Module III – Introduction to C++ Programming

Prof. Ismael H F Santos

Considerações Gerais

- **Objetivo:** *Discutir os principais conceitos e os princípios básicos da Orientação a Objetos usando a linguagem C++.*
- **A quem se destina :** *Alunos e Profissionais que desejem aprofundar seus conhecimentos sobre Linguagem C++ e suas aplicações*

Bibliografia

- *Thinking in C++ 2nd Edition by Bruce Eckel.*
  - *Very good introductory book*
- *The C++ Programming Language: 3rd Edition Bjarne Stroustrup,* Addison-Wesley, 1997, *ISBN 0-201-88954-4*.
  - A must-have for any C++ programmer.
- *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs by* Steve Maguire, Microsoft Press, ASIN 1556155514.
  - A very useful book showing many of the traps that you could run into when programming in C or C++.
- Effective C++, 2nd Edition
- More Effective C++
  - Scott Meyers, Addison-Wesley, ISBN 0-201-92488-9 and 0-201-63371

Bibliografia

- *Generic Programming and the STL: Using and Extending the C++ Standard Template Library* by Matthew H. Austern, Addison-Wesley Professional, ISBN 0-201-30956-4.
  - This book explains the STL—its concepts and algorithms, including examples.
- *The Most Important C++ Books...Ever*
- *C++ Free Computer Books*

Webliografia

- CppNotes - Fred Swartz
  - http://www.tecgraf.puc-rio.br/~ismael/Cursos/apostilas/Cpp-Notes/index.html
- The Code Project - C++
  - http://www.codeproject.com/index.asp?cat=2
- C/C++ reference
  - http://www.cppreference.com/index.html
- Boost - free peer-reviewed portable C++ source libraries
  - http://www.boost.org/

On-line resources

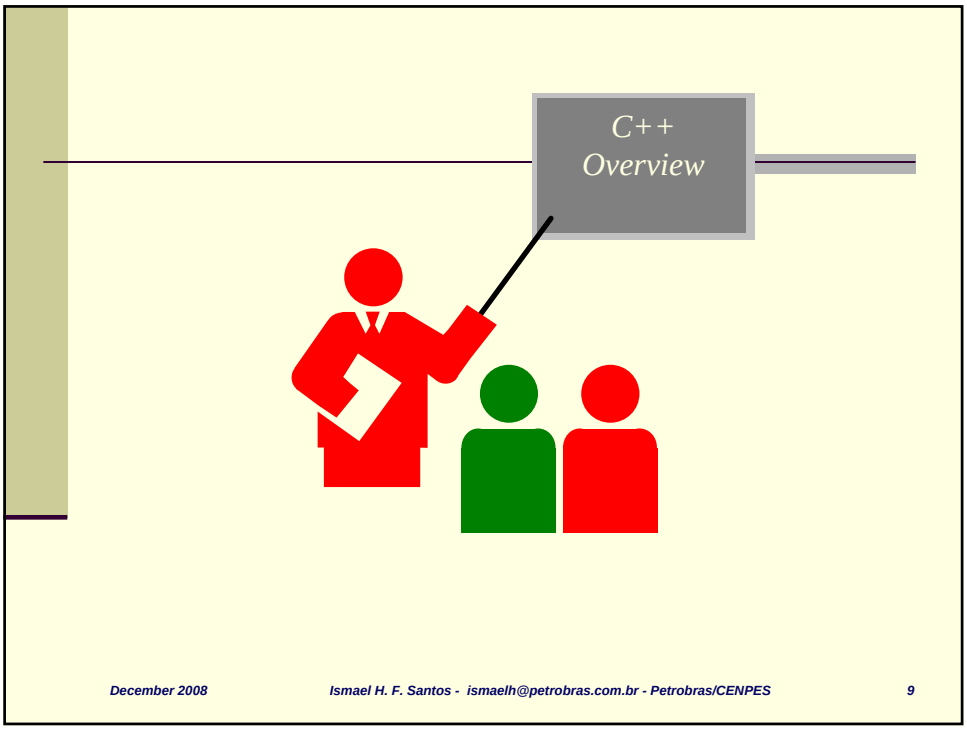
- Newsgroups
  - comp.lang.c++
  - comp.lang.c.moderated
- The C++ Users Journal
  - The C++ Users Journal is one of the best online C++-magazines and is designed for medium-to-advanced C++ programmers. It has a C++ experts column, featuring leading experts such as Herb Sutter, Andrei Alexandrescu, and others.
- GNU Compiler Collection
  - The GNU Compiler Collection (GCC) is one of the most popular free C++ compilers and implements most of the ANSI C++ standard.

On-line resources

- C++ libraries
  - The Boost Web site is a very useful collection of libraries for C++. It is almost impossible not to find at least one library you can successfully use in your current project. The documentation is very good. If you run into any problems, you can subscribe to the boost-developer list, post your problem, and you'll quickly get an answer.
- STL implementation
  - STL-port is a very popular implementation of the STL, in case your compiler vendor has not provided one.
- ANSI C++ compiler
  - Comeau Computing provides a C++ compiler considered to be the best at implementing the ANSI C++ standard. Also, you can compile your C++ code online.

Agenda

- *C++ Overview*
- *Pointers and References*
- *Operator Overloading*
- *InputStream & OutputStream*
- *Exception Handlers*
- *String Processing*
- *Generic Programming*



C++ Overview

- Apresentação
  - Projetada por Bjarme Stroustrup, 1980, na Bell Labs
  - Linguagem híbrida, estendendo as definições da linguagem C para incorporar o paradigma OO
  - Suporta a maioria dos conceitos de OO, porém não é considerada uma linguagem OO pura.

C++ Overview

- C++ is a multi-paradigm programming language, which allows, among other things: generic programming, Object Oriented Programming (OOP), and procedural programming.
- It derives its roots from C, which is a procedural language. One of its greatest features is that it trusts you - the programmer - by giving you full control.
- C++ is best used when you need efficient, generic programming requiring low-level, compact code that interacts with large amounts of data or with file systems.

C++ Strengths

- As with any language, C++ has both strengths and weaknesses. Because it is designed to be efficient, C++ is often used in back-end applications (servers), while front-end operations are frequently designed in Rapid Application Development (RAD) environments, using Visual C# or Java for example.

C++ Strengths

Support for generic programming	Generic programming allows programmers to create algorithms that will work with any data type, as long as the data satisfies a certain concept. Therefore, the reusability of generic programming classes and algorithms is an order of magnitude higher than OOP. C++ STL can handle arrays and collections very efficiently.
Efficiency	C++ is the younger and safer brother of C. Since C was all about efficiency, C++ was designed to be as efficient as possible while also providing extra type-safety.
Both high- and low-level programming	C++ allows you to program at the level you desire, depending on the module you're working on. While high-level programming is often preferable, there are times when you need to get down and dirty and program at low levels (for instance, when developing drivers).

