

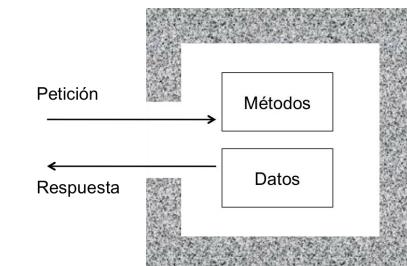


PROGRAMACIÓN II

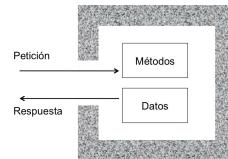
Tema 2

Tipos Abstractos de Datos

1. Modularidad
2. Tipo abstracto de datos
3. Clases, objetos



Tema 2. TAD
marzo de 2019



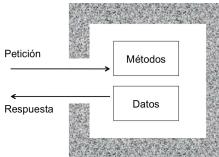
Modularidad

- **La modularidad**

- Mantiene la complejidad de un programa grande dentro de unos límites manejables:
 - Controlando la interacción de sus componentes.
- Aísla errores.
- Elimina las redundancias.

- **Un programa modular es...**

- Más fácil de escribir
- Más fácil de leer
- Más fácil de modificar



Abstracción de datos

- **Objetivos:**
software reutilizable, fiable y fácil de mantener
- **Abstraer:**
quedarse con lo relevante en un contexto dado
- **Términos relacionados:**
encapsulación, ocultamiento de la información

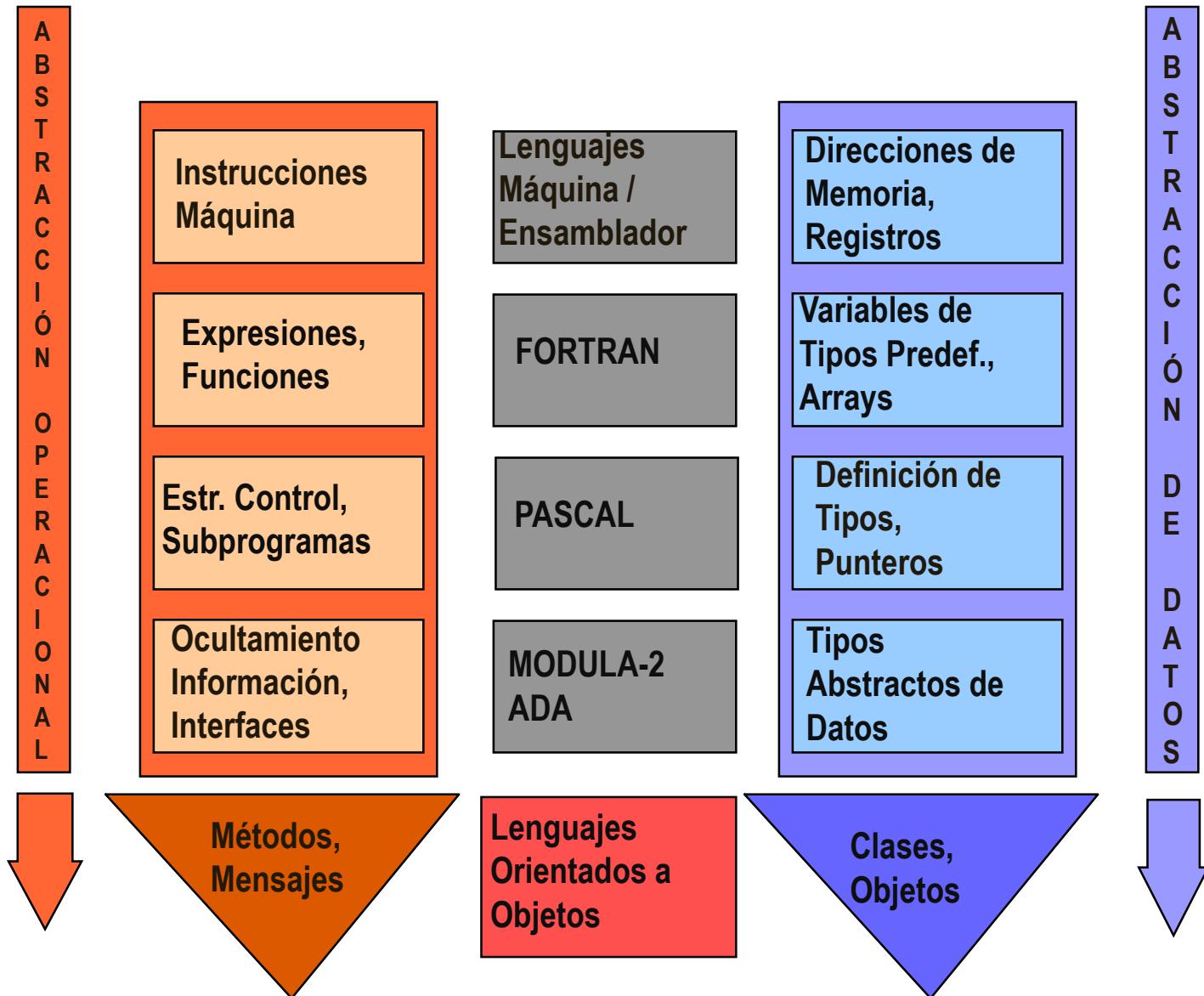
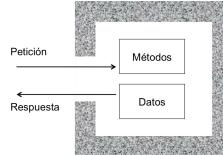
Años 60

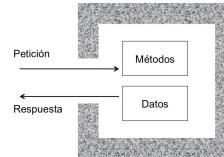
Edsger Dijkstra

Larry Constantine

David Parnas

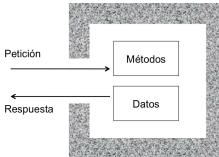
Hacia la abstracción de datos





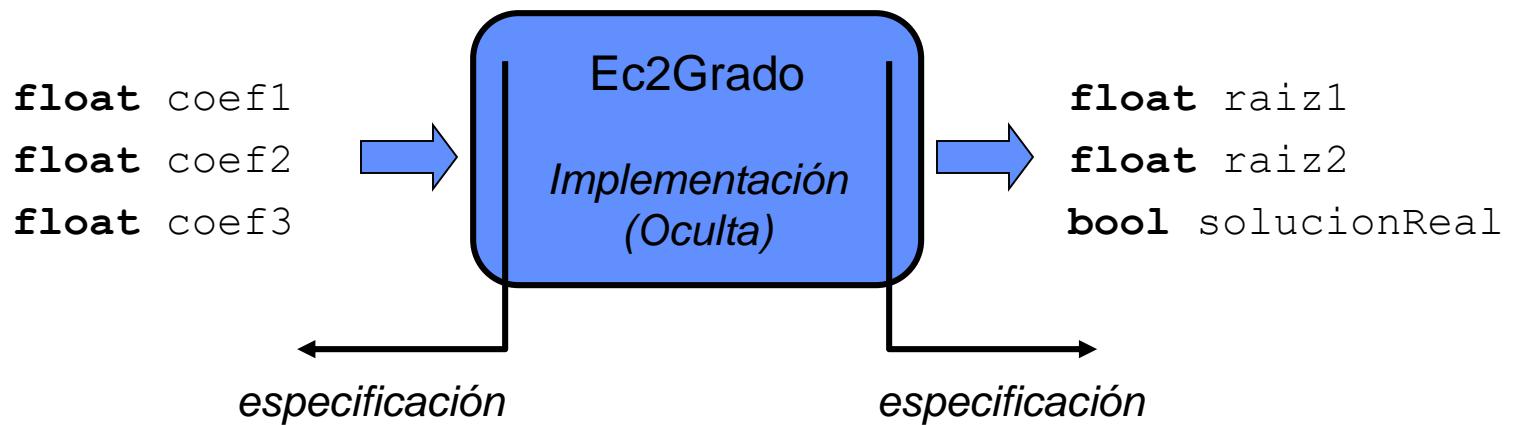
Abstracción funcional

- **Separa el propósito y uso de un módulo de su implementación**
- **La especificación** de un módulo debería...
 - ¤ Dar detalles de cómo se comporta el módulo.
 - ¤ Ser independiente de la implementación del módulo.
- **Ocultamiento de información**
 - ¤ Oculta detalles de implementación.
 - ¤ Hace esos detalles inaccesibles desde fuera.

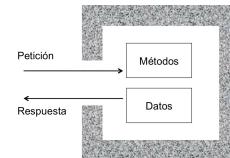


Abstracción funcional

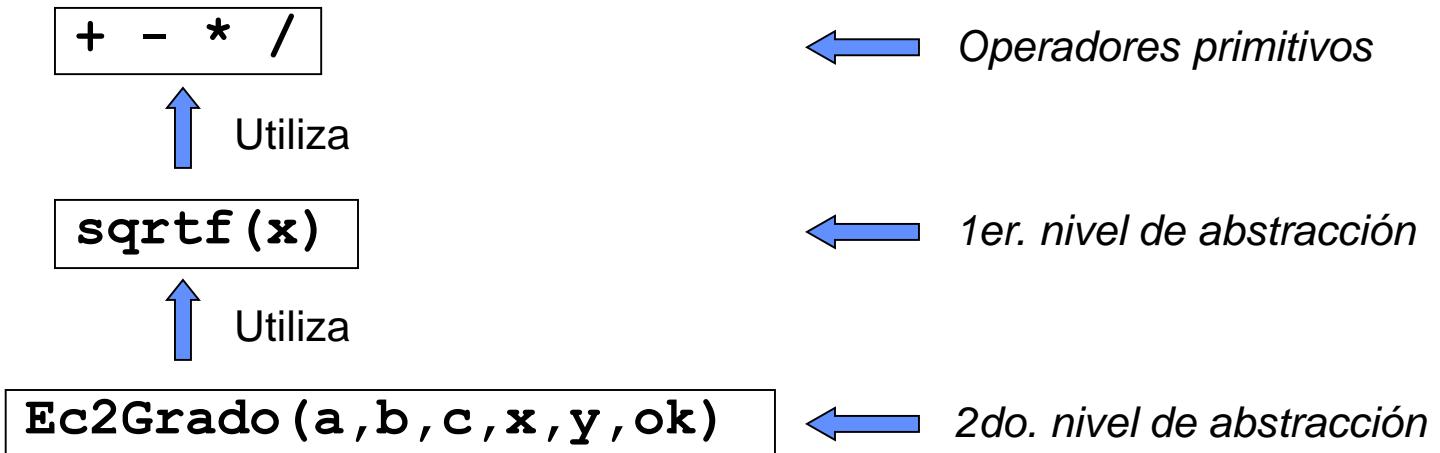
Separación: **especificación (QUÉ) – implementación (CÓMO)**



