AULA 16

USO AVANÇADO DE PONTEIROS

Nas aulas 9 a 15 vimos formas importantes de uso de ponteiros: como parâmetros de funções simulando passagem por referência e como elementos da linguagem C que podem acessar indiretamente outros compartimentos de memória, seja uma variável, uma célula de um vetor ou de uma matriz ou ainda um campo de um registro, usando, inclusive uma aritmética específica para tanto.

Nesta aula, baseada nas referências [2, 7], veremos outros usos para ponteiros: como auxiliares na alocação dinâmica de espaços de memória, como ponteiros para funções e como ponteiros para outros ponteiros.

16.1 Ponteiros para ponteiros

Como vimos até aqui, um ponteiro é uma variável cujo conteúdo é um endereço. Dessa forma, podemos acessar o conteúdo de uma posição de memória através de um ponteiro de forma indireta. Temos visto exemplos de ponteiros e suas aplicações em algumas das aulas anteriores e também na seção anterior.

Por outro lado, imagine por um momento que uma variável de um tipo básico qualquer contém um endereço de uma posição de memória que, por sua vez, ela própria também contém um endereço de uma outra posição de memória. Então, essa variável é definida como um **ponteiro para um ponteiro**, isto é, um ponteiro para um compartimento de memória que contém um ponteiro. Ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes, como veremos adiante na seção 16.2. Ponteiros para ponteiros são mais complicados de entender e exigem maturidade para sua manipulação.

O conceito de indireção dupla é então introduzido neste caso: uma variável que contém um endereço de uma posição de memória, isto é, um ponteiro, que, por sua vez, contém um endereço para uma outra posição de memória, ou seja, um outro ponteiro. Certamente, podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla, etc. No entanto, a compreensão de um programa fica gradualmente mais difícil à medida que indireções múltiplas vão sendo utilizadas. Entender bem um programa com ponteiros para ponteiros, ou ponteiros para ponteiros para ponteiros, e assim por diante, é bastante complicado e devemos usar esses recursos de forma criteriosa. Algumas estruturas de dados, porém, exigem que indireções múltiplas sejam usadas e, portanto, é necessário estudá-las.

Considere agora o programa 16.1.

Programa 16.1: Um exemplo de indireção dupla.

```
#include <stdio.h>

int main(void)
{
   int x, y, *pt1, *pt2, **ptpt1, **ptpt2;

   x = 1;
   y = 4;
   printf("x=%d y=%d\n", x, y);
   pt1 = &x;
   pt2 = &y;
   printf("*pt1=%d *pt2=%d\n", *pt1, *pt2);
   ptpt1 = &pt1;
   ptpt2 = &pt2;
   printf("**ptpt1=%d **ptpt2=%d\n", **ptpt1, **ptpt2);

   return 0;
}
```

Inicialmente, o programa 16.1 faz a declaração de seis variáveis do tipo inteiro: `x` e `y`, que armazenam valores desse tipo; `pt1` e `pt2`, que são ponteiros; e `ptpt1` e `ptpt2` que são ponteiros para ponteiros. O símbolo `**` antes do identificador das variáveis `ptpt1` e `ptpt2` significa que as variáveis são ponteiros para ponteiros, ou seja, podem armazenar um endereço onde se encontra um outro endereço onde, por sua vez, encontra-se um valor, um número inteiro nesse caso.

Após essas declarações, o programa segue com atribuições de valores às variáveis `x` e `y` e com atribuições dos endereços das variáveis `x` e `y` para os ponteiros `pt1` e `pt2`, respectivamente, como já vimos em outros exemplos.

Note então que as duas atribuições abaixo:

```
   ptpt1 = &pt1;
   ptpt2 = &pt2;
```

fazem das variáveis `ptpt1` e `ptpt2` ponteiros para ponteiros. Ou seja, `ptpt1` contém o endereço da variável `pt1` e `ptpt2` contém o endereço da variável `pt2`. Por sua vez, a variável `pt1` contém o endereço da variável `x` e a variável `pt2` contém o endereço da variável `y`, o que caracteriza as variáveis `ptpt1` e `ptpt2` declaradas no programa 16.1 como ponteiros para ponteiros.

Observe finalmente que para acessar o conteúdo do endereço apontado pelo endereço apontado por `ptpt1` temos de usar o símbolo de indireção dupla `**`, como pode ser verificado na última chamada à função `printf` do programa.

Veja a figura 16.1 que ilustra indireção dupla no contexto do programa 16.1.

