Alocação dinâmica de memória

Linguagem de programação
Prof. Allan Rodrigo Leite

Variáveis primitivas e compostas

- Variáveis comuns
  - Permitem armazenar um único dado
    - Usam tipos de dados primitivos
  - Possuem um endereço de memória predefinido

- Variáveis compostas
  - Permitem armazenar mais de um dado
    - Vetores (homogêneas)
    - Estruturas (heterogêneas)
  - Possuem um endereço de memória predefinido
    - Espaço de memória contínuo para armazenar o conjunto de dados

Variáveis primitivas e compostas

- Vetores
  - O acesso a cada elemento é realizado a partir de um índice
    - Se um índice inválido for acessado, é invadida outra posição na memória
    - Os índices apontam para um dado em memória que pertence ao vetor
  - A linguagem C não possui tratamentos em relação à índices inválidos
    - Em outras linguagens, como Java, este tratamento é implícito na tecnologia

```c
int main() {
    int arr[] = {1,2,3,4,5};

    printf("arr[0]: %d\n", arr[0]);
    printf("arr[9]: %d\n", arr[9]); //índice 9 está fora dos limites do vetor

    return 0;
}
```

Ponteiros

- Ponteiros
  - Tipo de dado cujo valor aponta para uma outra área da memória
  - O valor de um ponteiro se refere à um endereço de memória
    - Também é chamado de tipo de referência

- Propósito
  - Úteis para estruturas de dados que não são capazes de serem alocadas continuamente em memória
    - Devido ao tamanho ou necessidade de alocação dinâmica do dado
  - Passagem de parâmetros para funções como referência

Ponteiros

- Operadores unários &, * e **
  - Operador &
    - Endereço de ...
    - Utilizado para retornar o endereço de memória de uma variável
  - Operador *
    - Conteúdo de ...
    - Utilizado para indicar o endereço de memória apontado
  - Operador **
    - Ponteiro de ponteiro
    - Representação de vetores unidimensionais de ponteiros

Ponteiros

- Operadores unários &, * e ** (cont.)

```c
int main() {
    int a = 10;
    int *b = &a;

    printf("Endereco de a %p\n",&a);
    printf("Endereco de b %p\n",b);
    printf("Valor de a: %d\n", a);
    printf("Valor de b: %d\n", *b);

    a = 20;
    printf("Valor de a: %d\n", a);
    printf("Valor de b: %d\n", *b);

    *b = 30;
    printf("Valor de a: %d\n", a);
    printf("Valor de b: %d\n", *b);
}
```

Declarando ponteiros

- Declaração de ponteiros
  - É preciso indicar o tipo de dados a ser utilizado endereço do ponteiro
  - O tipo `void` pode ser utilizado para indicar qualquer tipo de dados
    - O uso de `void*` requer conversões explícitas para o tipo em questão

```
<tipo de dado>*
int *var; //ponteiro cujo conteúdo a ser armazenado no endereço de memória é do tipo int

<tipo de dado>**
int **var; //ponteiro de ponteiro (vetor de ponteiros do tipo int)
```

- Vetores são essencialmente ponteiros
  - O acesso aos índices é chamado de aritmética de ponteiros

Declarando ponteiros

- Aritmética de ponteiros

```c
int main() {
    int v[5] = { 1, 2, 3, 4, 5 };

    printf("%p\n", v);
    printf("%d\n", v[0]);
    printf("%d\n", *v);
    printf("%p\n", v + 3);
    printf("%d\n", *(v + 3));
}
```

Declarando ponteiros

- Aritmética de ponteiros (cont.)

```c
int main() {
    int v[5] = { 1, 2, 3, 4, 5 };

    printf("%p\n", v);          //endereço de v
    printf("%d\n", v[0]);       //valor do primeiro elemento de v (valor 1)
    printf("%d\n", *v);         //valor do primeiro elemento de v (valor 1)
    printf("%p\n", v + 3);      //endereço do quarto elemento de v
    printf("%d\n", *(v + 3));   //valor no quarto elemento de v (valor 4)
}
```

Declarando ponteiros

- Aritmética de ponteiros (cont.)
  - Exemplos
    - `*(v + 0)` ≡ `v[0]`: aponta para o primeiro elemento do vetor
    - `*(v + 1)` ≡ `v[1]`: aponta para o segundo elemento do vetor
    - `*(v + 4)` ≡ `v[4]`: aponta para o último elemento do vetor
  - Ponteiros e endereço de memória
    - `&v[i]` é equivalente a `(v+i)`
    - `v[i]` é equivalente a `*(v+i)`
  - Esta aritmética é válida devido ao armazenamento em memória
    - O armazenamento ocorre de forma contínua na memória

Alocação dinâmica de memória

- Uso de memória em um programa desenvolvido em C
  - É possível reservar espaços de memória em tempo de execução

- Maneiras para manipular espaços de memória
  - Variáveis globais e estáticas
    - Espaço reservado em memória existe ao longo da execução do programa
  - Variáveis locais
    - Espaço em memória existe apenas enquanto a função está em execução
    - Quando a execução da função termina, o epaço é liberado
    - Portanto, o espaço é mantido enquanto o escopo da variável existir
  - Reservar memória em tempo de execução
    - Solicita-se ao sistema operacional um espaço de um dado tamanho

