Ponteiros

Alocação de Memória

Prof. Edson Alves - UnB/FGA

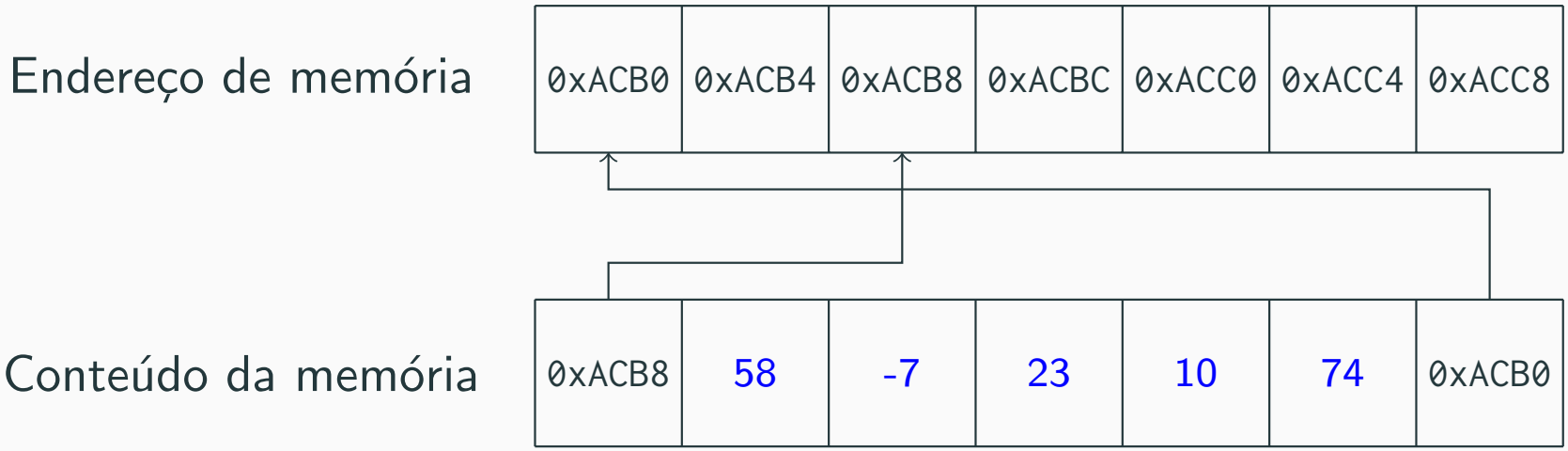
2018

Sumário

Ponteiros

Definição de ponteiros

- Um ponteiro é uma variável que contém um endereço de memória, normalmente a posição de outra variável na memória
- Se uma variável contém o endereço de outra, então diz-se que a primeira variável aponta para a segunda



Sintaxe para ponteiros

Sintaxe para declaração de ponteiros

```
[armazenamento][acesso][modificaor]tipo *nome[=valor_inicial];
```

- Observe que a sintaxe é idêntica à de declaração de variáveis, com exceção do símbolo $*$
- É uma boa prática inicializar um um ponteiro no momento de sua declaração
- O valor 0 (zero) não é um endereço válido, e indica um ponteiro nulo
- Existem dois operadores unários associados à ponteiros:
  1. O operador `&` (endereço de) devolve a posição que seu operando ocupa na memória
  2. O operador $*$ (valor de), quando aplicado a um ponteiro, retorna o valor armazenado no endereço de memória apontado

Exemplo de uso de ponteiros

```c
#include <stdio.h>

int main()
{
    int n = 11;
    int *p = 0;
    int **q = 0;

    p = &n;

    printf("No endereco de memoria %p encontra-se o valor %d\n", p, *p);

    q = &p;

    printf("No endereco de memoria %p encontra-se o valor %p\n", q, *q);

    return 0;
}
```

