



Aspectos históricos, C é crucial para o processamento de sinal em sistemas de comunicação, como um software escrito em C é transformado em um programa executável, utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c, tipos de variáveis básicas, constantes, operadores aritméticos, operadores de incremento e decremento, precedência de operadores, operadores relacionais e lógicos, operador de atribuição, conversão de tipos de dados (cast), introdução à strings, introdução à funções, controle do fluxo do programa (if-else, switch, case, for, while, do-while, break, goto e continue) , vetores, strings e matrizes.

**Centro de Tecnologia – Departamento de Eletrônica e Computação**

**Engenharia de Telecomunicações**

**O BÁSICO DE LINGUAGEM C PARA PROCESSAMENTO DE SINAL**

**Prof. Fernando DeCastro**

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

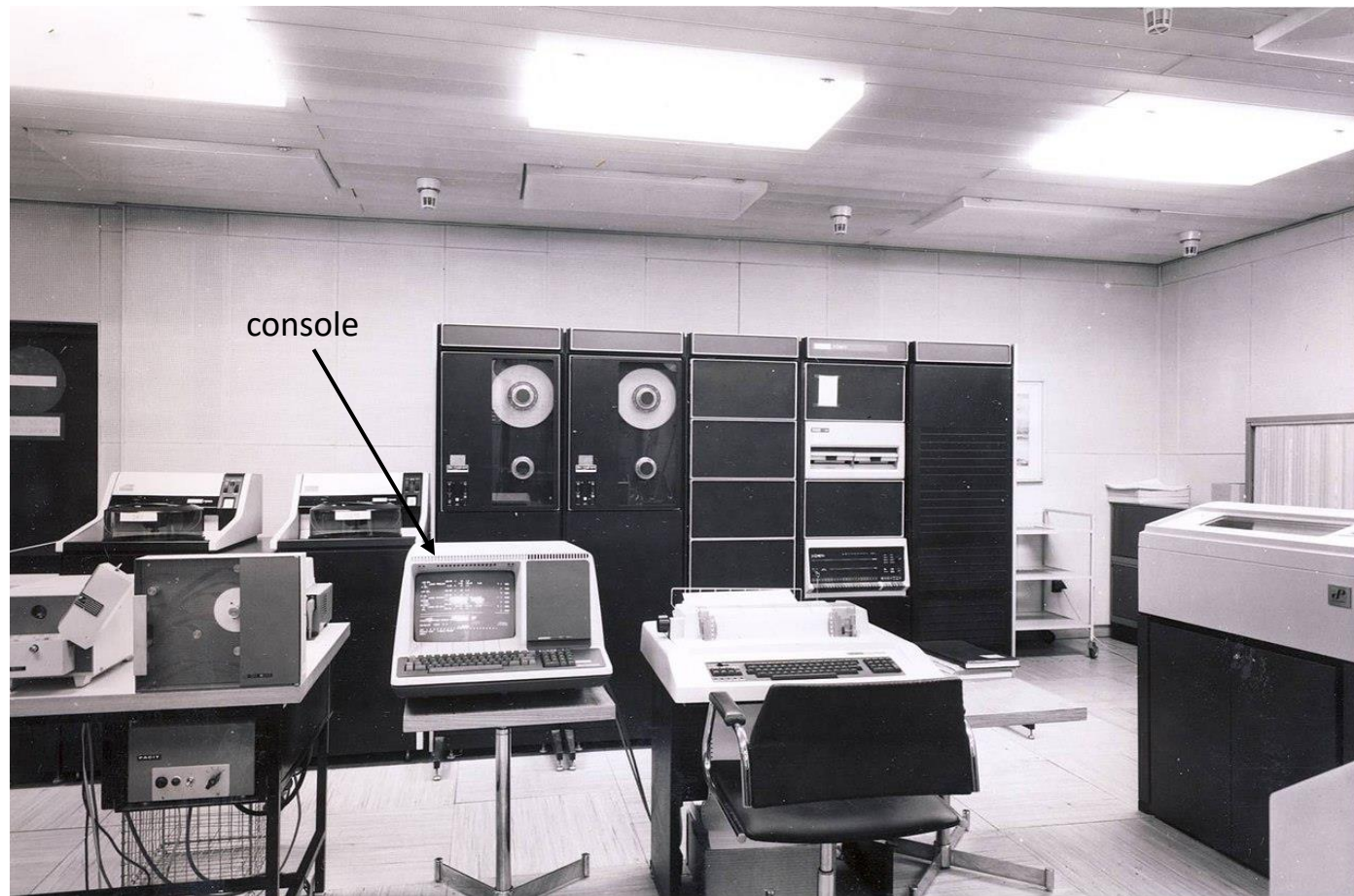
## Aspectos históricos

A linguagem de programação C foi criada por Dennis Ritchie no Bell Labs (<https://www.bell-labs.com/usr/dmr/www/chist.html>) no início dos anos 70 como uma versão melhorada da linguagem B de Ken Thompson.

A “bíblia” original descrevendo a sintaxe dos comandos da linguagem C e as estruturas de programação da linguagem foi publicada em 1978, com autoria de Ken Thompson e Dennis Ritchie, frequentemente referida como “K&R-C” em homenagem aos dois autores do livro (a 2ª edição do “K&R-C” está disponível no link <https://www.fccdecastro.com.br/CursoC&C++/CProgrammingLanguage2nd%20-%20K&R.pdf>). Posteriormente, em 1983, o *American National Standards Institute* iniciou a padronização da linguagem C ([https://en.wikipedia.org/wiki/ANSI\\_C](https://en.wikipedia.org/wiki/ANSI_C)).

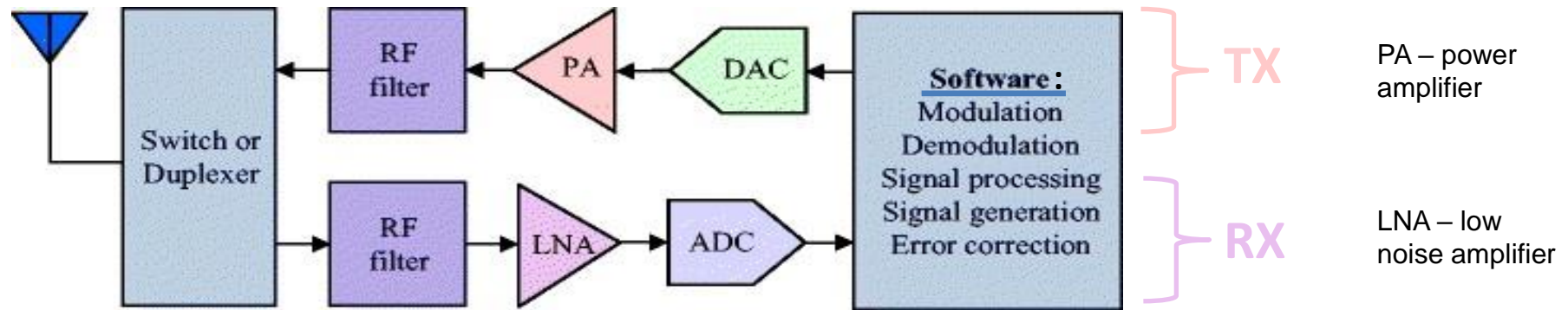
Alguns autores postulam que o evento que deu origem ao nascimento da linguagem C no início dos anos 70 ocorreu no momento em que Dennis Ritchie executou o programa `hello.c` (vide capa deste capítulo) em uma máquina DEC PDP-11/70 (vide foto ao lado) tendo resultado na tela do console do PDP-11/70 a frase “Hello world”.

Vide DEC PDP-11/70 em <https://en.wikipedia.org/wiki/PDP-11>.



## C é crucial para o processamento de sinal em sistemas de comunicação

O **escopo** deste curso é o **C para processamento de sinal em sistemas de comunicação**. Este escopo decorre da tendência na arquitetura de sistemas de comunicação no sentido de maximizar as funcionalidades do sistema implementadas em software, o que minimiza o uso de hardware específico e dedicado. Um exemplo abrangente desta tendência generalizada é a abordagem adotada em SDRs (*software defined radios* – ver [https://en.wikipedia.org/wiki/Software-defined\\_radio](https://en.wikipedia.org/wiki/Software-defined_radio) ) em que tanto o TX como o RX são, em sua maior parte, implementados via software , conforme diagrama simplificado abaixo.



Observe no diagrama acima que o bloco que implementa as funcionalidades em software precisa executar todo o rol de operações, processos e algoritmos em tempo real. Portanto, é necessário que o software (e o hardware) seja adequado para a execução em tempo real de programas. E, neste contexto, não há muitas outras linguagens de alto nível além do C compatíveis com a implementação de programas executáveis em tempo real. Esta compatibilidade com a implementação de operações, processos e algoritmos em tempo real decorre do fato de que o C foi originalmente concebido para:

- (I) Prover acesso de baixo nível à registradores e à área de memória do hardware utilizando construções de linguagem que mapeiam eficientemente as instruções do hardware (ver [https://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_(programming_language)) )
- (II) Minimizar a complexidade computacional das operações do *runtime system* (ver [https://en.wikipedia.org/wiki/Runtime\\_system](https://en.wikipedia.org/wiki/Runtime_system) ), sistema que faz o controle da pilha (*stack*), área de memória alocada dinamicamente (*heap*), *threads* ([https://en.wikipedia.org/wiki/Thread\\_computing](https://en.wikipedia.org/wiki/Thread_computing) ) e demais processos básicos necessários à execução de um software em uma CPU.

Um exemplo adicional desta tendência de uso do C é a adoção do C como coadjuvante no *design flow* de sistemas de comunicação implementados em ASIC (*Application-Specific Integrated Circuit*) e/ou em FPGA (*Field-Programmable Gate Array*), ou como linguagem descritiva principal no *design* de sistemas baseados em SoC (*System On Chip*) através do uso do C no *High Level Synthesis* do sistema (ver slides 25 e 26 de [https://www.fccdecastro.com.br/pdf/SCD1\\_Capl.pdf](https://www.fccdecastro.com.br/pdf/SCD1_Capl.pdf) ).

## Como um software escrito em C é transformado em um programa executável

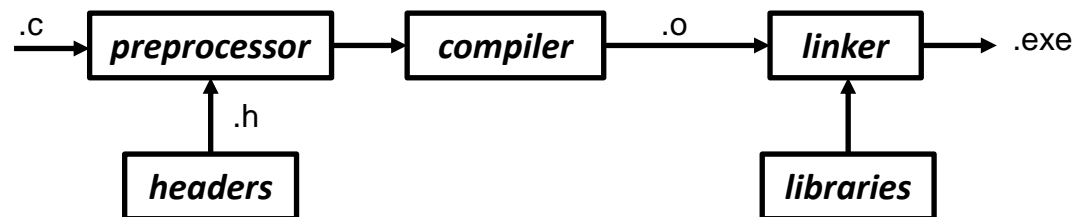
Um programa em linguagem C é descrito em um arquivo texto com extensão `.c`. O conteúdo deste arquivo `.c` é formado por um código descrito por caracteres ASCII imprimíveis (ver tabela em <https://pt.wikipedia.org/wiki/ASCII> ) e é denominado **código fonte** . O código fonte é um código inteligível a uma pessoa que conhece a linguagem C quando esta pessoa abre o arquivo `.c` em um editor de texto, como, por exemplo, o Bloco de Notas (notepad.exe) ou o Microsoft Word. Um exemplo de conteúdo de um arquivo `.c` é o código fonte na capa deste capítulo.

Um código fonte `.c` não pode ser executado diretamente em um computador, microprocessador, microcontrolador, etc ... É necessário um processo de 3 etapas para transformar o arquivo do código-fonte descrito no arquivo `.c` em um arquivo executável `.exe`. Essas três etapas são: Pré-processamento, compilação e linkagem, que são respectivamente implementadas pelo **preprocessor** (pré-processador), **compiler** (compilador) e **linker**, conforme diagrama abaixo.

**preprocessor** - Processa diretivas (comandos que começam com um caractere #) que modificam o código-fonte antes dele ser compilado de modo a orientar o compilador sobre particularidades que se deseja serem obedecidas no processo de compilação. Diretivas são encontradas em [https://www.techonthenet.com/c\\_language/directives/index.php](https://www.techonthenet.com/c_language/directives/index.php). Adicionalmente, o pré-processador consulta um conjunto de arquivos cabeçalho (*headers*), que são arquivos com extensão `.h` que contém declarações e definições de macros na **sintaxe da linguagem C** e que servem como pre-definições a serem usadas no *compiler* (ver principais arquivos `.h` em [https://www.techonthenet.com/c\\_language/directives/include.php](https://www.techonthenet.com/c_language/directives/include.php) ). **Sintaxe de uma linguagem de programação** é o conjunto de regras que define as combinações de caracteres ASCII imprimíveis, combinações que são consideradas declarações ou expressões corretamente estruturadas nessa linguagem.

**compiler** – Converte o código-fonte modificado em **código objeto** binário. O arquivo `.o` resultante na saída do compilador ainda não é um arquivo executável.

**linker** – Combina o código objeto com o código de bibliotecas de suporte (*libraries*) para aplicações específicas. Biblioteca C padrão (**standard C library**), biblioteca de funções gráficas, biblioteca de operações com números em ponto flutuante, biblioteca para operações com matrizes são apenas alguns exemplos.





## Como um software escrito em C é transformado em um programa executável

Em geral, as 3 etapas realizadas respectivamente pelo pré-processador, compilador e linker são realizadas por um único aplicativo, denominado de **ambiente de desenvolvimento integrado** (IDE – *integrated development ambient*).

A biblioteca C padrão (*standard C library*) referida no slide anterior é usualmente incluída em qualquer IDE disponível comercialmente ou mesmo em IDEs *free*. Uma descrição do rol de funções disponíveis na biblioteca C padrão e qual a utilidade/aplicação de cada uma destas funções pode ser independentemente encontrada nos 3 links abaixo:

<https://www.fccdecastro.com.br/CursoC&C++/TheStandardCLibrary%20-%20Plaucer.pdf>

<https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf>

<https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceManual.pdf>

Embora o conceito de função já tenha sido discutido na disciplina “Algoritmos e Programação - UFSM00013”, é instrutivo revisar funções em linguagem C na seção 217 (pag 108) até a seção 225 (pag 114) de [fccdecastro.com.br/CursoC&C++/Programando em C C++ A Biblia - LARS KLANDER, KRIS JAMSA.pdf](https://www.fccdecastro.com.br/CursoC&C++/Programando em C C++ A Biblia - LARS KLANDER, KRIS JAMSA.pdf).

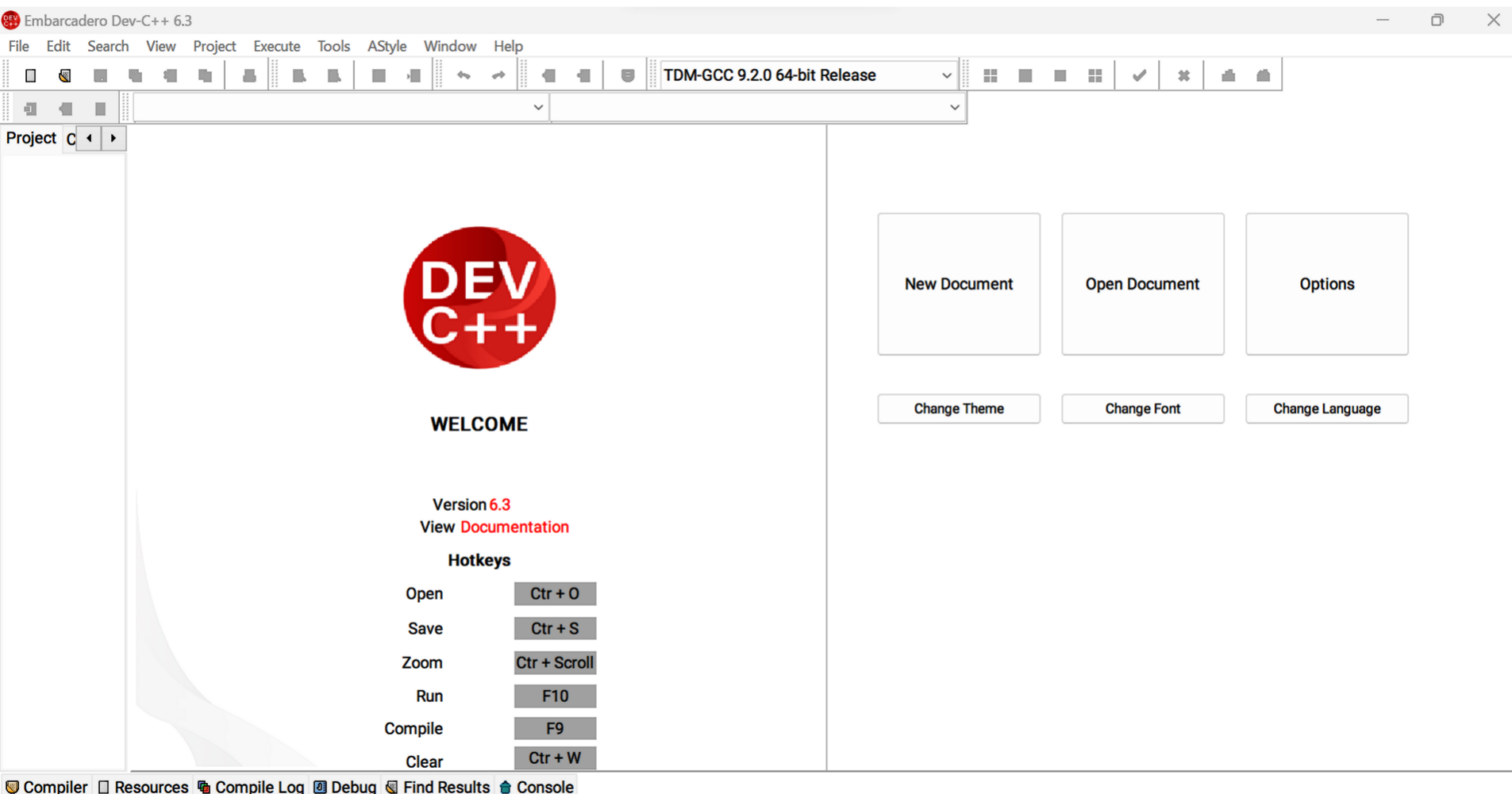
O **protótipo** de cada função de uma biblioteca encontram-se definidos em *headers .h*, conforme discutido no slide anterior. O protótipo de uma função em C é uma declaração preliminar da função que especifica o tipo de dado que a função retorna, o nome e o número e tipos de dados de seus argumentos de entrada (= parâmetros da função). O protótipo atua como um “contrato” entre a definição da função e a instância do programa que chama a função, fornecendo informações sobre as entradas e saídas esperadas da função.

Neste estudo da linguagem C vamos adotar o IDE denominado **DevC++** para gerar os arquivos executáveis correspondentes aos diversos códigos fonte que analisaremos e usaremos na implementação de processos e algoritmos de interesse em engenharia (ver <https://www.embarcadero.com/br/free-tools/dev-cpp>).

O DevC++ é *free* e seu *download* pode ser efetuado a partir do link <https://www.embarcadero.com/br/free-tools/dev-cpp/free-download>.

# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Uma vez instalado o DevC++ em um PC-Windows, a interface com o usuário do DevC++ apresenta o seguinte visual:



## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

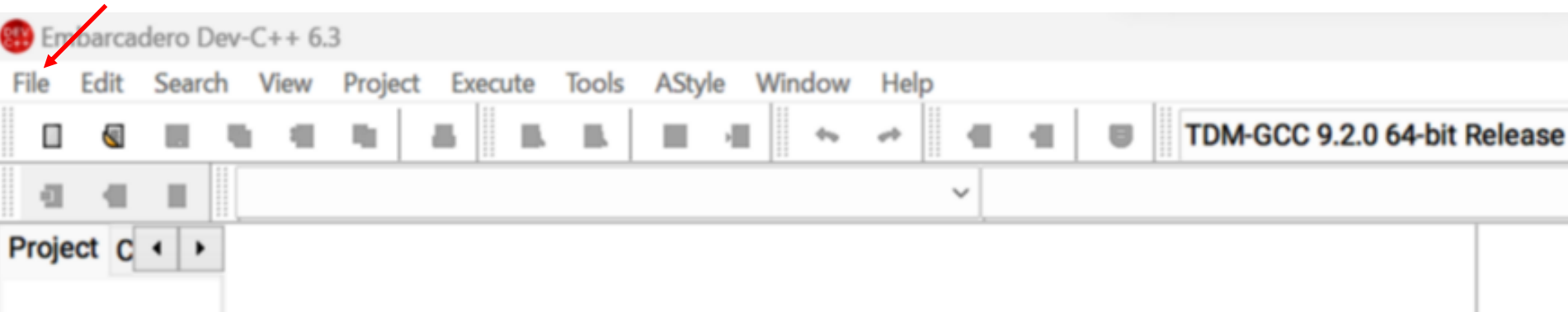
Vamos supor que queiramos que o DevC++ converta o código fonte “PesMetrosV1.c”, disponível em <https://www.fccdecastro.com.br/CursoC&C++/C/PesMetrosV1.c>, código fonte que é descrito abaixo:

```
1  /* conversao de pes para metros V1 - NumPes eh declarado como uma
2  variavel de numeros inteiros (int) e ValorMetros eh declarado como
3  uma variavel de numeros em ponto flutuante (float) */
4
5  #include <stdio.h> /* inclui arquivo stdio.h c/ as definicoes
6  para a biblioteca de funcoes padrao "standard I/O" */
7
8  int main() /* funcao main() - funcao principal atraves da qual
9  todo programa em C comeca a ser executado. A main() retorna um valor
10 inteiro (int) para o sistema operacional do PC. Este valor de retorno
11 foi convencionado ser "0" em caso de nao ocorrer erro na execucao, ou,
12 em caso de ocorrer algum erro, retorna um valor inteiro correspondente
13 ao tipo de erro ocorrido */
14 {
15     int NumPes; /* declaracao da variavel inteira NumPes */
16     float ValorMetros; /* declaracao da variavel em ponto flutuante ValorMetros */
17
18     printf("Informe o numero de pes: "); /* imprime na tela do console o que estah estre aspas */
19     scanf("%d", &NumPes); /* Le NumPes do teclado. Nota: & -> operador "endereco na memoria de" */
20     ValorMetros = NumPes * 0.3048; /* conversao de pes para metros */
21     printf("%d pes = %f metros\n", NumPes, ValorMetros); /* imprime na tela do console as variaveis
22                                                         NumPes e ValorMetros */
23     return 0; /* sucesso na execucao - nao houve erro - retorna o valor 0 ao sistema operacional */
24 }
```

