Ponteiros em C

Prof. Ricardo Reis
Universidade Federal do Ceará
Campus de Quixadá

22 de março de 2013

1 Introdução

Um **ponteiro** é definido como uma variável cujo conteúdo é o endereço de memória de uma outra variável. Diz-se que um ponteiro *aponta* para a variável cujo endereço de memória ele armazena. A variável apontada por um ponteiro pode ser lida ou alterada através desse ponteiro indiretamente.

Listagem 1: Uso básico de Ponteiros

```
1 int x = 23;
2 int* p = &x;
3 *p = 67;
4 printf("%d", x); // 67
```

Na Listagem-1 são ilustrados os usos dos operadores básicos de ponteiros em C: `&` e `*`. O operador `&` (que não é o mesmo da conjunção bit-a-bit, apesar da mesma simbologia) aparece prefixado a variável x indicando que o endereço dela está sendo requisitado. Assim a expressão na linha-2, `&x`, denota o endereço da variável `x`.

O operador `*` (que *não* é o mesmo da multiplicação aritmética, apesar também da mesma simbologia) tem duas aplicações distintas no âmbito dos ponteiros. Na linha-2 ele aparece como *operador de declaração* de um novo ponteiro e é disposto entre um tipo existente (neste caso **`int`**) e o nome da nova variável ponteiro (neste caso `p`). Diz-se que **`int`** é o *tipo base* do ponteiro `p`. Assim a expressão **`int`**`* p = &x` atribui a `p` o endereço de `x` (`p` aponta para `x`).

É conveniente que ponteiros apontem para variáveis que compartilhem de seu mesmo tipo base, mas nada impede que sejam distintos (de fato algumas vantagens podem ser tiradas neste sentido, como veremos adiante).

O segundo uso do operador `*` ocorre na linha-3 e é conhecido como *desreferenciamento* de ponteiro. Consiste num mecanismo que permite o acesso ao valor da variável apontada pelo ponteiro através deste ponteiro. A prefixação de `*` à variável ponteiro `p` indica que a célula de memória a ser lida ou alterada é aquela apontada por este ponteiro e *não* a do próprio ponteiro. Assim a expressão `*p = 2655` desreferencia `p` e armazena 2655 na variável `x` apontada. Em contrapartida fazer a atribuição `p = 2655` alterá o endereço de memória armazenado por `p` para `0xA5F`, que é a representação hexadecimal de 2655. Isso é perigoso pois não existem garantias que se pode ler ou escrever nesta parte da memória. Veja a sessão-8.

Nem todo ponteiro em C possui um tipo base. Na declaração **`void`**`* pt` o ponteiro `pt` não possui um tipo base. De forma geral a construção **`void`**`*` é utilizada para declarar ponteiros sem tipo base. Um ponteiro com tipo base é denominado *ponteiro tipado* ao passo que um declarado com **`void`**`*` é denominado *ponteiro não-tipado*. Um ponteiro não-tipado pode receber o endereço de qualquer tipo de variável. No exemplo, **`void`**`* pt = (`**`void`**`*) &x` a variável `x` pode ser de diversos tipos. Observe o uso do modelador `(`**`void`**`*)` para evitar mensagens de advertência do compilador.

2 Aplicações dos ponteiros

Ponteiros possuem duas aplicações: o *referenciamento indireto* e o *gerenciamento dinâmico de memória*, descritos nas sessões seguintes.

2.1 Referenciamento Indireto

Trata-se por referenciamento indireto o mecanismo ilustrado na Listagem-1. Nesta listagem o ponteiro `p` *refere-se indiretamente* a variável `x` com permissões de leitura e escrita.

Outra aplicação do referenciamento indireto está na relação entre ponteiros e vetores em C. Na Listagem-2 o ponteiro `p` é apontado para a primeira célula do vetor `u` pela atribuição `int* p = &u[0]` e depois utilizado, dentro do laço **for** (linhas 4 e 5), exatamente como o próprio vetor `u` incluindo a capacidade de uso do operador pós fixo `[]`. Em outras palavras `u[k]` é equivalente a `p[k]` (com $k \in 0..3$)

Listagem 2: Relação entre ponteiros e vetores

```
1 int fnc() {
2     int u[] = {2, 7, 9, 11}, k;
3     int* p = &u[0];
4     for (k=0; k<4; k++)
5         printf("%d ", p[k]);
6 }
```

O comportamento do operador `[]` pós-fixado à ponteiros em C revela uma característica importante da linguagem: um vetor é na realidade um ponteiro que aponta para uma sequencia de valores de mesmo tipo (o tipo base). Na declaração, por exemplo, `float w[3] = {7.0, 0.3, 9.8}`, `w` é um ponteiro de tipo base **float** e `&w[0], &w[1], &w[2]` são respectivamente os endereços de suas células.

Descrevemos a seguir a *aritmética de ponteiros* em C. Ponteiros não podem ser somados, subtraídos, multiplicados ou divididos entre si mas podem ser somados ou subtraídos a valores inteiros, operações essas que retornam valores de endereços nas vizinhanças do endereço armazenado no ponteiro. O total somado ou subtraído de um ponteiro tipado *não* equivale a um número de bytes de deslocamento, mas sim a um total de blocos de bytes com tamanho igual ao do tipo base, por exemplo, se um ponteiro é declarado como `double* ptr`, então `ptr + 3` e `ptr - 3` representam os endereços obtidos deslocando-se 24 bytes (um **double** possui 8 bytes) respectivamente para frente e para trás em relação ao endereço armazenado em `ptr`.

