

TDA

- Especificación de TDAs en Nereus
- Clases de C++
- Pasaje de Nereus a clases
- Clases parametrizadas
- Ejemplos

Definición de TDA

- Desde el punto de vista de la implementación:
 - Los TDA involucran un conjunto de **valores** (miembros) y **operaciones** para manipularlos (métodos).
 - Proveen un mecanismo de niveles de acceso que permite el **encapsulamiento de datos**.
 - Al encapsular los datos, es necesario definir **una interfaz formada por los métodos** públicos.
 - Así se consigue **la abstracción de datos**:
 - Una interfaz que determina el comportamiento
 - Una implementación y estructuras de datos ocultas que pueden cambiar independientemente de la interfaz

Especificación de los TDAs

- El objetivo de la especificación es describir la interfaz del TDA.
- Se define el comportamiento enfocándose en determinar **qué** hace cada operación.
- **No** se debe entrar en los detalles de **cómo** se implementará el TDA. Un TDA puede tener varias implementaciones que se ajustan a una misma interfaz.
- Para esto, utilizaremos un lenguaje de especificación formal algebraico llamado **Nereus**.
- Nereus permite definir los tipos de datos y las operaciones para manipularlos.

Nereus (1)

- Hasta la cláusula OPERATIONS, se incluye todo lo necesario para definir el nombre del TDA, los tipos de los cuales depende o extiende, así como las operaciones con su interfaz completa (nombre, número y tipo de parámetros y el tipo de retorno).
- En IMPORTS no es necesario listar los tipos básicos (Real, Boolean, Integer, etc)

```
CLASS Punto
IMPORTS Real, Boolean
BASIC CONSTRUCTORS crear
EFFECTIVE
TYPE Punto
OPERATIONS
    crear: Real * Real → Punto;
    coordx: Punto → Real;
    coordy: Punto → Real;
    trasladar: Punto * Real * Real → Punto;
    distancia: Punto * Punto → Real;
    ==: Punto * Punto → Boolean;
```

...

Nereus (2)

- La cláusula AXIOMS permite especificar sin ambigüedades **qué** debe hacer cada operación.

```
...
AXIOMS  x, y, x1, y1: Real;
          coordx(crear(x,y)) = x;
          coordy(crear(x,y)) = y;
          trasladar(crear(x,y), x1, y1) = crear(x+x1, y+y1);
          (crear(x,y) = crear(x1,y1)) = IF (x=x1 AND y=y1)
THEN true ELSE false ENDIF;
          distancia(crear(x,y), crear(x1,y1)) = ((y1-y)^2 +
(x1-x)^2)^1/2;
END_CLASS
```

Se asume: **CLASS** Real

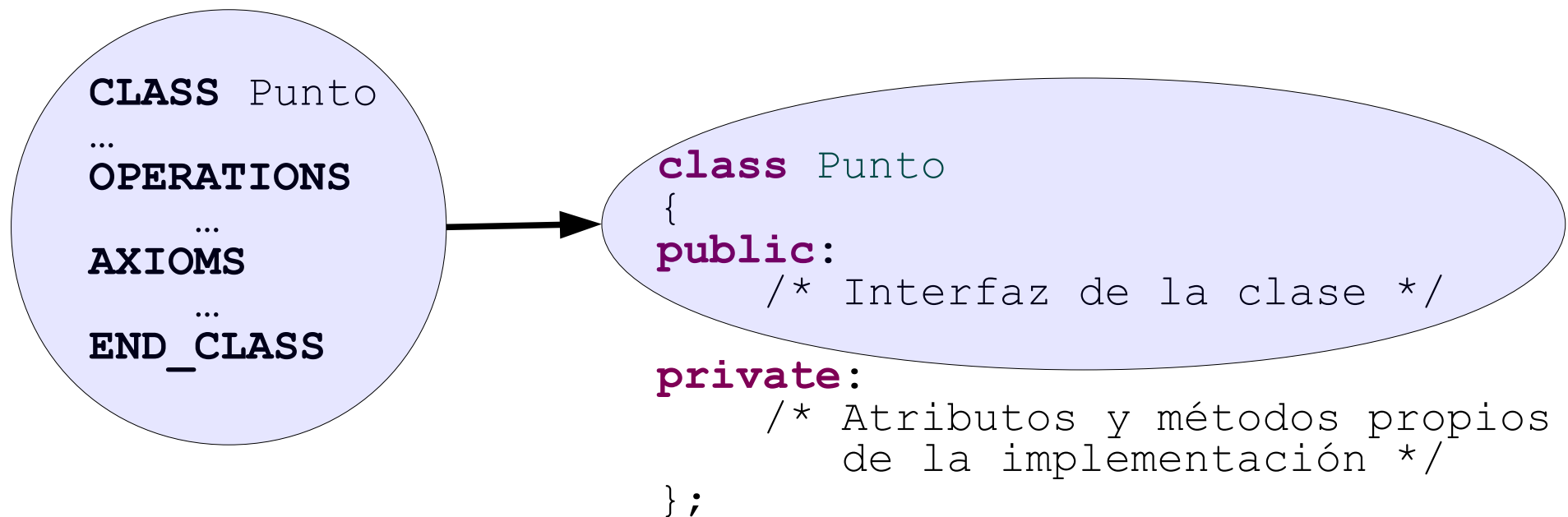
```
...
OPERATIONS
  +_ : Real * Real → Real;
  -_ : Real * Real → Real;
  ^_ : Real * Real → Real;
  =_ : Real * Real → Boolean;
...
END_CLASS
```

