Universidade Tecnológica Federal do Paraná
UTFPR – Campus Curitiba

Orientação a Objetos
Programação em C++

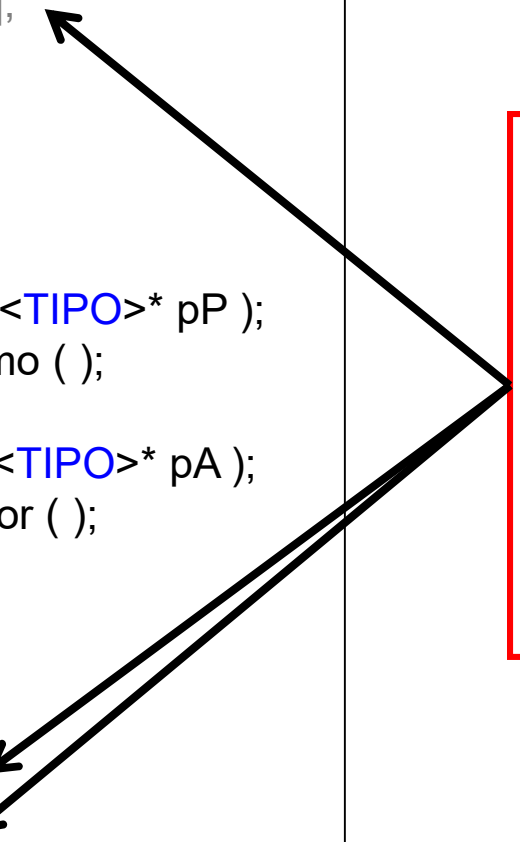
Grupo de Slides 10 – Parte B:
Utilização efetiva e desacoplada de
Lista encadeada com *templates*

Prof. Dr. Jean Marcelo **SIMÃO**
Aluno monitor (2013): Fabiano Cezar Domingos

Melhorando a solução apresentada no grupo de slides 10 – Parte A...

Elemento.h

```
template<class TIPO>
class Elemento
{
private:
    Elemento<TIPO>*         pProximo;
    Elemento<TIPO>*         pAnterior;
    TIPO*                   pInfo;
    //char                  nome[150];

public:
    Elemento ( );
    ~Elemento ( );

    void setProximo ( Elemento<TIPO>* pP );
    Elemento<TIPO>* getProximo ( );

    void setAnterior ( Elemento<TIPO>* pA );
    Elemento<TIPO>* getAnterior ( );

    void setInfo ( TIPO* pI );
    TIPO* getInfo ( );

    //void setNome ( char* n );
    //char* getNome ( );
};
```



Atributo **nome** da classe Elemento não deve existir. Deve-se evitar a replicação de dados!

Dados necessários para uma posterior listagem devem ser obtidos através do ponteiro **pInfo**.

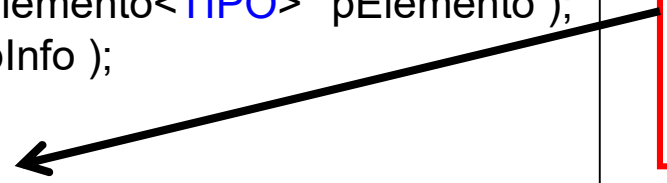
Elemento.h

```
template<class TIPO>
Elemento<TIPO>::Elemento ( )
{
   pProximo = NULL;
   pAnterior = NULL;
   pInfo =   NULL;
}

template<class TIPO>
Elemento<TIPO>::~Elemento ( )
{
   pProximo = NULL;
   pAnterior = NULL;
   pInfo =      NULL;
}

template<class TIPO>
void Elemento<TIPO>::setProximo (
                              Elemento<TIPO>* pP )
{
   pProximo = pP;
}

template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getProximo ( )
{
   return pProximo;
}
```

```
template<class TIPO>
void Elemento<TIPO>::setAnterior (
                                    Elemento<TIPO>* pA )
{
   pAnterior = pA;
}

template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getAnterior ( )
{
   return pProximo;
}

template<class TIPO>
void Elemento<TIPO>::setInfo (TIPO* pI )
{
   pInfo = pI;
}

template<class TIPO>
TIPO* Elemento<TIPO>::getInfo()
{
   return pInfo;
}
```

Lista.h

```
template<class TIPO>
class Lista
{
private:
    Elemento<TIPO>*  pPrimeiro;
    Elemento<TIPO>*  pAtual;

public:

    Lista ( );
    ~Lista ( );

    void inicializa ( );
    void limpar();

    bool incluaElemento ( Elemento<TIPO>* pElemento );
    bool incluaInfo (TIPO* pInfo );

    //void listeInfos ( );


    Elemento<TIPO>* getpPrimeiro();
    Elemento<TIPO>* getpAtual();

};
```



A Lista *template* **não** deve possuir métodos para listar os seus dados. Não há como generalizar os algoritmos de listagem.

