Aula 7 - Gerenciando dados e memória

Docupedia Export

Author:Balem Luis (CtP/ETS)
Date:04-Apr-2023 17:49

Table of Contents

- Heap, Stack e ponteiros
- Tipos por Referência e Tipos por valor
- Class vs Struct
- Null, Nullable e propagação nula
- Associação, Agregação e Composição
- Garbage Collector
- Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

1 Heap, Stack e ponteiros

Todo programa C# pode guardar seus dados em 2 lugares: O Heap e o Stack. O Heap é uma área na memória onde dados ficam armazenados fora de ordem, e são referenciados para uso. Isso significa que para acessar algo no Heap nós precisamos de algo chamado ponteiro. O ponteiro é uma variável que armazena não o valor, mas sim o endereço de onde algo está no Heap. Pode não parecer, mas ao criarmos um objeto de uma classe estamos usando um ponteiro escondido.
Por outro lado, alguns valores podem ser armazenados na Stack. A stack é uma região onde os valores são colocados sequencialmente. Quando criamos a variável aluno, os dados do objeto Aluno foram colocados no Heap, porém a referência foi colocada na stack. Ao usarmos a variável acessamos a Stack, buscamos a referência e assim vamos buscar os dados na Stack. Não só isso, vários dados como int's e outros tipos primitivos são armazenados na Stack. Quando chamamos uma função, o ponto de retorno, ou seja, para onde a função deve retornar depois de ser executada fica guardada na Stack. E quando chamamos muitas funções a Stack pode crescer tanto que invade a área do Heap resultado no famoso erro Stack Overflow.

2 Tipos por Referência e Tipos por valor

No C# existem tipos que são por referência e tipos que são por valor. Isso significa que ao chamar uma função ou fazer qualquer atribuição, os dados são copiados se for um tipo por valor, e caso for um tipo por referência, uma referência é armazenada na nova variável para os mesmos dados. Observe o exemplo a seguir que demonstra essa dinâmica:

```
1  using static System.Console;
2
3  int idade = 16;
4  int idade2 = idade;
5  idade2++;
6  WriteLine(idade); // 16
7  WriteLine(idade2); // 17
8
9  Aluno aluno = new Aluno();
10 aluno.Idade = 16;
11 Aluno aluno2 = aluno;
12 aluno2.Idade++;
13 WriteLine(aluno.Idade); // 17
14 WriteLine(aluno2.Idade); // 17
15
16 Aluno aluno = new Aluno();
17 aluno.Idade = 16;
18 Aluno aluno2 = new Aluno();
19 aluno2.Idade = aluno.Idade;
20 aluno2.Idade++;
21 WriteLine(aluno.Idade); // 16
22 WriteLine(aluno2.Idade); // 17
23
24 public class Aluno
25 {
26     public string Nome { get; set; }
27     public int Idade { get; set; }
28 }
```

Podemos observar que ao passar o valor 16 de uma variável int para outra, esse valor é copiado. Ao alterar o 'idade2', o 'idade' não é alterado pois a variável 'idade2' tem só uma mera cópia do seu valor. No segundo exemplo, criamos um objeto do tipo Aluno que é por referência, ao escrevermos 'aluno2 = aluno', estamos copiando a referência que a variável 'aluno' tem na Stack para a variável 'aluno2'. Já no terceiro exemplo, os dados de idade que estão no Heap são copiados de um objeto para outro, mas dois objetos diferentes são criados.

Ao chamarmos uma função, fenômenos semelhantes acontecem:

```
1  int numero = 76;
2  AlterarNumero(numero);
3  // Aqui número vale 76, seu valor foi copiado e não foi alterado
4
5  Aluno aluno = new Aluno();
6  aluno.Nome = "Pamella";
7  AlterarNome(aluno);
8  // Aqui o nome do Aluno é Gilmar, foi passado a referência do objeto que foi alterado dentro da função
9
10 void AlterarNumero(int valor)
11 {
12     valor = valor + 1;
13 }
14
15 void AlterarNome(Aluno aluno)
16 {
17     aluno.Nome = "Gilmar";
18 }
19
20 public class Aluno
21 {
22     public string Nome { get; set; }
23     public int Idade { get; set; }
24 }
```

3 Class vs Struct

Neste sentido o C# possibilita a utilização de duas estruturas diferentes: a Classe e a Estrutura. Um struct é extremamente parecido com uma classe, sua utilização é semelhante, porém, enquanto a classe cria um tipo por referência, o struct cria um tipo por valor. Você pode ter ganhos de desempenho ao usar struct pois acessar/remover dados da Stack é muito mais rápido do que no Heap. Abaixo um exemplo de uso de struct:

```
1  Aluno aluno = new Aluno("Gilmar", 25);
2
3  public struct Aluno
4  {
5      public Aluno(string nome, int idade)
6      {
7          this.Nome = nome;
8          this.Idade = idade;
9      }
10
11     public string Nome { get; set; }
12     public int Idade { get; set; }
13 }
```