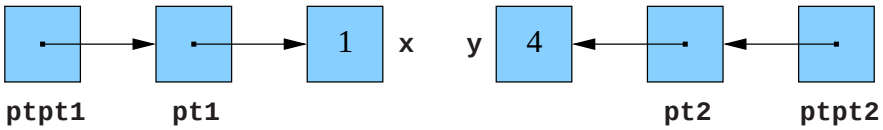

Figura 16.1: Exemplo esquemático de indireção dupla.

16.2 Alocação dinâmica de memória

As estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo. Por exemplo, uma vez que um programa foi compilado, a quantidade de elementos de um vetor ou de uma matriz é fixa. Isso significa que, para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente.

Felizmente, a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa. Usando alocação dinâmica, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa. Apesar de disponível para qualquer tipo de dados, a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas.

Aprendemos até aqui a declarar uma variável composta homogênea especificando um identificador e sua(s) dimensão(ões).

Por exemplo, veja as declarações a seguir:

```
int vet[100];
float mat[40][60];
```

Nesse caso, temos a declaração de um vetor com identificador `vet` e 100 posições de memória que podem armazenar números inteiros e uma matriz com identificador `mat` de 40 linhas e 60 colunas que são compartimentos de memória que podem armazenar números de ponto flutuante. Todos os compartimentos dessas variáveis compostas homogêneas ficam disponíveis para uso durante a execução do programa.

Em diversas aplicações, para que os dados de entrada sejam armazenados em variáveis compostas homogêneas com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões), o que é então solicitado a um(a) usuário(a) do programa logo de início. Por exemplo, o(a) usuário(a) da aplicação pode querer informar a quantidade de elementos que será armazenada no vetor `vet`, um valor que será mantido no programa para verificação do limite de armazenamento e que não deve ultrapassar o limite máximo de 100 elementos. Note que a previsão de limitante máximo deve sempre ser especificada também. Do mesmo modo, o(a) usuário(a) pode querer informar, antes de usar essa estrutura, quantas linhas e quantas colunas da matriz `mat` serão usadas, sem ultrapassar o limite máximo de 40 linhas e 60 colunas. Se o(a) usuário(a), por exemplo, usar apenas 10 compartimentos do vetor `vet` ou apenas 3×3 compartimentos da matriz `mat` durante a execução do programa, os compartimentos restantes não serão usados, embora tenham sido alocados na memória durante suas declarações, ocupando espaço desnecessário na memória do sistema computacional.

Essa alocação que acabamos de descrever, e que conhecemos bem de muito tempo, é chamada de **alocação estática de memória**, o que significa que, no início da execução do programa, quando encontramos uma declaração como essa, ocorre a reserva na memória principal de um número fixo de compartimentos correspondentes ao número especificado na declaração. Esse espaço é fixo e não pode ser alterado durante a execução do programa de forma alguma. Ou seja, não há possibilidade de realocar espaço na memória, nem diminuindo-o nem aumentando-o.

Alocação estática de variáveis e compartimentos não-usados disponíveis na memória podem não ter impacto significativo em programas pequenos, que fazem pouco uso da memória principal, como a grande maioria de nossos programas que desenvolvemos até aqui. No entanto, devemos sempre ter em mente que a memória é um recurso limitado e que em programas maiores e que armazenam muitas informações na memória principal temos, de alguma forma, de usá-la de maneira eficiente, economizando compartimentos sempre que possível. Essa diretriz, infelizmente, não pode ser atingida com uso de alocação estática de compartimentos de memória.

Nesse sentido, se pudéssemos declarar, por exemplo, variáveis compostas homogêneas – vetores e matrizes em particular – com o número exato de compartimentos que serão de fato usados durante a execução do programa, então é evidente que não haveria esse desperdício mencionado.

No exemplo anterior das declarações das variáveis `vet` e `mat`, economizaríamos espaço significativo na memória principal, caso necessitássemos usar apenas 10 compartimentos no vetor `vet` e 9 compartimentos na matriz `mat`, por exemplo, já que ambas foram declaradas com muitos compartimentos mais. No entanto, em uma outra execução subseqüente do mesmo programa que declara essas variáveis, poderiam ser necessárias capacidades diferentes, bem maiores, para ambas as variáveis. Dessa forma, fixar um valor específico baseado na execução do programa torna-se inviável e o que melhor podemos fazer quando usamos alocação estática de memória é prever um limitante máximo para esssas quantidades, conforme vimos fazendo até aqui.