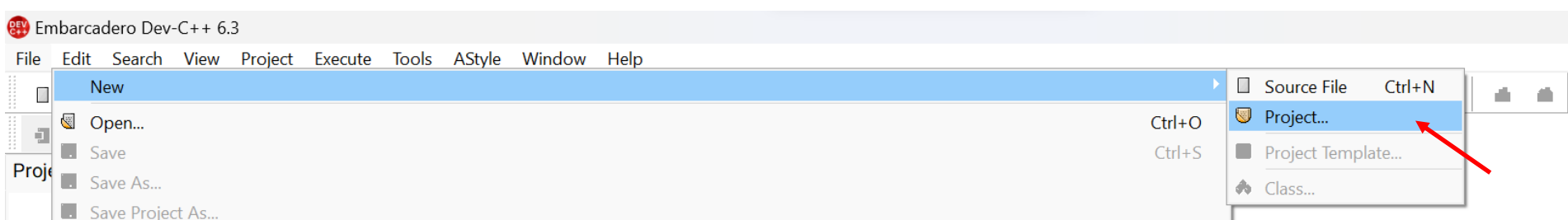
Note no código fonte acima que cada comentário explicativo sobre o código encontra-se delimitado entre o par de caracteres `/*` (antes do início do comentário) e o par de caracteres `*/` (após o fim do comentário). Alternativamente, o par de caracteres `//` colocado no início de uma linha faz o compilador considerar como comentário todo o texto até o final da linha. **Dado o grau de liberdade que a linguagem C concede a um programador que dela faça uso, é imperativo que todo código fonte C seja suficientemente e coerentemente comentado de modo a permitir que qualquer outro programador tenha condições de entender o que está sendo codificado.**

## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Para converter o código fonte “PesMetrosV1.c” em um arquivo executável , primeiramente vamos criar uma pasta de trabalho , que , neste exemplo será “C:\Users\fccde\DATA\UFSM\UFSM 2024\_I\Linguagem C para processamento de sinais\C\PesMetros” e, a seguir , vamos copiar o fonte “PesMetrosV1.c” para esta pasta. A seguir, vamos criar um *Project* no DevC++, com o nome que queremos p/ o arquivo .exe a ser gerado. Vamos chamar este *Project* de “Convpm ”, o que vai gerar o arquivo executável Convpm.exe. Para tanto, primeiramente damos um *left-click* em File no IDE do DevC++ (seta vermelha abaixo):



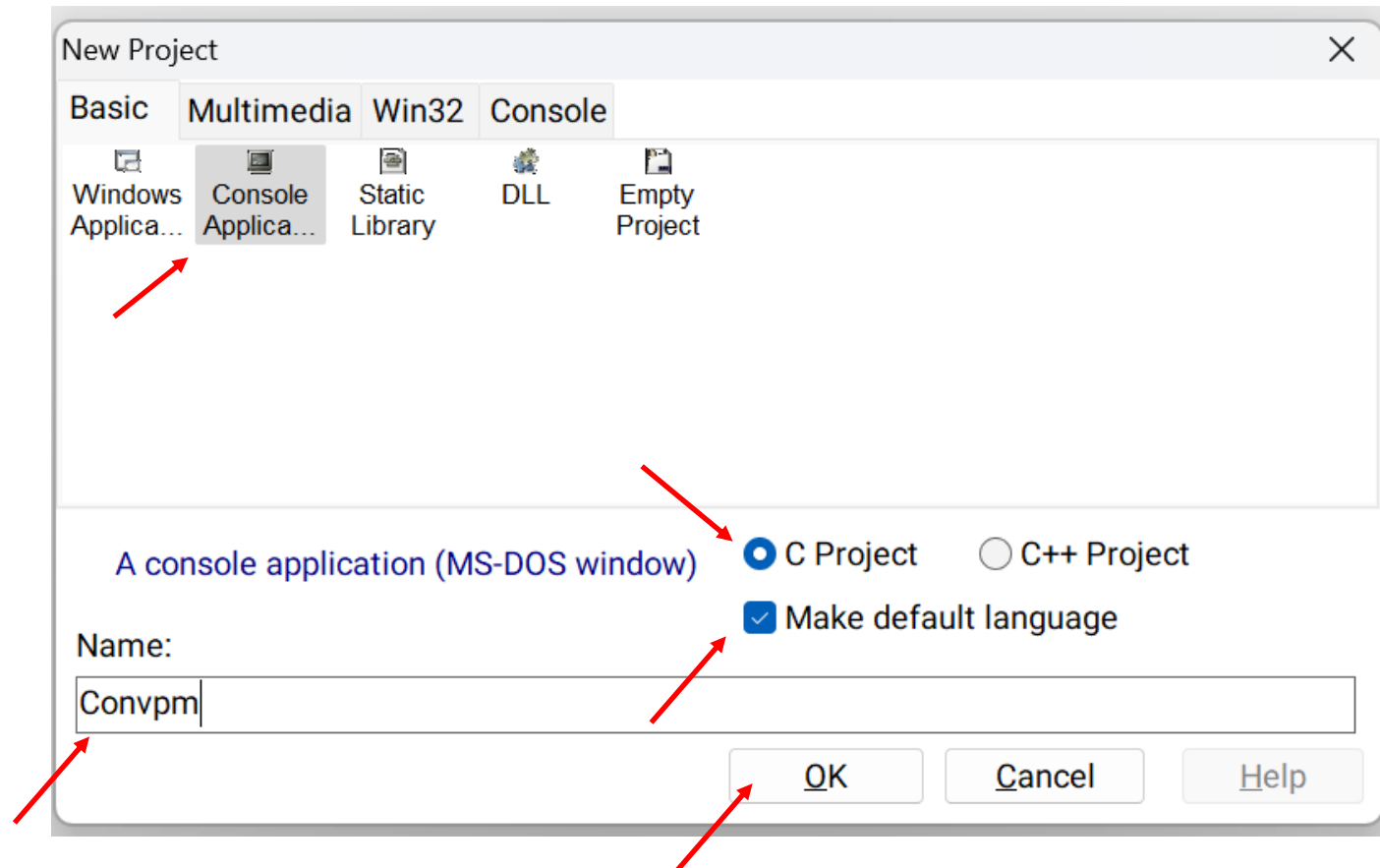
Daí , *left-click* em New-> Project:





## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

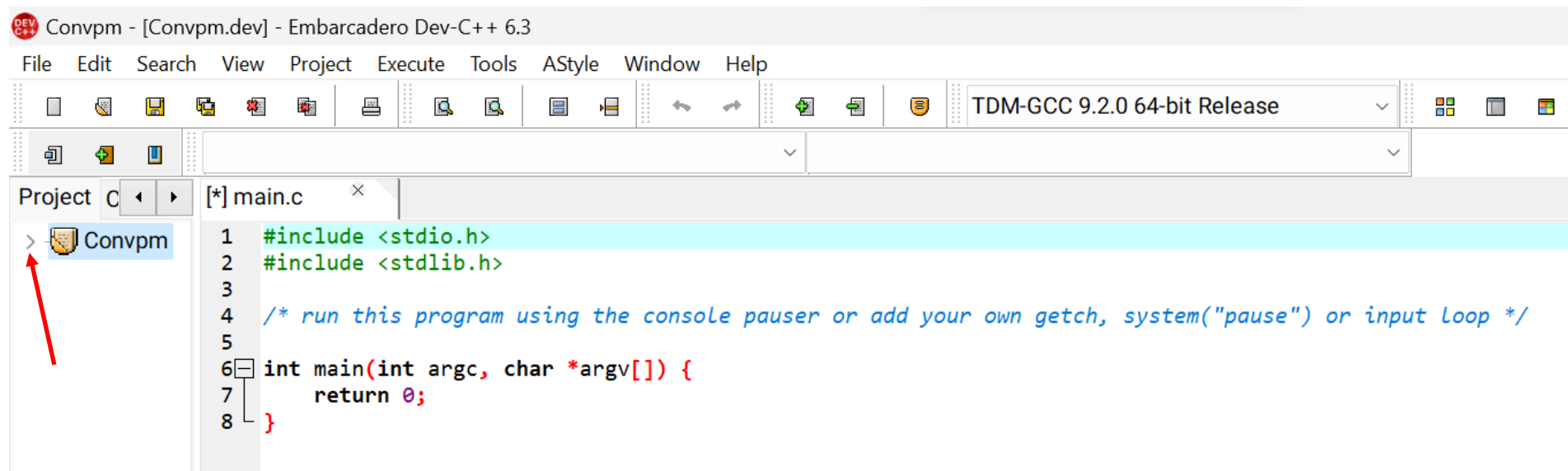
E configuramos a janela de configuração do *Project* com os parâmetros/*checkbox* abaixo (setas vermelhas abaixo) :



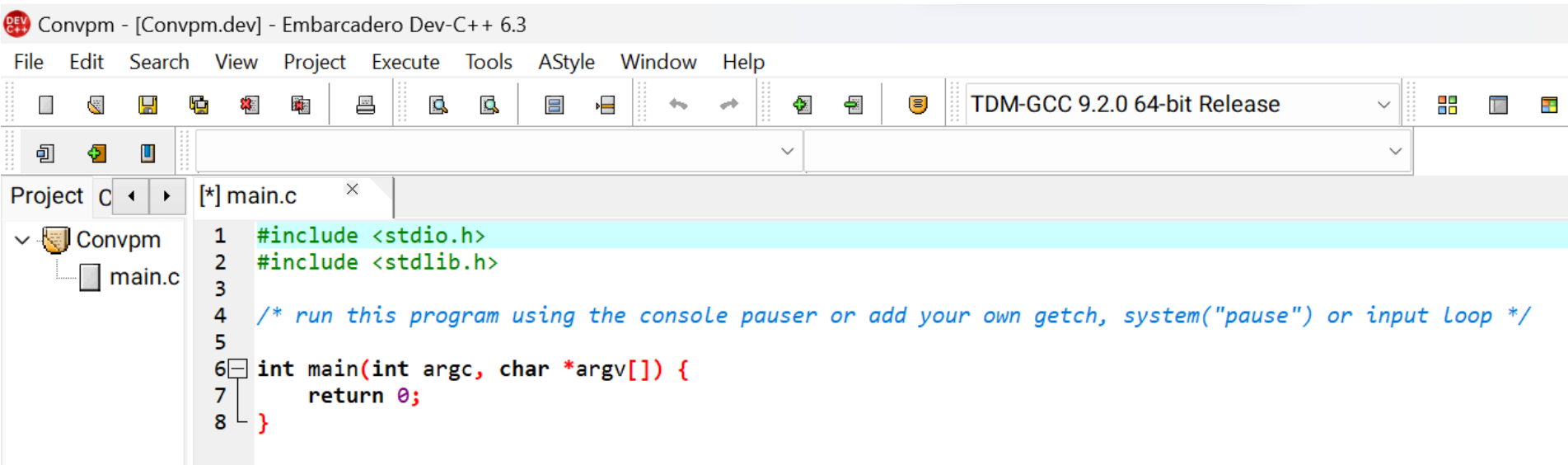
A seguir, *left-click* em “OK” e salvamos o *Project* assim configurado no recém criado arquivo “Convpm.dev”. O arquivo “Convpm.dev” deve ser salvo obrigatoriamente na pasta de trabalho “C:\Users\fccde\DATA\UFSM\UFSM 2024\_I\Linguagem C para processamento de sinais\C\PesMetros”.

# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Um vez salvo o *Project* “Convpm.dev” na pasta de trabalho, a interface com o usuário do DevC++ mostra o seguinte visual:

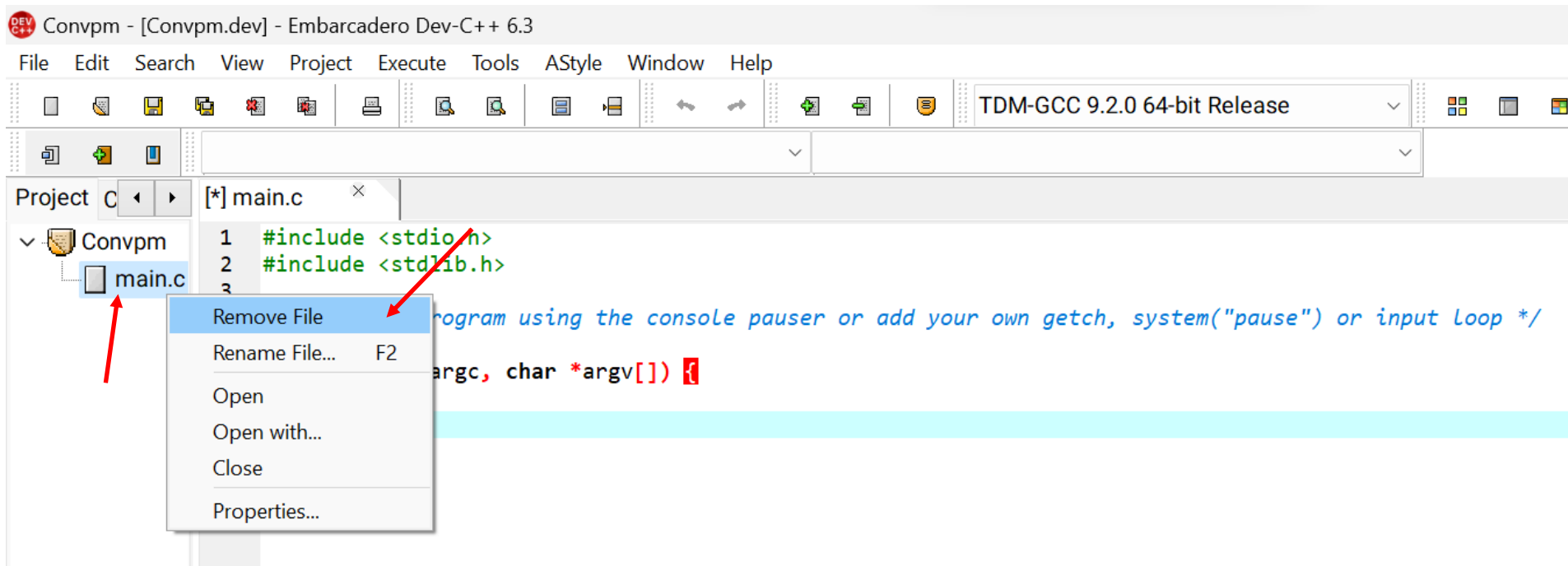


Daí, *left-click* em “>” (seta vermelha acima) :



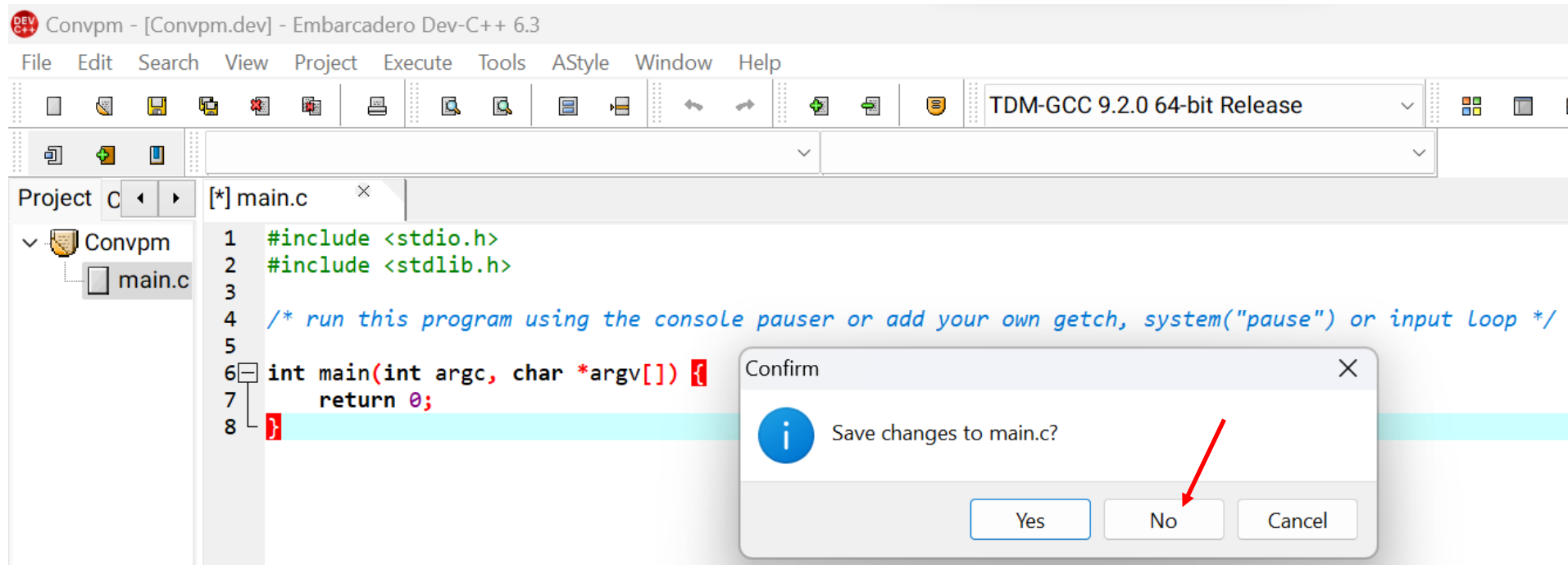
# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

A seguir, *right-click* em “main.c” -> Remove File (setas vermelhas abaixo) :



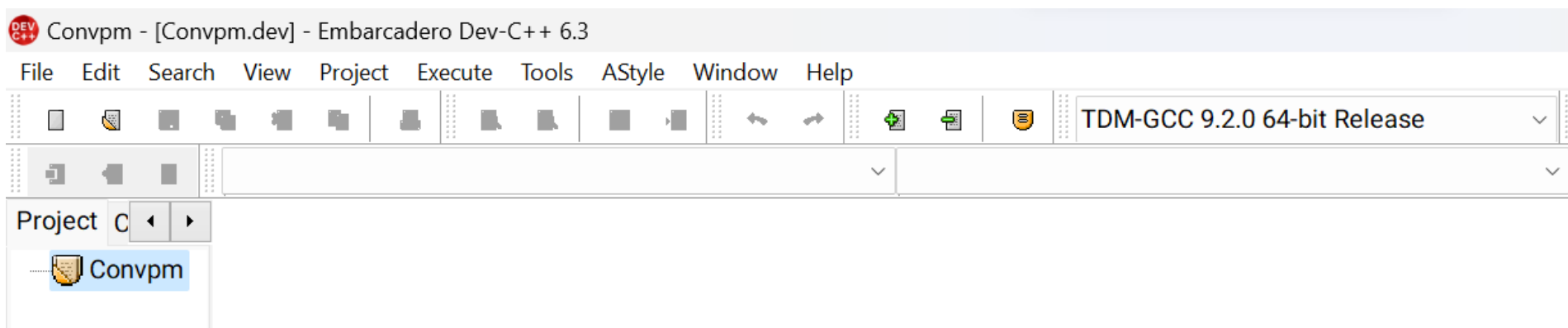
# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

A seguir, não salvar “main.c” (seta vermelha abaixo) :



## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

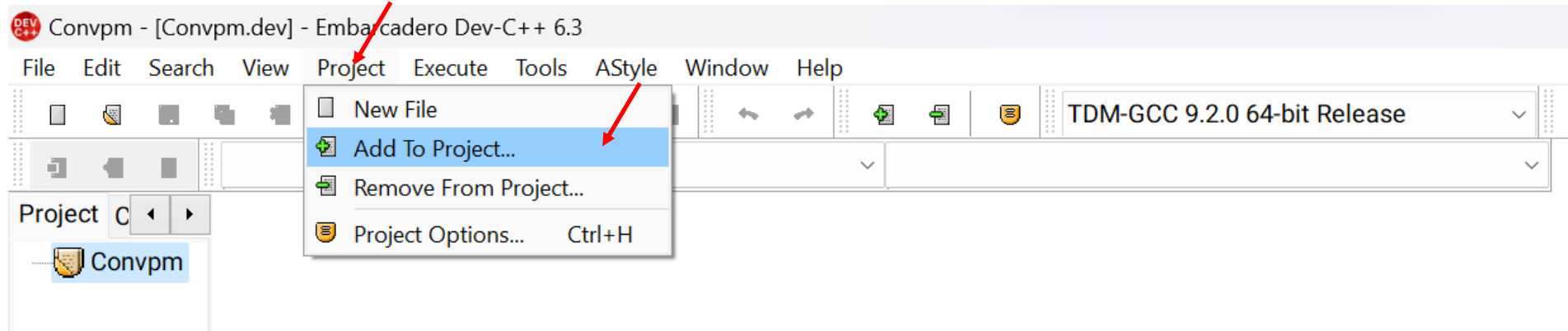
Um vez não salvando “main.c”, a interface com o usuário do DevC++ passa a mostrar o seguinte visual:





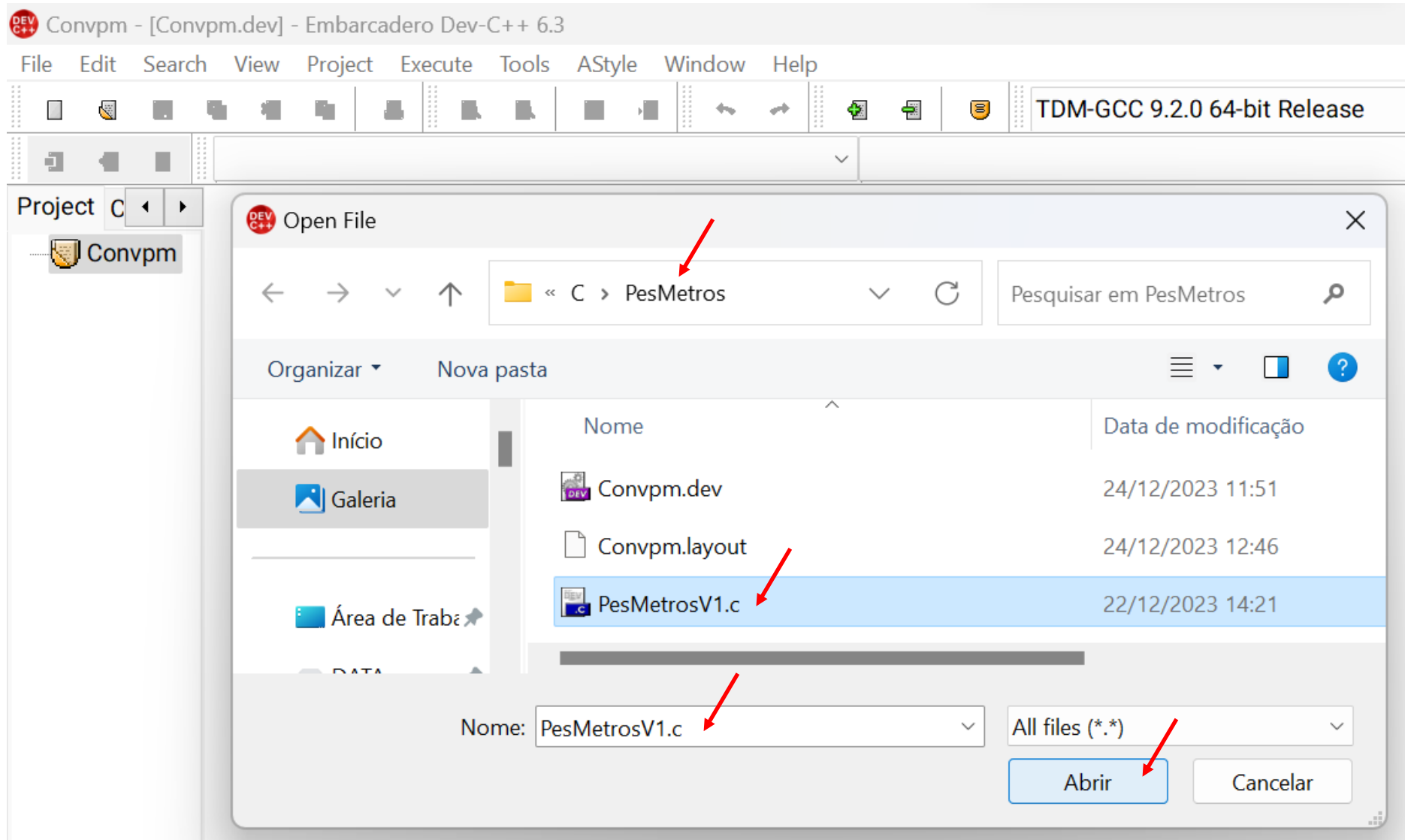
# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

