderíamos criar um conjunto de funções estatísticas chamadas `media()`, `dsvpd()`, `moda()`, `max()`, `min()`, etc. definindo-as em um arquivo chamado `stat.h`.

6.4 Hierarquia de Funções

Sempre é possível que um programa principal chame uma função que por sua vez chame outra função... e assim sucessivamente. Quando isto acontece dizemos que a função chamadora tem hierarquia maior (ou superior) a função chamada. Ou que a função chamadora está em um nível hierárquico superior a função chamada.

Quando isto ocorre, devemos ter o cuidado de definir (ou incluir) as funções em ordem crescente de hierarquia, isto é, uma função **chamada** é escrita antes de uma função **chamadora**. Isto se deve ao fato de que o compilador deve conhecer uma função antes de que chamada seja compilada.

Programa exemplo: No arquivo `e0605.cpp` existe um jogo de "*jackpot*" que ilustra o uso de várias rotinas por um programa principal. Observe também que estas funções chamam-se umas as outras.

Neste programa exemplo, os níveis hierárquicos das funções podem ser colocados da seguinte maneira:

```
                                main()

  regras()  abertura()  sorte()  plim_plim()  saida()  simnao()

                          imprimec()   roleta()

                             pinta()   bip()
```

No exemplo acima temos um primeiro nível onde se encontra a função main() [o programa principal também é uma função] que chama as funções x, y, z. A função x por sua vez chamas as funções s, r, e t. Observe que neste exemplo os protótipos das funções estão colocados de modo a que as funções de menor hierarquia são escritas antes das funções de maior hierarquia.

6.5 Regra de escopo para variáveis

A regra de escopo define o **âmbito de validade** de variáveis. Em outras palavras define onde as variáveis e funções são reconhecidas. Em C, uma variável só pode ser **usada** após

ser **declarada** (ver seção 2.3.2). Isto por que o processador deve reservar um local da memória para armazenar os valores atribuídos à variável. Porém o local, do programa, onde uma variável é declarada define ainda seu **escopo** de validade. Uma variável pode ser **local**, **global** ou **formal** de acordo com o local de declaração.

Variáveis Locais: Uma variável é dita *local*, se for declarada **dentro do bloco** de uma função. Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco. O espaço de memória alocado para esta variável é **criado** quando a execução do bloco é iniciada e **destruído** quando encerrado, assim variáveis de mesmo nome mas declaradas em blocos distintos, são para todos os efeitos, variáveis distintas.

Exemplo:

```
float media2(float a, float b){
  float med;
  med = (a + b) / 2.0;
  return(med);
}
void main(){
  float num_1, num_2, med;
  puts(”Digite dois números:”);
  scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);
  printf(”\nA media destes números e´ %f”, med);
}
```

No exemplo acima, `med` é uma variável local definida pela função `media()`. Outra variável `med` é também definida pela função `main()`. Para todos os efeitos estas variáveis são distintas.

Variáveis Formais: Uma *variável formal* é uma variável local declarada na **lista de parâmetros** de uma função. Deste modo, tem validade apenas dentro da função onde é declarada, porém serve de suporte para os valores passados pelas funções. As variaveis formais na **declaração** da função e na **chamada** da função podem ter nomes distintos. A única exigência é de que sejam do mesmo tipo.

Por serem variáveis locais, os valores que uma função passa para outra não são alterados pela função chamada. Este tipo de passagem de argumentos é chamado de

passagem por valor pois os valores das variáveis do programa chamador são copiados para as correspondentes variáveis da função chamada. Veremos no capitulo ? como alterar os valores das variáveis do programa chamador. Chamaremos esta passagem de **passagem por endereço**.

No exemplo acima, a e b são parâmetros formais declarados na função media2(). Observe que a função é chamada com os valores de `num_1` e `num_2`. Mesmo que os valores de a e b fossem alterados os valores de `num_1` e `num_2` não seriam alterados.

Variáveis Globais: Uma variável é dita *global*, se for declarada **fora do bloco** de uma função. Uma variável global tem validade no escopo de todas as funções, isto é, pode ser acessadas e modificada por qualquer função. O espaço de memória alocado para esta variável é **criado** no momento de sua declaração e **destruído** apenas quando o programa é encerrado.

Exemplo: Uso de variáveis globais.

```
float a, b, med;
void media2(void){
med = (a + b) / 2.0;
}
void main(){
  puts(”Digite dois números:”);
  scanf(”%f %f”, &a, &b);
 media2();
  printf(”\nA media destes números e´ %f”, med);
}
```

No exemplo acima, `a, b, med` são variáveis globais definidas fora dos blocos das funções `media()` e `main()`. Deste modo ambas as funções tem pleno acesso as variáveis, podendo ser acessadas e modificadas por quaisquer uma das funções. Assim não é necessário a passagem de parâmetros para a função.

6.6 Recursividade

A recursividade talvez seja a mais importante vantagem das funções em C. **Recursão** é o processo pelo qual uma função chama a si mesma repetidamente um numero finito de vezes. Este recurso é muito útil em alguns tipos de algoritmos chamados de **algoritmos recursivos**.

