USP-ICMC-BInfo

Ponteiros em C

SCC501 - ICC-II

2011

Prof. João Luís

Ponteiros

- Três propriedades que um programa deve manter quando armazena dados:
  - onde a informação é armazenada;
  - que valor é mantido lá;
  - que tipo de informação é armazenada.
- A definição de uma variável simples obedece a estes três pontos. A declaração provê o tipo e um nome simbólico para o valor. Também faz com que o programa aloque memória para o valor e mantenha o local internamente.

Operador de endereço: `&`

- Segunda estratégia baseada em ponteiros, que são variáveis que armazenam endereços ao invés dos próprios valores.
- Mas antes de discutir ponteiros, vejamos como achar endereços explicitamente para variáveis comuns.
- Aplique o operador de endereço, **`&`**, a uma variável para pegar sua posição; por exemplo, se **`lar`** é uma variável, **`&lar`** é seu endereço.

Operador de endereço: &

```
#include <stdio.h>
void main()
{
    int rosquinhas = 6;
    double xicaras = 4.5;
    printf("valor das rosquinhas = %d", rosquinhas);
    printf(" e endereco das rosquinhas = %d\n",
   &rosquinhas);
    printf("valor das xicaras = %g", xicaras);
    printf(" e endereco das xicaras = %d\n",
   &xicaras);
}
```

- A saída deste programa é:

  valor das rosquinhas = 6 e endereco das rosquinhas = 1245052

  valor das xicaras = 4.5 e endereco das xicaras = 1245044

Operador de dereferenciação: `*`

- O uso de variáveis comuns trata o valor como uma quantidade nomeada e a posição como uma quantidade derivada. A nova estratégia, usando ponteiros, trata a posição como a quantidade nomeada e o valor como uma quantidade derivada. Este tipo especial de variável – o *ponteiro* – armazena o endereço de um valor. Então, o nome do ponteiro representa a posição. Aplicando o operador **`*`**, chamado de operador de valor indireto ou de dereferenciação, fornece o valor da posição. Suponha por exemplo, que **`ordem`** é um ponteiro. Então, **`ordem`** representa um endereço, e **`*ordem`** representa o valor naquele endereço. **`*ordem`** torna-se equivalente a um tipo comum.

Operador de dereferenciação: *

```
#include <stdio.h>
void main()
{
    int atualiza = 6;          // declara uma variável
    int * p_atualiza;          // declara ponteiro para um int
    p_atualiza = &atualiza;    // atribui endereço do int para o
   ponteiro
    // expressa valores de duas formas
    printf("Valores: atualiza = %d", atualiza);
    printf(", *p_atualiza = %d\n", *p_atualiza);
    // expressa endereço de duas formas
    printf("Enderecos: &atualiza = %d", &atualiza);
    printf(", p_atualiza = %d\n", p_atualiza);
    // usa ponteiro para mudar valor
    *p_atualiza = *p_atualiza + 1;
    printf("Agora atualiza = %d\n", atualiza);

}
```

