

CC3642 - Orientação a Objetos

Prof. Danilo H. Perico
2019

Conteúdo Programático



- Introdução: Orientação a Objetos?, C++, Java
- Conceitos de Classes e Objetos, Atributos e Métodos, Encapsulamento
- Gerenciamento de memória, Construtores (e Destrutores), Reusabilidade
- Intro à Linguagem de Modelagem Unificada (UML), Documentação, Sobrecargas (método e operador (C++))
- Estruturas de dados de armazenamento sequencial: Arrays / ArrayLists (Java) / Vector (C++)
- Herança
- P1
- Correção / Vista de prova
- Polimorfismo / Classes Abstratas / Interfaces (Java)
- Projeto OO Guiado por Padrões (*Design Patterns*) / Genéricos (Java) e *Templates* (C++)
- Exceções e Tratamentos de Erros
- Componentes de GUI
- P2

Gerenciamento de Memória

Stack vs. Heap

Stack e Heap

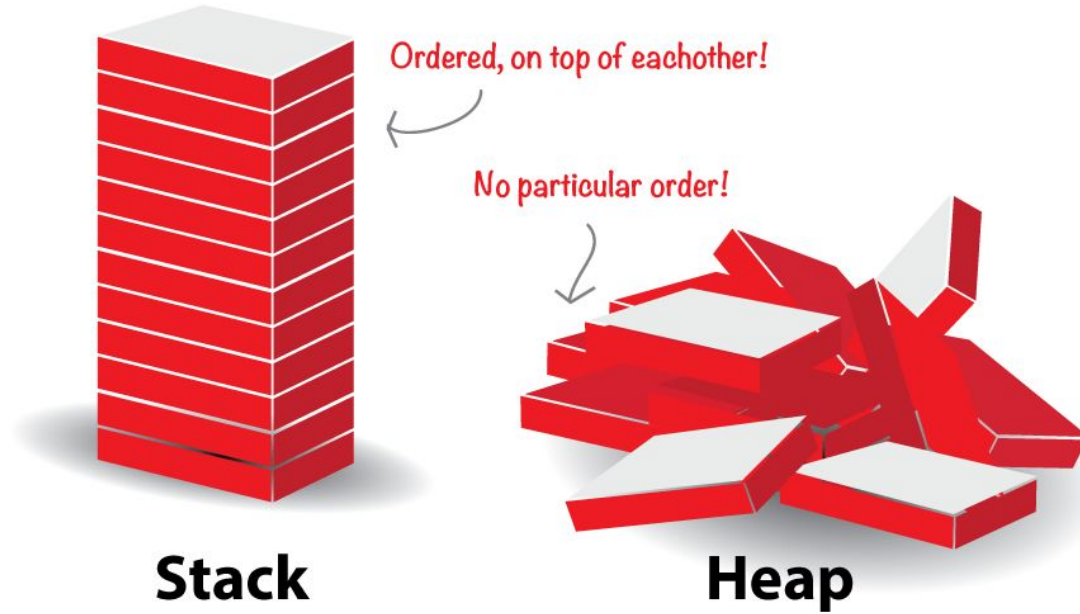
- A alocação de ambos costumam ser realizados na Memória Principal (*RAM*)



Stack e Heap



Stack e Heap



Stack e Heap

- Stack (pilha) - Alocação Estática
 - Ordenado
 - Região da memória que armazena temporariamente variáveis criadas por funções
 - Escopo local - Quando uma função termina, todas as variáveis são eliminadas
 - Tem acesso muito rápido
 - Não precisa desalocar as variáveis explicitamente
 - É limitada em espaço (depende do Sistema Operacional)
 - Variáveis não podem ser redimensionadas

Stack e Heap

- Heap - Alocação Dinâmica
 - Desordenado (aleatório)
 - Variáveis podem ser acessadas globalmente
 - Não tem limite de tamanho
 - Acesso mais lento
 - O programador deve gerenciar a alocação e desalocação
 - Variáveis podem ser redimensionadas

Stack e Heap

- Quando usar um ou outro?



