INF 110
Programação I

Prof. Fabio R. Cerqueira
frcerqueira@gmail.com
DPI / UFV

Aula 14:
Ponteiros ou apontadores em C/C++

Ponteiros

- Definição: Um *apontador* ou *ponteiro* é um tipo de dado que representa um endereço de memória.

- Assim, se uma variável for declarada como ponteiro, o conteúdo da mesma será um endereço de memória. Deste modo, a variável fará referência à área cujo endereço ela armazena.

Ponteiros

- Deve-se especificar que tipo de dado um ponteiro irá referenciar.

- Por exemplo, se a área a que uma variável-ponteiro fará referência (área de memória cujo endereço será armazenado no ponteiro) for alocada para um tipo de dado inteiro, então a variável-ponteiro deverá ser declarada como um ponteiro para o tipo inteiro.

Ponteiros

- Eis a sintaxe para declaração de uma variável-ponteiro, ou simplesmente ponteiro, em C/C++:

```
tipo * nome-da-variavel-ponteiro;
```

- O tipo é qualquer um que seja válido, primitivo ou criado pelo programador. O '*' é o que indica que a variável é um ponteiro, ou seja, que seu tipo de dado é um endereço de memória.

- Assim, a linha acima declara uma variável (como outra qualquer, obedecendo inclusive as mesmas restrições de formação de nome) cujo tipo é: ponteiro para o tipo especificado.

- Ou seja, o que a variável armazenará é um endereço de memória. Para ser mais específico, um endereço de uma área que será alocada para o tipo especificado acima.

Ponteiros

- Veja um exemplo. Suponha a seguinte sequência de código em C++:

```
int x = 7;
int *px; //variável px é do tipo ponteiro para int
px = &x; //O & fornece o endereço de x
```

- Suponha que a variável *x* tenha sido alocada no endereço 3 da memória (suponha também 1 *byte* para *int*, para facilitar) e *px* na posição 5 da memória. Assim, após a execução das linhas acima teremos a seguinte situação:

Endereço	Dado
00000000	
00000001	
00000010	x
00000011	0 0 0 0 0 1 1 1
00000100	px
00000101	0 0 0 0 0 0 1 1
00000110	
00000111	
...	

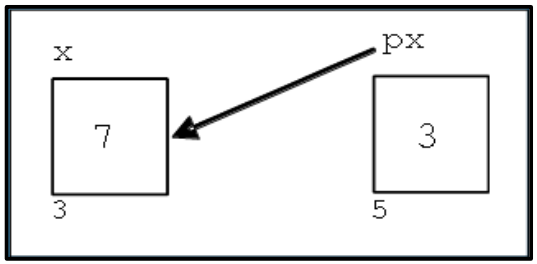
Ponteiros

```
int x = 7;
int *px;
px = &x;
```

- Portanto, o operador '&' é utilizado para obter o endereço da área de memória alocada para uma variável.

- A última linha do trecho acima faz com que px armazene o endereço da área de memória alocada para a variável x. Ou seja, faz com que px referencie x ou, como também é comum se dizer, faz com que px aponte para x.

- Ilustrativamente, a memória poderia ser representada da seguinte forma, após a execução do trecho acima:



Ponteiros

- Existe ainda o operador '*' (diferente do utilizado na declaração) que serve para acessar a área de memória apontada pelo ponteiro. Por exemplo, a segunda atribuição abaixo:

```
px = &x
*px = 6;
```

coloca o valor 6 na variável apontada por px (coloca o valor 6 na área de memória cujo endereço está armazenado em px), ou seja, em x. Isto quer dizer que fazer x=6 ao invés de *px=6 surtiria o mesmo efeito.

- Portanto, com o ponteiro acima, pode-se escrever um valor na área alocada para x de duas formas. Primeiro, utilizando o rótulo x, como estamos acostumados. Segundo, via ponteiro px.

Ponteiros

- Cuidado, portanto, com o significado do símbolo '*'. Como ele é utilizado de formas distintas em diferentes contextos, o programador deve saber identificar cada uma destas situações (quando é declaração, quando é o acesso da área apontada, quando é multiplicação, quando é comentário de código, etc).

