

>>> Programação Orientada a Objetos (POO)

... De volta ao C++

Prof: André de Freitas Smaira

>>> Escopo

>>> **Escopo**

- * **Região de validade** de identificadores (variáveis, funções, classes, etc)
- * **Escopo local** (ou de bloco): escopo apenas nesse bloco (depois da definição) e nos blocos interiores

```
for(int i=0; i<10; i++) {  
    int a = 10;  
    std::cout << i << a << std::endl; // funciona  
}  
std::cout << i << std::endl; // não funciona!  
std::cout << a << std::endl; // não funciona!
```

>>> Escopo

* **Escopo de função:** utilizados dentro de uma função

```
#include <iostream>
```

```
void funcaoCorreta() {
```

```
    int x = 0;
```

```
    // Rótulo com escopo dentro da função
```

```
    inicio:
```

```
    std::cout << "Valor de x: " << x << std::endl;
```

```
    x++;
```

```
    if (x < 3) {
```

```
        goto inicio; // Correto: rótulo dentro da função
```

```
    }
```

```
}
```

```
int main() {
```

```
    funcaoCorreta(); // Funciona
```

```
    goto inicio; // ERRO!
```

```
    return 0;
```

```
[~]$ _ }
```

>>> Escopo

- * **Escopo de arquivo** (ou global): Identificadores declarados exteriormente a todos os blocos e classes. Podem ser utilizados em qualquer ponto desse arquivo após a declaração.

```
#include <iostream>
```

```
int contador = 0;
```

```
void incrementarContador() {  
    contador++;  
}
```

```
int main() {  
    incrementarContador();  
    std::cout << "Contador: " << contador << std::endl;  
    incrementarContador();  
    std::cout << "Contador: " << contador << std::endl;  
    return 0;  
}
```

>>> Escopo

- * **Escopo de classe:** Identificadores que declaram membros de classe. Somente podem ser utilizados em métodos da classe ou através de um objeto da classe com uso dos operadores de acesso a membro `.` ou `->` ou com uso do operador de resolução de escopo `::`.

```
#include <iostream>
#include <string>
```

```
class Pessoa {
private:
    std::string nome;
    int idade;
public:
    Pessoa(std::string nome, int idade)
    : nome(nome), idade(idade) {}

    void exibirInformacoes() const {
        std::cout << "Nome: " << this->nome << std::endl;
        std::cout << "Idade: " << this->idade << std::endl;
    }

    std::string getNome() const {
        return nome;
    }

    void setIdade(int novaIdade) {
        idade = novaIdade;
    }
};
```

```
int main() {
    Pessoa pessoa1("Maria", 30);
    pessoa1.exibirInformacoes();
    std::cout << "Nome: " << pessoa1.getNome()
               << std::endl;
    pessoa1.setIdade(31);
    pessoa1.exibirInformacoes();
    // std::cout << pessoa1.nome << std::endl; // Erro: nome é privado
    // pessoa1.idade = 32; // Erro: idade é privado
    return 0;
}
```

>>> **Escopo**

- * **Escopo de namespace**: Um conjunto de identificadores pode ser declarado em um espaço de escopo explicitamente distinto, utilizando a construção conhecida como **namespace**.
- * Os três últimos podem ser **acessados explicitamente**

```
>>> Escopo global
```

```
int x;
```

```
void f() {
```

```
    int y;
```

```
    y = x; // acessa x global
```

```
}
```

```
void g() {
```

```
    int x, y;
```

```
    y = x; // acessa x local
```

```
    y = ::x; // acessa x global
```

```
}
```

```
void h(int x) {
```

```
    int y;
```

```
    y = x; // acessa parametro x
```

```
    y = ::x; // acessa x global
```

```
}
```

```
[~]$ _
```



```
>>> Escopo de classe
```

```
class Data {  
    int dia, mes, ano;  
  
public:  
    /* ... */ // outros membros  
  
    enum Dia_da_semana {  
        domingo, segunda, terca, quarta, quinta, sexta, sabado  
    };  
  
    /* ... */ // outros membros  
};  
  
Data::Dia_da_semana ds;  
ds = Data::Dia_da_semana::segunda;
```