Vejamos um exemplo *clássico* para esclarecermos o conceito: calculo do **fatorial** de um número. A definição de fatorial é:

```
n! = n . (n-1) . (n-2) . ... . 3 . 2 . 1
0! = 1
```

onde n é um numero inteiro positivo. Uma propriedade (facilmente verificável) dos fatoriais é que:

```
n! = n . (n-1)!
```

Esta propriedade é chamada de propriedade recursiva: o fatorial de um numero pode ser calculado através do fatorial de seu antecessor. Ora, podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o calculo de fatoriais. Para criarmos uma rotina recursiva, em C, basta criar uma chamada a própria função dentro dela mesma, como no exemplo a seguir.

Programa exemplo: No arquivo `e0606.cpp` existe uma rotina recursiva para o calculo de fatoriais.

Uma função recursiva cria a cada chamada um novo conjunto de variáveis locais. Não existe ganho de velocidade ou espaço de memória significativo com o uso de funções recursivas. Teoricamente uma algoritmo recursivo pode ser escrito de forma iterativa e vice-versa. A principal vantagem destes algoritmos é que algumas classes de algoritmos [de inteligência artificial, simulação numérica, busca e ordenação em arvore binaria, etc.] são mais facilmente implementadas com o uso de rotinas recursivas. O estudo deste tipo de algoritmo está, porém, além do alcance deste texto.

7. Vetores

Neste capítulo estudaremos o conceito de vetor, sua declaração e uso. Como são usados vetores em argumentos de funções. E, ainda, como trabalhar com vetores de mais de uma dimensão.

7.1 Introdução

Em muitas aplicações queremos trabalhar com conjuntos de dados que são **semelhantes em tipo**. Por exemplo o conjunto das alturas dos alunos de uma turma, ou um conjunto de seus nomes. Nestes casos, seria conveniente poder colocar estas informações sob um mesmo conjunto, e poder referenciar cada dado individual deste conjunto por um número índice. Em programação, este tipo de estrutura de dados é chamada de **vetor** (ou *array*, em inglês) ou, de maneira mais formal **estruturas de dados homogêneas**.

Exemplo: A maneira mais simples de entender um vetor é através da visualização de um **lista**, de elementos com um nome coletivo e um índice de referência aos valores da lista.

```
n   nota
0   8.4
1   6.9
2   4.5
3   4.6
4   7.2
```

Nesta lista, n representa um número de referência e `nota` é o nome do conjunto. Assim podemos dizer que a 2ª nota é `6.9` ou representar `nota[1] = 6.9`

Esta não é a única maneira de estruturar conjunto de dados. Também podemos organizar dados sob forma de tabelas. Neste caso, cada dado é referenciado por dois índices e dizemos que se trata de um **vetor bidimensional** (ou **matriz**)[1]. Vetores de mais de uma dimensão serão vistos na seção 7.5.

[1] Alguns autores preferem chamar todos os tipos de estruturas homogêneas, não importando o numero de índices de referência (ou dimensões) de *vetores*. Outros preferem chamar de *matrizes*. Outros ainda distinguem *vetores* (uma dimensão) de *matrizes* (mais de uma dimensão), etc. Não vamos entrar no mérito da questão (existem boas justificativas para todas as interpretações) e, nesta apostila, vamos usar a primeira nomenclatura: toda estrutura homogênea de dados será chamada de vetor.

7.2 Declaração e inicialização de vetores

7.2.1 Declaração de vetores

Em C, um vetor é um conjunto de variáveis de um **mesmo tipo** que possuem um nome identificador e um índice de referência.

Sintaxe: A sintaxe para a declaração de um vetor é a seguinte:

```
tipo nome[tam];
```

onde:

tipo é o **tipo** dos elementos do vetor: `int, float, double` ...

nome é o **nome** identificador do vetor. As regras de nomenclatura de vetores são as mesmas usadas em variáveis (seção 2.2.1).

tam é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar.

Exemplo: Veja as declarações seguintes:

```
int idade[100]; // declara um vetor chamado 'idade' do tipo
                // 'int' que recebe 100 elementos.
float nota[25]; // declara um vetor chamado 'nota' do tipo
                // 'float' que pode armazenar 25 números.
char nome[80];  // declara um vetor chamado 'nome' do tipo
                // 'char' que pode armazenar 80 caracteres.
```

Na declaração de um vetor estamos reservando espaço de memória para os elementos de um vetor. A quantidade de memória (em *bytes*) usada para armazenar um vetor pode ser calculada como:

*quantidade de memória = tamanho do tipo * tamanho do vetor*

Assim, no exemplo anterior, a quantidade de memória utilizada pelos vetores é, respectivamente, `200(2x100)`, `100(4x25)` e `80(80x1)` *bytes*.

7.2.2 Referência a elementos de vetor

Cada elemento do vetor é referenciado pelo **nome** do vetor seguido de um **índice** inteiro. O **primeiro** elemento do vetor tem índice `0` e o **último** tem índice `tam-1`. O índice de um vetor deve ser **inteiro**.

Ex
emplo: Algumas referências a vetores:

```
#define MAX 5
int i = 7;
float valor[10];                // declaração de vetor
valor[1] = 6.645;
valor[MAX] = 3.867;
valor[i] = 7.645;
valor[random(MAX)] = 2.768;
valor[sqrt(MAX)] = 2.705;     // NÃO é válido!
```

7.2.2 Inicialização de vetores

Assim como podemos inicializar variáveis (por exemplo: `int j = 3;`), podemos inicializar vetores.

Sintaxe: A sintaxe para a inicialização dos elementos de um vetor é:

`tipo nome``[`*`tam`*`] = {`*`lista de valores`*`};`

onde:

`lista de valores` é uma lista, separada por vírgulas, dos valores de cada elemento do vetor.

