

# Programação e Desenvolvimento de Software 2

## Programação Orientada a Objetos (Tópicos Avançados)

---

Prof. Luiz Chaimowicz  
(slides adaptados do Prof. Douglas Macharet)

# Introdução

- Polimorfismo Estático
  - Tempo de compilação
  - Ligação Prematura (Early/Static binding)
  - Sobrecarga
- Polimorfismo Dinâmico
  - Tempo de execução
  - Ligação Tardia (Late/Dynamic binding)
  - Sobrescrita

# Introdução

- Polimorfismo Estático
  - Geralmente considerado mais eficiente
- Polimorfismo Dinâmico
  - Apresenta uma flexibilidade maior
- Como são implementados
  - *Static Dispatch x Dynamic Dispatch*

# Exemplo 1

```
class A {  
  
    public:  
        virtual void f() {  
            cout << "A::f()" << endl;  
        }  
};  
  
class B : public A {  
  
    public:  
        void f() override {  
            cout << "B::f()" << endl;  
        }  
};
```

```
int main() {  
  
    A *pa = new B();  
    pa->f();  
  
    return 0;  
}
```

**Saída:**

B::f()

## Exemplo 2

```
int main () {  
  
    A a;  
    B b;  
    a = b;  
    a.f();  
  
    return 0;  
}
```

**Saída:**

A :: f ()

Apenas a parte relativa a A de 'b' é copiada em 'a'.

# Static vs. Dynamic Dispatch

- Method Dispatch
  - Como a linguagem decide qual implementação do método usar
- Static Dispatch (compilação)
  - Garantia de que há apenas uma única implementação do método
- Dynamic Dispatch (execução)
  - Adiar a seleção da implementação até o *runtime type* ser conhecido

[https://en.wikipedia.org/wiki/Static\\_dispatch](https://en.wikipedia.org/wiki/Static_dispatch)  
[https://en.wikipedia.org/wiki/Dynamic\\_dispatch](https://en.wikipedia.org/wiki/Dynamic_dispatch)

# Static vs. Dynamic Dispatch

## ■ Quando C++ usa Static/Dynamic Dispatch?

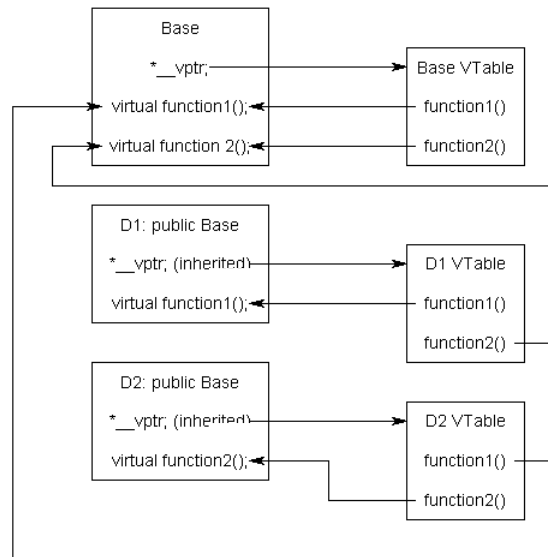
Tipo	Valor ou Referência?	Chamada	Como f declarada em A?	Static ou Dynamic?
A a;	Valor	a.f()	Virtual	Static
A a;	Valor	a.f()	Não virtual	Static
A* pa;	Referência	pa->f()	Virtual	Dynamic
A* pa;	Referência	pa->f()	Não virtual	Static

# Static vs. Dynamic Dispatch

## Virtual Method Table (vtable)

- Diferentes formas de implementar Dynamic Dispatch
- C++ / C# → Tabelas de Métodos Virtuais

```
class Base {  
    public:  
        virtual void function1() {};  
        virtual void function2() {};  
};  
  
class D1: public Base {  
    public:  
        void function1() override {};  
};  
  
class D2: public Base {  
    public:  
        void function2() override {};  
};
```



Quando uma classe define um método virtual, o compilador adiciona um membro oculto à classe que aponta para uma matriz de ponteiros para funções (virtuais).