>>> **Escopo de namespace**

```
namespace Geometria {  
    namespace Bidimensional {  
        int calcularAreaQuadrado(int lado) {  
            return lado * lado;  
        }  
    }  
}  
  
namespace Tridimensional {  
    int calcularVolumeCubo(int lado) {  
        return lado * lado * lado;  
    }  
}
```

```
>>> Escopo de namespace
```

```
namespace Fisica {  
    namespace Mecanica {  
        double calcularVelocidade(double distancia, double tempo) {  
            return distancia / tempo;  
        }  
    }  
}
```

>>> Escopo de namespace

```
#include <iostream>

int main() {
    int areaQuadrado = Geometria::Bidimensional::calcularAreaQuadrado(5);
    std::cout << "Área do quadrado: " << areaQuadrado << std::endl;

    int volumeCubo = Geometria::Tridimensional::calcularVolumeCubo(3);
    std::cout << "Volume do cubo: " << volumeCubo << std::endl;

    double velocidade = Fisica::Mecanica::calcularVelocidade(100, 20);
    std::cout << "Velocidade: " << velocidade << std::endl;

    return 0;
}
```

>>> Escopo de namespace

Vai usar uma função com **muita frequência**?

```
#include <iostream>
```

```
int main() {  
    using Geometria::Bidimensional::calcularAreaQuadrado;  
    using Geometria::Tridimensional::calcularVolumeCubo;  
    using Fisica::Mecanica::calcularVelocidade;  
  
    int areaQuadrado = calcularAreaQuadrado(5);  
    std::cout << "Área do quadrado: " << areaQuadrado << std::endl;  
  
    int volumeCubo = calcularVolumeCubo(3);  
    std::cout << "Volume do cubo: " << volumeCubo << std::endl;  
  
    double velocidade = calcularVelocidade(100, 20);  
    std::cout << "Velocidade: " << velocidade << std::endl;  
  
    return 0;  
}
```

>>> Escopo de namespace

Vai usar um namespace com **muita frequência**?

```
#include <iostream>
```

```
using namespace std; // ;D
```

```
int main() {
```

```
    using Geometria::Bidimensional::calcularAreaQuadrado;
```

```
    using Geometria::Tridimensional::calcularVolumeCubo;
```

```
    using Fisica::Mecanica::calcularVelocidade;
```

```
    int areaQuadrado = calcularAreaQuadrado(5);
```

```
    cout << "Área do quadrado: " << areaQuadrado << endl;
```

```
    int volumeCubo = calcularVolumeCubo(3);
```

```
    cout << "Volume do cubo: " << volumeCubo << endl;
```

```
    double velocidade = calcularVelocidade(100, 20);
```

```
    cout << "Velocidade: " << velocidade << endl;
```

```
    return 0;
```

```
}
```

```
[~]$ _
```

```
>>> Namespace
```

- * Pode haver diversos namespaces com mesmo nome
- * Eles **se incrementam**

>>> Templates

>>> Generalização

- * Polimorfismo

- * **Ponteiro** para **classe base** consegue tratar de objetos das derivadas
- * **Geral** porém **restrito a classes**

- * Templates

- * **Generaliza** tipos não relacionados
- * Mas os **usos** são completamente **diferentes**

>>> Templates

```
double max(double a, double b)
{
    if( a >= b ) return a;
    else return b;
}
```

- * Essa função retorna o máximo entre **a** e **b**, sendo **ambos double**

- * Também funciona para **int** dentre outros por causa da **conversão automática**

- * Contas com **double são lentas**, então seria melhor executar uma função com **int** mesmo

- * Uma solução é **sobrecarga de nomes**

```
int max(int a, int b)
{
    if( a >= b ) return a;
    else return b;
}
```