Aritmética de ponteiros

- As únicas duas operações aritméticas que podem ser feitas entre ponteiros e inteiros são a adição e a subtração
- Porém o valor indicado não é somado diretamente ao endereço: na verdade o ponteiro se movimenta o número de posições indicadas pelo inteiro, de acordo com o tamanho do tipo de dado do ponteiro
- A subtração funciona de forma similar à adição
- Os ponteiros podem ser incrementados ou decrementados, ou podem estar envolvidos em expressões que envolvam somas e subtrações entre um ponteiro e uma variável inteira
- Ao ser incrementado, o endereço apontado pelo ponteiro não é acrescido em uma unidade: o ponteiro é deslocado para o próximo elemento do tipo de dado do ponteiro

Aritmética de ponteiros

- Ao contrário das variáveis, é possível declarar ponteiros do tipo **`void`**
- Ponteiros para o tipo **`void`** armazenam endereços de memória de tipo não especificado, de modo que não é possível utilizar o operador $*$ em tais ponteiros
- Também não é possível realizar aritmética de ponteiros com um ponteiro do tipo **`void`**
- A diferença entre dois ponteiros do mesmo tipo resulta em um inteiro que indica o número de elementos daquele tipo entre os dois ponteiros
- A soma de dois ponteiros não tem sentido prático e não é permitida

Exemplo de aritmética de ponteiros

```c
#include <stdio.h>

int main() {
    int ns[] = {4, 8, 15, 16, 23, 42}, *p = ns, *q = 0;

    printf("Endereços: p = %p, q = %p\n", p, q);

    q = p + 6; p++;

    printf("Valores e endereços: p = %d (%p), q = %d (%p)\n",
        *p, p, *q, q);

    q -= 2;

    printf("Novos valores e endereços: p = %d (%p), q = %d (%p)\n",
        *p, p, *q, q);

    printf("Diferença: %ld\n", q - p);

    return 0;
}
```

Vantagens e desvantagens do uso de ponteiros

Vantagens	Desvantagens
O uso de ponteiros permite a uma função modificar o valor de seus argumentos	Ponteiros não inicializados podem causar uma pane no sistema
Os ponteiros são utilizados para dar suporte às rotinas de alocação dinâmica de memória	Erros no uso de ponteiros são fáceis de cometer e difíceis de se encontrar
O uso de ponteiros pode aumentar a eficiência computacional de algumas rotinas	A gerência da memória fica a cargo do programador

Vetores e Matrizes

Definição de matrizes

- Matrizes são coleções de objetos de um mesmo tipo, referenciadas por um nome comum
- Os elementos de uma matriz são acessados através de índices, que indicam as coordenadas do elemento na matriz
- Matrizes unidimensionais são denominadas vetores
- Em C/C++, diferentemente da matemática, os índices das matrizes começam em zero, não em 1 (um)
- Embora possam ser visualizadas bidimensionalmente, em computadores as matrizes são armazenadas de forma linear, uma linha por vez

Declaração de Matrizes

Sintaxe para declaração de matrizes

```
tipo nome_da_matriz[N_1][N_2]...[N_m]
```

onde N_i é o número de elementos na i-ésima dimensão

- Apenas a primeira dimensão é obrigatória, sendo as demais opcionais
- É possível inicializar um vetor durante sua declaração, deixando a dimensão em aberto (sem preencher) e listando os elementos entre chaves e separados por vírgulas
- Exemplo de inicialização de vetor:

```
int primes[] = { 2, 3, 5, 7, 11, 13, 17 };
```

Acessando os elementos de um vetor

- Os elementos de um vetor podem ser acessados, tanto para leitura quanto para escrita, usando-se o nome do vetor seguido pelo índice do elemento entre colchetes
- O primeiro elemento de um vetor de N tem índice 0 (zero)
- O último elemento tem índice $N - 1$
- Indicar como índice um número inteiro fora do intervalo $[0, N - 1]$ pode resultar, dentre outros, um erro de violação de memória
- O acesso de elementos em uma matriz multidimensional é semelhante: basta indicar, entre colchetes, o índice do elemento em cada uma das dimensões da matriz