Stack e Heap - Quando usar um ou outro?



A JVM / Hotspot / JIT é quem decide!

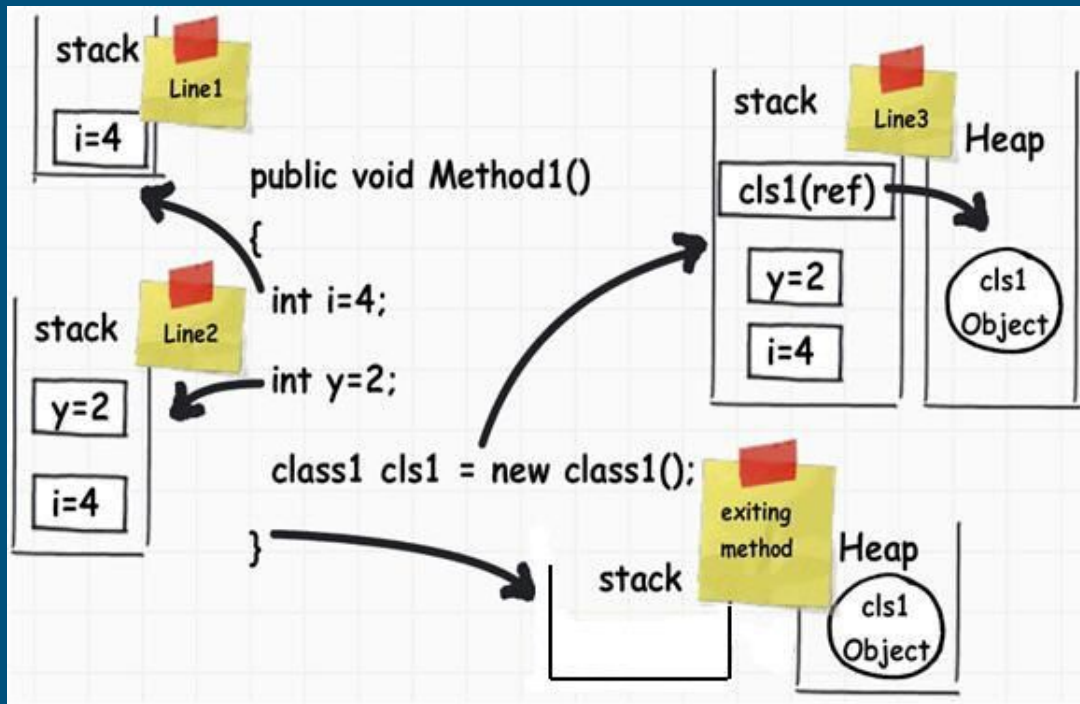
Regra geral:

- Objetos são alocados no Heap
- Tipos primitivos no Stack

Não precisa desalocar no Java: o Garbage Collector (GC) faz esse trabalho

Exemplo de alocação de memória

```
public void Method1()  
{  
    int i = 4;  
    int y = 2;  
    class1 cls1 = new class1();  
}
```





Stack e Heap

- Quando usar um ou outro?
 - Normalmente utiliza-se o Heap quando sabe-se que muita memória será necessária para o seu dado ou quando não se sabe ao certo quanta memória será necessária (como um vetor dinâmico).
 - Heap também é utilizado quando se deseja que o dado dure independentemente do escopo.
 - O resto é Stack!