A seguir, *left-click* Project-> Add To Project (setas vermelhas abaixo) :



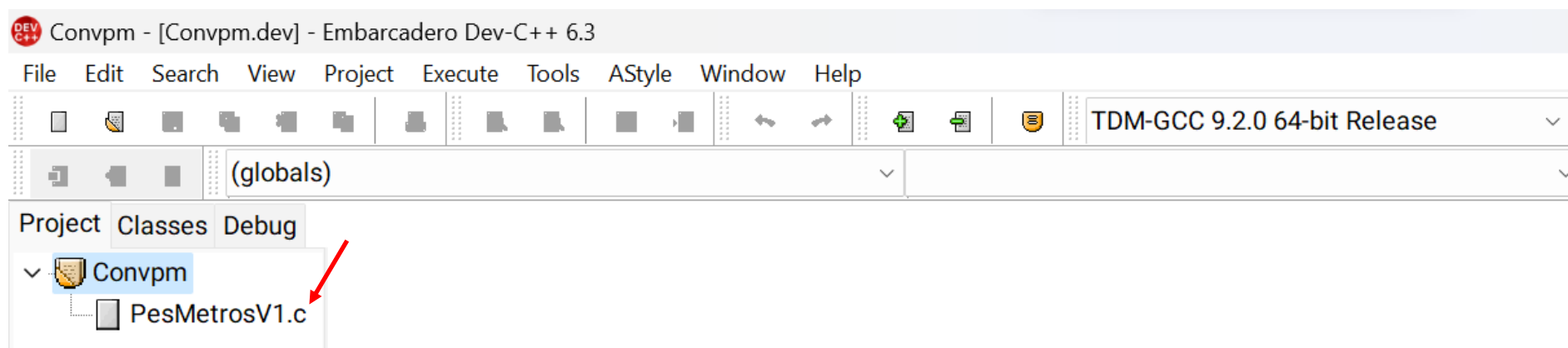
## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

E daí adicionamos ao *Project* o fonte “PesMetrosV1.c” já salvo na pasta de trabalho “PesMetros” (setas vermelhas abaixo) :



## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

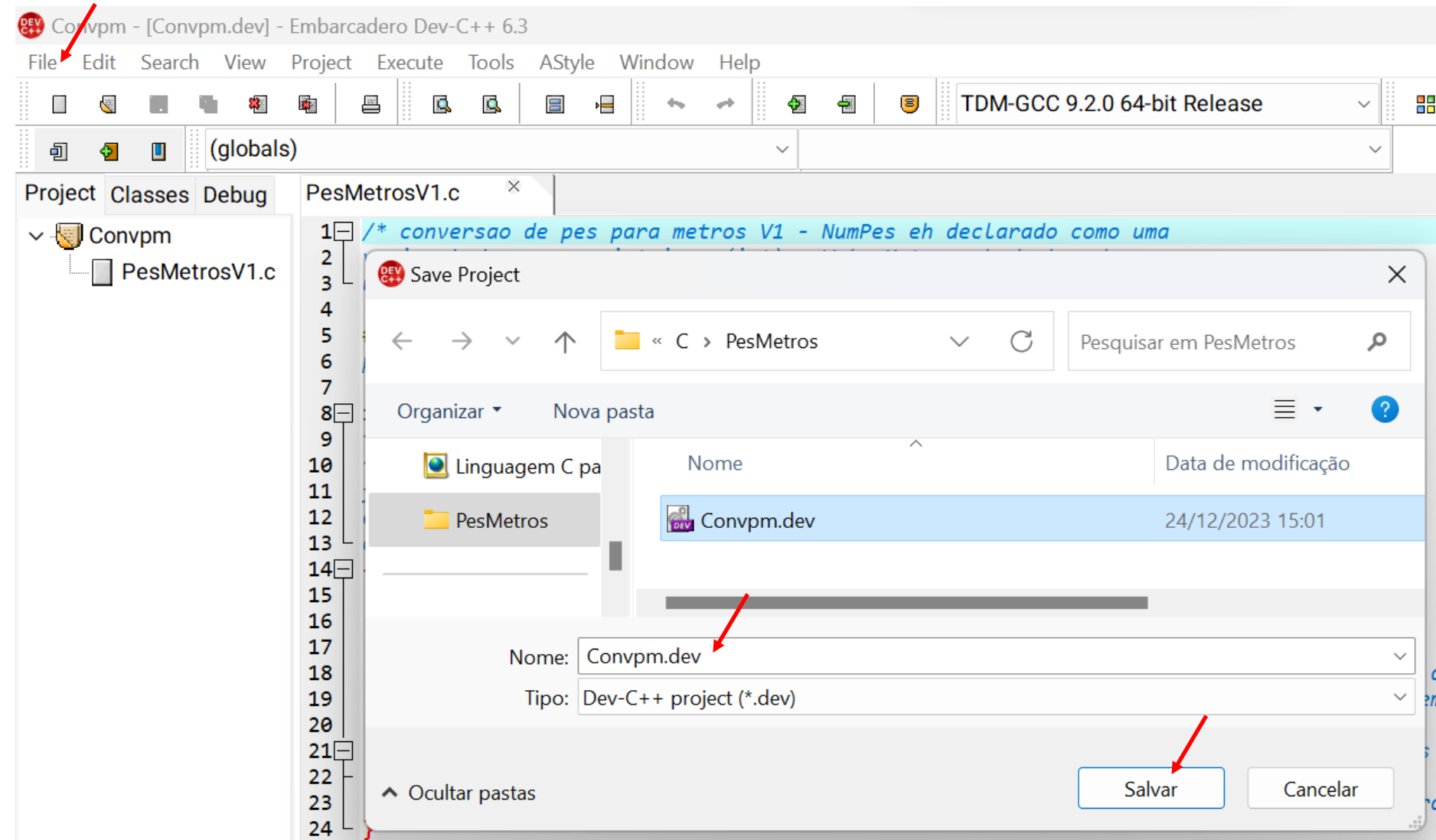
Um vez adicionando ao *Project* o fonte “PesMetrosV1.c”, a interface com o usuário do DevC++ passa a mostrar o seguinte visual:



Se dermos um *left-click* em “PesMetrosV1.c”, a interface com o usuário do DevC++ passa a mostrar o código fonte na janela da direita (ver slide 19).

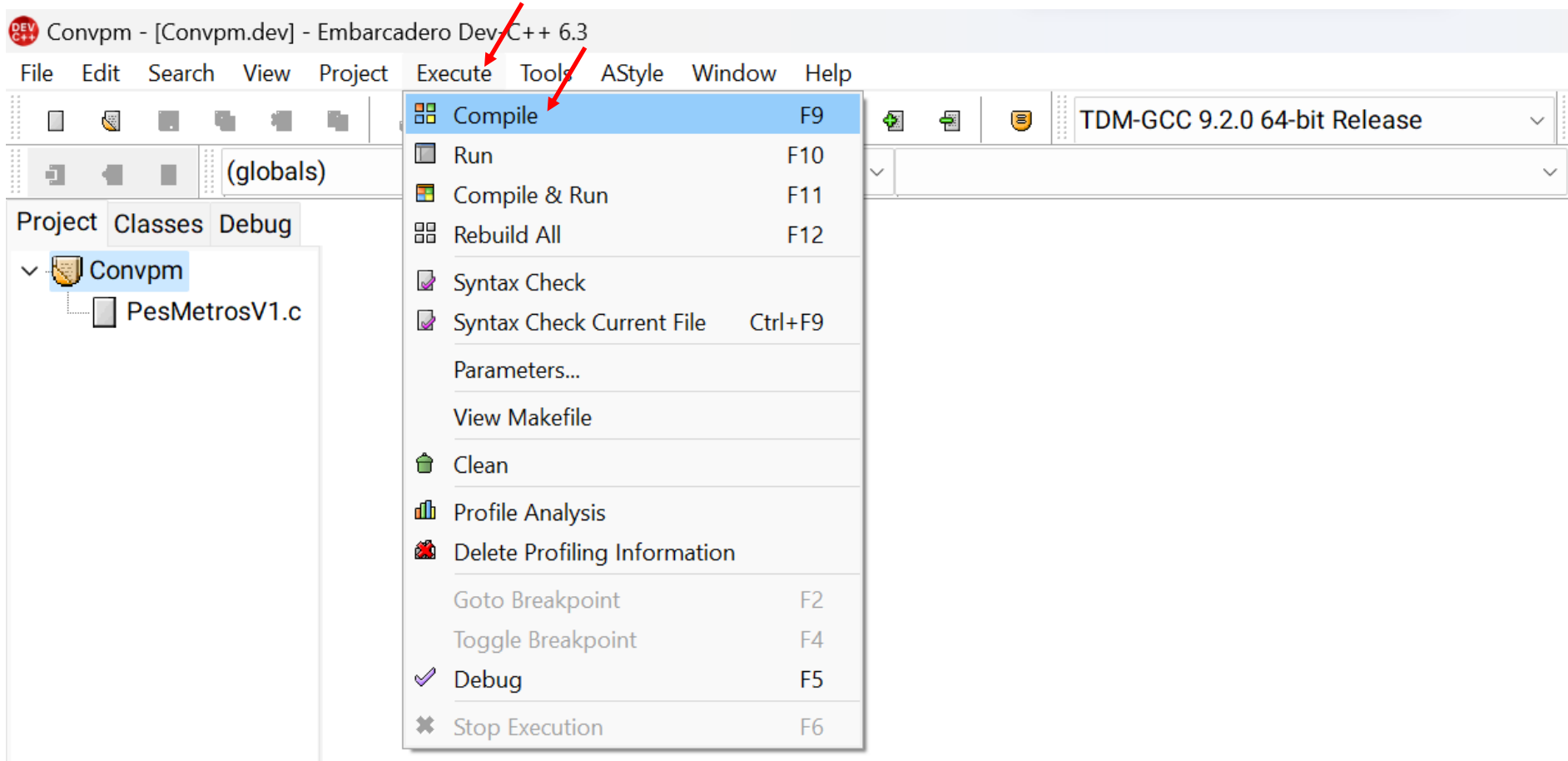
## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Neste ponto a especificação do *Project* está completa e é prudente salvar o mesmo no arquivo “Convpm.dev” na pasta de trabalho. Uma vez salvo o *Project* e encerrada a sessão de trabalho no DevC++, basta clicar no arquivo “Convpm.dev” na pasta de trabalho e o DevC++ automaticamente abrirá o *Project* salvo, iniciando nova sessão de trabalho.



## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Para criar o executável “Convpm.exe” na pasta de trabalho, basta pressionar <F9> ou dar um *left-click* em Execute -> Compile (setas vermelhas abaixo) :





# Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Note que, após pressionar <F9> e o *compiler* e o *linker* terem encerrado seus respectivos processos, a interface com o usuário do DevC++ passa a mostrar a janela “Compile log”, mostrando o resultado do processo de compilação e linkagem “- Errors: 0” e “- Warnings: 0”, indicando sucesso na criação do programa executável Convpm.exe na pasta de trabalho (setas vermelhas abaixo).

Convpm - [Convpm.dev] - Embarcadero Dev-C++ 6.3

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 9.2.0 64-bit Release

(globals)

Project Classes Debug

Convpm

PesMetrosV1.c

```
1 /* conversao de pes para metros V1 - NumPes eh declarado como uma
2 variavel de numeros inteiros (int) e ValorMetros eh declarado como
3 uma variavel de numeros em ponto flutuante (float) */
4
5 #include <stdio.h> /* inclui arquivo stdio.h c/ as definicoes
6 para a biblioteca de funcoes padrao "standard I/O" */
7
8 int main() /* funcao main() - funcao principal atraves da qual
9 todo programa em C começa a ser executado. A main() retorna um valor
10 inteiro (int) para o sistema operacional do PC. Este valor de retorno
11 foi convencionado ser "0" em caso de nao ocorrer erro na execucao, ou,
12 em caso de ocorrer algum erro, retorna um valor inteiro correspondente
13 ao tipo de erro ocorrido */
14 {
15     int NumPes; /* declaracao da variavel inteira NumPes */
16     float ValorMetros; /* declaracao da variavel em ponto flutuante ValorMetros */
17
18     printf("Informe o numero de pes: "); /* imprime na tela do console o que estah estre aspas */
19     scanf("%d", &NumPes); /* Le NumPes do teclado. Nota: & -> operador "endereco na memoria de" */
20     ValorMetros = NumPes * 0.3048; /* conversao de pes para metros */
21     printf("%d pes = %f metros\n", NumPes, ValorMetros); /* imprime na tela do console as variaveis
22                                                         NumPes e ValorMetros */
23     return 0; /* sucesso na execucao - nao houve erro - retorna o valor 0 ao sistema operacional */
24 }
25
```

Sempre declarar as variáveis locais de uma função no início do escopo na função (no caso, a função main()). O escopo de uma variável em C é o bloco ou região do programa onde uma variável é declarada, definida e usada. Fora desta região não podemos acessar a variável e ela é tratada como um identificador não declarado.

Sempre buscar o resultado “- Errors: 0 - Warnings: 0” após o processo de compilação/linkagem. Em caso negativo, o código fonte deve ser alterado até obtermos este resultado.

Abort Compilation

Compiler Resources Compile Log Debug Find Results Console Close

-----

- Errors: 0

- Warnings: 0

- Output Filename: C:\Users\fccde\DATA\UFSM\UFSM 2024\_I\Linguagem C para processamento de sinais\C\PesMetros\Convpm.exe

- Output Size: 322,7841796875 KiB

☐ Shorten compiler path

## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Abaixo é mostrado o conteúdo integral da janela “Compile log” após a compilação/linkagem:

Compiling project changes...

-----

- Project Filename: C:\Users\fccde\DATA\UFSM\UFSM 2024\_\Linguagem C para processamento de sinais\C\PesMetros\Convpm.dev
- Compiler Name: TDM-GCC 9.2.0 64-bit Release

Building makefile...

-----

- Filename: C:\Users\fccde\DATA\UFSM\UFSM 2024\_\Linguagem C para processamento de sinais\C\PesMetros\Makefile.win

Processing makefile...

-----

- Makefile Processor: C:\Program Files (x86)\Embarcadero\Dev-Cpp\TDM-GCC-64\bin\mingw32-make.exe
- Command: mingw32-make.exe -j16 -f "C:\Users\fccde\DATA\UFSM\UFSM 2024\_\Linguagem C para processamento de sinais\C\PesMetros\Makefile.win" all

```
gcc.exe -c PesMetrosV1.c -o PesMetrosV1.o -I"C:/Program Files (x86)/Embarcadero/Dev-Cpp/TDM-GCC-64/include" -I"C:/Program Files (x86)/Embarcadero/Dev-Cpp/TDM-GCC-64/x86_64-w64-mingw32/include" -I"C:/Program Files (x86)/Embarcadero/Dev-Cpp/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/9.2.0/include"
```

```
gcc.exe PesMetrosV1.o -o Convpm.exe -L"C:/Program Files (x86)/Embarcadero/Dev-Cpp/TDM-GCC-64/lib" -L"C:/Program Files (x86)/Embarcadero/Dev-Cpp/TDM-GCC-64/x86_64-w64-mingw32/lib" -static-libgcc
```

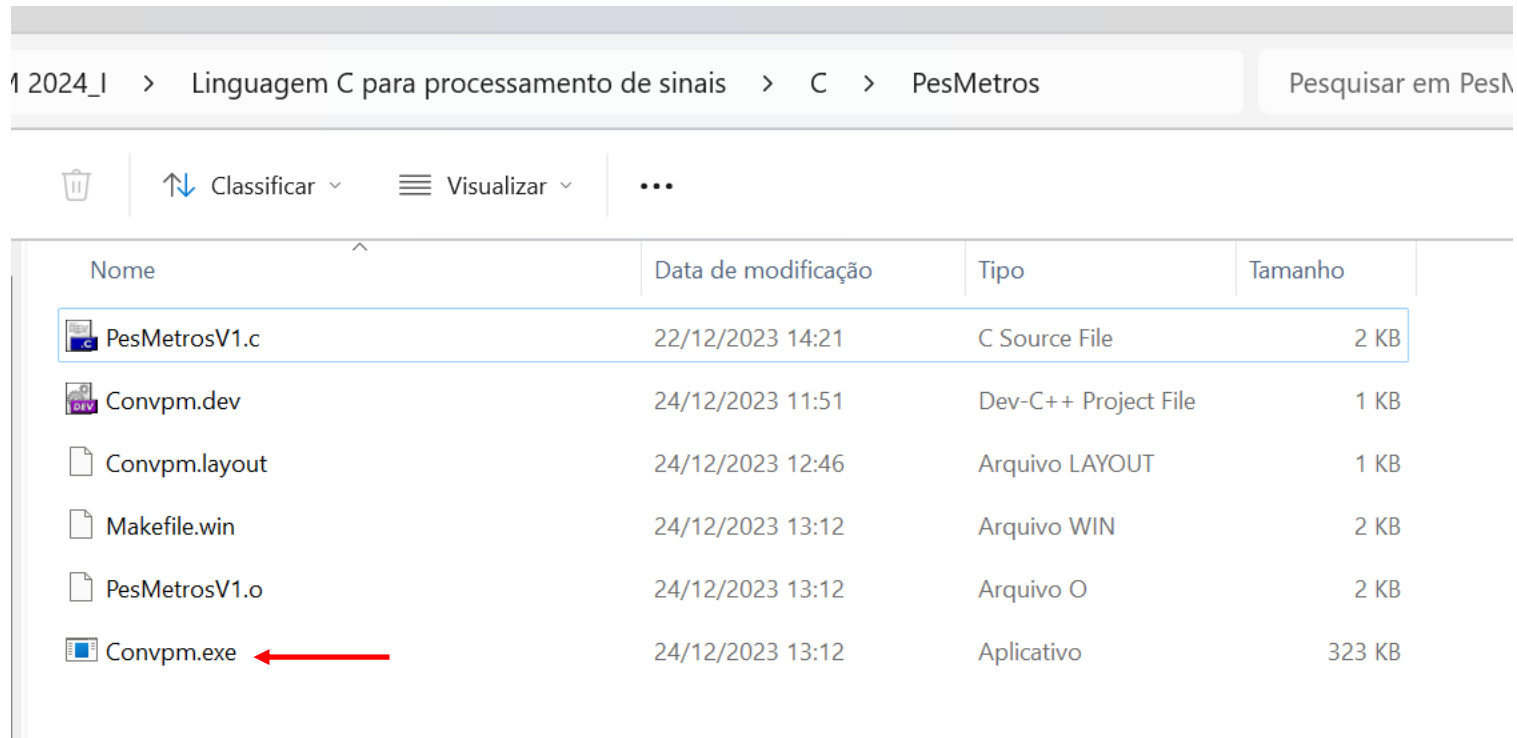
Compilation results...







-----

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\fccde\DATA\UFSM\UFSM 2024\_\Linguagem C para processamento de sinais\C\PesMetros\Convpm.exe
- Output Size: 322,7841796875 KiB
- Compilation Time: 2,44s

## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Note também que na pasta de trabalho “PesMetros” foi criado o programa final executável “Convpm.exe” (seta vermelha), bem como o arquivo PesMetrosV1.o, correspondente ao código objeto binário intermediário (ver slide 4):



Nome	Data de modificação	Tipo	Tamanho
 PesMetrosV1.c	22/12/2023 14:21	C Source File	2 KB
 Convpm.dev	24/12/2023 11:51	Dev-C++ Project File	1 KB
 Convpm.layout	24/12/2023 12:46	Arquivo LAYOUT	1 KB
 Makefile.win	24/12/2023 13:12	Arquivo WIN	2 KB
 PesMetrosV1.o	24/12/2023 13:12	Arquivo O	2 KB
 Convpm.exe	24/12/2023 13:12	Aplicativo	323 KB

É instrutivo abrir o arquivo PesMetrosV1.o no aplicativo Bloco de Notas (notepad.exe), e verificar que o conteúdo do arquivo é binário (i.e., ininteligível, em sua maior parte formado por caracteres ASCII não imprimíveis correspondentes às instruções em linguagem de máquina binária).

## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Para executar o programa final executável “Covpm.exe” criado na pasta de trabalho “PesMetros”, devemos manter em mente que configuramos o DevC++ para gerar uma “console application” (ver slide 9). Isto significa que o programa deve ser executado na janela do *command prompt* do Windows, que é uma janela que faz a emulação do console.

Há inúmeras maneiras de acionar o *command prompt* do Windows, como, por exemplo, é indicado nos links abaixo:

<https://support.kaspersky.com/common/windows/14637>

[https://www.majorgeeks.com/content/page/command\\_prompt\\_11.html](https://www.majorgeeks.com/content/page/command_prompt_11.html)

Qual a razão de criar programas do tipo “console application”? A maioria dos desenvolvedores de códigos para processamento de sinal e, em geral, os desenvolvedores de sistemas embarcados (<https://www.betterteam.com/embedded-developer-job-description>) desenvolvem seus programas inicialmente na forma de “console application”, o que permite facilmente e rapidamente passar parâmetros aos programas através da linha de comando (*command line*) e através do teclado do console. Portanto, neste estudo, a razão de trabalharmos com *command line* é a facilidade e a rapidez em configurar e passar parâmetros a um programa durante o seu desenvolvimento (ver seção “Why should I even care about using the terminal?” em <https://www.freecodecamp.org/news/command-line-for-beginners/> ).

## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Accionando então o *command prompt* do Windows através de qualquer uma das maneiras sugeridas nos 3 primeiros links do slide anterior, e, através dos comandos “cd” e “dir” digitados no teclado vamos (1) mudar a janela do *prompt* para acessar a pasta de trabalho (comando “cd” – seta vermelha abaixo) e (2) mostrar os arquivos na pasta de trabalho (comando “dir” – seta azul abaixo), conforme janela do *prompt* (= janela do console = tela do console) mostrada abaixo:

```
Prompt de Comando
Microsoft Windows [versão 10.0.22631.2428]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\fccde>cd C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>dir

O volume na unidade C é OS
O Número de Série do Volume é 0A34-9CE2

Pasta de C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros

24/12/2023  13:12    <DIR>          .
22/12/2023  13:22    <DIR>          ..
24/12/2023  15:01             932 Convpm.dev
24/12/2023  13:12          330.531 Convpm.exe
24/12/2023  15:01             92 Convpm.layout
24/12/2023  13:12           1.404 Makefile.win
22/12/2023  14:21           1.378 PesMetrosV1.c
24/12/2023  13:12           1.062 PesMetrosV1.o
                6 arquivo(s)          335.399 bytes
                2 pasta(s)    692.827.529.216 bytes disponíveis

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>
```

Uma lista dos comandos disponíveis no *command prompt* do Windows e a sua descrição/utilidade pode ser encontrada em <https://www.lifewire.com/list-of-command-prompt-commands-4092302> .

Especificamente, o *prompt* na janela do console é o caractere que fica piscando na tela, e é assim chamado porque ele fica em prontidão imediata à espera de um dos comandos acima referidos ser inserido no *prompt* via teclado .