[https://en.wikipedia.org/wiki/Virtual\\_method\\_table](https://en.wikipedia.org/wiki/Virtual_method_table)

g++ -fdump-class-hierarchy ou -fdump-lang-class



# Static vs. Dynamic Dispatch

## Construtores e Destrutores

```
class ClasseBase {  
  
    public:  
        ClasseBase() {  
            cout << "BASE Constructor..." << endl;  
        }  
        ~ClasseBase() {  
            cout << "BASE Destructor..." << endl;  
        }  
};  
  
class ClasseDerivada : public ClasseBase {  
  
    public:  
        ClasseDerivada() {  
            cout << "DERIVADA Constructor..." << endl;  
        }  
        ~ClasseDerivada() {  
            cout << "DERIVADA Destructor..." << endl;  
        }  
};
```

# Static vs. Dynamic Dispatch

## Construtores e Destrutores

```
int method() {
    ClasseDerivada d;
}

int main() {

    method();

    cout << "-----" << endl;

    ClasseBase *b = new ClasseDerivada();
    delete b;

    return 0;
}
```

[Wandbox](#)

### Saída:

BASE Constructor...  
DERIVADA Constructor...  
DERIVADA Destructor...  
BASE Destructor...

-----  
BASE Constructor...  
DERIVADA Constructor...  
BASE Destructor...

# Static vs. Dynamic Dispatch

## Destrutores virtuais

- Destrutores de classes base devem sempre ser virtuais
  - Se pretende utilizar de maneira polimórfica e desalocar o objeto do tipo derivado por meio de um ponteiro para seu tipo base
  - Ocorre um *static dispatch* se não for virtual
- Tipo estático vs. Tipo dinâmico
  - Tipo da variável declarada (contrato/referência)
  - Tipo do objeto na memória (comportamento)

Effective C++: Pg. 40 – Item 7: Declare destructors virtual in polymorphic base classes.

- Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

# Static vs. Dynamic Dispatch

## Destrutores virtuais

```
class A {
public:
    A() { cout << "A::Constructor()" << endl; }
    virtual ~A() { cout << "A::Destructor()" << endl; }
};

class B : public A {
public:
    B() { cout << "B::Constructor()" << endl; }
    ~B() { cout << "B::Destructor()" << endl; }
};

class C : public B {
public:
    C() { cout << "C::Constructor()" << endl; }
    ~C() { cout << "C::Destructor()" << endl; }
};
```

```
int main() {

    A *a = new C();
    delete a;

    return 0;
}
```

### Saída:

```
A::Constructor()
B::Constructor()
C::Constructor()
C::Destructor()
B::Destructor()
A::Destructor()
```

# Programação e Desenvolvimento de Software 2

POO e Gerenciamento de memória

---

Prof. Luiz Chaimowicz  
(slides adaptados do Prof. Douglas Macharet)

# Introdução

- Em linguagens de baixo nível, é importante compreensão do modelo de memória e operações próprias para manipulação
- Gerenciamento explícito da memória
  - new / delete (single variables)
  - new[] / delete[] (array variables)
- Prós/Cons
  - Uso eficiente (personalizado) da memória
  - Fácil ter programas problemáticos

# Introdução

- Mau gerenciamento de memória
  - Usar variáveis (posições de memória) não inicializadas
  - Alocar memória e não excluí-la quando necessário
  - Tentar acessar um valor (posição) que não é mais válido
- Boas práticas
  - Sempre inicializar as variáveis (verificar antes de usar)
  - Sempre liberar a memória após o uso (alertar sobre isso)
  - Certificar que a variável (memória) não é mais utilizada

# Construtores

- Construtores
  - Inicialização dos membros após a alocação na memória
  - Baseado nos parâmetros informados na assinatura
- Construtor de cópia
  - Padrão / User-defined
    - Recebe um objeto (referência) e copia os valores dos atributos
  - Tipos
    - Shallow: apenas copia os valores/referências no novo objeto
    - Deep: caso demandado, faz uma nova alocação antes da cópia

[https://en.wikipedia.org/wiki/Copy\\_constructor\\_\(C%2B%2B\)](https://en.wikipedia.org/wiki/Copy_constructor_(C%2B%2B))

<https://www.geeksforgeeks.org/copy-constructor-in-cpp/>




# Construtores

- Quando o construtor de cópia é chamado?
  - Objeto é retornado ou passado por valor como argumento
  - Objeto construído com base em outro objeto da mesma classe
  - Compilador precisa gerar um objeto temporário
- Quando preciso definir um próprio?
  - Quando a classe possui ponteiros/alocação durante execução
  - Basicamente para elementos no Heap → Deep Copy