Exemplo: Veja as inicializações seguintes. Observe que a inicialização de `nota` gera o vetor do exemplo do início desta seção.

```
int dia[7] = {12,30,14,7,13,15,6};
float nota[5] = {8.4,6.9,4.5,4.6,7.2};
char vogal[5] = {'a', 'e', 'i', 'o', 'u'};
```

Opcionalmente, podemos inicializar os elementos do vetor enumerando-os um a um.

Exemplo: Observe que estas duas inicializações são possíveis:

```
int cor_menu[4] = {BLUE,YELLOW,GREEN,GRAY};
```

ou

```
int cor_menu[4];
cor_menu[0] = BLUE;
cor_menu[1] = YELLOW;
cor_menu[2] = GREEN;
cor_menu[3] = GRAY;
```

Programa Exemplo: O arquivo `e0701.cpp` contém um programa que mostra o uso de vetores: declaração, inicialização, leitura e escrita de elementos...

7.3 Tamanho de um vetor e segmentação de memória

7.3.1 Limites

Na linguagem C, devemos ter cuidado com os limites de um vetor. Embora na sua declaração, tenhamos definido o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento dentro do vetor ou não.

Por exemplo se declaramos um vetor como `int valor[5]`, teoricamente só tem sentido usarmos os elementos `valor[0], ..., valor[4]`. Porém, o C não acusa **erro** se usarmos `valor[12]` em algum lugar do programa. Estes testes de limite **devem** ser feitos **logicamente** dentro do programa.

Este fato se deve a maneira como o C trata vetores. A memória do microcomputador é um espaço (físico) particionado em porções de 1 *byte*. Se **declaramos** um vetor como `int vet[3]`, estamos **reservando** 6 *bytes* (3 segmentos de 2 *bytes*) de memória para armazenar os seus elementos. O primeiro segmento será reservado para `vet[0],` o segundo segmento para `vet[1]` e o terceiro segmento para `vet[2]`. O segmento inicial é chamado de segmento **base**, de modo que `vet[0]` será localizado no segmento base. Quando acessamos o elemento `vet[i]`, o processador acessa o segmento localizado em **base+i**. Se `i` for igual a 2, estamos acessando o segmento **base+2** ou `vet[2]`(o ultimo segmento reservado para o vetor). Porém, se `i` for igual a 7, estamos a acessando segmento **base+7** que **não foi reservado** para os elementos do vetor e que provavelmente está sendo usado por uma outra variável ou contém informação espúria (lixo).

Observe que acessar um segmento fora do espaço destinado a um vetor pode **destruir informações** reservadas de outras variáveis. Estes erros são difíceis de detectar pois o

compilador não gera nenhuma mensagem de erro... A solução mais adequada é sempre avaliar os limites de um vetor antes de manipulá-lo.

A princípio este fato poderia parecer um defeito da linguagem, mas na verdade trata-se de um recurso muito poderoso do C. Poder manipular sem restrições todos os segmentos de memória é uma flexibilidade apreciada pelos programadores.

Programa Exemplo: O arquivo `e0702.cpp` contém um programa que mostra o acesso de elementos dentro e fora de um vetor. Note que o compilador não acusa nenhum erro de sintaxe!

7.3.2 Tamanho parametrizado

Na linguagem C não é possível, usando a sintaxe descrita acima, declarar um vetor com tamanho **variável**.

Exemplo: O trecho de código seguinte faz uma declaração **errada** de vetor.

```
...
int num;
puts("Quantos números?");
scanf("%d”, &num);
float valor[num];  // declaração de vetor (errado!)
...
```

Mas é possível declarar um vetor com tamanho **parametrizado**: usando uma **constante simbólica**. Declaramos uma constante simbólica (parâmetro) com a diretiva `#define` no cabeçalho do programa e depois declaramos o vetor com esta constante simbólica como tamanho do vetor. Deste modo podemos alterar o número de elementos do vetor **antes** de qualquer **compilação** do programa. Esta é uma maneira simples de administrar o espaço de memória usado pelo programa, e também testar os limites de um vetor.

Programa Exemplo: O arquivo `e0703.cpp` contém um programa que mostra a declaração de um vetor com tamanho parametrizado. Mostra também o uso deste parâmetro como teste de limite do vetor. Compile este programa com outros valores para o parâmetro `MAX` e verifique que a execução do programa é alterada automaticamente.

No capitulo seguinte, seção ?, será vista uma maneira de declararmos um vetor com um número variável de elementos, usando ponteiros. Este tipo de declaração é chamada de *alocação dinâmica de memória.*

7.4 Passando Vetores para Funções

Vetores, assim como variáveis, podem ser usados como argumentos de funções. Vejamos como se declara uma função que recebe um vetor e como se passa um vetor para uma função.

Sintaxe: Na *passagem* de vetores para funções usamos a seguinte sintaxe:
nome_da_função(*nome_do_vetor*)

onde:
nome_da_função é o nome da função que se está chamando.
nome_do_vetor é o nome do vetor que queremos passar. Indicamos **apenas** o nome do vetor, **sem** índices.

Sintaxe: Na *declaração* de funções que recebem vetores:

```
tipo_função nome_função(tipo_vetor nome_vetor[]){
...
}
```

onde:
tipo_função é o tipo de retorno da função.
nome_função é o nome da função.
tipo_vetor é o tipo de elementos do vetor.
nome_vetor é o nome do vetor. Observe que depois do nome do vetor temos um índice vazio `[]` para indicar que estamos recebendo um vetor.