Na Listagem-3, linha-5, a expressão `p + k` é utilizada ao invés de `p[k]`. De forma geral a aritmética de ponteiros em C define que, se `p` é um ponteiro tipado que aponta para a primeira célula de um vetor, então `p + k` equivale a `p[k]`. Consequentemente `p+0`, que vale `p`, refere-se ao endereço da primeira célula do vetor apontado. Por essa razão na linha-3, Listagem-3, a atribuição `int* p = u` substitui `int* p = &u[0]`.

Listagem 3: Aritmética de ponteiros

```
1 int fnc() {
2     int u[] = {2, 7, 9, 11}, k;
3     int* p = u; // &u[0]
4     for (k=0; k<4; k++)
5         printf("%d ", p + k);
6 }
```

No caso de ponteiros não-tipados, como não existe tipo base, a aritmética é feita somando-se ou subtraindo-se quantidades de bytes ao/do endereço que o ponteiro armazena. O exemplo da Listagem-4 ilustra essa questão. Em cada iteração do laço **for** o valor do endereço em `p` (inicializado com o endereço de `u`, linha-5) é incrementado na linha-8 do total de bytes de um inteiro (`sizeof(int)`) fazendo com que `p` aponte em cada passo para uma célula distinta de `u`. Como `p` é não tipado então para resgatar o valor inteiro da memória deve-se primeiro usar um modelador `(int*)` para tipar o ponteiro e em seguida desreferenciá-lo (linha-7).

Listagem 4: Aritmética de ponteiros não tipados

```
1 #include <stdio.h>
2
3 int main() {
4     int u[] = {2, 7, 9, 11}, k;
5     void* p = (void*) u;
```

```
6       for (k=0; k<4; k++) {
7           printf("%d ", *( (int*) p ));
8           p += sizeof(int);
9       }
10      return 0;
11  }
```

2.2 Gerenciamento Dinâmico de Memória

Todas as variáveis utilizadas por um programa precisam naturalmente de memória. Em linguagem C existem quatro mecanismos que permitem alocação de memória para uma variável. São estes,

a. Alocação estática

b. Alocação dinâmica fixa em *stack*

c. Alocação dinâmica flexível em *stack*

d. Alocação dinâmica em *heap*

Na *alocação estática* as variáveis possuem nome e tipo definido em código e uma vez alocadas se mantêm até o fim execução quando então a memória é *devolvida* ao sistema. Na Listagem-5 as variáveis `x` e `y` são alocadas estaticamente. Em C o modificador **`static`** ocasiona alocação estática tardia a variáveis locais, ou seja, `y` só será alocada após a primeira chamada a função `fnc()` mantendo-se alocada mesmo quando esta devolver o controle ao chamador. Em chamadas futuras de `fnc()` a variável `y` não será mais inicializada: de fato ainda terá o valor que assumia no fim da última chamada a `fnc()` (diz-se que `y` é *sensível ao contexto*).

Na alocação dinâmica em *stack* (termo em inglês que significa *pilha* e que indica que dados são sistematicamente empilhados, utilizados e depois desempilhados), memória é alocada temporariamente dentro de uma função e devolvida ao sistema quando esta retorna o controle a seu chamador.

No caso de variáveis escalares (não-vetores) e vetores de comprimento constante (definido em código), como `u` na Listagem-5, a alocação dinâmica em *stack* é *fixa*, ou seja, a quantidade de memória alocada pela variável não varia entre chamadas distintas da função que a encapsula.

Vetores locais podem ter seu comprimento expresso em função de um ou mais argumentos de entrada da função onde estão definidos, ou seja, podem variar de comprimento entre chamadas distintas desta função. Por exemplo, o comprimento do vetor `v` na Listagem-5 é definido pelo argumento `n` de entrada de `fnc()`. Este forma de alocação é chamada dinâmica *flexível* em *stack*.

Listagem 5: Tipos de alocação em C

```
1   #include <malloc.h>
2
3   int x[3] = {1, 5, 9};
4
5   void fnc(int n) {
6       static int y[3] = {11, -6, 8};
7       int u[3];
8       int v[n];
9       int* w =
10          (int*)malloc(3*sizeof(int));
11      int k;
12      if (n==3)
13          for (k=0;k<3; k++)
14              u[k] = v[k]
15                   = w[k]
16                   = k*k + 1;
17      free(w);
18  }
```

Em ambas categorias de alocação em *stack* as variáveis alocadas não são *persistentes*, ou seja, no final do escopo são desalocadas. Na Listagem-5 as variáveis `u` e `v` são eliminadas sempre que `fnc()` retorna o controle ao chamador. Utilizando ponteiros entretanto é possível recorrer à alocação dinâmica para alocar dados que persistem ao fim do escopo onde são criados. É a *alocação dinâmica em heap* (termo em inglês que, apesar de também ser traduzido como pilha, não é semanticamente equivalente a *stack* sendo utilizado comumente para se referir a memória de alocação/desalocação manual).

A alocação dinâmica em heap na linguagem C é obtida através do uso de funções apropriadas (outras linguagens utilizam

operadores dedicados, como o C++). Estas funções são acessíveis via arquivo cabeçalho `malloc.h`. As principais são `malloc()` utilizada para alocação e `free()` utilizada para desalocação. A função `malloc()` possui como argumento único a quantidade de bytes que deve alocar e retorna um ponteiro **`void`**`*` que aponta para a região alocada. É comum usar a função **`sizeof`**`()` para calcular a quantidade de memória que se deseja alocar. Na linha-10 da listagem-5 é alocado um bloco com tamanho de 3 inteiros (`3 *` **`sizeof`**`(`**`int`**`)`). A saída de `malloc()` é `void*` que precisa, se for o caso, ser modulado para se tornar um ponteiro tipado de base apropriada. Na listagem-5 a variável `w`, que é de tipo **`int`**`*`, recebe a saída de `malloc()` modulada por `(`**`int`**`*)`.

