8. Endereços e Ponteiros

Neste capítulo veremos a definição e os principais usos de ponteiros. Veremos as operações fundamentais como ponteiros , a estreita relação de ponteiros vetores e strings e ainda a alocação dinâmica de memória e a passagem de funções para funções com o uso de ponteiros..

9.1 Introdução

Toda informação (dado armazenado em variável simples ou vetor) que manipulamos em um programa está armazenado na memória do computador. Cada informação é representada por um certo conjunto de *bytes* (Ver capítulo 2). Por exemplo: caracter (`char`): 1 *byte*, inteiro (`int`): 2 *bytes*, etc.

Cada um destes conjuntos de *bytes*, que chamaremos de **bloco**, tem um nome e um **endereço** de localização especifica na memória.

Exemplo: Observe a instrução abaixo:

```
int num = 17;
```

ao interpretar esta instrução, o processador pode especificar:

Nome da informação: `num`

Tipo de informação: `int`

Tamanho do bloco (número de *bytes* ocupados pela informação): `2`

Valor da informação: `17`

Endereço da informação (localização do primeiro ***byte***): `8F6F:FFF2` (hexadecimal)

Em geral, interessa ao **programador** apenas os nomes simbólicos que representam as informações, pois é com estes nomes que são realizadas as operações do seu algoritmo. Porém, ao **processador** interessa os endereços dos blocos de informação pois é com estes endereços que vai trabalhar.

Programa Exemplo: O arquivo `e0801.cpp` contém um programa com instruções para inspecionar o endereço de uma variável, usando o recurso *Inspect* do Turbo C++. Observe que o

endereço mostrado corresponde ao **primeiro** *byte* do bloco, mesmo que o bloco ocupe mais de um *byte*: No caso, um float ocupa um bloco de 4 *bytes*.

8.2 Ponteiros

Ponteiros são variáveis que contém endereços. Neste sentido, estas variáveis *apontam* para algum determinado endereço da memória. Em geral, o ponteiro aponta para o endereço de alguma variável declarada no programa.

8.2.1 Declaração de ponteiros.

Quando declaramos um ponteiro, devemos declará-lo com o mesmo tipo (int, char, etc.) do bloco a ser apontado. Por exemplo, se queremos que um ponteiro aponte para uma variável `int` (bloco de 2 bytes) devemos declará-lo como `int` também.

Sintaxe: A sintaxe da declaração de um ponteiro é a seguinte:

tipo_ptr `*`*nome_ptr_1*`;`

ou

tipo_ptr`*` *nome_ptr_1*`,` *nome_ptr_2*`, ...;`

onde:

tipo_ptr `:` é o tipo de bloco para o qual o ponteiro apontará.

`* :` é um operador que indica que `nome_ptr` é um ponteiro.

nome_ptr_1`,` *nome_ptr_2*`,...:` são os nomes dos ponteiros (os nomes dos ponteiros obedecem as mesmas regras da seção 2.2.1)

Exemplo: Veja as seguintes instruções:

```
int *p;
float* s_1, s_2;
```

A primeira instrução declara um ponteiro chamado `p` que aponta para um inteiro. Este ponteiro aponta para o **primeiro** endereço de um bloco de **dois** *bytes*. Sempre é necessário declarar o tipo do ponteiro. Neste caso dizemos que declaramos um ponteiro tipo `int`.

A segunda instrução declara dois ponteiros (`s_1` e `s_2`) do tipo `float`. Observe que o `*` está justaposto ao tipo: assim todos os elementos da lista serão declarados ponteiros.

8.2.2 Operadores `&` e `*`

Quando trabalhamos com ponteiros, queremos fazer duas coisas basicamente:

- conhecer **endereço** de uma variável;
- conhecer o **conteúdo** de um endereço.

Para realizar estas tarefas a linguagem C nos providencia dois operadores especiais:

- o operador de **endereço**: &
- o operador de **conteúdo**: *

O operador de **endereço (`&`)** determina o endereço de uma variável (o primeiro *byte* do bloco ocupado pela variável). Por exemplo, `&val` determina o endereço do bloco ocupado pela variável `val`. Esta informação não é totalmente nova pois já a usamos antes: na função `scanf()`.

Exemplo: Quando escreve-se a instrução:

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

estamos nos referimos **endereço** do bloco ocupado pela variável `num`. A instrução significa: "*leia o buffer do teclado, transforme o valor lido em um valor inteiro (2 bytes) e o armazene no bloco localizado no endereço da variável* `num`".

