

SCC0504 - PROGRAMAÇÃO ORIENTADA A OBJETOS

Linguagem C++

Prof. Jose Fernando Rodrigues **Junior**

<http://www.icmc.usp.br/~junio>

junio@icmc.usp.br

Slides: Prof. Marcelo Manzato

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO - USP -

C++



- Linguagem desenvolvida no início da década de **1980** por **Bjarne Stroustrup**
- Ideia: **estender a linguagem C** a fim de incluir conceitos de orientação a objetos
- É (aproximadamente) um **superconjunto da linguagem C**
- **Java** apresenta **similaridades com C e C++**, mas é uma linguagem puramente orientada a objetos

C++

- Compilador para este curso:
 - ▣ GNU Compiler Collection (GCC)
 - ▣ Front-end para C++: **g++**
 - ▣ Exemplo: `g++ -o myProgram main.cpp`

- Pode-se utilizar várias IDEs:
 - ▣ Netbeans
 - ▣ QT
 - ▣ DevC++
 - ▣ <http://www.codeblocks.org/downloads>

C++ versus Java



- C++ mantém a filosofia de C de que **desempenho é crítico**
- Programas em **Java são geralmente mais lentos** do que programas em C++
 - ▣ Por que?
- **C++ é uma linguagem mais complexa**

C++ e Java: Semelhanças

- Comandos de **decisão e de repetição** são os mesmos (são baseados na linguagem C)
- **Operadores** são na grande maioria os mesmos
- **Type-casting** é utilizado em ambas (funcionalidade também proveniente de C)
- Tratamento de **exceções** é bastante semelhante

C++ e Java: Diferenças

- Java tem o tipo `boolean`, **C++ tem o tipo `bool`**
 - ▣ Ambos aceitam `true` ou `false`
 - ▣ Mas C++ aceita 1 como verdadeiro e 0 como falso (exatamente como em C)
- Em C++, **`main()` não precisa fazer parte de uma classe:**

```
int main() {  
    . . .  
}
```

C++ e Java: Diferenças

- Em C++ é possível escrever **funções e declarar variáveis que não fazem parte de classe alguma** (exatamente como em C)
 - ▣ Evite variáveis globais
- Em Java, toda classe é implicitamente derivada de **Object**. Em C++, se nenhuma superclasse for especificada, efetivamente não há superclasse
- C++ aceita o modificador `unsigned`, Java não

C++ e Java: Diferenças

- ❑ O intervalo aceito por **tipos numéricos em C++** é **dependente de plataforma**
- ❑ **Strings em C++** são **formadas por caracteres ASCII**, não Unicode
- ❑ Não há **interfaces em C++**
- ❑ **C++ aceita herança múltipla**
- ❑ Utiliza-se **const** para especificar **constantes em C++**
 - ❑ `const int DAYS_PER_YEAR = 365;`

C++: Primeiro programa



```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

C++: Entrada/saída padrão

- Os fluxos padrões de entrada e saída são representados pelos objetos `cin` e `cout`, respectivamente

```
double x;  
cout << "Please enter x: ";  
cin >> x;  
cout << "x is " << x;
```

C++: Classes – Arquivo .h

```
class Point {  
public:  
    Point();  
    Point(double xval, double yval);  
    void move(double dx, double dy);  
    double getX() const;  
    double getY() const;  
private:  
    double x;  
    double y;  
};
```

Modificadores de acesso public/protected/private
são separados em seções

A classe pode conter
apenas os protótipos dos
seus métodos

A palavra-chave const evita que os atributos
da classe sejam alterados pelo método

Deve aparecer um ponto-e-vírgula
após a chave

C++: Classes – Arquivo .cpp

- A implementação dos métodos aparece dentro da classe (**inline**) ou após a definição da classe:

```
Point::Point() {  
    x = 0;  
    y = 0;  
}
```

```
void Point::move(double dx, double dy) {  
    x = x + dx;  
    y = y + dy;  
}
```