A função `free()`, que também só possui um argumento, recebe como entrada o ponteiro que aponta para a região de memória que precisa ser desalocada (Chamadas a `free()`, que recebem como entrada um ponteiro para um vetor estático ou dinâmico em stack, não terão efeito algum). Na linha-17 da listagem-5 a região apontada por `w` é desalocada. É importante observar que `free()` não afeta o ponteiro que recebe como argumento e sim a parte da memória cujo endereço ele armazena.

O bloco de memória alocado por `malloc()` é uma variável *sem nome* que precisa de um ponteiro para ser acessada. Assim a instrução na linha-10, listagem-5, envolve de fato duas variáveis, a primeira de alocação estática, `w`, e a segunda, sem nome, de alocação dinâmica. Neste exemplo, quando o escopo de `w` se encerra esta variável é eliminada mas aquela que não possui nome se mantém. Por essa razão é feita a chamada de `free()` previamente para que ela não fique *perdida* em memória. Se essa variável precisar ser retornada (já que ela pode persistir ao fim do escopo onde fora criada, então essa possibilidade existe) então naturalmente `free()` não deverá ser utilizado. O retorno de um endereço por uma função será abordado na sessão-4.

3 Ponteiros e Estruturas

Uma estrutura em C é similar a um vetor no que diz respeito a agregação de dados contíguos em memória. A diferença está na heterogeneidade dos dados, ou seja, uma estrutura em C permite que dados de tipos diferentes estejam numa mesma variável. Estruturas substituem o esquema de indexação (inerente da homogeneidade dos vetores) por um sistema de nomeação em rótulos, ou seja, cada parte de uma variável estrutura precisa de um nome, ou *campo*, para ser acessada.

Listagem 6: Declaração de estruturas em C

```
1  struct nome_tipo_1 {     // modo 1
2      /* campos */
3  };
4
5  typedef struct {     // modo 2
6      /* campos */
7  } nome_tipo_2;
8
9  struct nome_tipo_1 a;
10 nome_tipo_2 b;
```

A listagem-6 ilustra dois modos diferentes de definição de tipos estruturas em C. Pelo *modo 1* o nome da estrutura não é definido como novo tipo fazendo a declaração de uma variável por este modo requerer a palavra reservada **`struct`** como ilustrada na linha-9 pela variável `a`. No *modo 2* o uso de **`typedef`** define o nome da estrutura como um novo tipo permitindo a remoção de **`struct`** e sintetizando a construção. Isso é ilustrado na linha-10 pela declaração de `b`.

Os campos de uma estrutura são contíguos em memória e por essa razão pode-se inicializar uma variável estrutura com uma lista cujos elementos se disponham na ordem original da definição. Na listagem-7, linha-9, a variável `x` de tipo estrutura `T` é inicializada conforme esse recurso. Observe que se os campos da estrutura `T` fossem trocados, a inicialização da linha-9 não funcionaria apropriadamente.

Cada campo de uma variável estrutura pode ser tratado como uma variável independente. O acesso é feito com o operador

ponto (.) pela sintaxe `<var>.<campo>` como é ilustrado por `y.ch` na linha-11 e `y.data` na linha-14, listagem-7.

Listagem 7: Inicialização, cópia e uso de uma estrutura

```
1  #include <stdio.h>
2
3  typedef struct {
4      char ch;
5      int data[3];
6  } T;
7
8  int main() {
9      T x = { 'c', 45, 89, 11 };
10     T y = x;
11     printf("%c\n", y.ch);
12     int k;
13     for (k=0; k<3; k++)
14         printf("%d ", y.data[k]);
15     return 0;
16 }
```

Para copiar o conteúdo de uma variável estrutura para outra de mesmo tipo, pode-se utilizar uma atribuição simples como na linha-10, listagem-7. Entretanto, se ponteiros estiverem presentes entre os campos da estrutura, então a cópia poderá não proceder como esperado. Este aspecto é discutido na sessão-3.1.

3.1 Ponteiros como Campos de Estruturas

Um campo ponteiro em uma estrutura funciona exatamente como qualquer outro ponteiro e é acessado via operador ponto (.) como qualquer outro campo de estrutura. A listagem-8 mostra o uso básico de ponteiros em estruturas. As linhas 12 a 14 ilustram o desreferenciamento e, como o operador ponto(.) possui precedência maior que o `*`, não é necessário o uso de parênteses (como em `*(u.x) += 1`).

Listagem 8: Operando campos ponteiros em estruturas

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  typedef struct {
5      int *x;
6      int *y;
7  } res;
8
9  int main() {
10     int x = 9, y = 11;
11     res u = {&x, &y};
12     *u.x += 1;
13     *u.y += 1;
14     printf("%d %d", *u.x, *u.y);
15     return 0;
16 }
```

A listagem-9 ilustra uma estrutura contento um campo ponteiro que deve apontar para um vetor alocado dinamicamente (em heap). Alocação e desalocação ocorrem como já descrito. Entretanto a cópia entre variáveis de tipo estrutura, contendo ponteiros, requer atenção. Em caso de atribuição direta, como na linha-14, a cópia será *rasa*, ou seja, o campo ponteiro da variável-cópia (`y`) conterá o mesmo endereço que o campo ponteiro da variável original (`x`). Conclusão, a cópia rasa não copia a parte alocada dinamicamente, ao invés disso faz todas as partes envolvidas apontarem para o mesmo vetor. Tal comportamento está demonstrado nas linhas 17 e 18: as alterações feitas no campo `data` de `y` afetam o campo `data` de `x`, simplesmente por se tratarem do mesmo espaço de memória.

Listagem 9: Ponteiro como campo de uma estrutura

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  typedef struct {
5      int len;
6      int *data;
7  } T;
8
9  int main() {
10     T x = {
11         5,
12         (int*) malloc( 5*sizeof(int) )
13     };
14     T y = x;
15     int k;
16     for (k=0; k<x.len; k++) {
17         y.data[k] = k*k+1;
18         printf("%d ", x.data[k]);
19     }
20     free(x.data);
21     return 0;
22 }
```