>>> Templates

- * REDUNDÂNCIA

- * implementações **exatamente iguais**, só mudando os tipos

- * É aí que entram os **templates**

```
template<typename tipo>
tipo max(tipo a, tipo b)
{
    if( a >= b ) return a;
    else return b;
}
```

```
>>> Templates
```

```
* Template de função parametrizada por tipo (ou função genérica)
```

```
int i, j;
```

```
float a, b;
```

```
char c, d;
```

```
double e, f;
```

```
i = max(i, j); // int max(int, int)
```

```
a = max(a, b); // float max(float, float)
```

```
d = max(c, d); // char max(char, char)
```

```
e = max(e, f); // double max(double, double)
```

>>> Templates

- * O compilador gera uma função diferente para cada combinação de tipos
- * Tipos definidos pelo usuário podem ser utilizados

>>> Templates

* Esse código **vai funcionar?**

```
int k;  
double x, y;
```

```
y = max(x, k);
```

* E esse?

```
int k;  
double x, y;
```

```
y = max(x, (double)k);
```

* E esse?

```
int k;  
double x, y;
```

```
y = max<double>(x, k);
```

```
>>> Templates de classe
```

- * Podemos criar também **tipos parametrizados**
- * Vamos criar uma **classe de Pares** (tupla de dois elementos de python)

>>> Templates de classe

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
template <typename T>
```

```
class Par {
```

```
private:
```

```
    T _primeiro;
```

```
    T _segundo;
```

```
public:
```

```
    Par(T primeiro, T segundo)
```

```
    : _primeiro(primeiro), _segundo(segundo) {}
```

```
    void imprimir() const;
```

```
    T getPrimeiro() const { return _primeiro; }
```

```
    T getSegundo() const { return _segundo; }
```

```
};
```

```
template <typename T>
```

```
void Par<T>::imprimir() const {
```

```
    cout << "(" << _primeiro << ", " << _segundo << ")" << endl;
```

```
}
```


>>> Templates de classe

```
int main() {  
    Par<int> parInteiros(10, 20);  
    parInteiros.imprimir();  
    Par<string> parStrings("Olá", "Mundo");  
    parStrings.imprimir();  
    Par<double> parDoubles(3.14, 2.71);  
    parDoubles.imprimir();  
    return 0;  
}
```

>>> Templates de classe

Além de parametrizar por **tipo**, podemos parametrizar por **valor**

```
template<int N>
class Contador_circular : public Contador {
public:
    Contador_circular(int i = 0) : Contador(i) {}

    void ajusta_valor(int i);
    void avanca();
};

template<int N>
void Contador_circular<N>::ajusta_valor(int i) {
    _valor = i % N;
    if (_valor < 0) _valor += N;
}

template<int N>
void Contador_circular<N>::avanca() {
    _valor = (_valor + 1) % N;
}
```

>>> Templates de classe

Também podemos usar **valores assumidos**

```
template<typename T = int>
class Par {
    // Idem ao anterior
};
```

```
template<int N = 10>
class Contador_circular : public Contador {
    // Idem ao anterior
};
```

```
Par<double> pd; // pd guarda valores double
Par<> pi; // pi guarda valores int
Contador_circular<5> c5; // conta 5 passos
Contador_circular<> c10; // conta 10 passos
```

>>> Templates de classe

Um método de uma classe template também pode ser um template

```
template<typename T>
class Par {
    // Idem ao anterior
public:
    // Pares de tipos diferentes serão amigas
    template<typename T2>
    friend class Par;

    // Construtor de conversao
    // cria par do tipo T a partir de par do tipo T2
    template<typename T2>
    Par(Par<T2> const &s);
    // Restante identico
};

// Construtor de copia e conversao
template<typename T>
template<typename T2>
Par<T>::Par(Par<T2> const &s) {
    _primeiro = s._primeiro;
    _segundo = s._segundo;
}

Par<int> pi(3);
Par<double> pd(pi);
```

>>> Templates

- * Não se pode usar compilação separada
- * Declarar uma **classe como template**, declara implicitamente todos seus **métodos como template**

