Polimorfismo - Definição

- Capacidade de assumir formas diferentes.
- Em OO, se refere à capacidade de um objeto de uma classe base poder armazenar um objeto de uma de suas classes derivadas.
- Em C++, isso é feito através de ponteiros: Um ponteiro para uma classe base pode armazenar o endereço de uma classe derivada.

Importância do polimorfirsmo

- Favorece a componentização e consequentemente, o reuso de software, na medida em que um comportamento básico pode ser estendido para atender às peculiaridades das classes derivadas.

- Favorece a abstração e o encapsulamento, na medida em que nomes iguais escondem a implementação diferente

Formas de polimorfismo

- Há quatro formas de polimorfismo nas linguagens de programação OO:
  - ***Sobrecarga ou overloading***: um mesmo nome é usado com diferentes comportamentos e diferentes argumentos;
  - ***Redefinição ou overriding***: uma classe derivada estabelece um comportamento diferente para um método herdado de sua classe base;
  - ***Variáveis polimórficas***: uma variável que pode assumir valores de diferentes tipos durante a execução.
  - ***Templates***: uma forma de criar modelos para classes parametrizando os tipos. (observação: essa última não será vista nesta aula!)

Sobrecarga/overload

- Um termo sobrecarregado pode ser usado em diferentes contextos, com significados diferentes.

- O mesmo ocorre com a linguagem natural:
  - Uma manga de camisa
  - Uma fruta manga

- O contexto permite identificar o significado atribuído ao termo.

Sobrecarga/overload

- A sobrecarga é a forma já conhecida de polimorfismo!
- Seja o operador +
- Em C++, o mesmo operador pode ser usado em diferentes contextos:
  - 4 + 5 (adição de inteiros)
  - 3.14 + 2.0 (adição em ponto flutuante)
  - “cata”+ “vento" (concatenação de strings!)
- Dá para imaginar que a implementação da função + é diferente para cada caso!

Sobrecarga/overload

- A sobrecarga pode ser de operadores e também de funções.

- C++ permite que o programador faça mais de uma definição para um mesmo nome dentro do mesmo escopo desde que as declarações tenham:
  - assinaturas diferentes e
  - implementações diferentes

Sobrecarga/overload

- Quando o compilador encontra uma função ou operador sobrecarregados
  - decide a implementação mais adequada com base na comparação das assinaturas dos métodos.

- Este processo é chamado de resolução da sobrecarga

Sobrecarga/overload

- Assinatura
  - Dentro do mesmo escopo, a assinatura dos métodos precisa ser diferente.
  - A assinatura é uma combinação do número e tipo dos argumentos e do tipo do retorno.
  - Não pode haver sobrecarga apenas variando o tipo de retorno.

Sobrecarga/overload

- Exemplo: Soma

```
int soma (int a){
        return a;
}
int soma (int a, int b){
        return a+b;
}
int soma (int a, int b, int c) {
        return a+b+c;
}
int main() {
        // testando a sobrecarga por assinatura
        cout << "6 = " << soma(6) << endl;
        cout << "6 + 6 = " << soma (6,6) << endl;
        cout << "6 + 6 + 6 = " << soma (6,6,6) << endl;
        return 0;
}
```

Redefinição/Overriding

- A redefinição acontece quando uma classe derivada modifica um método que herdou da classe base.

- A função redefinida pela classe derivada tem a mesma declaração da classe base. Isto significa:
  - O mesmo nome
  - O mesmo tipo de retorno
  - A mesma assinatura (lista de parâmetros)

Overriding x Overload

- A redefinição só faz sentido no contexto da herança. A sobrecarga é numa mesma classe;
- As assinaturas precisam ser as mesmas;
- Os métodos redefinidos podem ser combinados com os que os redefinem;
- Em geral, é resolvida em tempo de execução e não em tempo de compilação

Exemplo: Pássaros

- Seja a classe Pássaro

- Suponha que todos os pássaros cantem, cada um de sua forma.

Exemplo 1: passaro.h

```
class Passaro {
     protected:
          bool emExtincao;
          string corPredominante;
     public:
          Passaro();
          ~Passaro();
          void canta();
          bool getEmExtincao();
          string getCorPredominante();
};
```

Exemplo 1: passaro.cpp

```
#include "Passaro.h"
Passaro::Passaro() {
	this->emExtincao = false;
	this->corPredominante = "cinza";
}
Passaro::~Passaro() {}
void Passaro::canta() {
	cout << "Piu Piu Piu" << endl;
}
bool Passaro::getEmExtincao () {
	return this->emExtincao;
}
string Passaro::getCorPredominante () {
	return this->corPredominante;
}
```

Exemplo 1: arara

```
class Arara: public Passaro {
      public:
            Arara();
            ~Arara();
            void canta ();
};

Arara::Arara():Passaro(){
      this->emExtincao = true;
      this->corPredominante = "azul";
}
Arara::~Arara() { }
void Arara::canta() {
      cout << "A-RA-RA --- A-RA-RA" << endl;
}
```

main.cpp

```
int main() {
	passaro *p = new Passaro ();
	cout  << "Passaro "
		<< p->getCorPredominante()
		<< " em extincao? R: "
		<< p->getEmExtincao() << endl;
	p->canta();
	arara *a = new Arara ();
	cout  << "Arara "
		<< a->getCorPredominante()
		<< " em extincao? R: "
		<< a->getEmExtincao() << endl;
	a->canta();
	delete p;
	delete a;
	return 0;
}
```

Cuidado!!!

