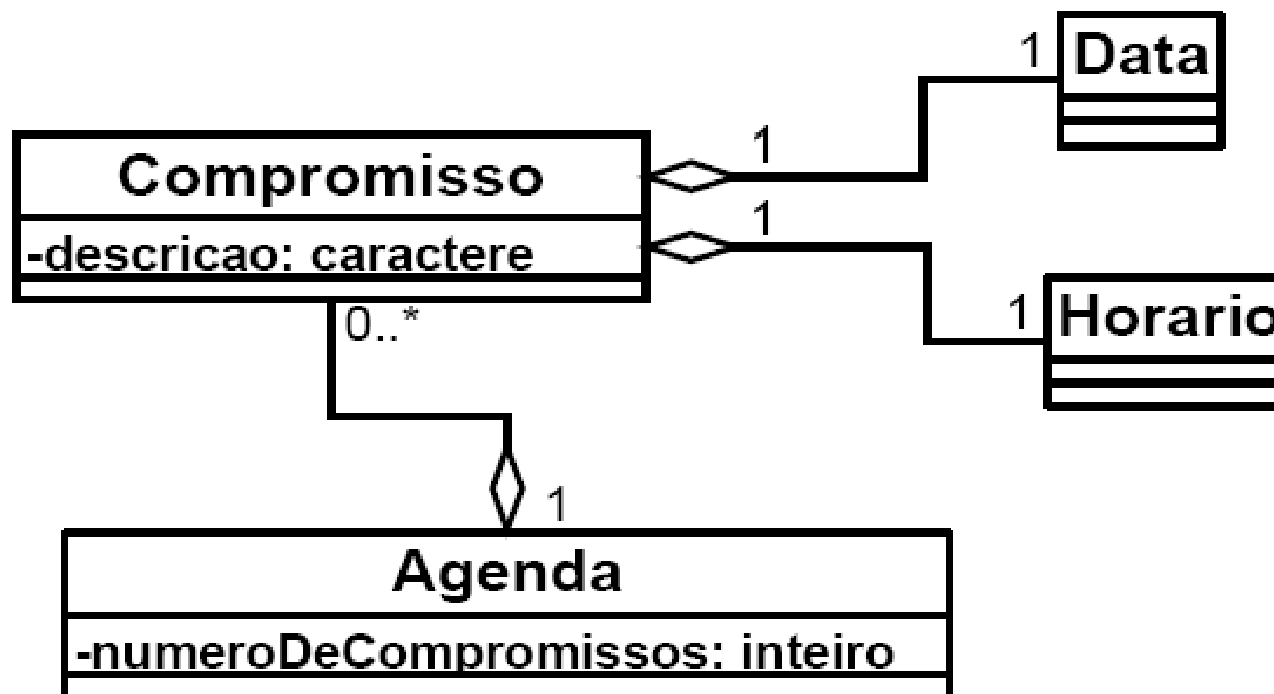
Exercício:

Construa o diagrama de classes UML para as classes envolvidas na solução do exercício do slide 253.



UNIVASF

Sobrecarga

Neste exercício vimos a sobrecarga de funções-membros, mais especificamente, a sobrecarga da função-membro construtor.

Conforme vimos anteriormente, a sobrecarga é um conceito relacionado ao polimorfismo e ocorre quando o nome (ou símbolo) de mais de um método (ou operador) definidos na mesma classe são o mesmo. Neste caso, se diz que o nome (ou símbolo) está sobrecarregado, mais adiante aprofundaremos a exploração deste conceito.

Detalhe sobre alocação dinâmica (new)

Em C++, quando é efetuada a alocação dinâmica de memória para armazenar um objeto de uma classe qualquer, o construtor desta classe é evocado para inicializar a área alocada. Por exemplo:

```
#include "data.h"
...
Data *p1 = new Data;
```

No exemplo acima o construtor padrão é chamado. Também existe a possibilidade de passarmos argumentos, para um construtor que os tolere, por exemplo:

```
#include "data.h"
...
Data *p2 = new Data(06, 09, 2006);
```

Exercício:

Como podemos perceber, na solução apresentada para o exercício do slide 249 não foram utilizadas funções set e get. Analise a solução proposta e, caso julgue necessário, a adapte introduzindo as funções set e get que você julgar pertinente para a solução em questão.