- Então, não se confunda no uso do símbolo '*':

```
float *px, x, y; // aqui o * soh serve
   // para indicar que px eh um ponteiro
x = 20;
px = &x;
y = *px + 10; // aqui o * serve para
// acessar a área apontada por px.
```

Ponteiros

- O mesmo cuidado serve para o símbolo '&' que também pode ser utilizado de várias formas na linguagem, com significados diferentes. No contexto de ponteiros, quando este símbolo aparece logo antes de um nome de variável, como vimos, deve ser entendido como obtenção do endereço da variável.

Exercício:

Ilustre o estado da memória para as variáveis do trecho de código do *slide* anterior após sua execução.

Ponteiros

- Qual o valor inicial de um apontador?
  - Observe o seguinte programa:

```
int main() {
  int x, *px;

  *px = 10;
  x = *px;
  return 0;
}
```

  - O que há de errado neste programa?

Ponteiros

- Analise os seguintes trechos de código em C++. Verifique se estão corretos ou incorretos. Caso estejam incorretos, apontem os prováveis erros. Para tanto, considere a declaração das variáveis abaixo:

```
int a, b, *c, *d;
```

a)
```
c = &b;
d = c;
*d = 10;
```

b)
```
c = *a;
d = 10 + *c;
```

c)
```
d = c;
c = &a;
a = 10;
b = *d;
```

d)
```
c = &a;
cin >> c;
d = c;
b = *d;
```

Ponteiros

```
int a, b, *c, *d;
```

e)
```
b = 10;
c = &b;
cout << c;
```

f)
```
b = 2;
c = &b;
*c += 2;
```

g)
```
b = 1;
c = *b;
*c = a + *c;
```

h)
```
a = 1;
c = &a;
for(b=0;b<10;b++)
  *c += 1;
```

i)
```
a = 100;
c = &a;
while (a>0){
  *c=*c-1;
}
```

Ponteiros e a passagem por referência

- Graças aos ponteiros pode-se de dentro de uma função A acessar as variáveis locais de uma função distinta B.

- Em C++ aprende-se que basta acrescentar o & a um parâmetro de uma função para que a passagem seja por referência. Mas você já parou para pensar como isto realmente funciona?

- Vamos analisar uma função bem simples para entender. Dados dois valores inteiros *a* e *b*, colocar esses valores em ordem crescente, de modo que $a \leq b$.

Ponteiros e a passagem por referência

```
// Assim não funciona
void ordena( int a, int b ) {
   if (a > b) {
     int temp = a; // Aqui estamos alterando
     a = b;        // só uma cópia dos valores
     b = temp;     // passados
   }
}

// Assim funciona
void ordena( int &a, int &b ) {
   if (a > b) {
     int temp = a;
     a = b;    // Aqui vamos alterar as
     b = temp; // variaveis passadas como
   }           // argumento para a funcao
}
```

Ponteiros e a passagem por referência

- E uma possível chamada de função poderia ser assim:

```
int main()
{
   int x, y;
   cin >> x >> y;
   ordena( x, y );
   ...
}
```

Ponteiros e a passagem por referência

- Para entendermos o que de fato ocorre, vamos ver como se faria a mesma coisa em C. Neste caso, o uso de ponteiros é explícito.

```
// Os parâmetros são declarados como ponteiros
void ordena( int *a, int *b ) {
   if (*a > *b) { // Utilizam-se os ponteiros
     int temp = *a; // explicitamente.
     *a = *b;
     *b = temp;
   }
}
int main() {
   ...
   ordena( &x, &y ); // Passam-se explicitamente
   ...          // os endereços. Assim, o ponteiro
// a (parâmetro) apontará para x e o b para y.
}
```