- Se uma arara é um pássaro também, o que deve acontecer neste caso?

```
int main() {
        passaro *p [3];
        p[0] = new Passaro ();
        p[1] = new Arara ();
        p[2] = new Arara ();
        for (int i = 0; i < 3; i++) {
              p[i]->canta();
        }
        for (int i = 0; i < 3; i++) {
              delete p[i];
        }
        return 0;
}
```

Por quê?

- Porque o compilador escolhe qual método ele vai acionar a partir do tipo da variável no código fonte, e no comando p[i]->canta() p é do tipo Passaro!
- Isto é, o compilador decide qual método usar antes de executar o programa (em tempo de compilação)
- Isto é chamado de ligação estática (static binding)!
- Observe que a atribuição p[1] = new Arara ();
- é válida!

Virtual

- Para que o compilador adie a decisão de qual método usar até o momento da execução do programa, declaramos o método como virtual.

- Isso indica ao compilador que espere a execução do programa para então decidir o método a ser usado com base na classe a que pertence o objeto.

- Este efeito é chamado de ligação dinâmica (dynamic binding)

Virtual

```
#include <iostream>
#include <cstring>
using namespace std;
class passaro {
      protected:
            bool emExtincao;
            string corPredominante;
      public:
            Passaro();
            virtual ~Passaro();
            virtual void canta();
            bool getEmExtincao();
            string getCorPredominante ();
};
```

Virtual

```
#include "Arara.h"

Arara::Arara():Passaro(){
	this->emExtincao = true;
	this->corPredominante = "azul";
}

Arara::~Arara() {
	cout << "Arara deletada!" << endl;
}

void Arara::canta() {
	cout << "A-RA-RA --- A-RA-RA" << endl;
}
```

Polimorfismo – Métodos virtuais

- Quando chamamos um método sobrecarregado de um ponteiro de classe base que armazena um objeto de uma classe derivada, o método chamado é o da classe base. Mas podemos usar a diretiva virtual no método da base para forçar a checagem em tempo de execução de qual método deve ser usado:

```
class Base
{
    public:
    Base(int);
    virtual void funcao(); ///metodo da base.
};

class Derivada : /**public,private,protected*/ Base
{
    public:
    Derivada(int);
    void funcao(); /// redefinida na derivada.
};

int main()
{
    Base* objetoB = new Base(10);
    Base* objetoD = new Derivada(10);
    objetoB->funcao(); /// Chama o método da base.
    objetoD->funcao(); /// Chama o método da derivada.
}
```

Polimorfismo – Classes virtuais puras.

- Podemos, em C++, definir classes que nunca serão instanciadas, definindo somente a sua interface, a ser implementada nas sua classes derivadas.
- Isso é feito declarando métodos virtuais puros:

```
class Base
{
    public:
    Base(int);
    virtual void funcao() = 0; ///metodo da base.
};

class Derivada : /**public,private,protected*/ Base
{
    public:
    Derivada(int);
    void funcao(); /// redefinida na derivada.
};

int main()
{
    //Base* objetoB = new Base(10); // Não será mais possível,
                                    // pois a classe Base é virtual pura
    Base* objetoD = new Derivada(10);
    objetoD->funcao(); /// Chama o método da derivada.
}
```

Virtual

```
int main() {
      Passaro *p [3];
      p[0] = new Passaro ();
      p[1] = new Arara ();
      p[2] = new Arara ();
      for (int i = 0; i < 3; i++) {
            p[i]->canta();
      }
      for (int i = 0; i < 3; i++) {
            delete p[i];
      }
      return 0;
}
```

Substituição x refinamento

- Quando um método é redefinido, o código no método da classe derivada pode:
  - Substituir completamente o código do método da classe base (substituição)

  ou

  - Chamar o método da classe base e acrescentar a ele o código específico da classe derivada (refinamento)

Construtores sempre usam refinamento

- O construtor da classe base é acionado quando construímos a classe derivada.

- Desta forma garante-se que toda inicialização da classe base acontece também para os objetos da classe derivada.

Exemplo de refinamento

```
Arara::Arara():Passaro(){
	this->emExtincao = true;
	this->corPredominante = "azul";
}

Arara::~Arara() { }

void Arara::canta() {
	cout << "A-RA-RA --- A-RA-RA" << endl;
}
```

Variáveis polimórficas

```
#include <iostream>
#include <cstring>
using namespace std;
class A { … };
class B: public A { … };
int main() {
      A *a = new A();
      a = new B();
}
```

This

- this é a variável polimórfica mais comum!
- Ela pode receber diferentes classes!