Lista.h

```
template<class TIPO>
Lista<TIPO>::Lista ( )
{
     inicializa ( );
}

template<class TIPO>
Lista<TIPO>::~Lista ( )
{
     limpar();
}

template<class TIPO>
void Lista<TIPO>::inicializa ( )
{
    pPrimeiro = NULL;
    pAtual    = NULL;
}

template<class TIPO>
Elemento<TIPO>* Lista<TIPO>::getpPrimeiro()
{
    return pPrimeiro;
}

template<class TIPO>
Elemento<TIPO>* Lista<TIPO>::getpAtual()
{
    return pAtual;
}
```

```
template<class TIPO>
void Lista<TIPO>::limpar ( )
{
  Elemento<TIPO>* paux1;
  Elemento<TIPO>* paux2;

  paux1 = pPrimeiro;
  paux2 = paux1;

  while (paux1 != NULL)
  {
                paux2 = paux1->getProximo();
                delete (paux1);
                paux1 = paux2;
  }

  pPrimeiro = NULL;
  pAtual    = NULL;
}
```

Lista.h

```
template<class TIPO>
bool Lista<TIPO>::incluaInfo (TIPO *pInfo )
{

    if (NULL != pInfo)
    {

        Elemento<TIPO>* pElemento = NULL;

        pElemento = new Elemento<TIPO>();

        pElemento->setInfo (pInfo);

        incluaElemento (pElemento);

        return true;

    }
    else
    {

        printf ( "Erro, elemento nulo(a) na lista. \n" );

        return false;

    }

}
```

```
template<class TIPO>
bool Lista<TIPO>::incluaElemento ( Elemento<TIPO>*
pElemento )
{
   if (NULL != pElemento)
   {

      if (NULL == pPrimeiro)
      {
         pPrimeiro = pElemento;
         // pPrimeiro->setAnterior ( NULL );
         // pPrimeiro->setProximo ( NULL );
         pAtual = pPrimeiro;
      }
      else
      {
         pElemento->setAnterior ( pAtual );
         //pElemento->setProximo ( NULL );
         pAtual->setProximo ( pElemento );
         pAtual = pAtual->getProximo ( );
      }
      return true;
   }
   else
   {
      cout << "Erro, elemento nulo na lista." << endl;
      return false;
   }
}
```

ListaAlunos.h

```
class ListaAlunos
{
private:

   //template class
   Lista< Aluno > LTAlunos;
public:

   ListaAlunos();
   ~ListaAlunos();

   void limpaLista();
   void incluaAluno(Aluno* pa);

   void listeAlunos();
   void listeAlunos2();

   void graveAlunos();
   void recupereAlunos();
};
```

A classe **ListaAlunos** armazenará os apontamentos de aluno por meio de uma Lista *template* parametrizada com Aluno.

Os métodos limpar() e incluaAluno() chamam diretamente os métodos da Lista *template*, visto que o código é generalizado.

Já o algoritmos de listagem para os Alunos são específicos e são definidos nesta classe.

```
ListaAlunos::ListaAlunos()
{ }

ListaAlunos::~ListaAlunos()
{
    limpaLista();
}

void ListaAlunos::limpaLista()
{
    LTAlunos.limpar();
}

void ListaAlunos::incluaAluno ( Aluno* pa )
{
   if  ( NULL != pa )
   {
       LTAlunos.incluaInfo(pa);
   }
   else
   {
       cout << "Erro! Aluno não includes.";
       cout << "Ponteiro Aluno inválido." << endl;
    }
}

void ListaAlunos::graveAlunos()
{ … }

void ListaAlunos::recupereAlunos()
{ … }
```

ListaAlunos.cpp

```
void ListaAlunos::listeAlunos()
{
   Elemento<Aluno>* pElAux = NULL;
   Aluno* pAlAux = NULL;
   pElAux = LTAlunos.getpPrimeiro();

   while (NULL != pElAux)
   {
      pAlAux = pElAux->getInfo();
      cout << " Aluno " << pAlAux->getNome()
           << " com RA " << pAlAux->getRA()
           << "." << endl;
      pElAux = pElAux->getProximo();
   }
}
void ListaAlunos::listeAlunos2()
{
   Elemento<Aluno>* pElAux = NULL;
   Aluno* pAlAux = NULL;
   pElAux = LTAlunos.getpAtual();

   while (NULL != pElAux)
   {
     pAlAux = pElAux->getInfo();
     cout << " Aluno " << pAlAux->getNome()
          << " com RA " << pAlAux->getRA()
          << "." << endl;
     pElAux = pElAux->getAnterior();
   }
}
```