Felizmente para nós, programadores e programadoras da linguagem C, é possível alocar dinamicamente um ou mais blocos de memória na linguagem C. Isso significa que é possível alocar compartimentos de memória durante a execução do programa, com base nas demandas do(a) usuário(a).

Alocação dinâmica de memória significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso naquele momento. Caso haja espaço suficiente disponível na memória principal, a solicitação é atendida e o espaço solicitado fica reservado na memória para aquele uso específico. Caso contrário, isto é, caso não haja possibilidade de atender a solicitação de espaço, uma mensagem de erro em tempo de execução é emitida para o(a) usuário(a) do programa. O(a) programador(a) tem de estar preparado para esse tipo de situação, incluindo testes de verificação de situações como essa no arquivo-fonte, informando também ao(à) usuário(a) do programa sobre a situação de impossibilidade de uso do espaço solicitado de memória. O(a) programador(a) deve solicitar ainda alguma decisão do(a) usuário(a) quando dessa circunstância e, provavelmente, encerrar a execução do programa.

Vejamos um exemplo no programa 16.2 que faz alocação dinâmica de memória para alocação de um vetor.

Programa 16.2: Um exemplo de alocação dinâmica de memória.

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

int main(void)
{
   int i, n, *vetor, *pt;

   printf("Informe a dimensão do vetor: ");
   scanf("%d", &n);

   vetor = (int *) malloc(n * sizeof(int));
   if (vetor != NULL) {
      for (i = 0; i < n; i++) {
         printf("Informe o elemento %d: ", i);
         scanf("%d", (vetor + i));
      }

      printf("\nVetor         : ");
      for (pt = vetor; pt < (vetor + n); pt++)
         printf("%d ", *pt);

      printf("\nVetor invertido: ");
      for (i = n - 1; i >= 0; i--)
         printf("%d ", vetor[i]);
      printf("\n");

      free(vetor);
   }
   else
      printf("Impossível alocar o espaço requisitado\n");

   return 0;
}
```

Na segunda linha do programa 16.2, incluímos o arquivo-cabeçalho `stdlib.h`, que contém a declaração da função `malloc`, usada nesse programa. A função `malloc` tem sua interface apresentada a seguir:

```
void *malloc(size_t tamanho)
```

A função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`. No programa acima, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`. Essa expressão, na verdade, reflete o número de bytes que serão alocados continuamente na memória, que depende do sistema computacional onde o programa é compilado. O operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão. Nesse caso, nas máquinas que usamos no laboratório, a expressão `sizeof(int)` devolve 4 bytes. Esse número é multiplicado por n, o número de compartimentos que desejamos para armazenar números inteiros. O endereço da primeira posição de

memória onde encontram-se esses compartimentos é devolvido pela função `malloc`. Essa função devolve um ponteiro do tipo `void`. Por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro. Por fim, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros. A partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor.

O programa 16.3 é um exemplo de alocação dinâmica de memória de uma matriz de números inteiros. Esse programa é um pouco mais complicado que o programa 16.2, devido ao uso distinto que faz da função `malloc`.

Programa 16.3: Um exemplo de alocação dinâmica de uma matriz.

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

int main(void)
{
   int i, j, m, n, **matriz, **pt;

   printf("Informe a dimensão da matriz: ");
   scanf("%d%d", &m, &n);
   matriz = (int **) malloc(m * sizeof(int *));
   if (matriz == NULL) {
      printf("Não há espaço suficiente na memória\n");
      return 0;
   }

   for (pt = matriz, i = 0; i < m; i++, pt++) {
      *pt = (int *) malloc(n * sizeof(int));
      if (*pt == NULL) {
         printf("Não há espaço suficiente na memória\n");
         return 0;
      }
   }

   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++) {
         printf("Informe o elemento (%d,%d): ", i, j);
         scanf("%d", &matriz[i][j]);
      }
   printf("\nMatriz:\n");
   pt = matriz;
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%d ", *(*(pt+i)+j));
      printf("\n");
   } printf("\n");

   for (i = 0; i < m; i++)
      free(matriz[i]);
   free(matriz);

   return 0;
}
```

Observe por fim que as linhas em que ocorrem a alocação dinâmica de memória no programa 16.3 podem ser substituídas de forma equivalente pelas linhas a seguir:

```
matriz = (int **) malloc(m * sizeof(int *));
for (i = 0; i < m; i++)
   matriz[i] = (int *) malloc(n * sizeof(int));
```