```
double Point::getX() const {  
    return x;  
}
```

C++: Classes



- ❑ **Não é possível especificar a visibilidade da classe**
(private, public, protected)

C++: Arquivo .cpp correspondente

- O **arquivo .h** deve ser incluído no **arquivo .cpp** através da diretiva de compilação **#include**
- **Todos os métodos declarados na classe devem ser implementados no arquivo .h**
- Além disso, **as informações que serão usadas apenas internamente pela classe e não serão necessitadas por usuários da classe devem ser colocadas nesse arquivo**
 - ▣ **Classes privadas podem ser criadas dessa maneira**

C++: Arquivo .h

```
#ifndef POINT_H
```

```
#define POINT_H
```

```
class Point {
```

```
public:
```

```
    Point();
```

```
    Point(double xval, double yval);
```

```
    void move(double dx, double dy);
```

```
    double getX() const;
```

```
    double getY() const;
```

```
private:
```

```
    double x;
```

```
    double y;
```

```
};
```

```
#endif
```

No arquivo com extensão .h diretivas de compilação devem ser colocadas de forma a evitar múltiplas inserções do mesmo arquivo

C++: Objetos

- Variáveis de **objeto em C++ armazenam valores**, não referências
- **Para construir um objeto**, basta fornecer os parâmetros do construtor depois do nome do objeto:

```
Point p(1, 2);
```

- Se os parâmetros não forem fornecidos, o construtor padrão será utilizado

C++: Objetos

- `Point p;`
- **Em Java, esse comando apenas cria uma referência não inicializada a um objeto da classe Point**
- **Em C++, um objeto é construído utilizando o construtor padrão e `p` armazena o estado do objeto**

C++: Objetos

- Em C++, atribuições realmente copiam objetos

```
Point p(1, 2);
```

```
Point q = p;
```

```
q.move(1, 1); /*soma a x e y*/
```

```
cout << p.getX() << "," << p.getY() << endl;
```

```
cout << q.getX() << "," << q.getY();
```

- O que será impresso em C++?

1,2

2,3

- O que seria impresso em Java?

2,3

2,3

C++: Objetos

- Portanto, o comportamento visto em Java é realizado em C++ por meio de ponteiros

C++:

```
Point p(1, 2);
```

```
Point *q = &p;
```

```
q->move(1, 1);
```

Java:

```
Point p = new Point(1, 2);
```

```
Point q = p;
```

```
q.move(1, 1);
```

```
cout << p.getX() << "," << p.getY() << endl;
```

```
cout << q->getX() << "," << q->getY();
```

- Imprime:

2,3

2,3

Exercício

- ❑ **Crie uma classe Data** que permita armazenar e recuperar **dia, mês e ano**
- ❑ **Separe as declarações da classe das respectivas implementações de métodos** (arquivo .h e .cpp)
- ❑ **Implemente um método `toString()` que retorna uma `string` representativa da data (DD/MM/AAAA)**
- ❑ **Crie um programa que usa essa classe**

```
//Arquivo Data.h
#include <iostream>
#include <string>
#include <sstream>
```

```
class Data {
```

```
public:
```

```
    Data(int, int, int);
```

```
    int getDia();
```

```
    int getMes();
```

```
    int getAno();
```

```
    string toString();
```


```
private:
```

```
    int dia;
```

```
    int mes;
```

```
    int ano;
```

```
};
```




Alternativamente, é possível especificar valores padrão
para parâmetros de construtores caso nenhum
argumento seja fornecido:

```
Data(int dia = 1, int mes = 1, int ano = 1900);
```

```
//Arquivo Data.cpp
```

```
#include "Data.h"
```

```
Data::Data(int dia, int mes, int ano) {  
    this->dia = dia;  
    this->mes = mes;  
    this->ano = ano;  
}
```



Note o uso do this.
Ele é um ponteiro para
o objeto atual.