C++ Strengths

Enforcing the const keyword	The const keyword in C++ makes programming safer. If you now that your function won't modify a parameter, you'll say so by making that parameter const. If by mistake you modify that parameter, a compile-time error will occur.

Streams	Writing to and reading from streams is very simple and straightforward. As a bonus, there are in-memory streams (istringstream, and ostringstream) that you can work with.
	`out << users << " users logged on at " << now; // writing`
	`in >> a >> b; // reading`

Multi-paradigm support	C++ does not force any single paradigm (such as OOP) on the programmer. Different applications and even different modules may require different paradigms.

C++ Weaknesses

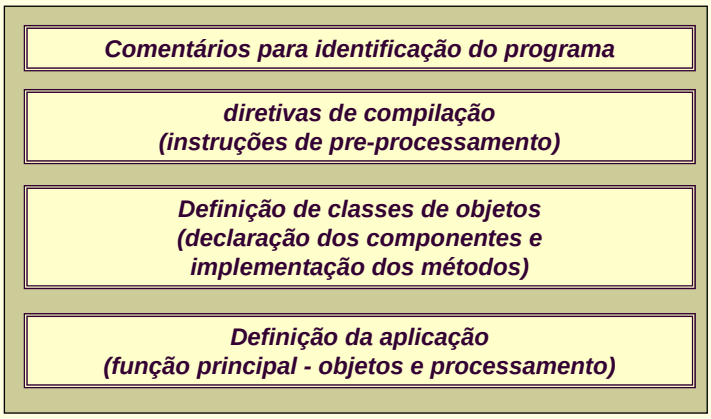
Backward compatibility with C	C++ was developed on top of C, and it has therefore retained a very high degree of C compatibility. However, many C weaknesses have been merged into C++ as well. These involve dealing with raw pointers, some not-so-straightforward automatic conversions, and a lot of error-prone functions such as strncpy, memcpy, etc.
GUI	C++ does not yet mix well with GUIs, except by the Qt toolkit. Simple tasks - such as adding or removing a splitter, manipulating Property Pages, showing tool tips, and handling menus - are really complicated to implement in C++
No conforming compilers	There's no compiler that fully implements the ANSI C++ standard, which was released in 1998. Because C++ has so many features that interact with each other, sometimes in a surprising manner, compiler errors are common. Depending on the code in question, compilers may generate a compile error on a good program, or successfully compile an ill-formed program.

C++ Weaknesses

Portability	C++ was developed on top of C, and it has therefore retained a very high degree of C compatibility. However, many C weaknesses have been merged into C++ as well. These involve dealing with raw pointers, some not-so-straightforward automatic conversions, and a lot of error-prone functions such as strncpy, memcpy, etc.
Bad compile-time error messages when using generic programming	A compile-time error generated when using generic programming libraries will be so cryptic that you will usually need the help of an experienced programmer to decipher it.
Many string implementations	A plethora of string classes exists, many with dubious features. In addition, there still are programmers using raw functions like strcat. The issue is complicated even more by the fact that you can have ANSI strings (formed of one-byte chars) and Unicode strings (formed of two-byte or four-byte chars).

Introduction to C++

- Estrutura de um Programa em C++



Introduction to C++

- Hello World program

```
/*
Program Hello world ← comentários para identificação
File: hello.cpp       do programa
*/
#include <iostream>  ← diretiva de compilação
                     ← nenhuma definição de classe ....
int main() {         ← definição entry-point da aplicação
  std::cout<<“Hello World”<<std::endl;
  return 0;

}
```

- *Exercicio: Editar o programa no IDE e executa-lo !!!*

Introduction to C++

- Including a header file

  #include <myfile.ext> // or

  #include “myfile.ext”
- You can include any file with any extension. However, the most common extensions are .h and .hpp. Also, note that Standard Template Library (STL) header files do not have any extension.
- The difference is that the .h extension will first look within the system headers (the ones that come with your C++ compiler), while the .hpp extension will first look within your project headers.

Another Example

Exemplo 1: classe Interval

```
class Interval {
    float left_, right_;   // private attributes
public:
    // class construtor
    Interval(float e, float d) { left_= e; right_= d; }
    // class methods
    float left() { return left_; }
    float right(){ return right_; }
    int   contains(float f) { return f>=left_&& f<right_; }
    int   contains(Interval x) {
            return left_<= x.left() && right_>= x.right(); }
    // quebra encapsulamento !
    friend void printInterval(Interval);
}

void printInterval(Intervalo x) { // outside class Interval
    printf("(%f, %f)", x.left_, x.right_);
}
```

First Example

Exemplo 1: classe Interval (cont'd)

```
#include <iostream>

int main() {
  Interval x(0, 1), x1(0.5, 0.75);
  float f = 0.5;
  std::cout<<x.left()<<x.right();
  std::cout<<x.contains(f);
  std::cout<<x.contains(x1);
}
--------- Output ----------
……
```

Exercicio: Implementar e testar !

Introduction to C++

- Método Construtor
  - É o serviço que inicializa uma instância (objeto) da classe
  - Possui o mesmo nome da classe
  - Não possui tipo de retorno
  - Pode possuir ou não parâmetros
  - É chamado automaticamente durante a criação de um objeto
  - Se não for definido, um construtor "default" é utilizado

Introduction to C++

- Método Destrutor
  - É o serviço executado quando um objeto é destruído
  - O método destrutor não retorna e nem recebe parâmetros
  - Se nenhum foi definido, um destrutor “default” é utilizado
  - Se o nome da classe for X, o método destrutor se chama ~X

Introduction to C++

- Ponteiro **this**
  - Ponteiro especial criado pelo compilador para cada classe de objetos.
  - Utilizado durante o processamento de um método
  - Aponta sempre para o objeto corrente que contém o método em processamento

Introduction to C++

- Comandos

Comando

- **expressão de atribuição**
- **formas pré-fixadas ou pós-fixadas de ++ e --**
- **chamada de métodos**
- **criação de objetos**
- **comandos de controle de fluxo**
- **bloco**

Bloco = { <lista de comandos> }

Introduction to C++

- **if-else**

```
if( a>0 && b>0 )
  m = media(a, b);
else {
  errno = -1;
  m = 0;
}
```

- **if-else-if**

```
if( a>0 && b>0 ) {
  m = media(a, b);
} else if (a < 0) {
  m = media(-a, b);
} else {
  errno = -1;
  m = 0;
}
```

```
if (expressão booleana)
   instrução_simples;

if (expressão booleana) {
   instruções
}
```

```
if (expressão booleana) {
   instruções
} else if (expressão booleana) {
   instruções
} else {
   instruções
}
```

Fundamentos da Linguagem

- **switch-case-default**

```
switch( i=f() ) {
  case -1:
    ...
    break;
  case 0:
    ...
    break;
  default:
    ...
}
```

