

# Estruturas de Dados

A. G. Silva, A. von Wangenheim, J. E. Martina

14 de agosto de 2018

- Declaração de classes semelhante ao Java (Java se inspirou no C++);
- As melhores práticas são declarar no arquivo *header* e implementar no *source*;
- Isso tem por objetivo separar instrumentação do compilador de código objeto;
- Nada impede a implementação de métodos no *header*;
- Templates vão exigir a implementação dentro do *header* pois o pré-processador do compilador deve criar as variações.

- Pessoa.h

```
class Pessoa {  
    private:  
        // declaracao dos atributos privados  
    protected:  
        // declaracao dos metodos protegidos  
    public:  
        // declaracao dos metodos publicos  
}
```

- Pessoa.cpp

```
#include "Pessoa.h"

void Pessoa::metodo1() {
    // implementacao do metodo 1
}

int Pessoa::metodo2() {
    // implementacao do metodo 2
}
```

# Construtores e destrutores

```
// Pessoa.h
class Pessoa {
    private:
        // declaracao dos atributos privados
    protected:
        // declaracao dos metodos protegidos
    public:
        Pessoa(); // construtor padrao
        Pessoa(char n[], int i); // com parametros
        virtual ~Pessoa(); // destrutor
}
```

# Construtores e destrutores

```
// Pessoa.cpp
#include "Pessoa.h"

Pessoa::Pessoa() {
    // implementacao
}

int Pessoa::Pessoa(char* n, int i) {
    // implementacao
}

int Pessoa::~~Pessoa() {
    // implementacao
}
```

# Construtores e destrutores

Usando os construtores e destrutores:

```
Pessoa p; // construtor padrao e invocado  
Pessoa p("Fulano", 27); // outro construtor  
p.~Pessoa(); // destrutor
```

```
Pessoa* p = new Pessoa(); // dinamico  
Pessoa* p = new Pessoa("Fulano", 27); // dinamico  
delete p; // dinamico
```

# Herança em C++

```
Class Empregado : public Pessoa {  
    // herda atributos e metodos da superclasse igual  
    em Java  
}  
Class Empregado : protected Pessoa {  
    // herda atributos e metodos da superclasse com  
    visao so para classe e para seus filhos  
}  
Class Empregado : private Pessoa {  
    // herda atributos e metodos da superclasse com  
    visao somente interna  
}
```

- Informações mais detalhadas podem ser consultadas na seguinte [wiki](#)



# Métodos virtuais

```
Class Empregado : public Pessoa {  
    private:  
        int _salario;  
        int _descontos;  
    public:  
        virtual int salarioLiquido() = 0;  
}
```

# Métodos virtuais

```
Class Estudante : public Empregado {  
    private:  
    public:  
        virtual int salarioLiquido() {  
            return _salario;  
        };  
        virtual void adicionaBonus() {  
            _salario = _salario + 0;  
        }  
}
```

# Métodos virtuais

```
Class Professor : public Estudante {  
    private:  
    public:  
        int salarioLiquido() {  
            return _salario * 0.725;  
        };  
        void adicionaBonus() {  
            _salario = _salario + 5000;  
        }  
}
```

# Classes abstratas e interfaces

- Classe abstrata possui ao menos um método puramente virtual (sem implementação);
- Interface possui todos os métodos puramente virtuais;
- Em C++, tudo se dá por herança múltipla;
- Isso causa problemas de ambiguidade como o “Problema do Diamante”;
- A solução é utilizar herança virtual (veja um exemplo em [wiki](#)):
  - `virtual public;`
  - `virtual protected;`
  - `virtual private.`

# Tratamento de exceções

- O tratamento de exceções permite capturar erros ocorridos durante a execução de um programa;
- Uma exceção é lançada com o comando `throw()`;
- O comando `throw()` deve ser usado dentro de um bloco `try {...}`;
- O fluxo de execução é desviado para o bloco `catch()`.

# Tratamento de exceções

```
// exceptions
#include <iostream>
using namespace std;

int main () {
    try {
        throw 20;
    }
    catch (int e) {
        cout << "An exception occurred. Exception Nr. "
              << e << '\n';
    }
    return 0;
}
```

# Exceções padronizadas no C++

```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception {
    virtual const char* what() const throw() {
        return "My_exception_happened";
    }
} myex;

int main () {
    try {
        throw myex;
    } catch (exception& e) {
        cout << e.what() << '\n';
    }
    return 0;
}
```

# Templates

- A utilização de *templates* nos permite criar funções e classes genéricas;
- As funções ou classes genéricas são baseadas em argumentos com tipo desconhecido;
- O tipo será definido quando forem usadas por entidades específicas.



# Templates

```
template<typename T>
class Lista {
    private:
        Lista *proximo;
        T *dado;
    public:
        T retornaDoInicio();
};

int main() {
    Lista<int> listaInteiro;
    Lista<char*> listaString;
}
```

# Documentação com Doxygen

- Documentação de referência a partir de um conjunto de códigos documentados;
- Suporte para saída em RTF (Word), PostScript, PDF com links, HTML compactado e man pages do Linux.;
- A documentação é extraída diretamente dos códigos;
- Você pode também gerar automaticamente as relações entre os vários elementos, o que inclui gráficos de dependência, diagramas de herança e diagramas colaborativos.

# Documentação com Doxygen

```
///! Uma classe de teste.  
/*!  
    Uma descricao mais elaborada da classe.  
*/  
class Teste {  
    public:  
        ///! Uma enumeracao.  
        /*! Descricao mais detalhada da enumeracao. */  
        enum TEnum {  
            TVal1, /*!< Valor enumeravel TVal1. */  
            TVal2, /*!< Valor enumeravel TVal2. */  
            TVal3  /*!< Valor enumeravel TVal3. */  
        }  
}
```

# Documentação com Doxygen

```
    //!< Ponteiro da enumeracao.
    /*! Detalhes. */
    *ptrEnum,
    //!< Variavel da enumeracao.
    /*! Detalhes. */
    varEnum;
//! Um construtor.
/*!
    Descricao mais elaborada do construtor.
*/
Teste();

//! Um destrutor.
/*!
    Descricao mais elaborada do destrutor.
*/
~Teste();
```

# Documentação com Doxygen

```
//! Um membro normal com dois argumentos e
    retornando um valor inteiro.
/*!
    \param a um argumento inteiro.
    \param s um ponteiro para char const.
    \return Os resultados do teste
    \sa Teste(), ~Teste(), meTesteTambem() and
        varPublica()
*/
int meTeste(int a, const char *s);
//! Um membro puramente virtual.
/*!
    \sa testMe()
    \param c1 o primeiro argumento.
    \param c2 o segundo argumento.
*/
virtual void meTesteTambem(char c1, char c2)=0;
```

# Documentação com Doxygen

```
    //! Uma variavel publica.  
    /*!  
        Detalhes.  
    */  
    int varPublica;  
  
    //! Uma variavel funcao.  
    /*!  
        Detalhes.  
    */  
    int (*handler)(int a,int b);  
};
```

Perguntas?





Este trabalho está licenciado sob uma Licença Creative Commons Atribuição 4.0 Internacional. Para ver uma cópia desta licença, visite

<http://creativecommons.org/licenses/by/4.0/>.