```
int Data::getDia() {  
    return dia;  
}
```

```
int Data::getMes() {  
    return mes;  
}
```

```
int Data::getAno() {  
    return ano;  
}
```



```
//Continuação de Data.cpp
```

```
string Data::toString() {  
    stringstream sstm;  
    sstm << dia << "/" << mes << "/" << ano;  
    return sstm.str();  
}
```

```
//Arquivo Main.cpp
```

```
#include <iostream>
```

```
#include "Data.h"
```

```
using namespace std;
```

```
int main() {
```

```
    Data tmp(15,6,2012);
```

```
    cout << tmp.getDia() << "/" << tmp.getMes() << "/"
```

```
        << tmp.getAno() << endl;
```

```
    cout << tmp.toString();
```

```
    return 0;
```

```
}
```


C++: Alocação dinâmica

- Em C++ **não há Garbage Collection**
- Para alocar memória, **use new**:
 - ▣ `Point *pnt = new Point(4,3);`
 - ▣ `int *intArray = new int[10];`
- Para liberar memória, **use delete**:
 - ▣ `delete pnt;`
 - ▣ `delete[] intArray;`
- **Análogo** a (e mais simples que) **malloc/free de C**
- Quando utilizar new, **sempre libere memória** (após uso) com delete

C++: Destrutor

- Se desejar realizar alguma operação no momento da liberação de memória de um objeto, **utilize o destrutor**:
- Notação:
`~MyClass();`

```

class Vector {
private:
    int sz;
    int* v;
public:
    Vector(int);
    ~Vector();
};

Vector::Vector (int s) {
    if (s <= 0)
        s = 10;

    sz = s;
    v = new int[s];
    cout << "Construtor\n";
}

Vector::~~Vector() {
    delete[] v;
    cout << "Destrutor\n";      /*Como há Garbage Collector em Java,
    não há destrutores*/
}

int main() {
    Vector v(10);
}

```

C++: Ponteiros

Exemplos de operações com ponteiros:

```
Point *p = NULL;  
Point *q = new Point(11,32);  
Point *r = q;  
Point  t(21,55);  
Point *s = &t;  
  
(*q).move(-4,2);  
q->move(-4,2);
```

C++: Ponteiros

- Diferentemente de Java, **C++ não verifica em tempo de execução se o ponteiro referencia NULL**
- **C++ também não acusará erro se você tentar acessar o conteúdo referenciado por um ponteiro caso esse conteúdo já tenha sido liberado da memória**
- Portanto, o funcionamento é como na linguagem C: **Gerenciar a alocação e acesso correto à memória é responsabilidade do programador**

C++: Ponteiros

- **D** **e** **de**
- **C** **o** **r** **o**
c
c
Existe a possibilidade do seu programa estar acessando **uma posição de memória cujo conteúdo não foi especificado** e sua aplicação **continuará funcionando**, muito embora **com resultados imprevisíveis** (não determinísticos).
- **P** **G**
r
Cada vez que se executar a aplicação se terá um comportamento diferente → problema difícil de identificar em muitos casos.
- **r**
responsabilidade do programador

Passagem de argumentos

- Por padrão, argumentos são passados por **valor (cópia)** a funções/métodos
- Para que alterações em argumentos sejam visíveis fora da função, é necessário passá-los por **referência (cópia apenas do endereço)**
- Em C++ isso é feito por meio do operador **&**

Passagem por valor

□ Exemplo:

```
void foo(int pValue) {  
    pValue = 6;  
}
```

```
int n = 5;  
foo(n);  
cout << n;
```

□ Saída:

5

Passagem de endereço

- Exemplo (passagem por referência em C):

```
void foo(int *pValue) {  
    *pValue = 6;  
}
```

```
int n = 5;  
foo(&n);
```

```
cout << n;
```

➔ Saída:

6

- Dentro da função, o conteúdo armazenado no endereço passado é alterado, mesmo tendo sido declarado fora da função

Passagem por referência

□ Exemplo:

```
void AddOne(int &y) {  
    y = y + 1;  
}
```

```
int x = 1;  
AddOne(x);  
cout << x;
```

□ Saída:

2

- Quando a função for chamada, y torna-se uma referência ao argumento

Passagem por referência

- ❑ **Referências são como aliases de uma variável.**
- ❑ Uma referência **não pode ser null**, e deve ser inicializada.
- ❑ Causa confusão o fato de **que há dois usos distintos** para o operador &:

1) `int x = 10;`

`int *p = &x;` /*p recebe o endereço de x, e pode receber outro endereço a qualquer momento*/

❑ 2) `int x = 10;`

`int y = 20;`

`int &p;` /*erro, referências precisam ser inicializadas*/

`int &p = x;` /*ok, p tem o mesmo endereço de x,
e não pode receber outro endereço*/

- ❑ Este segundo uso é o mesmo que se faz quando se faz passagem por referência para procedimento/função. Tentar interpretar a passagem por referência segundo o uso número 1) acima, causa apenas confusão. São usos distintos.

Passagem por referência

Mais sobre referências x ponteiros:

<http://www.cplusplus.com/articles/ENywwCM9/>

- Este segundo uso é o mesmo que se faz quando se faz passagem por referência para procedimento/função. Tentar interpretar a passagem por referência segundo o uso número 1) acima, causa apenas confusão. São usos distintos.

Passagem por referência

- **Passar argumentos por valor acarreta em cópia dos argumentos** para uso na função
- **Se o processo de cópia for muito custoso, é vantajoso passar por referência**, assim a cópia não será realizada
- **Mas, e se não quisermos que o argumento passado por referência seja alterado dentro da função?**
 - ▣ Adiciona-se o modificador **const**
- **Exemplo:** `void foo(const int &y) { ... }`

Membros static

- ❑ Cada objeto de uma classe tem sua própria cópia de todos os atributos da classe
- ❑ Em certos casos, deseja-se que todos os objetos de uma classe compartilhem somente uma cópia de uma variável
- ❑ Para isso usa-se atributos `static` (como em Java)

Membros static

- **Se um método for `static`, ele pode ser chamado mesmo se nenhum objeto tenha sido criado para a classe – assim com em Java**

- Para chamar um método `static`:

```
NomeDaClasse::NomeDoMétodo();
```

Sobrecarga de Operadores

- É o processo de usar operadores básicos da linguagem para manipular objetos
- Embora C++ não permita criar novos operadores, ela permite que a maioria dos operadores existentes sejam sobrecarregados
- Assim, quando esses operadores forem usados com objetos, eles terão funcionalidades próprias definidos para as respectivas classes

Sobrecarga de Operadores

- Os operadores são sobrecarregados escrevendo-se uma definição de função (com um cabeçalho e corpo) usando-se como nome da função a palavra-chave **operator** seguida pelo **símbolo** do operador que se deseja sobrecarregar

- Exemplo:

```
MyClass MyClass::operator+(const MyClass&);
```

→ Neste exemplo, o operador + recebe como parâmetro um objeto da classe do tipo MyClass, faz uma operação consigo mesmo (tipo MyClass) e retorna um objeto também desta classe

```

#include <iostream>

using namespace std;

class complx {
private:
    double real, imag;
public:
    complx(double real = 0.,
           double imag = 0.);
    complx operator+(const complx&) const;
    void print();
};

complx::complx(double r, double i) {
    real = r;
    imag = i;
}

complx complx::operator+ (const complx& c) const
{
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result;
}

```

```

void complx::print() {
    cout << real << "+" << imag << "*i";
}

int main() {
    complx x(3, 4);
    complx y(6, 7);
    complx z = x + y;