## Utilizando o DevC++ p/ gerar um programa executável .exe a partir de um código fonte .c

Finalmente, para executar o programa “Covpm.exe” digitamos “Covpm” no *prompt* da janela do console (seta vermelha abaixo). Inicialmente a função print() na linha 18 do fonte C (ver slide 19) imprime na tela “Informe o numero de pes:”, daí a função scanf() na linha 19 interrompe a execução do programa e fica aguardando a entrada do dado solicitado. Daí é inserido via teclado o valor 10 e é pressionado <ENTER> (seta azul abaixo). O programa volta então a ser executado e calcula o valor de pés convertido para metros na linha 20. A seguir a função printf() na linha 21 imprime na tela “10 pes = 3.048000 metros” (seta verde abaixo), a função main() retorna 0 ao sistema operacional (sucesso na execução) e o programa é encerrado.

A descrição da sintaxe de uso e detalhes de formatação para as funções printf() e scanf() estão disponíveis na seção “2.12.4 Formatted I/O Functions” na página 100 e na seção “1.1.2 Escape sequences” na página 17 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf>.

```
Prompt de Comando
Microsoft Windows [versão 10.0.22631.2428]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\fccde>cd C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>dir
O volume na unidade C é OS
O Número de Série do Volume é 0A34-9CE2

Pasta de C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros

24/12/2023  13:12    <DIR>          .
22/12/2023  13:22    <DIR>          ..
24/12/2023  15:01             932 Convpm.dev
24/12/2023  13:12          330.531 Convpm.exe
24/12/2023  15:01             92 Convpm.layout
24/12/2023  13:12           1.404 Makefile.win
22/12/2023  14:21           1.378 PesMetrosV1.c
24/12/2023  13:12           1.062 PesMetrosV1.o
               6 arquivo(s)          335.399 bytes
               2 pasta(s)      692.827.529.216 bytes disponíveis

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros> convpm
Informe o numero de pes: 10
10 pes = 3.048000 metros

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>
```

## Tipos de variáveis básicas

Note que na execução do programa “Covpm.exe”, ao entrarmos o número de pés com uma parte fracionária decimal (por exemplo, 10.8 pés) o programa trunca o valor após o ponto decimal, conforme podemos observar abaixo:

```
Prompt de Comando
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>convpm
Informe o numero de pes: 10.8
10 pes = 3.048000 metros
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>
```

Isto ocorre porque a variável “NumPes” foi declarada como uma variável inteira (int) no inicio da função main() no fonte “PesMetrosV1.c” (ver slide 19).

Para evitar este problema, é necessário declarar a variável “NumPes” como uma variável de ponto flutuante (float), conforme código fonte disponível em <https://www.fccdecastro.com.br/CursoC&C++/C/PesMetrosV2.c> e mostrado no próximo slide.

Note que o problema do truncamento do valor após o ponto decimal foi resolvido, conforme abaixo:

```
Prompt de Comando
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>convpm
Informe o numero de pes: 10.8
10.800000 pes = 3.291840 metros
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\PesMetros>
```

**Nota:** Para obtermos o resultado acima o fonte “PesMetrosV1.c” foi deletado do *Project* “Convpm.dev” e foi adicionado a este *Project* o fonte “PesMetrosV2.c”, conforme passo a passo nos slides 14 a 17.

## Tipos de variáveis básicas

PesMetrosV2.c

```
1  /* conversao de pes para metros V2 - NumPes e ValorMetros sao declaradas como
2  variaveis em ponto flutuante (float) */
3
4  #include <stdio.h> /* inclui arquivo stdio.h c/ as definicoes
5  para a biblioteca de funcoes padrao "standard I/O" */
6
7  int main(void) /* note que aqui explicitamos que a funcao main() não
8  tem qualquer argumento, declarando um argumento do tipo void . Mais adiante estudaremos
9  os argumentos de linha de comando int argc e char *argv[], e que nos permitirão passar parâmetros
10 ao programa na janela do console */
11 {
12     float NumPes; /* declaracao da variavel em ponto flutuante NumPes */
13     float ValorMetros; /* declaracao da variavel em ponto flutuante ValorMetros */
14
15     printf("Informe o numero de pes: "); /* imprime na tela do console o que estah estre aspas */
16     scanf("%f", &NumPes); /* Le NumPes do teclado. Nota: & -> operador "endereço na memoria de" */
17     ValorMetros = NumPes * 0.3048; /* conversao de pes para metros */
18     printf("%f pes = %f metros\n", NumPes, ValorMetros); /* imprime na tela do console as variaveis
19     NumPes e ValorMetros */
20     return 0; /* sucesso na execucao - nao houve erro - retorna o valor 0 ao sistema operacional */
21 }
22
```

Note portanto que **é crucial utilizar o tipo de variável adequado ao contexto de uso do programa que está sendo desenvolvido**. No próximos dois slides são mostradas tabelas com todos os tipos de variáveis básicas e modificadas em C.

## Tipos de variáveis básicas

A tabela abaixo mostra o tamanho em **bits** e o intervalo de variação (valor mínimo e valor máximo) dos tipos básicos de variáveis em C. O tamanho em **bytes** é obtido dividindo por 8 o tamanho em **bits**.

Tipo	Tamanho	Intervalo de variação
char	8 bits (=1 byte)	-128 a 127
int	16 bits (=2 bytes)	-32768 a 32767
float	32 bits (=4 bytes)	(±) 3.4E-38 a 3.4E+38
double	64 bits (=8 bytes)	(±) 1.7E-308 a 1.7E+308
void	0	sem valor

Ver também seção "1.2.2 Variables" na página 20 de

<https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf> .

O formato das palavras binárias que representam variáveis float e double são normalizadas pelo padrão IEEE-754 (ver [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754), [https://en.wikipedia.org/wiki/Single-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Single-precision_floating-point_format) e [https://en.wikipedia.org/wiki/Double-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Double-precision_floating-point_format) ).

Conforme já brevemente comentado no slide 19, o escopo de uma variável em C é o bloco ou região do programa onde uma variável é declarada, definida e usada. Fora da região do escopo não podemos acessar a variável e ela é tratada como um identificador não declarado.

Uma **variável local** (= variável automática) é toda variável declarada dentro do bloco de comandos que define uma função (preferencialmente no início da função) e o escopo da variável é limitado ao bloco de comandos que define a função (ver [https://en.wikipedia.org/wiki/Automatic\\_variable](https://en.wikipedia.org/wiki/Automatic_variable) ).

Uma **variável global** é toda variável declarada antes da função main(), fora do bloco de comandos que define toda e qualquer função do programa. Neste caso o escopo não tem limites, de forma que a variável global pode ser acessada em qualquer ponto do programa. Deve-se evitar usar nomes de variáveis globais que sejam iguais ao nome de uma variável local.

## Tipos de variáveis básicas

O tipos básicos de variáveis em C podem ser modificadas pelos **modificadores de tipo** signed, unsigned, long, short, conforme mostra a tabela abaixo:

**Todas as combinações possíveis dos tipos básicos e modificadores do C:**

Tipo	Tamanho(em bits) [8 bits = 1 byte]	Intervalo de variação
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	(±) 3.4E-38 a 3.4E+38
double	64	(±) 1.7E-308 a 1.7E+308
long double	80	(±) 3.4E-4932 a 1.1E+4932

**Nota:** No DevC++ valem os seguintes tamanhos em bytes:

sizeof(unsigned)=4

sizeof(int)=4

sizeof(unsigned long)=4

sizeof(unsigned short)=2

sizeof(short int)=2

sizeof(double)=8

sizeof(float)=4

sizeof(char)=1

sizeof(void)=1

**Nota:** Há controvérsia quanto ao uso de variáveis long double no DevC++. Ver <https://sourceforge.net/p/dev-cpp/discussion/48211/thread/dc01cc6b/> .



## Tipos de variáveis básicas

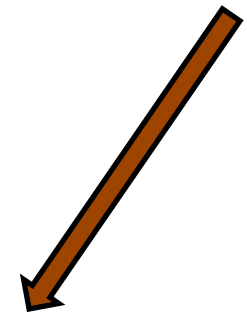
Um exemplo adicional do uso de variáveis básicas, agora utilizando variáveis do tipo char, é o código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/PrintAlfabeto.c> mostrado abaixo. O resultado da execução na janela do console é mostrado no próximo slide.

PrintAlfabeto.c

```
1  /* Imprime na janela do console o alfabeto de 'A' a 'Z' */
2
3  /* Nota: Para programas antivírus tipo McAfee pode ser necessário
4  desativar a varredura em tempo real (McAfee considera que escrever
5  bytes (char) diretamente na porta da janela do console sem passar
6  pela "censura" do Windows é possivelmente a ação de um hacker
7  ou de um vírus ...) */
8
9  #include <stdio.h> /* headers básicos */
10 #include <stdlib.h>
11
12 int main (void) /* função principal de qualquer programa em C */
13 {
14     char letra; /* declara variável letra como char */
15
16     printf("caractere\tvalor ASCII\n"); /* imprime na janela do console o que
17                                         estah entre aspas */
18
19     for(letra = 'A' ; letra <= 'Z' ; letra ++ ) { /* loop for variando o valor ASCII da
20                                                     variável letra de 'A'=65 atéh 'Z'=90 */
21
22         printf("%c\t\t%u\n", letra,letra); /* imprime na janela do console */
23
24     }
25
26     return 0;
27 }
```

Veremos loop "for" no Cap I.2. Mas, já adiantando, a sintaxe da declaração "for" pode ser encontrada na seção "1.6.5 for" na página 44 de

<https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf>



## Tipos de variáveis básicas

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
caractere      valor ASCII
A              65
B              66
C              67
D              68
E              69
F              70
G              71
H              72
I              73
J              74
K              75
L              76
M              77
N              78
O              79
P              80
Q              81
R              82
S              83
T              84
U              85
V              86
W              87
X              88
Y              89
Z              90
```

Note que a tela do console mostrada acima resulta da chamada da função **`printf("%c\t\t%u\n", letra,letra);`** na linha 22 do fonte C no slide anterior. Para cada valor da variável `letra` no loop `for`, a função `printf()` primeiro imprime na tela do console o caractere ASCII associado ao valor da variável `letra` (especificador de formato `"%c"`) e, após duas sequências de escape tab `"\t"`, imprime o valor da variável inteira `letra` (especificador de formato `"%d"`), e, finalmente, inicia uma nova linha na tela do console através da sequência de escape `"\n"`. A tabela ASCII é encontrada em <https://pt.wikipedia.org/wiki/ASCII>.

## Constantes

Tipo de Dado	Exemplos de valores de constantes
char	'a' 'n' '9' 97 110 57
int	1 123 21000 -234
long int	35000 -34
short int	10 -12 90
unsigned int	10000 987 40000
float	123.23 4.34e-3
double	123.23 12312333 -0.9876324

### Constantes Hexadecimais, Octais e Decimais :

A representação do valor de uma constante hexadecimal inicia com os caracteres “0x” e a representação do valor de uma constante octal inicia com o caractere “0”.

Por exemplo:

```
int Hex = 0xFF; /* a variável “Hex” é declarada como inteira (int) e é atribuído à variável Hex o valor da constante 0xFF em hexadecimal, que corresponde ao valor 255 em decimal. Portanto, o valor decimal armazenado em Hex após esta atribuição é 255 */
```

```
int Oct = 011; /* a variável “Oct” é declarada como inteira (int) e é atribuído à variável Oct o valor da constante 011 em octal, que corresponde ao valor 9 em decimal. Portanto, o valor decimal armazenado em Oct após esta atribuição é 9 */
```

```
float Dec = 3.14159; /* a variável “Dec” é declarada como de ponto flutuante de 32 bits (float) e é atribuído à variável Dec o valor da constante 3.14159 em decimal. Portanto, o valor decimal armazenado em Dec após esta atribuição é 3.14159 */
```

## Constantes

Sequências de escape (ver seção "1.1.2 Escape sequences" na página 17 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf> ) são constantes do tipo char.

Entre outras finalidades das sequências de escape, de particular interesse é a ação que resulta quando se insere uma sequência de escape na *string* (veremos *strings* adiante) de formato da família de funções printf(). Por exemplo, já discutimos no slide 30 o resultado da chamada de função **“printf(“%c\t\t%u\n”, letra,letra);”**: A função printf() primeiro imprime na tela do console o caractere ASCII associado ao valor da variável letra (especificador de formato “%c” ) e, após duas sequências de escape tab “\t”, imprime o valor da variável inteira letra (especificador de formato “%d” ), e, finalmente, inicia uma nova linha através da sequência de escape “\n”. Abaixo, a tabela das sequências de escape.

Sequência de escape	Significado
\b	Retrocesso
\f	Alimentação de formulário
\n	Nova linha
\r	Retorno para o início da linha
\t	Tabulação horizontal (avança 8 caracteres)
\"	Aspas
\'	Apóstrofo
\0	Nulo
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro
\N	Constante Octal (onde N é uma constante octal)
\xN	Constante hexadecimal (onde N é uma constante hexadecimal)

Note que uma sequência de escape é um dos caracteres definidos na tabela ASCII (ver <https://pt.wikipedia.org/wiki/ASCII>). Por exemplo, suponhamos que um código fonte declare em um determinado ponto do escopo a variável **char ch**. E que, em uma determinada linha do código fonte há a atribuição **ch = ‘\t’**; Pergunta: Qual o valor decimal armazenado em ch após esta atribuição? Resposta: Consultando a tabela ASCII de caracteres de controle (caracteres não-imprimíveis), observamos que o valor decimal de ‘\t’ é 9. Portanto 9 é o valor decimal armazenado na variável ch.

## Operadores aritméticos

Operador	Ação	
-	subtração, também o menos unário (-1*x)	
+	adição	
*	multiplicação	Obs) Operador unário é aquele que opera sobre um único argumento.
/	divisão	
%	resto da divisão	
--	decremento	
++	incremento	

Operador de atribuição	Ação
=	C=A+B atribui o valor de A+B à variável C
+=	C += A equivale a C = C + A
-=	C -= A equivale a C = C - A
*=	C *= A equivale a C = C * A
/=	C /= A equivale a C = C / A
%=	C %= A equivale a C = C % A

Operador de atribuição (ver slide 38)	Ação
<<=	C <<= A equivale a C = C << A
>>=	C >>= A equivale a C = C >> A
&=	C &= A equivale a C = C & A
^=	C ^= A equivale a C = C ^ A
=	C  = A equivale a C = C   A

## Operadores aritméticos

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/QuoRes.c> mostrado abaixo é um exemplo de uso de operadores aritméticos. A execução do programa na janela do console é mostrada no próximo slide.

```
1  /******  
2  * Este programa le dois numeros inteiros do teclado, determina o quociente  
3  * e o resto entre os dois numeros e imprime os resultados na tela do console  
4  *****/  
5  
6  /******  
7  * HEADERS:  
8  *****/  
9  #include <stdio.h>  
10 #include <stdlib.h>  
11  
12 /******  
13 * main()  
14 *****/  
15 void main(void) // nao ha argumentos e nao ha valor de retorno na main()  
16 {  
17     int A,B; // declara as variaveis inteiras A e B  
18  
19     printf("\nEste programa efetua a divisao entre dois inteiros A e B.\n");  
20     printf("Informe A e B separados de um espaco: "); // imprime na tela do console  
21     scanf("%d%d", &A, &B); /* interrompe a execucao, le dois valores do teclado  
22                             e armazena os valores lidos nos enderecos de memoria respectivos das  
23                             variaveis A e B */  
24  
25     printf("Quociente da divisao de %d por %d = %d\n",A,B,A/B); // imprime A, B e A/B na tela do console  
26     printf("Resto da divisao de %d por %d = %d\n",A,B,A%B); /* imprime A, B e A%B na tela do console  
27                     A%B = resto da divisao de A por B */  
28 }
```

## Operadores aritméticos



Prompt de Comando



```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Este programa efetua a divisao entre dois inteiros A e B.
```

```
Informe A e B separados de um espaco: 8 5
```

```
Quociente da divisao de 8 por 5 = 1
```

```
Resto da divisao de 8 por 5 = 3
```

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>
```

## Operadores de incremento e decremento

O comando “**x= x+1;**” pode ser alternativamente implementado de duas maneiras:

**++x; /\* primeiro incrementa de 1 o valor da variável inteira (int) x e depois usa o conteúdo de x \*/**  
ou  
**x++; /\* primeiro usa o conteúdo da variável inteira (int) x e depois incrementa de 1 o valor de x \*/**

Por exemplo:

**x=10; /\* atribui o valor 10 à variável inteira x \*/**  
**y= ++x; /\* primeiro executa x=x+1 e depois faz a atribuição y=x \*/**  
**/\* O resultado é: y=11 e x=11 \*/**

**x=10; /\* atribui o valor 10 à variável inteira x \*/**  
**y= x++; /\* primeiro faz a atribuição y=10 e depois executa x=x+1 \*/**  
**/\* O resultado é: y=10 e x=11 \*/**

De mesma forma, o comando “**x= x-1;**” pode ser alternativamente implementado de duas maneiras, i.e., através dos operadores de decremento **x--** e **--x**.



## Precedência de operadores

Abaixo é mostrado precedência de operadores aritméticos. A precedência de operadores define a ordem em que os operadores são executados. Operadores com maior precedência são executados primeiro.

Executado em 1º lugar: ++, --  
Executado em 2º lugar: - (unário)  
Executado em 3º lugar: \*, /, %  
Executado em 4º lugar: +, -

**Nota 1:** Operadores do mesmo nível de precedência são avaliados pelo compilador da esquerda para a direita.

**Nota 2:** Pares de parênteses podem ser inseridos para alterar a ordem de avaliação, forçando uma operação ou conjunto de operações para um nível de precedência mais alto. Aconselha-se sempre o uso de parênteses, já que além de deixarem o código mais claro, eles não acrescentam nenhum tempo de execução adicional ou consumo extra de memória.

# Operadores relacionais e lógicos

## Operadores relacionais

>	maior que
>=	maior ou igual que
<	menor que
<=	menor ou igual que
==	igual
!=	diferente

## Operadores lógicos

&&	AND(E)
	OR(OU)
!	NOT(NÃO)

## Operadores lógicos *bitwise*

Sejam dois valores int ou char A e B

**& : A&B** efetua a operação AND entre cada respectivo bit em A e B.

**| : A|B** efetua a operação OR entre cada respectivo bit em A e B.

**^ : A^B** efetua a operação XOR entre cada respectivo bit em A e B.

**<< : A<<B** shifta à esquerda B bits de A.

**>> : A>>B** shifta à direita B bits de A.

**~ : ~A** faz o complemento de 1 (NOT) de cada bit de A.

```
/* exemplos de uso de operadores bitwise */
#include <stdio.h>
void main(void)
{
/* a = 5(BIN: 00000101), b = 9(BIN: 00001001)
- verifique os valores binários correspondentes
com a calculadora do Windows no modo
"Programador - BYTE" */
unsigned char a = 5, b = 9;
printf("a = %d, b = %d\n", a, b);
printf("a&b = %d\n", a & b); //BIN: 00000001
printf("a|b = %d\n", a | b); //BIN: 00001101
printf("a^b = %d\n", a ^ b); //BIN: 00001100
printf("~a = %d\n", a = ~a); //BIN: 11111010
printf("b<<1 = %d\n", b << 1); //BIN: 00010010
printf("b>>1 = %d\n", b >> 1); //BIN:00000100
}
```

Execução na tela do console:

```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
~a = 250
b<<1 = 18
b>>1 = 4
```

## Operadores relacionais e lógicos

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/OperRel.c> mostrado abaixo é um exemplo de uso de operadores relacionais:

```
1  *****
2  * Este programa le dois numeros inteiros do teclado e imprime na tela do
3  * console o valor logico (1-Verdadeiro 0-Falso) resultante de cada um dos 6
4  * operadores relacionais aplicados na relacao entre os dois numeros
5  *****/
6
7  *****
8  * HEADERS:
9  *****/
10 #include <stdio.h>
11 #include <stdlib.h>
12
13 *****
14 * main()
15 *****/
16 void main(void) // nao ha argumentos e nao ha valor de retorno na main()
17 {
18     int i, j; /* declara as variaveis inieiras i e j */
19
20     /* imprime na tela do console: */
21     printf("\nInforme dois numeros inteiros separados por espaco: ");
22
23     scanf("%d%d", &i, &j); /* interrompe a execucao, le dois valores do teclado
24     e armazena os valores lidos nos enderecos de memoria respectivos das
25     variaveis i e j */
```

Nota: O código fonte acima continua no próximo slide

## Operadores relacionais e lógicos

```
26 |
27 | /* imprime na tela do console o valor logico (1-Verdadeiro
28 | 0-Falso) resultante de cada um dos 6 operadores relacionais
29 | aplicados nas variaveis i e j: */
30 | printf("\nNa tabela abaixo 1 = V e 0 = F:\n");
31 | printf("%d == %d -> %u\n", i, j, i==j);
32 | printf("%d != %d -> %u\n", i, j, i!=j);
33 | printf("%d <= %d -> %u\n", i, j, i<=j);
34 | printf("%d >= %d -> %u\n", i, j, i>=j);
35 | printf("%d < %d -> %u\n", i, j, i<j);
36 | printf("%d > %d -> %u\n", i, j, i>j);
37 | }
```

A execução do programa na janela do console é mostrada abaixo:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Informe dois numeros inteiros separados por espaco: -7 3
```

```
Na tabela abaixo 1 = V e 0 = F:
```

```
-7 == 3 -> 0
```

```
-7 != 3 -> 1
```

```
-7 <= 3 -> 1
```

```
-7 >= 3 -> 0
```

```
-7 < 3 -> 1
```

```
-7 > 3 -> 0
```

## Operadores relacionais e lógicos

Portanto, conforme vimos no exemplo do slide anterior, o resultado de um operador relacional aplicado à duas variáveis é o valor 1 quando a relação entre os valores das duas variáveis é verdadeira e é o valor 0 quando a relação entre os valores das duas variáveis é falsa.

Note que os operadores relacionais podem ser aplicados a qualquer tipo de dado básico. Por exemplo, no código abaixo é mostrada a comparação entre os valores de duas variáveis `ch1` e `ch2` do tipo `char`:

```
char ch1='A'; // o valor decimal 65 é atribuído à ch1 (ver tabela ASCII)
```

```
char ch2='B'; // o valor decimal 66 é atribuído à ch1 (ver tabela ASCII)
```

```
if(ch2 > ch1) printf("o valor ASCII de ch1 eh maior que o de ch2"); // imprime na tela do console se ch2 eh maior que ch1
```

```
else printf("o valor ASCII de ch1 nao eh maior que o de ch2"); // imprime na tela do console se ch2 nao eh maior que ch1
```

Veremos as declarações **if** e **else** no Cap I.2.

## Operadores relacionais e lógicos

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/OperLog.c> mostrado abaixo é um exemplo de uso de operadores lógicos. A execução do programa na janela do console é mostrada no próximo slide.