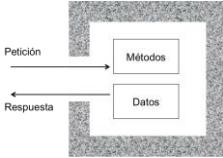
Abstracción funcional



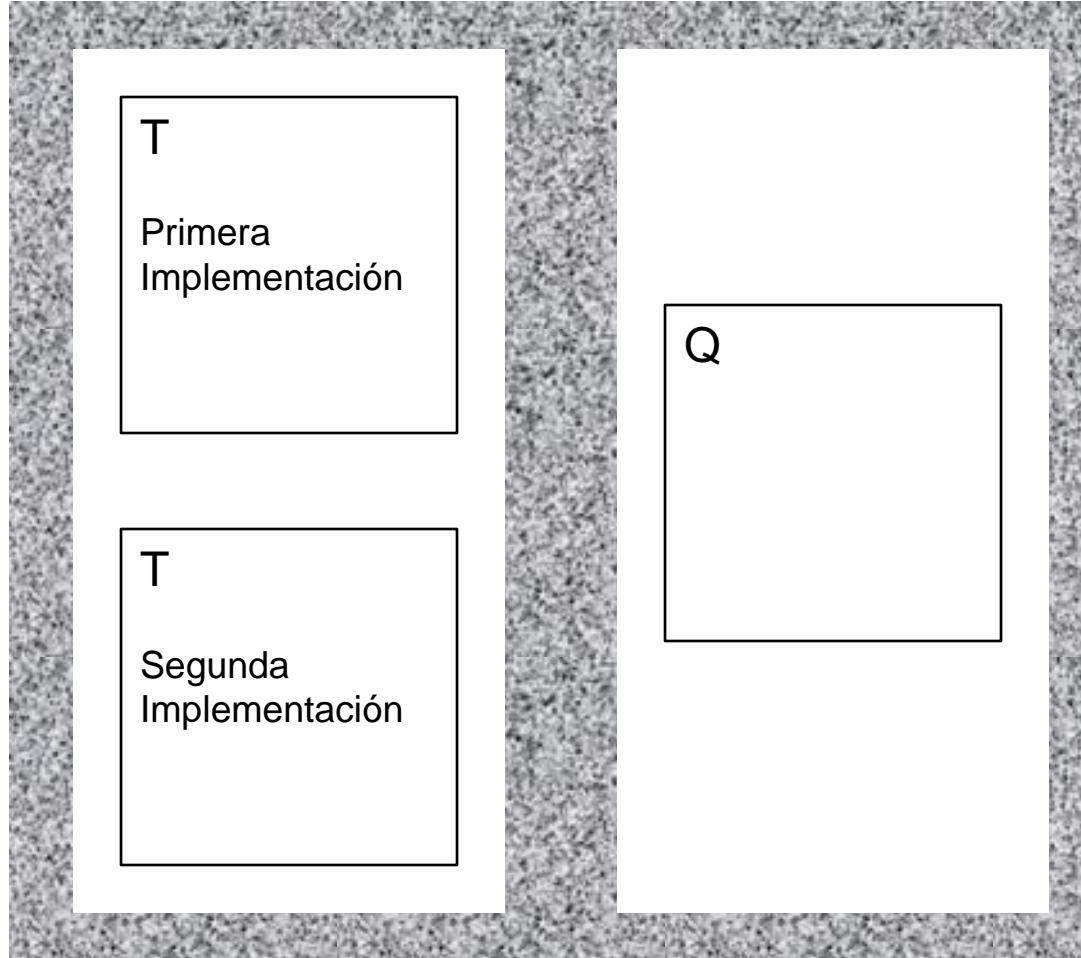
- Ejemplo



- Subprogramas de C y Pascal, bibliotecas

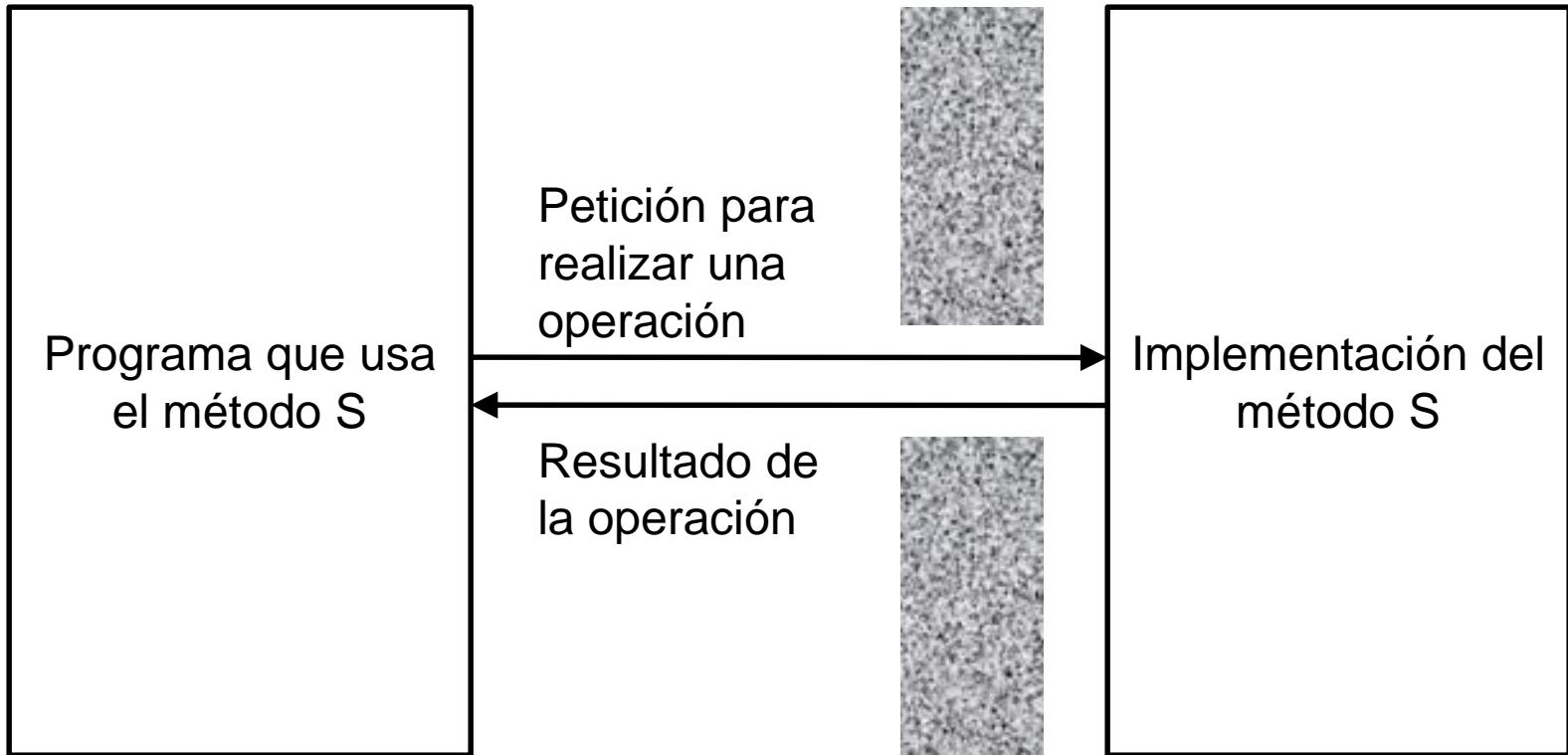
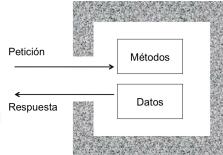


La abstracción funcional aisla

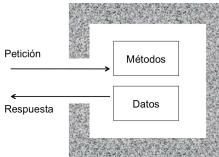


La implementación de T no afecta a la tarea Q

Comunicación a través de interfaz

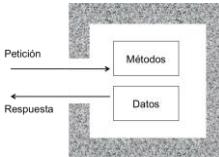


La especificación de la interfaz de una función gobierna la manera en que los demás módulos se comunican con ella.



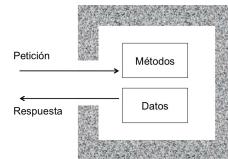
Tipos Abstractos de Datos

- **La abstracción de datos**
 - ¤ Te permite pensar **qué** puedes hacer con una colección de datos independientemente de **cómo** puedes hacerlo.
 - ¤ Te permite desarrollar cada estructura de datos de manera aislada al resto de la solución.
 - ¤ Es una extensión natural de la abstracción funcional.



Tipos Abstractos de Datos

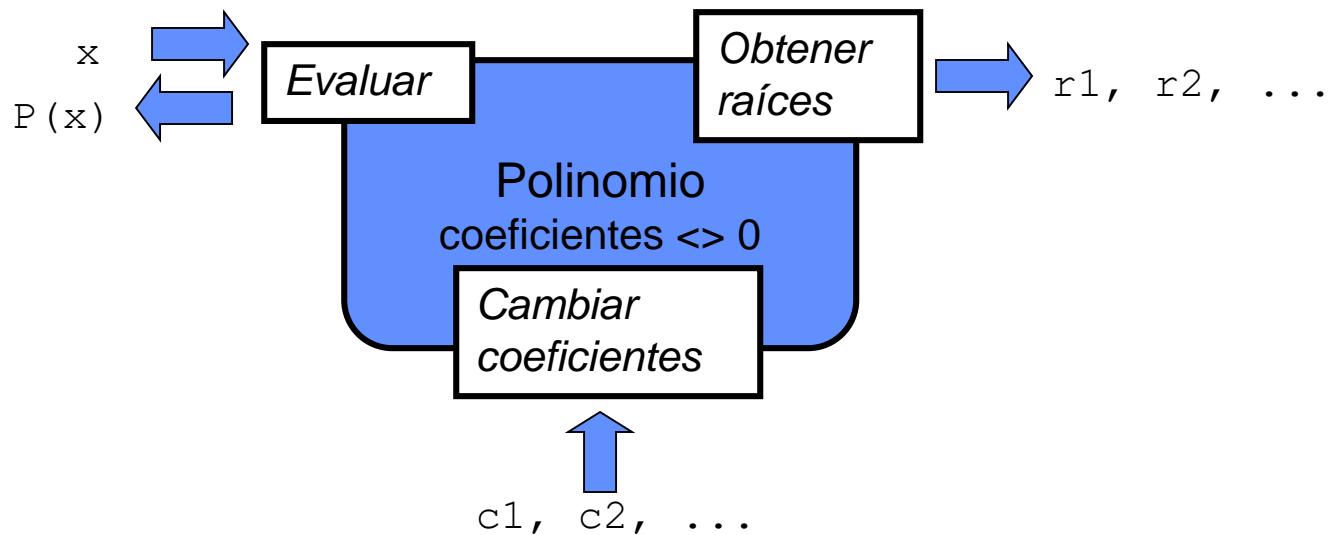
- **Tipos Abstractos de datos (TAD)**
 - ¤ Un TAD está compuesto de
 - Una colección de datos.
 - Un conjunto de operaciones sobre los datos.
 - ¤ Las especificaciones de un TAD indican
 - **Qué** operaciones hace el TAD, no **cómo** se implementan éstas.
 - ¤ La implementación de un TAD
 - Incluye la elección de una estructura de datos en particular.



