

Métodos Computacionais em Física I (FIW234)

Turmas IFA e IFB

Ponteiros, Alocação Dinâmica de Memória e Biblioteca
GSL (Gnu Scientific Library)

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

Aula 12

- Primeiramente, é bom dizer que para nós as expressões ponteiro e apontador serão equivalentes. Você encontrará ambas na literatura.

- Primeiramente, é bom dizer que para nós as expressões ponteiro e apontador serão equivalentes. Você encontrará ambas na literatura.
- Razões pelas quais você não ignorar ponteiros:
 - 1 Eles são de longe a característica responsável pelo sucesso do C como linguagem moderna de programação.
 - 2 Eles são usados em praticamente todo lugar em C, afetando a linguagem como um todo. Com exceção dos programas mais simples, representam a característica mais facilmente notável num código em C.
- Já usamos ponteiros desde as primeiras aulas: funções scanf e scanf, tipo FILE, etc.

- Como qualquer outra variável em C, ponteiros devem ser declarados antes de serem usados. Da mesma forma, todo ponteiro deve ter um tipo associado.
- O tipo de um ponteiro não é double, nem float, nem int, etc. Os ponteiros estão associados a uma classe especial de tipos chamada de "ponteiros para", de forma que falamos em ponteiros para inteiros, ponteiros para double, ponteiros para float, etc.
- Exemplo:

```
int *ip;  
float *fp;
```

- As declarações acima criam dois tipos de ponteiro (identificados como tal pelo símbolo *): um ponteiro para inteiro ip e um ponteiro para float fp. Ou seja, criou-se na memória dois objetos, ip e fp, cuja função é apontar para dois outros objetos, um do tipo inteiro e outro do tipo float.

Ponteiros

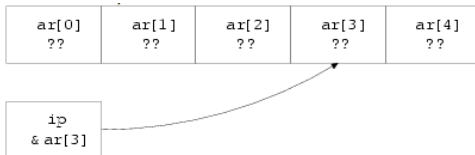
- E assim como qqer outra variável, ponteiros devem também ser inicializados. Abaixo um exemplo de como fazer isso para um ponteiro para inteiros:

- Exemplo:

```
int ar[5],*ip;  
ip = &ar[3];
```

onde criou-se inicialmente um array ar de 5 inteiros na memória e um ponteiro ip para inteiros. A instrução subsequente, inicializa o ponteiro ip por meio do operador & ("endereço de").

- A aplicação do operador unário & a um operando, retorna o apontador associado àquele operando. Dessa forma, você pode pensar num ponteiro como o endereço de uma variável.



- Perceba que como ar não foi inicializado, o valor do objeto apontado por ip ainda está indefinido, sabemos apenas que deve ser do tipo inteiro e daí as interrogações no diagrama acima.

Ponteiros

- Se você tem nas mãos um ponteiro p , como acessar "a coisa" apontado por p ???

Ponteiros

- Se você tem nas mãos um ponteiro p, como acessar "a coisa" apontado por p???
- RESPOSTA: Usando-se o operador unário *;

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x, *p;          /* declaracao de um inteiro e de um
                        /* ponteiro para inteiro */

    p = &x;             /* inicializa o ponteiro */
    *p = 0;             /* inicializa o valor de x em zero */
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);

    *p += 1;            /* incrementa o valor apontado por p */
    printf("x is %d\n", x);

    (*p)++;             /* incrementa o valor apontado por p */
    printf("x is %d\n", x);

    return 0;
}
```

Ponteiros e passagem por referência

- Você deve estar se perguntando qual a vantagem de se usar ponteiros?

Ponteiros e passagem por referência

- Você deve estar se perguntando qual a vantagem de se usar ponteiros?
- A primeira delas é que ponteiros nos fornecem uma alternativa à passagem de argumentos por valor para funções. São comuns as situações em que uma função deve retornar ao ponto em que foi chamada mais do que um valor. Já vimos situações desse tipo em classe.
- Nesses casos, podemos passar como argumentos da função, os ponteiros das variáveis cujos valores desejamos que sejam retornados pela função. No interior da função, esses valores são alterados por meio dos seus apontadores utilizando-se o operador `*`.
- Imagine uma função que deve retornar um valor de mês e um dia desse mês. Na próxima transparência há um esqueleto de código indicando em linhas gerais como isso poderia ser feito com ponteiros.

Eliminação de Gauss-Jordan