    z.print();
}

```

Sobrecarga de Operadores

- Nem todos os operadores em C++ podem ser sobrecarregados
- Lista dos operadores que se encaixam nesse grupo:
.
. *
::
?:

Sobrecarga de Operadores

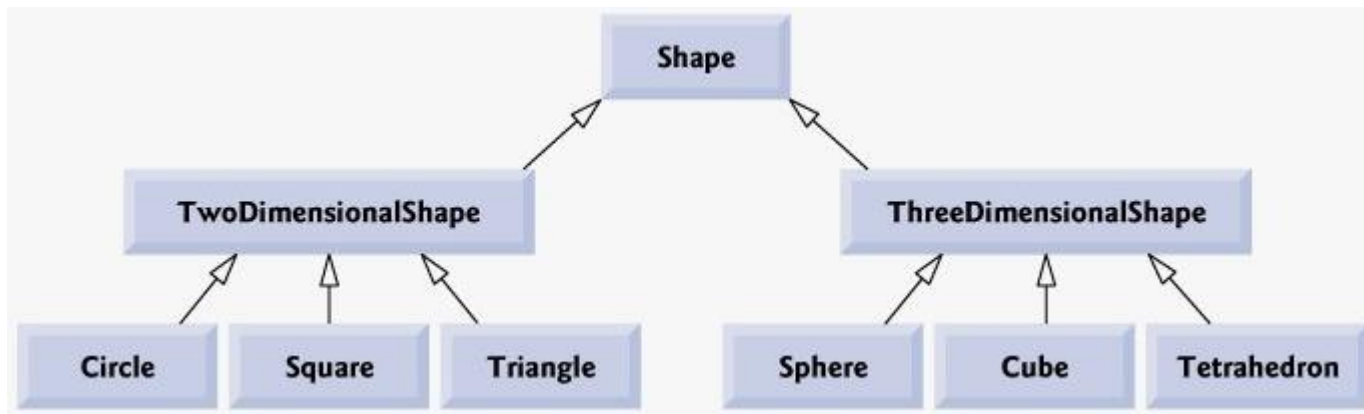
- A sobrecarga **não deve mudar o significado** de como um operador funciona com objetos de **tipos primitivos**
- Funciona **somente com objetos de tipos definidos pelo usuário**, ou com uma mistura de um objeto de um tipo definido pelo usuário e um tipo primitivo

Herança

- Em C++ **não existe a palavra-chave *extends***, como em Java
- C++ suporta **herança múltipla**, ao contrário de Java
- A sintaxe para herança é:

```
class <classe_derivada> : <tipo_de_acesso1> <classe_base1>,  
                        <tipo_de_acesso2> <classe_base2>,  
                        ...
```

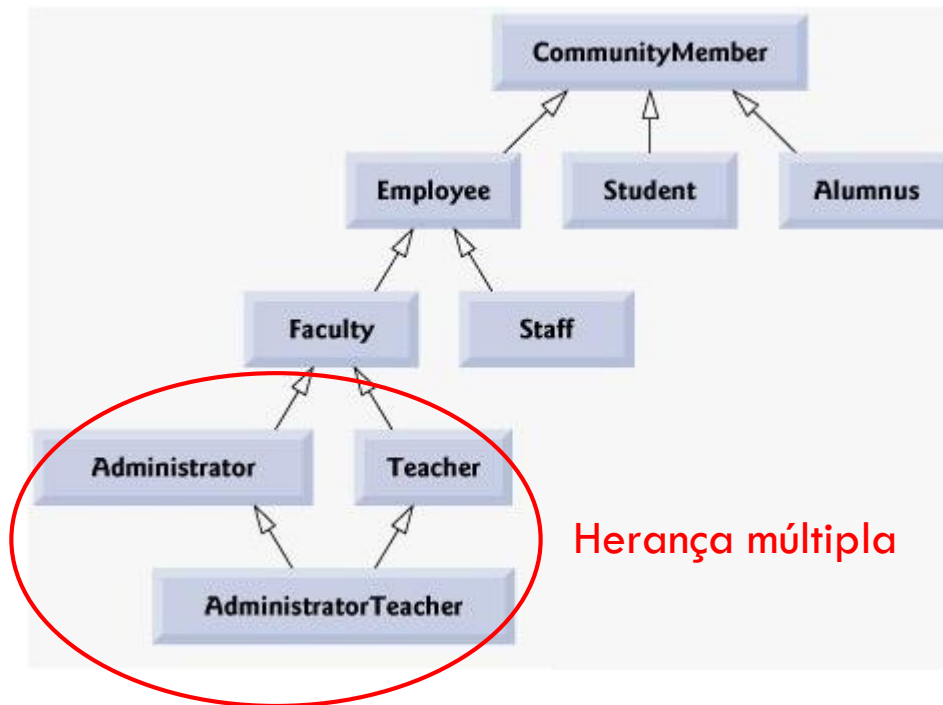
Herança



Exemplo: Cabeçalho da declaração da classe **TwoDimensionalShape**

```
class TwoDimensionalShape : public Shape
```

Herança



Herança

- Exemplo: Considere a seguinte classe base