Clases

- Los TDA serán implementados como **clases** de C++.
 - El mecanismo de clases es una de las extensiones principales que incorpora el lenguaje C++, a la programación procedural propia de C.
- A partir de las clases se definen nuevos tipos de datos.
- Las clases están formadas por *miembros* y *métodos* de instancia.
- Un **objeto** es una *instancia* de una clase determinada.
- Los objetos son los encargados de procesar los mensajes recibidos.
- Mediante los métodos se puede manipular los datos propios de los objetos.

De Nereus a C++ (1)

- Generalmente, cada TDA especificado se implementará como una clase nueva en C++.
- A partir de la especificación se puede traducir fácilmente la *interfaz* de la clase.



De Nereus a C++ (2)

...
OPERATIONS
crear: Real * Real → Punto;
coordx: Punto → Real;
coordy: Punto → Real;
trasladar: Punto * Real * Real → Punto;
distancia: Punto * Punto → Real;
=: Punto * Punto → Boolean;

AXIOMS

...



...
public:
Punto(float x, float y);
float coordX() const;
float coordY() const;
void trasladar(float x, float y);
float distancia(const Punto & otroPunto) const;
bool operator==(const Punto & otroPunto) const;

...

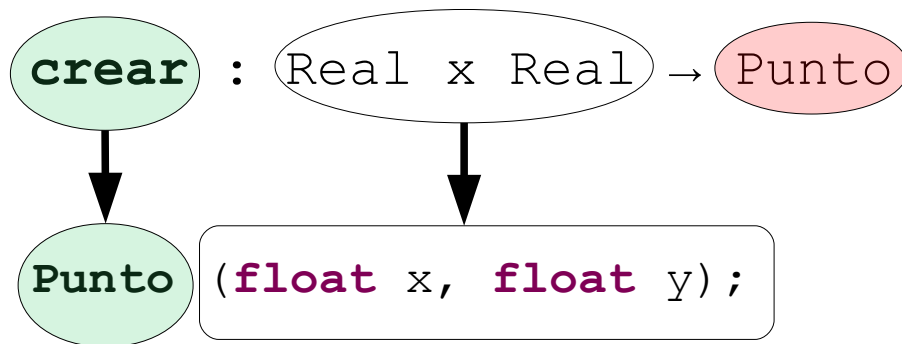
Interfaz: Constructores

- Los objetos necesitan inicializar los miembros y estructuras internas que determinarán su estado inicial.
- Para este fin, las clases poseen funciones especiales llamadas **constructores** que tienen el *mismo nombre de la clase*.

CLASS Punto

...

BASIC CONSTRUCTORS crear
OPERATIONS



Punto pA(1,2);

Punto pB(7,5);

Punto * pC = new Punto(3,4);

Interfaz:

Tipos de métodos

- **Observadores**

- sólo consultan la información acerca del estado de los objetos *sin modificarlos*.

- **Modificadores**

- alteran el estado interno de los objetos; es decir, modifican el valor de sus atributos.