```
>>> Templates de classe
```

Para funções **friend**

```
template<typename T>
```

```
class Par {
```

```
    // ...
```

```
public:
```

```
    // ...
```

```
    friend T max<>(Par const &a);
```

```
    // ...
```

```
};
```

```
template<typename T>
```

```
T max(Par<T> const &a) {
```

```
    // ...
```

```
}
```

>>> Especialização de Templates

```
template<typename T>
T max(T a , T b)
{
    if( a > b ) return a;
    else return b;
}
```

Esse template não faz sentido para **T = const char***

```
template<>
const char* max<const char*>(const char* a, const char* b) {
    char const *theone;
    if (strcmp(a,b) > 0) theone = a;
    else theone = b;
    return strdup(theone);
}
```

É possível fazer herança de ou para template

>>> Exceções

>>> Exceções

Primeira utilidade: **separar código** de **tratamento de erro**

```
#include <iostream>
#include <stdexcept>
```

```
int main() {
    int numerador, denominador;
    std::cout << "Digite o numerador: ";
    std::cin >> numerador;
    std::cout << "Digite o denominador: ";
    std::cin >> denominador;

    try {
        if (denominador == 0)
            throw std::runtime_error("Divisão por zero!");
        double resultado = double(numerador) / denominador;
        std::cout << "Resultado da divisão: " << resultado << std::endl;
    } catch (const std::runtime_error& erro) {
        std::cerr << "Erro: " << erro.what() << std::endl;
    }

    return 0;
}
```

Mas é claro que poderíamos tratar com apenas um teste, **sem lançar exceção**

>>> Exceções

Mas nem sempre é **tão simples assim...**

```
//g++ teste4.cpp -lcpprest -lboost_system -lcrypto -std=c++11
```

```
#include <cpprest/http_client.h>
```

```
#include <iostream>
```

```
using namespace utility;
```

```
using namespace web;
```

```
using namespace web::http;
```

```
using namespace web::http::client;
```

```
int main() {
```

```
    try {
```

```
        http_client client(U("http://epicleet.team"));
```

```
        pplx::task<http_response> requestTask = client.request(methods::GET);
```

```
        http_response response = requestTask.get();
```

```
        if (response.status_code() == status_codes::OK)
```

```
            std::cout << response.extract_string().get() << std::endl;
```

```
        else
```

```
            std::cerr << "Erro HTTP: " << response.status_code() << std::endl;
```

```
    } catch (const http_exception& e) {
```

```
        std::cerr << "Erro na requisição: " << e.what() << std::endl;
```

```
    }
```

```
    return 0;
```

```
}
```

>>> Exceções

E esse com vários **erros do mesmo tipo?**

```
#include <stdio>
#include <stdlib>

int main() {
    FILE *in, *out1, *out2;

    in = fopen("input.txt", "r");
    if (in == NULL) {
        fprintf(stderr, "ERRO na abertura.\n");
        exit(1);
    }

    out1 = fopen("output1.txt", "w");
    if (out1 == NULL) {
        fprintf(stderr, "ERRO na abertura.\n");
        exit(2);
    }
```

```
    out2 = fopen("output2.txt", "w");
    if (out2 == NULL) {
        fprintf(stderr, "ERRO na abertura.\n");
        exit(2);
    }

    int result = fclose(in);
    if (result == EOF) {
        fprintf(stderr, "ERRO no fechamento.\n");
    }

    result = fclose(out1);
    if (result == EOF) {
        fprintf(stderr, "ERRO no fechamento.\n");
    }

    result = fclose(out2);
    if (result == EOF) {
        fprintf(stderr, "ERRO no fechamento.\n");
    }
}
```

Muitos testes de erros no meio do código para um programa que **não faz nada...**

>>> Exceções

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

FILE *open(const char *nome, const char *modo) {
    FILE *arq = fopen(nome, modo);
    if(!arq)
        throw std::runtime_error("ERRO na abertura");
    return arq;
}

void close(FILE *arq) {
    if (fclose(arq) == EOF)
        throw std::runtime_error("ERRO no fechamento");
}
```

```
int main() {
    FILE *in, *out1, *out2;

    try {
        in = open("input.txt", "r");
        out1 = open("output1.txt", "w");
        out2 = open("output2.txt", "w");
        close(in);
        close(out1);
        close(out2);
        return 0;
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}
```