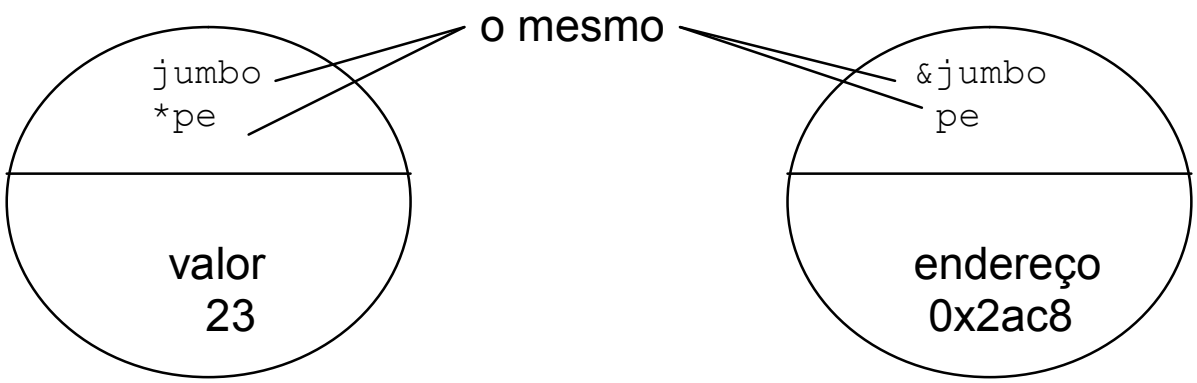
Operador de dereferenciação: *

- A saída deste programa é:

```
Valores: atualiza = 6, *p_atualiza = 6
Enderecos: &atualiza = 1245052, p_atualiza = 1245052
Agora atualiza = 7
```

* e &

Imagine o seguinte exemplo:
int jumbo = 23;
int *pe = &jumbo;



Dois lados da mesma moeda

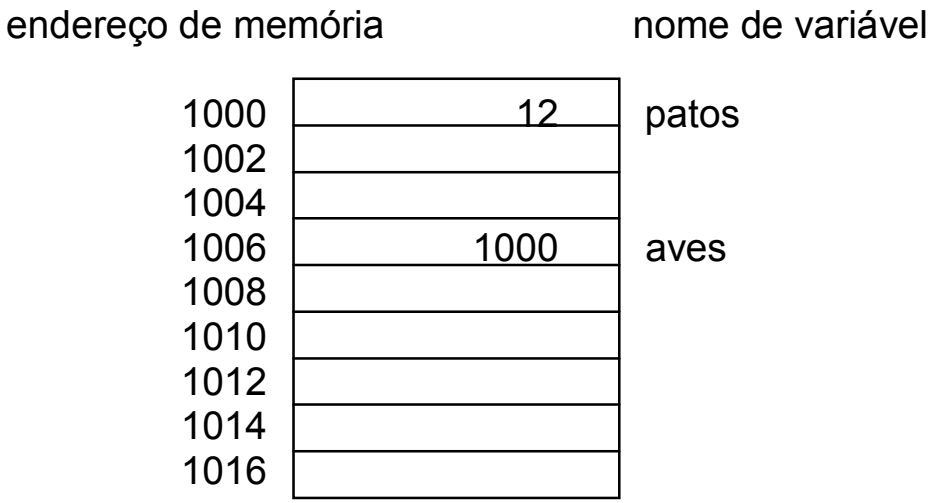
Declarando e iniciando ponteiros

- O computador deve manter o tipo do valor para o qual um ponteiro se refere. Por exemplo, o endereço de um **`char`** parece igual ao endereço de um **`double`**, mas **`char`** e **`double`** usam diferentes números de bytes e diferentes formatos internos para armazenarem valores. Portanto, uma declaração de ponteiro deve especificar que tipo de dado para o qual o ponteiro aponta. Por exemplo, o último exemplo tem esta declaração:

```
int *p_atualiza;
```

- **`p_atualiza`** tem de ser um ponteiro e que aponta para o tipo **`int`**.

Declarando e iniciando ponteiros




```
int patos = 12;
```

cria uma variável `patos`, armazena o valor `12` na variável.

```
int *aves = &patos;
```

cria uma variável `aves`, armazena o endereço de `patos` na variável.

Declarando e iniciando ponteiros

- Usa-se a mesma sintaxe para declarar ponteiros para outros tipos:

```
double *tax_ptr;
char *str;
```

- Uma variável ponteiro nunca é simplesmente um ponteiro. É sempre um ponteiro para um tipo específico. Assim, como os vetores, os ponteiros são derivados de outros tipos. Apesar de **`tax_ptr`** e **`str`** apontarem para tipos de tamanhos diferentes, estas variáveis têm o mesmo tamanho (tamanho de um endereço).

```
int higgens = 5;
int * pi = &higgens;
```

- faz **`pi`** e não **`*pi`** igual a **`&higgens`**.