Disciplina.h

```
class Disciplina
{
private:
    int id;
    int cont_alunos;
    int numero_alunos;
    char nome[150];
    char area_conhecimento[150];

    Departamento* DeptoAssociado;
    ListaAlunos ObjLAlunos;

  public:
    Disciplina(int i, int na = 45, char* ac = "");
    ~Disciplina();

    void setId(int i);
    int getId();
    void setNome(char* n);
    char* getNome();
    void setDepartamento(Departamento* d);
     Departamento* getDepartamento();

    void incluaAluno(Aluno* pa);
    void listeAlunos();
    void listeAlunos2();
};
```

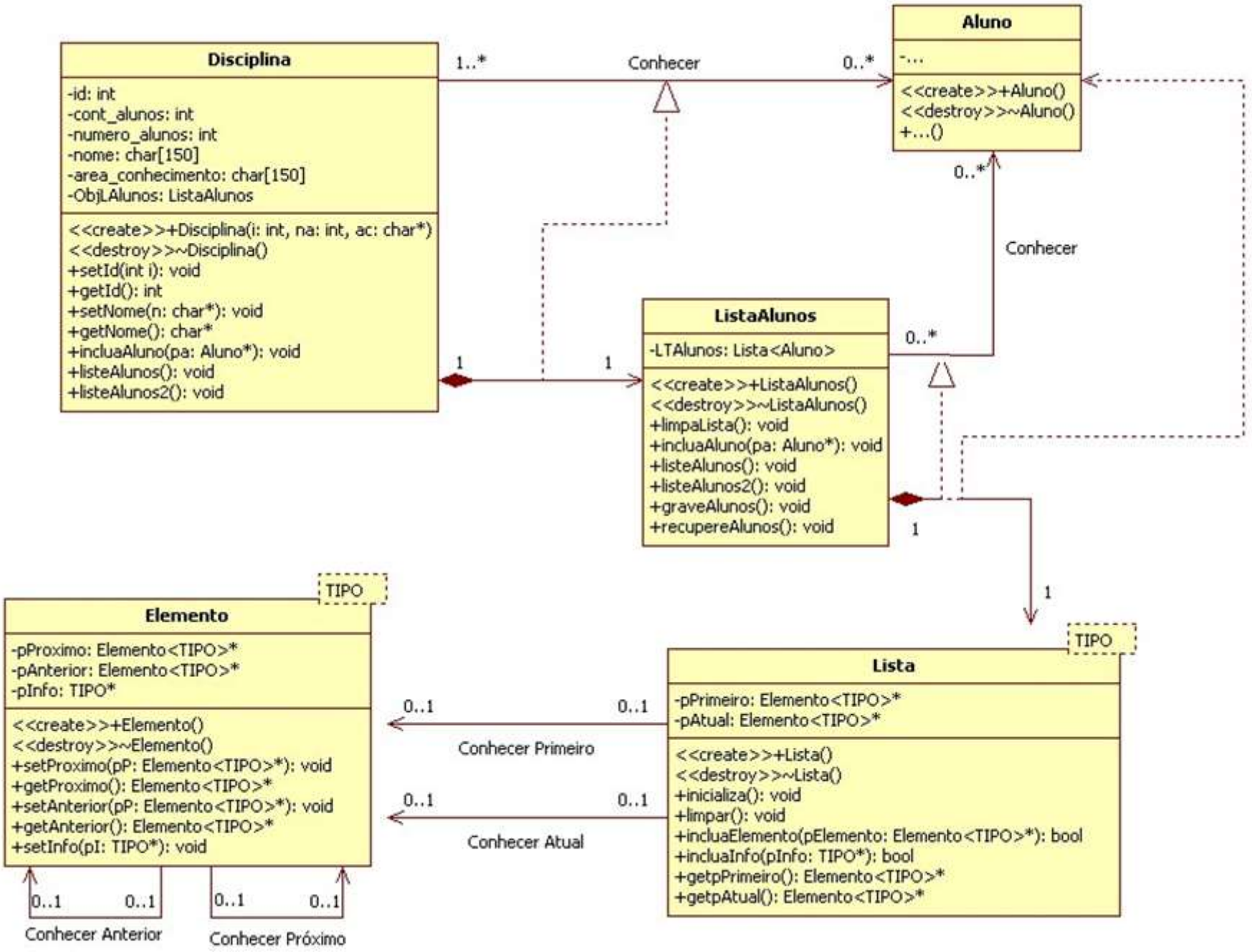
A classe **Disciplina** irá gerenciar os seus ponteiros de **Aluno** por meio de **ObjLAlunos**.

Como os métodos de inclusão e listagem já foram definidos na classe **ListaAlunos**, basta chamá-los diretamente.