- En C++ los métodos de consulta llevan **const** a continuación de los parámetros.

```
float coordX() const;    // Deben ser métodos de  
float coordY() const;    // una clase para ser const
```

```
void trasladar(float x, float y);
```

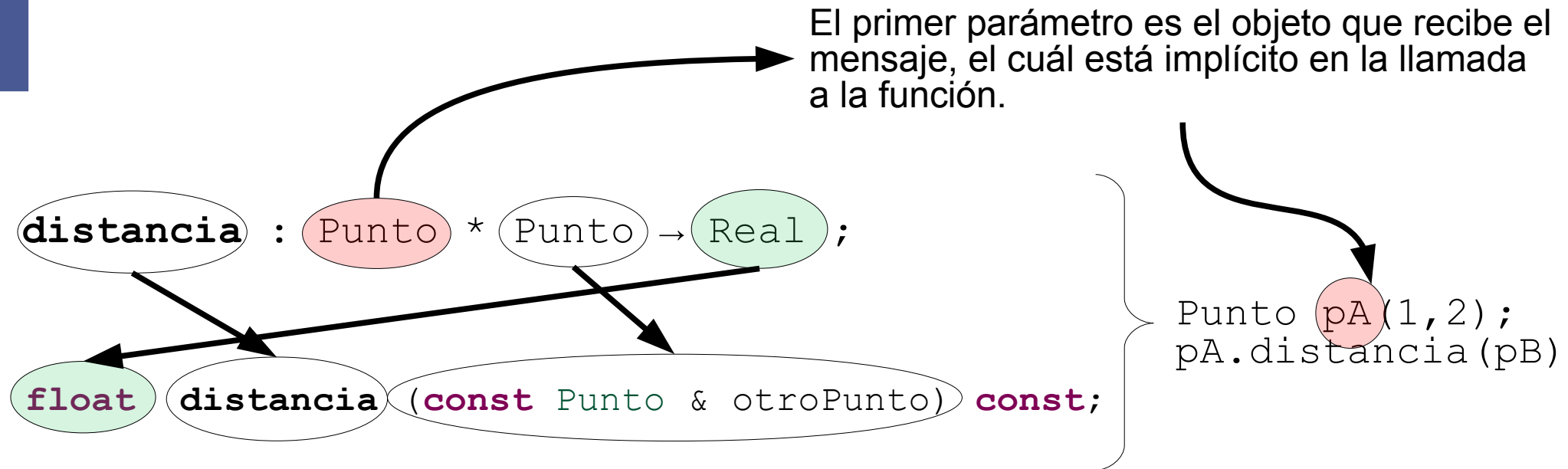
- Cuando se utiliza una referencia constante (**const** <Clase> &) a un objeto, sólo se puede invocar a los métodos de consulta.

```
float distancia(const Punto & otroPunto) const;  
bool operator==(const Punto & otroPunto) const;
```

Interfaz:

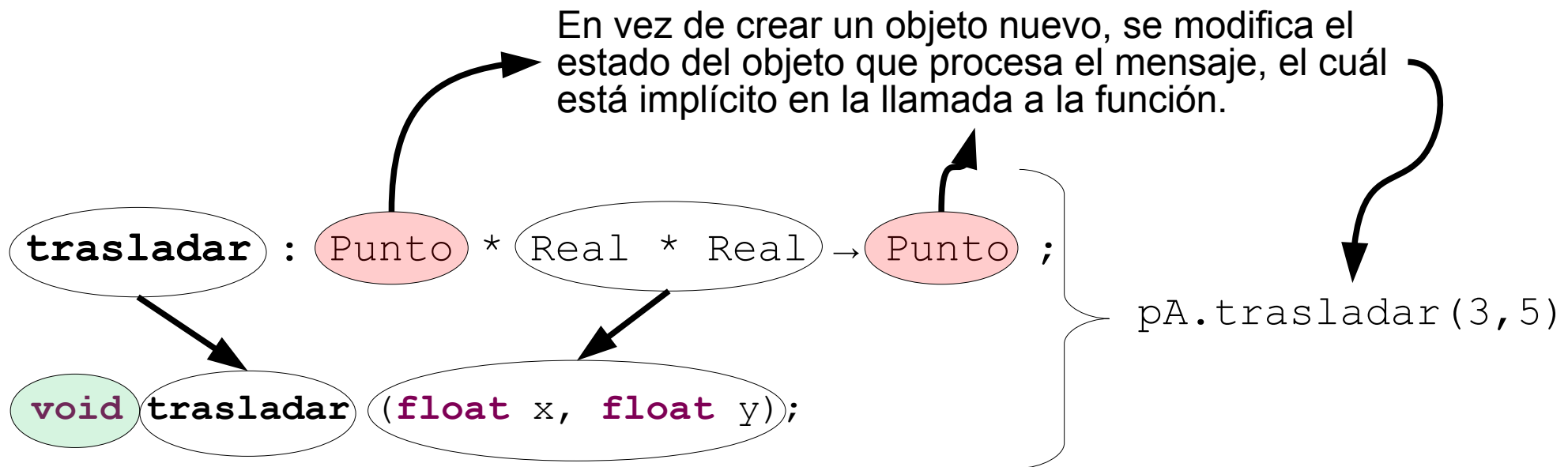
Métodos observadores

- Todas las funciones en Nereus -excepto las constructoras- reciben como primer parámetro un elemento del tipo del TDA que se está especificando.
- En los métodos de las clases este parámetro está implícito, ya que se trata del objeto que procesa el mensaje.



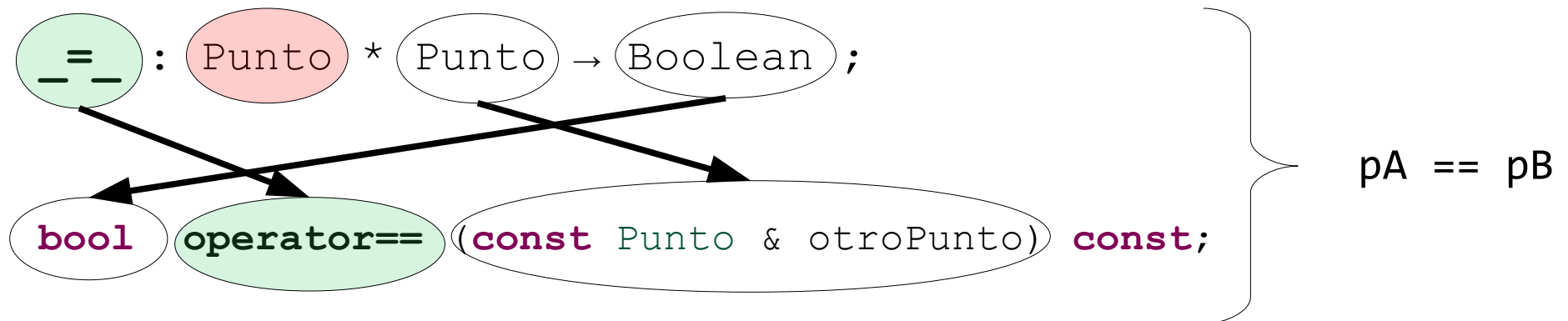
Interfaz: Métodos modificadores

- En la especificación, las funciones modificadoras son funciones que generan nuevos términos del álgebra.
- A nivel de implementación, las operaciones modifican los atributos propios del objeto que procesa el mensaje.



Interfaz: Operadores

- Los **operadores** son métodos que se invocan con una sintaxis especial.
- C++ permite (re)definir el comportamiento de muchos operadores en el contexto de las clases.
- Siguen las mismas reglas que los métodos consultores y modificadores.



Objetos: instanciando las clases

- Los métodos son mensajes que procesan los objetos.
- A través de los métodos es que se consulta o modifica el estado de los objetos. Así se evita acceder directamente a la estructuras de datos internas.

```
Punto pA(1.3,2.5);
Punto pB(7.9,5.0);
cout << "Punto A (" << pA.coordX() << "," << pA.coordY() << ")\n";
cout << "Punto B (" << pB.coordX() << "," << pB.coordY() << ")\n";
cout << "La distancia entre A y B es " << pA.distancia(pB) << "\n";
if (!(pA == pB)) { // Operador ==
    Punto pC(pA.coordX(), pA.coordY());
    pA.trasladar(pB.coordX()-pA.coordX(), pB.coordY()-pA.coordY());
    pB.trasladar(pC.coordX()-pB.coordX(), pC.coordY()-pB.coordY());
}
cout << "Punto A (" << pA.coordX() << "," << pA.coordY() << ")\n";
cout << "Punto B (" << pB.coordX() << "," << pB.coordY() << ")\n";
```