Stack e Heap - Objetos

- Alocação Estática

`Calculadora calc;`

- Acesso aos membros da classe

`ponto (.)`

- Desalocação Estática

Não é diretamente feita pelo programador;
Depende do ciclo de vida / do escopo

- Alocação Dinâmica

`Calculadora *calc = new Calculadora;`

- Acesso aos membros da classe

`ponteiro (->)`

- Desalocação Dinâmica

`delete calc;`

```

1 #include <string>
2 #include <iostream>
3
4 using namespace std;
5
6 class Veiculos
7 {
8     public:
9         void moveFrente()
10        {
11            cout << "frente" << endl;
12        }
13
14        void moveRe()
15        {
16            cout << "re" << endl;
17        }
18
19        void viraDireita()
20        {
21            cout << "direita" << endl;
22        }
23
24        void viraEsquerda()
25        {
26            cout << "esquerda" << endl;
27        }
28     private:
29         string tipo;
30         int tamanho;
31 };

```

```

34 int main()
35 {
36     cout << "Objetos no Heap" << endl;
37     Veiculos *v1 = new Veiculos;
38     cout << v1 << endl;
39     v1->viraEsquerda();
40
41     Veiculos *v2 = new Veiculos;
42     Veiculos *v3 = new Veiculos;
43     Veiculos *v4 = new Veiculos;
44     Veiculos *v5 = new Veiculos;
45
46     cout << endl;
47     cout << "Objetos no Stack" << endl;
48     Veiculos v6;
49     cout << &v6 << endl;
50     v6.viraDireita();
51
52     Veiculos v7;
53     Veiculos v8;
54     Veiculos v9;
55     Veiculos v10;
56
57 }

```

```

perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$ ./veiculos
Objetos no Heap
0x1173010
esquerda

Objetos no Stack
0x7ffc8ee77dc0
direita

```

Construtores

Construtores - O que são?

- Um **construtor** é um tipo especial de método **chamado** para **criar** um **objeto**.
- O **construtor** prepara o novo objeto para uso, aceitando **argumentos** para **inicializar** os **atributos**.
- Toda **classe** tem pelo menos um **construtor**, se ele não for declarado explicitamente, o compilador fornece um **construtor-padrão**

Construtores - Definição

- O **construtor** tem o **mesmo nome** da **classe** (no Java e no C++)
 - Exemplo:
 - Classe Soma
 - Construtor: Soma()
- Usualmente, o construtor **não** pode retornar um **valor** (nem mesmo *void*)

Construtores no Java



- **Construtor-padrão:** não tem argumentos; pode ser explícito ou é chamado quando nenhum construtor é declarado.
- **Construtor parametrizado:** Aceita um ou mais parâmetros.

Construtores no Java



Construtor-padrão / sem argumento

```
1 class Student
2 {
3     private String name;
4     private int age;
5     private String branch;
6
7     public Student( ) //construtor
8     {
9         age = 10; branch = "QRT";
10    }
11 }
12
13
14 class TesteStudent
15 {
16     public static void main( String args[])
17     {
18         Student s1 = new Student( );
19     }
20 }
```

Construtor parametrizado

```
1 class Student
2 {
3     private String name;
4     private int age;
5     private String branch;
6
7     public Student( int a, String b ) //construtor
8     {
9         age = a; branch = b;
10    }
11 }
12
13
14 class TesteStudent
15 {
16     public static void main( String args[])
17     {
18         Student s1 = new Student(21, "CSE");
19     }
20 }
```



Sobrecarga de construtores

- Construtores sobrecarregados são construtores que tem o mesmo nome com assinaturas diferentes