```
class Shape {  
    public:  
        void setWidth(int w) {  
            width = w;  
        }  
        void setHeight(int h) {  
            height = h;  
        }  
    protected:  
        int width;  
        int height;  
};
```


- Podemos criar uma subclasse da seguinte maneira:

```
class Rectangle: public Shape {  
    public:  
        int getArea() {  
            return (width * height);  
        }  
};
```

Note o uso de um
especificador de acesso
para a herança

- Exemplo de uso:

```
int main(void) {  
    Rectangle Rect;  
  
    Rect.setWidth(5);  
    Rect.setHeight(7);  
  
    cout << "Total area: " << Rect.getArea() << endl;  
  
    return 0;  
}
```

□ Exemplo de uso do construtor da classe base na classe derivada:

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

class MyBaseClass {
public:
    MyBaseClass( string msg ) {
        cout << "Construtor da classe base: " << msg;
    }
};

class MyDerivedClass : public MyBaseClass {
public:
    MyDerivedClass( string msg ) : MyBaseClass( msg ) {
        // ...
    }
};

int main() {
    MyDerivedClass ex("Ola, sou um argumento do construtor da classe derivada!");
}
```

Herança

- Uma classe derivada não pode acessar diretamente os membros **private** da classe base
- Membros **protected** de uma classe base podem ser acessados dentro do corpo da classe derivada
- Quando um método da classe derivada redefine um método da classe base, o método da classe base pode ser acessado a partir da classe derivada utilizando o seguinte formato:

```
<NomeDaClasseBase>::<NomeDoMétodo> /*equivalente ao super do Java*/
```

Herança

- Tipos de herança (especificador de acesso):

`class <classe_derivada>: <tipo_de_acesso> <classe_base>`

- `<tipo_de_acesso>`:

- **Public:** Membros public, protected e private da classe base mantêm seus modificadores de acesso na classe derivada
- **Protected:** Membros public da classe base tornam-se membros protected da classe derivada
- **Private:** Membros public e protected da classe base tornam-se membros private da classe derivada

Herança

- Estude o exemplo:
“CppHeranca”, disponível no TIDIA-Ae

Polimorfismo

- *Relembrando:*

Polimorfismo permite escrever programas que processam objetos de classes que fazem parte da mesma hierarquia como se fossem objetos da classe base

- Em C++, faz-se uso de métodos **virtuais**

- Por exemplo, uma classe chamada Shape pode ter o seguinte método:

```
virtual void draw();
```

Sendo que se o método draw for sobrescrito em classes derivadas de Shape (como Circle, Rectangle, Square), teremos que o comportamento polimórfico será possível graças ao tipo virtual do método draw

Polimorfismo

- Se o seu programa chama um método virtual a partir de um objeto da classe derivada (objeto este referenciado por um ponteiro para a classe base), o programa escolherá o método em tempo de execução baseando-se no tipo do objeto referenciado (**dynamic binding** ou **late binding**)
 - Note que é necessário utilizar ponteiros!

Polimorfismo

- **Se o método não for virtual, o método a ser executado será escolhido de acordo com o tipo do ponteiro (e não do objeto instanciado) em tempo de compilação (*static binding*)**
 - ▣ Dessa maneira, o comportamento polimórfico não será obtido
 - ▣ De maneira semelhante, se não for utilizado ponteiro (mesmo com métodos virtuais), o comportamento polimórfico também não será obtido

Polimorfismo

- Estude o exemplo:

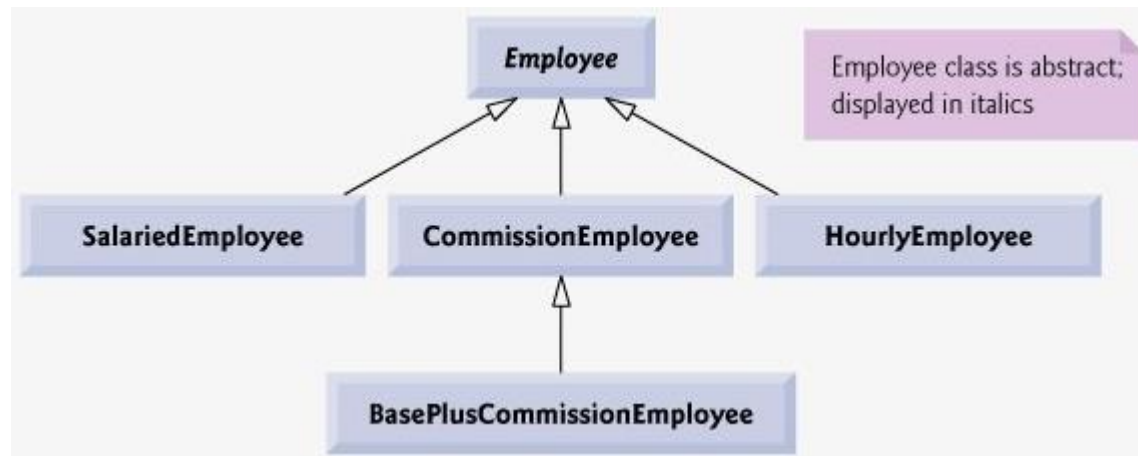
“[CppPolimorfismo e Herança I](#)”, disponível no TIDIA-Ae

Classes Abstratas

- Uma classe abstrata define uma interface pública comum a diversas classes em uma hierarquia
- Uma classe abstrata contém um ou mais métodos **virtuais puros** que devem ser sobrescritos por subclasses concretas
- Um método virtual puro tem "**= 0**" especificado em seu protótipo:

```
virtual void draw() = 0;
```
- Em Java, seria: `abstract void draw();`
- Embora não seja possível instanciar objetos de classes abstratas, é possível declarar ponteiros para essas classes a fim de obter comportamento polimórfico

Herança, Classes Abstratas e Polimorfismo



Estude “**CppPolimorfismo e Herança2**” disponível no TIDIA-Ae

Outros Tópicos Importantes

- I/O estilo C++
- Tratamento de Exceções
- Templates (programação genérica)
- STL (Standard Template Library)
 - ▣ Containers
 - deque, list, map, queue, set, stack, vector, etc
 - ▣ Algoritmos
 - `binary_search`, `copy`, `max`, `merge`, `replace`, `sort`, etc
 - ▣ Iterators
- E mais...

C++: Entrada/saída padrão

- Para imprimir em diferentes bases numéricas:

```
int numero = 10;  
cout << hex;  
cout << numero;
```

- **Base 16:** cout << hex;
- **Base 10:** cout << dec;
- **Base 8:** cout << oct;

C++: Entrada/saída padrão

□ Leitura de linhas:

```
char name[256], title[256];

cout << "Enter your name: ";
cin.getline (name, 256);

cout << "Enter your favourite movie: ";
cin.getline (title, 256);

cout << name << "'s favourite movie is " << title;
```

Namespaces

- **Namespaces** permitem agrupar entidades como classes, variáveis e funções
- **O escopo global pode então ser agrupado em escopos menores, cada qual com seu nome**

```
namespace identificador {  
    entidades  
}
```

Namespaces

□ Exemplo:

```
namespace myNamespace {  
    int a, b;  
}
```

```
int main() {  
  
    myNamespace::a = 1;  
    myNamespace::b = 2;  
  
    return 0;  
}
```


Namespaces

□ Exemplo:

```
#include <iostream>

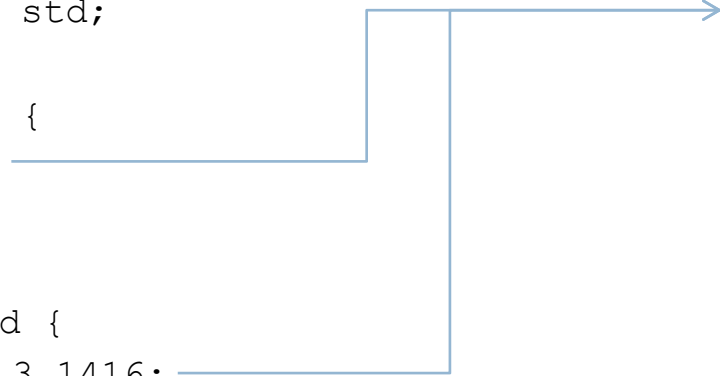
using namespace std;

namespace first {
    int var = 5;
}

namespace second {
    double var = 3.1416;
}

int main () {
    cout << first::var << endl;
    cout << second::var << endl;

    return 0;
}
```