>>> Exceções

Podemos criar **exceções personalizadas**

```
#include <iostream>
#include <cstdio>

using namespace std;

class File_error {};
class Open_error : public File_error {
    char const *_nome;
public:
    Open_error(char const *n="<<Desconhecido>>")
        : _nome(n) {}
    char const *_nome() const { return _nome; }
};
class Close_error : public File_error {};

FILE *open(const char *_nome, const char *_modo) {
    FILE *arq = fopen(_nome, _modo);
    if(!arq) throw Open_error(_nome);
    return arq;
}

void close(FILE *arq) {
    if (fclose(arq) == EOF) throw Close_error();
}
```

```
int main() {
    FILE *in, *out1, *out2;

    try {
        in = open_input("input.txt");
        out1 = open_output("output1.txt");
        out2 = open_output("output2.txt");
        close_file(in);
        close_file(out1);
        close_file(out2);
        return 0;
    }
    catch (Open_error const &e) {
        cerr << "Erro ao abrir arquivo " << e._nome() <<
    }
    catch (Close_error const &) {
        cerr << "Erro ao fechar arquivo" << endl;
    }
    catch (...) {
        cerr << "Erro desconhecido" << endl;
    }
    return 1;
}
```

Se quisermos poder tratar como **exceção gnérica**, precisa herdar de **std::exception**, ou usar ...

>>> Exceções

```
int main() {  
    FILE *in, *out1, *out2;  
  
    try {  
        in = open("input.txt", "r");  
        out1 = open("output1.txt", "w");  
        out2 = open("output2.txt", "w");  
        close(in);  
        close(out1);  
        close(out2);  
        return 0;  
    } catch (...) {  
        cerr << typeid(e).name() << endl;  
        cerr << e.what() << endl;  
    }  
}
```

Exceções **criadas** pelo usuário podem ter os **métodos** que quiser

>>> Entrada e Saída

>>> 0 que já vimos?

```
printf("0 resultado é %lf\n", x);  
cout << "0 resultado é " << x << endl;
```

Mas ficou claro que **não são impressos quando o comando é executado?**

```
cout << flush;  
cout << endl;
```



```
>>> Mas tem mais...
```

`cout` tem outros métodos

```
cout.put('x');
```

E pode ser feito em sequência

```
cout.put('x').put('\n');
```

* `cout.write(const char *str, int n)` imprime os `n` primeiros bytes de `str`

>>> Mas tem mais...

`cin` também tem outros métodos

- * `cin.eof()` verifica se chegou o fim do arquivo
- * `cin.get()` próximo caracter ou EOF
- * `cin.get(char &c)` salva o próximo caracter em `c`
- * `cin.get(char *buffer, int n, char sep='\n')` salva no máximo `n` caracteres em `buffer` a não ser que encontre `sep` antes
- * `cin.getline (char *, int, char = '\n')` faz o mesmo que o anterior, mas retirando o `sep` da entrada
- * `cin.ignore(int n=1, int end=EOF)` ignora no máximo `n` caracteres a não ser que encontre `end` antes
- * `cin.peek()` retorna o próximo caracter, mas sem tirá-lo
- * `cin.read(char *str, int n)` salva os próximos `n` bytes em `str`
- * `cin.gcount()` retorna o número de caracteres lidos após chamada de `read`, `get` ou `getline`

```
>>> iomanip
```

Manipuladores são usados assim:

```
cout << manipulador;
```

- * **dec, oct, hex, setbase(int)** determina em qual base serão escritos os inteiros
- * **setprecision(int)** determina a precisão dos números
- * **setw(int)** determina o espaço (número de caracteres) reservado para o próximo valor escrito
- * **setfill(char)** determina qual caracter vai ser usado no preenchimento quando a manipulação anterior for usada
- * **ws** extrai todos os caracteres **espaço** até encontrar um não espaço
- * **endl** insere um fim de linha e esvazia o buffer
- * **flush** esvazia o buffer