# Construtores

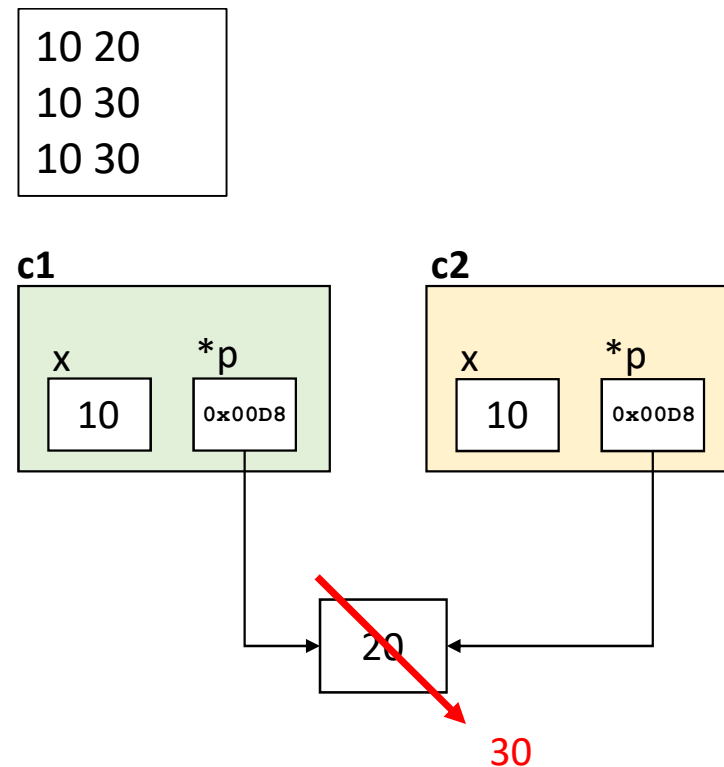
## Exemplo 1

```
class ClasseTeste {  
    public:  
        int x, *p;  
  
    ClasseTeste() {  
        this->p = new int;   
    }  
  
    void display() {  
        cout << this->x << " " << *this->p << endl;  
    }  
};
```

# Construtores

## Exemplo 1

```
int main() {  
  
    ClasseTeste c1;  
    c1.x = 10;  
    *c1.p = 20;  
    c1.display();  
  
    ClasseTeste c2 = c1;  
    *c2.p = 30;  
    c2.display();  
  
    c1.display();  
  
    return 0;  
}
```



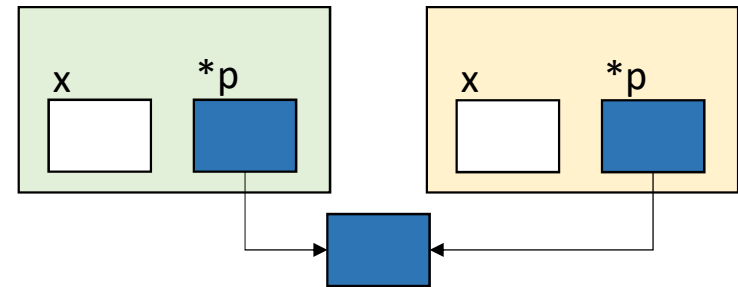
# Construtores

## Exemplo 1

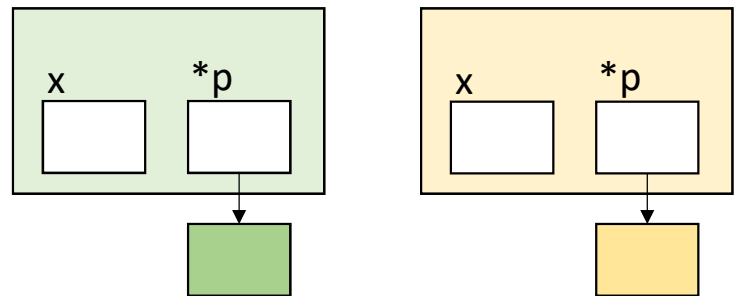
```
class ClasseTeste {  
    public:  
        int x, *p;  
  
    ClasseTeste() {  
        this->p = new int;  
    }  
  
    ClasseTeste(ClasseTeste &source) {  
        this->x = source.x;  
        this->p = new int;  
        *this->p = *source.p; ←  
    }  
  
    void display() {  
        cout << this->x << " " << *this->p << endl;  
    }  
};
```

[Wandbox](#)