- *Sintaxe*

qualquer expressão que resulte em valor inteiro (incl. char)

```
switch (seletor_inteiro) {
  case valor_inteiro_1 :
    instruções ;
    break ;
  case valor_inteiro_2 :
    instruções ;
    break ;
   ...
  default:
    instruções ;
}
```

uma constante inteira (inclui char)

Fundamentos da Linguagem

- **while**

```
int i = 0;
while( ++i<10 ) {
 cout>>i;
}
//Que números serão impressos?
//E se trocarmos por i++<10
-while( ++i<10 )->1..9
-while( i++<10 )->1..10
```

```
while (expresão booleana )
{
  instruções;
}
```

- **do while**

```
int i = 0;
do {
 cout>>i;
} while( ++i<10);
//Que números serão impressos?
//E se trocarmos por i++<10
```

```
do
{
  instruções;
} while (expresão booleana ) ;
```

Fundamentos da Linguagem

- **for**

```
for(int i=0; i<10; ++i)
 cout>>i;
//Que números serão impressos?
//E se trocarmos por i++
- for(int i=0; i<10; ++i)->0..9
- for(int i=0; i<10; i++)->0..9
- for(int i=0; i++<10;  )->1..10
- for(int i=0; ++i<10;  )->1..9
```

- **return**

```
int média(int a, int b) {
  return (a+b)/2;
}
```

```
for ( inicialização;
      expressões booleanas;
      passo da repetição )
{
  instruções;
}
```

```
boolean método() {
   if (condição) {
      instrução;
      return true;
   }
   resto do método
   return false;
}
```


Fundamentos da Linguagem

- **break**

```
int i = 0;
while( true ) {
  if (++i==10) break;
 cout>>i
}
//Que números serão impressos?
//E se trocarmos por i++
if (++i==10) break; -> 1..9
if (i++==10) break; -> 1..10
```

- **continue**

```
int i = 0;
while ( true ) {
  if (++i%2 == 1) continue;
  cout>>i;
}
//Que números serão impressos?
//E se trocarmos por i++
if (++i%2 == 1) continue; -> 2, 4, .
if (i++%2 == 1) continue; -> 1, 3, .
```

```
while (!terminado) {
  passePagina();
  if (alguemChamou == true) {
    break;       // caia fora deste loop
  }
  if (paginaDePropaganda == true) {
    continue;    // pule esta iteração
  }
  leia();
}
restoDoPrograma();
```

Introduction to C++

- Forma Comum de Definição de Classe

```
class <nomeClasse>
{
  <formaAcesso>:
        <declarações de atributos>;
  <formaAcesso>:
        <declarações (assinaturas) de serviços>;
} ;
```


Introduction to C++

- Definição dos Métodos da Classe
  - Dentro da classe (“inline”)

```
class <nomeClasse>
{
 --- --- --- --- --- --- --- --- --- --- --- --- --- ---
 <tipo> nomeMetodo(<declaração parâmetros formais>)
 {
   <declarações de variáveis locais>
   <processamento>
 }
} ;
```

Introduction to C++

- Definição dos Métodos da Classe
  - Fora da classe

```
class <nomeClasse>
{
  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---
  <tipo> nomeMetodo (<declaração parâmetros formais>);
} ;


<tipo> <nomeClasse>:: nomeMetodo (<declar. parâm. formais>)
{
  <declarações de variáveis locais>
  <processamento>
}
```


Introduction to C++

- Polimorfismo
  - Métodos Virtuais
    - Indicam os métodos que serão redefinidos por subclasses (mesma assinatura)
    - Uma vez declarado um método como virtual, todos os seus descendentes, nas classes derivadas, serão virtuais

Introduction to C++

- Polimorfismo
  - Métodos Virtuais Puros
    - Utilizados para indicar que seus herdeiros devem implementar um certo método
    - A assinatura de um método virtual puro é similar a de um método virtual comum acrescentado de = 0;
    - As classes que implementam métodos virtuais puros não podem ser instânciadas, apenas herdadas, sendo conhecidas como classes abstratas

Introduction to C++

- Sobrecarga
  - Sobrecarga de Funções
    - Permite a definição de vários métodos com mesmo nome, porém, com parâmetros diferentes
    - Reduz a quantidade de nomes de funções

Introduction to C++

- Sobrecarga
  - Sobrecarga de operadores
    - Permite com que operadores atuem sobre diferentes tipos de operandos
    - O operador +, por exemplo, opera sobre reais e inteiros
    - A sobrecarga é feita com o uso de uma função denominada “operator” que deve ser membro de uma classe de objetos

Pointers and References

References x Pointers

- **Referência** é um modificador que permite a criação de novos tipos derivados. Assim como podemos criar um tipo ponteiro para um inteiro, pode-se criar uma referência para um inteiro.
- A declaração de uma referência é análoga à de um ponteiro, usando o caracter & no lugar de *. A diferença está também na utilização.
- Vejamos alguns exemplos onde utilizamos a declaração de referências em três situações: variáveis locais, parâmetros e tipo de retorno de função.

References x Pointers

```
Exemplo 1:  Variável Local
{
   int  a = 10;
   int& b = a;           // Ok, b é referência para a
   int& c;               // Erro!
   int& d = 12;          // Erro!
   int* e = &a;          // Ok, e é ponteiro para a
   printf("a= %d, b= %d", a, b); //produz a= 10, b= 10
   std::cout<<"a= "<<a<<", b= "<<b;
   printf("endA= %d, endB= %d", &a, &b); //produz a= 1024, b= 1028  !!!

   a = 3;
   printf("a= %d, b= %d", a, b); // produz a= 3, b= 3
   b = 7;
   printf("a= %d, b= %d", a, b); // produz a= 7, b= 7
   *e = 5;
   printf("a= %d, b= %d, *e= %d, &e= %p, e= %p", a, b, *e, &e, e);
  //produz a= 5, b= 5, *e= 5, &e= endereço de e, e= endereço de a
}
```

References x Pointers

```
int a1 = 10;
int a2 = 20;
int &b = a1;
int &c = a2;

int &d = c;
cout<<b<<c<<d;  -> Imprime 10  20  20
a2=30;
cout<<b<<c<<d;  ->  Imprime 10  30  30
```


References x Pointers

Exemplo 2: **Argumento de Função**

```
void f (int a1, int& a2, int *const a3, const int *a4)
{
   a1  = 1;              // altera cópia local de a1
   a2  = 2;              // altera variável passada (b2 de main)
   *a3 = 3;              // altera o conteúdo do endereço de b3
    a4 = 0x1024;
  ------------------------------------------------------
   a3  = &a2;         // Error – pointer is constant !!!
   *a4 = 1;           // Error – cannot change pointed value !!!
}

void main (void)
{
   int b1= 10, b2= 20, b3= 30; b4= 40;
   f( b1, b2, &b3, &b4 );
   printf("b1= %d, b2= %d, b3= %d, b4= %d", b1, b2, b3, b4);
 // produz b1= 10, b2= 2, b3= 3, b4= 40
}
```

References x Pointers

Exemplo 3: **Retorno de uma Função**

```
int& f (void) {
   static int global;
   return global;    // retorna referência para a variável
}
void main (void) {
   f() = 12;         // altera a variável global
    cout<< f() + 5;  // imprime 17
}
```

- Observe que este exemplo é válido porque ***global*** é uma variável local estática da função ***f()***, isto é, é uma variável global porém com escopo limitado a função ***f()***; se a variável não fosse static o valor de retorno não seria válido pois após o retorno da função ***f()*** a variável ***global*** não mais existiria já que ela teria sido alocada na stack !