```
#include <stdio.h>
#include <stdlib.h>
void date(int *, int *);    /* prototipo da funcao */

main(){
    int month, day;
    date (&day, &month);
    printf("day is %d, month is %d\n", day, month);
    return 0;
}

void date(int *day_p, int *month_p){
    int day_ret, month_ret;
    /* Nesse ponto, calcule o dia e o mes e coloque-os
       em day_ret e month_ret, respectivamente. */
    *day_p = day_ret;
    *month_p = month_ret;
}
```

- Note como no protótipo da função exige-se que dois os argumentos sejam ambos do tipo ponteiro para inteiro. Assim, na chamada da função em main(), os operadores unários devem ser usados para obter os ponteiros correspondentes das variáveis inteiras month e day.

Aritmética de ponteiros e arrays

- A segunda grande vantagem do uso de ponteiros é na manipulação de arrays.

Aritmética de ponteiros e arrays

- A segunda grande vantagem do uso de ponteiros é na manipulação de arrays.
- Os elementos de um array são como qqer outra variável e portanto possuem endereços.

```
int ar[20], *ip;  
ip = &ar[5];  
*ip = 0;          /* equivalente a ar[5] = 0; */
```

Aritmética de ponteiros e arrays

- A segunda grande vantagem do uso de ponteiros é na manipulação de arrays.
- Os elementos de um array são como qualquer outra variável e portanto possuem endereços.

```
int ar[20], *ip;  
ip = &ar[5];  
*ip = 0;          /* equivalente a ar[5] = 0; */
```

- O ato de adicionar 1 a um ponteiro produz outro ponteiro do mesmo tipo. Assim, adicionar n a um ponteiro que aponta para um determinado elemento de um array produz um ponteiro que aponta para um elemento do mesmo array, mas deslocado de n posições.
- Dessa forma, um laço sobre elementos de um array da forma:

```
int ar[20], i;  
for(i = 0; i < 20; i++)  
    ar[i] = 0;
```

pode ser substituído por

```
int ar[20], *ip;  
for(ip = &ar[0]; ip < &ar[20]; ip++)  
    *ip = 0;
```

Aritmética de ponteiros e arrays

- Mas qual a vantagem do segundo laço em relação ao primeiro???

Aritmética de ponteiros e arrays

- Mas qual a vantagem do segundo laço em relação ao primeiro???
- RESPOSTA: velocidade!!!

Aritmética de ponteiros e arrays

- Mas qual a vantagem do segundo laço em relação ao primeiro???
- RESPOSTA: velocidade!!!
- O acesso ao elemento de um array via índices, como no primeiro laço, envolve necessariamente um cálculo de endereço antes do acesso em si. Esse cálculo, por sua vez, é formado, na maioria das implementações, de uma multiplicação e de uma adição.
- O acesso ao elemento de um array via ponteiros não envolve qualquer aritmética. Isso se deve ao fato de que enquanto os elementos de um array não são armazenados de forma sequencial na memória (pedaços do array podem estar em partes desconexas de memória), ponteiros para um array estão necessariamente armazenados em posição sequencial.
- E eficiência pode ser uma questão importante. Em códigos contendo um grande número de laços com operações sobre arrays, cada milissegundo economizado pode ser valioso.
- De fato, pode-se dizer que a linguagem C não sabe o que é indexação, pois já durante o pré-processamento, toda expressão do tipo $x[n]$ é convertida em $*(x+n)$ e o nome do array (x) é convertido para um apontador para o primeiro elemento do array.
- Nessa situação, diz-se que o código que chega ao compilador está **ponterizado**.

Ponteiro void (vazio)

- Ponteiros de tipos diferentes não podem ser misturados numa mesma expressão.
- ponteiros de tipos diferentes não são iguais. Além disso, não há qualquer conversão implícita de um tipo a outro: **ponteiro de inteiro** $\neq$ **ponteiro de double**, etc
- às vezes, em situações bem específicas, é desejável contornar essa limitação. A solução é lançar mão de um tipo especial de ponteiro: o do tipo void (vazio).
- um ponteiro vazio, por construção, pode apontar para qualquer tipo, podendo ser inicializado com ponteiros de qualquer tipo da mesma forma que pode ser usado para inicializar qualquer outro ponteiro.