Exemplo de uso de matrizes

```c
#include <stdio.h>

#define NUMERO_DE_JOGOS 5

int main()
{
    int tabela[NUMERO_DE_JOGOS][2];
    int SG = 0, GC = 0, GP = 0;
    int V = 0, E = 0, D = 0, PG = 0, i;
    float AP;

    for (i = 0; i < NUMERO_DE_JOGOS; i++) {
        printf("Insira o resultado do jogo %02d: ", i + 1);
        scanf("%d x %d", &tabela[i][0], &tabela[i][1]);
    }

    for (i = 0; i < NUMERO_DE_JOGOS; i++) {
        GP += tabela[i][0];
        GC += tabela[i][1];

```

Exemplo de uso de matrizes

```
21          if (tabela[i][0] > tabela[i][1])
22              V++;
23          else if (tabela[i][0] < tabela[i][1])
24              D++;
25          else
26              E++;
27      }
28
29      SG = GP - GC;
30      PG = V*3 + E;
31      AP = (float) 100*PG / (NUMERO_DE_JOGOS*3);
32
33      printf("\nResumo da campanha: \n");
34      printf("%d vitórias, %d empates, %d derrotas\n", V, E, D);
35
36      printf("Gols: pró = %d, contra = %d, saldo = %d\n", GP, GC, SG);
37
38      printf("Pontos ganhos: %d (%3.2f%% de aproveitamento)\n", PG, AP);
39
40      return 0;
41  }
```

Relação entre ponteiro e vetor

- O nome de um vetor é um ponteiro para o seu primeiro elemento
- Os ponteiros fornecem uma alternativa para o acesso aos elementos de um vetor: se a é um vetor e p um ponteiro para o primeiro elemento a, as expressões `a[i]` e `*(p+i)` são equivalentes
- Em alguns contextos, o acesso via ponteiros pode ser mais rápido que o acesso via índices
- No caso de matrizes, a expressão `a[i][j]` é equivalente à expressão `*(*(p + i) + j)`
- Em uma matriz linearizada, o elemento da posição `(i, j)` seria acessado através da expressão `a[j + i*M]`, onde `M` é igual ao número de colunas da matriz a

Exemplo da relação entre ponteiro e vetor

```cpp
#include <iostream>

using namespace std;

int main() {
    string keywords[] = {"int", "char", "double", "float", "bool",
        "void", "class"};
    string *p = keywords;

    cout << "Endereco: " << p << ", valor: " << *p << endl;
    cout << "Terceiro elemento do vetor: " << *(p + 2) << endl;

    int matrix[][2] = { {1, 2}, {3,4}, {5,6}, {7,8} };
    int *q = &matrix[0][0], (*r)[2] = matrix;

    cout << "Elemento 3x2: via índices = " << matrix[2][1] <<
        ", via ponteiro = " <<  *(*(r + 2) + 1) << ", linearizada = "
        << q[1 + 2*2] << endl;

    return 0;
}
```

Strings

- Em C, as strings são vetores de caracteres terminados com o caractere 0 (zero)
- As operações em strings (atribuição, cópia, comparação, etc) não podem ser feitas diretamente
- A biblioteca `string.h` contém funções para manipulação de strings em C
- A falta do zero terminador é uma causa comum de *bugs* e falhas de segurança
- Em C++, strings são objetos de uma classe, embora guardem a notação de acesso aos seus caracteres idêntica a usada em C
- Uma string C++ pode receber uma string C em uma atribuição
- Para obter uma string C equivalente a uma string C++, basta invocar o método `c_str()`

Exemplo de uso de strings em C

```c
#include <stdio.h>
#include <string.h>

int main()
{
    char s1[] = "UnB - FGA";
    char s2[] = "EDA";

    printf("%s = %lu caracteres\n", s1, strlen(s1));

    printf("Primeira string, na ordem alfabética: %s\n",
        (strcmp(s1, s2) > 0 ? s2 : s1));

    printf("Localização do F: %ld\n", strchr(s1, 'F') - s1);

    printf("Cópia: %s\n", strcpy(s1, s2));

    printf("Concatenação: %s\n", strcat(s1, s2));

    return 0;
}
```