References x Pointers

- Quando uma classe não declarar construtores o compilador gera automaticamente o construtor padrão e construtor de cópia.

Exemplo 4:

```
class x {
   int a;
};
```

Compilador...

```
class X {
    int a;
  public:
    X () {}       // Construtor padrao
    X (const X&); // Construtor de cópia recebe referência
                  // constante p/ objeto da classe X
};
```

References x Pointers

- O construtor de cópia constrói um objeto a partir de outro do mesmo tipo. O novo objeto é um cópia byte a byte do objeto passado como parâmetro.

```
void main (void)
{
   A a1;                      // Usa construtor padrão
   A a2;                      // Usa construtor padrão
   A a3 = a2; ou A a3(a2); //  Usa construtor de cópia
   A a4;
    ......
   a4 = a3; // Não usa construtor de cópia, usa operator=
            // faz cópia byte a byte
}
```


Static Members & Functions

Introduction to C++

- Membros Estáticos
  - Um membro “static” possui somente uma cópia para todas as instâncias da classe, sendo assim compartilhado.
  - Se uma instância da classe altera uma variável de instância “static”, esta alteração é visível para todas as outras instâncias.

Introduction to C++

- Membros Estáticos
  - Se uma função membro é “static”, qualquer aplicação que contenha a declaração da classe a qual a função membro pertence, pode fazer uso da função sem a necessidade da existencia de uma instância da classe !

Classes e Métodos static

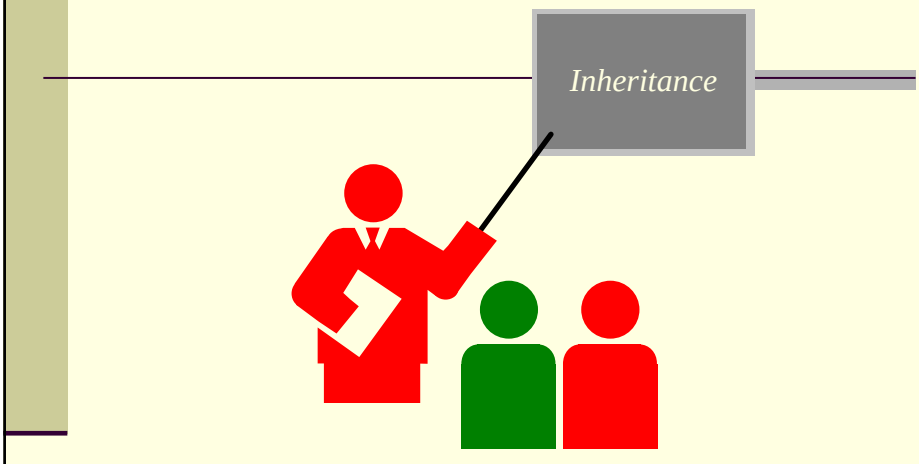
- Exemplo de função membro estático

```
// Construct a Stack with LE implementation - factoryMethod
   Stack* s = Stack::create(Stack::STACK_LE);

class Stack {  // Classe Abstrata !!!
public:
    enum TpStack {
      STACK_VE,
      STACK_LE
    };
    virtual void push(int) = 0; // abstract method ...
    virtual int  pop()     = 0; // Late binding !!!
    virtual bool isEmpty() = 0;
    virtual bool isFull()  = 0;
    virtual int  top()     = 0;
    virtual StackIterator *iterator() = 0;

    // Factory Method para construcao de Stack
    static Stack *create(TpStack); // factory method
  };
```


Classes e Métodos static

Introduction to C++

- Funções Amigas - Friend
  - Não são membros da classe porém podem utilizar os membros “private” ou “protected” da classe
  - Viola o princípio de encapsulamento



Introduction to C++

- Herança
  - Herança Simples

```
class <nomeClasse> :
        <especificadorAcessoMembrosHerdados> <nomeClasseBase>
{
  <formaAcesso>:
          <declarações de atributos>;
  <formaAcesso>:
          <declaração (assinaturas) de serviços>
} ;
```


Introduction to C++

- Herança
  - Herança Múltipla

```
class <nomeClasse> :
        <especificadorAcessoMembrosHerdados> <nomeClasseBase>,
        <especificadorAcessoMembrosHerdados> <nomeClasseBase>,
        --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
{
  <formaAcesso>:
          <declarações de atributos>;
  <formaAcesso>:
          <declaração (assinaturas) de serviços>
} ;
```

Interfaces em C++

- Interfaces
  - *Algumas linguagens OO incorporam o conceito de duas classes serem compatíveis através do uso de compatibilidade estrutural ou da implementação explícita do conceito de interface.*
  - Java não permite herança múltipla com herança de código, mas implementa o conceito de interface. É possível herdar múltiplas interfaces.
  - Em Java, uma classe estende uma outra classe e implementa zero ou mais interfaces. Para implementar uma interface em uma classe, usamos a palavra implements.

Operator Overloading

Operator Overloading

- O uso de funções sobrecarregadas não só uniformiza chamadas de funções para diferentes objetos como também permite que os nomes sejam mais intuitivos.
- Um operador executa algum código com alguns parâmetros. A aplicação de um operador é equivalente à chamada de uma função. Duas maneiras de implementar operadores para classes de objetos são possíveis: como funções membro e como funções globais:
  - objeto x e um operador unário "@", a expressão:

```
@x →    x.operator@();   // função membro ou
        operator@(x);    // função global,

X x;    -x -> x.operator-(); // função membro
        -x -> operator-(x);  // função global - operator-(X &);
```


Operator Overloading - Complex

- objetos x e y e um operador binário "!", a expressão:

```
x ! y → x.operator!(y);  // função membro ou
        operator! (x, y);// função global

X x1,x2; x1 + x2 -> x1.operator+(x2); // função membro
         x1 + x2 -> operator+(x1, x2);// operator+(X &, X &);
```

- Observe que a função **y.operator!(x)** nunca será considerada pelo compilador para resolver **x!y.** Vejamos o caso de Complex numbers