Abstracción de datos

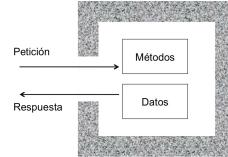
- Abstracción de datos

Encapsulación de datos (ocultos) junto a sus *operaciones* de forma que se puede acceder a los datos sólo a través de sus operaciones:

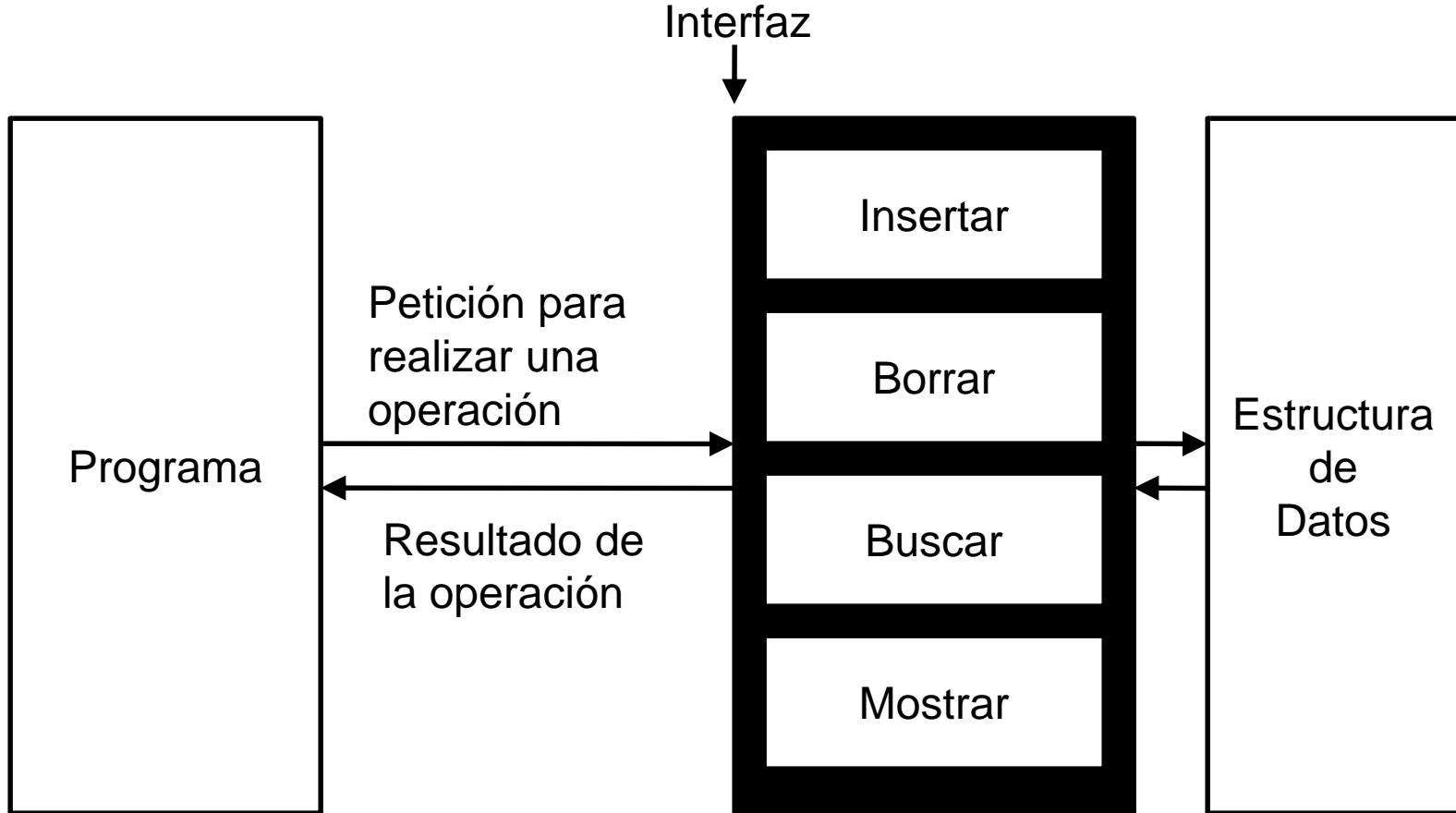


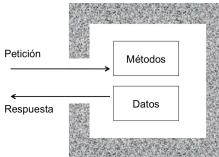
- Paquetes de Ada83, Fortran 90

Tipos Abstractos de Datos



- Un “muro” de operaciones del TAD aísla una estructura de datos del programa que la usa.

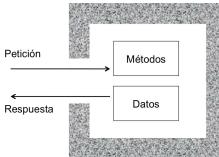




Especificación de los TAD

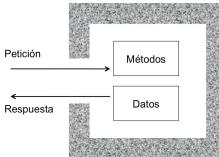
- **Consiste en:**

- **Definir en qué consiste la estructura de datos:**
 - Organización de los datos.
 - Tipo de estructura: datos homogéneos o heterogéneos.
 - Relaciones entre los datos: orden, posición, etc.
 - Restricciones de acceso a los datos.
- **Definir las operaciones:**
 - inserción,
 - eliminación,
 - consulta, etc.



Especificación del TAD Lista

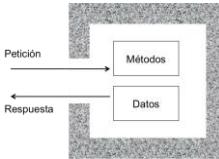
- **Definición:** Una lista es una colección homogénea de elementos, con una relación lineal entre ellos, en la que se puede acceder a los elementos mediante su posición.
- Se puede consultar, insertar y suprimir en cualquier posición de la lista.



Especificación del TAD Lista

Operaciones sobre listas:

- Crear una lista vacía
- ¿Esta vacía?
- ¿Está llena?
- Insertar un elemento en una posición
- Eliminar el elemento de una posición
- Consultar el elemento de una posición
- Longitud
- Destruir la lista



Especificación del TAD Lista

crearLista()

destruirLista()

estaVacia():booleano {**consulta**}

obtenerLongitud():entero {**consulta**}

insertar(**in** indice:entero, **in** nuevoElemento:TpElemLista,
out exito:booleano)

eliminar(**in** indice:entero, **out** exito:booleano)

consultar(**in** indice:entero, **out** elemento:TpElemLista,
out exito:booleano) {**consulta**}

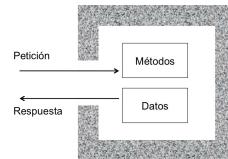
Podemos emplear una notación como UML para especificar las OPERACIONES

Entrada

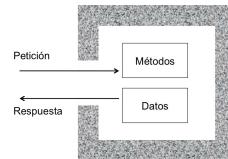
Salida

La operación no modificará los datos, se sólo una consulta

TAD con programación modular



- Podemos implementar un TAD mediante módulos en C++.
- Un módulo permite agrupar (encapsular) definiciones de tipos y operaciones.
- En C++:
 - ¤ Un fichero .hpp con las definiciones de tipos y prototipos de operaciones.
 - ¤ Un fichero .cpp con la implementación de las operaciones.
 - ¤ Los **espacios de nombres** (namespace) permiten agrupar identificadores y evitar colisiones con otros módulos.
 - ¤ La **compilación separada** permite compilar módulos por separado y enlazarlos para formar nuevos programas.



Definición del módulo complejos

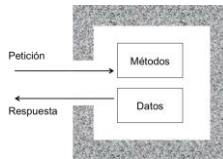
complejos.hpp

```
#ifndef _complejos_      // guarda para evitar inclusión duplicada
#define _complejos_      //

struct Complejo {
    float real;
    float imag;
};

void Crear(Complejo& num, float real, float imag);
void sumar(Complejo& res, const Complejo& x, const Complejo& y);
void mult(Complejo& res, const Complejo& x, const Complejo& y);

#endif
```



Implementac. módulo complejos

```

#include "complejos.hpp"
#include <cmath>
/*
 * Implementación privada del módulo
 */

struct Polar {
    float rho;
    float theta;
};

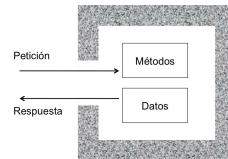
float sq(float x) {
    return x*x;
}

void cartesiana_a_polar(Polar& pol, const Complejo& cmp) {
    pol.rho = sqrt(sq(cmp.real) + sq(cmp.imag));
    pol.theta = atan2(cmp.imag, cmp.real);
}

void polar_a_cartesiana(Complejo& cmp, const Polar& pol) {
    cmp.real = pol.rho * cos(pol.theta);
    cmp.imag = pol.rho * sin(pol.theta);
}

```

complejos.cpp



Implementac. módulo complejos

```
/*
 * Implementación correspondiente a la parte pública del modulo
 */
```

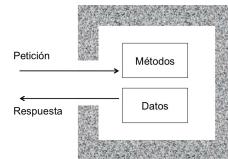
```
void Crear(Complejo& num, float real, float imag) {
    num.real = real;
    num.imag = imag;
}
```

complejos.cpp

```
void sumar(Complejo& res, const Complejo& x, const Complejo& y) {
    res.real = x.real + y.real;
    res.imag = x.imag + y.imag;
}
```

```
void mult(Complejo& res, const Complejo& x, const Complejo& y) {
    Polar pr, p1, p2;

    cartesiana_a_polar(p1, x);
    cartesiana_a_polar(p2, y);
    pr.rho = p1.rho * p2.rho;
    pr.theta = p1.theta + p2.theta;
    polar_a_cartesiana(res, pr);
}
```



Uso del módulo complejos

```

#include <iostream>
#include "complejos.hpp"

using namespace std;

int main() {
    Complejo a, b, c;

    Crear(a, 1.5, 2.3);
    Crear(b, 2.5, 5.5);
    sumar(c, a, b);

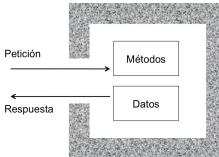
    cout << "Suma: (" << c.real << ", " << c.imag << ")" << endl;
    mult(c, a, b);

    cout << "Multiplicación: (" << c.real << ", " << c.imag << ")" << endl;
}

```

usocomplejos.cpp

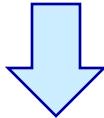
SALIDA POR PANTALLA:
Suma: (4, 7.8)
Multiplicación: (-8.9, 14)



Compilando y enlazando

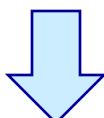
- **Compilando el módulo complejos:**

```
g++ -c -o complejos.o complejos.cpp -Wall -Werror -Wextra -ansi
```



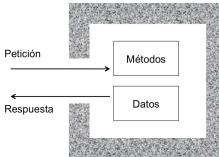
- **Compilando el cliente (programa principal):**

```
g++ -c -o usocomplejos.o usocomplejos.cpp -Wall -Werror -Wextra -ansi
```



- **Enlazando y generando el ejecutable:**

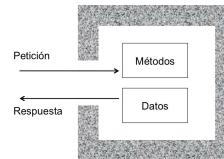
```
g++ -o usocomplejos complejos.o usocomplejos.o -Wall -Werror -Wextra -ansi
```



Espacios de nombres

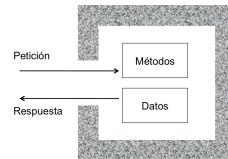
- Es un mecanismo para agrupar de manera lógica declaraciones y definiciones (de identificadores) dentro de una región declarativa común.
- Se evita así la colisión de identificadores entre varios módulos.

```
namespace miEspacioDeNombres
{
    // Declaraciones . . .
} //fin miEspacioDeNombres
```



Espacios de nombres

- El contenido del *namespace* puede ser accedido por el código desde dentro o desde fuera del *namespace*.
 - ¤ Debe usarse el operador de resolución de ámbito (`::`) para acceder a los elementos desde fuera del *namespace*.
 - ¤ Alternativamente, la declaración `using` permite que los identificadores del *namespace* puedan ser usados directamente: se incorporan al espacio de nombres global.