Alocação dinâmica de memória

- Memória estática
  - Armazena instruções do programa, variáveis globais e estáticas
  - É possível definir o tamanho da memória estática antes de executar o programa
- Memória dinâmica
  - Armazena variáveis locais, pilhas de execução e memória alocada dinamicamente
  - Tamanho da memória pode variar de acordo com o fluxo de execução do programa
  - A memória livre é utilizada para alocação dinâmica
    - Pode crescer ou diminuir a partir das funções `malloc`, `realloc` ou `free`

Alocação dinâmica de memória

- Observações sobre memória dinâmica
  - O espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa
  - Por isto, é possível alocar dinamicamente um espaço de memória em uma função e acessá-lo em outra
  - A partir do momento que libera-se o espaço, ele estará disponibilizado para outros usos e não é possível acessá-lo mais
  - Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar

Alocação dinâmica de memória

- Função `sizeof(size_t n)`
  - Retorna o número de bytes requeridos por um tipo de dado

```
int t = sizeof(float); //retorna 4
```

- Função `malloc(size_t n)`
  - Aloca dinamicamente um espaço com n bytes

```
int *a = (int*) malloc(sizeof(int)); //aloca dinamicamente 4 bytes (int)
*a = 10;
```

Alocação dinâmica de memória

- Função `malloc(size_t n)` (cont.)
  - Verificando se foi possível alocar a memória requerida

```c
int *a = (int*) malloc(sizeof(int)); //aloca dinamicamente 4 bytes (int)

if (a == NULL) {
    printf("Nao foi possivel alocar memoria");
}
```

- Criando vetores dinâmicos

```c
int *a = (int*) malloc(3 * sizeof(int)); //vetor de 3 índices com 4 bytes em cada

a[0] = 10;
a[1] = 20;
a[2] = 30;
```

Alocação dinâmica de memória

- Função `calloc(size_t n)`
  - Aloca n espaços contínuos de memória com size tamanho
  - O `calloc` inicializa a memória alocada com zero
  - O `malloc` não realiza nenhum tipo de inicialização

- Criando vetores dinâmicos

```
int *a = (int*) calloc(3, sizeof(int)); //vetor de 3 índices com 4 bytes em cada

a[0] = 10;
a[1] = 20;
a[2] = 30;
```

Alocação dinâmica de memória

- Função `relloc(void *p, size_t n)`
  - Funcionamento
    - Recebe o endereço de memória previamente alocado
    - Aloca um novo espaço de memória de acordo com o tamanho informado
    - Copia o conteúdo da memória original e desaloca esta memória
    - Devolve o endereço do novo bloco

- Redimensionamento vetores dinâmicos

```c
int *a = (int*) calloc(2, sizeof(int)); //vetor de 2 índices com 4 bytes em cada

a[0] = 10;
a[1] = 20;

a = (int*) realloc(a,sizeof(int) * 10); //redimenciona vetor com 10 índices
```

Alocação dinâmica de memória

- Função `free(void *p)`
  - Desaloca a memória ocupada por um ponteiro

```c
int *a = (int*) calloc(3, sizeof(int)); //vetor de 3 índices com 4 bytes em cada

a[0] = 10;
a[1] = 20;
a[2] = 30;

free(a);
```

Alocação dinâmica de memória

- Funções `malloc`, `calloc` e `realloc`
  - O ponteiro retornado deve prever o tipo de ponteiro esperado
    - Para tal, o tipo de ponteiro retornado deve ser convertido explicitamente
  - Biblioteca `stdlib.h` possui estas funções para alocação de memória
    - Declaração das funções `malloc`, `calloc` e `free`
    - Declaração do tipo `NULL`
  - O `calloc` inicializa a memória alocada com zero
  - O `malloc` não realiza nenhum tipo de inicialização

Exercícios

1. Faça um programa que reproduza o comportamento da função `realloc`. O programa deve ler um conjunto de valores inteiros, armazená-los em um vetor alocado dinamicamente e, em seguida, realocar um novo espaço de memória que comporte o dobro de elementos, movendo todo conteúdo para este novo espaço de memória.

2. Faça um programa que encontre o maior e menor inteiro dentro de um vetor inteiros alocado dinamicamente. Em seguida, deve ser exibido o maior e menor inteiro e a soma dos dois. Todas as operações de manipulação do vetor deve ser realizado a partir da aritmética de ponteiros.

3. Faça um programa que leia um conjunto de números inteiros e armazene-os em um vetor, o qual representa um conjunto de dados. Em seguida, o programa deve ler novamente um novo conjunto de números reais com o mesmo tamanho do conjunto anterior e o armazene em um novo vetor, o qual representa um conjunto de pesos. Por fim, o programa deve retornar a média aritmética ponderada do conjunto de valores. Todos os vetores devem ser alocados dinamicamente.

Exercícios

4. Faça um programa que leia um conjunto de números inteiros e armazene-os em um vetor que representa este conjunto de dados. Em seguida, o programa deve retornar a média, desvio padrão e variância dos valores do vetor. O vetor deve ser alocado dinamicamente e manipulado por meio da aritmética de ponteiros.

5. Faça uma nova versão do programa anterior para exibir as seguintes informações a partir dos valores do vetor: i) mediana; ii) moda; iii) outliers (usando o método Z score); e iv) agrupamento dos valores em primeiro, segundo e terceiro quartil. O vetor deve ser alocado dinamicamente e manipulado por meio da aritmética de ponteiros.

Alocação dinâmica de memória

Linguagem de programação
Prof. Allan Rodrigo Leite