Operator Overloading

- Overloading operators
  - You can overload some operators in your classes; however, it is recommended that you overload only those that make sense. A good example of operator overloading is the std::string class.

```
#include <string>
#include <iostream>
struct name {
  std::string first, last;
  name(const std::string & first, const std::string & last) :
       first(first), last(last) {}
};
bool operator==( const name & me, const name & you) {
   return me.first == you.first && me.last == you.last;
}
int main() {
  name john("John","Doe"), james("James","Gucci");
  std::cout<<( john==james ? "John=James" :
                             "John not James")<<std::endl;
}
```


Operator Overloading - Complex

- Classe Complex – complex.h

```
class Complex {
        float r_, i_;
   public:
        Complex(float re = 0; float im = 0) { r_ = re; i_ = im;}
         Complex(float re) { r_ = re; i_ = 0; }
        float re() {return r_;}
        float im() {return i_;}
        bool operator==(Complex &c);
        Complex operator+(const Complex &c);
        Complex operator-(const Complex &c);
        Complex operator*(const Complex &c);
        Complex operator/(const Complex &c);
};
```

Operator Overloading - Complex

- Classe Complex - complex.cpp (funções membro)

```
#include "complex.h"
bool Complex::operator==( const Complex &c ) {
   return r_ == c.re() && i_ == c.im();
}
Complex Complex::operator+( const Complex & c ) {
   return Complex(r_ + c.re(), i_ + c.im()); //novo Complex
}
---------------------------
#include "complex.h"
int main() {
  Complex x(1,0),y(1,1),z;// Criados 3 complexos (1,0),(1,1),(0,0)
  if( z == (x + y)  )    // z = x.operator+(y);-> (2,1) temporario
       …..               // z.operator==( x.operator+(y) );
}
```

Exercício **– Terminar o restante...**

Operator Overloading - Complex

- Classe Complex – complex2.cpp (funções globais)

```
#include "complex2.h"
int operator==(const Complex& c1, const Complex& c2) {
   return c1.re() == c2.re() && c1.im() == c2.im();
}
Complex operator+(const Complex& c1, const Complex& c2) {
   return Complex(c1.re() + c2.re(), c1.im() + c2.im());
}
---------------------------
#include "complex2.h"
int main() {
  Complex x(1,0),y(1,1),z;// Criados 3 complexos (1,0),(1,1),(0,0)
  if( z == (x + y)  )    // operator+(x, y); ->(2,1) temporario
      ……                 // operator=(z, operator+(x, y)) -> false!
}
```

- ***Exercício*** **– Terminar o restante...**

Operator Overloading - Complex

- Observamos que na expressão:

```
c1 = c2 + 3.0f:                    // c2(0,1)
c1 = c2 + Complex(3.0f);           // (3,0)
c1 = c2.operator+(Complex(3.0f)); // (3,1) novo objeto temp
```

- Neste caso a conversão foi feita porque a função membro operator+ espera um Complex, e o valor 3.0f pode ser automaticamente convertido para um Complex. (Porque ?)
- No caso c1= 3.0f + c2; observe que não existe operator+ para o tipo float que o some a um Complex. Assim nenhuma conversão poderá ser feita e temos um erro de compilação. A solução é fazer:

c1 = Complex(3.0f) + c2; ou declarar operador+ como método global:

```
Complex operator+(float f, const Complex& c){
   return Complex( f + c.re(), c.im());
}
```


Operator Overloading - Vector

- Outro exemplo interessante é o caso de uma classe de vetores que permitisse a identificação do acesso a elementos do vetor fora de seus limites. Isto pode ser feito redefinindo-se o operador "[ ]".

```
Módulo Vector.h
class Vector {
   float *v_;
   int    sz_;
public:
   Vector (int);
   virtual ~Vector () {delete [] v_;}
   int  getSize(void) {return sz_};
   void setSize (int);
   float& operator[](int);  // Redefinição do Operador[]
   float& elem(int i) {return v_[i];} // v.elem(i) <-> v[i]
};
```

Operator Overloading - Vector

- Entretanto se tivéssemos declarado o método **operator[ ]** retornando um float teríamos problemas:

```
float Vector::operator[] ( int i ) {
   if (i >= 0 && i < sz_) return v_[i];
   else {
      printf("Indice %d Inválido !!! \n", i); return –1.0;
   }
}
 Vector vector(100);
  a = vet[10];   // 1 - Válida
  vet[20] = 2.0; // 2 – Inválida  - Explique o porquê !
```

- A solução é o método operator[ ] retornar uma referência para um float. Assim o valor de retorno pode ser o endereço, necessário no caso 2, ou um valor como no caso 1. Caberá ao compilador decidir qual o papel que será assumido pelo retorno do método em cada situação.

Operator Overloading - Vector

Módulo Vector.cpp

```
#include "Vector.h"
Vector::Vector(int s) {
   if(s <= 0) error ("bad vector size");
   sz_ = s;   v_ = new float [s];
}
float& Vector::operator[]( int i ) {
   if (i >= 0 && i < sz_) return v_[i];
   else {
        static float lixo;
        printf ("Indice %d inválido !!! \n", i);
        return lixo;
   }
}
```

- A primeira vista parece-nos esquisito a declaração da variável estática lixo, retornada em caso de erro !

Revisitando referencia !!!

```
class Vector2 {
  int *v_;
  int size_;
  public:
    Vector2(int s) { size_ = s; v_ = new int[size_]; }
    int& operator[](int i) { …….. }
    int* getAddress(int i) { if( 0<= i && i < size_ )
                               return &v_[i];
                             else return 0L; }
    int getValue(int i) {if( 0<= i && i < size_ )
                           return v_[i];
                         else return -99999; }
}
```

```
Vector2 v(2);
for( int i=0; i<2; i++ )
  v.getAddress(i) = 2*v.getValue(i);
```

```
Vector2 v(2);
for( int i=0; i<2; i++ )
  v[i] = 2*v[i];
// v.operator[](i) =
//      2*v.operator[](i)
```


Operator Overloading - Arquivo

- Podemos redefinir operadores especiais para conversão de tipos. Além das conversões padrão, o programador pode definir como um objeto pode ser convertido para algum outro tipo como abaixo:

Módulo Arquivo.cpp

```
#include <stdio.h>
class Arquivo {
     FILE *file;
   public:
     Arquivo(char *nome) { file=fopen(nome, "r"); }
     ~Arquivo() { fclose(file); }
     char read(void)   { return file?fgetc(file):EOF; }
     int  aberto(void) { return file != NULL; };
     operator FILE*(void){ return file; }  //Redefine FILE*
};
```

Operator Overloading - Arquivo

- Observe que esta estratégia é melhor que tornar o ponteiro para FILE uma variável pública!