Note o uso de dois
identificadores iguais
em namespaces
diferentes

□ Exemplo:

```
#include <iostream>
```

```
using namespace std;
```

```
namespace first {  
    int x = 5;  
    int y = 10;  
}
```

```
namespace second {  
    double x = 3.1416;  
    double y = 2.7183;  
}
```

```
int main () {  
    using first::x;  
    using second::y;  
    cout << x << endl;  
    cout << y << endl;  
    cout << second::x << endl;  
    cout << first::y << endl;  
    return 0;  
}
```

A palavra-chave **using** pode ser empregada para evitar o uso frequente de identificadores de namespaces

Exemplo:

```
#include <iostream>

using namespace std;

namespace first {
    int x = 5;
    int y = 10;
}

namespace second {
    double x = 3.1416;
    double y = 2.7183;
}

int main () {
    using namespace first;
    cout << x << endl;
    cout << y << endl;
    cout << second::x << endl;
    cout << second::y << endl;
    return 0;
}
```

A palavra-chave **using** também pode ser empregada para introduzir um namespace completo

A biblioteca padrão declara todas suas entidades no namespace **std**

□ Exemplo:

```
#include <iostream>
```

```
using namespace std;
```

```
namespace
```

```
int
```

```
int
```

```
}
```

```
namespace
```

```
doubl
```

```
doubl
```

```
}
```

```
int ma
```

```
using namespace first;
```

```
cout << x << endl;
```

```
cout << y << endl;
```

```
cout << second::y << endl;
```

```
cout << second::x << endl;
```

```
return 0;
```

```
}
```

A palavra-chave **using** também pode ser empregada para introduzir um namespace completo

Documentação da standard library:

http://www.sgi.com/tech/stl/table_of_contents.html

Não se compara ao Javadoc

A palavra-chave **using** declara todas suas entidades no namespace **std**

Strings

- Você pode **utilizar strings C tradicionais**
 - ▣ Vetores de char terminados em `'\0'`
- Ou utilizar a classe **string**
- Exemplos:

```
//construtor padrão, string vazia  
string my_string;
```

```
//construtor com string inicial  
string my_string("starting value");
```

Strings

- **Objetos `string` podem ser facilmente comparados**

```
string example;  
cin >> example;  
if (example == "POO")  
    cout << "É igual a POO";  
else if (example > "POO")  
    cout << "É maior que POO";  
else  
    cout << "É menor que POO";
```

Strings

- Para acessar o **tamanho da string**, use os métodos **length** ou **size**

```
string example = "string de 23 caracteres";  
int len = example.length(); // ou .size();  
cout << len;
```

- Se precisar acessar um caracter específico, faça como em C, ou seja, utilize notação de vetores:

```
example[0] = 'S';  
for(i = 0; i < example.length(); i++) {  
    cout << example[i];  
}
```

Strings

□ Alterando strings por remoção ou inclusão:

```
string example = "Programação desorientada";  
example.erase(12, 3); //resulta em "Programação orientada"
```

```
example.insert(example.length(), " a objetos"); //resulta  
em "Programação orientada a objetos"
```

```
example.erase(0, str.length()); //apaga a string inteira
```


Strings

- Entrada e saída de objetos `string` pode ser feita normalmente utilizando `cin` e `cout`
- É possível **concatenar strings por meio do operador `+`**

```
string my_string1 = "Linguagem";  
string my_string2 = " C++";  
string my_string3 = my_string1 + my_string2;
```

Strings

□ Mais sobre concatenação:

```
string s = "Programação " + "orientada"; //#include <string>
```

■ Erro!

```
string s = "Programação ";  
s = s + "orientada";
```

■ Ok!

```
string s = "Versão ";  
s = s + 1.0;
```

■ Erro!

```
stringstream ss; //#include <sstream>  
ss << s << 1.0;  
ss.str();
```

■ Ok!