Creando y usando *namespaces*

- Creando un *namespace*

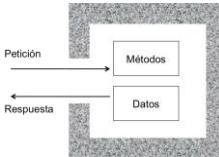
```
namespace miNamespace
{
    int cuenta = 0;
    void abc();
} // fin miNamespace
```

- Usando un *namespace*

```
using namespace miNamespace;
cuenta +=1;
abc();
```

Alternativamente...

```
miNamespace::cuenta +=1;
miNamespace::abc();
```



Namespaces evitan colisiones

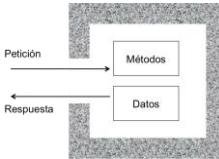
```
// namespaces
#include <iostream>
using namespace std;

namespace primero
{
    int var = 5;
}

namespace segundo
{
    double var = 3.1416;
}

int main ()
{
    cout << primero::var << endl; // Saldrá 5 por pantalla
    cout << segundo::var << endl; // Saldrá 3.1416 por pantalla
}
```

Varios *namespaces* pueden declarar o definir el mismo identificador: la cualificación mediante el operador de ámbito evita la ambigüedad.

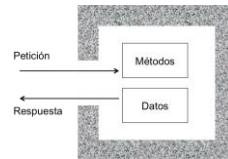


Espacio de nombres estándar

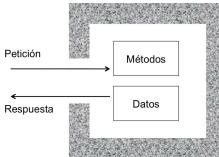
- Los elementos que se declaran en la biblioteca estándar de C++ están declarados en el *namespace std*.
- Cuando incluimos ficheros (headers) de funciones estándar, éstas están declaradas en el *namespace std*.
 - ¤ Por ejemplo, para incluir funciones de entrada salida de la biblioteca de C++, escribimos:

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

Módulo complejos + namespace



```
*****  
* Implementación del fichero MComplejos.hpp con la definición del módulo  
* de números complejos usando espacios de nombres.  
* *****  
  
#ifndef _complejos_      // guarda para evitar inclusion duplicada  
#define _complejos_      //  
  
namespace bblProgII{  
    struct Complejo {  
        float real;  
        float imag;  
    };  
  
    void Crear(Complejo& num, float real, float imag);  
    void sumar(Complejo& res, const Complejo& x, const Complejo& y);  
    void mult(Complejo& res, const Complejo& x, const Complejo& y);  
} // Del namespace MComplejos  
#endif  
//-- fin: MComplejos.hpp -----
```



Módulo complejos + namespace

```

//-- fichero: MComplejos.cpp -----
#include "MComplejos.hpp"
#include <cmath>

/* Implementacion privada del modulo */
namespace {
    using namespace std;
    using namespace bblProgII;

    struct Polar {
        float rho;
        float theta;
    };

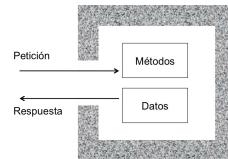
    float sq(float x) {
        return x*x;
    }

    void cartesiana_a_polar(Polar& pol, const Complejo& cmp) {
        pol.rho = sqrt(sq(cmp.real) + sq(cmp.imag));
        pol.theta = atan2(cmp.imag, cmp.real);
    }

    void polar_a_cartesiana(Complejo& cmp, const Polar& pol) {
        cmp.real = pol.rho * cos(pol.theta);
        cmp.imag = pol.rho * sin(pol.theta);
    }
} // Fin del namespace sin nombre: fin de parte privada del módulo

```

Un *namespace* sin nombre permite definir elementos privados al módulo (unidad de compilación) donde se crea: no son accesibles desde el exterior



Módulo complejos + namespace

```

/* Implementacion correspondiente a la parte publica del modulo */
namespace bblProgII{ // Nuevos elemento se añaden al namespace bblProgII
    void Crear(Complejo& num, float real, float imag) {
        num.real = real;
        num.imag = imag;
    }

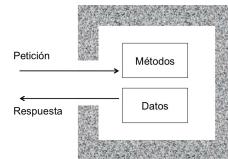
    void sumar(Complejo& res, const Complejo& x, const Complejo& y) {
        res.real = x.real + y.real;
        res.imag = x.imag + y.imag;
    }

    void mult(Complejo& res, const Complejo& x, const Complejo& y) {
        Polar pr, p1, p2;

        cartesiana_a_polar(p1, x);
        cartesiana_a_polar(p2, y);
        pr.rho = p1.rho * p2.rho;
        pr.theta = p1.theta + p2.theta;
        polar_a_cartesiana(res, pr);
    }
} // Fin el namespace MComplejos
// - fin: MComplejos.cpp -----

```

En MComplejos.cpp podemos seguir añadiendo elementos al namespace MComplejos, volviéndolo a abrir: todo lo definido en el namespace MComplejos en MComplejos.hpp es visible dentro del namespace MComplejos.



Módulo complejos + namespace

```

#include <iostream>
#include "MComplejos.hpp"

using namespace std;

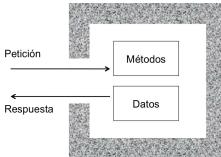
using namespace bblProgII;

int main() {
    Complejo a, b, c;

    Crear(a, 1.5, 2.3);
    Crear(b, 2.5, 5.5);
    sumar(c, a, b);
    cout << "Suma: (" << c.real << ", " << c.imag << ")" << endl;
    mult(c, a, b);
    cout << "Multiplicación: (" << c.real << ", " << c.imag << ")" << endl;
}

```

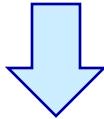
usoMComplejos.cpp



Compilando y enlazando

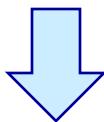
- **Compilando el módulo complejos:**

```
g++ -c -o MComplejos.o MComplejos.cpp -Wall -Werror -Wextra -ansi
```



- **Compilando el cliente (programa principal):**

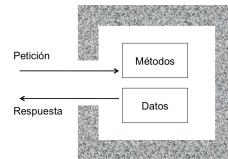
```
g++ -c -o usoMComplejos.o usoMComplejos.cpp -Wall -Werror -Wextra -ansi
```



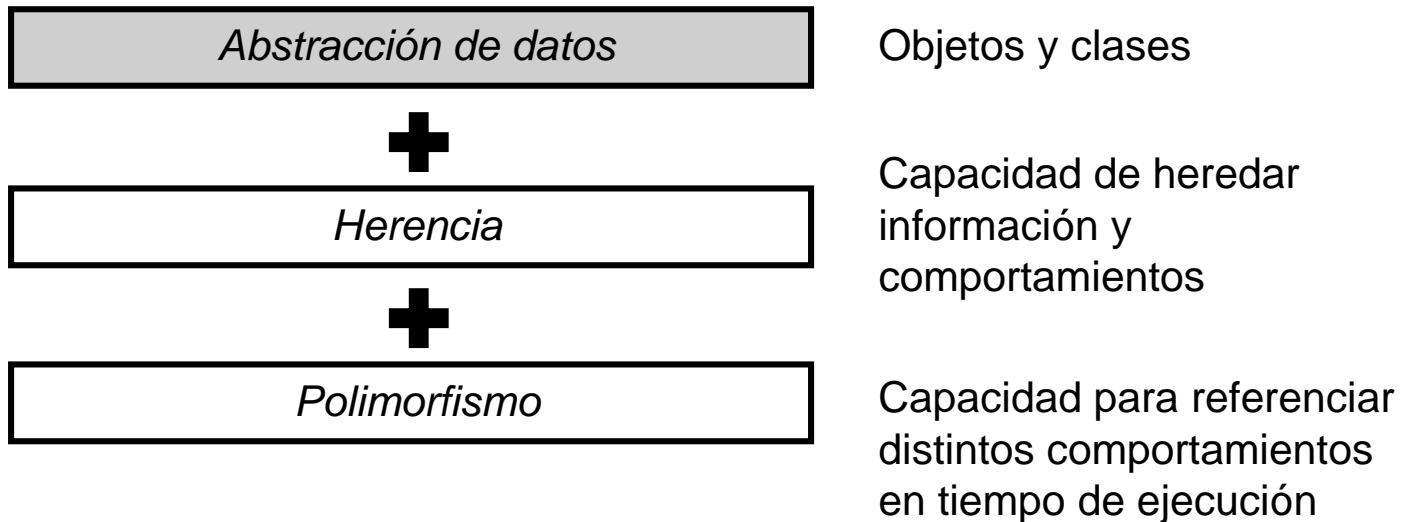
- **Enlazando y generando el ejecutable:**

```
g++ -o usoMComplejos MComplejos.o usoMComplejos.o -Wall -Werror -Wextra -ansi
```

Abstracción de datos y OO

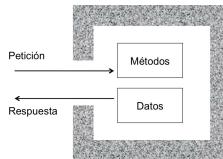


Orientación a objetos:

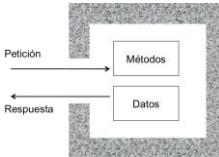


Clases en Smalltalk, Java, C++

Evolución de los lenguajes OO



- ¤ **Simula** (Nygaard, 60s)
- ¤ **Smalltalk** (Xerox PARC, 70s)
- ¤ **Eiffel** (Meyer, 80s)
- ¤ **C++** (Stroustrup, 80s)
- ¤ **Java** (Sun Microsystems, 90s)
- ¤ **C#** (Microsoft, 00s)



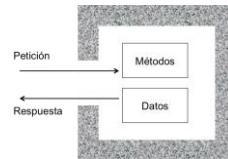
Conceptos básicos de POO

CLASE = SUBPROGRAMAS + VARIABLES

- Criterio de Modularización
- Estado + Comportamiento
- Entidad estática
- Clase \approx Tipo

OBJETO = Instancia de una CLASE

- Entidad dinámica
- Cada objeto tiene su propio estado
- Objetos de una misma clase comparten un comportamiento
- Objeto \approx Variable



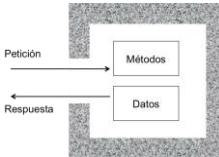
¿Qué es una clase?