Exemplo: Observe o exemplo abaixo:
Na declaração da função:

```
float media(float vetor[],float N){  // declaração da função
  ...
}
```

Na chamada da função:

```
void main(){
  float valor[MAX]; // declaração do vetor
  ...
  med = media(valor, n); // passagem do vetor para a função
  ...
}
```

Programa Exemplo: O arquivo `e0704.cpp` contém um programa que mostra a passagem de vetores para funções.

Atenção: Ao contrário das variáveis comuns, o conteúdo de um vetor **pode ser modificado** pela função chamada. Isto significa que podemos passar um vetor para uma função e alterar os valores de seus elementos. Isto ocorre porque a passagem de **vetores** para funções é feita de modo especial dito *Passagem por endereço*. Uma abordagem mais detalhada deste procedimento será feita no capítulo ? sobre ponteiros.

Portanto devemos ter cuidado ao manipularmos os elementos de um vetor dentro de uma função para não modifica-los por descuido.

Programa Exemplo: O arquivo `e0705.cpp` contém um programa que mostra a modificação de um vetor por uma função. Neste caso a modificação é **desejada** pois queremos ordenar os elementos do vetor.

7.5 Vetores Multidimensionais

Vetores podem ter mais de uma dimensão, isto é, mais de um índice de referência. Podemos ter vetores de duas, três, ou mais dimensões. Podemos entender um vetor de duas dimensões (por exemplo) associando-o aos dados de um tabela.

Exemplo: Um vetor bidimensional pode ser visualizado através de uma **tabela**.

nota	0	1	2
0	8.4	7.4	5.7
1	6.9	2.7	4.9
2	4.5	6.4	8.6
3	4.6	8.9	6.3
4	7.2	3.6	7.7

Nesta tabela representamos as notas de 5 alunos em 3 provas diferentes (matemática, física e química, por exemplo). O nome `nota` é o nome do conjunto, assim podemos dizer que a nota do 3º aluno na 2ª prova é `6.4` ou representar `nota[2,1] = 6.4`

7.5.1 Declaração e inicialização

A declaração e inicialização de vetores de mais de uma dimensão é feita de modo semelhante aos vetores unidimensionais.

Sintaxe: A sintaxe para declaração de vetores multidimensionais é:

```
tipo nome[tam_1][tam_2]...[tam_N]={{lista},{lista},...{lista}};
```

onde:

`tipo` é o tipo dos elementos do vetor.

`nome` é o nome do vetor.

`[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão do vetor.

`{{lista},{lista},...{lista}}` são as listas de elementos.

Exemplo: veja algumas declarações e inicializações de vetores de mais de uma dimensão. Observe que a inicialização de `nota` gera a tabela do exemplo do início desta seção.

```
float nota[5][3] = {{8.4,7.4,5.7},
                    {6.9,2.7,4.9},
                    {4.5,6.4,8.6},
                    {4.6,8.9,6.3},
                    {7.2,3.6,7.7}};
int tabela[2][3][2] = {{{10,15}, {20,25}, {30,35}},
                       {{40,45}, {50,55}, {60,65}};
```

Neste exemplo, `nota` é um vetor **duas** dimensões (`[][]`). Este vetor é composto de 5 vetores de 3 elementos cada. `tabela` é vetor de três dimensões (`[][][]`). Este vetor é composto de 2 vetores de 3 sub-vetores de 2 elementos cada.

7.5.2 Passagem de vetores multidimensionais para funções

A sintaxe para *passagem* de vetores multidimensionais para funções é semelhante a passagem de vetores unidimensionais: chamamos a função e passamos o **nome** do vetor, **sem** índices. A única mudança ocorre na declaração de funções que recebem vetores:

Sintaxe: Na *declaração* de funções que recebem vetores:

```
tipo_f função(tipo_v vetor[tam_1][tam_2]...[tam_n]){
...
}
```

Observe que depois do nome do vetor temos os índices com contendo os tamanhos de cada dimensão do vetor.

Exemplo: Observe o exemplo abaixo:

Na declaração da função:

```
int max(int vetor[5][7],int N, int M){  // declaração da função
...
}
```

Na chamada da função:

```
void main(){
  int valor[5][7]; // declaração do vetor
  ...
  med = media(valor, n); // passagem do vetor para a função
  ...
}
```

Programa Exemplo: O arquivo `e0706.cpp` contém um programa que mostra a manipulação de vetores bidimensionais: leitura de elementos, escrita, passagem para funções, etc.

Observações: Algumas observações a respeito de vetores multidimensionais podem ser feitas:

- Do mesmo modo que vetores unidimensionais, os vetores multidimensionais podem ter seus elementos modificados pela função chamada.
- Os índices dos vetores multidimensionais, também começam em `0`. Por exemplo: `vet[0][0]`, é o primeiro elemento do vetor.
- Embora uma tabela não seja a única maneira de visualizar um vetor bidimensional, podemos entender o **primeiro** índice do vetor como o índice de **linhas** da tabela e o **segundo** índice do vetor como índice das **colunas**.

8. Endereços e Ponteiros

Neste capítulo veremos a definição e os principais usos de ponteiros. Veremos as operações fundamentais como ponteiros , a estreita relação de ponteiros vetores e strings e ainda a alocação dinâmica de memória e a passagem de funções para funções com o uso de ponteiros..