>>> Detalhes de Formatação

Podemos especificar detalhes de formatação

- * `cout.setf(ios::left, ios::adjustfield)` ativa a flag `left`
- * `cout.setf(ios::showpoint | ios::showpos)` ativa as flags `showpoint` e `showpos`
- * `cout << setiosflags(ios::showpoint | ios::showpos) ->`
alternativa ao anterior
- * `cout.flags(ios::showpoint | ios::showpos)` ativa as flags `showpoint` e `showpos` e desativa todas as outras
- * `setf`, `unsetf` e `flags` retornam o `estado anterior` para eventual restauração

>>> Todas as Flags

Flag	Significado
ios::skipws	brancos devem ser pulados
ios::right	saída ajustada à direita
ios::left	saída ajustada à esquerda
ios::internal	sinal ou base ajustados à esquerda, número ajustado à direita, espaço interno restante preenchido
ios::dec	base decimal (10)
ios::oct	base octal (8)
ios::hex	base hexadecimal (16)
ios::showbase	força o indicador de base de um número inteiro a ser impresso
ios::showpoint	força número de ponto flutuante a ser impresso com ponto decimal e todos os zeros finais especificados pela precisão atual
ios::uppercase	força todas as letras usadas para impressão de números (como X e E) a serem maiúsculas
ios::showpos	mostra sinal também para números positivos
ios::scientific	força a saída de um número de ponto flutuante em formato científico
ios::fixed	força a saída a utilizar um número específico de casas decimais após a vírgula
ios::unitbuf	streams são esvaziados após toda inserção (semelhante a cerr)
ios::stdio	streams de saída e erro padrão de C são esvaziados após toda inserção (para programas misturando C++ e C)

>>> Amarrando Entrada e Saída

Lembre-se de que você pode escrever pedindo uma entrada do usuário e **ele não receber a informação**

Para **cin** e **cout** isso já é resolvido. Para outros:

is.tie(&os)

>>> Em caso de arquivos

- * Tudo anteriormente citado funciona
- * `seekg(long)` muda o cursor de leitura para a posição especificada
- * `seekg(long, seek_dir)` muda o cursor de leitura para a posição especificada pelo primeiro parâmetro a partir do segundo (referência)

Referência	Descrição
<code>ios::beg</code>	valor em relação ao início do arquivo
<code>ios::cur</code>	valor em relação ao ponto corrente
<code>ios::end</code>	valor em relação ao final do arquivo

- * `seekp(long)` `seekp(long, seek_dir)` -> mesmo que o `seekg`, mas para escrita
- * `tellg()` retorna a posição atual de leitura
- * `tellp()` retorna a posição atual de escrita

>>> Entrada de string

```
#include <iostream>
#include <string>
#include <sstream>
int main() {
    char buffer[256];
    std::cout << "Dobraduras\nEscolha seus numeros.\n\n";
    while (true) {
        std::cout << ": ";
        std::cin.getline(buffer, 256);
        std::istringstream is(buffer);

        double val;
        is >> val;
        if (is.fail()) {
            std::istringstream is(buffer);
            std::string comando;
            is >> comando;
            if (comando == "sair")
                break;
            else
                std::cout << "\n0 que?\n";
        } else
            std::cout << 2 * val << std::endl;
    }
    return 0;
}
[~]$ _
```


>>> Saída para string

```
#include <iostream>
#include <sstream>
#include <string>

// Código mal-educado
int main() {
    std::string nome;
    std::ostringstream output;
    std::cout << "Quem esta ai?" << std::endl;
    std::cin >> nome;
    std::cout << std::endl;
    output << "Nao suporto a presenca de " << nome;

    std::string saida = output.str();
    std::cout << saida << "!" << std::endl;

    std::cout << std::endl << "Eu disse: \"" << saida
        << "\", ouviu?" << std::endl << std::endl;

    std::cout << "Como eh, nao vai quebrar o monitor?!" << std::endl;

    return 0;
}
```

>>> Referências

* Apostila e Aulas do **Gonzalo Travieso** (IFSC/USP)