Para construir uma cópia *profunda*, ou seja, com dados idênticos e não comparti-

lhados, é necessário alocar, para a nova variável, espaço novo para ser apontado pelos campos ponteiros e copiar para ele os dados originais. A função `copia()`, listagem-10, executa esta tarefa (supõe-se nestes exemplos que o comprimento do vetor dinâmico apontado por `data` é igual ao valor do campo `len`). Mais detalhes sobre ponteiros e funções consulte a sessão-4.

Listagem 10: Cópia profunda de uma estrutura contendo campo ponteiro apontando para um vetor alocado dinamicamente

```
1  T copia(T x) {
2      T res = {
3          x.len,
4          (int*) malloc(x.len*sizeof(int))
5      };
6      int k;
7      for (k=0; k<x.len; k++)
8          res.data[k] = x.data[k];
9      return res;
10 }
```

3.2 Ponteiros de Estruturas

A linguagem C também permite que estruturas completas sejam referenciadas por ponteiros. São os *ponteiros de estruturas*. Através deles é possível acessar individualmente cada um dos campos da estrutura apontada. Associando-se as sintaxes do desreferenciamento (uso do `*` prefixado) e de resolução de campos (uso do operador ponto) constrói-se expressões de acesso como no exemplo, `(*ptr).x`, onde `ptr` é um ponteiro de estrutura e `x` um campo da estrutura apontada. O uso dos parênteses em `(*p).x` é obrigatório porque a precedência de `*` é menor que a do ponto (`.`).

A sintaxe `(*var).campo` pode ser reduzida pelo uso do operador nativo *seta*, `->`. Este operador executa conjuntamente o desreferenciamento e resolução de campo e tem sintaxe básica `var->campo`. Na listagem-11, linha-11, é ilustrado o uso do operador seta.

Listagem 11: Ponteiro de estrutura e o operador seta.

```
1  #include <stdio.h>
2
3  typedef struct {
4      float x, y;
5  } ponto;
6
7  int main() {
8      ponto Q = {1.5, 8.1};
9      ponto *p = &Q;
10     printf("%f %f\n",
11            p->x, p->y);
12     // equivalente a,
13     // printf("%f %f\n",
14     //        (*p).x, (*p).y);
15     return 0;
16 }
```

3.3 Vetores de Estruturas

De forma análoga a vetores de escalares, um vetor de estruturas em C pode ser manipulado pelo uso da indexação. Neste caso o operador `->` pode ser dispensado porque a aritmética do operador de indexação, `[]`, promove o desreferenciamento. Por exemplo, na listagem-12, linha-15, a expressão `L[k].x` utiliza ponto em vez de seta porque o desreferenciamento já foi resolvido por `[k]`. Quando `->` é utilizado com o nome do vetor (que é de fato um ponteiro) o efeito é o acesso apenas a primeira célula do vetor conforme ilustrado na linha-12. Entretanto se aritmética explícita for utilizada, o operador `->` se tornará obrigatório. O comentário na linha-18 mostra uma alternativa a linha-15 utilizando aritmética explícita e operador seta.

Listagem 12: Utilizando um vetor de estruturas.

```
1  #include <stdio.h>
2
3  typedef struct {
4      int x, y;
5  } ponto;
6
7  int main() {
8      ponto L[] =
9         { {1,2}, {5,7}, {9,1} };
10     int k;
11     printf("%d %d\n",
12        L->x, L->y);
13     for (k=0; k<3; k++)
14         printf("%d %d\n",
15                L[k].x, L[k].y);
16 //  equivalente a,
17 //      printf("%d %d\n",
```

```
18 //              (L+k)->x, (L+k)->y);
19     return 0;
20 }
```

De forma geral, se `vec` é um ponteiro para um vetor de estruturas então `(vec+k)->campo` equivale a `vec[k].campo`.

4 Ponteiros e Funções

Ponteiros podem ser argumentos em funções, retornados por funções ou mesmo apontarem para funções. As sessões a seguir tratam destes aspectos.

4.1 Ponteiros como Argumentos de Funções

De forma geral um argumento-ponteiro numa função tem duas aplicações,

1. Contornar a cópia de dados do argumento real (o que é repassado) para o formal (o que recebe, na função) útil em situações onde os dados a serem repassados ocupam muito espaço (na prática repassá-los significa usar o dobro de memória!).

2. Permitir que a função possa *escrever* provisoriamente para uma variável não **const** fora de seu escopo.

Como exemplo da aplicação-(1) tem-se o caso de repasse de um vetor como argumento para uma função. Na listagem-13 a função `soma()` recebe um vetor através do ponteiro `p` e seu comprimento através de `n` e retorna o somatório dos valores deste vetor. Na linha-12 é construída uma chamada a `soma()` onde `v` é repassado *por referência* (sem duplicação de dados).

Listagem 13: Função de soma dos elementos de um vetor de inteiros

```
1  #include <stdio.h>
2
3  int soma(int *p, int n) {
4      int res = 0, k;
5      for (k=0; k<n; k++)
6          res += p[k];
7      return res;
8  }
9
10 int main() {
11     int v[] = {1,7,23,5,9};
12     printf("%d\n", soma(v, 5)); //45
13     return 0;
14 }
```

Uma alternativa sintática ao protótipo na função `soma` na listagem-13 é,

```
int soma(int p[], int n);
```

As expressões **int*** `p` e **int** `p[]`, quando utilizadas como argumentos de funções, são equivalentes. Em geral o segundo caso é aplicado quando se quer enfatizar que o ponteiro repassado aponta para um vetor e não para um escalar.