Podemos ver que o uso é idêntico a classe, só trocamos a palavra reservada e pronto: Temos um tipo por valor. Neste ponto fica cada vez mais evidente que int, byte, long, float, bool, char, entre outros tipos, são structs quando analisados por baixo dos panos.
Todos os vetores, incluindo string que é um vetor de caracteres são tipos por referências e são armazenados no Heap.
Lembrando que, tipos por valor que estiverem dentro de objetos por refência estarão no Heap, pois estarão dentro de coisas que ficam dentro do Heap.

4 Null, Nullable e propagação nula

Para tipos por referência, você pode ainda criar ponteiros que não apontam para nada. Isso é útil quando queremos criar variáveis que ainda não tem valor algum. Para isso usamos a palavra reservada 'null'.

```
1  Aluno aluno = null;
2
3  var nome = aluno.Nome; // Erro, aluno é nulo e não podemos ler seu nome, o famoso NullPointerException (erro /exceção de
   ponteiro nulo)
```

Isso ajuda a inicializar objetos mas também trará uma dor de cabeça (inevitável) ao usar ponteiros que não foram ainda atribuidos com referências. Ainda podemos criar tipos nulos a partir de tipos por valor. Abaixo uma implementação de como isso seria feito:

```
1  //int j = null; // Erro de compilação: int é um tipo por valor e não pode receber nulo
2  int? i = null; // OK
3
4  // Testa para ver se i é nulo, caso seja, atribui um valor
5  if (!i.HasValue)
6      i = 76;
7
8  int valorReal = i.Value; // Da erro se i for nulo
```

Ainda que sejam ferramentas úteis, por vezes (em várias linguagens) os nulos trazem grande dor de cabeça. Observe o exemplo a seguir:

```
1  Aluno aluno = null;
2  string nome = aluno.Nome; // NullPointerException
3
4  public struct Aluno
5  {
6      public string Nome { get; set; }
7      public int Idade { get; set; }
8  }
```

Para tratar isso C# trás uma brilhante feature que é a propagação nula:

```
1  Aluno aluno = null;
2  string nome = aluno?.Nome; // Se aluno for nulo, não temos um erro, mas .Nome retornará nulo, automaticamente.
3
4  public struct Aluno
5  {
```

```
    public string Nome { get; set; }
    public int Idade { get; set; }
}
```

Além disso, temos ainda outra tratativa para valores nulos que é bem interessante:

```

Aluno aluno = null;
string nome = aluno?.Nome ?? "Pamella"; // Se aluno?.Nome resultar em nulo, temos o valor padrão "Pamella".

public struct Aluno
{
    public string Nome { get; set; }
    public int Idade { get; set; }
}
```

5 Associação, Agregação e Composição

Além dos tipos que usamos para Campos e Propriedades de uma classe, também podemos usar um ponteiro a outros tipos. O nome disso é Associação, quando dois tipos estão ligados por uma referência. Ainda temos a Agregação e Composição que essencialmente são a mesma coisa mas com formas de utilização diferente. Enquanto a Associação é fraca e objetos sabem que outros existem mas são independentes, uma Agregação ocorre quando um objeto B está referenciado por um objeto A e o segundo não faz sentido sozinho como objeto sem o A. Uma Composição é quando A não faz sentido sem o B. Abaixo um exemplo de Agregação: A data não faz sentido se não tiver um significado associado ao cliente, mas o cliente continua a ser um cliente mesmo sem data de aniversário.

```
1  Cliente cliente = new Cliente();
2  cliente.Nome = "Pamella";
3
4  Data data = new Data();
5  data.Dia = 7;
6  data.Mes = 6;
7  data.Ano = 2000;
8
9  cliente.DataNascimento = data;
10
11 public class Data
12 {
13     public int Dia { get; set; }
14     public int Mes { get; set; }
15     public int Ano { get; set; }
16 }
17
18 public class Cliente
19 {
20     public string Nome { get set; }
21     public Data DataNascimento { get; set; }
22 }
```

O importante desta discussão é apenas perceber que é possível usar seus tipos como variáveis dentro de outras classes e fazer estruturas muito complexas. Note que quando fazemos isso com classes estamos falando de ponteiros. Quando usamos estruturas estamos falando dos dados em si. Isso significa que se eu fizer uma struct A com uma propriedade do tipo A estaria colocando um loop infinito de dados, pois A está dentro do A que tem um A dentro e assim por diante. Isso resulta em um erro de compilação.