```
void *vp;
```

Ponteiros e alocação dinâmica de memória

- Vimos no curso que uma maneira de se alocar memória para abrigar um certo tipo de variável é utilizando-se arrays.
- Entretanto, vimos também que o tamanho de tais estruturas deve estar definido no momento da compilação e nunca durante a execução do código.

Ponteiros e alocação dinâmica de memória

- Vimos no curso que uma maneira de se alocar memória para abrigar um certo tipo de variável é utilizando-se arrays.
- Entretanto, vimos também que o tamanho de tais estruturas deve estar definido no momento da compilação e nunca durante a execução do código.
- Muitas vezes, no problema a ser resolvido, a quantidade elementos a serem armazenados não pode ser definida *a priori* e gostaríamos de ter uma estrutura cujo tamanho pudesse ser definido durante a execução. Pense por exemplo na simulação de um processo de produção de partículas que depende da energia inicial disponível. Para grandes valores da energia inicial, muitas partículas são produzidas.
- Em C, a função malloc pode ser usada para esse fim. Uma chamada dessa função, faz com que um pedaço de memória, cujo tamanho deve ser passado como argumento, seja alocado na memória, retornando um apontador para o início do trecho de memória.
- Como esse pedaço de memória pode ser usado para armazenar, em princípio, qualquer tipo válido em C, malloc é do tipo ponteiro void. Assim, é tarefa do usuário fazer com que esse ponteiro vazio aponte para um tipo apropriado.

Ponteiros e alocação dinâmica de memória

- Por exemplo:

```
#include <stdlib.h>      /* contem a declaracao de malloc() */
float *fp;

fp = (float *)malloc(sizeof(float));
```

- Veja que o argumento de malloc é o valor de retorno da função sizeof, nesse caso, o tamanho de uma variável do tipo float. O ponteiro fp acima, termina então apontando para o início de um pedaço de um pedaço de memória, cujo tamanho é aquele necessário para se armazenar um float.
- Se o usuário desejasse armazenar N float, onde N só será definido durante a execução:

```
#include <stdlib.h>      /* contem a declaracao de malloc() */
float *fp;
int N;
...
/* calculo de N nesse trecho de codigo */
...
fp = (float *)malloc(N*sizeof(float));
```

Ponteiros e funções

- Uma função pode retornar um ponteiro:

```
/* funcao retornando ponteiro para inteiro */  
int *func(int a, float b);
```

- Mas a também um tipo especial de ponteiro que é o ponteiro para função:

```
/* ponteiro para uma funcao que retorna inteiro */  
int (*func)(int a, float b);
```

em que a chamada numa parte qualquer do código pode ser:

```
...  
(*func)(1,2);  
...
```

Ponteiros e funções

- Um exemplo simples:

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

void func(int);

int main(){
    void (*fp)(int);

    fp = func;

    (*fp)(1);

    return 0;
}

void func(int arg){
    printf("%d\n", arg);
}
```

- Arrays permitem que objetos idênticos sejam agrupados sob um mesmo nome. Isso é apropriado para um número de tarefas, mas não muito flexível, pois a maioria dos objetos são coisa complicadas com uma estrutura que não se ajusta bem ao estilo dos arrays.
- A linguagem C permite a criação de objetos cujo conteúdo pode apresentar entidades de tipos diferentes. Por exemplo, a declaração (fora do escopo de qualquer função):

```
struct new_type{  
    double a;  
    int i, j, k;  
};
```

cria efetivamente um novo tipo de variável, que pode então ser utilizado como qualquer outro tipo do C.