Como exemplo da aplicação-(2) tem-se a busca pelos valores extremos (mínimo e máximo) em um vetor de inteiros, conforme listagem-14. A função `minmax()` possui três argumentos-ponteiros: o primeiro para receber o endereço do vetor a ser processado (`p`) e os outros dois (`pmin` e `pmax`) para encaminhar, ás variáveis externas que serão repassadas, os respectivos valores mínimo e máximo encontrados. Conforme linha-20 estas variáveis são `x` e `y` (`minmax()` adquire controle provisório sobre `x` e `y` pois recebe seus endereços). O resultado é uma função *com dois retornos*, mas sem uso de **return**.

Listagem 14: Determinação dos valores extremos em um vetor de inteiros

```
1  #include <stdio.h>
2
3  void minmax(int *p,
4              int n,
5              int *pmin,
6              int *pmax)
7  {
8      int k=0;
9      *pmin = *pmax = p[0];
10     for (k=1; k<n; k++)
11         if (p[k]<*pmin)
12             *pmin = p[k];
13         else if (p[k]>*pmax)
14             *pmax = p[k];
15 }
16
17 int main() {
18     int v[] = {1,7,23,5,9};
19     int x, y;
```

```
20      minmax(v, 5, &x, &y);
21      printf("%d %d\n", x, y);
22      return 0;
23  }
```

O que ocorre se um vetor como,

```
const int v[] = {1,7,23,5,9};
```

é repassado para uma função como a da listagem-13? Note que `v` contém um modificador **`const`** que impede a alteração de seus dados. Em tais casos o compilador indica um erro pois o argumento-ponteiro, de tipo **`int`** `*`, teoricamente, pode ler e escrever para onde aponta, ao contrário de **`const int`**`*`, que só pode ler. Esta é uma violação de acesso.

No caso da função `soma()` na listagem-13, onde não há necessidade de modificação do vetor de entrada, o protótipo poderá ser reescrito como,

```
int soma(const int* p, int n)
```

operando assim tanto ponteiros **`const`** quanto não-**`const`**.

4.2 Funções que Retornam Ponteiros

Se uma função em C precisa retornar um ponteiro, um operador `*` precisa ser disposto entre um tipo base (ou um **`void`**) e o nome da função. Por exemplo, o protótipo **`int`**`* fnc(`**`int`** `x)` denota uma função que recebe um valor inteiro através de `x` e retorna um endereço de inteiro.

Um ponteiro retornado por função normalmente ou é uma cópia de algum ponteiro passado como argumento a esta função ou contém o endereço de alguma variável alocada dinamicamente em heap no escopo da função. Não há sentido em retornar ponteiros de variáveis estáticas locais ou dinâmicas em stack porque são desalocadas quando a função se encerra (veja sessão-8).

A listagem-15 trás a função `maiuscula()` que converte uma variável string para sua versão maiúscula. Observe que `maiuscula()` retorna uma cópia do ponteiro de entrada `str` cuja string apontada é modificada pela função. Isso permite a `puts()` receber diretamente a string já convertida (linha-13).

Listagem 15: Função que retorna cópia de um ponteiro de entrada.

```
1   #include <stdio.h>
2
3   char* maiuscula(char* str) {
4       char* p;
5       for (p = str; *p != '\0'; p++)
6           if (*p>='a' && *p<='z')
7               *p = *p - 'a' + 'A';
8       return str;
9   }
10
11  int main() {
12      char x[] = "teste de entrada";
13      puts( maiuscula(x) );
14      return 0;
15  }
```

4.3 Ponteiros de Funções

Funções também podem ser referenciadas por ponteiros em C. Um *tipo função* consiste basicamente de seu protótipo sem o nome dos argumentos como ilustra a declaração na linha-7, listagem-16. Neste exemplo **`typedef`** define um tipo-função `T` que se refere a funções cujos argumentos de entrada são dois valores de tipo **`int`** e saída **`int`**. Na linha-10 um ponteiro de `T` recebe o endereço da função `soma()` cujo protótipo casa com o tipo `T` (do contrário ocorreria um erro em tempo de compilação). Observe que não é necessário o uso de `&` prefixado a `soma` na linha-10. Isso ocorre porque, análogo aos vetores, um nome de função sem seus parênteses refere-se ao endereço desta função. Entretanto, curiosamente, a declaração `T *p = &soma` funciona da mesma forma que a da linha-10. A chamada de uma função através de um ponteiro que a refere utiliza a mesma sintaxe de chamada a funções naturalmente utilizando o nome do ponteiro como nome de função (veja a linha-11).

Listagem 16: Ponteiro de função.

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

```
2
3  int soma(int x, int y) {
4      return x+y;
5  }
6
7  typedef int T(int, int);
8
9  int main() {
10     T *p = soma;
11     printf("%d\n", p(11, 85)); //96
12     return 0;
13 }
```

Caso seja necessário um tipo-função-ponteiro, a declaração na linha-7, listagem-16, pode ser substituída por,

```
typedef int (*T)(int, int);
```

Em tais casos, parênteses envolvendo um asterisco e o nome `T` do tipo, são obrigatórios. A declaração na linha-10 deve ser substituída por,

```
T p = soma;
```

Ponteiros de funções são utilizados normalmente em C para passarem funções como argumentos a outras funções. Na prática esta técnica permite a construção de funções de uso mais abrangente, ou seja, que possuem uma implementação capaz de englobar muitas situações deixando em forma de argumentos funções acessórias que tratem de detalhes específicos.