9.1 Introdução

Toda informação (dado armazenado em variável simples ou vetor) que manipulamos em um programa está armazenado na memória do computador. Cada informação é representada por um certo conjunto de *bytes* (Ver capítulo 2). Por exemplo: caracter (`char`): 1 *byte*, inteiro (`int`): 2 *bytes*, etc.

Cada um destes conjuntos de *bytes*, que chamaremos de **bloco**, tem um nome e um **endereço** de localização especifica na memória.

Exemplo: Observe a instrução abaixo:

```
int num = 17;
```

ao interpretar esta instrução, o processador pode especificar:

Nome da informação: `num`

Tipo de informação: `int`

Tamanho do bloco (número de *bytes* ocupados pela informação): 2

Valor da informação: 17

Endereço da informação (localização do primeiro ***byte***): `8F6F:FFF2` (hexadecimal)

Em geral, interessa ao **programador** apenas os nomes simbólicos que representam as informações, pois é com estes nomes que são realizadas as operações do seu algoritmo. Porém, ao **processador** interessa os endereços dos blocos de informação pois é com estes endereços que vai trabalhar.

Programa Exemplo: O arquivo `e0801.cpp` contém um programa com instruções para inspecionar o endereço de uma variável, usando o recurso *Inspect* do Turbo C++. Observe que

o endereço mostrado corresponde ao **primeiro** *byte* do bloco, mesmo que o bloco ocupe mais de um *byte*: No caso, um float ocupa um bloco de 4 *bytes*.

8.2 Ponteiros

Ponteiros são variáveis que contém endereços. Neste sentido, estas variáveis *apontam* para algum determinado endereço da memória. Em geral, o ponteiro aponta para o endereço de alguma variável declarada no programa.

8.2.1 Declaração de ponteiros.

Quando declaramos um ponteiro, devemos declará-lo com o mesmo tipo (int, char, etc.) do bloco a ser apontado. Por exemplo, se queremos que um ponteiro aponte para uma variável `int` (bloco de 2 bytes) devemos declará-lo como `int` também.

Sintaxe: A sintaxe da declaração de um ponteiro é a seguinte:

```
tipo_ptr *nome_ptr_1;
```

ou

```
tipo_ptr* nome_ptr_1, nome_ptr_2, ...;
```

onde:

`tipo_ptr` : é o tipo de bloco para o qual o ponteiro apontará.

`*` : é um operador que indica que `nome_ptr` é um ponteiro.

`nome_ptr_1, nome_ptr_2,...:` são os nomes dos ponteiros (os nomes dos ponteiros obedecem as mesmas regras da seção 2.2.1)

Exemplo: Veja as seguintes instruções:

```
int *p;
float* s_1, s_2;
```

A primeira instrução declara um ponteiro chamado p que aponta para um inteiro. Este ponteiro aponta para o **primeiro** endereço de um bloco de **dois** *bytes*. Sempre é necessário declarar o tipo do ponteiro. Neste caso dizemos que declaramos um ponteiro tipo `int`.

A segunda instrução declara dois ponteiros (`s_1` e `s_2`) do tipo `float`. Observe que o `*` está justaposto ao tipo: assim todos os elementos da lista serão declarados ponteiros.

8.2.2 Operadores & e *

Quando trabalhamos com ponteiros, queremos fazer duas coisas basicamente:

- conhecer **endereço** de uma variável;
- conhecer o **conteúdo** de um endereço.

Para realizar estas tarefas a linguagem C nos providencia dois operadores especiais:

- o operador de **endereço**: `&`
- o operador de **conteúdo**: `*`

O operador de **endereço (&)** determina o endereço de uma variável (o primeiro *byte* do bloco ocupado pela variável). Por exemplo, `&val` determina o endereço do bloco ocupado pela variável `val`. Esta informação não é totalmente nova pois já a usamos antes: na função `scanf()`.

Exemplo: Quando escreve-se a instrução:

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

estamos nos referimos **endereço** do bloco ocupado pela variável `num`. A instrução significa: "*leia o buffer do teclado, transforme o valor lido em um valor inteiro (2 bytes) e o armazene no bloco localizado no endereço da variável* `num`".

Exemplo: Para se atribuir a um ponteiro o endereço de uma variável escreve-se:

```
int *p, val=5;  // declaração de ponteiro e variável
p = &val;       // atribuição
```

Observe que o ponteiro p deve ser declarado anteriormente com o mesmo tipo da variável para a qual ele deve apontar.

O operador **conteúdo** (*) determina o conteúdo (valor) do dado armazenado no endereço de um bloco apontado por um ponteiro. Por exemplo, `*p` determina conteúdo do bloco apontado pelo ponteiro p. De forma resumida: o operador (*) determina o conteúdo de um endereço.

Exemplo: Para se atribuir a uma variável o conteúdo de um endereço escreve-se:

```
int *p = 0x3f8, val;  // declaração de ponteiro e variável
val = *p;             // atribuição
```

Observações:

- O operador **endereço** (&) somente pode ser usado em uma única variável. Não pode ser usado em expressões como, por exemplo, `&(a+b)`.
- O operador **conteúdo** (*) somente pode ser usado em variáveis ponteiros.