Exemplo: Para se atribuir a um ponteiro o endereço de uma variável escreve-se:

```
int *p, val=5;  // declaração de ponteiro e variável
p = &val;       // atribuição
```

Observe que o ponteiro `p` deve ser declarado anteriormente com o mesmo tipo da variável para a qual ele deve apontar.

O operador **conteúdo** (`*`) determina o conteúdo (valor) do dado armazenado no endereço de um bloco apontado por um ponteiro. Por exemplo, `*p` determina conteúdo do bloco apontado pelo ponteiro p. De forma resumida: o operador (`*`) determina o conteúdo de um endereço.

Exemplo: Para se atribuir a uma variável o conteúdo de um endereço escreve-se:

```
int *p = 0x3f8, val;  // declaração de ponteiro e variável
val = *p;             // atribuição
```

Observações:

- O operador **endereço** (`&`) somente pode ser usado em uma única variável. Não pode ser usado em expressões como, por exemplo, `&(a+b)`.
- O operador **conteúdo** (`*`) somente pode ser usado em variáveis ponteiros.

Programa Exemplo: O arquivo `e0802.cpp` contém um programa que mostra como se manipulam ponteiros e variáveis. Como se transportam informações entre ponteiros e variáveis.

8.3 Operações elementares com ponteiros

Ponteiros são variáveis especiais e obedecem a regras especiais. Deste modo, existem uma série de operações (aritméticas, lógicas, etc.) envolvendo ponteiros que são permitidas e outras não. A seguir são destacadas algumas operações que podem ser executadas com ponteiros.

- A um ponteiro pode ser atribuído o endereço de uma variável comum.

Exemplo: Observe o trecho abaixo:

```
...
int *p;
int s;
p = &s;  // p recebe o endereço de s
...
```

- Um ponteiro pode receber o valor de outro ponteiro, isto é, pode receber o endereço apontado por outro ponteiro, desde que os ponteiros sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
float *p1, *p2, val;
p1 = &val;  // p1 recebe o endereço de val...
p2 = p1;    // ...e p2 recebe o conteúdo de p2 (endereço de val)
```

- Um ponteiro pode receber um endereço de memória diretamente. Um endereço é um numero inteiro. Em geral, na forma hexadecimal (`0x....`). Nesta atribuição devemos, em geral, forçar uma conversão de tipo usando *casting* para o tipo de ponteiro declarado.

Exemplo: Observe o trecho abaixo:

```
...
int *p1;
float p2ç
p1 = 0x03F8;  // endereço da porta serial COM1
p2 = (float)0x0FF6; // casting
...
```

- A um ponteiro pode ser atribuído o valor **nulo** usando a constante simbólica `NULL` (declarada na biblioteca `stdlib.h`). Um ponteiro com valor `NULL` não aponta para lugar nenhum! Algumas funções a utilizam para registrar uma atribuição ilegal ou sem sucesso (ver função `malloc()` adiante)

Exemplo: Observe o trecho abaixo:

```
#include <stdlib.h>
...
char *p;
p = NULL;
...
```

- Uma quantidade inteira pode ser adicionada ou subtraída de um ponteiro. A adição de um inteiro `n` a um ponteiro `p` fará com que ele aponte para o endereço do `n`-*ésimo* bloco seguinte.

Exemplo: Observe o trecho abaixo:

```
...
double *p, *q, var;
p = &var
q = ++p;  // q aponta para o bloco seguinte ao ocupado por var
p = q - 5;// p aponta para o quinto bloco anterior a q
...
```

- Dois ponteiros podem ser comparados (usando-se operadores lógicos) desde que sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
if(px == py){  // se px aponta para o mesmo bloco que py ...
if(px > py){   // se px aponta para um bloco posterior a py ...
if(px != py){  // se px aponta para um bloco diferente de py ...
if(px == NULL) // se px é nulo...
...
```

Programa Exemplo: O arquivo `e0803.cpp` contém um programa que mostra como se utilizam algumas operações elementares com ponteiros com ponteiros.

8.4 Ponteiros, endereços e funções

Porque usar ponteiros? A primeira vantagem da utilização de ponteiros em programas talvez esteja relacionada a sua utilização como argumentos de funções.