### Shallow Copy



### Deep Copy



# Construtores

## Exemplo 2

- Atenção, às vezes esse pode ser o comportamento desejado!

```
class Curso {  
  
    public:  
        string nome;  
        int credits;  
  
};  
  
class Estudante {  
  
    public:  
        string nome;  
        Curso* curso;  
  
};
```

```
int main() {  
    Curso curso;  
    curso.nome = "PDS2";  
    curso.credits = 4;  
  
    Estudante e1;  
    e1.nome = "Maria";  
    e1.curso = &curso;  
  
    Estudante e2(e1);  
    e2.nome = "Joao";  
  
    cout << e1.curso->nome << " " << e1.nome << endl;  
    cout << e2.curso->nome << " " << e2.nome << endl;  
    return 0;  
}
```

[Wanbdox](#)

# Construtores

- Regra simples (nem sempre recomendada)
  - Sempre que o operador **new** for utilizado, deve-se ser capaz de identificar quando a exclusão será feita (ou seja, **delete** associado)
- Formas de evitar problemas
  - Ocultar a alocação de memória em um *resource handle*
    - Ele passa a ser o responsável pelo gerenciamento
    - Ao ser destruído, ele deve excluir essa memória
    - SmartPointers (C++ 11)
  - Manter uma contagem das referências
    - Operadores/ferramentas auxiliares da linguagem

<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-newdelete>

# Destrutores

- Destrutores
  - Podem ser responsáveis/utilizados para desalocar qualquer memória dinâmica (ponteiros) associada aos atributos da classe
- Utilize, mas entenda os riscos
  - Não são chamados em algumas situações
    - Remoção por um ponteiro base, sem destrutor virtual
    - Lançamento de exceção no construtor
    - Término prematuro do programa (exit)

# Destrutores

## Exemplo 1

```
class ClasseTeste {
public:
    int *x, *p;

    ClasseTeste() {
        this->x = new int;
        if (this->x == nullptr) {
            cout << "Memoria insuficiente!" << endl;
            exit(1);
        }

        this->p = new int;
        if (this->p == nullptr) {
            cout << "Memoria insuficiente!" << endl;
            exit(1);
        }
    }

    ~ClasseTeste() {
        delete this->x;
        delete this->p;
    }
};
```





# Rule of Three

- Se uma classe precisa que um (ou mais) dos seguintes membros seja definido pelo usuário, provavelmente deverá definir todos os três:
  - Destrutor
  - Construtor de cópia
  - Operador de atribuição de cópia
- Se a versão padrão para uma não se ajusta às necessidades da classe, então provavelmente as outras funções padrões também não servem
- Outras “regras”: Rule of Five / Rule of Zero

[https://en.wikipedia.org/wiki/Rule\\_of\\_three\\_\(C%2B%2B\\_programming\)](https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming))

[https://en.cppreference.com/w/cpp/language/rule\\_of\\_three](https://en.cppreference.com/w/cpp/language/rule_of_three)

# Rule of Three

## Exemplo 1

```
class Test {  
    public:  
    Test() {  
        cout << "Constructor called." << endl;  
    }  
    ~Test() {  
        cout << "Destructor called." << endl;  
    }  
    Test(const Test &t) {  
        cout << "Copy constructor called." << endl;  
    }  
    Test& operator = (const Test &t) {  
        cout << "Assignment operator called." << endl;  
        return *this;  
    }  
};
```

```
int main() {  
    Test t1, t2;  
    t2 = t1;  
    Test t3 = t1;  
    return 0;  
}
```

Constructor called.  
Constructor called.  
Assignment operator called.  
Copy constructor called.  
Destructor called.  
Destructor called.  
Destructor called.

# Smart Pointers (C++11)

- Tipo Abstrato de Dado que simula um ponteiro tradicional
  - Envolvem ponteiros e sobrecarregam os operadores ( $\rightarrow$ ,  $*$ ,  $=$ )
- Gerenciamento automático de memória
  - Quando o *smart pointer* não está mais em uso (sai do escopo), a memória para a qual ele aponta é desalocada automaticamente
- Tipos principais
  - `std::unique_ptr`
  - `std::shared_ptr`

[https://en.wikipedia.org/wiki/Smart\\_pointer](https://en.wikipedia.org/wiki/Smart_pointer)

<https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170>

# Smart Pointers (C++11)

- `std::unique_ptr`
  - Possui um recurso alocado dinamicamente
  - Apenas ele pode apontar para o recurso
- `std::shared_ptr`
  - Possui um recurso alocado compartilhado
  - Mantém um contador interno com o número de `shared_ptr` que possuem o mesmo recurso