A assinatura de um construtor é constituída pelo seu **nome** e pela lista de seus parâmetros, considerando: **tipo, número e ordem**



```
1 class Student
2 {
3     public String name;    //deveria ser private
4     public int age;        //deveria ser private
5     public String branch;  //deveria ser private
6
7     public Student( ) //construtor
8     {
9         age = 10; branch = "QRT";
10    }
11
12    public Student( int a, String b, String c)
13    {
14        age = a; branch = b; name = c;
15    }
16 }
17
18
19 class TesteStudent
20 {
21     public static void main( String args[])
22     {
23         Student s1 = new Student( );
24         Student s2 = new Student( 15, "CSE", "Fulano" );
25
26         System.out.println("s1" + " " + s1.age + " " + s1.name + " " + s1.branch);
27         System.out.println("s2" + " " + s2.age + " " + s2.name + " " + s2.branch);
28
29     }
30 }
```

```
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$ javac student.java
```

```
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$ java TesteStudent
```

```
s1 10 null QRT
```

```
s2 15 Fulano CSE
```

```
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$
```

```

4 public class Time2
5 {
6     private int hour;    // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // construtor sem argumento Time2 : inicializa cada variável de instância
11    // com zero; assegura que objetos Time2 iniciam em um estado consistente
12    public Time2()
13    {
14        this( 0, 0, 0 ); // invoca o construtor Time2 com três argumentos
15    } // fim do construtor sem argumento Time2
16
17    // Construtor Time2: hora fornecida, minuto e segundo padronizados para 0
18    public Time2( int h )
19    {
20        this( h, 0, 0 ); // invoca o construtor Time2 com três argumentos
21    } // fim do construtor de um argumento Time2
22
23    // Construtor Time2: hora e minuto fornecidos, segundo padronizado para 0
24    public Time2( int h, int m )
25    {
26        this( h, m, 0 ); // invoca o construtor Time2 com três argumentos
27    } // fim do construtor de dois argumentos Time2
28
29    // Construtor Time2: hour, minute e second fornecidos
30    public Time2( int h, int m, int s )
31    {
32        setTime( h, m, s ); // invoca setTime para validar a data/hora
33    } // fim do construtor de três argumentos Time2
34
35    // Construtor Time2: outro objeto Time2 fornecido
36    public Time2( Time2 time )
37    {
38        // invoca o construtor de três argumentos Time2
39        this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // fim do construtor Time2 com um argumento de objeto Time2
41
42    // Métodos set
43    // configura um novo valor de data/hora usando UTC; assegura que
44    // os dados permaneçam consistentes configurando valores inválidos como zero
45    public void setTime( int h, int m, int s )
46    {
47        setHour( h ); // configura hour
48        setMinute( m ); // configura minute
49        setSecond( s ); // configura second
50    } // fim do método setTime

```

```

52    // valida e configura a hora
53    public void setHour( int h )
54    {
55        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56    } // fim do método setHour
57
58    // valida e configura os minutos
59    public void setMinute( int m )
60    {
61        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    } // fim do método setMinute
63
64    // valida e configura os segundos
65    public void setSecond( int s )
66    {
67        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68    } // fim do método setSecond

```

operador
condicional
(?)

```

1 // Fig. 8.6: Time2Test.java
2 // Construtores sobrecarregados utilizados para inicializar objetos Time2.
3
4 public class Time2Test
5 {
6     public static void main( String args[] )
7     {
8         Time2 t1 = new Time2();           // 00:00:00
9         Time2 t2 = new Time2( 2 );       // 02:00:00
10        Time2 t3 = new Time2( 21, 34 );   // 21:34:00
11        Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12        Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13        Time2 t6 = new Time2( t4 );       // 12:25:42
14    }

```

Exercício 13

Crie uma classe *Funcionario* com os seguintes atributos:

- string **nome**; int **idade**, string **sexo**, int **numero**
- Faça um **construtor padrão** (sem parâmetro nenhum) para inicializar o objeto.
- Faça um **construtor com 4 parâmetros**, um para cada atributo.
- Teste a classe instanciando vários objetos da classe Funcionario.

Exercício 14

(Deitel 8.16) Crie uma classe *Data* com as seguintes capacidades:

- A. Gerar saída em múltiplos formatos:
 - i. MM/DD/YYYY
 - ii. Março 02, 2019
 - iii. DDD YYYY
- B. Utilizar construtores sobrecarregados para criar objetos *Data* inicializados com datas nos formatos da parte (A). No primeiro caso, o construtor deve receber 3 valores inteiros. No segundo caso deve receber uma String e dois valores inteiros. No terceiro caso deve receber dois valores inteiros, o primeiro sendo o número de dias no ano. [Dica: para converter a representação de string do mês em valor numérico, compare as strings utilizando o método *equals*. Por exemplo, se *s1* e *s2* forem strings, a chamada de método *s1.equals(s2)* retornará true se as strings forem idênticas.