8.4.1 Passagem de dados por valor ou por referencia

No capítulo 6 (seção 6.5) afirma-se que o valor de uma variável `var` de uma função `fun_1()` passada para uma outra função `fun_2()` **não** podem ser alterado pela função `fun_2()`. De fato, isto é verdade se passamos o **valor** da variável `var` para a função `fun_2()`. Mas o valor de `var` pode ser alterado por `fun_2()` passamos seu **endereço.**

No primeiro caso, dizemos que a passagem de dados de uma função para outra ocorreu por **valor**. No segundo caso, dizemos que houve uma passagem por **referência**. Vejamos em detalhe uma definição destes tipos de passagem de dados entre funções:

Passagem por Valor: A passagem por valor significa que passamos de uma função para outra o **valor** de uma variável, isto é, a função chamada recebe um cópia do valor da variável. Assim qualquer alteração deste valor, pela função chamada, será uma alteração de uma cópia do valor da variável. O valor original na função chamadora **não é alterado** pois o valor original e copia ficam em blocos de memória diferentes.

Passagem por Referencia: A passagem por referencia significa que passamos de uma função para outra o **endereço** de uma variável, isto é, a função chamada recebe sua **localização** na memória

através de um ponteiro. Assim qualquer alteração no conteúdo apontado pelo do ponteiro será uma alteração no conteúdo da variável original. O valor original **é alterado**.

Sintaxe: A sintaxe da passagem de endereço é a seguinte:

- na função chamada:

```
tipof nomef(tipop nomep){
...
```

onde:

tipof é o tipo de retorno da função.

nomef é o nome da função chamada.

tipop é o tipo do ponteiro (igual ao tipo da variável passada).

nomep é o nome do ponteiro.

- na função chamadora:

```
...
nomef( end_var);
...
```

onde:

nomef é o nome da função chamada.

end_var é o endereço da variável passada como argumento.

Exemplo: Observe o exemplo abaixo:

```
void troca(int *p1, int *p1){ // declaração da função
                              //   Observe: ponteiros
  int temp;      // variável temporária
  temp = *p1;    // temp recebe o conteúdo apontado por p1
  *p1 = *p2;     // o conteúdo de p1 recebe o conteúdo de p2
  *p2 = temp;    // o conteúdo de p2 recebe o valor de temp
}
void main(){             // programa principal
  int a,b;               // declaração das variáveis
  scanf("%d %d",&a,&b);  // leitura das variáveis
  troca(&a,&b);          // passagem dos endereços de a e b
  printf("%d %d",a,b);   // imprime valores (trocados!)
}
```

Neste exemplo temos uma função `troca()` que troca entre si os valores de duas variáveis. Esta função recebe os **endereços** das variáveis passadas pela função `main()`, armazenando-os nos ponteiros `p1` e `p2`. Dentro da função, troca-se os **conteúdos** dos endereços apontados.

Programa Exemplo: O arquivo `e0804.cpp` contém um programa que mostra a diferença entre a passagem de dados por valor e passagem por referencia.

8.4.2 Retorno de dados em funções

A **passagem por referencia** permite que (formalmente) uma função retorne quantos valores se desejar. Dissemos no capítulo 6 (seção 6.2) que uma função somente pode retornar um valor. Isto continua sendo valido pois o C assim define funções. Porem com o uso de ponteiros pode-se contornar esta situação. Vejamos:

Imagine que queremos escrever uma função `stat()` com a finalidade de calcular a *media aritmética* e o *desvio padrão* de um conjunto de dados. Observe: o retorno destes dados não poder ser via instrução `return()` pois isto não é permitido. A solução é criar (na função `main()`, por exemplo) duas variáveis para armazenar os valores desejados (`med` e `desvio`, por exemplo) e então passar os endereços destas variáveis para a função `stat()`. A função recebe esses endereços e os armazena em ponteiros (`pm` e `pd`, por exemplo). Em seguida, faz os cálculos necessários, armazenando-os nos endereços recebidos. Ao término da execução da função os valores de `med` e `desvio` serão atualizados automaticamente. Este recurso é bastante utilizado por programadores profissionais.

Programa Exemplo: O arquivo `e0805.cpp` contém um programa que mostra o *'retorno'* de vários dados de uma função. Na verdade trata-se da passagem de valores por referencia.

8.4.3 Ponteiro como argumento de função

Observe que nos exemplos acima, a passagem de endereços foi feita através do *operador endereço* (&). Também é possível passar um endereço através de um ponteiro já que o conteúdo de um ponteiro **é** um endereço.