Exemplo de uso de strings em C++

```cpp
#include <iostream>

using namespace std;

int main() {
    string s1 = "UnB - FGA";
    string s2 = "EDA";

    cout << s1 << " = " << s1.size() << " caracteres" << endl;

    cout << "Primeira string, na ordem alfabética: "
        << (s1.compare(s2) > 0 ? s2 : s1) << endl;

    cout << "Localização do F: " << s1.find('F') << endl;

    cout << "Cópia: " << (s1 = s2) << endl;

    cout << "Concatenação: " << s1 + s2 << endl;

    return 0;
}
```

Alocação de dinâmica de memória

Alocação dinâmica de memória

- A alocação dinâmica de memória permite ao programador requerer, em tempo de execução, uma determinada quantidade de memória
- Ela é útil em situações onde não é possível saber, na compilação, a quantidade de variáveis, entradas e ou recursos que um programa irá utilizar durante sua execução
- O uso dela permite a construção de programas mais flexíveis e dinâmicos, sendo utilizada na maioria dos programas escritos em C/C++
- A liberação da memória alocada, contudo, é de responsabilidade do programador
- O uso constante de alocação dinâmica de memória, sem a devida liberação dos recursos, pode levar a erros de execução do programa

Alocação dinâmica de memória em C

- As funções para alocação e liberação de memória em C fazem parte da biblioteca `stdlib.h`
- A principal é a função `malloc()`, que possui a seguinte assinatura:

```
void * malloc(size_t size);
```

- A função `malloc()` retorna, se bem sucedida, um ponteiro para os `size` *bytes* requisitados
- Em caso de falha, ela retorna um ponteiro nulo (`NULL` ou zero)
- Como o retorno é um ponteiro do tipo **`void *`**, ele deve receber uma coerção (`typecast`) para o tipo da variável que receberá o retorno
- Exemplo de coerção:

```
int *p = (int *) malloc(10*sizeof(int));
```

Alocação dinâmica de memória em C

- Finalizado o uso da memória alocada, ela deve ser liberada através da chamada da função `free()`, cuja assinatura é

```
void free(void *ptr);
```

- O parâmetro da função `free()` é o ponteiro que armazenou o retorno da função `malloc()`
- Uma alternativa para `malloc()` é a função `calloc()`, cuja assinatura é

```
void * calloc(size_t nmemb, size_t size);
```

- Os parâmetros da função `calloc()` são o número de elementos `nmemb` a serem alocados e o tamanho `size` de cada elemento
- O retorno é idêntico ao da função `malloc()`
- A diferença entre as funções `malloc()` e `calloc()` é que a segunda inicializa todos os *bytes* da memória alocada com o valor zero, sendo assim mais lenta em tempo de execução

Exemplo de uso de alocação de memória em C

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

int main() {
    int N, i;
    float *notas = NULL, media = 0;

    printf("Insira o número de notas: ");
    scanf("%d", &N);

    if (N < 1) {
        printf("Nada a ser feito!\n");
        return 0;
    }

    notas = (float *) malloc(N*sizeof(float));

    if (!notas) {
        fprintf(stderr, "Sem memória!\n");
        return -1;
    }
```

Exemplo de uso de alocação de memória em C

```
22
23      for (i = 0; i < N; i++)
24      {
25          printf("Insira a nota %d: ", i + 1);
26          scanf("%f", notas + i);
27      }
28
29      for (i = 0; i < N; i++)
30          media += notas[i]/N;
31
32      printf("A média das notas é igual a %.2f\n", media);
33
34      free(notas);
35
36      return 0;
37  }
```

Alocação dinâmica de memória em C++

- A alocação e liberação de memória em C++ são realizadas pelos operadores `new` e `delete`, respectivamente
- O operador `new` tem a seguinte sintaxe:

```
tipo *p = new { tipo | tipo(valor_inicial) | tipo[tamanho] }
```

- O operador `new` retorna, se bem sucedido, um ponteiro para o elemento ou vetor de elementos do tipo requisitado
- Em caso de falha, ele retorna uma exceção do tipo `std::bad_alloc`, que está definida na biblioteca **`new`**
- Ao contrário do que acontece em C, não é preciso fazer uma coerção (*typecast*) no retorno do operador `new`. Ex:

```
int *p = new int[10];
```

Alocação dinâmica de memória em C++

- Finalizado o uso da memória alocada, ela deve ser liberada através da chamada do operador `delete`, cuja sintaxe é

```
delete p;
delete [] array;
```

- Se um único elemento foi alocado, ele deve ser desalocado com o operador **`delete`**
- caso um vetor tenha sido alocado, deve ser invocado o operador **`delete`** `[]` para a liberação da memória
- Os operadores **`new`** e **`delete`** podem ser sobrescritos para operar da maneira desejada pelo programador
- Estes operadores estão incorporados na linguagem, não sendo necessária a inclusão de nenhuma biblioteca para seu uso

Exemplo de uso de alocação dinâmica em C++

```
1  #ifndef ITEM_H
2  #define ITEM_H
3
4  #include <string>
5
6  using namespace std;
7
8  class Item {
9  public:
10     virtual ~Item() {}
11
12     virtual string nome() const = 0;
13     virtual float preco() const = 0;
14 };
15
16 #endif
```

Exemplo de uso de alocação dinâmica em C++

```
1  #ifndef PASTEL_H
2  #define PASTEL_H
3
4  #include "item.h"
5
6  class Pastel : public Item {
7  public:
8      Pastel(const string& sabor);
9
10     string nome() const;
11     float preco() const;
12
13 private:
14     static const float _preco;
15     string _sabor;
16 };
17
18 #endif
```

Exemplo de uso de alocação dinâmica em C++

```
1  #include "pastel.h"
2
3  const float Pastel::_preco = 2.50f;
4
5  Pastel::Pastel(const string& sabor) : _sabor(sabor)
6  {
7  }
8
9  string
10 Pastel::nome() const
11 {
12     return "Pastel de " + _sabor;
13 }
14
15 float
16 Pastel::preco() const
17 {
18     return _preco;
19 }
```

Exemplo de uso de alocação dinâmica em C++

```
1  #ifndef CALDO_DE_CANA_H
2  #define CALDO_DE_CANA_H
3
4  #include "item.h"
5
6  enum Quantidade {COPO_200_ML, COPO_300_ML, COPO_500_ML, LITRO};
7
8  class CaldoDeCana : public Item {
9  public:
10     CaldoDeCana(Quantidade qtd = COPO_300_ML);
11
12     string nome() const;
13     float preco() const;
14     Quantidade qtd() const;
15
16 private:
17     Quantidade _qtd;
18 };
19
20 #endif
```

Exemplo de uso de alocação dinâmica em C++

```cpp
#include "caldodecana.h"

CaldoDeCana::CaldoDeCana(Quantidade qtd) : _qtd(qtd) { }

string CaldoDeCana::nome() const {
    string descricao = "Caldo de cana ";

    switch (_qtd) {
    case COPO_200_ML:
        return descricao + "200 ml";

    case COPO_300_ML:
        return descricao + "300 ml";

    case COPO_500_ML:
        return descricao + "500 ml";

    default:
        return descricao + "jarra 1L";
    }
}
```

Exemplo de uso de alocação dinâmica em C++