Um exemplo de função abrangente é a *redução* de vetores de inteiros. Seja L um vetor de inteiros e $f(a, b)$ uma função acessório que opera dois números inteiros e retorna outro. Então a redução de L por f equivale a,

1. Processar as duas primeiras chaves de L utilizando f.
2. Tomar o valor obtido no último processamento e, utilizando f, processá-lo com a próxima chave de L ainda não processada.
3. Repetir o processo anterior até todo L ter sido processado.
4. O valor final obtido equivale a redução de L por f.

Note que, na descrição da redução, a função acessório, f, não precisa de especificações além de seu protótipo (recebe dois inteiros e retorna um inteiro).

Listagem 17: Redução de vetores de inteiros

```
1  #include <stdio.h>
2
3  typedef int (*T)(int, int);
4
5  int reduzir(int L[], int n, T fnc) {
6      if (n<2) return 0;
7      int res = fnc(L[0], L[1]), k;
8      for (k=2; k<n; k++)
9          res = fnc(res, L[k]);
10     return res;
11 }
12
13 int soma(int x, int y) {
14     return x+y;
15 }
16
17 int mult(int x, int y) {
18     return x*y;
19 }
20
21 int max(int x, int y) {
22     return x>y ? x : y;
23 }
24
25 int main() {
26     int M[] = {1, 9, 6, 5, 2, 7};
27     int s = reduzir(M, 6, soma);
28     int m = reduzir(M, 6, mult);
29     int v = reduzir(M, 6, max);
30     printf("soma:%d\n", s);
31     printf("multiplicacao:%d\n", m);
32     printf("valor maximo:%d\n", v);
33     return 0;
34 }
```

Na listagem-17 a função `reduzir()` implementa o processo de redução de vetores de inteiros. As funções das linhas 13, 17 e 21 correspondem a funções de protótipos do tipo `T` (linha-3) equivalendo assim a funções-acessórias em reduções. Na função principal é processado o vetor `M` (linha-26) com cada função-acessória respectivamente resultando na soma, produto e valor máximo das chaves de `M`. As chamadas a `reduzir()` ocorrem entre as linhas 27 e 29.

5 Ponteiros de Matrizes

Matrizes em C são de fato vetores cujos elementos são referenciados por mais de um índice (o total destes índices denota o número de dimensões da matriz). Logo seus elementos estão distribuídos contiguamente em memória.

Se um ponteiro apontar para o primeiro elemento de uma matriz então ele poderá ser utilizado para acessar todos os elementos desta matriz na ordem em que se dispõem na memória. Tal ordem de disposição se obtém pelo desenrolar da matriz, ou seja, pelo enfileiramento de linhas da matriz de cima para baixo. Esquematicamente,

$$\begin{pmatrix} \text{linha}_1 \\ \text{linha}_2 \\ \cdots \\ \text{linha}_n \end{pmatrix} \Rightarrow (\text{linha}_1, \text{linha}_2, \cdots \text{linha}_n)$$

A listagem-18 demonstra tal propriedade das matrizes. A matriz `mat`, que é 3×3, é manipulada, através do ponteiro `p`, como um vetor de 9 elementos. A atribuição na linha-8 mostra como o endereço da primeira célula de `mat` é copiado em `p`.

Listagem 18: Acessando uma matriz como um vetor utilizando um ponteiro

```
1  #include <stdio.h>
2
3  int main() {
4      int k;
5      int mat[3][3] = { {1, 3, 2},
6                        {4, 6, 2},
7                        {7, 9, 3} };
8      int* p = &mat[0][0];
9      for (k=0; k<9; k++)
10         printf("%d ", p[k]);
11     // 1 3 2 4 6 2 7 9 3
12     return 0;
13 }
```

Ponteiros de matrizes ajudam a passá-las como argumentos de funções. Porém, quando uma matriz é repassada via ponteiros, naturalmente é perdida a possibilidade de uso dos múltiplos índices que a estrutura original oferece. Isso entretanto pode ser resolvido facilmente como ilustra a função `print_mat()` na listagem 19 cujo objetivo é imprimir a matriz de entrada linha a linha. A função recebe o ponteiro `ptr` que aponta para a primeira célula da matriz e os números de colunas e linhas, respectivamente `ncols` e `nlins`, desta matriz. A expressão `lin*ncols + cols` na linha-10 calcula a posição exata da célula no vetor (`col` denota coluna e `lin` a linha em processamento).

Listagem 19: Tratamento das dimensões de uma matriz passada a uma função através de um ponteiro

```
1  #include <stdio.h>
2
3  void print_mat(int * ptr,
4                 int ncols,
5                 int nlins)
6  {
7      int col, lin;
8      for (lin = 0; lin<nlins; lin++) {
9          for (col=0; col<ncols; col++) {
10             int e = lin*ncols + col;
11             int x = ptr[e];
12             printf("%d ", x);
13         }
14         printf("\n");
15     }
16 }
17
18 int main() {
19     int k;
20     int mat[3][3] = {
21        {1, 3, 2},
22        {4, 6, 2},
23        {7, 9, 3}
24     };
25     print_mat(&mat[0][0], 3, 3);
26     /* 1 3 2
27        4 6 2
28        7 9 3 */
29     return 0;
30 }
```

Ponteiros de ponteiros *não* equivalem às matrizes como os ponteiros a vetores. Apesar disso um ponteiro declarado como `int** M` suporta o desreferenciamento duplo com expressões do tipo `M[1][2]`. Isso é ilustrado na listagem-20. A função `fnc()` utiliza o ponteiro de ponteiro `L` para receber uma lista de strings e imprimi-la como uma matriz de caracteres.