Ponteiro para ponteiro

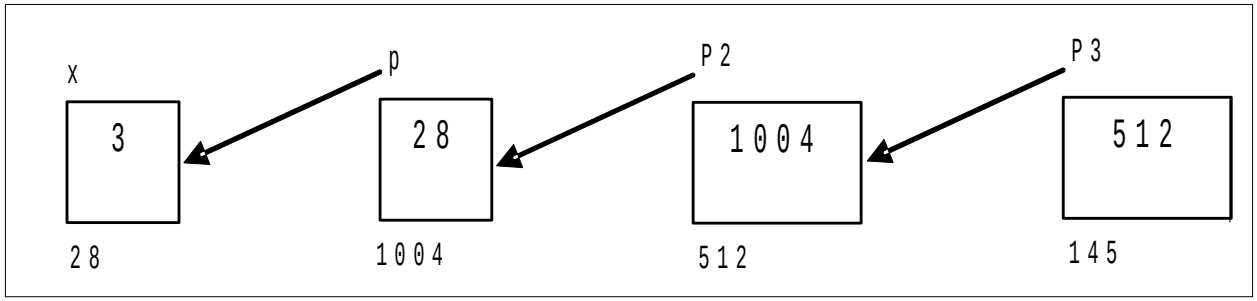
- Podemos ter ponteiros para qualquer tipo válido, incluindo tipos que criamos com *struct*. Também sabemos que ponteiros são variáveis como qualquer outra e que armazenam endereços de memória. Podemos concluir então que é perfeitamente possível se ter um ponteiro que aponte para outro ponteiro...

Ponteiro para ponteiro

- Portanto, o trecho de código abaixo é válido:

```
int x=3, *p, **p2, ***p3;
p = &x;
p2 = &p;
p3 = &p2;
```

e produziria o seguinte efeito (esquemático) na memória (abaixo das células observam-se os endereços hipotéticos destas, em representação decimal para simplificar):



- Desta forma, poder-se-ia acessar o conteúdo de x assim:

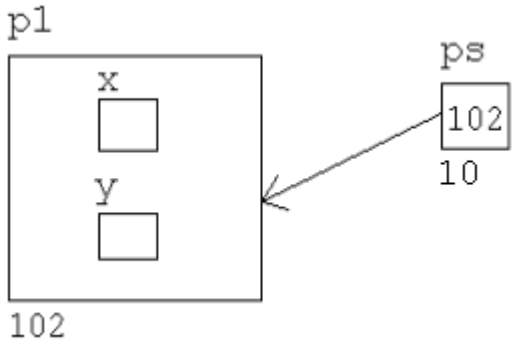
```
cout << ***p3; // Mostre outras 3 formas válidas.
```

Ponteiro para tipo composto

- Quando se cria uma *struct* em C/C++, um novo tipo é estabelecido. Já que é possível empregar ponteiros para qualquer tipo válido, criar ponteiros para um tipo definido através de uma *struct* é perfeitamente possível. Abaixo é dado um exemplo:

```
struct Ponto {
  float x;
  float y;
};

Ponto p1, *ps;
ps = &p1;
```



Ponteiro para tipo composto

- Assim, o trecho de código:

```
p1.x = 5;
p1.y = 8;
```

- Pode ser escrito também como:

```
(*ps).x = 5;
(*ps).y = 8;
```

- Ou utilizando a forma mais compacta:

```
ps->x = 5;
ps->y = 8;
```

- Cuidado para não misturar as formas!

