

Métodos Computacionais em Física I (FIW234)

Turmas IFA e IFB

Noções Básicas de Linguagem C

Edivaldo M. Santos e João R. T. de Mello Neto

Aula 4

Tópicos de linguagem C a serem abordados

- Diretivas de préprocessamento
- Identificadores
- Tipos
- Variáveis e Aritmética
- Operadores e precedência
- Efeitos colaterais
- Laços: while and do, for, switch, break, continue
- controle de fluxo e expressões lógicas: if
- Funções
- Passagem de argumentos e recursividade
- Arrays
- Entrada e Saída
- Apontadores
- Alocação Dinâmica de Memória
- Bibliotecas e Funções Matemáticas

Tópicos de linguagem C a serem abordados

- Diretivas de préprocessamento
- Identificadores
- Tipos
- Variáveis e Aritmética
- Operadores e precedência
- Efeitos colaterais
- Laços: `while` and `do`, `for`, `switch`, `break`, `continue`
- controle de fluxo e expressões lógicas: `if`
- Funções
- Passagem de argumentos e recursividade
- Arrays
- Entrada e Saída
- Apontadores
- Alocação Dinâmica de Memória
- Bibliotecas e Funções Matemáticas

Na aula passada vimos:

- representação numérica em um computador;
- identificadores e palavras-chaves;
- um pouco mais acerca de laços;
- operadores (binários e unários) e precedência.

Na aula passada vimos:

- representação numérica em um computador;
- identificadores e palavras-chaves;
- um pouco mais acerca de laços;
- operadores (binários e unários) e precedência.

Na aula de hoje veremos:

- arrays (variáveis indexadas);
- funções;
- passagem de argumentos;
- manipulação de arquivos.

- Já vimos alguns conceitos relacionados a funções desde a primeira aula. Vimos também como exemplos de funções `printf` e `scanf` que realizam instruções de entrada e saída. Agora, é preciso aprofundar um pouco mais o conceito de função.
- Sabemos já que toda função deve possuir um tipo e retornar um valor compatível com o seu tipo. **Em C, as funções podem retornar qualquer tipo exceto arrays e funções.**
- Além do tipo, as funções possuem argumentos e um corpo, formado por um conjunto de instruções, que serão tratados pelo compilador como uma unidade, e delimitado por chaves `{}`.
- A linguagem também proíbe a declaração de uma função dentro de uma outra função.

- Já vimos alguns conceitos relacionados a funções desde a primeira aula. Vimos também como exemplos de funções `printf` e `scanf` que realizam instruções de entrada e saída. Agora, é preciso aprofundar um pouco mais o conceito de função.
- Sabemos já que toda função deve possuir um tipo e retornar um valor compatível com o seu tipo. **Em C, as funções podem retornar qualquer tipo exceto arrays e funções.**
- Além do tipo, as funções possuem argumentos e um corpo, formado por um conjunto de instruções, que serão tratados pelo compilador como uma unidade, e delimitado por chaves `{}`.
- A linguagem também proíbe a declaração de uma função dentro de uma outra função.
- Para uma função, existe uma diferença entre a **declaração** e a **definição**. Vejamos a diferença...

Funções: Declaração versus definição

- **Declarar uma função** significa somente informar ao compilador qual o tipo da função e dos seus argumentos.
- **Definir uma função** significa fornecer também o seu corpo.

Funções: Declaração versus definição

- **Declarar uma função** significa somente informar ao compilador qual o tipo da função e dos seus argumentos.
- **Definir uma função** significa fornecer também o seu corpo.

```
int main(){  
    double ret;  
    double func();  
    ret = func();  
    return 0;  
}
```

- A segunda instrução dentro do corpo da função main é uma declaração da função func. Ela diz ao compilador o tipo da função (double) e que tal função não recebe qualquer argumento. A partir desse momento, o compilador já sabe que ele deve esperar encontrar em algum ponto do código fonte a declaração de uma função com tais características: nome, tipo e argumentos.

Funções: Declaração versus definição

- **Declarar uma função** significa somente informar ao compilador qual o tipo da função e dos seus argumentos.
- **Definir uma função** significa fornecer também o seu corpo.

```
int main(){  
    double ret;  
    double func();  
    ret = func();  
    return 0;  
}
```

- A segunda instrução dentro do corpo da função main é uma declaração da função func. Ela diz ao compilador o tipo da função (double) e que tal função não recebe qualquer argumento. A partir desse momento, o compilador já sabe que ele deve esperar encontrar em algum ponto do código fonte a declaração de uma função com tais características: nome, tipo e argumentos.
- Já o código abaixo é um exemplo de definição de uma função:

```
double func(){  
    return 1.;  
}
```

Funções: Declaração versus definição

- O código contendo a declaração de uma função também é conhecido como **protótipo**.
- Mas qual a real utilidade de um protótipo?