Módulo main.cpp

```
#include "arquivo.h“

void main( void ) {
    int i;
    Arquivo arq("teste.txt");
   fscanf((FILE*)arq, "%d", &i);
}
```

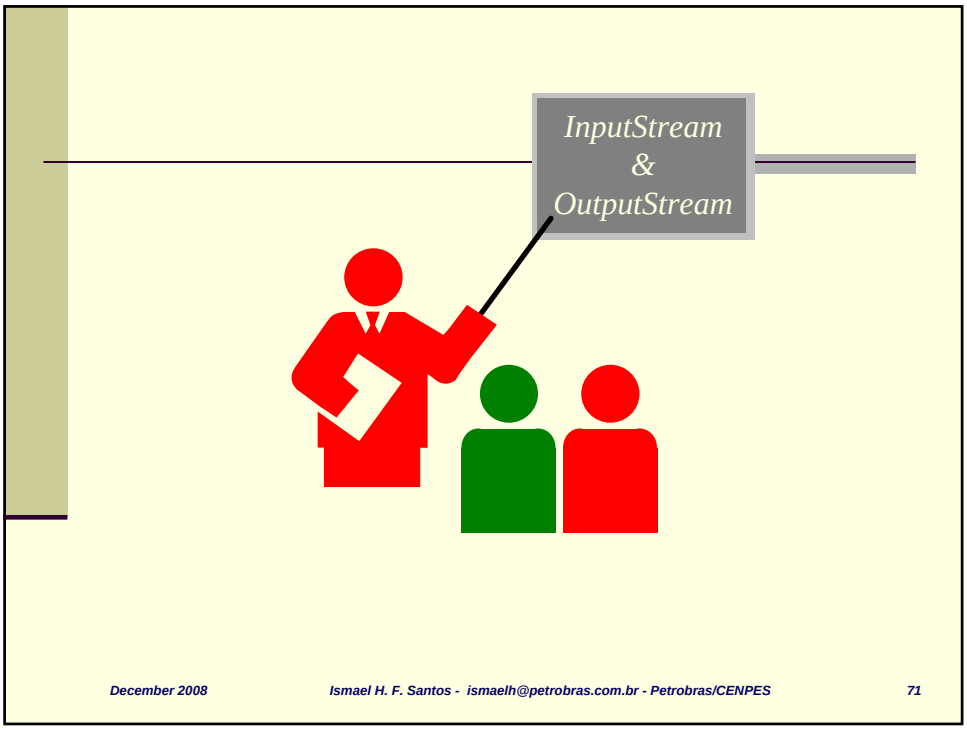

Printing Complex

- A impressão de elementos da classe Complex pode ser feita através da redefinição do operator<< de cout para imprimir o complexo z=(real, imag) como: (real, i imag).

```
#include <iostream>
ostream operator<<(ostream s, Complex z) { // std::cout<<z;
   return s << “(“ << z.re() << “, i” << z.im() << “)”;
}
```

- Observe que, como quase todos os operadores em C++, << agrupa da esquerda para a direita, isto é, a << b << c equivale a (a << b) << c. A maneira mais eficiente seria utilizar a chamada por referência: (porque ?)

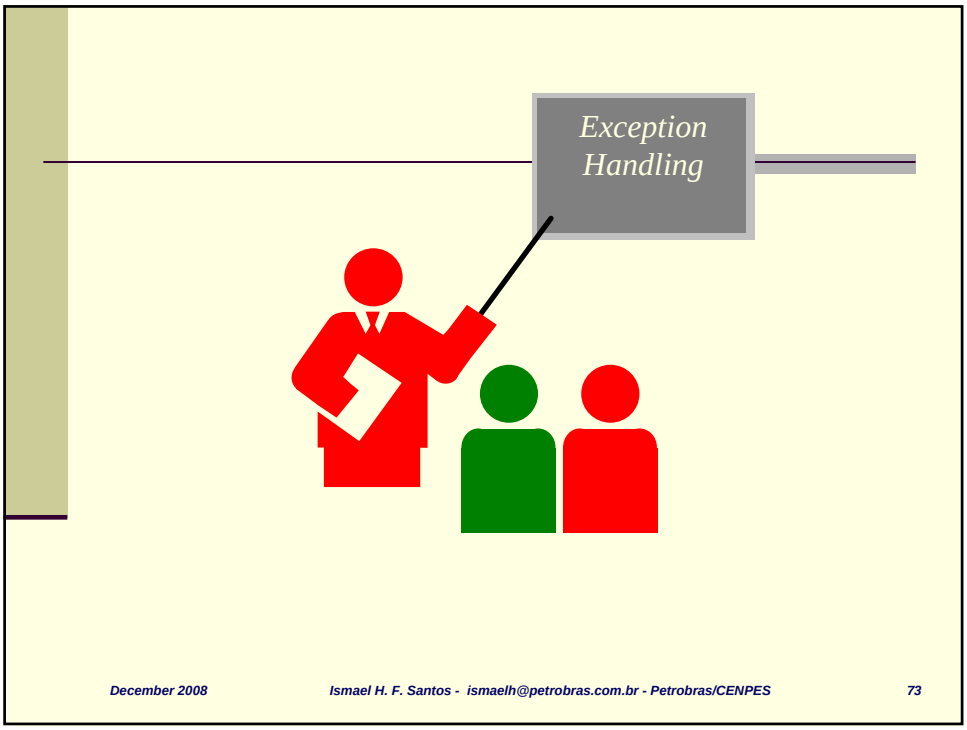
```
ostream& operator<<(ostream& s, Complex& z) {
   return s << “(“ << z.re() << “,” << z.im() << “)”;
}
É válido declarar: Complex c;
                   std::cout<<c; // imprime (0,0)
```



InputStream & OutputStream

- Reading from a file
  - This program reads from the readme.txt file and gathers all lines into an array (“lines”). Then, it dumps all read lines to the console.