UNIVASF

Funções amigas

Em algumas situações pode ser mais cômodo (embora, em geral, isto deva ser evitado) permitir que uma função tenha acesso a membros de dados privados de uma classe, *mesmo não sendo tal função uma função-membro da classe*. Para tanto, pode-se especificar esta função **externa** como **amiga** (*friend*) da classe em questão, prefixando seu protótipo na declaração da classe como friend. Uma mesma função pode ser amiga de mais de uma classe.

Para uma melhor compreensão analisaremos agora um exemplo.

```
//exemplo de função amiga
#include <iostream>
using namespace std;
class Dois; //especificação antecipada
class Um
{
  public:
    friend void show (Um &p, Dois &q);
    void init (int a, float b);
  private:
    int k;
    float x;
};
class Dois
{
  public:
    friend void show (Um &p, Dois &q);
    void inicial (int p1, float p2);
  private:
    int campo1;
    float campo2;
};
```

```
//exemplo de função amiga - continuação
void Um::init (int a, float b)
{
  k=a;
  x=b;
}
void Dois::inicial (int p1, float p2)
{
  campo1=p1;
  campo2=p2;
}
void show (Um &p, Dois &q)
{
  cout << “Dados Classe Um:” << endl;
  cout << “k = ” << p.k << endl << “x = ” << p.x << endl;
  cout << “Dados Classe Dois:” << endl;
  cout << “campo1 = ” << q.campo1 << endl << “campo2 = ” <<
q.campo2 << endl;
}
...
  Um o1; Dois o2;
  show (o1, o2);
```

Funções amigas

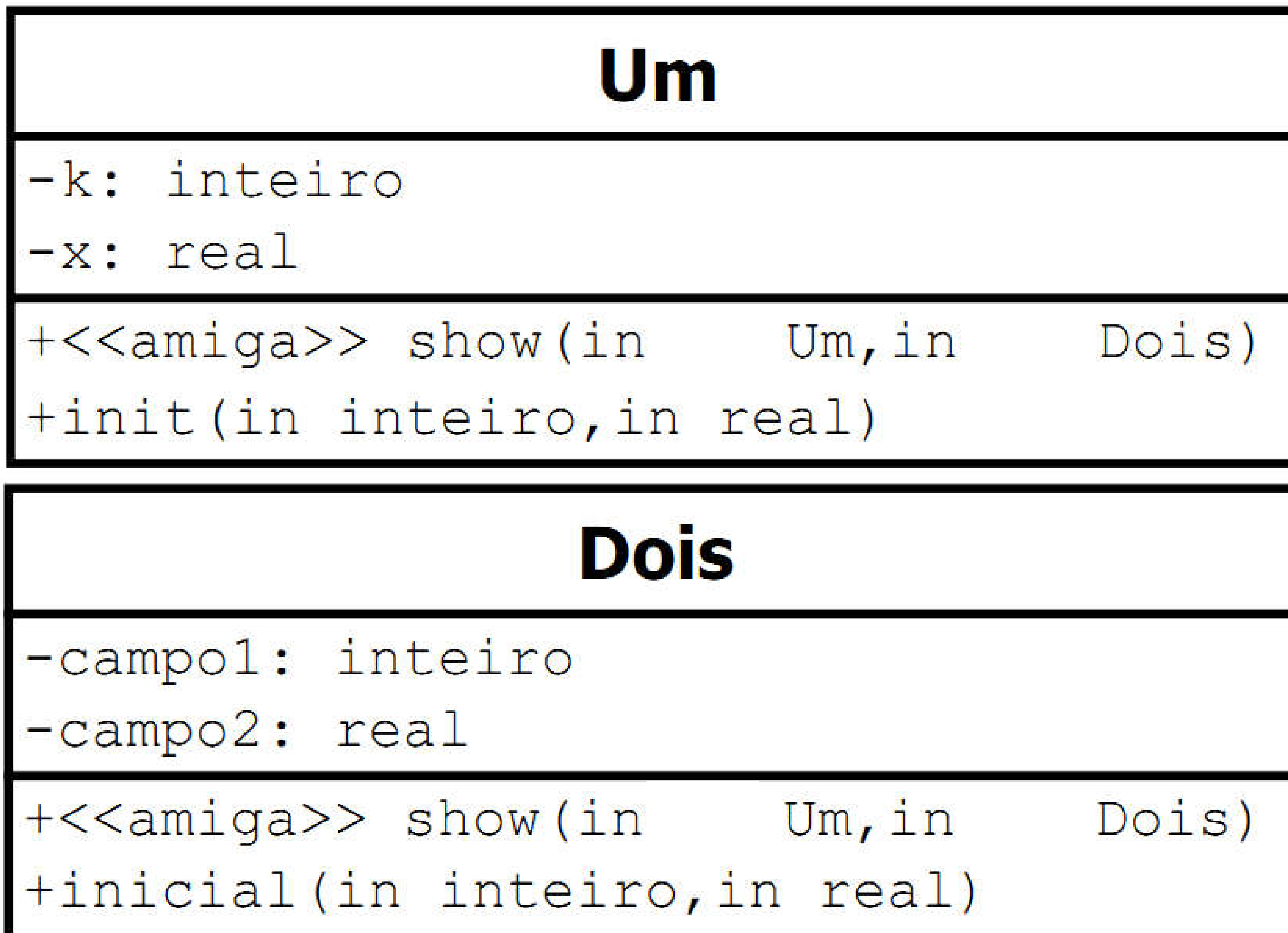
Observe que a função show é uma função comum, não membro de classe. Normalmente, ela não teria acesso aos membros privados de objetos de qualquer classe. No entanto, como amiga das classe Um e Dois pode operar sobre membros de dados destas classes.