Implementación: Definición de los atributos

- Lo primero a determinar son los atributos que constituirán el estado de los objetos.
- Los mismos estarán siempre ocultos a los usuarios de la clase.
 - Se declaran siempre con un nivel de acceso **private** o **protected**; al igual que cualquier método auxiliar que se requiera.

```
class Punto
{
public:
    Punto(float x, float y);
    float coordX() const;
    float coordY() const;
    void trasladar(float x, float y);
    float distancia(const Punto & otroPunto) const;
    bool operator==(const Punto & otroPunto) const;
private:
    float x;
    float y;
};
```

Implementación: Definición de los métodos (1)

- La *definición* de la clase determina la *declaración* de los métodos.
- Una clase está completa si todos sus métodos están **definidos**.
- Los métodos se deben *definir* en el **ámbito de la clase** correspondiente:

```
void Clase::metodo() { ... }
```

- Un método puede acceder a todos los miembros de su clase.
- Además, los métodos pueden acceder a una referencia al objeto que procesa el mensaje: un puntero llamado **this**.
 - Los atributos pueden accederse directamente a través del nombre del miembro e indirectamente a través del puntero **this** (x y this->x, por ejemplo).

Implementación: Definición de los métodos (2)

```
Punto::Punto(float x, float y) {  
    this->x = x;  
    this->y = y;  
}  
  
float Punto::coordX() const {  
    return x;  
}  
  
float Punto::coordY() const {  
    return y;  
}  
  
void Punto::trasladar(float x, float y) {  
    this->x += x;  
    this->y += y;  
}  
  
float Punto::distancia(const Punto & otroPunto) const {  
    return sqrt(pow(x-otroPunto.x, 2.0) + pow(y-otroPunto.y, 2.0));  
}  
  
bool Punto::operator==(const Punto & otroPunto) const {  
    return (x == otroPunto.x) && (y == otroPunto.y);  
}
```

Ejemplo: Segmento (1)

```
CLASS Segmento
IMPORTS Punto, Real, Boolean
BASIC CONSTRUCTORS crear
EFFECTIVE
TYPE Segmento
OPERATIONS
    crear: Punto * Punto → Segmento;
    extremo1: Segmento → Punto;
    extremo2: Segmento → Punto;
    ==: Segmento * Segmento
        → Boolean;
    trasladar: Segmento * Real * Real
        → Segmento;
    longitud: Segmento → Real;
AXIOMS
...
END_CLASS
```



```
class Segmento
{
public:
    Segmento(const Punto & ext1,
             const Punto & ext2);
    const Punto & extremo1() const;
    const Punto & extremo2() const;
    float longitud() const;
    void trasladar(float x, float y);
    bool operator==(const Segmento &
                     otroSegmento) const;
    ...
};
```

Ejemplo: Segmento (2)

```
class Segmento {
    ...
private:
    Punto ext1, ext2;
};

Segmento::Segmento(const Punto & ext1, const Punto & ext2) {
    this->ext1 = ext1;
    this->ext2 = ext2;
}

const Punto & Segmento::extremo1() const {
    return ext1;
}

const Punto & Segmento::extremo2() const {
    return ext2;
}

float Segmento::longitud() const {
    return ext1.distancia(ext2);
}

void Segmento::trasladar(float x, float y) {
    ext1.trasladar(x, y);
    ext2.trasladar(x, y);
}

bool Segmento::operator==(const Segmento & otroSegmento) const {
    return (ext1 == otroSegmento.ext1) && (ext2 == otroSegmento.ext2);
}
```

Clases parametrizadas (1)

- Hay ciertos TDA a los cuales queremos generalizar desde el punto de vista de los tipos de valores que manejan para algunas operaciones.

Por ejemplo:

```
CLASS Punto[Elem : Real]
```

```
...
```

```
OPERATIONS
```

```
  crear: Elem * Elem → Punto;
```

```
  coordx: Punto → Elem;
```

```
  coordy: Punto → Elem;
```

```
  trasladar: Punto * Elem * Elem → Punto;
```

```
...
```

```
AXIOMS
```

```
...
```

```
END CLASS
```

```
CLASS Real
```

```
...
```

```
END_CLASS
```

```
CLASS Entero
```

```
...
```

```
INHERITS Real
```

```
...
```

```
END CLASS
```