Estructura (caja negra) que oculta en su implementación. **Miembros:**

- ¤ **Atributos:** variables que codifican el **estado** de una instancia de la clase (objeto)
- ¤ **Métodos:** subprogramas que describen el **comportamiento** de un objeto de la clase

Una clase es semejante a un tipo:

- ¤ Atributos: estructura de datos
- ¤ Métodos: operaciones sobre el tipo

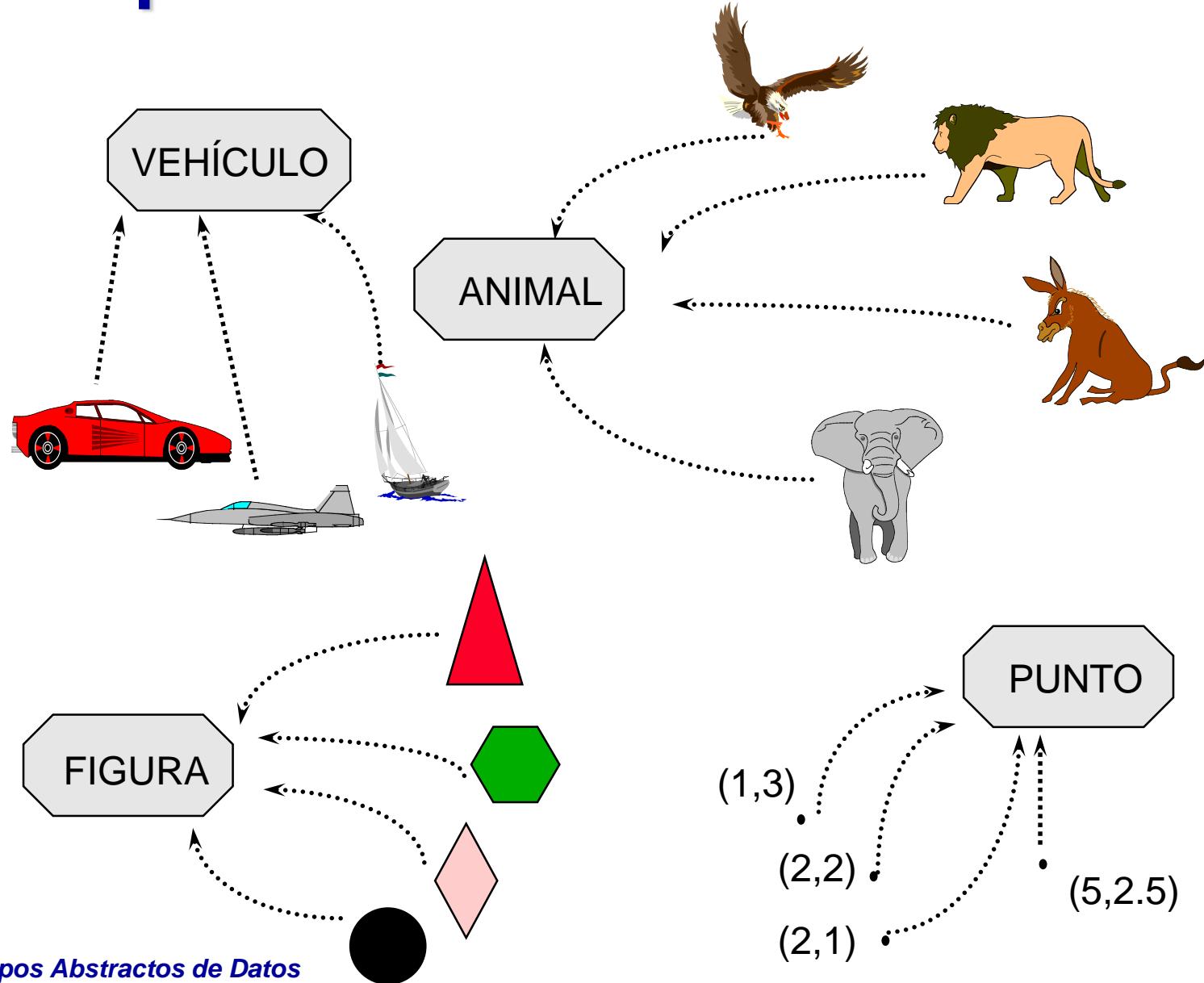
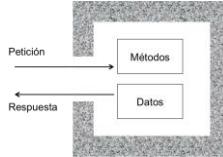


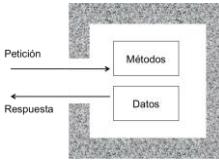
¿Qué es un objeto?

Instancia de una clase:

- ¤ Cada objeto de una clase tiene su propia copia de los atributos (**estado propio**)
- ¤ Todos los objetos de una clase comparten los mismos métodos (**comportamiento común**)

Conceptos básicos de POO

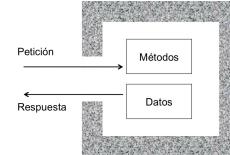




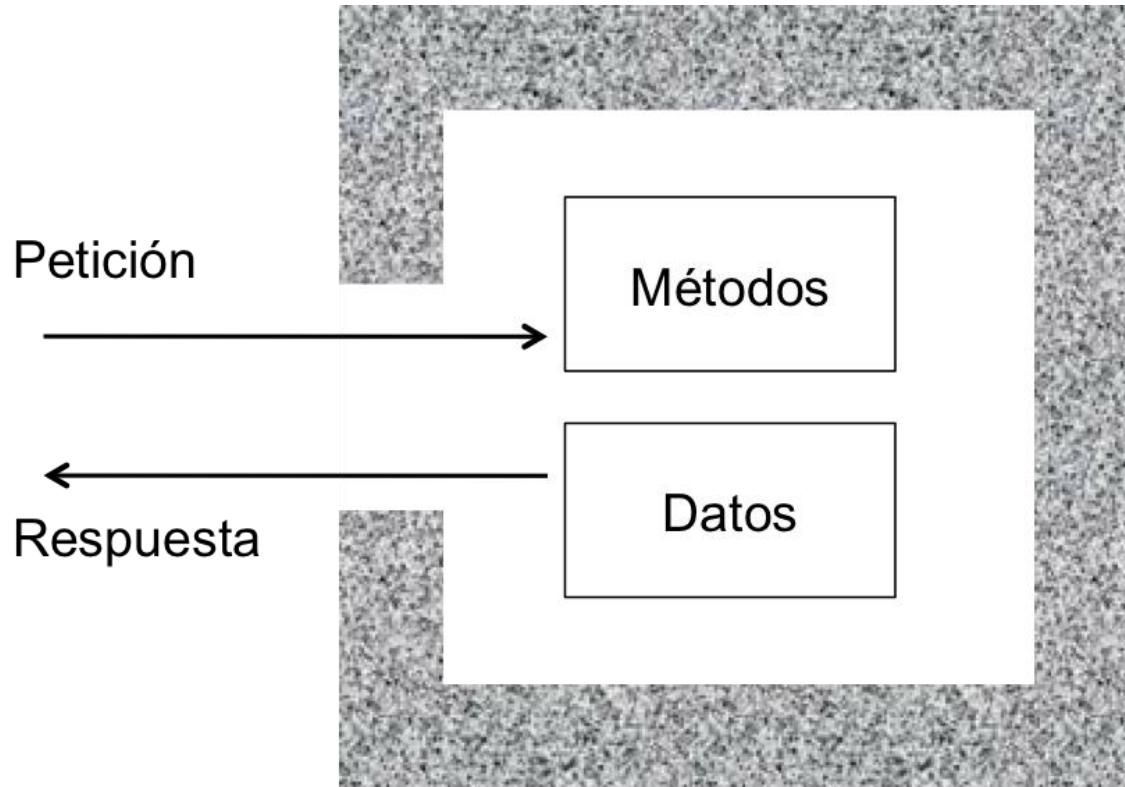
Encapsulación: clases en C++

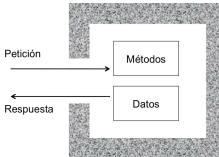
- **La encapsulación combina datos y operaciones en una sola entidad, el **objeto****
 - ¤ La clase especifica la estructura (datos) y el comportamiento de todos los objetos de esa clase.
 - ¤ Un objeto es una instancia de una clase.
 - ¤ Una clase define “un nuevo tipo de datos”.
 - ¤ Una clase contiene **miembros** que pueden ser **atributos** (datos) o **métodos** (operaciones).
 - ¤ Por defecto, todos los miembros de una clase son **privados**.
 - Pero pueden ser especificados como *públicos*.
 - ¤ La encapsulación oculta los detalles de implementación.

Clases en C++



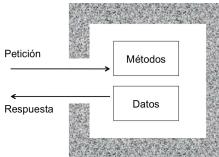
- Un objeto en C++ encapsula datos y operaciones





Clases en C++

- La definición de una clase se realiza en un fichero de cabecera (*header file*):
 - ¤ nombreClase.hpp
 - ¤ nombreClase.h
- La implementación de los métodos de una clase se realiza en un fichero de implementación:
 - ¤ nombreClase.cpp



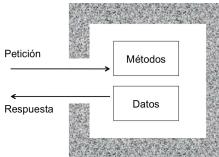
Clases C++: definición (.h/.hpp)

```
/** @fichero CESfera.hpp */
namespace bblProgII{
    class CESfera
    {
        public:
            CESfera(); // Constructor por defecto
            CESfera(double radioInicial); // Constructor
            void establecerRadio(double nuevoRadio);
            double obtenerRadio() const;
            double obtenerDiametro() const;
            double obtenerCircunferencia() const;
            double obtenerArea() const;
            double obtenerVolumen() const;
            void mostrarEstadisticas() const;
        private:
            double radio; // Los atributos deben ser privados
    }; // fin de la clase CESfera
} // fin de bblProgII
```

Parte pública de la clase:
prototipos de métodos que
serán visibles desde fuera de
la clase.

La declaración **const**
evita que el método pueda
modificar los atributos.

Parte privada de la clase:
atributos y prototipos de
métodos auxiliares.



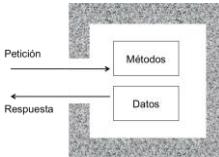
Clases en C++: constructores

- **Los constructores**

- Crean e inicializan nuevas instancias de una clase.
 - Se invocan cuando se declara una instancia (objeto) de una clase.
- Tienen siempre el mismo nombre que la clase.
- No devuelven ningún tipo, ni siquiera `void`

- **Un clase puede tener varios constructores**

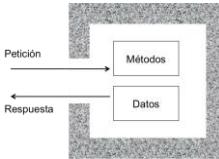
- El constructor por defecto no tiene parámetros.
- El compilador generará un constructor por defecto si no se define ningún constructor en la clase.



Clases en C++: constructores

- En la implementación de un método es necesario cualificar su nombre con el operador de ámbito `::`
- La implementación de un constructor
 - ¤ Inicializa atributos a sus valores iniciales
 - Se puede usar un *inicializador*. Ejemplo:

```
CEsfera::CEsfera() : radio(1.0)
{ // Inicializa radio a 1.0
} // fin del constructor por defecto
```
 - ¤ No puede usarse `return` para devolver un valor



Clases en C++: destructor

- **Destructor**

¤ “Destruye” una instancia de una clase cuando el “tiempo de vida” del objeto ha finalizado:

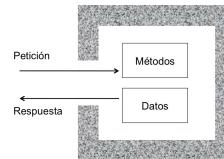
- Por ejemplo, cuando se sale del ámbito (función, bloque, etc.) donde ha sido declarado el objeto.
- Se encarga de liberar los “recursos” que se han asignado explícitamente para el objeto: ficheros, memoria dinámica, etc.

- **Cada clase tiene un único destructor**

¤ Tiene el nombre de la clase, precedido por ~:

- En la definición de la clase: `~CESfera () ; //Destructor`

¤ El destructor puede omitirse, en cuyo caso el compilador genera uno por defecto (vacío).