Isto constitui uma violação do princípio do encapsulamento, mas com este comportamento torna-se pelo menos mais fácil obter certas soluções (como no caso da sobrecarga de operadores stream).

Por convenção as declarações *friends* aparecem em primeiro lugar na definição de classe.

UNIVASF

Funções amigas – Diagrama UML



Ponteiro *this*

É interessante frisar que, no contexto das funções membros que trabalhamos, estas sempre têm acesso aos membros de dados dos objetos da classe. Porém, como estas determinam de qual dentre os objetos da classe devem manipular os parâmetros?

Isto é possível porque toda função-membro de uma classe tem um parâmetro adicional, oculto, implicitamente declarado pelo compilador, que, na invocação, recebe uma referência ao objeto receptor.

Vamos analisar um exemplo:

```
#include <iostream>
using std::cout;
using std::endl;
class Teste {
  public:
    Teste( int = 0 );
    void imprime();
  private:
    int x;
};
```

```
Teste::Teste( int valor ) : x( valor )
{
}
void Teste::imprime()
{
  cout << "              x = " << x;
  cout << "\n  this->x = " << this->x;
  cout << "\n(*this).x = " << ( *this ).x << endl;
}
int main()
{
  Teste objetoTeste( 12 );
   objetoTeste.imprime();
  return 0;
}
```

Ou seja, tudo se passa como no exemplo anterior, como se a função membro imprime() tivesse sido assim declarada:

```
void Teste::imprime(Teste *this)
{
  cout << "       x = " << x;
  cout << "\n  this->x = " << this->x;
  cout << "\n(*this).x = " << ( *this ).x << endl;
}
```


UNIVASF

O principal uso do ponteiro *this* é nas funções membros que retornam uma referência para o próprio objeto receptor. Este tipo de utilização permite a chamada de função membro em cascata.