```
1  /* *****
2  * Este programa le dois numeros inteiros 0 ou 1 do teclado e imprime na tela do
3  * console o valor resultante de cada um dos 3 operadores logicos aplicados na
4  * relacao entre os dois numeros
5  ***** */
6  /* *****
7  * HEADERS:
8  ***** */
9  #include <stdio.h>
10 #include <stdlib.h>
11 /* *****
12 * main()
13 ***** */
14 void main(void) // nao ha argumentos e nao ha valor de retorno na main()
15 {
16     unsigned i, j; // declara as variaveis inteiras sem sinal i e j
17
18     /* imprime na tela do console: */
19     printf("\nInforme dois numeros separados por espacos (cada um sendo 0 ou 1):");
20
21     scanf("%u%u", &i, &j); /* interrompe a execucao, le dois valores do teclado
22 e armazena os valores lidos nos enderecos de memoria respectivos das
23 variaveis i e j */
24
25     /* imprime na tela do console o valor resultante de cada um dos 3 operadores
26 logicos aplicados na relacao entre os valores das variaveis i e j: */
27     printf("%u AND %u = %u\n", i, j, i && j);
28     printf("%d OR  %d = %u\n", i, j, i || j);
29     printf("NOT %u = %u\n", i, !i);
30 }
```

## Operadores relacionais e lógicos

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Informe dois numeros separados por espacos (cada um sendo 0 ou 1):1 0
```

```
1 AND 0 = 0
```

```
1 OR 0 = 1
```

```
NOT 1 = 0
```

## Operadores relacionais e lógicos

Conforme vimos nos exemplos anteriores, todas as expressões lógicas e relacionais produzem um resultado 0 ou 1. Portanto, o seguinte programa não só está correto como produzirá o número 1 na tela:

```
#include <stdio.h>
void main(void)
{
    int x;
    x=100;
    printf("%u", x>10);
}
```

E para encerrar os exemplos com operadores relacionais e lógicos, segue abaixo um exemplo com o operador resto da divisão “%”, cujo resultado é imprimir na janela do console os números pares entre 1 e 100

```
/* Imprime na tela do console os números pares entre 1 e 100. */
#include <stdio.h>
void main(void)
{
    int i;
    for(i=1; i<=100; i++)
    if(!(i%2)) printf("%d ",i); /* a declaração resto da divisão da variável i pelo inteiro 2 “(i%2)” resulta falso (zero) quando */
                                /*a variável i armazenar um número par. Esse valor zero é invertido pelo operador NOT “!” */
                                /* habilitando a execução de “printf(“%d ",i);”.
```



## Operador de atribuição

Em C o operador de atribuição é o sinal de igual ' = '. Uma particularidade do C é ser permitido que o operador de atribuição seja usado em conjunto com expressões relacionais ou lógicas, conforme programa abaixo:

```
#include <stdio.h>
void main(void)
{
    int x, y, produto;

    printf("\nInforme dois numeros inteiros separados por espaco: ");
    scanf("%d%d", &x, &y);

    if( (produto=x*y) < 0) { /* expressão relacional c/ operador de atribuição incluso */
        printf ("Um dos numeros eh negativo e seu produto eh %d\n", produto);
    }
    else { /* caso contrário da condição do if */
        printf("O produto dos numeros positivos eh: %d\n", produto);
    }
}
```

O resultado na tela do console para duas situações de entrada de dados distintas é mostrada abaixo:

Informe dois numeros inteiros separados por espaco: 2 3  
O produto dos numeros positivos eh: 6

Informe dois numeros inteiros separados por espaco: -2 3  
Um dos numeros eh negativo e seu produto eh -6

## Conversão de tipos de dados (cast):

A operação de **cast** converte um determinado tipo de dado em outro tipo de dado, desde que a conversão faça sentido (se não fizer sentido o compilador emite um **warning** ou, em alguns casos, até mesmo um **error**). A sintaxe do **cast** é conforme abaixo:

**(novo\_tipo\_de\_dado) expressão**

onde **novo\_tipo\_de\_dado** é qualquer tipo de dado em C , incluindo tipos de dados definidos pelas declarações **typedef** e **struct** para representar, por exemplo, números complexos (veremos adiante o uso das declarações **typedef** e **struct** na definição de dados não-básicos do C, como, por exemplo, números complexos). Por exemplo, o programa abaixo mostra o **cast** para **float** necessário quando se faz a divisão entre dois inteiros e é desejado um resultado em ponto flutuante:

```
#include <stdio.h>
void main(void)
{
    int soma=14, contagem=4;
    float media;

    media = (float) soma / contagem;    /* o operador de cast (float) tem precedencia
                                         sobre o operador de divisao /. Portanto o valor
                                         de soma eh primeiro convertido para o novo tipo
                                         float e, a seguir, eh dividido pelo valor da variavel
                                         contagem resultando em um valor do tipo float */

    printf("Valor da media = %f\n", media);
}
```

O resultado na tela do console é mostrado abaixo:

Valor da media = 3.500000

## Conversão de tipos de dados (cast):

O que acontece se não usarmos o **cast** quando se faz a divisão entre dois inteiros e é desejado um resultado em ponto flutuante? O programa abaixo mostra a consequência de não usar o **cast**:

```
#include <stdio.h>
void main(void)
{
    int soma=14, contagem=4;
    float media;

    media = soma / contagem; /* sem cast */

    printf("Valor da media = %f\n", media);
}
```

O resultado na tela do console é mostrado abaixo:

Valor da media = 3.000000

O que aconteceu é que `soma / contagem` é uma divisão entre inteiros, então as casas decimais após o ponto decimal são truncadas.

Note que o truncamento continuará a ocorrer se usarmos a declaração **`media = (float) (soma / contagem);`**, porque os parênteses forçam **`soma / contagem`** a ser uma divisão inteira e o cast para float apenas irá converter o resultado já truncado.

## Introdução à *strings*

Em C uma *string* é uma sequência de caracteres (char) armazenada sequencialmente na forma de elementos de um vetor, sendo o vetor terminado pelo caractere ASCII '\0' (NULL), cujo valor decimal é zero (ver <https://pt.wikipedia.org/wiki/ASCII>). O terminador NULL serve para marcar o final do vetor que armazena a *string*.

**Há várias funções na biblioteca padrão para manipulação de *strings* (ver seção "2.13.2 String Functions" na página 112 e seção "2.14 string.h" na página 125 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide-Huss.pdf>).**

Uma maneira “burra” mas didática de inicializarmos uma *string* com, por exemplo, a sequência de caracteres 'H' 'e' 'l' 'l' 'o' '!' '\0', e a seguir imprimirmos a *string* “Hello!” na tela do console é conforme segue:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char txt[7]; /* declara o vetor de 7 elementos do tipo char, 6 elementos
                  para os caracteres de "Hello!" mais 1 elemento para o terminador '\0' */

    /* inicializa o vetor txt[7] elemento por elemento: */
    txt[0]='H';
    txt[1]='e';
    txt[2]='l';
    txt[3]='l';
    txt[4]='o';
    txt[5]='!';
    txt[6]='\0';

    printf("%s\n",txt); /* imprime na tela do console o vetor txt[7].
                        Note o especificador de formato %s para imprimir strings */
}
```

Resultado na tela do console: Hello!

## Introdução à *strings*

A maneira usual de inicializar uma *string* é conforme abaixo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char txt[]="Hello!"; /* declara e inicializa a string txt[] com "Hello!". O
                           terminador '\0' eh automaticamente adicionado. O tamanho necessario
                           ao vetor txt[] também eh determinado automaticamente */

    printf("%s\n",txt); /* imprime na tela do console a string txt[].
                           Note o especificador de formato %s para imprimir strings */
}
```

O resultado na tela do console é mostrado abaixo ( o mesmo resultado do fonte C do slide anterior):

Hello!

Uma característica importante de uma *string* é que o seu nome equivale ao endereço na memória do elemento inicial do vetor que armazena os caracteres da *string*. Por exemplo, no programa acima a declaração “**printf("%s\n",txt);**” equivale a “**printf("%s\n",&txt[0]);**”, onde & é o operador ‘endereço de’.

## Introdução à *strings*

Para ler uma *string* do teclado, primeiro criamos um vetor de caracteres de tamanho suficiente para armazenar a *string*, mais o terminador nulo. Para entrada via teclado, um vetor de 80 elementos é mais do que suficiente. E daí, para ler uma *string* do teclado, pode-se usar a função `scanf()` ou podemos usar a função `gets()`. O argumento da função `gets()` é o nome da *string*, e `gets()` vai lendo caracteres do teclado até que a tecla <ENTER> seja pressionada. A tecla <ENTER> não é armazenada como caractere, mas sim substituída pelo terminador nulo. O fonte C mostrado abaixo é um exemplo de uso da função `gets()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char txt[80]; /* declara o vetor de caracteres txt[80] com 80 elementos */

    printf("\nDigite o seu nome: ");
    gets(txt); /* vai lendo caracteres do teclado e armazenando em txt
               ateh que a tecla <ENTER> seja pressionada */

    printf("Ola %s!\n", txt); /* imprime na tela do console a string txt[] */
}
```

A execução do programa na janela do console é mostrada abaixo:

```
Digite o seu nome: Alfa Beta Gama
Ola Alfa Beta Gama!
```

## Introdução à *strings*

O programa abaixo é semelhante ao do slide anterior mas usa a função `scanf()` para ler uma *string* do teclado:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char txt[80]; /* declara o vetor de caracteres txt[80] com 80 elementos */

    printf("\nDigite o seu nome: ");
    scanf("%s",txt); /* le a string digitada no teclado e
                     apos a tecla <ENTER> ser pressionada armazena a string em txt */
    /* Nota: scanf("%s",txt) equivale a scanf("%s",&txt[0]) */

    printf("Ola %s!\n", txt); /* imprime na tela do console a string txt[] */
}
```

A execução do programa na janela do console é mostrada abaixo.

```
Digite o seu nome: Alfa Beta Gama
Ola Alfa!
```

Note no resultado acima que a função `scanf()` interrompe o armazenamento de caracteres no vetor `txt[80]` quando encontra um espaço na sequencia de caracteres digitada no teclado, diferente do caso da função `gets()` usada no programa do slide anterior.

## Introdução à funções

Um programa em C é basicamente um conjunto de funções que opera sobre um conjunto de dados, cada função executando uma determinada tarefa no contexto da aplicação do programa. Esta tarefa pode ser tanto retornar um valor a partir dos dados fornecidos à função (=argumentos da função) como também pode ser a de um procedimento que executa uma tarefa sem valor de retorno.

**Para desenvolver um programa em C para processamento de sinais, primeiramente escrevemos o código C de cada uma das funções que se fazem necessárias, testamos e validamos cada função individualmente utilizando um aplicativo como o Matlab e o Mathcad, e daí, integramos o conjunto de funções em um único programa. Somente após a integração é feito o teste e validação do programa como um todo, sempre tendo como premissa que todas as funções que constituem o programa já tenham sido todas previamente testadas e validadas individualmente.**

Cada função deve ter um único nome que a identifica. Quando este nome é encontrado em qualquer lugar do programa a função é chamada e acessada, e os comandos usados para definir a função são executados.

O bloco de comandos que define uma função pode incluir chamadas a outras funções previamente definidas (incluindo a própria função, permitindo a chamada recursiva da própria função), as quais podem conter chamadas a outras funções pré-definidas e assim por diante.

**Funções são, portanto, os tijolos básicos de construção de um programa em C**, incluindo aqui as funções disponibilizadas na biblioteca padrão do C (ver slide 5), utilizadas para inúmeras finalidades.

Quando uma função necessita de entrada de dados para gerar algum resultado, dados que são denominados argumentos ou parâmetros, estes aparecem na chamada da função dentro de parênteses após o nome da função e separados por vírgulas:

**nome\_da\_função(arg1,arg2,arg3,...);**

Podemos interpretar o comando acima como uma instrução dada ao sistema para executar a função **nome\_da\_função**, usando como argumentos os valores arg1, arg2, ...

Quando uma função não requer argumentos, a notação do C ainda exige que coloquemos parênteses após a chamada da função, ainda que não haja nada dentro deles.

Algumas funções necessitam de um valor fixo de argumentos, enquanto outras podem fazer uso de um número variável, dependendo do contexto de uso da função.



## Introdução à funções

O que faz uma função depende inteiramente dos comandos usados na sua definição e dos valores particulares fornecidos aos seus argumentos (se houver argumentos). Um exemplo simples poderia ser a função `Cubo(N)`, que calcula o cubo de um argumento simples `N`, conforme veremos no próximo slide. Chamando a função `Cubo(2)` esta retorna o valor  $2^3 = 8$ .

O resultado de uma função pode ser um valor de retorno que será utilizado dentro do escopo da função chamadora ou pode ser simplesmente uma ação sem valor de retorno, como, por exemplo, a impressão de uma mensagem na janela do console.

**Durante a fase de integração das funções que constituem um programa, que é a fase final de desenvolvimento de um programa, após cada função que constitui o programa ter sido testada e validada por seu respectivo desenvolvedor, é crucial que olhemos para estas funções como "caixas pretas".** Clarificando, "caixa preta" no sentido de que não nos interessa os comandos dentro de cada função porque elas já foram testadas e validadas. Apenas nos interessa saber qual é o conjunto de argumentos, a especificação do tipo de cada argumento bem como o tipo do(s) valor(es) de retorno da função.

**Olhar para funções como "caixas pretas" validadas é crucial para um grupo de desenvolvedores, em que cada desenvolvedor desenvolve, testa e valida uma função para, ao final, este conjunto de funções validadas serem integradas em um único programa.**

Uma vez terminada a fase de integração de um programa para processamento de sinais este deve ser exaustivamente testado quanto à coerência operacional das funcionalidades de escrita e leitura na memória, quanto à coerência operacional das funcionalidades de I/O e quanto à coerência operacional das funcionalidades gráficas (se houver). Uma funcionalidade é coerente quando a sua execução durante a operação do programa ocorre conforme esperado e/ou conforme especificação inicial do programa.

Um bug em um programa é a ocorrência de incoerência operacional em alguma funcionalidade durante a execução do programa. Debugar um programa significa analisar, testar e corrigir o código fonte de modo a eliminar as incoerências operacionais das funcionalidades do programa observadas durante a sua execução.

## Introdução à funções

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/MostraNum.c> mostrado abaixo é um exemplo de uso de funções. Note a flexibilidade do C quanto à sintaxe da chamada de funções, permitindo o uso do resultado de uma expressão matemática como argumento (linha 69), o uso do resultado de uma atribuição como argumento (linha 79), o uso de uma chamada recursiva de função como argumento (linha 83), e por aí vai ... A execução do programa na janela do console é mostrada no slide 57.

```
1  /******  
2  * Este programa le dois numeros inteiros do teclado, faz operacoes entre eles  
3  * usando funcoes e operadores, e imprime na tela do console os valores  
4  * resultantes das operacoes. Uma funcao especifica MostraNum() mostra na tela  
5  * cada valor resultante da respectiva operacao, sendo o valor resultante  
6  * precedido de uma mensagem de texto (string) explicativa da operacao que deu  
7  * origem ao resultado.  
8  *****/  
9  /******  
10 * HEADERS:  
11 *****/  
12 #include <stdio.h>  
13 #include <stdlib.h>  
14 #include <string.h>  
15 #include <math.h>  
16 /******  
17 * PROTOTIPOS DAS FUNCOES:  
18 *****/  
19 void MostraNum(char *Txt, int Num);  
20 int Cubo(int n);  
21 /******  
22 * main()  
23 *****/  
24 void main(void) // nao ha argumentos e nao ha valor de retorno na main()  
25 {  
26     const int K1 = -2; /* variavel int K1 estah "congelada" pelo modificador const */  
27     /* K1 se comporta como uma constante e seu valor  
28     nao pode ser alterado ao longo do codigo fonte */
```

Sempre declarar o protótipo de cada função antes da main() conforme indicado pela seta e sempre expressar a definição de cada função após a main() conforme mostrado nos slides 56 e 57.

## Introdução à funções

```
29
30 const int K2 = 3; /* variavel int K2 estah "congelada" pelo modificador const */
31      /* K2 se comporta como uma constante e seu valor
32      nao pode ser alterado ao longo do codigo fonte */
33
34 int Resultado, NumA, NumB; /* declara variaveis inteiras automaticas*/
35      /* estas variaveis sao da classe de armazenamento auto por default */
36      /* ou seja, elas sao automaticamente criadas quando main() inicia */
37      /* e desaparecem quando main() termina. Elas sao inacessiveis fora */
38      /* do escopo da funcao main() */
39
40 /* declara strings de mensagens de texto: */
41 char Texto1[]="O valor de NumA eh";
42 char Texto2[]="O valor de NumB eh";
43 char Texto3[]="O valor de K1 eh";
44 char Texto4[]="O valor de K2 eh";
45 char Texto5[]="O valor de K1*NumA+K2*NumB eh";
46 char Texto6[]="O valor de Resultado=K2^3 eh";
47 char Texto7[]="O valor mostrado por MostraNum(Resultado++) eh";
48 char Texto8[]="O valor de --Resultado eh";
49 char Texto9[]="O valor de Resultado = NumA*(Resultado + NumB) eh";
50 char Texto10[]="O valor de NumA^3 eh";
51 char Texto11[]="O valor de (NumB+1)^3 eh";
52 char Texto12[]="O valor de ((NumA)^3)^3 eh";
53
54 system("cls"); /* limpa a tela do console */
55      /* cls eh um comando do command prompt do Windows */
56
57 /* imprime na tela do console: */
58 printf("Informe dois numeros inteiros separados por espaco:");
59
60 scanf("%u%u", &NumA, &NumB); /* interrompe a execucao, le dois valores do teclado
61 e armazena os valores lidos nos enderecos de memoria respectivos das
62 variaveis NumA e NumB */
```

## Introdução à funções

```
63
64  /* mostra na tela do console o resultado de diversas operacoes: */
65  MostraNum(Texto1,NumA);
66  MostraNum(Texto2,NumB);
67  MostraNum(Texto3,K1);
68  MostraNum(Texto4,K2);
69  MostraNum(Texto5,K1*NumA+K2*NumB);
70
71  Resultado = Cubo(K2); /*chama Cubo() com um argumento constante K2 */
72  MostraNum(Texto6,Resultado);
73
74  /* testa operadores de incremento e decremento: */
75  MostraNum(Texto7,Resultado++); MostraNum(Texto8,--Resultado);
76
77  Resultado += NumB; // Resultado = Resultado + NumB
78
79  MostraNum(Texto9,Resultado *= NumA); // Resultado = NumA*Resultado
80
81  MostraNum(Texto10,Cubo(NumA));
82  MostraNum(Texto11,Cubo(NumB+1));
83  MostraNum(Texto12,Cubo(Cubo(NumA)));
84  }
85
86  /******
87  * FUNC: void MostraNum(char *Txt,int Num)
88  *
89  * DESC: Mostra na tela do console o valor do argumento Num precedido do texto
90  *       armazenado na string Txt.
91  * *****/
92  void MostraNum(char *Txt, int Num) /* o argumento *Txt eh um ponteiro (um endereco)
93  /* da string passada como argumento na funcao chamadora */
94  {
95  printf("%s %d\n",Txt,Num);
96  }
97
```

# Introdução à funções

```
98  /******  
99  * FUNC: int Cubo(int n)  
100  *  
101  * DESC: Retorna o valor inteiro correspondente a  $n^3$ , onde int n eh o argumento  
102  *       da funcao.  
103  *****/  
104  int Cubo(int n)  
105  {  
106  return n*n*n; /* valor retornado por Cubo() */  
107  }
```

A execução do programa na janela do console é conforme mostrado abaixo:

```
Informe dois numeros inteiros separados por espaco:-5 4  
0 valor de NumA eh -5  
0 valor de NumB eh 4  
0 valor de K1 eh -2  
0 valor de K2 eh 3  
0 valor de K1*NumA+K2*NumB eh 22  
0 valor de Resultado=K2^3 eh 27  
0 valor mostrado por MostraNum(Resultado++) eh 27  
0 valor de --Resultado eh 27  
0 valor de Resultado = NumA*(Resultado + NumB) eh -155  
0 valor de NumA^3 eh -125  
0 valor de (NumB+1)^3 eh 125  
0 valor de ((NumA)^3)^3 eh -1953125
```

## Controle do fluxo do programa

O controle de fluxo é essencial em um programa em C. O controle de fluxo é responsável pela tomada de decisões como também é responsável pela execução de tarefas repetitivas e no tratamento de diversas situações no âmbito da execução do programa. Os principais comandos de controle de fluxo em C são: **if-else**, **switch case**, **loops (for, while, do-while)** e instruções **break** e **continue**.

### Declaração if-else:

A forma geral da declaração **if** é:

```
if(condição) {bloco_de_declarações_condição_V;}  
else {bloco_de_declarações_condição_F;}
```

A cláusula **else** é opcional. Se **condição** é V (i.e., !=0), então o **bloco\_de\_declarações\_condição\_V** é executado, caso contrário, i.e., se **condição** é F, então o **bloco\_de\_declarações\_condição\_F** é executado. Se o bloco de declarações é formado por uma única declaração não há necessidade das chaves { } delimitando a declaração.

O programa a seguir ilustra o uso do **if**. Este programa usa dois especificadores de formato para `scanf()` e `printf()`, que são o `%x` e `%o`, associados à leitura (`scanf()`) e impressão (`printf()`) respectivamente de dados hexadecimais e octais.

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/HexDec.c> mostrado no próximo slide é um exemplo de uso da declaração **if**.

## Controle do fluxo do programa

```
1 |
2 | /*****
3 |  * Este programa faz as seguintes conversoes de base numerica:
4 |  * decimal      --> hexadecimal
5 |  * hexadecimal  --> decimal
6 |  * decimal      --> octal
7 |  * octal        --> decimal
8 |  *****/
9 | /*****
10 |  * HEADERS:
11 |  *****/
12 | #include<stdio.h>
13 | /*****
14 |  * main()
15 |  *****/
16 | void main(void)
17 | {
18 |     int opcao;
19 |     int valor;
20 |
21 |     printf("Converter:\n");
22 |     printf("\t1: decimal para hexadecimal\n");
23 |     printf("\t2: hexadecimal para decimal\n");
24 |     printf("\t3: decimal para octal\n");
25 |     printf("\t4: octal para decimal\n");
26 |
27 |     printf("informe a sua opcao:");
28 |     scanf("%d", &opcao);
29 |
30 |     if(opcao==1) {
31 |         printf("informe um valor em decimal:");
32 |         scanf("%d", &valor);
33 |         printf("%d em hexadecimal eh %X", valor, valor);
34 |     }
```



## Controle do fluxo do programa

```
35 |
36 | if(opcao==2) {
37 |     printf("informe um valor em hexadecimal:");
38 |     scanf("%x", &valor);
39 |     printf("%x em decimal eh %d", valor, valor);
40 | }
41 |
42 | if(opcao==3){
43 |     printf("informe um valor em decimal:");
44 |     scanf("%d", &valor);
45 |     printf("%d em octal eh %o", valor, valor);
46 | }
47 |
48 | if(opcao==4){
49 |     printf("informe um valor em octal:");
50 |     scanf("%o", &valor);
51 |     printf("%o em decimal eh %d", valor, valor);
52 | }
53 | }
```

A execução do programa na janela do console é mostrada abaixo:

Converter:

- 1: decimal para hexadecimal
- 2: hexadecimal para decimal
- 3: decimal para octal
- 4: octal para decimal

informe a sua opcao:1

informe um valor em decimal:9874765

9874765 em hexadecimal eh 96AD4D



O código fonte abaixo é um exemplo de uso da declaração **if-else**.

```
/* Um exemplo de if-else.*/  
#include<stdio.h>  
void main(void)  
{  
    int i;  
    printf("informe um numero:");  
    scanf("%d", &i);  
  
    if(i<0) printf("o numero eh negativo");  
    else printf ("o numero eh positivo ou zero");  
}
```

A execução do programa na janela do console é mostrada abaixo:

```
informe um numero:3  
o numero eh positivo ou zero
```

```
informe um numero:-7  
o numero eh negativo
```

## Encadeamento if-else-if:

A forma geral é:

```
if(condição1)
{bloco_de_declarações_condição1_V;}
else if(condição2)
{bloco_de_declarações_condição2_V;}
else if(condição3)
{bloco_de_declarações_condição3_V;}
.
.
.
else
{bloco_de_declarações_todas_as_condições_anteriores_F;}
```

Se um bloco de declarações é formado por uma única declaração não há necessidade das chaves { } delimitando a declaração.