```
22
23 float CaldoDeCana::preco() const {
24
25     switch (_qtd) {
26     case COPO_200_ML:
27         return 1.5f;
28
29     case COPO_300_ML:
30         return 2.0f;
31
32     case COPO_500_ML:
33         return 3.0f;
34
35     default:
36         return 5.0f;
37     }
38 }
39
40 Quantidade CaldoDeCana::qtd() const {
41     return _qtd;
42 }
```

Exemplo de uso de alocação dinâmica em C++

```cpp
#include <iostream>
#include <vector>
#include "pastel.h"
#include "caldodecana.h"

using namespace std;

typedef struct {
    int codigo;
    string nome;
    Quantidade qtd;
} Tabela;

static const Tabela menu[] = {
    {1, "Pastel"},
    {2, "Caldo de cana"},
    {3, "Finalizar pedido"},
    {0, "NULL"}
};

```

Exemplo de uso de alocação dinâmica em C++

```
21 static const Tabela tabela_pastel[] = {
22     {1, "Carne"},
23     {2, "Queijo"},
24     {3, "Presunto"},
25     {4, "Palmito"},
26     {5, "Retornar"},
27     {0, "NULL"}
28 };
29
30 static const Tabela tabela_caldo[] = {
31     {1, "Copo 200 ml", COPO_200_ML},
32     {2, "Copo 300 ml", COPO_300_ML},
33     {3, "Copo 500 ml", COPO_500_ML},
34     {4, "Jarra 1L", LITRO},
35     {5, "Retornar"},
36     {0, "NULL"}
37 };
38
39 void imprime_tabela(const Tabela tabela[]) {
40     for (int i = 0; tabela[i].codigo; i++)
41         cout << tabela[i].codigo << ". " << tabela[i].nome << endl;
42 }
```

Exemplo de uso de alocação dinâmica em C++

```
44 int main() {
45     vector<Item *> pedido;
46     int opcao, tipo;
47     bool encerrar = false;
48 
49     do {
50         do {
51             cout << endl << "O que deseja pedir?" << endl;
52             imprime_tabela(menu);
53             cout << "Digite sua opcao: ";
54             cin >> opcao;
55         } while (opcao < 0 && opcao > 3);
56 
57         switch (opcao) {
58         case 1:
59             do {
60                 cout << endl << "Qual sabor?" << endl;
61                 imprime_tabela(tabela_pastel);
62                 cout << "Digite sua opcao: ";
63                 cin >> tipo;
64             } while (tipo < 0 && tipo > 5);
```

Exemplo de uso de alocação dinâmica em C++

```
66              if (tipo != 5)
67                  pedido.push_back(new Pastel(tabela_pastel[tipo-1].nome));
68          break;
69
70          case 2:
71              do {
72                  cout << endl << "Qual " "quantidade?" << endl;
73                  imprime_tabela(tabela_caldo);
74                  cout << "Digite sua opcao: ";
75                  cin >> tipo;
76              } while (tipo < 0 && tipo > 5);
77
78              if (tipo != 5)
79                  pedido.push_back(new CaldoDeCana(tabela_caldo
80                      [tipo - 1].qtd));
81          break;
82
83          case 3:
84              encerrar = true;
85          }
86      } while (!encerrar);
```

Exemplo de uso de alocação dinâmica em C++

```
88      if (pedido.size() < 1) {
89          cout << endl << "Nenhum pedido" << endl;
90      } else {
91          float valor = 0;
92
93          cout << endl << "Seu pedido: " << endl;
94          cout.precision(2);
95
96          for (size_t i = 0; i < pedido.size(); i++) {
97              cout << pedido[i]->nome() << ": R$ "
98                  << fixed << pedido[i]->preco() << endl;
99              valor += pedido[i]->preco();
100         }
101
102         cout << endl << "Total: R$ " << fixed << valor << endl;
103     }
104
105     for (size_t i = 0; i < pedido.size(); i++) delete pedido[i];
106
107     return 0;
108 }
```

Referências

1. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
2. **SCHILDT**, Herbert. *C Completo e Total*, 1997.
3. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.
4. C++ Reference[1].

[1] https://en.cppreference.com/w/