Exercício 15

(Deitel 3.13) Crie uma classe chamada *Fatura* que uma loja de suprimentos de informática possa utilizar para representar uma fatura de um item vendido na loja. Uma fatura deve incluir quatro partes de informação: um número identificador, uma descrição, a quantidade comprada de um item e o preço por item. Sua classe deve ter um construtor que inicializa os quatro atributos. Forneça uma função *set* e uma função *get* para cada variável de instância. Além disso, forneça uma função-membro chamada *getValor* que calcula a quantia da fatura (isto é, multiplica a quantidade pelo preço, por item) e depois retorna o valor. Se o valor total da fatura não for positivo, ele deve ser configurada como 0 (zero). Se o preço por item não for positivo, ele deve ser configurado como 0 (zero). Escreva um programa de teste que demonstre as capacidades da classe *Fatura*.



Construtores no C++

- **Construtor-padrão:** não tem argumentos; pode ser explícito ou pode ser chamado quando nenhum construtor é declarado.
- **Construtor parametrizado:** Aceita um ou mais parâmetros.



Construtores no C++

Construtor-padrão

```
1 class Student
2 {
3 private:
4     string name;
5     int age;
6     string branch;
7 public:
8     Student() //Construtor
9     {
10         age = 20;
11         branch = "CSE";
12     }
13 };
14
15 int main()
16 {
17     Student s1;
18 }
```

Construtor parametrizado

```
1 class Student
2 {
3 private:
4     string name;
5     int age;
6     string branch;
7 public:
8     Student(int a, string b) //Constructor
9     {
10         age = a;
11         branch = b;
12     }
13 };
14
15 int main()
16 {
17     Student s1(21, "CSE");
18 }
```



Construtores no C++

Construtor-padrão: Variações

```
1 class Student
2 {
3 private:
4     string name;
5     int age;
6     string branch;
7 public:
8     Student() //Construtor
9     {
10         age = 20;
11         branch = "CSE";
12     }
13 };
14
15 int main()
16 {
17     Student s1;
18 }
```

```
6 class Student
7 {
8 private:
9     string name;
10    int age;
11    string branch;
12 public:
13     Student( ) : age(20), branch("CSE") { }
14 };
15
16
17 int main()
18 {
19     Student s1;
20 }
```

lista de inicializadores de
membro



Construtores no C++

Construtor parametrizado: Variações

```
1 class Student
2 {
3 private:
4     string name;
5     int age;
6     string branch;
7 public:
8     Student(int a, string b) //Constructor
9     {
10         age = a;
11         branch = b;
12     }
13 };
14
15 int main()
16 {
17     Student s1(21, "CSE");
18 }
```

```
6 class Student
7 {
8 private:
9     string name;
10    int age;
11    string branch;
12 public:
13    Student(int a, string b) : age(a), branch(b) { }
14 };
15
16
17 int main()
18 {
19     Student s1(21, "CSE");
20 }
```

lista de inicializadores de
membro

Destructor



Função que é chamada sempre que o escopo de duração do objeto encerra-se ou quando o comando **delete** é usado;

Objetivo:

- liberação de memória
- finalização de dispositivos ou subsistemas que tenham sido ativados

```
~<NomeDaClasse>( ) { ... }
```

O nome do destrutor é o da classe com um til (~) anexado no início.



- Não existe o conceito de destrutores em Java
- O Garbage Collector (GC) irá destruir o objeto na hora que ele achar conveniente e o programador não tem controle sobre isso.