Vamos analisar um exemplo:

```
//conteúdo do arquivo horario.h
#ifndef HORARIO_H
#define HORARIO_H
  class Horario
  {
    public:
      Horario( int = 0, int = 0, int = 0 );
      Horario &setHorario( int, int, int );
      Horario &setHora( int );
      Horario &setMinuto( int );
      Horario &setSegundo( int );
      int getHora();
      int getMinuto();
      int getSegundo();
      void imprime();
    private:
      int hora;
      int minuto;
      int segundo;
  };
#endif
```

```
//conteúdo do arquivo horario.cpp
#include <iostream>
using std::cout;
#include <iomanip>
using std::setfill;
using std::setw;
#include "horario.h"
Horario::Horario( int hor, int min, int seg )
{
  setHorario( hor, min, seg );
}
Horario &Horario::setHorario( int h, int m, int s )
{
  setHora( h );
  setMinuto( m );
  setSegundo( s );
  return *this;
}
Horario &Horario::setHora( int h )
{
  hora = ( h >= 0 && h < 24 ) ? h : 0;
  return *this;
}
Horario &Horario::setMinuto( int m )
{
  minuto = ( m >= 0 && m < 60 ) ? m : 0;
  return *this;
}
```

```
Horario &Horario::setSegundo( int s )
{
   segundo = ( s >= 0 && s < 60 ) ? s : 0;
   return *this;
}
int Horario::getHora()
{
   return hora;
}
int Horario::getMinuto()
{
   return minuto;
}
int Horario::getSegundo()
{
   return segundo;
}
void Horario::imprime()
{
   cout << setfill( '0' ) << setw( 2 ) << hora << ":"
      << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
}
```

```
//conteúdo do arquivo principalHorario.cpp
#include <iostream>
using std::cout;
using std::endl;
#include "horario.h"
int main()
{
   Horario t;
   t.setHora( 18 ).setMinuto( 30 ).setSegundo( 22 );
   cout << “Horario: ";
   t.imprime();
   cout << endl << endl << “Novo horario: ";
   t.setHorario( 20, 20, 20 ).imprime();
   cout << endl;
   return 0;
}
```


UNIVASF

Herança:

Como vimos anteriormente, um tipo de dado definido como classe pode dar origem a outro tipo mediante o mecanismo de derivação por especialização denominado *herança*.

Através dele, uma nova classe pode ser definida aproveitando-se o que uma classe já tem, acrescendo-se detalhes de modo a especializar a descrição (torná-la menos abstrata, mais detalhada).

A linguagem C++ disponibiliza a possibilidade de implementarmos este conceito da OO.

Para uma compreensão adequada analisaremos um exemplo.

Herança (continuação):

Exemplo: Em uma determinada aplicação de aviação temos a classe denominada Aviao. Esta tem definido em si uma função membro denominado curva, além de um construtor, funções set e get, e um membro de dados denominado curso. A classe Aviao trata de atividades ou informações pertinentes a qualquer tipo de máquina voadora.

Através dela, uma nova classe pode ser definida aproveitando-se o que esta classe já tem, acrescendo-se detalhes de modo a especializar a descrição (torná-la menos abstrata, mais detalhada).

Veremos agora dois arquivos, um com a interface e outro com a implementação das funções membros da classe Aviao.

Linguagem de Programação C++

```
// conteúdo do arquivo aviao.h
#ifndef AVIAO_H
   #define AVIAO_H
   class Aviao
   {
      public:
         Aviao (void);
         void setCurso (int);
         int getCurso (void);
         void curva (int);
      private:
         int curso;
   };
#endif
```

Linguagem de Programação C++

```
// conteúdo do arquivo aviao.cpp
#include “aviao.h”
Aviao::Aviao (void)
{
  setCurso(0);
}
void Aviao::setCurso (int graus)
{
  curso = graus;
}
int Aviao::getCurso (void)
{
  return curso;
}
void Aviao::curva (int graus)
{
  setCurso(getCurso () + graus);
}
```


UNIVASF

Herança (continuação):

Porém a tipos especiais de aviões que executam atividades especiais e requerem informações especiais.

Por exemplo, um planador executa atividades especiais como soltar o cabo que o reboca e consequentemente tem que registrar a informação referente ao cabo de reboque, se este está ou não ligado (conectado) nele.

Logo, podemos definir uma nova classe Planador derivada da classe Aviao. Planador terá, além das funções get e set, uma função membro denominada soltarCabo e um membro de dados denominado seConectado.

Linguagem de Programação C++

```
// conteúdo do arquivo maqvoadoras.h
#ifndef MAQVOADORAS_H
  #define  MAQVOADORAS_H
  class Aviao
  {
    public:
      Aviao (void);
      void setCurso (int);
      int getCurso (void);
      void curva (int);
    private:
      int curso;
  };
  class Planador:Aviao
  {
    public:
      void setSeConectado (char);
      char getSeConectado (void);
      void soltarCabo(void);
    private:
      char seConectado;
  };
#endif
```

Linguagem de Programação C++

```
// conteúdo do arquivo maqvoadoras.cpp
#include "maqvoadoras.h"
Aviao::Aviao (void)
{
  setCurso(0);
}
void Aviao::setCurso (int graus)
{
  curso = graus;
}
int Aviao::getCurso (void)
{
  return curso;
}
void Aviao::curva (int graus)
{
  setCurso(getCurso () + graus);
}
```

Linguagem de Programação C++

```
void Planador::setSeConectado (char estado)
{
  seConectado = estado;
}

char Planador::getSeConectado (void)
{
  return seConectado;
}

void Planador::soltarCabo()
{
  setSeConectado(0);
}
```

Exercício:

Construa um programa driver que se utilize das classes definidas explorando suas interfaces.