Além da função `malloc` existem duas outras funções para alocação de memória na `stdlib.h`: `calloc` e `realloc`, mas a primeira é mais freqüentemente usada que essas outras duas. Essas funções solicitam blocos de memória de um espaço de armazenamento conhecido também como ***heap*** ou ainda **lista de espaços disponíveis**. A chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo. Pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário. Considere o exemplo a seguir:

```
p = malloc(...);
q = malloc(...);
p = q;
```

Após as duas primeiras sentenças serem executadas, p aponta para um bloco de memória e q aponta para um outro. No entanto, após a atribuição de q para p na última sentença, as duas variáveis apontam para o mesmo bloco de memória, o segundo bloco, sem que nenhuma delas aponte para o primeiro. Além disso, não poderemos mais acessar o primeiro bloco, que ficará perdido na memória, ocupando espaço desnecessário. Esse bloco é chamado de **lixo**.

A função `free` é usada para ajudar os programadores da linguagem C a resolver o problema de geração de lixo na memória durante a execução de programas. Essa função tem a seguinte interface na `stdlib.h`:

```
void free(void *pt)
```

Para usar a função `free` corretamente, devemos lhe passar um ponteiro para um bloco de memória que não mais necessitamos, como fazemos abaixo:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

A chamada à função `free` devolve o bloco de memória apontado por p para o *heap*, que fica disponível para uso em chamadas subseqüentes das funções de alocação de memória.

O argumento da função `free` deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória.

16.3 Ponteiros para funções

Como vimos até este ponto, ponteiros podem conter endereços de variáveis de tipos básicos, de elementos de vetores e matrizes, de campos de registros ou de registros inteiros. Um ponteiro pode também apontar para outro ponteiro. Um ponteiro pode ainda ser usado para alocação dinâmica de memória. No entanto, a linguagem C não requer que ponteiros contenham apenas endereços de dados. É possível, em um programa, ter ponteiros para funções, já que as funções ocupam posições de memória e, por isso, possuem um endereço na memória, assim como todas as variáveis.

Podemos usar ponteiros para funções assim como usamos ponteiros para variáveis. Em particular, passar um ponteiro para uma função como um argumento de outra função é bastante comum em programas da linguagem C.

Suponha que estamos escrevendo a função `integral` que integra uma função matemática `f` entre os pontos `a` e `b`. Gostaríamos de fazer a função `integral` tão geral quanto possível, passando a função `f` como um argumento seu. Isso é possível na linguagem C pela definição de `f` como um ponteiro para uma função. Considerando que queremos integrar funções que têm um parâmetro do tipo `double` e que devolvem um valor do tipo `double`, uma possível interface da função `integral` é apresentada a seguir:

```
double integral(double (*f)(double), double a, double b)
```

Os parênteses em torno de `*f` indicam que `f` é um ponteiro para uma função, não uma função que devolve um ponteiro.

Também é permitido definir `f` como se fosse uma função:

```
double integral(double f(double), double a, double b)
```

Do ponto de vista do compilador, não há diferença alguma entre as interfaces apresentadas acima.

Quando chamamos a função `integral` devemos fornecer um nome de uma função como primeiro argumento. Por exemplo, a chamada a seguir integra a função seno de 0 a $\pi/2$:

```
result = integral(sin, 0.0, PI / 2);
```

O argumento `sin` é o nome/identificador da função seno, que foi incluída no programa através da biblioteca `math.h`.

Observe que não há parênteses após o identificador da função `sin`. Quando o nome de uma função não é seguido por parênteses, o compilador produz um ponteiro para a função em vez de gerar código para uma chamada da função. No exemplo acima, não há uma chamada à função `sin`. Ao invés disso, estamos passando para a função `integral` um ponteiro para

a função `sin`. Podemos pensar em ponteiros para funções como pensamos com ponteiros para vetores e matrizes. Relembrando, se, por exemplo, v é o identificador de um vetor, então v`[i]` representa um elemento do vetor enquanto que v representa um ponteiro para o vetor, ou melhor, para o primeiro elemento do vetor. Da forma similar, se `f` é o identificador de uma função, a linguagem C trata `f(x)` como uma chamada da função, mas trata `f` como um ponteiro para a função.

Dentro do corpo da função `integral` podemos chamar a função apontada por `f` da seguinte forma:

```
y = (*f)(x);
```

Nessa chamada, `*f` representa a função apontada por `f` e `x` é o argumento dessa chamada. Assim, durante a execução da chamada `integral(sin, 0.0, PI / 2)`, cada chamada de `*f` é, na verdade, uma chamada de `sin`.