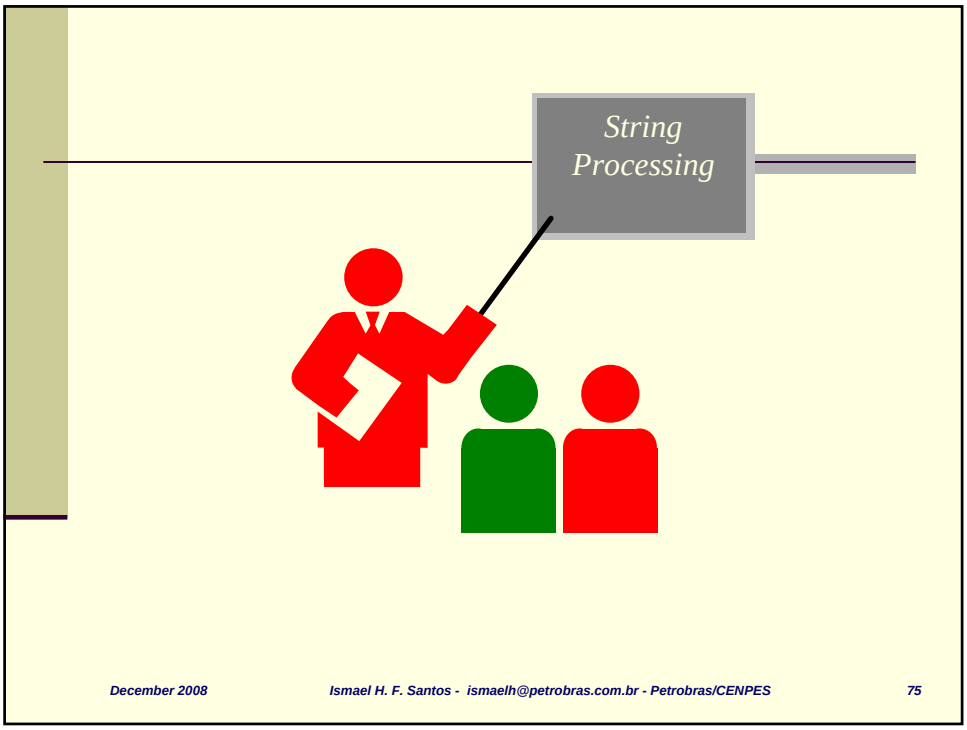
```
#include <string>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <vector>

int main() {
  std::ifstream in(“readme.txt”);
  std::vector<std::string> lines;
  std::string cur_line;
  while ( std::getline(in, cur_line) )
    lines.push_back(cur_line);
  std::copy( lines.begin(), lines.end(),
  std::ostream_iterator<std::string>(std::cout,”\n”));
}
```



Exception Handling

- An exception is thrown using the throw keyword. In order to catch an exception, you have to surround the suspected code in a try/catch block. In the catch clause, you specify the types of objects you can handle. (You can specify more than one type.)

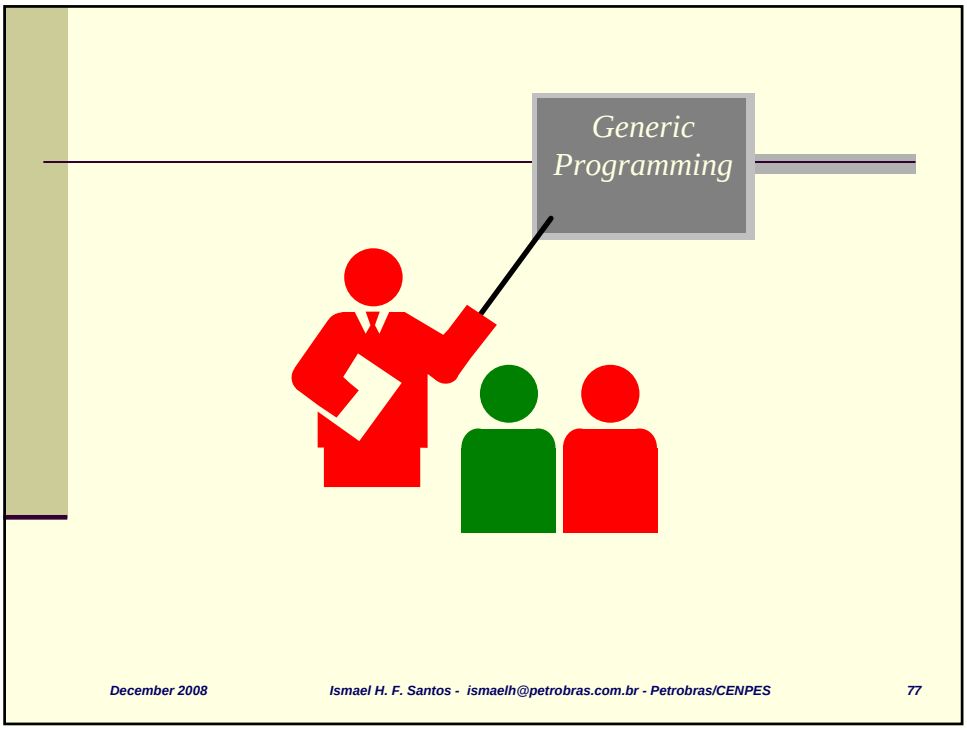
```
#include <stdexcept>
#include <iostream>
#include <string>
void func() { throw std::runtime_error (“oops!”); }
int main() {
  try {
    func();
    std::cout << “no exception” << std::endl;
  } catch( std::exception & exc) {
    // catches all exceptions derived from std::exception
    std::cout << “caught “ << exc.what() << std::endl;
  }
}
```



String Processing

- Parsing strings

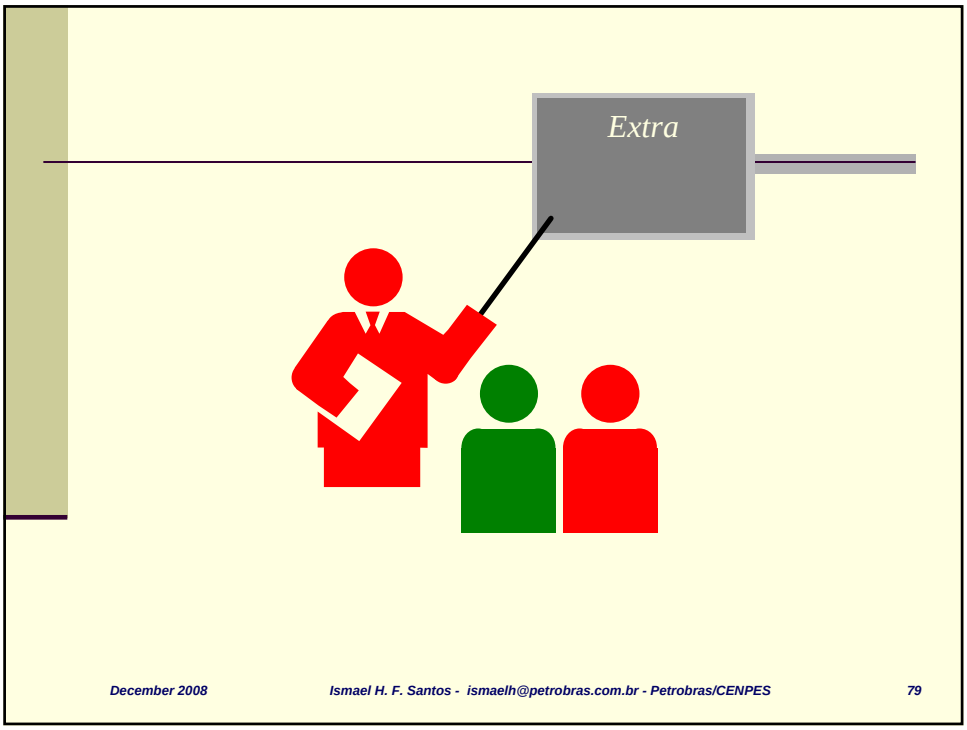
```
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
#include <map>
void print_word_info(const std::pair<std::string, int> & val) {
  std::cout<<“word: “ <<val.first<<“ appeared “ <<
           val.second<<“ times.\n”;
}

int main() {
  std::istringstream in(“me and you and others & again me and you”);
  std::map<std::string, int> counts;
  std::string word;
  while ( in >> word )
    ++counts[word];
  std::for_each( counts.begin(), counts.end(), print_word_info);
}
```