Programa Exemplo: O arquivo `e0802.cpp` contém um programa que mostra como se manipulam ponteiros e variáveis. Como se transportam informações entre ponteiros e variáveis.

8.3 Operações elementares com ponteiros

Ponteiros são variáveis especiais e obedecem a regras especiais. Deste modo, existem uma série de operações (aritméticas, lógicas, etc.) envolvendo ponteiros que são permitidas e outras não. A seguir são destacadas algumas operações que podem ser executadas com ponteiros.

- A um ponteiro pode ser atribuído o endereço de uma variável comum.

Exemplo: Observe o trecho abaixo:

```
...
int *p;
int s;
p = &s;  // p recebe o endereço de s
...
```

- Um ponteiro pode receber o valor de outro ponteiro, isto é, pode receber o endereço apontado por outro ponteiro, desde que os ponteiros sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
float *p1, *p2, val;
p1 = &val;  // p1 recebe o endereço de val...
p2 = p1;    // ...e p2 recebe o conteúdo de p2 (endereço de val)
```

- Um ponteiro pode receber um endereço de memória diretamente. Um endereço é um numero inteiro. Em geral, na forma hexadecimal (`0x...`). Nesta atribuição devemos, em geral, forçar uma conversão de tipo usando *casting* para o tipo de ponteiro declarado.

Exemplo: Observe o trecho abaixo:

```
...
int *p1;
float p2ç
p1 = 0x03F8;  // endereço da porta serial COM1
p2 = (float)0x0FF6; // casting
...
```

- A um ponteiro pode ser atribuído o valor **nulo** usando a constante simbólica `NULL` (declarada na biblioteca `stdlib.h`). Um ponteiro com valor `NULL` não aponta para lugar nenhum! Algumas funções a utilizam para registrar uma atribuição ilegal ou sem sucesso (ver função `malloc()` adiante)

Exemplo: Observe o trecho abaixo:

```
#include <stdlib.h>
...
char *p;
p = NULL;
...
```

- Uma quantidade inteira pode ser adicionada ou subtraída de um ponteiro. A adição de um inteiro n a um ponteiro p fará com que ele aponte para o endereço do n-*ésimo* bloco seguinte.

Exemplo: Observe o trecho abaixo:

```
...
double *p, *q, var;
p = &var
q = ++p;  // q aponta para o bloco seguinte ao ocupado por var
p = q - 5;// p aponta para o quinto bloco anterior a q
...
```

- Dois ponteiros podem ser comparados (usando-se operadores lógicos) desde que sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
if(px == py){  // se px aponta para o mesmo bloco que py ...
if(px > py){   // se px aponta para um bloco posterior a py ...
if(px != py){  // se px aponta para um bloco diferente de py ...
if(px == NULL) // se px é nulo...
...
```

Programa Exemplo: O arquivo `e0803.cpp` contém um programa que mostra como se utilizam algumas operações elementares com ponteiros com ponteiros.

8.4 Ponteiros, endereços e funções

Porque usar ponteiros? A primeira vantagem da utilização de ponteiros em programas talvez esteja relacionada a sua utilização como argumentos de funções.

8.4.1 Passagem de dados por valor ou por referencia

No capítulo 6 (seção 6.5) afirma-se que o valor de uma variável `var` de uma função `fun_1()` passada para uma outra função `fun_2()` **não** podem ser alterado pela função `fun_2()`. De fato, isto é verdade se passamos o **valor** da variável `var` para a função `fun_2()`. Mas o valor de `var` pode ser alterado por `fun_2()` passamos seu **endereço.**

No primeiro caso, dizemos que a passagem de dados de uma função para outra ocorreu por **valor**. No segundo caso, dizemos que houve uma passagem por **referência**. Vejamos em detalhe uma definição destes tipos de passagem de dados entre funções:

Passagem por Valor: A passagem por valor significa que passamos de uma função para outra o **valor** de uma variável, isto é, a função chamada recebe um cópia do valor da variável. Assim qualquer alteração deste valor, pela função chamada, será uma alteração de uma cópia do valor da variável. O valor original na função chamadora **não é alterado** pois o valor original e copia ficam em blocos de memória diferentes.

Passagem por Referencia: A passagem por referencia significa que passamos de uma função para outra o **endereço** de uma variável, isto é, a função chamada recebe sua **localização** na memória através de um ponteiro. Assim qualquer alteração no conteúdo apontado pelo do ponteiro será uma alteração no conteúdo da variável original. O valor original **é alterado**.

Sintaxe: A sintaxe da passagem de endereço é a seguinte:

- na função chamada:

```
tipof nomef(tipop nomep){
...
```

onde:

tipof é o tipo de retorno da função.

nomef é o nome da função chamada.

tipop é o tipo do ponteiro (igual ao tipo da variável passada).

nomep é o nome do ponteiro.

- na função chamadora:

```
...
nomef(end_var);
...
```

onde:

nomef é o nome da função chamada.

end_var é o endereço da variável passada como argumento.

Exemplo: Observe o exemplo abaixo:

```
void troca(int *p1, int *p1){ // declaração da função
                              //   Observe: ponteiros
  int temp;      // variável temporária
  temp = *p1;    // temp recebe o conteúdo apontado por p1
  *p1 = *p2;     // o conteúdo de p1 recebe o conteúdo de p2
  *p2 = temp;    // o conteúdo de p2 recebe o valor de temp
}
void main(){             // programa principal
  int a,b;               // declaração das variáveis
  scanf("%d %d",&a,&b);  // leitura das variáveis
  troca(&a,&b);          // passagem dos endereços de a e b
```

```
  printf("%d %d",a,b);   // imprime valores (trocados!)
}
```

Neste exemplo temos uma função `troca()` que troca entre si os valores de duas variáveis. Esta função recebe os **endereços** das variáveis passadas pela função `main()`, armazenando-os nos ponteiros `p1` e `p2`. Dentro da função, troca-se os **conteúdos** dos endereços apontados.