Funções: Declaração versus definição

- O código contendo a declaração de uma função também é conhecido como **protótipo**.
- Mas qual a real utilidade de um protótipo?
- Há duas funções importantíssimas ligadas aos protótipos:
 - 1 Uma vez que o compilador conhece de antemão os tipos dos argumentos de uma função, tendo-os visto num protótipo, ele é capaz de verificar se usos posteriores da função estão de acordo com a declaração. Incompatibilidades podem ser acusadas já durante a **compilação**.
 - 2 Se uma função prototipada é invocada em algum ponto do código com argumentos diferentes daqueles do protótipo, a presença do protótipo permite a conversão de tipos durante a **execução** do programa, caso tais conversões façam sentido. Por exemplo, uma função declarada no protótipo como recebendo float como argumento pode ser chamada com um argumento int. Nesse caso, a conversão de int para float é feita durante a execução, sem maiores problemas.

Funções: protótipo

```
#include <stdio.h>
/* Calcula o numero de raizes reais de uma equacao de 2o grau */
int main(){

    int nraizes(float, float, float); /* prototipo */
    float a, b, c; /* coeficientes da eq. de 2o grau */
    int n; /* numero de raizes */
    printf("Digite 3 coeficientes de uma eq. do 2o grau (ax2 + bx +c):");
    scanf("%f %f %f",&a, &b, &c);
    n = nraizes(a,b,c);
    printf("A equacao %f x2 + %f x + %f tem %d raiz(es)\n",a,b,c,n);

    return 0;

}

int nraizes(float a, float b, float c){

    float delta;
    int n;

    delta = b*b - 4*a*c;

    if (delta>0)
        n = 2;
    else if (delta==0)
        n = 1;
    else
        n = 0;

    return n;

}
```

Funções: passagem de argumentos

- Há 2 maneiras de se passar argumentos a função em C: **passagem por valor** e **passagem por referência**.

Funções: passagem de argumentos

- Há 2 maneiras de se passar argumentos a função em C: **passagem por valor** e **passagem por referência**.
- Ao receber argumentos passados por valor, a função receptora cria uma cópia das variáveis sendo passadas.
- No caso do programa de cálculo do número de raízes da eq. de 2º grau, ao ser invocada pela função main :

```
n = nraizes(a,b,c);
```

a função nraizes cria dentro do seu corpo 3 novas variáveis, **que não são a, b, c da função main**, pois só existem no escopo de nraizes, e inicializa tais variáveis com os valores de a, b e c passados por main.

- Dessa forma, qualquer modificação feita na variáveis a, b e c de nraizes, não afeta de maneira alguma, as variáveis a, b e c de main. Essas variáveis vivem em escopos diferentes.
- Verifique isso: faça alguma modificação em a, b ou c em nraizes e certifique-se de que tais mudanças não afetam as variáveis de main.

Funções: passagem de argumentos

- E se por outro lado desejássemos que certas modificações ocorridas dentro do corpo de uma função fossem acessíveis na unidade que a invocou? De fato, essa é uma necessidade muito frequente e você se dará conta disso muito claramente ao longo do curso.

Funções: passagem de argumentos

- E se por outro lado desejássemos que certas modificações ocorridas dentro do corpo de uma função fossem acessíveis na unidade que a invocou? De fato, essa é uma necessidade muito frequente e você se dará conta disso muito claramente ao longo do curso.
- A maneira de fazer isso é passando argumentos por referência. Para discutir passagem de argumentos por referência de uma maneira minimamente decente precisamos do conceito de ponteiro (ou apontador). Veremos esse conceito em aulas futuras. Por enquanto, nos limitaremos a apresentar o "como fazer" e o processo ficará mais claro quando discutirmos ponteiros.
- Vale lembrar que as funções `printf` e `scanf` que temos visto há um certo tempo utilizam o segundo tipo de passagem.
- Passamos argumentos por referência a uma função utilizando o operador de referência `&`. Essa operação dá à função receptora o poder de modificar o conteúdo da variável sendo referenciada. A maneira exata como isso é feito só será desmistificada para vocês (**ou não!!!**) com o conceito de ponteiro.