Exemplo: Observe o trecho de programa abaixo.

```
...
float *p, x;
```

```
p = &x;
função(p); // passagem do ponteiro com o endereço de x.
...
```

Programa Exemplo: O arquivo `e0806.cpp` contém um programa que mostra a passagem de endereço para uma função usando um ponteiro. Observe a sintaxe alternativa para a função `scanf()`!

8.5 Ponteiros, vetores e *strings*

8.5.1 Ponteiros e vetores

No capitulo 7 foi mostrado que na passagem de vetores para funções especifica-se apenas o nome do vetor e que modificações nos elementos do vetor dentro da função chamada alteram os valores do vetor no programa chamador (seção 7.4). Isto se deve ao fato de que, na linguagem C, **vetores** são intimamente relacionados a **ponteiros**.

Em C, o **nome** de um vetor é tratado como o **endereço** de seu primeiro elemento. Assim ao se passar o nome de um vetor para uma função está se passando o endereço do primeiro elemento de um conjunto de endereços de memória.

Por exemplo, se `vet` é um vetor, então `vet` e `&vet[0]` representam o mesmo **endereço**. E mais, podemos acessar o endereço de qualquer elemento do vetor do seguinte modo: `&vet[i]` e equivalente a `(vet + i)`. Aqui deve-se ressaltar que `(vet + i)` não representa uma **adição** aritmética normal mas o **endereço** do *i-ésimo* elemento do vetor `vet` (endereço contado a partir do endereço inicial `vet[0]`).

Do mesmo modo que se pode acessar o **endereço** de cada elemento do vetor por ponteiros, também se pode acessar o **valor** de cada elemento usando ponteiros. Assim `vet[i]` é equivalente a `*(vet + i)`. Aqui se usa o operador conteúdo (`*`) aplicado ao endereço do *i-ésimo* elemento do vetor `vet`.

Programa Exemplo: O arquivo `e0807.cpp` contém um programa que mostra a equivalência entre ponteiros e vetores.

8.5.2 Ponteiros e *strings*

Como dissemos, vagamente na seção 2.3.4, uma *string* é um *conjunto ordenado de caracteres*. Podemos, agora, dizer muito mais: Em C, uma *string* é um **vetor unidimensional** de elementos caracteres ASCII, sendo o ultimo destes elementos o caracter especial `'\0'`.

Sintaxe: As duas maneiras mais comuns de declararmos uma *string* são:

```
char nome[tam];
```

ou

```
char *nome;
```

onde:

nome é o nome do vetor de caracteres e

tam seu tamanho.

Observe que sendo um vetor, uma *string* pode ser declarada também como um ponteiro. Alias a segunda declaração representa justamente isto. Sabendo isto podemos realizar uma grande variedade de manipulações com *strings* e caracteres. Existe uma biblioteca padrão C chamada `string.h` que providencia algumas funções de manipulação de *strings* muito úteis.

Programa Exemplo: O arquivo `e0808.cpp` contém um programa que mostra algumas operações usando-se strings (vetores e ponteiros).

8.6 Alocação Dinâmica de Memória

Os elementos de um vetor são armazenados **seqüencialmente** na memória do computador. Na declaração de um vetor, (por exemplo: `int vet[10]`) é dito ao processador reservar (**alocar**) um certo numero de blocos de memória para armazenamento dos elementos do vetor. Porem, neste modo de declaração, não se pode alocar um numero variável de elementos (veja seção 7.3.2).

A linguagem C permite alocar dinamicamente (em tempo de execução), blocos de memória usando ponteiros. Dada a intima relação entre ponteiros e vetores, isto significa que podemos declarar dinamicamente vetores de tamanho variável. Isto é desejável caso queiramos poupar memória, isto é não reservar mais memória que o necessário para o armazenamento de dados.

Para a alocação de memória usamos a função `malloc()`(*memory allocation*) da biblioteca `alloc.h`. A função `malloc()` reserva, seqüencialmente, um certo numero de blocos de memória e retorna, para um ponteiro, o endereço do primeiro bloco reservado.

Sintaxe: A sintaxe geral usada para a alocação dinâmica é a seguinte:

```
pont = (tipo *)malloc(tam);
```

onde:

pont é o nome do ponteiro que recebe o endereço do espaço de memória alocado.

tipo é o tipo do endereço apontado (tipo do ponteiro).

tam é o tamanho do espaço alocado: numero de *bytes*.