Reusabilidade e Separação entre Interface e Implementação



Reusabilidade

- Colocar uma classe em um arquivo separado (.h)

main.cpp
(código-cliente)



```
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // inclui a definição de classe GradeBook
8
9 // a função main inicia a execução do programa
10 int main()
11 {
12     // cria dois objetos GradeBook
13     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
14     GradeBook gradeBook2( "CS102 Data Structures in C++" );
15
16     // exibe valor inicial de courseName para cada GradeBook
17     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
18         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
19         << endl;
20     return 0; // indica terminação bem-sucedida
21 } // fim de main
```

GradeBook.h

```
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string> // a classe GradeBook utiliza a classe de string padrão C++
8 using std::string;
9
10 // Definição da classe GradeBook
11 class GradeBook
12 {
13 public:
14     // o construtor inicializa courseName com a string fornecida como argumento
15     GradeBook( string name )
16     {
17         setCourseName( name ); // chama a função set para inicializar courseName
18     } // fim do construtor GradeBook
19
20     // função para configurar o nome do curso
21     void setCourseName( string name )
22     {
23         courseName = name; // armazena o nome do curso no objeto
24     } // fim da função setCourseName
25
26     // função para obter o nome do curso
27     string getCourseName()
28     {
29         return courseName; // retorna courseName do objeto
30     } // fim da função getCourseName
31
32     // exibe uma mensagem de boas-vindas para o usuário GradeBook
33     void displayMessage()
34     {
35         // chama getCourseName para obter o courseName
36         cout << "Welcome to the grade book for\n" << getCourseName()
37             << "!" << endl;
38     } // fim da função displayMessage
39 private:
40     string courseName; // nome do curso para esse GradeBook
41 }; // fim da classe GradeBook
```

Separação entre Interface e Implementação da Classe

Interface

- A interface da classe deve descrever que serviços os clientes da classe podem utilizar e como solicitar esses serviços
- Contém os protótipos das funções-membro

Implementação

- A implementação é o código-fonte da classe;
- É onde as funções-membro são programadas;

Por que separar? (Deitel) *“é uma melhor engenharia de software definir funções-membro fora da definição de classe, para que os detalhes da sua implementação possam ficar ocultos do código-cliente. Essa prática assegura que os programadores não escrevam código-cliente que dependa dos detalhes de implementação da classe. Se eles precisassem fazer isso, o código-cliente provavelmente ‘quebraria’ se a implementação da classe fosse alterada”.*

Separação entre Interface e Implementação da Classe

Interface

GradeBook.h

```
2 #include <string>
3 using std::string;
4
5 class GradeBook
6 {
7 public:
8     GradeBook( string );
9     void setCourseName( string );
10    string getCourseName();
11    void displayMessage();
12 private:
13    string courseName;
14 };
```

E para usar a classe no código-cliente,
basta continuar a inserir:

```
#include "GradeBook.h"
```

Implementação

GradeBook.cpp

```
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 #include "GradeBook.h"
7
8 GradeBook::GradeBook( string name )
9 {
10     setCourseName( name );
11 }
12
13 void GradeBook::setCourseName( string name )
14 {
15     courseName = name;
16 }
17
18 string GradeBook::getCourseName()
19 {
20     return courseName;
21 }
22
23 void GradeBook::displayMessage()
24 {
25     cout << "Welcome to the grade book for\n" << getCourseName()
26         << "!" << endl;
27 }
28
29
30 }
```

Exercício 16

(Deitel 3.14) Crie uma classe chamada *Employee* que inclua três partes de informações como atributos — um nome (tipo string), um sobrenome (tipo string) e um salário mensal (tipo float). Sua classe deve ter um construtor que inicialize os três membros de dados. Forneça uma função *set* e uma função *get* para cada atributo. Se o salário mensal não for positivo, configure-o como 0. Escreva um programa de teste que demonstre as capacidades da class *Employee*. Crie dois objetos *Employee* de forma dinâmica (alocado no heap) e exiba o salário anual de cada objeto. Então dê a cada *Employee* um aumento de 10% e exiba novamente o salário anual de cada um. Crie a classe com os conceitos de reutilização e separação entre interface e implementação.