Funções: passagem de argumentos

- E se por outro lado desejássemos que certas modificações ocorridas dentro do corpo de uma função fossem acessíveis na unidade que a invocou? De fato, essa é uma necessidade muito frequente e você se dará conta disso muito claramente ao longo do curso.
- A maneira de fazer isso é passando argumentos por referência. Para discutir passagem de argumentos por referência de uma maneira minimamente decente precisamos do conceito de ponteiro (ou apontador). Veremos esse conceito em aulas futuras. Por enquanto, nos limitaremos a apresentar o "como fazer" e o processo ficará mais claro quando discutirmos ponteiros.
- Vale lembrar que as funções `printf` e `scanf` que temos visto há um certo tempo utilizam o segundo tipo de passagem.
- Passamos argumentos por referência a uma função utilizando o operador de referência `&`. Essa operação dá à função receptora o poder de modificar o conteúdo da variável sendo referenciada. A maneira exata como isso é feito só será desmistificada para vocês (**ou não!!!**) com o conceito de ponteiro.
- As variáveis `a`, `b` e `c` no programa `nraizes.c` foram passadas, num primeiro momento, por referência à função `scanf`:

```
scanf("%f %f %f",&a, &b, &c);
```

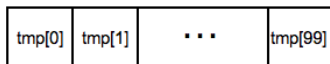
e depois por valor a `nraizes(a,b,c)` como já visto.

Arrays (variáveis indexadas)

Um array é uma coleção de variáveis com propriedades idênticas, em outras palavras, todas do mesmo tipo. Dessa forma, falamos em array de inteiros, array de doubles, array floats, etc. A declaração abaixo:

```
double tempe[100];
```

cria então um objeto, identificado pela palavra tmp, capaz de armazenar 100 números reais, representados em ponto flutuante e precisão dupla, um array de double. Um objeto da coleção é acessado com o auxílio de uma variável inteira (int, short int, long int, unsigned int, etc) que varia de 0 a 99.



Muita atenção para a convenção, o primeiro membro do array é tmp[0] e o último tmp[99]. **Tentar acessar o último elemento de um array de tamanho N tentando utilizar o valor de array[N] é o erro mais comum ao se manipular variáveis indexadas em C!**

Arrays (variáveis indexadas)

- Outro ponto importante é que declarações de arrays não permitem o uso de índices variáveis. Por exemplo, a seguinte declaração de array no interior da função `func` causará um erro de compilação:

```
int func(int index)
{
    double tmp[index];
    ...
}
```

Arrays (variáveis indexadas)

- Outro ponto importante é que declarações de arrays não permitem o uso de índices variáveis. Por exemplo, a seguinte declaração de array no interior da função `func` causará um erro de compilação:

```
int func(int index)
{
    double tmp[index];
    ...
}
```

- Isso ocorre porque o tamanho de um array deve ser definido durante a compilação e nunca durante a fase de execução de um programa.

Arrays (variáveis indexadas)

- Outro ponto importante é que declarações de arrays não permitem o uso de índices variáveis. Por exemplo, a seguinte declaração de array no interior da função `func` causará um erro de compilação:

```
int func(int index)
{
    double tmp[index];
    ...
}
```

- Isso ocorre porque o tamanho de um array deve ser definido durante a compilação e nunca durante a fase de execução de um programa.
- Se você entendeu bem o conceito de pré-processamento e de diretivas do tipo `define`, saberá responder rapidamente se a seguinte forma de declaração de array é correta ou não:

```
#include <stdio.h>
#define N 100
int main()
{
    double tmp[N];
    ...
}
```

Arrays (variáveis indexadas)

- Outro ponto importante é que declarações de arrays não permitem o uso de índices variáveis. Por exemplo, a seguinte declaração de array no interior da função `func` causará um erro de compilação:

```
int func(int index)
{
    double tmp[index];
    ...
}
```

- Isso ocorre porque o tamanho de um array deve ser definido durante a compilação e nunca durante a fase de execução de um programa.
- Se você entendeu bem o conceito de pré-processamento e de diretivas do tipo `define`, saberá responder rapidamente se a seguinte forma de declaração de array é correta ou não:

```
#include <stdio.h>
#define N 100
int main()
{
    double tmp[N];
    ...
}
```



Arrays multidimensionais

- Um array bidimensional de inteiros pode ser declarado assim:

```
int mat[2][3];
```

- Arrays bidimensionais fornecem uma maneira bastante adequada de se representar matrizes em C. Na declaração acima, o array pode ser usado para representar uma matriz de 2 linhas e 3 colunas.