Implementación de clases en C++

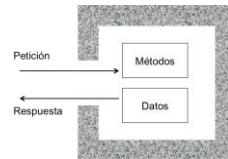
```

/** @fichero CESfera.cpp */
#include <iostream>
#include "CESfera.hpp" // header file
using namespace std;
namespace{ // espacio de nombres sin nombre
    const double PI = 3.14159;
}

namespace bblProgII{
    CESfera::CESfera() : radio(1.0)
    {
    } // fin del constructor por defecto

    CESfera::CESfera(double radioInicial)
    {
        if (radioInicial > 0.0) {
            radio = radioInicial;
        }else{
            radio = 1.0;
        }
    } // fin del constructor
}

```

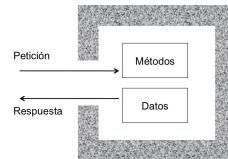


Implementación de clases en C++

```

void CESfera::establecerRadio(double
    nuevoRadio) {
    if (nuevoRadio > 0.0) {
        radio = nuevoRadio;
    } else {
        radio = 1.0;
    }
} // fin de establecerRadio
    
```

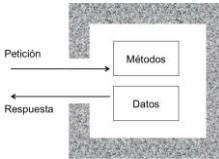
- El constructor podía haber llamado a establecerRadio:
`CESfera::CESfera (double radioInicial)`
`{`
 `establecerRadio(radioInicial);`
`} // fin del constructor`



Implementación de clases en C++

```
double CESfera::obtenerRadio() const
{
    return radio;
} // fin de obtenerRadio
. . .
```

```
double CESfera::obtenerArea() const
{
    return 4.0 * PI * radio * radio;
} // fin de obtenerArea
. . .
```



Usando la clase CESfera

```
#include <iostream>
#include "CESfera.hpp"      // header file
using namespace std;
using namespace bblProgII;
int main()    //cliente de la clase
{
    CESfera esferaUnidad;
    CESfera miEsfera(5.1);
    cout << miEsfera.obtenerDiametro()
        << endl;
    . . .
}
```

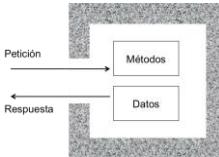
// fin main

Se llama al constructor por defecto

Se llama al constructor extendido

Notación punto (.) para acceso a miembros **públicos** de la clase

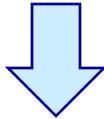
Se llama al destructor para esferaUnidad y miEsfera



Compilando y enlazando

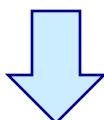
- **Compilando la clase:**

```
g++ -c -o CESfera.o CESfera.cpp -Wall -Werror -Wextra -ansi
```



- **Compilando el cliente (programa principal):**

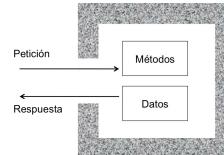
```
g++ -c -o CESferaDemo.o CESferaDemo.cpp -Wall -Werror -Wextra -ansi
```



- **Enlazando y generando el ejecutable:**

```
g++ -o CESferaDemo CESferaDemo.o CESfera.o -Wall -Werror -Wextra -ansi
```

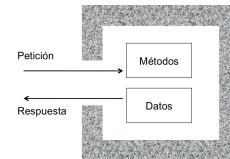
Implementación del TAD Lista



```
#ifndef _CLISTA_HPP_
#define _CLISTA_HPP_
#include <array>
namespace { // Espacio de nombres sin nombre (privado)
    // O en la parte privada de la clase: static const unsigned MAX_LISTA = 100;
    const unsigned MAX_LISTA = 100; // Máximo tamaño de la lista
}

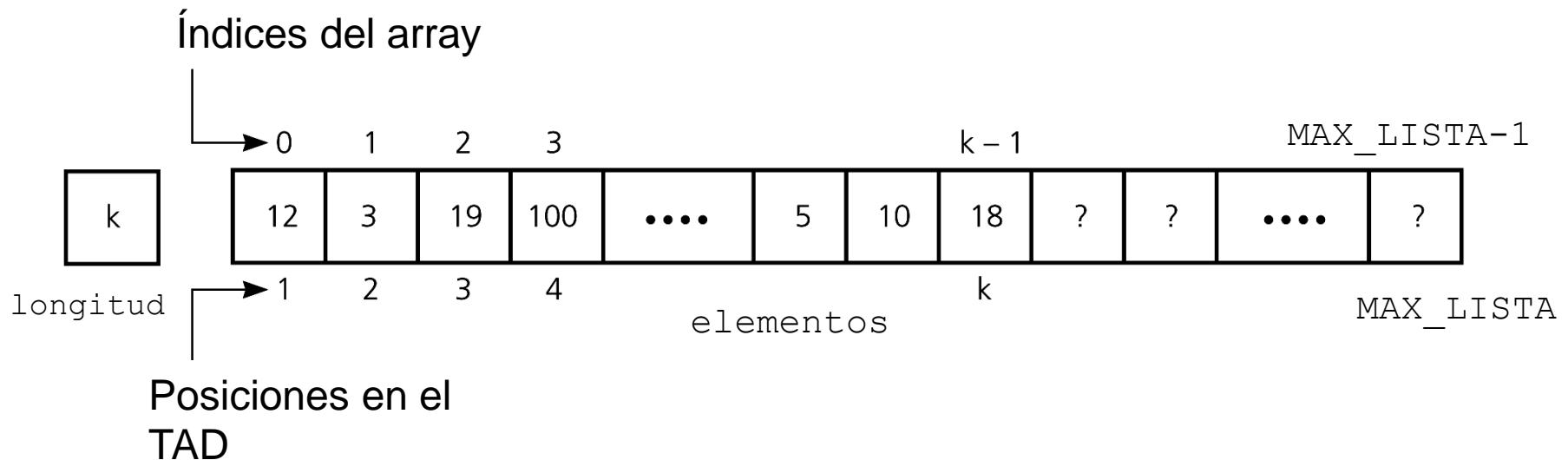
namespace bblProgII{
    typedef int TpElemLista; // Tipo base para la lista
    class CLista{
        public:
            CLista(); // Constructor por defecto
            bool estaVacia() const;
            bool estaLlena() const;
            unsigned obtenerLongitud() const;
            void insertar(unsigned indice, const TpElemLista &nuevoElemento, bool &exito);
            void eliminar(unsigned indice, bool &exito);
            void consultar(unsigned indice, TpElemLista &elemento, bool &exito) const;
        private:
            // Opcional: static const unsigned MAX_LISTA = 100;
            std::array <TpElemLista, MAX_LISTA> elementos; // Atributo: array de elementos
            unsigned longitud; // Atributo: longitud de la lista
            unsigned traducir(unsigned indice) const; // Método privado
            void abrirHueco(unsigned indice); // Método privado
            void cerrarHueco(unsigned indice); // Método privado
    }; // Fin de CLista
} // Fin de bblProgII
#endif
```

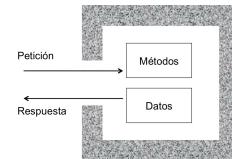
CLista.hpp



Implementación del módulo Lista

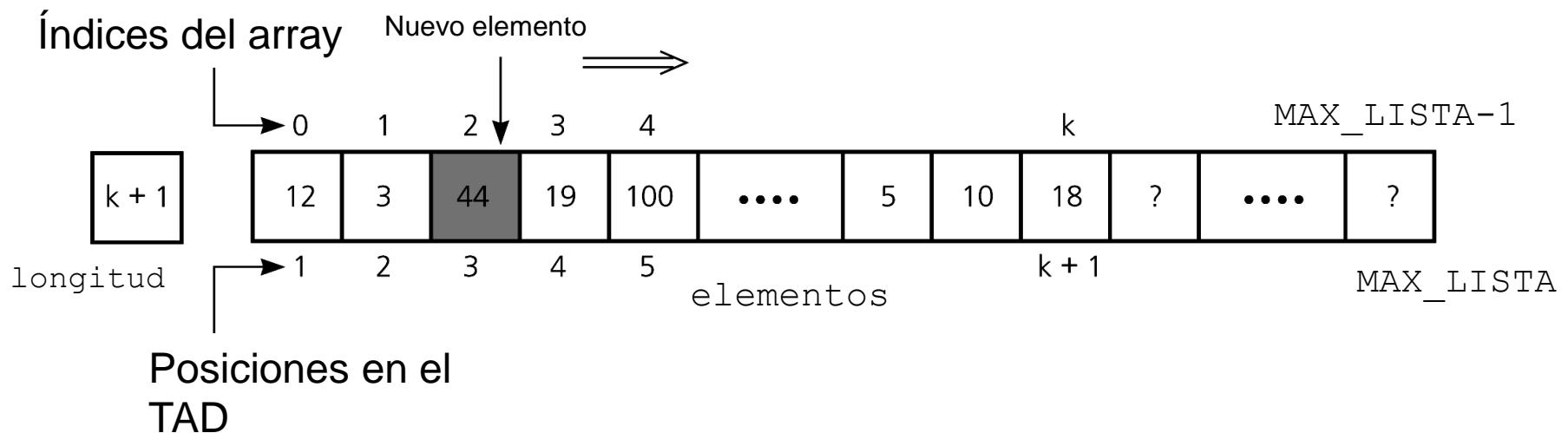
- Una lista basada en arrays
 - ¤ El elemento k -ésimo de la lista es almacenado en $\text{elementos}[k-1]$