Programa Exemplo: O arquivo `e0804.cpp` contém um programa que mostra a diferença entre a passagem de dados por valor e passagem por referencia.

8.4.2 Retorno de dados em funções

A **passagem por referencia** permite que (formalmente) uma função retorne quantos valores se desejar. Dissemos no capítulo 6 (seção 6.2) que uma função somente pode retornar um valor. Isto continua sendo valido pois o C assim define funções. Porem com o uso de ponteiros pode-se contornar esta situação. Vejamos:

Imagine que queremos escrever uma função `stat()` com a finalidade de calcular a *media aritmética* e o *desvio padrão* de um conjunto de dados. Observe: o retorno destes dados não poder ser via instrução `return()` pois isto não é permitido. A solução é criar (na função `main()`, por exemplo) duas variáveis para armazenar os valores desejados (`med` e `desvio`, por exemplo) e então passar os endereços destas variáveis para a função `stat()`. A função recebe esses endereços e os armazena em ponteiros (`pm` e `pd`, por exemplo). Em seguida, faz os cálculos necessários, armazenando-os nos endereços recebidos. Ao término da execução da função os valores de `med` e `desvio` serão atualizados automaticamente. Este recurso é bastante utilizado por programadores profissionais.

Programa Exemplo: O arquivo `e0805.cpp` contém um programa que mostra o *'retorno'* de vários dados de uma função. Na verdade trata-se da passagem de valores por referencia.

8.4.3 Ponteiro como argumento de função

Observe que nos exemplos acima, a passagem de endereços foi feita através do *operador endereço* (`&`). Também é possível passar um endereço através de um ponteiro já que o conteúdo de um ponteiro **é** um endereço.

Exemplo: Observe o trecho de programa abaixo.

```
...
float *p, x;
p = &x;
função(p); // passagem do ponteiro com o endereço de x.
...
```

Programa Exemplo: O arquivo `e0806.cpp` contém um programa que mostra a passagem de endereço para uma função usando um ponteiro. Observe a sintaxe alternativa para a função `scanf()`!

8.5 Ponteiros, vetores e *strings*

8.5.1 Ponteiros e vetores

No capitulo 7 foi mostrado que na passagem de vetores para funções especifica-se apenas o nome do vetor e que modificações nos elementos do vetor dentro da função chamada alteram os valores do vetor no programa chamador (seção 7.4). Isto se deve ao fato de que, na linguagem C, **vetores** são intimamente relacionados a **ponteiros**.

Em C, o **nome** de um vetor é tratado como o **endereço** de seu primeiro elemento. Assim ao se passar o nome de um vetor para uma função está se passando o endereço do primeiro elemento de um conjunto de endereços de memória.

Por exemplo, se `vet` é um vetor, então `vet` e `&vet[0]` representam o mesmo **endereço**. E mais, podemos acessar o endereço de qualquer elemento do vetor do seguinte modo: `&vet[i]` e equivalente a `(vet + i)`. Aqui deve-se ressaltar que `(vet + i)` não representa uma **adição** aritmética normal mas o **endereço** do *i-ésimo* elemento do vetor `vet` (endereço contado a partir do endereço inicial `vet[0]`).

Do mesmo modo que se pode acessar o **endereço** de cada elemento do vetor por ponteiros, também se pode acessar o **valor** de cada elemento usando ponteiros. Assim `vet[i]` é equivalente a `*(vet + i)`. Aqui se usa o operador conteúdo (`*`) aplicado ao endereço do *i-ésimo* elemento do vetor `vet`.

Programa Exemplo: O arquivo `e0807.cpp` contém um programa que mostra a equivalência entre ponteiros e vetores.

8.5.2 Ponteiros e *strings*

Como dissemos, vagamente na seção 2.3.4, uma *string* é um *conjunto ordenado de caracteres*. Podemos, agora, dizer muito mais: Em C, uma *string* é um **vetor unidimensional** de elementos caracteres ASCII, sendo o ultimo destes elementos o caracter especial `'\0'`.

Sintaxe: As duas maneiras mais comuns de declararmos uma *string* são:

```
char nome[tam];
```

ou

```
char *nome;
```

onde:

nome é o nome do vetor de caracteres e

tam seu tamanho.

Observe que sendo um vetor, uma *string* pode ser declarada também como um ponteiro. Alias a segunda declaração representa justamente isto. Sabendo isto podemos realizar uma grande variedade de manipulações com *strings* e caracteres. Existe uma biblioteca padrão C chamada `string.h` que providencia algumas funções de manipulação de *strings* muito úteis.

Programa Exemplo: O arquivo `e0808.cpp` contém um programa que mostra algumas operações usando-se strings (vetores e ponteiros).

8.6 Alocação Dinâmica de Memória

Os elementos de um vetor são armazenados **seqüencialmente** na memória do computador. Na declaração de um vetor, (por exemplo: `int vet[10]`) é dito ao processador reservar (**alocar**) um certo numero de blocos de memória para armazenamento dos elementos do vetor. Porem, neste modo de declaração, não se pode alocar um numero variável de elementos (veja seção 7.3.2).