Listagem 20: Ponteiro de ponteiro de **char** como lista de strings.

```
1  #include <stdio.h>
2
```

```
3  void fnc(char** L, int n) {
4      int i, j;
5      for (i = 0; i<n; i++) {
6          for (j = 0; L[i][j]; j++)
7              printf("%c ", L[i][j]);
8          printf("\n");
9      }
10 }
11
12 int main() {
13     char* lista[] =
14       {"bola", "casa",
15        "gato", "vela" };
16     fnc(lista, 4);
17     /*  b o l a
18         c a s a
19         g a t o
20         v e l a  */
21     return 0;
22 }
```

6 Ponteiros e Strings

Em C *não* existe um tipo nativo para representação de strings, as quais podem ser simuladas utilizando-se vetores ou ponteiros de caracteres. Nesta simulação de strings o compilador ainda oferece,

(i) Possibilidade de definição em código utilizando-se aspas duplas. Exemplo,

```
1  char x[] = "hello!";
```

(ii) Caracter terminal nulo, `'\0'`, automaticamente pós-fixado em declarações explicitas. Por exemplo, a declaração anterior equivale à,

```
1  char x[] =
2    {'h','e','l','l','o','!','\0'};
```

A pós fixação automática do caractere nulo exige que, em casos de definição explícita de comprimento do vetor de base da string, seja incluído uma unidade a mais para `'\0'`. Por exemplo, a declaração,

```
1  char s[3] = "abc";
```

gera um erro de compilação pois o comprimento do vetor de base , `3`, é exatamente o mesmo da string e logo não há espaço suficiente para pós fixar o `'\0'`. Remover `3`, neste caso, corrige o problema pois faz o compilador utilizar `4` implicitamente.

Na prática o vetor de base da string recebe comprimento maior que o necessário, como em,

```
1  char a[10] = "abc";
```

O motivo é proporcionar às strings um *comprimento variável*, ou seja, apesar do comprimento fixo do vetor de base, o *comprimento da string* varia em função da presença do caractere nulo. Assim, por exemplo, nesta última declaração, o vetor base, `a`, tem comprimento fixo igual a 10 e as string que ele representa podem ter comprimento entre zero e 9 (lembre-se de `'\0'`).

Listagem 21: Medidor de comprimento de string-C.

```
1  int str_len(char* s) {
2      int len = 0;
3      while (s) {
4          len++;
5          s++;
6      }
7      return len;
8  }
```

A função `str_len()`, listagem-21, mede o comprimento de uma string que é repassada por um `char*`. Note que nenhuma informação referente ao vetor de base, que contém a string, é repassada pois, de fato, basta *buscar* por `'\0'` para encontrar o final da string. Utilizando aritmética de ponteiros (linha-5), o laço `while` processa cada caractere da string de entrada e posteriormente se encerra quando um nulo é encontrado (caracteres não-nulos valem logicamente 1 ao passo que `'\0'` vale 0). O contador `len`, incrementado no laço, equivale, ao final, ao comprimento procurado.

7 Argumentos da Função Principal

Todo programa em C precisa de uma função principal, `main()` que age como gatilho da aplicação, ou seja, nela a execução é iniciada. Opcionalmente `main()` possui dois argumentos, em geral batizados de `argc` e `argv` conforme indica protótipo,

```
1 int main(int argc, char* argv[]);
```

Estes argumentos representam entrada de dados, repassada via linha de comando, antes da execução do programa. O argumento `argc` indica quantos argumentos foram repassados (o que inclui o nome do próprio executável) e `argv` indica a lista destes argumentos. Note que a declaração `char* argv[]`, equivalente a `char** argv`, representa um vetor de ponteiros de caracteres, ou seja, a lista de argumentos é um vetor de strings-C.

Listagem 22: Argumentos da função principal.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      if (argc > 1) {
5          int k;
6          for (k=0; k<argc; k++)
7              printf("%d: %s\n",
8                     k+1,
9                     argv[k]);
10     }
11     return 0;
12 }
```

A listagem-22 ilustra o uso de `argc` e `argv`. Os argumentos de entrada são simplesmente impressos em forma de lista como indica a saída a seguir (a execução foi realizada em Linux utilizando o executável de nome `listar` seguido de três argumentos),

```
$ ./listar gato cachorro cavalo
1: ./listar
2: gato
3: cachorro
4: cavalo
```

8 Problemas com Ponteiros

Ponteiros, em geral, podem causar dois tipos de problemas,

- Vazamento de memória.
- Ponteiros pendurados.

O *vazamento de memória* está ligado a alocação dinâmica de memória e ocorre quando uma variável alocada dinamicamente em heap não tem sua memória devidamente liberada. Na listagem-23 a função `criar_vec()` cria uma nova variável estrutura que possui um campo ponteiro `data` apontando para um vetor dinâmico. A chamada na linha-17 não repassa adiante o controle da variável alocada dinamicamente através de `data` tornando-a inacessível e caracterizando o vazamento de memória.

Listagem 23: Exemplo de vazamento de memória.