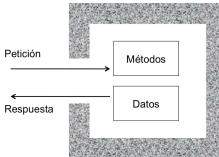




Implementación del módulo lista

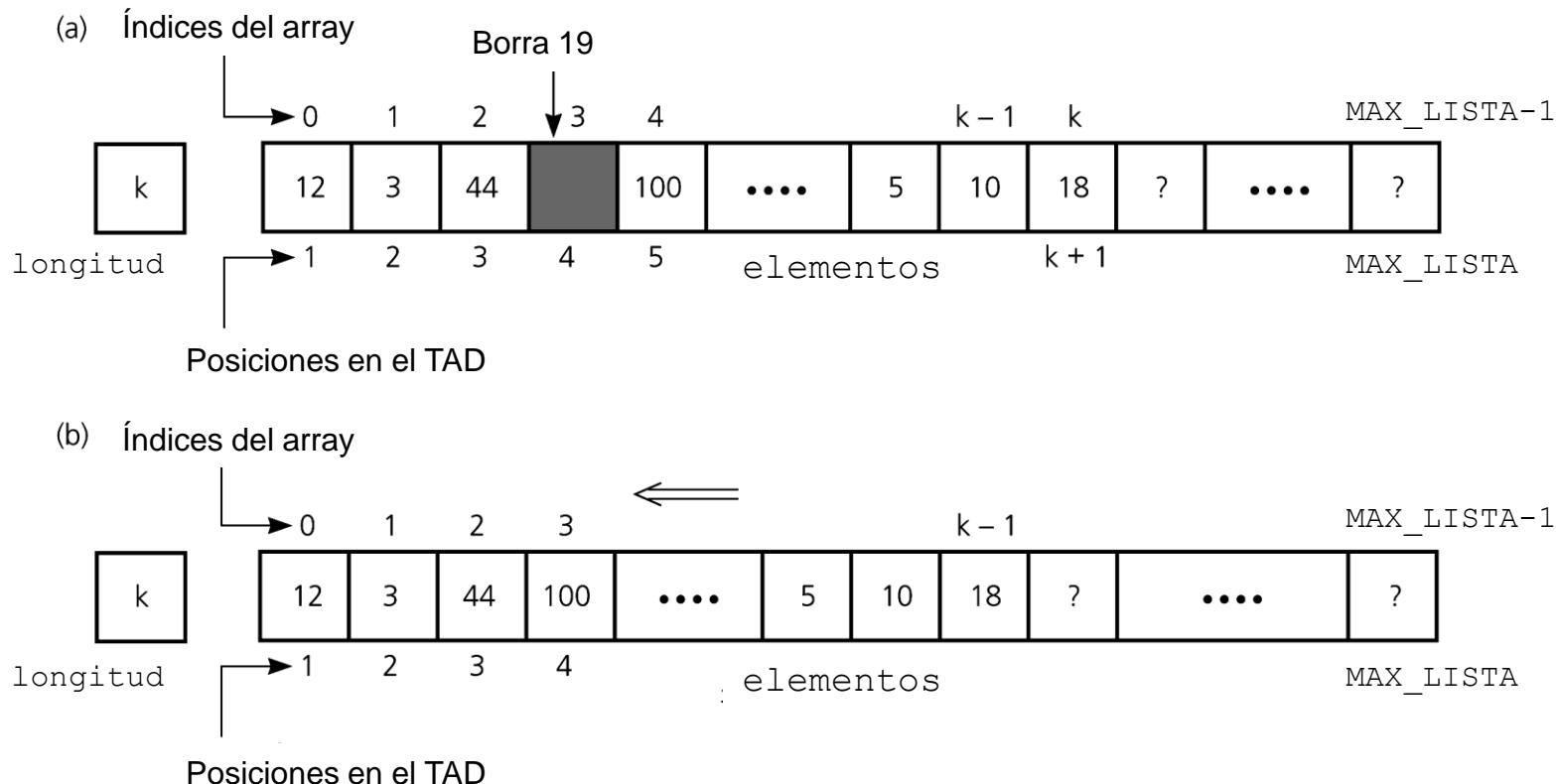
- Para insertar un elemento, hay que abrir un hueco:

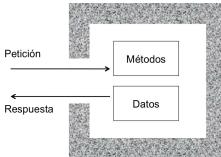




Implementación del módulo lista

- Para eliminar un elemento, hay que cerrar hueco:





Implementación del TAD Lista

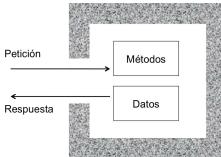
```

#include "CLista.hpp"

// Implementación de la clase CLista
namespace bblProgII{
    CLista::CLista(): elementos(), longitud(0){
    } // Fin del constructor por defecto
    bool CLista::estaVacia() const{
        return longitud == 0;
    } // Fin de estaVacia
    bool CLista::estaLlena() const{
        return longitud == unsigned(elementos.size());
    } // Fin de estaLlena
    unsigned CLista::obtenerLongitud() const{
        return longitud;
    } // Fin de obtenerLongitud()
    void CLista::insertar(unsigned indice, const TpElemLista &nuevoElemento, bool &exito){
        exito = (indice >= 1) &&
            (indice <= longitud+1) &&
            (!estaLlena());
        if (exito) {
            abrirHueco(traducir(indice));
            elementos[traducir(indice)] = nuevoElemento; // Asignación debe estar definida
            ++longitud; // Un elemento más en la lista
        } // Fin if
    } // Fin insertar
}

```

CLista.cpp



Implementación del TAD Lista

```

void CLista::eliminar(unsigned indice, bool &exito){
    exito = (indice >= 1) && (indice <= longitud);
    if (exito){
        cerrarHueco(traducir(indice));
        --longitud;
    } // Fin if
} // Fin eliminar

void CLista::consultar(unsigned indice, TpElemLista &elemento, bool &exito) const{
    exito = (indice >= 1) && (indice <= longitud);
    if (exito){
        elemento = elementos[traducir(indice)];
    }
} // Fin consultar

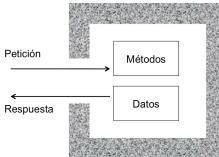
unsigned CLista::traducir(unsigned indice) const{
    return indice - 1;
} // Fin traducir

void CLista::abrirHueco(unsigned indice){
    for (unsigned pos = longitud; pos > indice; --pos){
        elementos[pos] = elementos[pos-1];
    } // Fin for
} // Fin abrirHueco

void CLista::cerrarHueco(unsigned indice){
    for (unsigned pos = indice; pos < longitud - 1; ++pos){
        elementos[pos] = elementos[pos+1];
    } // Fin for
} // Fin cerrarHueco

```

CLista.cpp



Usando el TAD Lista

```

#include <iostream>
#include "CLista.hpp"

using namespace std;
using namespace bblProgII;

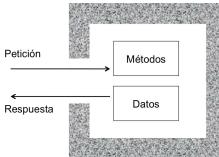
int main(){
    CLista listaEnteros;
    int elem;
    bool exito;

    cout << "Longitud inicial: " << listaEnteros.obtenerLongitud() << endl;
    if (listaEnteros.estaVacia()){
        cout << "La lista está inicialmente vacía." << endl;
    }
    listaEnteros.insertar(1, 10, exito);
    listaEnteros.insertar(1, 5, exito);
    listaEnteros.insertar(1, -5, exito);

    cout << "Nueva longitud de la lista: " << listaEnteros.obtenerLongitud() << endl;
    for (unsigned pos = 1; pos <= listaEnteros.obtenerLongitud(); ++pos){
        listaEnteros.consultar(pos, elem, exito);
        if (exito){
            cout << elem << " ";
        }
    }
    cout << endl;
}

```

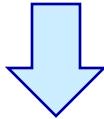
usoCLista.cpp



Compilando y enlazando

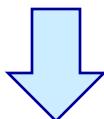
- **Compilando la clase Lista:**

```
g++ -c -o CLista.o CLista.cpp -Wall -Werror -Wextra -ansi
```



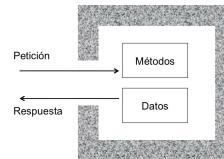
- **Compilando el cliente (programa principal):**

```
g++ -c -o usoCLista.o usoCLista.cpp -Wall -Werror -Wextra -ansi
```



- **Enlazando y generando el ejecutable:**

```
g++ -o usoCLista CLista.o usoCLista.o -Wall -Werror -Wextra -ansi
```



Paso de objetos como parámetro

- Los objetos pueden pasarse como parámetro a procedimientos, funciones y/o métodos:

```
void PintarEsfera(const CESfera &esfera);
```

```
bool Iguales(const CLista &l1, const CLista &l2);
```

- Los objetos pueden pasarse como parámetro por valor o por referencia:

- Se aconseja pasar siempre los objetos como parámetro por referencia (constante o no) para evitar la copia del parámetro real en el parámetro formal.
- Si el objeto se pasa por valor se invoca al constructor de copia para copiar el parámetro real en el parámetro formal:

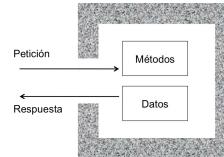
```
void PintarEsfera(const CESfera esfera);
```

...

```
CESfera miEsf(2.5); PintarEsfera(miEsf);
```



Copia de
miEsf en
esfera

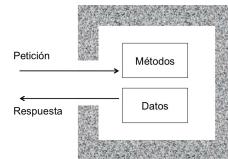


Constructor de copia

- Es un tipo especializado de constructor.
 - Se llama automáticamente cuando un objeto se pasa por valor:
 - se construye una copia local del objeto que se pasa como parámetro.
 - También se llama cuando un objeto se declara e inicializa con otro objeto del mismo tipo:
- ```

CESfera e1(2.5); CESfera e2(e1); CESfera e3=e1
// e1, e2 y e3 son esferas de radio 2.5

```
- Por defecto, el compilador de C++ incluye un constructor de copia que invoca a los constructores de copia de los atributos de la clase.
  - El constructor de copia puede implementarse.



# Implement. constructor de copia

- Tiene el siguiente formato:

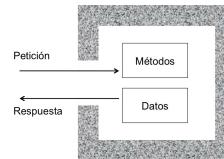
```
tipo_clase::tipo_clase(const tipo_clase &obj);
```

- Ejemplo:

```
class CESfera{ // Declaración: CESfera.hpp
public:
 CESfera (const CESfera &esf);
 ...
// Definición: CESfera.cpp
CESfera::CESfera(const CESfera &esf):radio(esf.radio) {}

...
int main();
CESfera e1(2.5);
CESfera es(e1); // Uso del constructor de copia
```

¡OJO!, por referencia constante.

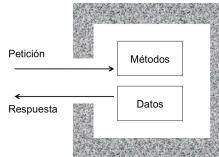


# Objetos devueltos y asignación

- Los objetos de la misma clase pueden asignarse entre sí.
  - ¤ C++ sobrecarga el operador de asignación (=) para objetos de clases definidas por el usuario.

```
CComplejo c1, c2; c1 = c2;
```
- La asignación supone una copia miembro a miembro.
- Un objeto puede ser devuelto por una función o un por un método. Ejemplo:

```
// Función para sumar dos números complejos:
CComplejo Sumar(const CComplejo &c1, const CComplejo &c2);
int main()
{
 CComplejo a, b, c;
 ...
 c = Sumar(a, b); // Sumar devuelve un objeto
```



# Sobrecarga de métodos

- Igual que ocurre con las funciones, pueden definirse varios métodos con el mismo identificador que se diferencien en el número y/o tipo de los parámetros.
- Ejemplo:

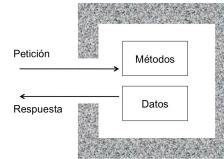
```

class CComplejo{ // Numeros complejos
public:
 // 1) Suma un número complejo al número complejo actual
 void Sumar(const CComplejo &otro);

 // 2) Suma dos números complejos y devuelve el resultado
 void Sumar(const CComplejo &otro, CComplejo &resultado) const;
 ...

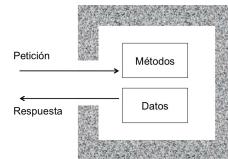
int main() {
 CComplejo a, b, c;
 ...
 a.Sumar(b); // Llamada a 1): a es el resultado de sumar a + b
 a.Sumar(b, c); // Llamada a 2): c es el resultado de sumar a + b
}

```



# Sobrecarga de operadores

- Hace posible manipular objetos de clases con operadores estándar: +, -, \*, [], <<, etc.
- Una función operador:
  - consta de la palabra reservada `operator...`
  - ... seguida por un operador unitario o binario:  
`operatoroperador`
- Se definen con el nombre `operatorX`, donde X es el símbolo del operador. Ejemplos: `operator+`, `operator++`, `operator<<`, `operator=`, etc.
- Las funciones operador (excepto `new`, `delete` y `->`) pueden devolver cualquier tipo.
- El número de operandos y la prioridad es la misma que para el operador predefinido.