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/HexDec1.c> mostrado no próximo slide é um exemplo de uso do encadeamento **if-else-if**.

# Controle do fluxo do programa

```
1  /******  
2  * Este programa faz as seguintes conversoes de base numerica:  
3  * decimal    --> hexadecimal  
4  * hexadecimal --> decimal  
5  * decimal    --> octal  
6  * octal      --> decimal  
7  *****/  
8  /******  
9  * HEADERS:  
10 *****/  
11 #include<stdio.h>  
12 /******  
13 * main()  
14 *****/  
15 void main(void)  
16 {  
17     int opcao;  
18     int valor;  
19  
20     printf("Converter:\n");  
21     printf("\t1: decimal para hexadecimal\n");  
22     printf("\t2: hexadecimal para decimal\n");  
23     printf("\t3: decimal para octal\n");  
24     printf("\t4: octal para decimal\n");  
25  
26     printf("informe a sua opcao:");  
27     scanf("%d", &opcao);
```

## Controle do fluxo do programa

```
28 |
29 | if(opcao==1) {
30 |     printf("informe um valor em decimal:");
31 |     scanf("%d", &valor);
32 |     printf("%d em hexadecimal eh %X", valor, valor);
33 | }
34 |
35 | else if(opcao==2) {
36 |     printf("informe um valor em hexadecimal:");
37 |     scanf("%x", &valor);
38 |     printf("%x em decimal eh %d", valor, valor);
39 | }
40 |
41 | else if(opcao==3){
42 |     printf("informe um valor em decimal:");
43 |     scanf("%d", &valor);
44 |     printf("%d em octal eh %o", valor, valor);
45 | }
46 |
47 | else if(opcao==4){
48 |     printf("informe um valor em octal:");
49 |     scanf("%o", &valor);
50 |     printf("%o em decimal eh %d", valor, valor);
51 | }
52 | else{
53 |     printf("opcao invalida!");
54 | }
55 | }
```

## Controle do fluxo do programa

A execução do programa na janela do console é mostrada abaixo:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:1
informe um valor em decimal:16
16 em hexadecimal eh 10
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:4
informe um valor em octal:2036
2036 em decimal eh 1054
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:7
opcao invalida!
```

## Controle do fluxo do programa

Importante notar que o `if()` decide V ou F com base no valor lógico `0=F`, `!0=V` de qualquer expressão colocada em `()`. Por exemplo, considere o programa abaixo, que faz o tratamento da situação de divisão por zero:

```
/* Divide o primeiro numero pelo segundo */  
#include<stdio.h>  
void main(void)  
{  
  int a, b;  
  printf("informe dois numeros inteiros separados por espaco: ");  
  scanf("%d%d", &a, &b);  
  if(b) printf("%d/%d = %f\n",a,b,(float) a/b); /* soh executa se b!=0 */  
  else printf("nao posso dividir por zero\n");  
}
```

A execução do programa na janela do console é mostrada abaixo:

informe dois numeros inteiros separados por espaco: 2 3

2/3 = 0.666667

informe dois numeros inteiros separados por espaco: -7 5

-7/5 = -1.400000

informe dois numeros inteiros separados por espaco: 8 0

nao posso dividir por zero

Note que não é necessário explicitamente usar **`“if(b != 0) printf(“%d/%d = %f\n”,a,b,(float) a/b);”`** porque o **`if()`** é capaz de decidir V ou F com base no valor lógico `0=F` ou `!0=V` do denominador da divisão armazenado na variável inteira **`“b”`**.

## Declaração switch:

A declaração **switch** difere da declaração **if-else-if** por só poder testar igualdades. No entanto, gera um código executável mais rápido do que **if-else-if**. A forma geral da declaração **switch** é:

```
switch(variável) {  
    case constante1:  
        {bloco_de_declarações_variável==constante1;}  
        break;  
    case constante2:  
        {bloco_de_declarações_variável==constante2;}  
        break;  
    case constante3:  
        {bloco_de_declarações_variável==constante3;}  
        break;  
    .  
    .  
    .  
    default:  
        {bloco_de_declarações_variável_não_igual_a_qualquer_constante}  
}
```

Se um bloco de declarações é formado por uma única declaração não há necessidade das chaves { } delimitando a declaração.

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/HexDec2.c> mostrado no próximo slide é um exemplo de uso do **switch**.

# Controle do fluxo do programa

```
1  /******  
2  * Este programa faz as seguintes conversoes de base numerica:  
3  * decimal    --> hexadecimal  
4  * hexadecimal --> decimal  
5  * decimal    --> octal  
6  * octal      --> decimal  
7  *****/  
8  /******  
9  * HEADERS:  
10 *****/  
11 #include<stdio.h>  
12 /******  
13 * main()  
14 *****/  
15 void main(void)  
16 {  
17     int opcao;  
18     int valor;  
19  
20     printf("Converter:\n");  
21     printf("\t1: decimal para hexadecimal\n");  
22     printf("\t2: hexadecimal para decimal\n");  
23     printf("\t3: decimal para octal\n");  
24     printf("\t4: octal para decimal\n");  
25  
26     printf("informe a sua opcao:");  
27     scanf("%d", &opcao);  
28
```



# Controle do fluxo do programa

```
29 switch(opcao) {
30
31 case 1: {
32     printf("informe um valor em decimal:");
33     scanf("%d", &valor);
34     printf("%d em hexadecimal eh %X", valor, valor);
35     break;
36 }
37
38 case 2: {
39     printf("informe um valor em hexadecimal:");
40     scanf("%x", &valor);
41     printf("%x em decimal eh %d", valor, valor);
42     break;
43 }
44
45 case 3: {
46     printf("informe um valor em decimal:");
47     scanf("%d", &valor);
48     printf("%d em octal eh %o", valor, valor);
49     break;
50 }
51
52 case 4:{
53     printf("informe um valor em octal:");
54     scanf("%o", &valor);
55     printf("%o em decimal eh %d", valor, valor);
56     break;
57 }
58
59 default:{
60     printf("opcao invalida!");
61 }
62 }
63 }
64 }
```

## Controle do fluxo do programa

A execução do programa na janela do console é mostrada abaixo:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:3
informe um valor em decimal:16
16 em octal eh 20
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:1
informe um valor em decimal:16
16 em hexadecimal eh 10
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Converter:
    1: decimal para hexadecimal
    2: hexadecimal para decimal
    3: decimal para octal
    4: octal para decimal
informe a sua opcao:5
opcao invalida!
```

## Loop for:

A forma geral do *loop for* é:

**for(inicialização; condição; incremento) {bloco\_de\_declarações;}**

- **inicialização** é uma declaração de atribuição que inicializa a variável de controle do *loop*.
- **condição** é uma expressão relacional que determina qual situação terminará a execução recorrente do bloco de declarações do *loop* através do teste lógico (V ou F) da variável de controle em relação a um determinado valor numérico. Uma vez que a condição resulte em F o *loop* é desativado e o programa seguirá na declaração seguinte ao término do escopo do *loop for*.
- **incremento** define a maneira que variável de controle varia ao longo da execução do *loop*.
- **bloco de declarações** é a sequência de declarações que o *loop for* recorrentemente executa enquanto ativo. Se o bloco de declarações é formado por uma única declaração não há necessidade das chaves { } delimitando a declaração.

Seguem exemplos de uso do *loop for* e respectivos resultados de execução na tela do console:

```
/* Imprime numeros de 100 a 1 na tela do console*/  
#include <stdio.h>  
void main(void)  
{  
    int x;  
    for(x=100; x>0; x--) printf("%d ",x);  
}
```

Execução na tela do console:

```
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62  
61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22  
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

## Controle do fluxo do programa

```
/* Imprime numeros de 0 a 100 de 5 em 5 */  
#include <stdio.h>  
void main(void)  
{  
    int x;  
    for(x=0; x<=100; x=x+5) printf("%d ",x);  
}
```

Execução na tela do console:

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

```
/* Imprime o quadrado dos numeros de 0 a MAXNUM */  
#include <stdio.h>  
#define QDR(u) (u*u) // macro q calcula o quadrado de u, i.e., u^2  
void main(void)  
{  
#define MAXNUM 10 // define o valor de MAXNUM  
    int i;  
    for(i=0; i<MAXNUM+1; i++){  
        printf("i = %d",i);  
        printf(" -> i^2 = %d\n", QDR(i));  
    }  
}
```

Execução na tela do console:

i = 0 -> i<sup>2</sup> = 0  
i = 1 -> i<sup>2</sup> = 1  
i = 2 -> i<sup>2</sup> = 4  
i = 3 -> i<sup>2</sup> = 9  
i = 4 -> i<sup>2</sup> = 16  
i = 5 -> i<sup>2</sup> = 25  
i = 6 -> i<sup>2</sup> = 36  
i = 7 -> i<sup>2</sup> = 49  
i = 8 -> i<sup>2</sup> = 64  
i = 9 -> i<sup>2</sup> = 81  
i = 10 -> i<sup>2</sup> = 100

*/\* este loop for nunca eh executado porque o teste condicional eh realizado no início da execucao do loop. Isto significa que printf("%d", y); nunca eh executado porque a condicao eh F logo no inicio do loop \*/*

```
#include<stdlib.h>
```

```
#include<stdio.h>
```

```
void main(void){
```

```
int y, x=10;
```

```
for(y=10; y!=x; ++y) printf("%d", y);
```

```
printf("fim do loop");
```

```
}
```

Execução na tela do console:

fim do loop

Este resultado da execução na tela do console acontece porque o *loop for* nunca é executado dado que  $x=y$  já no início do *loop*. Portanto a condição “ $y!=x$ ” é avaliada como falsa e o *loop* é terminado. A seguir, é executado “`printf("fim do loop");`” e o programa termina.

## Controle do fluxo do programa

**`/* Este programa imprime os numeros de 0 a 98 de 2 em 2 na tela do console */`**

**`#include<stdio.h>`**

**`#include<stdlib.h>`**

**`void main(void)`**

**`{`**

**`int x, y;`**

**`for(x=0,y=0; x+y<100; ++x,y++) /* Duas variaveis de controle: x e y */`**

**`printf("%d ", x+y);`**

**`}`**

**`/* Em C a virgula eh um operador que significa "faz isto E aquilo". Essa  
caracteristica do C valida a construcao do loop for com duas ou mais variaveis  
de controle. O incremento pode ser todas as combinacoes de incremento anterior (++x)  
e posterior (x++), jah que neste caso o fato do incremento ser anterior ou posterior  
nao afeta a execucao do loop */`**

Execução na tela do console:

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82  
84 86 88 90 92 94 96 98

```
/* teste sua capacidade de efetuar a soma de 2 numeros*/  
#include<stdio.h>  
#include<conio.h>  
void main(void)  
{  
    int i, j, resposta;  
    char fim = ' ';  
  
    for(i=1; i<100 && fim!='N';i++){ /* loop ativo enquanto i<100 e fim != 'N' */  
        for(j=1; j<10; j++) {  
            printf("Quanto eh %d + %d? ", i, j);  
            scanf("%d", &resposta);  
            if(resposta != i+j) printf("Errado!\n");  
            else printf("Correto!\n");  
        }  
        printf("Continuar? (digite N para nao continuar)");  
        fim=getche(); /* A função getche() le um único caractere do teclado  
            que eh exibido imediatamente na tela do console  
            sem esperar pela tecla <Enter>. getche() nao eh  
            definida na biblioteca padrao do C, mas seu  
            prototipo estah definido no header conio.h */  
        printf("\n");  
    }  
}
```

### Execução na tela do console:

Quanto eh  $1 + 1$ ? 2

Correto!

Quanto eh  $1 + 2$ ? 3

Correto!

Quanto eh  $1 + 3$ ? 4

Correto!

Quanto eh  $1 + 4$ ? 6

Errado!

Quanto eh  $1 + 5$ ? 6

Correto!

Quanto eh  $1 + 6$ ? 7

Correto!

Quanto eh  $1 + 7$ ? 9

Errado!

Quanto eh  $1 + 8$ ? 9

Correto!

Quanto eh  $1 + 9$ ? 10

Correto!

Continuar? (digite N para nao continuar)N



## Controle do fluxo do programa

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/QuadrLoco.c> a seguir dá uma ideia do grau de liberdade que o C concede ao programador, em particular quanto à sintaxe flexível e compacta dos comandos. Há quem critique esta liberdade – frequentemente o empresário que contrata o programador é um crítico desta liberdade ...

```
1  /* *****
2  /* Este programa recorrentemente le um numero inteiro do teclado, calcula seu
3  /* quadrado e imprime o resultado na tela do console. O loop de recorrência
4  /* eh interrompido quando o numero lido do teclado eh 0.
5  /* *****
6  /* *****
7  /* HEADERS:
8  /* *****
9  #include <stdio.h>
10 #include <stdlib.h>
11 /* *****
12 /* PROTOTIPOS DAS FUNCOES:
13 /* *****
14 void Prompt(void);
15 int LeNum(void);
16 void PrintQuadrNum(int Num);
17 /* *****
18 /* main()
19 /* *****
20 void main(void)
21 {
22     int t;
23
24     for(Prompt(); t=LeNum(); Prompt()) /* se t==0 loop for termina */
25         PrintQuadrNum(t); /* Imprime na tela do console o quadrado do argumento t
26             /* enquanto t!=0 */
27
28 }
29 /* O loop for acima funciona porque (inicializacao;condicao;incremento)
30 /* podem ser qualquer expressao valida em C */
31
```

## Controle do fluxo do programa

```
32 /*****  
33  * FUNC: void Prompt(void)  
34  *  
35  * DESC: Imprime na tela do console "Digite um numero inteiro (0 p/ encerrar):"  
36  *****/  
37 void Prompt(void)  
38 {  
39     printf("Digite um numero inteiro (0 p/ encerrar):");  
40 }  
41 /*****  
42  * FUNC: int LeNum(void)  
43  *  
44  * DESC: Le do teclado um numero inteiro e retorna o valor do numero lido  
45  *****/  
46 int LeNum(void)  
47 {  
48     int t;  
49     scanf("%d",&t);  
50     return t;  
51 }  
52 /*****  
53  * FUNC: void PrintQuadrNum(int Num)  
54  *  
55  * DESC: Imprime na tela do console o quadrado do argumento Num  
56  *****/  
57 void PrintQuadrNum(int Num)  
58 {  
59     printf("O quadrado do numero %d eh %d.\n",Num, Num*Num);  
60 }
```

### Execução na tela do console:

```
Digite um numero inteiro (0 p/ encerrar):1
O quadrado do numero 1 eh 1.
Digite um numero inteiro (0 p/ encerrar):-1
O quadrado do numero -1 eh 1.
Digite um numero inteiro (0 p/ encerrar):2
O quadrado do numero 2 eh 4.
Digite um numero inteiro (0 p/ encerrar):7
O quadrado do numero 7 eh 49.
Digite um numero inteiro (0 p/ encerrar):8
O quadrado do numero 8 eh 64.
Digite um numero inteiro (0 p/ encerrar):0
```

## Controle do fluxo do programa

Ainda com relação à liberdade que o C concede ao programador, note ainda que é permitido deixar faltando partes da definição do *loop for*. Por exemplo, o programa abaixo lê a senha do teclado e testa se é igual 135. Se for igual, **condição** se torna falsa e o *loop for* encerra com a mensagem "Senha correta!". Note que falta o **incremento** na declaração (**inicialização; condição; incremento**).

```
/* le recorrentemente a senha do teclado e  
se a senha digitada eh correta (135) encerra  
o loop de recorrencia e imprime a mensagem "Senha  
correta!" na tela do console */
```

```
#include <stdio.h>  
void main(void)  
{  
    int x;  
    puts("Digite a senha:");  
    for(x=0; x!=135;) scanf("%d", &x);  
    printf("Senha correta!");  
}
```

A função puts() está descrita na seção "2.12.5.10 puts" na página 109 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf> .

Execução na tela do console:

Digite a senha:

17

79

568

135

Senha correta!

## Controle do fluxo do programa

```
/* loop infinito */  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
void main(void){  
    for(;;) {  
        printf("Este loop serah executado para sempre!\n");  
        printf("Pressione CTRL+C para interromper!\n\n");  
    }  
}
```

Execução na tela do console:

```
Este loop serah executado para sempre!  
Pressione CTRL+C para interromper!
```

```
Este loop serah executado para sempre!  
Pressione CTRL+C para interromper!
```

```
Este loop serah executado para sempre!  
Pressione CTRL+C para interromper!
```

```
Este loop serah executado para sempre!  
Pressione CTRL+C para interromper!
```

```
^C
```

**CTRL+C**

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>
```

## O comando break:

**/\* Interupcao do loop for com o comando break.**

**Le recorrentemente um caractere do teclado ateh que  
seja digitado o caractere 'A', quando entao o  
loop de recorrancia eh interrompido \*/**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <conio.h>**

**void main(void)**

**{**

**char ch;**

**puts("\nDigite varios caracteres em sequencia. Se voce digitar A o programa termina.\n");**

**for(;;){**

**ch=getche(); /\* le um caractere do teclado \*/**

**if(ch == 'A') break; /\* break: sai do loop e vai p/ a declaracao seguinte \*/**

**}**

**printf("\n\nVoce digitou um A! Tchau!\n");**

**}**

Execução na tela do console:

Digite varios caracteres em sequencia. Se voce digitar A o programa termina.

asvffkHFHKKA

Voce digitou um A! Tchau!

## Loop while:

A forma geral do loop **while** é :

```
while(condição) {bloco_de_declarações;}
```

O **bloco de declarações** é recorrentemente executado enquanto **condição** é verdadeira ( $V \neq 0$ ). Se o bloco de declarações é formado por uma única declaração não há necessidade das chaves **{ }** delimitando a declaração. Note que se **condição** é falsa ( $F = 0$ ) no teste de **condição** na primeira iteração do *loop* o **bloco de declarações** nunca será executado.

Seguem exemplos de uso do *loop for* e respectivos resultados de execução na tela do console.

```
/* Le recorrentemente um caractere do teclado ateh que  
seja digitado o caractere 'A', quando entao o  
loop while que faz a recorrencia eh interrompido */  
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
void main(void)  
{  
char ch='\0'; /*inicializa ch com o caractere nulo*/  
  
puts("\nDigite varios caracteres em sequencia. Se voce  
digitar A o programa termina.\n");  
  
while(ch!='A') ch= getche(); /* le um caractere do teclado  
ateh que 'A' seja digitado */  
  
printf("\n\nVoce digitou um A! Tchau!\n");  
}
```

Execução na tela do console:

Digite varios caracteres em sequencia. Se voce digitar A  
o programa termina.

asdrkdfklFHJJA

Voce digitou um A! Tchau!

```
/* contagem regressiva 10 ateh 0 para explosao "KABUM!" */  
# include <stdio.h>  
void main(void)  
{  
    int contagem = 10;  
  
    while (contagem>=0) printf("Contagem = %d\n",contagem--);  
    puts("KABUM!");  
}
```

Execução na tela do console:

```
Contagem = 10  
Contagem = 9  
Contagem = 8  
Contagem = 7  
Contagem = 6  
Contagem = 5  
Contagem = 4  
Contagem = 3  
Contagem = 2  
Contagem = 1  
Contagem = 0  
KABUM!
```



## Controle do fluxo do programa

```
/* contagem regressiva 10 ateh 0 - aborta explosao "KABUM!" em contagem == ABORTA_EM */
# include <stdio.h>
#define ABORTA_EM 7
void main(void)
{
    int contagem = 10;

    while (contagem>=0)    {
        if(contagem==ABORTA_EM)
        {
            printf("Contagem abortada em %d!\n",contagem);
            break; // interrompe o loop while
        }

        printf("Contagem = %d\n",contagem--);
    }

    if(contagem!=ABORTA_EM) puts("KABUM!");
}
```

Execução na tela do console:

Contagem = 10

Contagem = 9

Contagem = 8

Contagem abortada em 7!

### *Loop infinito com while:*

```
while(1) {bloco_de_declarações;}
```

é idêntico a

```
for(;;) {bloco_de_declarações;}
```

dado que a **condição** em **while(condição)** é sempre  $1 \neq 0 = V$ .

## Loop do-while (loop “faça enquanto”):

do

{bloco\_de\_declarações;}

while(condição);

O **bloco de declarações** é recorrentemente executado enquanto **condição** é verdadeira ( $V=1$ ). Se o bloco de declarações é formado por uma única declaração não há necessidade das chaves `{ }` delimitando a declaração. Note que se **condição** é falsa ( $F=0$ ) no teste de **condição** na primeira iteração do *loop*, mesmo assim o **bloco de declarações** é executado uma única vez.

Seguem exemplos de uso do *loop do-while* e respectivos resultados de execução na tela do console.

```
/* le numeros do teclado e imprime na tela do console ateh que um deles seja negativo */
```

```
#include<stdio.h>
```

```
void main(void)
```

```
{
```

```
    int num;
```

```
    puts("\nDigite numeros inteiros positivos.
```

```
    Se o numero for negativo o programa encerra.\n");
```

```
do {
```

```
    printf("Digite: ");
```

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

```
    printf("Voce digitou %d.\n\n", num);
```

```
    }while(num>0);
```

```
    puts("Voce digitou um numero negativo. Tchau!\n");
```

```
}
```

Execução na tela do console:

Digite numeros inteiros positivos. Se o numero for negativo o programa encerra.

Digite: 25

Voce digitou 25.

Digite: 49

Voce digitou 49.

Digite: -2

Voce digitou -2.

Voce digitou um numero negativo. Tchau!

## Controle do fluxo do programa

Uma aplicação usual do *loop do-while* é na seleção de opções em um menu, garantindo que o usuário digite uma opção válida. Conforme mostrado abaixo, depois que `scanf()` adquire o valor da opção digitada pelo usuário, o *loop* é recorrentemente executado até que o usuário digite uma opção válida:

```
/* reapresenta recorrentemente o menu de opcoes ateh que o usuario digite uma opcao valida*/  
#include <stdio.h>  
void main(void)  
{  
    int opcao;  
    do{  
        printf("Converter:\n");  
        printf("    1: decimal para hexadecimal\n");  
        printf("    2: hexadecimal para decimal\n");  
        printf("    3: decimal para octal\n");  
        printf("    4: octal para decimal\n");  
        printf("Digite a sua opcao: ");  
        scanf("%d", &opcao);  
    } while(opcao<1 | opcao>4);  
  
    printf("Voce digitou a opcao %d.", opcao);  
  
}
```

Execução na tela do console:

Converter:

- 1: decimal para hexadecimal
- 2: hexadecimal para decimal
- 3: decimal para octal
- 4: octal para decimal

Digite a sua opcao: 7

Converter:

- 1: decimal para hexadecimal
- 2: hexadecimal para decimal
- 3: decimal para octal
- 4: octal para decimal

Digite a sua opcao: 3

Voce digitou a opcao 3.