Generic Programming

- Generic max<> function
  - This sample code shows a very simple example of a generic algorithm: a max<> algorithm that works for any type that has a < (less-than) operator.
  - All built-in types have the < operator; however, custom-created types can choose to overload it.)

```
#include <string>
#include <iostream>
template<class T>
   T Max( const T & a, const T & b) { return a < b ? b : a; }

int main() {
  std::cout << "\nMax(1,2)=" << Max(1,2);          /* ints */
  std::cout << "\nMax(3.5,2.3)=" << Max(3.5,2.3); /* doubles */
  std::string first = "fox", second = "other";
  std::cout << "\nMax('fox','other')=" <<          /* strings */
     Max(first, ecnod);
}
```



Upcasting

- *Tipos genéricos (acima, na hierarquia) sempre podem receber objetos de suas subclasses:* ***upcasting***

```
Veiculo v = new Carro();
```

  - *Há garantia que subclasses possuem pelo menos os mesmos métodos que a classe*
  - *v só tem acesso à "parte Veiculo" de Carro. Qualquer extensão (métodos definidos em Carro) não faz parte da extensão e não pode ser usada pela referência v.*

Downcasting

- *Tipos específicos (abaixo, na hierarquia) não podem receber explicitamente seus objetos que foram declarados como referências de suas superclasses:* ***downcasting***

```
Carro c = v; // não compila!
```

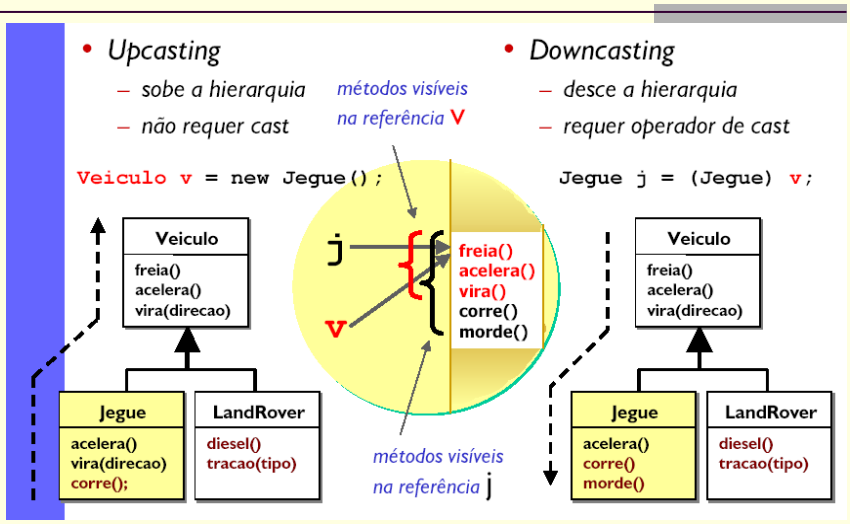
  - *O código acima não compila, apesar de v apontar para um Carro! É preciso converter a referência:*

```
Carro c = (Carro) v;
```

  - *E se v for Onibus e não Carro?*

Upcasting e Downcasting

- *Upcasting*
  - *sobe a hierarquia*
  - *não requer cast*
- *Downcasting*
  - *desce a hierarquia*
  - *requer operador de cast*



ClassCastException

- *O downcasting explícito sempre é aceito pelo compilador se o tipo da direita for superclasse do tipo da esquerda*

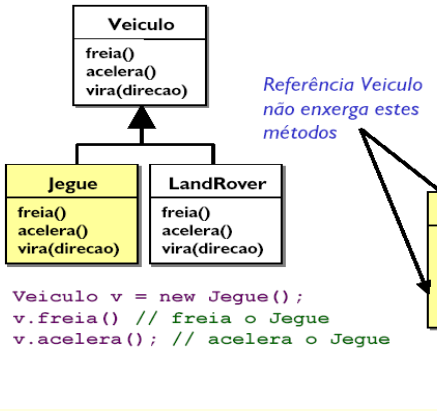
```
Veiculo v = new Onibus();
Carro c = (Carro) v; // passa na compilação
```

  - *Object, portanto, pode ser atribuída a qualquer tipo de referência*
- *Em tempo de execução, a referência terá que ser ligada ao objeto*
  - *Incompatibilidade provocará ClassCastException*
- *Para evitar a exceção, use instanceof*

```
if (v instanceof Carro)
        c = (Carro) v;
```

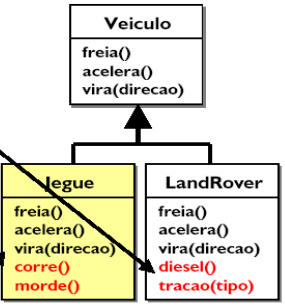

Herança Pura x Extensão

- *Herança pura: referência têm acesso a todo o objeto*




```
Veiculo v = new Jegue();
v.freia() // freia o Jegue
v.acelera(); // acelera o Jegue
```

- *Extensão: referência apenas tem acesso à parte definida na interface da classe base*



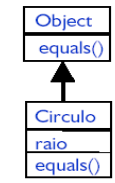
Referência Veiculo não enxerga estes métodos

```
Veiculo v = new Jegue();
v.corre() // ERRADO!
v.acelera(); //OK
```

23

Ampliação da Referência

- *Uma referência pode apontar para uma classe estendida, mas só pode usar métodos e campos de sua interface*
  - *Para ter acesso total ao objeto que estende a interface original, é preciso usar referência que conheça toda sua interface pública*
- *Exemplo*



ERRADO: raio não faz parte da interface de Object

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (this.raio == obj.raio)
      return true;
    return false;
  }
} // CÓDIGO ERRADO!
```

verifica se obj realmente é um Circulo

cria nova referência que tem acesso a toda a interface de Circulo

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (obj instanceof Circulo) {
      Circulo k = (Circulo) obj;
      if (this.raio == k.raio)
        return true;
    }
    return false;
  }
}
```

Como k é Circulo possui raio

Classe com apenas 1 instância – Design Pattern Singleton

```
public class Highlander {
  private Highlander() {}
  private static Highlander instancia;
  public static Highlander criarInstancia() {
    if (instancia == null)
      instancia = new Highlander();
    }
    return instancia;
  }
}
```

Esta classe implementa o padrão de projeto Singleton

```
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.criarInstancia();
    h3 = Highlander.criarInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

Esta classe cria apenas um objeto Highlander

End of Module III