Um arranjo é representado por um ponteiro constante para a primeira posição

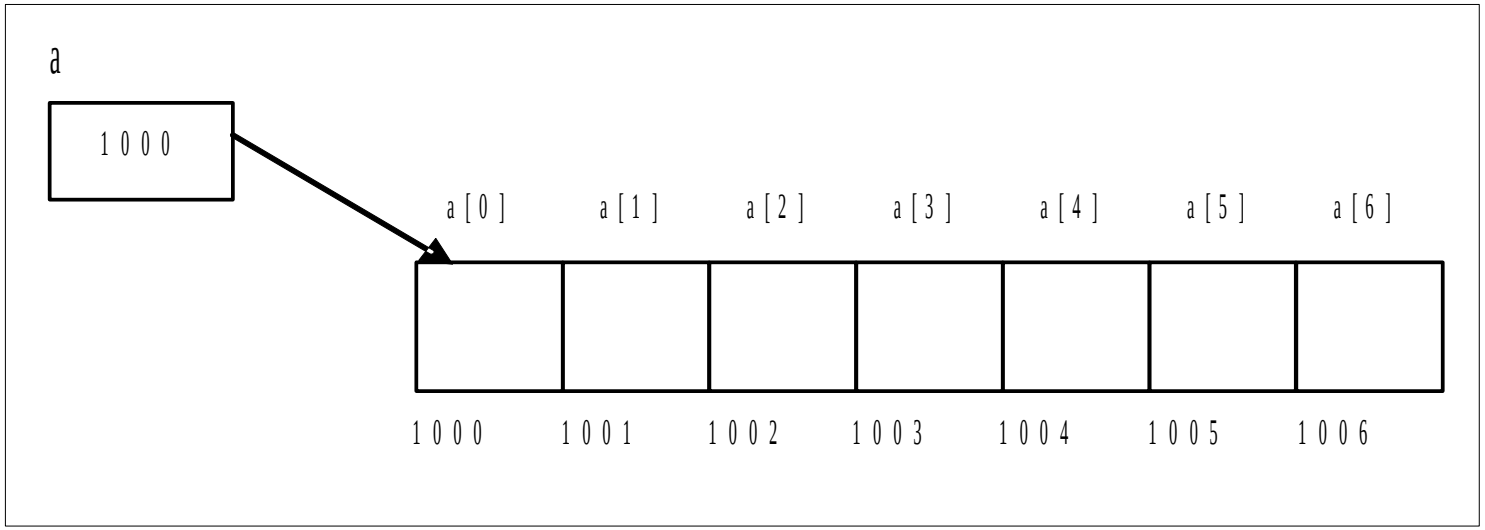
- O nome de um arranjo em C/C++ é um apontador constante (sempre aponta para a mesma área). Quando um arranjo é criado, seu nome é convertido para uma constante que aponta para a primeira posição do arranjo.

- Assim, se temos a declaração: `int a[TAM];`
  - *a* fará referência à primeira posição do arranjo.
  - *a*+*i* fará referência à *i*-ésima posição do arranjo.

Um arranjo é representado por um ponteiro constante para a primeira posição

- Seja a declaração: `int a[7];`

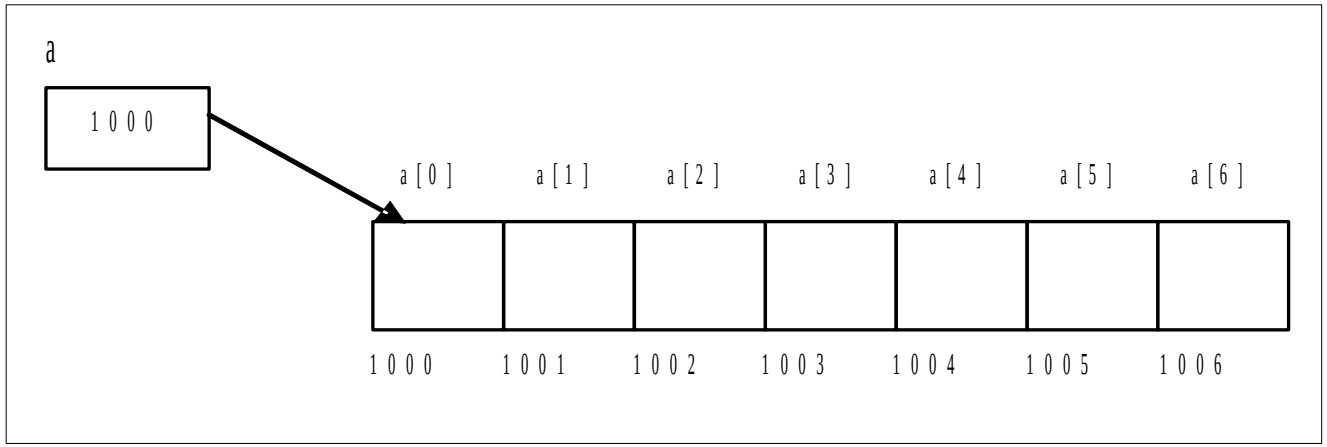
- O que (esquematicamente) se tem na memória é (suponha que os endereços de memória das posições do arranjo vão de 1000 a 1006):



Um arranjo é representado por um ponteiro constante para a primeira posição

- Claro que os endereços mostrados não variam de 1 em 1. Como o tipo de dado é `int` os endereços variam de 4 em 4 (normalmente, já que `int` costuma ser representado por 4 bytes).