Herança (continuação):

Os objetos da classe Planador, herdam da classe Aviao suas funções membros e seus membros de dados, contudo, em C++, existem os especificadores de modo acesso, os quais vimos anteriormente e neste ponto aprofundaremos nossa análise sobre estes.

Se em nosso exemplo anterior, implementarmos uma função membro imprimirCurso na classe Planador e esta tentar manipular diretamente o membro de dados curso, ou seja:

```
...
void Planador::imprimirCurso()
{
  cout << curso;
}
```

Herança (continuação):

Outra observação necessária é quanto às funções membros herdadas.

Com a definição das classes vistas anteriormente, um objeto da classe Planador não pode atender a mensagem **curva**, ou seja, se em um programa driver constar o seguinte trecho de código:

```
...
int main()
{
   ...
   Planador p1;
   p1.curva(180);
   /* 'void planador:: curva(int)' is inaccessible */
   ...
```


UNIVASF

Herança (continuação):

Sempre que desejarmos que um membro de dados ou uma função membro de uma superclasse não se torne oculto em suas subclasses devemos especificar este não como **private** mas sim como **protected**.

Em nosso exemplo, devemos fazer:

```
class Aviao
{
   public:
      ...
   protected:
      int curso;
};
```

Herança (continuação):

Neste caso, o membro de dados **curso** torna-se privado na classe deriva (Planador), podendo ser manipulado pelas funções membros da mesma e a função membro **curva** torna-se uma função membro privada da classe Planador, podendo ser utilizada da seguinte forma:

```
class Planador:Aviao
{
  ...
  public:
    void curvaPlanador(int);
};
...
void Planador::curvaPlanador(int c)
{
  curva(c);
}
```

Herança (continuação):

Isto ocorre em função do especificador de acesso assumido por omissão ser o **private**, ou seja, é como se tivéssemos derivado **Aviao** da seguinte forma:

class Planador: **private** Aviao { ... };

Neste caso, como vimos, todos os membros de dados e funções membros privadas da superclasse são herdados como ocultos e os protegidos e públicos são herdados como privados.

Herança (continuação):

Os outros especificadores de acesso (**protected** e **public**) podem ser utilizados no processo de derivação, gerando os seguintes efeitos:

O **protected** especifica que todos os membros *protegidos* e *públicos* da classe base tornam-se protegidos na classe derivada. Ou seja, as funções membros da classe derivada têm acesso aos membros de dados originalmente protegidos ou públicos da classe base, mas os membros de dados públicos não mais serão visíveis externamente.

Herança (continuação):

Ao utilizarmos o **public**:

class planador: **public** aviao { ... };

Poderíamos fazer:

```
...
Planador p1;
p1.curva(180);
...
```

Pois, **public** estabelece que todos os membros protegidos e públicos da classe base mantêm sua condição na classe derivada. Este é o método de recepção mais usado, por ser o mais intuitivo.

Usa-se **protected** ou **private** quando se deseja restringir o acesso às classes derivadas.

Herança (continuação):

Observação: membros de classe com especificador de acesso private, sempre são herdados como ocultos.

UNIVASF

Exercício:

Com base na classe **Ponto2D**, complemente o arquivo cabeçalho especificado anteriormente, definindo uma nova classe denominada **Circulo**, que além dos membros de dados herdados de **Ponto2D**, terá o membro de dados raio e deve ser capaz de receber todas as mensagens definidas em Ponto2D e ainda as mensagem **area** e **pontoPertenceAoCirculo**. Construa um programa driver que se utilize adequadamente de objetos instanciados de ambas as classes, explorando totalmente suas interfaces, exceto as funções sets e gets.