## Loop dentro de loop (loops “aninhados”):

Em geral *loops* aninhados são usados para gerar matrizes, tabelas, ou qualquer estrutura de dados bidimensional. O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/PowTable.c> a seguir mostra um exemplo de *loops* aninhados. O resultado da execução na tela do console é mostrado no próximo slide.

```
1  /*Imprime a tabela das MAXPOT primeiras potencias dos numeros de 1 a MAXNUM.*/
2  #include<stdio.h>
3  #define MAXNUM 10
4  #define MAXPOT 7
5  void main(void)
6  {
7      int i, j, k, temp;
8
9      /* imprime na tela do console a primeira linha da tabela: */
10     printf("          i",j);
11     for(j=2; j<(MAXPOT+1); j++)printf("          i^%d",j);
12     printf("\n");
13
14     /* corpo da tabela: */
15     for(i=1;i<(MAXNUM+1); i++){ /* laço externo: define o numero base i*/
16         for(j=1; j<(MAXPOT+1); j++){ /* 1º nivel de aninhamento - calcula as potencias i^j */
17             temp = 1;
18             for (k=0; k<j; k++) temp = temp*i; /* 2º nivel de aninhamento
19                                     - laco mais interno: calcula i^j */
20             printf("%9d",temp); /* o numero 9 colocado entre o % e o d
21                                     faz com que printf() use um campo de
22                                     9 caracteres numericos a cada valor impresso */
23         } // for j
24
25         printf("\n"); // newline
26     } // for i
27 }
```

## Controle do fluxo do programa

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
  i      i^2      i^3      i^4      i^5      i^6      i^7
  1        1        1        1        1        1        1
  2        4        8       16       32       64      128
  3        9       27       81      243      729     2187
  4       16       64      256     1024     4096    16384
  5       25      125      625     3125    15625    78125
  6       36      216     1296     7776    46656   279936
  7       49      343     2401    16807   117649   823543
  8       64      512     4096    32768   262144  2097152
  9       81      729     6561    59049   531441  4782969
 10      100     1000    10000   100000  1000000 10000000
```

## Loops com temporização:

Em determinadas aplicações é necessário medir o tempo entre dois eventos durante a execução de um programa. Em geral isto é efetuado através da função **time()**, função que retorna quantos segundos transcorreram desde a hora 00:00:00 UTC de 1º de janeiro de 1970. O uso da função **time()** está formalmente descrito na seção "2.15 time.h" na página 135 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide - Huss.pdf>. Exemplos podem ser encontrados em <https://www.geeksforgeeks.org/time-function-in-c/>.

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/Timer.c> a seguir mostra um exemplo de temporização de um *loop for*. O resultado da execução na tela do console é mostrado no próximo slide.

```
1  /* Este programa verifica o seu senso de tempo */
2  #include <stdio.h>
3  #include <time.h>
4  #include <conio.h>
5  #define DELTA_TEMPO 5
6  void main(void)
7  {
8      unsigned long tm;
9      printf("\nEste programa testa o seu senso de tempo.\n");
10     printf("Pressione a tecla <ENTER>, espere %u segundos\n", DELTA_TEMPO);
11     printf("e pressione qualquer tecla.");
12     getche();
13     printf("\n");
14
15     tm = time(0); /* retorna o tempo transcorrido em segundos desde a
16                  hora 00:00:00 UTC de 1º de janeiro de 1970 */
17     for (;;) if(kbhit()) break; /* kbhit() eh definida em conio.h e retorna um valor
18                                diferente de zero quando uma tecla eh pressionada no teclado */
19     if(time(0)-tm==DELTA_TEMPO)
20     printf("Voce acertou! Voce tem um bom senso de tempo.\n");
21     else
22     printf("Voce errou. Erro de %d segundos!\n", (int) time(0)-tm-DELTA_TEMPO);
23 }
```

## Controle do fluxo do programa

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Este programa testa o seu senso de tempo.  
Pressione a tecla <ENTER>, espere 5 segundos  
e pressione qualquer tecla.  
Voce acertou! Voce tem um bom senso de tempo.
```

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Este programa testa o seu senso de tempo.  
Pressione a tecla <ENTER>, espere 5 segundos  
e pressione qualquer tecla.  
Voce errou. Erro de -2 segundos!
```

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
```

```
Este programa testa o seu senso de tempo.  
Pressione a tecla <ENTER>, espere 5 segundos  
e pressione qualquer tecla.  
Voce errou. Erro de 2 segundos!
```

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos> |
```



### Interrompendo com **break** o *loop* dentro de um *loop* (*loops* “aninhados”):

O **break** no programa abaixo interrompe somente o *loop for* interno. Se quiséssemos interromper o *loop for* externo seria necessário um **break** fora do escopo de *loop* interno mas dentro do escopo do *loop* externo. Mesma observação vale para **while** ou **switch**: Um **break** usado dentro de uma declaração **while** ou **switch** somente tem efeito dentro do escopo relacionado com este **while** ou **switch** mas não tem efeito sobre qualquer outro *loop* externo. Idem para **do-while**.

*/\* Este código imprime 40 vezes na tela do console os numeros de 1 a 9. A cada vez que o break eh encontrado, o controle eh devolvido para o loop for externo.\*/*

```
#include<stdio.h>
void main(void)
{
    int count = 1;
    int t;

    for(t=0; t<40; t++){
        count=1;
        for(;;){
            printf("%d", count);
            count++;
            if(count==10) break;
        }
        printf(" ");
    }
}
```

Execução na tela do console:

```
123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789
123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789
123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789
123456789 123456789 123456789 123456789
```

## O comando continue:

**Continue** opera da mesma maneira que o **break**, mas em vez de terminar o *loop* e seguir na declaração seguinte ao *loop*, o comando **continue** força a ativação da próxima recorrência do *loop* (= próxima iteração do *loop*), pulando qualquer código seguinte ao comando **continue**. Seguem exemplos de uso de **continue**.

```
/* Imprime na tela os numeros pares ateh 99 */
#include<stdio.h>
void main(void)
{
    int x;
    for(x=0; x<100; x++){
        if(x%2) continue; /* se o resto nao eh zero (indicando que x
                           nao eh par) inicia a proxima recorrencia
                           do loop sem executar o printf() */
        printf("%d ", x);
    }
}
```

Execução na tela do console:

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82  
84 86 88 90 92 94 96 98

## Controle do fluxo do programa

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/TxtEncoder.c> a seguir mostra um outro exemplo de uso de **continue**. O resultado da execução na tela do console é mostrado no slide 97.

```
1  /*****  
2  * Este programa le um caractere do teclado, incrementa  
3  * de 1 o valor ASCII do caractere e imprime na tela do  
4  * console o caractere resultante. O processo continua ateh  
5  * que o caracter $ seja digitado.  
6  *****/  
7  /*****  
8  * HEADERS:  
9  *****/  
10 #include<stdio.h>  
11 #include<conio.h>  
12 /*****  
13 * PROTOTIPOS DAS FUNCOES:  
14 *****/  
15 void Encode(void);  
16 /*****  
17 * main()  
18 *****/  
19 void main(void)  
20 {  
21     printf("Digite a sequencia de caracteres ASCII que voce quer codificar.\n");  
22     printf("Note que cada caractere ASCII digitado eh impresso na tela\n");  
23     printf("do console com seu valor ASCII incrementado de 1.\n");  
24     printf("Digite $ quando a sequencia terminar.\n");  
25  
26     Encode();  
27 }
```

## Controle do fluxo do programa

```
28 /******  
29 */ FUNC: void Encode(void)  
30 *  
31 * DESC: Le um caractere do teclado, incrementa de 1 o valor ASCII do caractere  
32 * e imprime na tela do console o caractere resultante. O processo continua  
33 * ateh que o caracter $ seja digitado.  
34 *****  
35 void Encode(void)  
36 {  
37     char pronto = 0, ch;  
38  
39     while(!pronto){  
40  
41         ch=getch();  
42         printf("%c", ch+1); /* imprime ch com valor ASCII aumentado de 1 */  
43  
44         if(ch=='$'){  
45             pronto=1;  
46             continue;  
47         }  
48  
49     }  
50 }
```

## Controle do fluxo do programa

Execução na tela do console quando é digitado “abcdefg\$”:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>prgm
Digite a sequencia de caracteres ASCII que voce quer codificar.
Note que cada caractere ASCII digitado eh impresso na tela
do console com seu valor ASCII incrementado de 1.
Digite $ quando a sequencia terminar.
bcdefgh%
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\Exemplos>
```

## A declaração goto label:

A declaração **goto** faz a execução de uma função “pular” do ponto dentro do escopo da função onde **goto label** é declarado para outro ponto dentro do escopo da mesma função onde **label:** é declarado. Note que **goto label** somente pode ser utilizado para “pulos” dentro do escopo da mesma função, não podendo “pular” a execução do programa de uma função para outra.

```
/* O código abaixo implementa um loop de 1 a 100 usando goto */
#include<stdio.h>
void main(void){
int x=1;
Loop1: /* note a sintaxe do label, que sempre termina com ":" */
printf("%d ",x++); // imprime x e depois incrementa x
if(x<=100)goto Loop1;
}
```

Execução na tela do console:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

## Controle do fluxo do programa

O goto é útil quando é necessário sair de uma rotina profundamente aninhada, como é o caso abaixo:

```
for(...){  
  for(...){  
    while(...){  
      if(...)goto Lstop; /* se ocorre determinada situação pula p/ o label Lstop */  
      .  
      .  
      .  
    }  
  }  
}  
  
Lstop:  
  printf("erro no programa\n")
```

## Controle do fluxo do programa

Note que se usarmos um **break** em vez do **goto** seremos forçados a usar uma série de testes adicionais, já que, conforme vimos no slide 93, o **break** somente interrompe o *loop* de cujo escopo ele faz parte:

```
stop=0; /* stop é a variável de controle da saída da situação detectada pelo if */
for(...){
    for(...){
        while(...){
            if(...) /* se ocorre determinada situação stop=1 */
            {stop=1;
             break;}
        }
        .
        .
        .
    }
    if(stop)break;
}
if(stop) break;
}

printf("erro no programa\n")
```



## Vetores (matrizes unidimensionais):

A forma geral de um vetor é

**tipo nome[tamanho]**

- **tipo** declara o tipo de variável que identifica os elementos do vetor (**char**, **int**, **float**, **double**, **COMPLEX**, etc.). Veremos como criar e operar vetores e matrizes do tipo de dado **COMPLEX** no Cap III.

- **nome** define o nome do vetor.

- **tamanho** é o número de elementos armazenados no vetor

Por exemplo, abaixo é declarado e inicializado um vetor de 5 elementos do tipo **float** (32 bits = 4bytes) que armazena os valores dos respectivos 5 coeficientes de um filtro digital:

```
float CoefFiltro[]={0.12, 0.34, -0.97, -1.41, 0.77}; // CoefFiltro[] ocupa 5*32bits = 160 bits = 20 bytes de memoria.
```

Em C todos os vetores e matrizes tem o zero como índice do seu primeiro elemento. Portanto, a declaração acima aloca espaço na memória para o vetor **CoefFiltro[]**, de 5 elementos tipo **float**, que são **CoefFiltro[0]=0.12**, **CoefFiltro[1]=0.34**, **CoefFiltro[2]=-0.97**, **CoefFiltro[3]=-1.41** e **CoefFiltro[4]=0.77**.

Se estivermos desenvolvendo um programa para a camada física de um sistema de comunicação digital, muito possivelmente estaremos trabalhando com números de valor complexo (i.e., com parte real e imaginária). Neste caso os filtros têm coeficientes de valor complexo, conforme declarado abaixo:

```
COMPLEX CoefFiltro[5];
```

A declaração acima aloca espaço na memória para o vetor **CoefFiltro[]**, de 5 elementos do tipo **COMPLEX**, que são **CoefFiltro[0]**, **CoefFiltro[1]**, **CoefFiltro[2]**, **CoefFiltro[3]**, **CoefFiltro[4]**.

## Vetores, strings e matrizes

O código fonte <https://www.fccdecastro.com.br/CursoC&C++/C/AvgStdDev.c> a seguir mostra um exemplo de uso de vetores. Este programa lê NUM\_ELEM valores em ponto flutuante do teclado, armazena os valores no vetor Dat[NUM\_ELEM] e calcula a media Avg e o desvio padrão StdDev dos elementos armazenados em Dat[]. A seguir imprime Dat[], Avg, StdDev e a variância StdDev^2 na tela do console. Ver [https://pt.wikipedia.org/wiki/Desvio\\_padr%C3%A3o](https://pt.wikipedia.org/wiki/Desvio_padr%C3%A3o). O resultado da execução na tela do console é mostrado no slide 107.

```
1  /*****
2  * Este programa le NUM_ELEM valores em ponto flutuante do teclado,
3  * armazena os valores no vetor Dat[NUM_ELEM] e calcula a media Avg e o desvio
4  * padrao StdDev dos elementos armazenados em Dat[]. A seguir imprime
5  * Dat[], Avg, StdDev e a variancia StdDev^2 na tela do console.
6  *****/
7  /*****
8  * HEADERS:
9  *****/
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <math.h>
13 #include <time.h>
14 /*****
15 * PROGRAM DEFINITIONS:
16 *****/
17 #define NUM_ELEM 5
18 /*****
19 * MACROS:
20 *****/
21 static double sqrarg;
22 #define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
23 /*****
24 * FUNCTION PROTOTYPES:
25 *****/
26 float Mean (float *Vet, unsigned NumElem);
27 float SDev(float *Vet, float Mean, unsigned NumElem);
```

A macro SQR(a) usa o **operador ternário condicional (?:)** do C. A macro atribui o valor do argumento “a” à variável global sqrarg (double e static) e daí usa o operador ternário condicional: Se a==0.0 a macro SQR(a) retorna 0.0, caso contrário retorna sqrarg\*sqrarg.

**Nota 1:** Sempre usar variáveis double quando for calculado potências numéricas dado à insuficiente precisão da quantização de variáveis float neste caso. **Nota 2:** Variáveis static têm a propriedade de preservar seu valor mesmo depois de saírem de seu escopo. Portanto, uma variável static preserva o valor nela armazenado no escopo anterior e seu valor não é re-inicializado em qualquer novo escopo.



## Vetores, strings e matrizes

```
28- /******
29-  * main():
30-  *****/
31- void main(void){
32-
33-     float Avg, StdDev;
34-     unsigned i;
35-     float Dat[NUM_ELEM];
36-
37-     /* imprime na tela do console as instrucoes de uso */
38-     printf("\nEste programa le %u valores em ponto flutuante do teclado,", NUM_ELEM);
39-     printf("armazena cada i-esimo valor digitado em Dat[i], e dai imprime\n");
40-     printf("na tela do console o vetor Dat[], a media, o desvio padrao ");
41-     printf("e a variancia da sequencia numerica armazenada em Dat[].\n\n");
42-
43-     /* Le do teclado os valores a serem armazenados em Dat[]: */
44-     for(i=0; i<NUM_ELEM; i++){
45-         printf("Digite o valor de Dat[%u]:", i);
46-         scanf("%f", &Dat[i]);
47-     }
48-
49-     /* imprime na tela do console os valores armazenados em Dat[]: */
50-     printf("\nOs valores armazenados em Dat[] sao:\n");
51-     for(i=0; i<NUM_ELEM; i++){
52-         printf("Dat[%u]=%f\n", i, Dat[i]);
53-     }
```

## Vetores, strings e matrizes

```
54
55  /* imprime na tela do console a media dos valores armazenados em Dat[]: */
56  printf("\nA media dos valores em Dat[] eh %f\n",Avg=Mean(Dat,NUM_ELEM));
57  /* Nota 1: O argumento Dat equivale a &Dat[0]. Em C, o nome de uma variavel
58  que representa uma sequencia de elementos armazenados em um segmento na
59  memoria, seja o segmento representativo de um vetor ou matriz, eh equivalente
60  ao endereco na memoria do 1º elemento armazenado no segmento. Experimente
61  substituir a declaracao acima pela declaracao
62  printf("A media dos valores armazenados em Dat[] eh %f\n",Mean(&Dat[0],NUM_ELEM));
63  e comprove a equivalencia entre Dat e &Dat[0]. */
64
65  /* imprime na tela do console o desvio padrao dos valores armazenados em Dat[]: */
66  printf("O desvio padrao dos valores em Dat[] eh %f\n",StdDev=SDev(Dat,Avg,NUM_ELEM));
67  /* Ver Nota 1 acima */
68
69  /* imprime na tela do console a variancia dos valores armazenados em Dat[]: */
70  printf("A variancia dos valores em Dat[] eh %f\n",SQR(StdDev));
71  }
```

```

72
73 /*******/
74 * FUNC: float Mean (float *Vet, unsigned NumElem)
75 *
76 * DESC: Retorna a media dos NumElems valores armazenados no vetor Vet
77 ******/
78 float Mean (float *Vet, unsigned NumElem){
79 /* Nota 2: Observe que a variavel Vet eh declarada com um '*' precedendo o nome
80 da variavel, o que define Vet como um ponteiro (= apontador) para um endereco
81 na memoria. O fato de Vet ser um ponteiro significa que a variavel Vet armazena
82 o endereco na memoria do 1º elemento de uma sequencia de elementos armazenados
83 em um segmento na memoria. Ver Nota 1 acima. */
84
85 unsigned register i; /* register: forza a variavel a ser armazenada
86 nos registradores da CPU, reduzindo o tempo de
87 escrita e leitura */
88 float Acum=0.0; // inicializa acumulador Acum com o valor 0.0
89
90 for(i=0;i<NumElem;i++) Acum+=Vet[i]; /* efetua a soma dos valores dos elementos
91 armazenados em Vet[] */
92
93 Acum/=NumElem; // divide a soma em Acum pelo numero de elementos em Vet[]
94
95 return Acum; // retorna o valor da media
96 }

```

## Vetores, strings e matrizes

```
97
98- /******
99  * FUNC: float SDev (float *Vet, float Mean, unsigned NumElem)
100  *
101  * DESC: Retorna o desvio padrao dos NumElems valores armazenados no vetor Vet
102  *****/
103- float SDev(float *Vet, float Mean, unsigned NumElem){
104- /* Nota 3: Observe que a variavel Vet eh declarada com um '*' precedendo o nome
105  da variavel, o que define Vet como um ponteiro (= apontador) para um endereco
106  na memoria. O fato de Vet ser um ponteiro significa que a variavel Vet armazena
107  o endereco na memoria do 1º elemento de uma sequencia de elementos armazenados
108  em um segmento na memoria. Ver Nota 1 acima. */
109
110- unsigned register i; /* register: forca a variavel a ser armazenada
111                        nos registradores da CPU, reduzindo o tempo de
112                        escrita e leitura */
113  float Acum=0.0; // inicializa acumulador Acum com o valor 0.0
114
115- for(i=0;i<NumElem;i++) Acum+=SQR(Vet[i]-Mean); /* efetua a soma dos quadrados da diferenca
116                        entre cada valor armazenado em Vet[] e
117                        o valor da media Mean */
118
119- Acum/=NumElem; /* divide a soma em Acum pelo numero de elementos em Vet[],o que
120                        determina a variancia dos valores armazenados em Vet[] */
121
122
123  return sqrt(Acum); // retorna o valor do desvio padrao (=raiz da variancia)
124 }
```

### Execução na tela do console:

Este programa lê 5 valores em ponto flutuante do teclado, armazena cada *i*-ésimo valor digitado em `Dat[i]`, e daí imprime na tela do console o vetor `Dat[]`, a média, o desvio padrão e a variância da sequência numérica armazenada em `Dat[]`.

```
Digite o valor de Dat[0]:2.5
Digite o valor de Dat[1]:3.1416
Digite o valor de Dat[2]:2.718
Digite o valor de Dat[3]:2.999
Digite o valor de Dat[4]:0.637
```

Os valores armazenados em `Dat[]` são:

```
Dat[0]=2.500000
Dat[1]=3.141600
Dat[2]=2.718000
Dat[3]=2.999000
Dat[4]=0.637000
```

```
A média dos valores em Dat[] é 2.399120
O desvio padrão dos valores em Dat[] é 0.908632
A variância dos valores em Dat[] é 0.825612
```



Absolutamente importante notar que **o C não faz verificação dos limites da memória que está sendo acessada , tanto na leitura de um valor armazenado em um endereço de memória como na escrita de um valor em um endereço de memória**. O C assume que o programador saiba o que está fazendo com o endereços de memória, i.e., que o programador coerentemente controle os limites dos endereços de leitura ou escrita na memória de modo a não causar mau funcionamento à execução do programa ou ao sistema operacional. Por exemplo, o código fonte abaixo mostra a situação em que é alocado 160 bits (=20 bytes) na memória para um vetor de 10 números inteiros, mas o programa escreve em endereços de memória muito além dos endereços correspondentes aos 20 bytes do segmento de memória que foi reservado para o vetor. Isto significa que o programa estará escrevendo em sabe-se lá qual endereço de memória... Em sistema operacional Windows provavelmente apenas acontecerá que a execução do programa vai travar, porque o Windows limita o acesso à endereços de memória fora do *shell* da tela do console. Mas em sistemas operacionais como o Linux, em que este tipo de “censura” é mínima, o resultado pode ser catastrófico dado que a execução de um fonte C com erro de acesso à memória pode sobrescrever áreas de memória reservadas para o sistema operacional ou reservadas para outros programas ...

```
/* Programa incorreto. Nao execute!!!*/  
main()  
{  
    int vetor [10], i;  
    for (i=0; i<100; i++) vetor [i] =i; /* excede o limite do segmento de memoria reservado para vetor[10] */  
}
```



## Vetores, *strings* e matrizes

Um vetor dedicado ao armazenamento de caracteres de texto é o vetor do tipo *string*, conforme já vimos na introdução dos slides 48 a 51. Uma *string* é um vetor cujos elementos correspondem a uma sequência de caracteres de texto (char), sendo o último elemento do vetor o caractere ASCII '\0' (NULL). O terminador NULL serve para marcar o final do vetor que armazena a *string*.

Seguem exemplos de uso das principais funções básicas da biblioteca padrão do C, específicas para manipulação de *strings* (ver seção "2.13.2 String Functions" na página 112 e seção "2.14 string.h" na página 125 de <https://www.fccdecastro.com.br/CursoC&C++/CLibraryReferenceGuide-Huss.pdf>).

Por exemplo, para ler uma string do teclado podemos usar a função **scanf()**, conforme vimos no exemplo do slide 51. Mas a maneira mais fácil de inserir uma *string* via teclado é usar a função **gets()**, conforme já vimos no slide 50. A forma geral da função **gets()**, definida em string.h, é:

```
gets(nome_da_string);
```

```
/* le uma string do teclado usando gets() e imprime na tela do console */
#include <stdio.h>
void main(void)
{
    char str [80];
    printf("Digite uma string:");
    gets(str); /*le a string do teclado*/
    printf("Voce digitou a string '%s'", str);
}
```

Execução na tela do console:

```
Digite uma string:Alfa Beta Gama Delta
Voce digitou a string 'Alfa Beta Gama Delta'
```

Uma outra função bastante útil é a função **strcpy()**, que é usada para copiar a sequência de caracteres armazenados em uma *string* de origem e armazenar a referida sequência em uma *string* de destino. A forma geral da função **strcpy()**, definida em `string.h`, é:

```
strcpy(string_destino, string_origem);
```

```
/* exemplo de uso de strcpy() */  
#include <stdio.h>  
#include <string.h>  
void main(void)  
{  
    char str [80];  
    strcpy(str, "Hello!");  
    printf("%s", str);  
}
```

Execução na tela do console:

Hello!