# Smart Pointers (C++11)

## Exemplo 1

```
class ClasseA {  
    public:  
        int id;  
  
    ClasseA(int id) : id(id) {  
        cout << "ClasseA::Constructor:" << this->id << endl;  
    }  
  
    ~ClasseA() {  
        cout << "ClasseA::Destructor:" << this->id << endl;  
    }  
};
```

# Smart Pointers (C++11)

## Exemplo 1

```
#include <memory>   
#include <iostream>  
using namespace std;  
  
int main() {  
  
    ClasseA c1(1);  
    ClasseA *c2 = new ClasseA(2);  
  
    unique_ptr<ClasseA> c3(new ClasseA(3));  
    cout << c3->id << endl;  
  
    return 0;  
}
```

```
ClasseA::Constructor:1  
ClasseA::Constructor:2  
ClasseA::Constructor:3  
3  
ClasseA::Destructor:3  
ClasseA::Destructor:1
```

# Smart Pointers (C++11)

## Exemplo 2

```
int main() {  
  
    unique_ptr<ClasseA> c1(new ClasseA(1));  
  
    // Compile Error : unique_ptr object is not copyable  
    // unique_ptr<ClasseA> c2 = c1;  
  
    shared_ptr<ClasseA> c2(new ClasseA(2));  
    shared_ptr<ClasseA> c3 = c2;  
  
    cout << c2.use_count() << endl;  
  
    c3 = nullptr;  
    cout << c2.use_count() << endl;  
  
    return 0;  
}
```

```
ClasseA::Constructor:1  
ClasseA::Constructor:2  
2  
1  
ClasseA::Destructor:2  
ClasseA::Destructor:1
```

# Smart Pointers (C++11)

## Exemplo 3

```
class Animal {  
public:  
    virtual void fale() {  
        cout << "Fale padrao!" << endl;  
    };  
  
    ~Animal() {  
        cout << "Animal::Destructor" << endl;  
    }  
};
```

```
class Gato : public Animal {  
public:  
    void fale() override {  
        cout << "Miau!" << endl;  
    }  
  
    ~Gato() {  
        cout << "Gato::Destructor" << endl;  
    }  
};
```

```
class Cachorro : public Animal {  
public:  
    void fale() override {  
        cout << "Au! Au!" << endl;  
    }  
  
    ~Cachorro() {  
        cout << "Cachorro::Destructor" << endl;  
    }  
};
```



# Smart Pointers (C++11)

## Exemplo 3

```
#include <list>

int main() {

    list<Animal*> lista;

    for(int i=0; i<5;i++) {
        if (i % 2 == 0)
            lista.push_back(new Cachorro());
        else
            lista.push_back(new Gato());
    }

    for (auto a : lista)
        a->fale();

    return 0;
}
```

Au! Au!  
Miau!  
Au! Au!  
Miau!  
Au! Au!

Memory Leak!  
Nenhum destrutor é chamado!

# Smart Pointers (C++11)

## Exemplo 3

```
int main() {  
  
    list<unique_ptr<Animal>> lista;  
  
    for(int i=0; i<5;i++) {  
        if (i % 2 == 0)  
            lista.push_back(unique_ptr<Animal>(new Cachorro()));  
        else  
            lista.push_back(unique_ptr<Animal>(new Gato()));  
    }  
  
    for (auto const &a : lista)  
        a->fale();  
  
    return 0;  
}
```

Au! Au!  
Miau!  
Au! Au!  
Miau!  
Au! Au!  
Animal::Destructor  
Animal::Destructor  
Animal::Destructor  
Animal::Destructor  
Animal::Destructor

Memory Leak!  
Destructor derivado não é chamado!

# Smart Pointers (C++11)

## Exemplo 3

```
class Animal {  
    public:  
        virtual void fale() {  
            cout << "Fale padrao!" << endl;  
        };  
  
        virtual ~Animal() {  
            cout << "Animal::Destructor" << endl;  
        }  
};
```

[Wandbox](#)

Au! Au!  
Miau!  
Au! Au!  
Miau!  
Au! Au!  
Cachorro::Destructor  
Animal::Destructor  
Gato::Destructor  
Animal::Destructor  
Cachorro::Destructor  
Animal::Destructor  
Gato::Destructor  
Animal::Destructor  
Cachorro::Destructor  
Animal::Destructor

# Smart Pointers (C++11)

Mais alguns detalhes...

- Reset()
- Release()
- “Deleter”
- Sobrecargas

Consultar a documentação...

[https://cplusplus.com/reference/memory/unique\\_ptr/](https://cplusplus.com/reference/memory/unique_ptr/)