```
1  #include <malloc.h>
2
3  typedef struct {
4      int* data;
5      int len;
6  } T;
7
8  T criar_vec(int n) {
9      T r = {
10         (int*) malloc(sizeof(int)*n),
11         n
12     };
13     return r;
14 }
15
16 int main() {
17     criar_vec(10); //vazamento
18     return 0;
19 }
```

Sucessivos vazamentos de memória podem causar lentidão do sistema ou mesmo travamento porque inviabilizam acesso a memória principal a outros aplicativos.

Ponteiros pendurados também estão relacionados a alocação dinâmica em heap. Representam ponteiros cuja memória apontada foi desalocada por chamada explícita a `free()`, mas que mantêm o antigo endereço apontado. Um ponteiro pendurado é perigoso pois aparentemente ainda é operante. Sua utilização normalmente causa comportamentos estranhos da aplicação ou mesmo travamento.

Para prevenir eventuais acidentes com ponteiros pendurados é prática comum fazê-los *apontar para parte alguma*, ou seja, atribuindo-se `NULL` (`stdio.h`) ou zero direta-

mente logo após sua desalocação,

```
...
free(ptr);
ptr = 0;
...
```

9 Exercícios

1. O polinômio de segundo grau é definido por,

$$P(x) = ax^2 + bx + c$$

Onde suas raízes são determinadas por,

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

onde,

$$\Delta = b^2 - 4ac$$

Construa uma função que determine as raízes de um polinômio de segundo grau conforme protótipo,

```
int EQ2G(float C[],
         float *px1,
         float *px2);
```

onde `C` é o vetor dos coeficientes, `{c, b, a}`, e `px1` e `px2` os ponteiros para as variáveis que receberão as raízes, se existirem. Use o seguinte mapa para os valores de retorno da função: 0 para raízes calculadas com sucesso; 1 para polinômio inválido (`a` vale zero); 2 para raízes não reais ($\Delta < 0$).

2. Construa função que receba os vetores de inteiros L e Q e retorne a quantidade de elementos que eles têm em comum. Use o protótipo,

```
int fnc(int L[], int n,
        int Q[], int m);
```

onde `n` e `m` são os comprimentos de `L` e `Q` respectivamente.

3. Construa uma aplicação chamada `operar` que receba argumentos a partir da linha de comando seguindo o formato, `operar <op> <lista>`, onde `<op>` representa uma operação de soma ou produto (use s para soma e p para produto) e `<lista>` uma lista de valores numéricos separados por espaço. No exemplo,

```
$ opererar s 12 32 12
```

são repassados como argumentos a opção de soma e uma lista de três números. A saída, neste exemplo, é 56.

4. Construa funções que recebam um vetor de inteiros e retornem, dentre seus valores,

   (a) O maior.

   (b) Os dois maiores.

   (c) A mediana.

5. Uma função de *troca genérica* é aquela capaz de trocar os dados entre duas variáveis quaisquer de mesmo tipo desde que sejam conhecidos seus respectivos endereços e o tamanho em bytes do tipo base. Implemente a função de troca genérica dado que o protótipo em C é,

```
void Swap(void* p,
          void* q,
          int sise);
```

onde `p` e `q` são os endereços das variáveis cujos dados serão trocados e `size` o tamanho em bytes do tipo base.

6. Utilize a função de troca genérica implementada na questão-5 para trocar conteúdo entre as seguintes variáveis,

   (a) `float x = 78`
   `float y = 123.`

   (b) `int x[] = {1,7,2,4}`
   `int y[] = {-9,5,8,22}.`

   (c) `char x[] = "casa"`
   `char y[] = "mola".`

7. Construa uma função `concat` que concatena duas strings-C, `A` e `B`, passadas como argumento, como no protótipo,

```
1 void concat(char* A,
2              char* B);
```

Deve-se concatenar o conteúdo de `B` em `A` de forma que `A` muda, mas `B`, não.

8. Construir função, `str_cmp()`, que efetue a comparação lexicográfica de duas strings de entrada `A` e `B`. Se `A` for lexicograficamente maior que `B` então a função deverá retornar um inteiro positivo. Se `A` for lexicograficamente menor que `B` então um inteiro negativo deverá ser retornado. Se as strings forem lexicograficamente equivalentes então zero deverá ser retornado. Use o protótipo,

```
1 int str_cmp(char*A, char* B);
```

9. Considere o *mapeamento* de um vetor `M`, por uma função $f(x)$, a operação *in locus* (ou seja, que modifica `M` ao invés de gerar outro vetor ou escalar de saída) que corresponde a aplicar f para cada elemento de `M`. Por exemplo, sendo `M={1,2,3}` e $f(x) = x^2$ então o mapeamento de `M` por f modifica M para `{1,4,9}`. Construa uma função abrangente de mapeamento de vetores de inteiros com protótipo,

```
1 void map(int *M, int n, T f);
```

onde `M` denota o vetor de entrada de comprimento `n` e f a função de mapeamento de tipo `T` com definição,

```
1 typedef int (*T) (int);
```

10. Um *tipo de dados abstrato*, ou *TDA*, é um conjunto de estruturas de dados e funções que trabalham em conjunto para fornecer ao usuário um serviço especializado. Suponha, por exemplo, um TDA que simule vetores de inteiros de capacidade ilimitada. Uma proposta de estrutura de base é dada por,

```
1 typedef struct TVec = {
2     int* data;
3     int n;
4 };
```

onde `TVec` é o tipo que representa o vetor ilimitado, `data` o ponteiro para o vetor real de dados e `n` o comprimento do vetor apontado por `data`. O usuário que define uma variável de tipo `TVec` nunca deverá ou precisará acessar os campos da estrutura. Ao invés disso deve utilizar as funções que compõem o TDA conforme descrições de protótipos a seguir,

- `TVec Criar(int n)`: Cria um novo vetor ilimitado com capacidade inicial igual a `n`.
- `int Comp(T* vec)`: Determina o comprimento do vetor `vec` ilimitado dado como entrada (ponteiro).
- `int Obter(T* vec, int k)`: retorna a chave do vetor `vec` na posição `k`.
- `void Alterar(T* vec, int k, int x)`: Muda para `x` a `k`-ésima chave do vetor `vec`.
- `void Realocar(T* vec, int n)`: Muda comprimento de `vec` para `n`. Os dados devem persistir (exceto aqueles que se perdem por contração de `vec`).
- `void Destruir(T* vec)`: Elimina `vec`.