A função **strcat()** é usada para concatenar uma *string* anterior com uma *string* posterior, sendo o resultado da concatenação das duas *strings* armazenado na *string* anterior. Portanto, o vetor que armazena a *string* anterior deve ter um tamanho suficiente para acomodar o resultado da concatenação. A forma geral da função **strcat()**, definida em `string.h`, é:

**strcat(string\_anterior, string\_posterior);**

```
/* exemplo de uso de strcat() */
#include <stdio.h>
#include <string.h>
void main(void)
{
    char s1[80], s2[80];
    strcpy(s1, "Alfa Beta ");
    strcpy(s2, "Gama Delta");
    strcat(s1, s2);
    printf("%s", s1);
}
```

Execução na tela do console:

Alfa Beta Gama Delta

## Vetores, *strings* e matrizes

A função **strcmp()** compara duas *strings* s1 e s2 e retorna 0 se elas forem iguais. Se s1 é lexicograficamente maior que s2 (por exemplo: "BBBB">"AAAA" e "AAA">"X") então **strcmp()** retorna o valor +1. Se s1 é lexicograficamente menor que s2 então **strcmp()** retorna o valor -1. A forma geral da função **strcmp()**, definida em string.h, é:

```
strcat(string_s1, string_s2);
```

```
/* Verifica se a string digitada corresponde a senha 'PQRS' */
#include<stdio.h>
#include<string.h>
void main(void)
{
    char str[80];
    char Senha[]="PQRS";
    int retorno;
    printf("Informe a senha:");
    gets(str);
    if(retorno=strcmp(str, Senha)){ /*as strings comparadas sao diferentes*/
        printf("Senha invalida!\n");
    }else{ /* as strings comparadas sao iguais */
        printf("Senha correta!\n");
    }
    printf("strcmp() retornou %d.\n",retorno);
}
```

Execução na tela do console:

```
C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>strcmp
Informe a senha:pqrs
Senha invalida!
strcmp() retornou 1.

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>strcmp
Informe a senha:PQRS
Senha correta!
strcmp() retornou 0.

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>strcmp
Informe a senha:PQR
Senha invalida!
strcmp() retornou -1.

C:\Users\fccde\DATA\UFSM\UFSM 2024_I\Linguagem C para processamento de sinais\C\ExplosNoPost>strcmp
Informe a senha:PQRST
Senha invalida!
strcmp() retornou 1.
```

```
/* le strings ateh que se digite 'sair' */  
#include <stdio.h>  
#include <string.h>  
void main(void)  
{  
    char s[80];  
    for (;;) {  
        printf("Informe uma string:");  
        gets(s);  
        if(!strcmp("sair",s)) break;  
    }  
}
```

Execução na tela do console:

Informe uma string:alfa beta  
Informe uma string:gama delta  
Informe uma string:sair

A função **strlen(arg)** retorna o número de caracteres armazenado na *string*–argumento **arg**. O terminador nulo ‘\0’ não é incluído na contagem de caracteres.

```
/* imprime o tamanho da string digitada */  
#include<stdio.h>  
#include<string.h>  
void main(void)  
{  
    char str [80];  
    printf("Digite uma string:");  
    gets(str);  
    printf("A string armazena %d caracteres.\n", strlen(str));  
}
```

Execução na tela do console:

```
Digite uma string:Abracadabra  
A string armazena 11 caracteres.
```

```
/* Le string do teclado e imprime na ordem inversa.*/  
#include <stdio.h>  
#include <string.h>  
void main(void)  
{  
    char str [80];  
    int i;  
    printf("Digite uma string:");  
    gets(str);  
    for(i=strlen(str)-1; i>=0; i--) printf("%c", str[i]);  
}
```

Execução na tela do console:

Digite uma string:Abracadabra  
arbadacarbA



```
/* informa tamanho e igualdade entre duas strings e concatena */  
#include<stdio.h>  
#include<string.h>  
void main(void)  
{  
    char s1 [80], s2[80];  
    printf("Digite duas strings s1 e s2:\n");  
    gets(s1); gets(s2);  
    printf("Tamanhos: %d %d\n", strlen(s1), strlen(s2));  
    if(!strcmp(s1,s2))printf("As strings sao iguais!\n");  
    strcat(s1,s2);  
    printf("s1 concatenada com s2 eh %s\n", s1);  
}
```

Execução na tela do console:

```
Digite duas strings s1 e s2:  
Abra  
cadabra  
Tamanhos: 4 7  
s1 concatenada com s2 eh Abracadabra
```

```
/* converte uma string para letras maiusculas */  
#include<stdio.h>  
#include<string.h>  
#include<ctype.h>  
void main(void)  
{  
    char str[80];  
    int i;  
    printf("Informe uma string em letras minusculas:");  
    gets(str);  
    for(i=0;str[i];i++) str[i] = toupper(str[i]); /* for i termina quando */  
    printf("%s",str);                          /* str[i] = '\0' */  
    // Nota: A funcao complementar de toupper() eh tolower().  
}
```

Execução na tela do console:

Informe uma string em letras minusculas:abracadabra  
ABRACADABRA

```
/* uso nao usual de printf() */  
#include <stdio.h>  
#include <string.h>  
void main(void)  
{  
    char ByeBye[]="Goodbye world!\n";  
    char str[80];  
    strcpy(str, "Hello world!\n");  
    printf(str); /* funciona porque printf() interpreta str[]  
                como a sua string de controle (string de controle  
                sao os caracteres que ficam entre aspas no argumento  
                da printf() */  
    printf(ByeBye); /* funciona pelo mesmo motivo */  
}
```

Execução na tela do console:

```
Hello world!  
Goodbye world!
```

## Matrizes bidimensionais:

```

/* Armazena os numeros 1 a 12 em uma matriz 2D num[3][4] atraves
da atribuicao num[linha][coluna]=(4*linha)+coluna+1. Dai
imprime os elementos da matriz num[][] na tela do console*/
#include<stdio.h>
void main(void)
{
int num[3][4]; // declara a matriz int num[][] de 3 linhas por 4 colunas
int linha,coluna; // indices das linhas e colunas da matriz num[][]
    for(linha=0; linha<3; linha++){
        for(coluna=0;coluna<4;coluna++){
            num[linha][coluna]=(4*linha)+coluna+1; /* lei de formacao dos elementos da matriz num[][] */
            printf("num[%d][%d]= %d\n",linha,coluna,num[linha][coluna]);
        }
    }
}

```

Execução na tela do console:

```

num[0][0]= 1
num[0][1]= 2
num[0][2]= 3
num[0][3]= 4
num[1][0]= 5
num[1][1]= 6
num[1][2]= 7
num[1][3]= 8
num[2][0]= 9
num[2][1]= 10
num[2][2]= 11
num[2][3]= 12

```

**Nota:** Observe que `int num[3][4]` ocupa  $3 \times 4 \times 16$  bits = 192 bits = 24 bytes na memória. Ver slide 27.

### Matrizes de *strings*:

A instrução abaixo declara uma matriz MStr[][] de 30 *strings*, cada *string* com um tamanho máximo de 80 caracteres:

```
char MStr[30][80];
```

Para acessar uma *string* individual na matriz MStr[][] basta especificar apenas o índice da esquerda. Por exemplo, o comando abaixo vai lendo caracteres do teclado até que a tecla <ENTER> seja pressionada. Os caracteres lidos do teclado são atribuídos à terceira string da matriz MStr:

```
gets(MStr[2]);
```

Conforme já discutimos no slide 49, o nome de uma string é equivalente ao endereço na memória do elemento inicial do vetor que armazena os caracteres da *string*. Portanto a declaração “**gets(MStr[2]);**” é equivalente a :

```
gets(&MStr[2][0]);
```

O próximo slide mostra o código fonte de um programa que utiliza uma matriz de *strings* para armazenar uma “página” de texto digitada no teclado.

**/\* le do teclado ateh NLINHAS linhas de caracteres de texto (strings) e armazena na matriz Texto[NLINHAS][81], sendo cada linha terminada ao se pressionar <ENTER>. Cada linha armazena ateh 80 caracteres. A leitura das linhas eh interrompida pressionando <ENTER> sem nenhuma entrada de texto (linha em branco). Uma vez interrompida a leitura de linhas o programa imprime linha a linha na tela do console todas as linhas de texto lidas do teclado \*/**

```
#include <stdio.h>
void main(void)
#define NLINHAS 100
{
    register int linha, i;
    char Texto[NLINHAS][81];

    /* le linhas de texto do teclado: */
    for(linha=0;linha<NLINHAS;linha++){
        printf("L%d: ",linha);
        gets(Texto[linha]); /* le a linha do teclado */
        if(!Texto[linha][0]) break; /* interrompe se for
            uma linha em branco*/
    }

    /* imprime na tela do console
    todas as linhas de texto lidas do teclado: */
    puts("\nVoce digitou o texto abaixo:");
    for(i=0;i<linha;i++)
        printf("%s\n",Texto[i]); /* i indexa linhas */
    }
```

Execução na tela do console:

```
L0: Quando eu era pequeninho  
L1: minha mae me dava leite  
L2: agora que eu sou grande  
L3: a vaca morreu e eu uso olhos.  
L4:
```

```
Voce digitou o texto abaixo:  
Quando eu era pequeninho  
minha mae me dava leite  
agora que eu sou grande  
a vaca morreu e eu uso olhos.
```

### Matrizes multidimensionais:

O C permite matrizes com mais de duas dimensões. A forma geral de uma matriz N-dimensional é:

**tipo nome[tamanho1][tamanho2]...[tamanhoN];**

Por exemplo, a linha abaixo cria uma matriz 4x10x3 que armazena inteiros:

**int MTrid [4][10][3];**

Matrizes de três ou mais dimensões usualmente requerem uma quantidade grande de memória para armazenamento. O segmento de memória requerido para armazenar os elementos de tais matrizes é alocado permanentemente durante a execução do programa no ponto do programa em que a matriz é declarada.

Por exemplo, uma matriz de 4 dimensões do tipo de dado **char** (8bits = 1 byte), com dimensões 10x6x9x4 requer 10x6x9x4x1byte=2160 bytes.

Se a matriz for do tipo de dado **int** (16 bits = 2 bytes) serão necessários 10x6x9x4x2bytes=4320 bytes de memória.

Se a matriz for do tipo de dado **double** (64 bits = 8 bytes) serão necessários 10x6x9x4x8bytes=17280 bytes de memória.

O tamanho da memória a ser alocada para a matriz aumenta exponencialmente com o número de dimensões. Um programa com matrizes com mais de três ou quatro dimensões pode ficar rapidamente sem memória suficiente para ser executado.



### Inicialização de vetores e matrizes:

A forma geral da inicialização de vetores e matrizes é conforme mostrado abaixo:

**tipo nome[tamanho1]...[tamanhoN]={lista\_de\_valores};**

A `lista_de_valores` é uma lista de constantes separadas por vírgulas que é do tipo compatível com o tipo base do vetor ou matriz. A primeira constante é colocada na primeira posição do vetor/matriz, a segunda constante, na segunda posição e assim por diante. Note que um ponto e vírgula segue a chave `}`. No exemplo seguinte, um vetor de 10 elementos inteiros é inicializado como os números de 1 a 10:

**int dat[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**

Isso significa que `dat[0]` armazena o valor 1, `dat[1]` armazena o valor 2, e assim sucessivamente, até `dat[9]` que armazena o valor 10.

Importante notar que o tamanho na declaração de um vetor (e de uma matriz) não precisa ser necessariamente um valor constante ou definido por um `#define`. O tamanho na declaração de um vetor pode ser especificado através de uma variável inteira que armazena o valor desejado para o tamanho do vetor. Por exemplo, é possível usar uma variável inteira **NumElem** para armazenar um valor de tamanho desejado digitado no teclado e usar o valor da variável **NumElem** para especificar o tamanho de um vetor em ponto flutuante **Dat[]** através da declaração **“float Dat[NumElem];”**. Isto é exemplificado no código fonte mostrado no próximo slide.

No entanto, para vetores e matrizes de centenas de milhares de elementos, como os vetores e matrizes típicos de processamento de sinal em sistemas de comunicação, é muito mais eficiente, flexível e menos sujeito a resultar em bugs de memória quando usamos a técnica de **alocação dinâmica de memória**, técnica que estudaremos no Cap II.3.

```
/* declara, inicializa e opera 2 vetores com tamanho definido
por uma variavel inteira cujo valor eh lido do teclado */
#include <stdio.h>
void main (void){

    unsigned NumElem; // numero de elementos do vetor
    int i;

    printf("Digite o numero de elementos do vetor Dat[:");
    scanf("%u",&NumElem);

    float Dat[NumElem]; // declara vetor Dat[NumElem]
    float Copy[NumElem]; // declara vetor Copy[NumElem]

    /* le do teclado valores que inicializam o vetor Dat[: */
    for(i=0;i<NumElem;i++){
        printf("Digite o valor de Dat[%u]:",i);
        scanf("%f",&Dat[i]);
    }

    for(i=0;i<NumElem;i++)Copy[i]=Dat[i]; // copia Dat[] para Copy[]

    printf("\nA copia do vetor digitado eh:\n");
    for(i=0;i<NumElem;i++)printf("Copy[%u]=%f\n",i,Copy[i]);

}
```

Execução na tela do console:

```
Digite o numero de elementos do vetor Dat[]:4
Digite o valor de Dat[0]:-1.2
Digite o valor de Dat[1]:3.5
Digite o valor de Dat[2]:-8.9
Digite o valor de Dat[3]:10.5

A copia do vetor digitado eh:
Copy[0]=-1.200000
Copy[1]=3.500000
Copy[2]=-8.900000
Copy[3]=10.500000
```

## Vetores, *strings* e matrizes

Vetores de caracteres que armazenam *strings* permitem uma inicialização abreviada com a seguinte forma:

```
char nome_do_vetor[tamanho]="caracteres_da_string";
```

Por exemplo, a linha de código abaixo inicializa a *string* "str" com a palavra "Hello":

```
char str[] = "Hello";
```

Que é equivalente à inicialização:

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Ou, dado que o compilador conta os caracteres que estão dentro de {}, também está valendo esta inicialização :

```
char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Dado que as *strings* terminam com um nulo '\0', é importante termos certeza que o vetor que armazena a *string* tem tamanho suficiente para incluir o terminador nulo '\0'.

No entanto, na inicialização **char str[] = "Hello"** o compilador provê automaticamente espaço para o terminador nulo.

## Vetores, *strings* e matrizes

Matrizes multidimensionais são inicializadas da forma semelhante à inicialização de vetores. Por exemplo, a declaração abaixo inicializa a matriz `Sqr[][]` com os números de 1 a 10 e seus respectivos quadrados:

```
int Sqr[10][2] = {  
    1, 1,  
    2, 4,  
    3, 9,  
    4, 16,  
    5, 25,  
    6, 36,  
    7, 49,  
    8, 64,  
    9, 81,  
    10, 100  
};
```

Note que a declaração acima é idêntica, para todos os efeitos, à declaração:

```
int Sqr[10][2] = { 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, 8, 64, 9, 81, 10, 100 };
```

Isto acontece porque o compilador sempre armazena e “enxerga” os elementos da matriz `Sqr[Nlin][Ncol]`, com `Nlin=10` e `Ncol=2`, na forma de um vetor: Os elementos de `Sqr[Nlin][Ncol]` são sequencialmente armazenados em um vetor `V[Nlin*Ncol]`, cujos elementos são ordenadamente posicionados em endereços contíguos na memória (exatamente da mesma forma sequencial na inicialização acima). Quando no código fonte encontramos uma declaração para acessar o elemento `Sqr[i][j]`, significa que o compilador está “enxergando” o elemento `V[n]`, onde `n= (Ncol*i)+j`.

Por exemplo, o valor de `Sqr[3][1]` é 16. Daí `n= (Ncol*i)+j= (2*3)+1=7`. E , portanto, `V[n]= V[7]=16`, que confere com `Sqr[3][1]`.

Outro exemplo: O valor de `Sqr[4][0]` é 5. Daí `n= (Ncol*i)+j= (2*4)+0=8`. E , portanto, `V[n]= V[8]=5`, que confere com `Sqr[4][0]`.

### Inicialização de matrizes sem especificação de tamanho:

Vimos no slide anterior que o compilador “enxerga” qualquer matriz **Mat[Nlin][Ncol]** na forma de um vetor **V[Nlin\*Ncol]**, onde **Mat[i][j]=V[n]** com **n= (Ncol\*i)+j**. Sendo assim, o compilador calcula automaticamente qualquer uma das dimensões que não é especificada na inicialização de uma matriz.

Por exemplo, consideremos a inicialização da matriz **Sqr[][]** abaixo, em que o número de linhas **Nlin** não é especificado:

```
int Sqr[][2] = {
```

```
1, 1,  
2, 4,  
3, 9,  
4, 16,  
5, 25,  
6, 36,  
7, 49,  
8, 64,  
9, 81,  
10, 100  
};
```

O compilador “enxerga” **int V[20] = { 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, 8, 64, 9, 81, 10, 100 }** armazenado na memória, com os 20 elementos localizados em posições contíguas e sequenciais na memória. O compilador conta o número de elementos armazenados, e, portanto sabe que **Nlin\*Ncol=20**. Como **Ncol=2** é especificado na inicialização, então o compilador determina automaticamente o valor de **Nlin** através de **Nlin=20/Ncol=10**.

A vantagem neste exemplo é que a matriz pode ter seu número de linhas aumentado indefinidamente porque o compilador se encarrega de calcular automaticamente o número de linhas da matriz.

## Vetores, strings e matrizes

O programa a seguir mostra numericamente a maneira que o compilador “enxerga” os elementos de uma matriz **Mat[Ny][Nx]** na forma de um vetor **V[Ny\*Nx]** em que os elementos de **Mat[y][x]** são sequencialmente correspondentes aos elementos do vetor **V[n]** de acordo com a regra  **$n=(Nx*y)+x$** . Veremos a comprovação numérica disto no slide 19 do Cap II.1, quando utilizaremos o conceito de ponteiros para efeito de comprovação.

```
/* Armazena os indices n= 0 a Nx*Ny-1 de V[n] em uma matriz 2D Mat[Ny][Nx] atraves
da atribuicao Mat[y][x]=n, onde n= (Nx*y)+x. Dai atribui ao vetor V[n] os respectivos
valores de Mat[y][x] e imprime na tela do console os elementos de V[n] com referencia
aos indices y e x de Mat[y][x] */
#include<stdio.h>
#define Ny 5
#define Nx 4
void main(void)
{
int Mat[Ny][Nx]; // declara a matriz int Mat[][] de Ny elementos no eixo y e Nx elementos no eixo x
int V[Ny*Nx]; // declara o vetor V[] de Ny*Nx elementos
int y,x; // indices das coordenadas y e z dos elementos da matriz Mat[y][x]
int n; // indice n do vetor v[n];

    for(y=0;y<Ny; y++){
        for(x=0;x<Nx;x++){
            n=(Nx*y)+x; /* indice n do vetor V[n] */
            Mat[y][x]=n; /* Mat[y][x] recebe o indice do vetor V[n] */
            V[n]=Mat[y][x]; /* V[n] recebe o valor de Mat[y][x] */
            printf("Mat[%d][%d]= V[%d]\n",y,x,V[n]); /* imprime v[n] com referencia aos indices y e x de
Mat[y][x] */
        }
    }
}
```

Execução na tela do console:

```
Mat[0][0]= V[0]
Mat[0][1]= V[1]
Mat[0][2]= V[2]
Mat[0][3]= V[3]
Mat[1][0]= V[4]
Mat[1][1]= V[5]
Mat[1][2]= V[6]
Mat[1][3]= V[7]
Mat[2][0]= V[8]
Mat[2][1]= V[9]
Mat[2][2]= V[10]
Mat[2][3]= V[11]
Mat[3][0]= V[12]
Mat[3][1]= V[13]
Mat[3][2]= V[14]
Mat[3][3]= V[15]
Mat[4][0]= V[16]
Mat[4][1]= V[17]
Mat[4][2]= V[18]
Mat[4][3]= V[19]
```



## Vetores, strings e matrizes

No caso de uma matriz 3D, o programa a seguir mostra numericamente a maneira que o compilador “enxerga” os elementos da matriz **Mat[Nz][Ny][Nx]** na forma de um vetor **V[Nz\*Ny\*Nx]** em que os elementos de **Mat[x][y][x]** são sequencialmente correspondentes aos elementos do vetor **V[n]** de acordo com a regra  **$n=(Ny*Nx*z)+(Nx*y)+x$** .

```
/* Armazena os indices n= 0 a Nz*Nx*Ny-1 de V[n] em uma matriz 3D Mat[Nz][Ny][Nx] atraves
da atribuicao Mat[z][y][x]=n, onde n= (Ny*Nx*z)+(Nx*y)+x. Dai atribui ao vetor V[n] os respectivos
valores de Mat[z][y][x] e imprime na tela do console os elementos de V[n] com referencia
aos indices z, y e x de Mat[z][y][x] */
```

```
#include<stdio.h>
```

```
#define Nz 2
```

```
#define Ny 3
```

```
#define Nx 4
```

```
void main(void)
```

```
{
```

```
int Mat[Nz][Ny][Nx]; /* declara a matriz 3D Mat de Nz elementos em z, Ny elementos em y e Nx elementos em x */
```

```
int z,y,x; // indices das coordenadas z, y e x dos elementos da matriz Mat[z][y][x]
```

```
int V[Nz*Ny*Nx]; // declara o vetor V[] de Nz*Ny*Nx elementos
```

```
int n; // indice n do vetor V[n];
```

```
for(z=0;z<Nz; z++){
```

```
for(y=0;y<Ny; y++){
```

```
for(x=0;x<Nx;x++){
```

```
    n=(Ny*Nx*z)+(Nx*y)+x; /* indice n do vetor V[n] */
```

```
    Mat[z][y][x]=n; /* Mat[z][y][x] recebe o indice do vetor V[n] */
```

```
    V[n]=Mat[z][y][x]; /* V[n] recebe o valor de Mat[z][y][x] */
```

```
    printf("Mat[%d][%d][%d]= V[%d]\n",z,y,x,V[n]); /* imprime V[n] c/ referencia aos indices z,y e x de Mat[z][y][x] */
```

```
}
```

```
}
```

```
}
```

```
}
```

Execução na tela do console:

```
Mat[0][0][0]= V[0]
Mat[0][0][1]= V[1]
Mat[0][0][2]= V[2]
Mat[0][0][3]= V[3]
Mat[0][1][0]= V[4]
Mat[0][1][1]= V[5]
Mat[0][1][2]= V[6]
Mat[0][1][3]= V[7]
Mat[0][2][0]= V[8]
Mat[0][2][1]= V[9]
Mat[0][2][2]= V[10]
Mat[0][2][3]= V[11]
Mat[1][0][0]= V[12]
Mat[1][0][1]= V[13]
Mat[1][0][2]= V[14]
Mat[1][0][3]= V[15]
Mat[1][1][0]= V[16]
Mat[1][1][1]= V[17]
Mat[1][1][2]= V[18]
Mat[1][1][3]= V[19]
Mat[1][2][0]= V[20]
Mat[1][2][1]= V[21]
Mat[1][2][2]= V[22]
Mat[1][2][3]= V[23]
```