Estruturas

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

struct new_type{
    double a;
    int i, j, k;
};

int main()
{
    struct new_type meu_tipo;

    meu_tipo.a = 2.0;
    meu_tipo.i = 1;
    meu_tipo.j = (meu_tipo.i)+1;
    meu_tipo.k = (meu_tipo.i)-1;

    printf("a=%lf i=%d j=%d k=%d\n",meu_tipo.a, meu_tipo.i,meu_tipo.j,meu_tipo.k);

    return 0;
}
```

A biblioteca científica do GNU (GNU Scientific Libray) é uma coleção de rotinas para cálculo numérico escrita em C e apresenta uma interface moderna para programadores em C, permitindo que linguagens de alto nível possam fazer uso de suas funções. O código fonte é aberto e distribuído sob a licença GNU General Public License.

- Algumas rotinas disponíveis na GSL

Numeros Complexos
Funcoes Especiais
Permutacoes
Ordenamento
Algebra Linear
Transformadas de Fourier Rapidas
Numeros Aleatorios
Distribuicoes Aleatorias
Histogramas
Integracao por Monte Carlo
Equacoes Diferenciais
Derivacao Numerica
Aceleracao de Series
Zeros de Funcoes
Ajuste de Minimos Quadrados

Raizes de Polinomios
Vetores e Matrices
Combinatoria
Autovalores e Autovetores
Quadratura
Sequencias Pseudo-Aleatorias
Estatistica
Annealing Simulado
Interpolacao
Aproximacoes de Chebychev
Transformadas de Hankel Discretas
Interpolacao
Minimizacao de Funcoes
Constantes Fisicas

- Um pequeno programa exemplo utilizando funções de Bessel (exemplo_bessel.c)

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
int main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);
    return 0;
}
```

- Como compilar:

```
gcc -Wall -I/usr/local/include -o exemplo exemplo_bessel.c
```

onde a flag de compilação `-I` foi usada para informar ao pré-processador onde procurar pelos header files com os respectivos protótipos das funções da biblioteca.

A biblioteca GSL: Matrizes

- Como alocar, inicializar e ler elementos de uma matriz usando as funções `gsl_matrix_alloc`, `gsl_matrix_set` and `gsl_matrix_get`:

```
#include <stdio.h>
#include <gsl/gsl_matrix.h>
int main (void)
{
    int i, j;
    gsl_matrix * m = gsl_matrix_alloc (10, 3);
    for (i = 0; i < 10; i++)
        for (j = 0; j < 3; j++)
            gsl_matrix_set (m, i, j, 0.23 + 100*i + j);
    for (i = 0; i < 100; i++) /* OUT OF RANGE ERROR */
        for (j = 0; j < 3; j++)
            printf ("m(%d,%d) = %g\n", i, j,
                gsl_matrix_get (m, i, j));
    gsl_matrix_free (m);
    return 0;
}
```

A biblioteca GSL: Quadratura

O próximo código calcula a integral

$$\int_0^1 \sqrt{x} \ln(x) dx = 4$$

pelo método de quadratura de Gauss. Note que a função tem uma singularidade logarítmica na origem.

A biblioteca GSL: Quadratura

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
double f (double x, void * params) {
    double alpha = *(double *) params;
    double f = log(alpha*x) / sqrt(x);
    return f;
}

int main (void)
{
    gsl_integration_workspace * w = gsl_integration_workspace_alloc (1000);
    double result, error;
    double expected = -4.0;
    double alpha = 1.0;
    gsl_function F;
    F.function = &f;
    F.params = &alpha;
    gsl_integration_qags (&F, 0, 1, 0, 1e-7, 1000, w, &result, &error);
    printf ("result = % .18f\n", result);
    printf ("exact result = % .18f\n", expected);
    printf ("estimated error = % .18f\n", error);
    printf ("actual error = % .18f\n", result - expected);
    printf ("intervals = %d\n", w->size);
    gsl_integration_workspace_free (w);
    return 0;
}
```

A biblioteca GSL: Equações Diferenciais

- O arquivo `ode_gsl.c` contém o código para integração da equação diferencial de segunda ordem do oscilador de Van der Pol

$$x''(t) + \mu x'(t) (x^2(t) - 1) + x(t) = 0$$

, que pode ser colocada na forma no seguinte conjunto de duas equações acopladas de primeira ordem

$$\begin{aligned}x' &= y \\ y' &= -x + \mu y(1 - x^2)\end{aligned}$$