Declarando e iniciando ponteiros

```
#include <stdio.h>
void main()
{
   int higgens = 5;
   int * pi = &higgens;
   printf("Valor de higgens = %d, Endereco de higgens
   = %d\n", higgens, &higgens);
   printf("Valor de *pi = %d; Valor de pi = %d\n",
   *pi, pi);
}
```

- Saída deste programa:

```
Valor de higgens = 5, Endereco de higgens = 1245052
Valor de *pi = 5; Valor de pi = 1245052
```

Declarando e iniciando ponteiros

- Perigo quando se cria um ponteiro: o computador aloca memória para armazenar um endereço, mas ele não aloca memória para armazenar o dado para o qual o ponteiro aponta. Criar espaço para dados envolve um passo separado. Sem esse passo, pode ocorrer um desastre:

```
long *amigo;
*amigo = 223323;
```

- Onde o valor **223323** é colocado? Não se sabe. Ou seja, SEMPRE inicie um ponteiro a um endereço apropriado e definido antes de aplicar o operador de dereferenciação (*****) nele.

Alocando memória com **`malloc`**

- Usamos nos exemplos anteriores a iniciação de ponteiros a endereços de variáveis; em *tempo de compilação* há a alocação das variáveis e os ponteiros provêem outra forma de acesso a elas que já poderia ter sido feito através dos próprios nomes das variáveis. O verdadeiro valor dos ponteiros acontece quando se aloca memória não utilizada por outras variáveis durante o *tempo de execução*. Neste caso os ponteiros se tornam o único acesso àquela memória. Em C, pode-se alocar memória com a função **`malloc()`**, que tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

- A funçao toma o número de bytes que queremos alocar (**`num`**), aloca na memória e retorna um ponteiro **`void *`** para o primeiro byte alocado. O ponteiro **`void *`** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **`malloc()`** retorna um ponteiro nulo.

Alocando memória com **malloc**

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int * pi;
    double * pd;
    pi=(int *)malloc(sizeof(int));
    *pi = 1001;                 // armazena um valor lá
    printf("int ");
    printf("valor = %d: posicao = %d\n", *pi, pi);

    pd=(double *)malloc(sizeof(double));   // aloca espaço para um double
    *pd = 10000001.0;           // armazena um double lá
    printf("double ");
    printf("valor = %g: posicao = %d\n", *pd, pd);
    printf("tamanho de pi = %d", sizeof pi);
    printf(": tamanho de *pi = %d\n", sizeof *pi);
    printf("tamanho de pd = %d", sizeof pd);
    printf(": tamanho de *pd = %d\n", sizeof *pd);
    free(pi);
    free(pd);
}
```

Alocando memória com **`malloc`**

- A saída deste programa será:

```
int valor = 1001: posicao = 4390960
double valor = 1e+007: posicao = 4398016
tamanho de pi = 4: tamanho de *pi = 4
tamanho de pd = 4: tamanho de *pd = 8
```

Usando `malloc` para criar vetores dinâmicos

- Freqüentemente usa-se **`malloc`** com grandes pacotes de dados, como vetores, strings e estruturas. Aí que **`malloc`** é útil. Se você define um vetor através de sua declaração, este vetor estará ocupando memória durante o tempo todo, o programa usando-o ou não (vetor estático). Mas com **`malloc`** você pode criar um vetor dinâmico, ou seja, um vetor que vai ocupar memória apenas quando e se o programa precisar. Com o vetor estático, você tem que definir o tamanho do vetor na sua declaração. Com o vetor dinâmico, não precisa. Veremos agora como usar o operador **`malloc`** para criar um vetor e como usar um ponteiro para acessar elementos do vetor.

Criando um vetor dinâmico com **`malloc`**

- Basta dizer a **`malloc`** o tipo do elemento do vetor e o número de elementos que se deseja. A sintaxe requer que se siga o nome do tipo com o número de elementos entre colchetes. Por exemplo, se você precisa de um vetor de 10 **`ints`**, faça isto:

```
int * pvet;
pvet=(int *)malloc(10*sizeof(int));
// pega um bloco de 10 ints
```

- **`malloc`** retorna o endereço do primeiro elemento do bloco. Neste exemplo, este valor é atribuído ao ponteiro **`pvet`**. Para liberar esta área após o uso, deve-se usar a função **`free`**:

```
free(pvet);  // libera a área de um vetor
                dinâmico
```

Criando um vetor dinâmico com **`malloc`**

- Num vetor dinâmico, é responsabilidade do programador saber quantos elementos há em um bloco. Isto é, o compilador não mantém o fato que **`pvet`** aponta para o primeiro de 10 inteiros. Na verdade, o programa tem a informação do número de elementos, caso contrário não seria possível liberar a área de memória de um vetor através de **`free`**. Mas esta informação não está disponível publicamente, tal que não se pode usar o operador **`sizeof`** para achar o número de bytes de um vetor alocado dinamicamente.

Criando um vetor dinâmico com **malloc**

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

void main()
{
   double * p3;
   p3=(double *)malloc(3*sizeof(double)); // espaço para 3 doubles
   p3[0] = 0.2;                  // trata p3 como um nome de vetor
   p3[1] = 0.5;
   p3[2] = 0.8;
   printf("p3[1] eh %g.\n", p3[1]);
   p3 = p3 + 1;                  // incrementa o ponteiro
   printf("Agora p3[0] eh %g e ", p3[0]);
   printf("p3[1] eh %g.\n", p3[1]);
   p3 = p3 - 1;                  // aponta de volta ao começo
   free(p3);                  // libera a memória
}
```

- **A saída deste programa é:**

```
p3[1] eh 0.5.
Agora p3[0] eh 0.5 e p3[1] eh 0.8.
```

Criando um vetor dinâmico com **malloc**

```
#include <stdio.h>
void main()
{
   double wages[3] = {10000.0, 20000.0, 30000.0};
   short stacks[3] = {3, 2, 1};
// Aqui estão duas formas de se conseguir o endereço de um vetor
    double * pw = wages;     // nome de um vetor = endereço
    short * ps = &stacks[0]; // ou uso do operador de endereço
                                      // com elemento de vetor
    printf("pw = %d, *pw = %g\n", pw, *pw);
    pw = pw + 1;
    printf("adicione 1 ao ponteiro pw:\n");
    printf("pw = %d, *pw = %g\n\n", pw, *pw);
    printf("ps = %d, *ps = %d\n", ps, *ps);
    ps = ps + 1;
    printf("adicione 1 ao ponteiro ps:\n");
    printf("ps = %d, *ps = %d\n\n", ps, *ps);
    printf("acesse dois elementos com a notacao de vetor\n");
    printf("%d %d\n", stacks[0], stacks[1]);
    printf("acesse dois elementos com a notacao de ponteiro\n");
    printf("%d %d\n", *stacks, *(stacks + 1));
    printf("%d = tamanho do vetor wages\n", sizeof wages);
    printf("%d = tamanho do ponteiro pw\n", sizeof pw);

}
```

Criando um vetor dinâmico com **`malloc`**

- A saída deste programa é:

```
pw = 1245032, *pw = 10000
adicione 1 ao ponteiro pw:
pw = 1245040, *pw = 20000
ps = 1245024, *ps = 3
adicione 1 ao ponteiro ps:
ps = 1245026, *ps = 2
acesse dois elementos com a notacao de vetor
3 2
acesse dois elementos com a notacao de ponteiro
3 2
24 = tamanho do vetor wages
4 = tamanho do ponteiro pw
```

Ponteiros e Strings

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>                          // declara strlen(), strcpy()
void main()
{
    char animal[20] = "urso";                // animal armazena urso
    const char * passaro = "canario";        // passaro armazena endereço da
                                                    // string
    char * ps;                               // não inicializado
    printf("%s e ", animal);                 // mostra urso
    printf("%s\n", passaro);                 // mostra canário
    printf("Entre com um tipo de animal: ");
    scanf("%s", animal);                     // ok se entrada < 20 chars
    ps = animal;                             // faz ps apontar para string
    printf("%ss!\n", ps);                    // ok, mesmo como usar animal
    printf("Antes de usar strcpy():\n");
    printf("%s em %d\n", animal, (int *) animal);
    printf("%s em %d\n", ps, (int *) ps);
    ps=(char *)malloc(strlen(animal)+1);
    strcpy(ps, animal);                      // copia string ao novo local
    printf("Depois de usar strcpy():\n");
    printf("%s em %d\n", animal, (int *) animal);
    printf("%s em %d\n", ps, (int *) ps);
    free(ps);
}
```

Ponteiros e Strings

- Aqui está uma saída possível:

```
urso e canario
Entre com um tipo de animal: raposa
raposas!
Antes de usar strcpy():
raposa em 1245036
raposa em 1245036
Depois de usar strcpy():
raposa em 1245036
raposa em 4398016
```

Usando **`malloc`** para criar estruturas dinâmicas

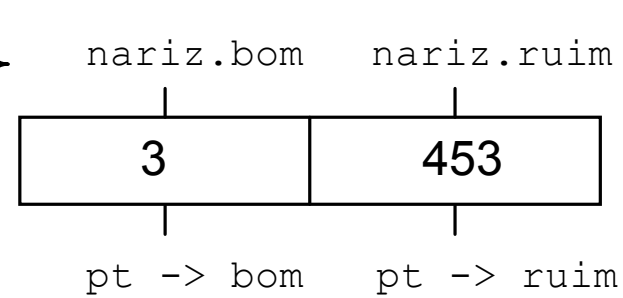
```
struct coisas
{
      int bom;
      int ruim;
};

coisas nariz = {3, 453};            // nariz é uma estrutura
coisas * pt = &nariz;               // pt aponta para a estrutura nariz
```

Use operador . com o nome da estrutura.

estrutura nariz

Use operador -> com ponteiro-para-estrutura.



Usando **`malloc`** para criar estruturas dinâmicas

```
#include <stdio.h>
#include <stdlib.h>
struct produto
{
     char nome[20];
     float volume;
     float preco;
};
void main()
{
    struct produto * ps;

    ps=(struct produto *)malloc(sizeof(struct produto));
     printf("Entre com o nome do item produto: ");
     gets(ps->nome);                // método 1 para acesso ao membro
     printf("Entre com o volume em centimetros cubicos: ");
     scanf("%f", &(*ps).volume);    // método 2 para acesso ao membro
     printf("Entre com o preco: R$");
     scanf("%f", &(ps->preco));
     printf("Nome: %s\n", (*ps).nome);      // método 2
     printf("Volume: %g centimetros cubicos\n", ps->volume);
     printf("Preco: R$%g\n", ps->preco);// método 1
}
```

Usando **`malloc`** para criar estruturas dinâmicas

- Uma saída possível para este programa:

```
Entre com o nome do item produto: Homem de Aço
Entre com o volume em centimetros cubicos: 23.2
Entre com o preco: R$23.99
Nome: Homem de Aço
Volume: 23.2 centimetros cubicos
Preco: R$23.99
```

Um exemplo de **malloc** e **free**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char * peganome(void);              // protótipo de função
void main()
{
    char * nome;           // cria ponteiro sem armazenamento
    nome = peganome();     // atribui endereço de string ao nome
    printf("%s em %d\n", nome, (int *) nome);
    free(nome);            // libera memória
    nome = peganome();     // reuso memória liberada
    printf("%s em %d\n", nome, (int *) nome);
    free(nome);            // libera memória de novo
}
char * peganome()          // retorna ponteiro para string nova
{
    char temp[80];         // armazenamento temporário
    char * pn;
    printf("Entre com o ultimo nome: ");
    gets(temp);
    pn=(char *)malloc(strlen(temp)+1);
    strcpy(pn, temp);      // copia string num espaço menor
    return pn;             // temp perdido quando a função termina
}
```

Um exemplo de **`malloc`** e **`free`**

- Uma saída possível seria:

```
Entre com o ultimo nome: Pinoquio
Pinoquio em 4398016
Entre com o ultimo nome: Nosferatu
Nosferatu em 4398016
```

Referências

- Prata, S. (1998). C++ Primer Plus. Mitchell Waite Signature Series. Waite Group Press.
- Curso de C do CPDEE/UFMG - 1996 – 1999 – Alocação Dinâmica: **http://www.mtm.ufsc.br/~azeredo/cursoC/aulas/ca60.html.**