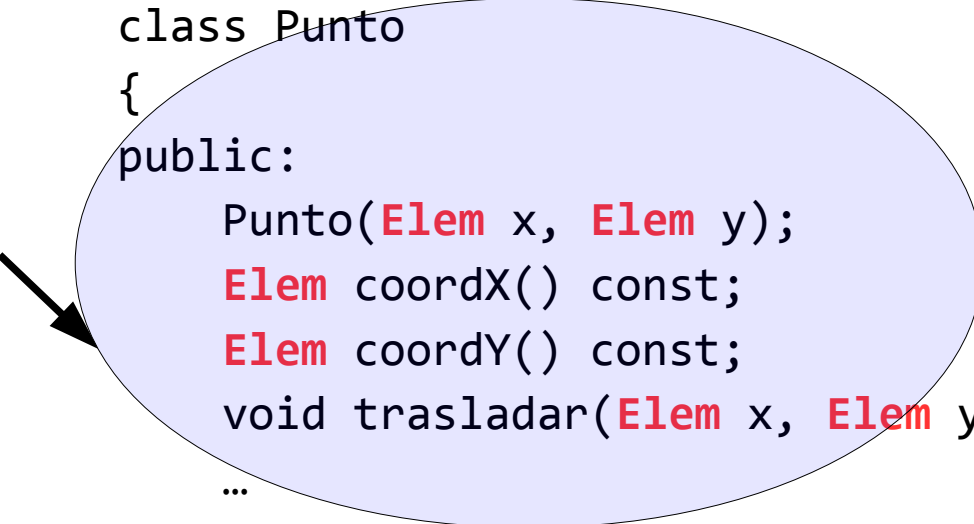
- Esto es especialmente cierto cuando estamos especificando contenedores: tipos que definen estructuras de datos cuya función es almacenar otros objetos.

Clases parametrizadas (2)

- C++ provee un mecanismo de **templates** (o plantillas), que es conceptual y funcionalmente equivalente.

```
CLASS Punto[Elem : Real]
...
OPERATIONS
  crear: Elem * Elem → Punto;
  coordx: Punto → Elem;
  coordy: Punto → Elem;
  trasladar: Punto * Elem * Elem
            → Punto;
...
AXIOMS
...
END_CLASS
```

```
template <typename Elem>
class Punto
{
public:
  Punto(Elem x, Elem y);
  Elem coordX() const;
  Elem coordY() const;
  void trasladar(Elem x, Elem y);
...
private:
  Elem x;
  Elem y;
};
```



Clases parametrizadas (3)

- Que un tipo esté parametrizado no significa que se pueda instanciar utilizando cualquier tipo de datos como parámetro.
- Veamos el ejemplo donde el tipo parametrizado debe ser numérico, debido a que los métodos realizan operaciones matemáticas con los datos:

```
Punto<unsigned int> pA(1,2);
Punto<unsigned int> pB(7,5);
cout << "Punto A (" << pA.coordX() << "," << pA.coordY() << ")\n";
cout << "Punto B (" << pB.coordX() << "," << pB.coordY() << ")\n";
cout << "La distancia entre A y B es " << pA.distancia(pB) << "\n";
```

...

- Al implementar los métodos se debe incluir la **definición de la parametrización** para cada uno:

```
template <typename Elem>
void Punto<Elem>::trasladar(Elem x, Elem y) {
    this->x += x;
    this->y += y;
}
```

Clases parametrizadas y encabezados (1)

- Al implementar una clase parametrizada utilizando encabezados

```
template <typename Elem>
class Punto
{
public:
    Punto(Elem x, Elem y);
    void trasladar(Elem x, Elem y);
    ...
};
```

Punto.h

```
template <typename Elem>
Punto<Elem>::Punto(Elem x, Elem y) {
    ...
}
template <typename Elem>
void Punto<Elem>::trasladar(Elem x, Elem y) {
    this->x += x;
    this->y += y;
}
```

Punto.cpp

```
#include "Punto.h"
...
Punto<unsigned int> pA(1,2);
main.cpp
```

- Esto genera un error en la compilación (undefined reference...)
- Punto.cpp y main.cpp son compilados por separado y en ninguna existe una compilación de la instancia `Punto<unsigned int>`.