Disciplina.cpp

```
Disciplina::Disciplina(int i, int na, char* ac)
{
   id = i;
   cont_alunos = 0;
   numero_alunos = na;
   DeptoAssociado = NULL;
   strcpy (area_conhecimento, ac );
}

Disciplina::~Disciplina()
{
     DeptoAssociado  = NULL;
}

…

void Disciplina::listeAlunos()
{
   cout << "\n\nAlunos matriculados na disciplina "
        << area_conhecimento << ": "<< endl;
   ObjLAlunos.listeAlunos();
}

void Disciplina::listeAlunos2()
{
   cout << "\n\nAlunos matriculados na disciplina "
        << area_conhecimento << ": "<< endl;
   ObjLAlunos.listeAlunos2();
}
```

```
void Disciplina::incluaAluno ( Aluno* pa )
{
    if (NULL != pa)
    {
        if ( ( cont_alunos < numero_alunos ) ||
             ( -1 == numero_alunos))
        {
              ObjLAlunos.incluaAluno(pa);
              cont_alunos++;
        }
        else
        {
              cout << "Aluno não incluído. “
                   << "Turma já lotada em "
                   << numero_alunos
                   << " alunos." << endl;
        }

   }
   else
   {
        cout << "Erro! Aluno não incluído. "
             << "Ponteiro Aluno inválido."
             << endl;
   }
}
```


Disciplina
-id: int
-cont_alunos: int
-numero_alunos: int
-nome: char[150]
-area_conhecimento: char[150]
-ObjLAlunos: ListaAlunos
<<create>>+Disciplina(i: int, na: int, ac: char*)
<<destroy>>~Disciplina()
+setId(int i): void
+getId(): int
+setNome(n: char*): void
+getNome(): char*
+incluaAluno(pa: Aluno*): void
+listeAlunos(): void
+listeAlunos2(): void
1..*
Conhecer
0..*
Aluno
-...
<<create>>+Aluno()
<<destroy>>~Aluno()
+...()
0..*
Conhecer
1
1
ListaAlunos
-LTAlunos: Lista<Aluno>
<<create>>+ListaAlunos()
<<destroy>>~ListaAlunos()
+limpaLista(): void
+incluaAluno(pa: Aluno*): void
+listeAlunos(): void
+listeAlunos2(): void
+graveAlunos(): void
+recupereAlunos(): void
0..*
1
1
TIPO
Elemento
-pProximo: Elemento<TIPO>*
-pAnterior: Elemento<TIPO>*
-pInfo: TIPO*
<<create>>+Elemento()
<<destroy>>~Elemento()
+setProximo(pP: Elemento<TIPO>*): void
+getProximo(): Elemento<TIPO>*
+setAnterior(pP: Elemento<TIPO>*): void
+getAnterior(): Elemento<TIPO>*
+setInfo(pI: TIPO*): void
0..1
0..1
Conhecer Primeiro
0..1
0..1
Conhecer Atual
TIPO
Lista
-pPrimeiro: Elemento<TIPO>*
-pAtual: Elemento<TIPO>*
<<create>>+Lista()
<<destroy>>~Lista()
+inicializa(): void
+limpar(): void
+incluaElemento(pElemento: Elemento<TIPO>*): bool
+incluaInfo(pInfo: TIPO*): bool
+getpPrimeiro(): Elemento<TIPO>*
+getpAtual(): Elemento<TIPO>*
0..1
0..1
Conhecer Anterior
0..1
0..1
Conhecer Próximo

Exercícios Propostos

a) Substituir todas as outras listas ‘específicas’ elaboradas nas versões anteriores do ‘sistema exemplo’, por ‘instâncias’ de listas cujo cerne provenha de gabarito ou *template* desenvolvido para o tratamento de listas.

b) Estender em classe especializada cada lista baseada em *template* para que suas instâncias tenham a ‘capacidade’ de gravar e recuperar as informações incluídas nela. Neste sentido, por exemplo, a classe *ListaAlunos*, seria derivada em *ListaAlunosGravadora*.

c) Retomando um exercício anterior: - Elaborar uma solução para armazenar as notas (1ª parcial, 2ª parcial e final) e número de faltas de cada aluno em cada disciplina.

Exercícios Suplementares

- 1) Conforme o subsequente grupo de Slides 12, fazer com que *Elemento<Tipo>* seja uma classe aninhada (privada) de *Lista<Tipo>*.

  – Exemplo de classe Lista template com classe Elemento aninhada disponível em:

  http://www.dainf.ct.utfpr.edu.br/~jeansimao/ProgramacaoAvancada/ProgAvancada.htm

    - Vide ali exemplo de prova disponível.

- 2) Substituir o objeto de Lista *template* **LTAlunos** na classe **ListaAlunos** por um *vector* ou *list* da STL, seguindo o grupo de Slides 16.