Também podemos armazenar ponteiros para funções em variáveis ou usá-los como elementos de um vetor, de uma matriz, de um campo de um registro. Podemos ainda escrever funções que devolvem ponteiros para funções. Como um exemplo, declaramos abaixo uma variável que pode armazenar um ponteiro para uma função:

```
void (*ptf)(int);
```

O ponteiro `ptf` pode apontar para qualquer função que tenha um único parâmetro do tipo `int` e que devolva um valor do tipo `void`.

Se `f` é uma função com essas características, podemos fazer `ptf` apontar para `f` da seguinte forma:

```
ptf = f;
```

Observe que não há operador de endereçamento antes de `f`.

Uma vez que `ptf` aponta para `f`, podemos chamar `f` indiretamente através de `ptf` como a seguir:

```
(*ptf)(i);
```

O programa 16.4 imprime uma tabela mostrando os valores das funções seno, cosseno e tangente no intervalo e incremento escolhidos pelo(a) usuário(a). O programa usa as funções `sin`, `cos` e `tan` de `math.h`.

Programa 16.4: Um exemplo de ponteiro para função.

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

/* Recebe um ponteiro para uma função trigonométrica, um inter-
   valo de valores reais e um incremento real e imprime o valor
   (real) da integral da função trigométrica neste intervalo */
void tabela(double (*f)(double), double a, double b, double incr)
{
   int i, num_intervalos;
   double x;

   num_intervalos = ceil((b - a) / incr);
   for (i = 0; i <= num_intervalos; i++) {
      x = a + i * incr;
      printf("%11.6f %11.6f\n", x, (*f)(x));
   }
}

/* Imprime a integral de funções trigonométricas, dados
   um intervalo de números reais e um incremento real */
int main(void)
{
   double inicio, fim, incremento;

   printf("Informe um intervalo [a, b]: ");
   scanf("%lf%lf", &inicio, &fim);
   printf("Informe o incremento: ");
   scanf("%lf", &incremento);

   printf("\n      x         cos(x)"
          "\n    ------    ------\n");
   tabela(cos, inicio, fim, incremento);
   printf("\n      x         sen(x)"
          "\n    ------    ------\n");
   tabela(sin, inicio, fim, incremento);
   printf("\n      x         tan(x)"
          "\n    ------    ------\n");
   tabela(tan, inicio, fim, incremento);

   return 0;
}
```

Exercícios

16.1 Dados dois vetores x e y, ambos com n elementos, $1 \leqslant n \leqslant 100$, determinar o produto escalar desses vetores. Use alocação dinâmica de memória.

16.2 Dizemos que uma seqüência de n elementos, com n par, é **balanceada** se as seguintes somas são todas iguais:

a soma do maior elemento com o menor elemento;
a soma do segundo maior elemento com o segundo menor elemento;
a soma do terceiro maior elemento com o terceiro menor elemento;
e assim por diante ...

Exemplo:

$2\ 12\ 3\ 6\ 16\ 15$ é uma seqüência balanceada, pois $16 + 2 = 15 + 3 = 12 + 6$.

Dados n (n par e $0 \leqslant n \leqslant 100$) e uma seqüência de n números inteiros, verificar se essa seqüência é balanceada. Use alocação dinâmica de memória.

16.3 Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui. Use alocação dinâmica de memória.

16.4 Dada uma matriz de números reais A com m linhas e n colunas, $1 \leqslant m, n \leqslant 100$, e um vetor de números reais v com n elementos, determinar o produto de A por v. Use alocação dinâmica de memória.

16.5 Dizemos que uma matriz quadrada de números inteiros distintos é um **quadrado mágico** se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

Exemplo:

A matriz

$$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

é um quadrado mágico.

Dada uma matriz quadrada de números inteiros $A_{n \times n}$, com $1 \leqslant n \leqslant 100$, verificar se A é um quadrado mágico. Use alocação dinâmica de memória.

16.6 Simule a execução do programa a seguir.

```
#include <stdio.h>

int f1(int (*f)(int))
{
   int n = 0;

   while ((*f)(n))
      n++;
   return n;
}

int f2(int i)
{
   return i * i + i - 12;
}

int main(void)
{
   printf("Resposta: %d\n", f1(f2));
   return 0;
}
```

16.7 Escreva uma função com a seguinte interface:

```
int soma(int (*f)(int), int inicio, int fim)
```

Uma chamada `soma(g, i, j)` deve devolver `g(i) + ... + g(j)`.