Clases parametrizadas y encabezados (2)

- Existen dos soluciones posibles para estos casos:

1) Incluir todo el código de la clase (definición e implementación) en el .h

```
template <typename Elem>
class Punto
{
public:
    Punto(Elem x, Elem y);
    Elem coordX() const;
    Elem coordY() const;
    void trasladar(Elem x, Elem y);
};

template <typename Elem>
Punto<Elem>::Punto(Elem x, Elem y) {
    ...
}

template <typename Elem>
void Punto<Elem>::trasladar(Elem x, Elem y) {
    this->x += x;
    this->y += y;
}
```

Punto.h

Clases parametrizadas y encabezados (3)

- 2) Agregar al final del .cpp las posibles instanciaciones de tipos para la clase que se está implementando.

```
template <typename Elem>
Punto<Elem>::Punto(Elem x, Elem y) {
    ...
}

template <typename Elem>
void Punto<Elem>::trasladar(Elem x,
                             Elem y) {
    this->x += x;
    this->y += y;
}

template class Punto<unsigned int>;
template class Punto<float>;
...
```

Punto.cpp

Ejemplo: Pila (1)

- El TDA Pila representa un contenedor, con la propiedad que los elementos se acceden mediante la secuencia LIFO (*last in, first out* / último que entra es el primero en salir)

```
CLASS Pila[Elem]
IMPORTS Boolean
BASIC CONSTRUCTORS crear, agregar
EFFECTIVE
TYPES Pila
OPERATIONS
    crear: → Pila;
    agregar: Pila * Elem → Pila;
    esvacia: Pila → Boolean;
    tope: Pila(p) → Elem;
        pre: not esvacio(p)
    sacar: Pila(p) → Pila;
        pre: not esvacio(p)

AXIOMS
...
END_CLASS
```

```
template <typename T>
class Pila {
public:
    Pila();
    void agregar(const T & elemento);
    const T & tope() const;
    bool sacar();
    bool es_vacia() const;
    ...
};
```

Ejemplo: Pila (2)

- La decisión de implementación más importante es la utilización de una secuencia de nodos vinculados.
- Debido a esto se crea la estructura *Nodo* y se agrega el destructor.
- En este caso, además de los atributos correspondientes, se incluyen métodos auxiliares y la definición de la estructura en la parte privada.

```
template <typename T>
class Pila {
public:
    Pila();    // Constructor
    ~Pila();   // Destructor
    void agregar(const T & elemento);
    const T & tope() const;
    bool sacar();
    bool es_vacia() const;

private:
    struct Nodo {
        T elemento;
        Nodo * siguiente;
    };

    void vaciar();

    Nodo * tope;
};
```

Ejemplo: Pila (3)

```
#include <cassert>

template <typename T> Pila<T>::Pila() {
    tope = NULL;
}

template <typename T> Pila<T>::~~Pila() {
    vaciar();
}

template <typename T> void Pila<T>::agregar(const T & elemento) {
    Nodo * aux = new Nodo;
    aux->elemento = elemento;
    aux->siguiente = tope;
    tope = aux;
}

template <typename T> const T & Pila<T>::tope() const {
    assert(!es_vacia());
    return tope->elemento;
}

template <class T> bool Pila<T>::sacar() {
    if (tope != 0) {
        Nodo * aux = tope;
        tope = tope->siguiente;
        delete aux;
        return true;
    } else
        return false;
}
```

Ejemplo: Pila (4)

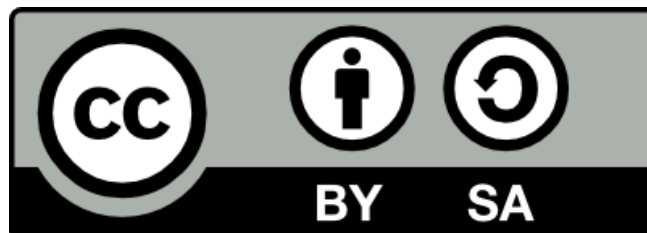
```
template <class T> bool Pila<T>::es_vacia() const {  
    return (tope == NULL);  
}  
  
template <typename T> Pila<T>::vaciar() {  
    Nodo * aux;  
    while (tope != NULL) {  
        aux = tope->siguiente;  
        delete tope;  
        tope = aux;  
    }  
    tope = NULL;  
}
```

- En el código se ven dos formas en las que se resolvieron las precondiciones de las operaciones sacar y tope:
 - En un caso se utilizó la función assert, la cual si no se verifica la condición termina el programa con un error.
 - En el otro, se modificó el comportamiento especificado para que se evite eliminar un elemento cuando la estructura se encuentra vacía.

Consultas: laboratorio.ayda@alumnos.exa.unicen.edu.ar

Licencia creative commons

Atribución-Compartir Obras Derivadas Igual 2.5 Argentina



<http://creativecommons.org/licenses/by-sa/2.5/ar/>