Arrays multidimensionais

- Um array bidimensional de inteiros pode ser declarado assim:

```
int mat[2][3];
```

- Arrays bidimensionais fornecem uma maneira bastante adequada de se representar matrizes em C. Na declaração acima, o array pode ser usado para representar uma matriz de 2 linhas e 3 colunas.
- De fato, o objeto criado na memória não tem o formato de uma matriz, mas é idêntico a um array unidimensional de tamanho 2×3 .

mat[0][0]	mat[0][1]	mat[0][2]	mat[1][0]	mat[1][1]	mat[1][2]
-----------	-----------	-----------	-----------	-----------	-----------

e veja novamente que o primeiro índice do array varia de 0 a 1 e o segundo de 0 a 2.

Arrays multidimensionais

- Um array bidimensional de inteiros pode ser declarado assim:

```
int mat[2][3];
```

- Arrays bidimensionais fornecem uma maneira bastante adequada de se representar matrizes em C. Na declaração acima, o array pode ser usado para representar uma matriz de 2 linhas e 3 colunas.
- De fato, o objeto criado na memória não tem o formato de uma matriz, mas é idêntico a um array unidimensional de tamanho 2×3 .

mat[0][0]	mat[0][1]	mat[0][2]	mat[1][0]	mat[1][1]	mat[1][2]
-----------	-----------	-----------	-----------	-----------	-----------

e veja novamente que o primeiro índice do array varia de 0 a 1 e o segundo de 0 a 2.

- Seguindo esse raciocínio, pode-se declarar arrays de dimensões ainda maiores:

```
int mat3[2][3][3];  
double mat4[5][2][1][3];
```

Arrays

- Vamos ver um exemplo de aplicação de arrays para fixar o conceito.
- Tarefa: Escrever um programa que dado um ano, um mês e um dia, calcula o número de dias do ano transcorridos até aquela data.

Arrays

```
#include <stdio.h>

/* dia_do_ano: determina dia do ano a partir de mes e dia */
int dia_do_ano(int ano, int mes, int dia)
{
    /* tabela com numero de dias para anos normais e anos bissextos */
    int tabela[2][13] = {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
    };

    int i, bis; /* contador e variavel bis: bis = 0 (ano normal), 1 (ano bissexto) */

    bis = ano%4 == 0 && ano%100 != 0 || ano%400 == 0;

    /* Nesse laço for varre-se todos os meses anteriores a mes e soma-se a dia o numero
       de dias de cada um desses meses. Lembre-se que o valor da variavel dia recebida
       do programa principal ja' contem os dias do mes atual. */
    for (i = 1; i < mes; i++)
        dia += tabela[bis][i];

    return dia;
}

int main(){
    int ano, mes, dia, dias;

    printf("Entre com um ano, mes e dia separados por espaco:");
    scanf("%d %d %d",&ano, &mes, &dia);

    dias = dia_do_ano(ano, mes, dia);

    printf("O dia do ano e' %d \n",dias);

    return 0;
}
```

Arquivos de entrada e saída

- Arquivos são um tipo em C. O trecho de código abaixo cria dois arquivos:

```
FILE *entrada, *saida;
```

- Arquivos podem ser utilizados para leitura ou para escrita e precisam ser abertos antes de serem lidos ou escritos.

```
    entrada = fopen("leitura.txt","r");
```

```
    saida    = fopen("escrita.txt","w");
```

- A função `fopen` recebe dois argumentos, o primeiro é o nome do arquivo a ser aberto e o segundo é um caractere que indica a natureza do arquivo a ser aberto, se de leitura ("r" read) ou escrita ("w" write).

Arquivos de entrada e saída

- Arquivos são um tipo em C. O trecho de código abaixo cria dois arquivos:

```
FILE *entrada, *saida;
```

- Arquivos podem ser utilizados para leitura ou para escrita e precisam ser abertos antes de serem lidos ou escritos.

```
entrada = fopen("leitura.txt","r");  
saida = fopen("escrita.txt","w");
```

- A função `fopen` recebe dois argumentos, o primeiro é o nome do arquivo a ser aberto e o segundo é um caractere que indica a natureza do arquivo a ser aberto, se de leitura ("r" read) ou escrita ("w" write).
- Para se escrever ou ler dados de um arquivo, utilizamos as funções `fscanf` `fprintf`. Por exemplo, nesse pedaço de código:

```
fscanf (entrada, "%f", &x);
```

`fscanf` lê o arquivo de entrada e joga o conteúdo lido na variável `x`. Ela é muito semelhante a `scanf`, mas o dispositivo de entrada agora é um arquivo e não o teclado.

- De maneira semelhante:

```
fprintf(saida, " %f \n", x);
```

escreve o valor da variável `x` no arquivo de saída e pula depois disso.

- Para fechar um arquivo, utiliza-se a função `fclose`.

```
fclose(entrada);  
fclose(saida);
```