- Mas pode-se, para facilitar, pensar na forma simplificada mostrada no desenho, pois se um ponteiro `p` aponta para o primeiro (tem endereço 1000) e fazemos `p++`, o mesmo passa a apontar para o segundo, mesmo dando a impressão de que o incrementamos de somente 1 unidade. Na verdade, ele foi incrementado de 4.



- Portanto, o trecho de código:

```
for(int i=0; i<7 i++)
  a[i]=0;
```

- Pode ser escrito também como:

```
for(int i=0; i<7 i++)
  *(a+i)=0;
```

- Vê-se, então, que é possível utilizar os operadores aritméticos + e - em endereços de memória (Lembrando que a soma (ou subtração) que de fato é realizada é de `i*num_bytes_do_tipo_em_questao`).

Um arranjo é representado por um ponteiro constante para a primeira posição

- Exercício: Seja a *struct* abaixo bem como o arranjo utilizando o tipo criado. Mostre como preencher o arranjo com os valores (2,5),(6, 3) e (15,10) utilizando operações sobre o nome (ponteiro) *a*, ao invés dos índices, como feito tradicionalmente.

```
struct Ponto {
  float x;
  float y;
};

Ponto a[3];
```

Manipulando partes do arranjo

- Vimos então que uma variável arranjo é um ponteiro para o primeiro elemento. Isto significa que podemos analisar parte de um arranjo através de ponteiros.

- Por exemplo, uma solução para o exercício de separar dia, mês e ano de uma *string* representando uma data onde as informações estivessem separadas por '/' seria:

```
int main() {
  char data[11], dia[3], mes[3], ano[5], *p;
  cin.getline(data,11);
  p=data;
  data[2]=data[5]='\0';
  strcpy(dia,p);
  p+=3;
  strcpy(mes,p);
  p+=3;
  strcpy(ano,p);
  data[2]=data[5]='/'; // reestabelecendo a data
  cout << dia << '/' << mes << '/' << ano;
  return 0;
}
```

Manipulando partes do arranjo

- Ou simplesmente como mostrado abaixo:

```
int main(){
  char data[11], dia[3], mes[3], ano[5];

  cin.getline(data,11);
  data[2]=data[5]='\0';
  strcpy(dia,data);
  strcpy(mes,data+3);
  strcpy(ano,data+6);
  data[2]=data[5]='/'; // reestabelecendo a data

  cout << dia << '/' << mes << '/' << ano;
  return 0;
}
```

Manipulando partes do arranjo

- Exercício: considere o algoritmo de seleção, como visto na aula sobre ordenação. Se trocássemos a função `indiceMenor`, como visto na aula, por:

```
// Retorna o indice do menor elemento de a
int indiceMenor(int a[], int n)
{
  int m = 0;
  for ( int i=1; i < n; i++ )
    if (a[i] < a[m]) m = i;
  return m;
}
```

qual(is) adaptação(ões) seria(m) necessária(s) na função `ordenaSeleção` para a mesma continuar funcionando?

Ponteiro nulo

- Por fim, há o conceito de ponteiro nulo. Um ponteiro nulo possui um valor reservado, geralmente zero, indicando que ele não se refere a qualquer objeto. São usados frequentemente para representar condições especiais.

```
//inicializando um ponteiro com NULL.
Ponto *p=NULL;
```

- Outro exemplo: Lembre-se do retorno da função `strchr`. Esta função retorna um ponteiro para a (o endereço de memória da) área onde se detectou a primeira ocorrência do caractere sendo procurado ou então retorna `NULL` se o caractere não está presente na *string* sendo avaliada.