A sintaxe seguinte, porem, é mais clara:

```
pont = (tipo*)malloc(num*sizeof(tipo));
```

onde:

num é o numero de elementos que queremos poder armazenar no espaço alocado.

Exemplo: Se queremos declarar um vetor chamado `vet`, tipo `int`, com `num` elementos podemos usar o trecho abaixo:

```
...
int *vet;  // declaração do ponteiro
vet = (int*)malloc(num*2); // alocação de num blocos de 2 bytes
...
```

ou ainda

```
...
int *vet; // declaração do ponteiro
vet = (int*) malloc(num * sizeof(int));
...
```

Caso não seja possível alocar o espaço requisitado a função `malloc()` retorna a constante simbólica `NULL`.

Sintaxe: Para liberar (desalocar) o espaço de memória se usa a função `free()`, cuja sintaxe é a seguinte:

```
free( pont);
```

onde:

pont é o nome do ponteiro que contem o endereço do inicio do espaço de memória reservado.

Programa Exemplo: O arquivo `e0809.cpp` contém um programa que mostra como se utiliza a Alocação Dinâmica de Memória.

8.7 Ponteiros para Funções

Até agora usamos ponteiros para apontar para endereços de memória onde se encontravam as variáveis (dados). Algumas vezes é necessário apontar para funções, isto é, apontar para o endereço de memória que contem o inicio das instruções de uma função. Quando assim procedemos, dizemos que usaremos *ponteiros para funções*.

8.7.1 Ponteiros como chamada de função.

Um uso de ponteiros para funções é passar uma função como **argumento** de outra função. Mas também se pode usar ponteiros para funções ao invés de funções nas chamadas normais de funções.

Sintaxe: A sintaxe de declaração de ponteiro para funções é a seguinte:

```
tipo_r (*nome_p)(lista);
```

onde:

tipo_r é o tipo de retorno da função apontada.

nome_p é o nome do ponteiro que apontara para a função.

lista é a lista de argumentos da função.

Exemplo: Suponha que temos uma função é declarada como:

```
float fun(int a, int b){
  ...
}
```

o ponteiro correspondente será:

```
float (*pt)(int,int);
```

Observe que o ponteiro para função **deve ser** declarado entre parênteses. Observe também que o ponteiro e a função retornam o mesmo tipo de dado e que tem os mesmos argumentos.

Sintaxe: Para atribuirmos o endereço de uma função para um ponteiro usamos a seguinte sintaxe:

```
pont = &função;
```

onde:

pont é o nome do ponteiro.

função é o nome da função.

Se um ponteiro contem o endereço de uma função, ele pode ser usado no lugar da chamada da função.

Exemplo: o trecho de programa abaixo usa um ponteiro para chamar uma função:

```
float fun(int a,int b){
  ...
}
void main(void){
  float temp;
  float (*pt)(int,int);
  pt = &fun;
  temp = (*pt)(10,20); // eqüivale a: temp = fun(10,20);
  ...
}
```

Programa Exemplo: O arquivo `e0810.cpp` contém um programa que mostra como se utiliza o ponteiro para função.

8.7.2 Passando uma função como argumento de outra função.

Outra utilização de ponteiros para funções é na passagem de uma **função** como **argumento** para outra função. Para que isso ocorra necessitamos:

- Na declaração da **função a ser passada**:

i) Nada de especial, apenas a definição normal:

```
tipo nome_p(lista){
  ...
}
```

Exemplo:

```
float soma(float a,float b){
  ...
}
```

- Na **função receptora**:

i) Declarar o **ponteiro** que recebe a **função passada** na lista de argumentos:

```
tipo nome_r(..., tipo (*pt)(lista), ...){
```

Exemplo:

```
float grad(float x, float y, float (*p)(float,float)){
```

ii) Usar o ponteiro para funções nas chamadas da função passada:

```
var = (*pt)(lista);
```

Exemplo:

```
valor = (*p)(x,y);
```

- Na função principal:

i) Passar o nome da função chamada para a função receptora:

```
var = nome_g(... , nome_p , ...);
```

Exemplo:

```
g = grad(x,y,soma);
```

Programa Exemplo: O arquivo `e0811.cpp` contém um programa que mostra como se utiliza ponteiros na passagem de funções