A linguagem C permite alocar dinamicamente (em tempo de execução), blocos de memória usando ponteiros. Dada a intima relação entre ponteiros e vetores, isto significa que podemos declarar dinamicamente vetores de tamanho variável. Isto é desejável caso queiramos poupar memória, isto é não reservar mais memória que o necessário para o armazenamento de dados.

Para a alocação de memória usamos a função `malloc()`(*memory allocation*) da biblioteca `alloc.h`. A função `malloc()` reserva, seqüencialmente, um certo numero de blocos de memória e retorna, para um ponteiro, o endereço do primeiro bloco reservado.

Sintaxe: A sintaxe geral usada para a alocação dinâmica é a seguinte:

```
pont = (tipo *)malloc(tam);
```

onde:

pont é o nome do ponteiro que recebe o endereço do espaço de memória alocado.

tipo é o tipo do endereço apontado (tipo do ponteiro).

tam é o tamanho do espaço alocado: numero de *bytes*.

A sintaxe seguinte, porem, é mais clara:

```
pont = (tipo*)malloc(num*sizeof(tipo));
```

onde:

num é o numero de elementos que queremos poder armazenar no espaço alocado.

Exemplo: Se queremos declarar um vetor chamado `vet`, tipo `int`, com `num` elementos podemos usar o trecho abaixo:

```
...
int *vet;  // declaração do ponteiro
vet = (int*)malloc(num*2); // alocação de num blocos de 2 bytes
```

```
...
```

ou ainda

```
...
int *vet; // declaração do ponteiro
vet = (int*) malloc(num * sizeof(int));
...
```

Caso não seja possível alocar o espaço requisitado a função `malloc()` retorna a constante simbólica `NULL`.

Sintaxe: Para liberar (desalocar) o espaço de memória se usa a função `free()`, cuja sintaxe é a seguinte:

`free(`*`pont`*`);`

onde:

`pont` é o nome do ponteiro que contem o endereço do inicio do espaço de memória reservado.

Programa Exemplo: O arquivo `e0809.cpp` contém um programa que mostra como se utiliza a Alocação Dinâmica de Memória.

8.7 Ponteiros para Funções

Até agora usamos ponteiros para apontar para endereços de memória onde se encontravam as variáveis (dados). Algumas vezes é necessário apontar para funções, isto é, apontar para o endereço de memória que contem o inicio das instruções de uma função. Quando assim procedemos, dizemos que usaremos *ponteiros para funções*.

8.7.1 Ponteiros como chamada de função.

Um uso de ponteiros para funções é passar uma função como **argumento** de outra função. Mas também se pode usar ponteiros para funções ao invés de funções nas chamadas normais de funções.

Sintaxe: A sintaxe de declaração de ponteiro para funções é a seguinte:

```
tipo_r (*nome_p)(lista);
```

onde:

tipo_r é o tipo de retorno da função apontada.

nome_p é o nome do ponteiro que apontara para a função.

lista é a lista de argumentos da função.

Exemplo: Suponha que temos uma função é declarada como:

```
float fun(int a, int b){
  ...
}
```

o ponteiro correspondente será:

```
float (*pt)(int,int);
```

Observe que o ponteiro para função **deve ser** declarado entre parênteses. Observe também que o ponteiro e a função retornam o mesmo tipo de dado e que tem os mesmos argumentos.

Sintaxe: Para atribuirmos o endereço de uma função para um ponteiro usamos a seguinte sintaxe:

```
pont = &função;
```

onde:

pont é o nome do ponteiro.

função é o nome da função.

Se um ponteiro contem o endereço de uma função, ele pode ser usado no lugar da chamada da função.

Exemplo: o trecho de programa abaixo usa um ponteiro para chamar uma função:

```
float fun(int a,int b){
  ...
}
void main(void){
  float temp;
  float (*pt)(int,int);
```

```
  pt = &fun;
  temp = (*pt)(10,20); // eqüivale a: temp = fun(10,20);
  ...
}
```

Programa Exemplo: O arquivo `e0810.cpp` contém um programa que mostra como se utiliza o ponteiro para função.

8.7.2 Passando uma função como argumento de outra função.

Outra utilização de ponteiros para funções é na passagem de uma **função** como **argumento** para outra função. Para que isso ocorra necessitamos:

- Na declaração da **função a ser passada**:

i) Nada de especial, apenas a definição normal:

```
tipo nome_p(lista){
  ...
}
```

Exemplo:

```
float soma(float a,float b){
  ...
}
```

- Na **função receptora**:

i) Declarar o **ponteiro** que recebe a **função passada** na lista de argumentos:

```
tipo nome_r(..., tipo (*pt)(lista), ...){
```

Exemplo:

```
float grad(float x, float y, float (*p)(float,float)){
```

ii) Usar o ponteiro para funções nas chamadas da função passada:

```
var = (*pt)(lista);
```

Exemplo:

```
valor = (*p)(x,y);
```

- Na função principal:

i) Passar o nome da função chamada para a função receptora:

```
var = nome_g(... , nome_p , ...);
```

Exemplo:

```
g = grad(x,y,soma);
```

Programa Exemplo: O arquivo `e0811.cpp` contém um programa que mostra como se utiliza ponteiros na passagem de funções.