6 Garbage Collector

Para finalizar estes tópicos avançados sobre memória e ponteiros, é importante mencionar o Garbage Collector, Coletor de Lixo ou simplesmente GC. O GC é basicamente um subsistema do .NET que tem a missão de limpar dados não utilizados ao longo do tempo.
Em muitas linguagens você faz o gerenciamento de memória. Isso significa que se você criar dados dinâmicos, caso você esqueça de apagá-los, eles ficarão para sempre na sua memória até que a aplicação seja finalizada. Isso é o que chamamos de Memory Leak e que faz com que algumas aplicações fiquem muito mais pesadas do que deveriam após intenso uso.
O GC automatiza isso para você. Todas as informações gerenciadas, ou seja, que ficam no Heap são então gerenciadas pelo GC e quando não existem mais ponteiros para esses dados, eles são automaticamente limpos.
É claro que isso gera alguns efeitos colaterais. Abusar do Heap pode deixar sua aplicação muito mais lenta e criar gargalos por muitos motivos. Um deles é a desfragmentação, que é o que ocorre quando o GC move os dados para tirar buracos que aparecem quando dados são limpos no meio da memória. É um processo pesado, mas necessário para otimizar seu uso.
Isso nos torna ainda mais fãs dos dados não gerenciados, que podemos usar utilizando tipos por valor em funções que ficarão na Stack, que não é gerenciada pelo GC.

7 Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

```
1  using static System.Console;
2
3  LinkedList clientes = new LinkedList();
4
5  Cliente cliente1 = new Cliente();
6  cliente1.Nome = "Gilmar";
7  clientes.Add(cliente1);
8
9  Cliente cliente2 = new Cliente();
10 cliente2.Nome = "Pamella";
11 clientes.Add(cliente2);
12
13 // Se Get retornar null, Nome deve retornar null ao invés de estourar um erro, se Nome retornar null deve-se substituir
   pelo valor padrão
14 // 'Cliente não encontrado'
15 var result = clientes.Get(1)?.Nome ?? "Cliente não encontrado";
16 WriteLine(result);
17
18 // Cadastrando uma quantidade variável de clientes
19 string input = ReadLine();
20 while (input != "")
21 {
22     Cliente newClient = new Cliente();
23     newClient.Nome = input;
24     newClient.Cpf = long.Parse(ReadLine());
25     clientes.Add(newClient);
26
27     input = ReadLine();
28 }
29
30 WriteLine("Procure um  cpf de cliente pelo nome:");
31 input = ReadLine();
32 for (int i = 0; i < clientes.Count; i++)
33 {
```

```
34      var pesquisa = clientes.Get(i);
35      if (pesquisa?.Nome == input)
36      {
37          WriteLine(pesquisa.Cpf);
38          break;
39      }
40  }
41
42  // Classe cliente a qual queremos armazenas
43  public struct Cliente
44  {
45      public string Nome { get; set; }
46      public long Cpf { get; set; }
47  }
48
49  // Um nó representa um valor na lista com um ponteiro para o próximo valor
50  public class LinkedListNode
51  {
52      public Cliente Value { get; set; }
53      public Node Next { get; set; }
54  }
55
56  public class LinkedList
57  {
58      // Ponteiro vazio = lista vazia
59      private LinkedListNode first = null;
60
61      public int Count { get; private set; }
62
63      // Função para adicionar um valor no final da lista
64      public void Add(Cliente value)
65      {
66          Count++;
67
68          // Se first for nulo, vamos inicializá-lo com um novo elemento
69          if (first == null)
70          {
71              first = new LinkedListNode();
72              first.Value = value;
```

```
            return;
        }

        // Caso first != null precisamos buscar o último elemento da lista para então
        // preenche-lo

        // Nó atual
        var crr = first;
        // Enquanto existir um próximo, avance para ele
        while (crr.Next != null)
            crr = crr.Next;

        // Aqui crr.Next é nulo
        var next = new LinkedListNode();
        next.Value = value;
        crr.Next = next;
    }

    // Função para ler em uma posição específica do vetor, retornal null se o indice for inválido (fora da lista)
    public Cliente? Get(int index)
    {
        if (index < 0)
            return null;

        // Busca até atingir o índice ou ter crr nulo
        var crr = first;
        for (int i = 0; i < index && crr != null; i++)
            crr = crr.Next;

        // Se crr for nulo, retorna nulo, caso contrário retorna seu Value
        return crr?.Value;
    }
}
```