# Sobrecarga de operadores

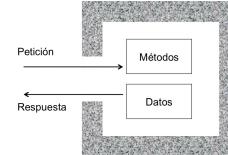
Se puede definir el comportamiento de los siguientes operadores:

|                                   |   |                    |
|-----------------------------------|---|--------------------|
| ( ) [ ] -> ->*                    | B | asoc. izq. a dcha. |
| ++ -- tipo()                      | U | asoc. dcha. a izq. |
| ! ~ + - * &                       | U | asoc. dcha. a izq. |
| * / %                             | B | asoc. izq. a dcha. |
| + -                               | B | asoc. izq. a dcha. |
| << >>                             | B | asoc. izq. a dcha. |
| < <= > >=                         | B | asoc. izq. a dcha. |
| == !=                             | B | asoc. izq. a dcha. |
| &                                 | B | asoc. izq. a dcha. |
| ^                                 | B | asoc. izq. a dcha. |
|                                   | B | asoc. izq. a dcha. |
| &&                                | B | asoc. izq. a dcha. |
|                                   | B | asoc. izq. a dcha. |
| = += -= *= /= %= &= ^=  = <<= >>= | B | asoc. dcha. a izq. |
| ,                                 | B | asoc. izq. a dcha. |
| new new[] delete delete[]         | U | asoc. izq. a dcha. |

(U: unitario; B: binario)

Los siguientes operadores no podrán ser definidos:

:: . . \* ?: sizeof typeid



# Ejemplo de sobrecarga de operadores

## MATERIAL OPCIONAL

```

#ifndef __CLASS__PUNTO2D__
#define __CLASS__PUNTO2D__

namespace bblProgII{

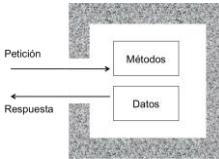
 class CPunto2D{
 public:
 CPunto2D();
 CPunto2D(double coorX, double coorY);
 void mover(double nuevaX, double nuevaY);
 void asignarX(double nuevaX);
 void asignarY(double nuevaY);
 double obtenerX() const;
 double obtenerY() const;
 double Distancia(const CPunto2D &otro) const;
 CPunto2D operator+ (const CPunto2D &otro) const;
 CPunto2D operator- () const;
 CPunto2D operator- (const CPunto2D &otro) const;

 private:
 double x, y;
 }; // De CPunto2D
} // de bblProgII
#endif

```

Clase para manejo de puntos en un espacio bidimensional

Miembros operadores: suma, menos unitario y resta (menos binario).  
 Devuelven objetos **CPunto2D**, sin modificar el objeto actual.



# Implementación de CPunto2D

```
#include <cmath>
#include "CPunto2D.hpp"

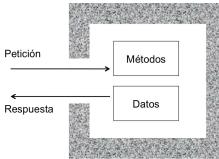
namespace bblProgII{
 // Constructor por defecto
 CPunto2D::CPunto2D(): x(0.0), y(0.0) {}

 // Constructor extendido
 CPunto2D::CPunto2D(double coorX, double coorY): x(coorX), y(coorY) {}

 // Mover punto
 void CPunto2D::mover(double nuevaX, double nuevaY) {
 x = nuevaX; y = nuevaY;
 }

 // Cambiar la coordenada X
 void CPunto2D::asignarX(double nuevaX) {
 x = nuevaX;
 }

 // Cambiar la coordenada Y
 void CPunto2D::asignarY(double nuevaY) {
 y = nuevaY;
 }
}
```



# Implementación de CPunto2D

```

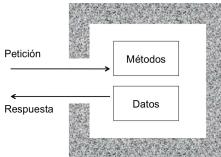
// Devolver la X
double CPunto2D::obtenerX() const{
 return x;
}

// Devolver la Y
double CPunto2D::obtenerY() const{
 return y;
}

// Operador de suma de puntos
CPunto2D CPunto2D::operator+ (const CPunto2D &otro) const{
 CPunto2D res(x + otro.x, y + otro.y);
 return res;
}

// Operador menos unitario
CPunto2D CPunto2D::operator- () const{
 CPunto2D res(-x, -y);
 return res;
}

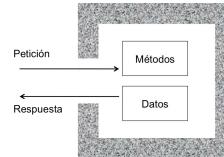
```



# Implementación de CPunto2D

```
// Operador de resta de puntos
CPunto2D CPunto2D::operator- (const CPunto2D &otro) const{
 CPunto2D res(x - otro.x, y - otro.y);
 return res;
}

// Cálculo de la distancia euclídea
double CPunto2D::Distancia(const CPunto2D &otro) const{
 return (sqrt(pow(x - otro.x, 2.0) + pow(y - otro.y, 2.0)));
}
} // de bblProgII
```



# Uso de CPunto2D

```

#include "CPunto2D.hpp"
#include <iostream>

using namespace std;
using namespace bblProgII;

int main(){
 CPunto2D a(1.0, 2.0), b(-1.0, 4.0), c;

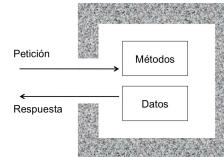
c = a + b; // Llamada al operador de suma
// Puede hacerse, de manera equivalente, como c = a.operator+(b)
cout << "c = a + b = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

c = a - b; // Llamada al operador de resta
// Puede hacerse, de manera equivalente, como c = a.operator-(b)
cout << "c = a - b = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

c = -b; // Llamada al operador de menos unitario
// Puede hacerse, de manera equivalente, como c = b.operator-()
cout << "c = -b = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;
}

```

**usoCPunto2D.cpp**

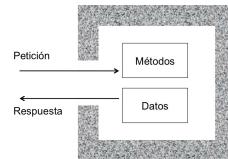


# Sobrecarga de la asignación

- Por defecto, C++ sobrecarga el operador de asignación para objetos de clases definidas por el usuario: copia miembro a miembro.
- Formato general del operador de asignación:

```
// Devolvemos un objeto de la clase para poder hacer asignaciones
// en cadena: a = b = c (siendo a, b y c de la clase nombreClase)
// La referencia evitará una llamada al constructor de copia.
nombreClase & nombreClase::operator=(const nombreClase &o)
{
 // Para evitar autoasignaciones:
 if (this != &o) {
 // semántica de la copia de objetos de esta clase
 }
 return *this; // Devolvemos el objeto actual. Esto permitirá
 // asignaciones en cadena:
 // nombreClase a, b, c; ... a = b = c
}
```

Objeto actual: `this` apunta al objeto actual



# Asignando objetos de CPunto2D

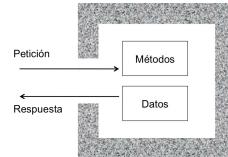
```
// OPERADOR DE ASIGNACIÓN DE PUNTOS
CPunto2D & operator=(const CPunto2D &otro);
```

CPunto2D.hpp

```
// OPERADOR DE ASIGNACIÓN DE PUNTOS
CPunto2D & CPunto2D::operator=(const CPunto2D &otro) {
 if (this != &otro) {
 x = otro.x; // Copiamos miembro a miembro
 y = otro.y;
 }
 return *this;
}
```

CPunto2D.cpp

# Bibliografía



- **F.M. Carrano. Data Abstraction and Problem Solving with C++.**  
**Walls and Mirrors.** Addison Wesley, 4th edition, 2005.
- **H. M. Deitel, P. J. Deitel** Cómo programar en C/C++. Prentice-Hall, 1994.
- **Joyanes Aguilar, L.** Programación en C++. Algoritmos, estructuras de datos y objetos. Ed. McGrawHill, 2000.
- **Savitch, W.** Resolución de problemas con C++. Segunda Edición. , Prentice Hall 2000
- **Stroustup, B** El lenguaje de programación C++. Edición Especial, Adisson-Wesley, 2001
- **C. G. Rodríguez, L.F. Llana, R. Martínez, P. Palao, C. Pareja** Ejercicios de Programación creativos y recreativos en C++. Prentice-Hall, 2002.

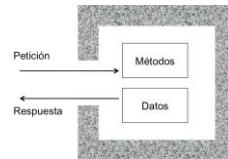


UNIVERSIDAD  
DE MÁLAGA

## Anexo 1

# Genericidad mediante plantillas

*Tema 2. Tipos Abstractos de Datos  
marzo de 2019*



# Genericidad mediante plantillas

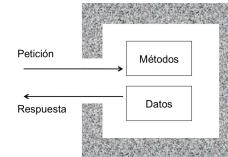
- En la implementación del TAD lista mediante clases, se ha usado una definición de tipo para el tipo base de la estructura de datos:

```
typedef int TpElemLista; // Para una lista de enteros
```

- Para cambiar el tipo base hay que modificar la definición y ¡recompilar!

```
typedef char TpElemLista; // Para una lista de caracteres
```

- Las plantillas (*templates*) de C++ permiten definir tipos abstractos que pueden ser instanciados en el momento de su uso.



# Plantillas de funciones

- Una plantilla de función especifica un conjunto infinito de funciones y describe propiedades genéricas de la función.
- Las plantillas se declaran normalmente en un archivo de cabecera.
- **Sintaxis:**

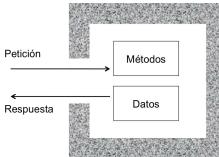
`template <declaraciones-parametros plantilla>`  
declaración o definición de la función

- Cada parámetro formal consta de la palabra reservada `typename` seguida por un identificador. Ejemplo:

```

template <typename T1, typename T2>
void miFuncion(T1 a, T2 &b, T1 &c) {
 // Cuerpo de la función
}

```

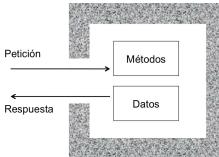


# Ejemplo de plantilla de función

```
#ifndef __MINIMO_HPP__
#define __MINIMO_HPP__
namespace bblProgII{
 // Calcula el mínimo de dos valores
 // Precondiciones:
 // - T debe permitir el operador <=
 // - T puede ser devuelto con return
 // Postcondiciones: Devuelve el mínimo
 template <typename T>
 T minimo(T a, T b) {
 T res;

 if (a <= b){ // El operador <= debe estar definido
 res = a;
 }else{
 res = b;
 }
 return res;
 }
}
#endif
```

minimo.hpp



# Ejemplo de plantilla de función

```
#include <iostream>
#include "minimo.hpp"

using namespace std;
using namespace bblProgII;

int main() {
 int ea = 1, eb = 5;
 cout << "(int): " << minimo<int>(ea, eb) << endl;

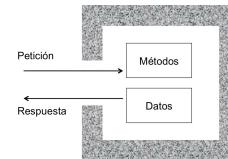
 long la = 1, lb = 5;
 cout << "(long): " << minimo<long>(la, lb) << endl;

 char ca = 'a', cb = 'x';
 cout << "(char): " << minimo<char>(ca, cb) << endl;

 double da = 423.654 , db = 789.12;
 cout << "(double): " << minimo<double>(da, db) << endl;
}
```

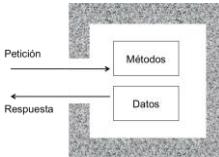
Si no se especifica el tipo, el compilador puede deducirlo a partir de los parámetros

usoMinimo.cpp



# Plantilla del TAD Lista

- Queremos dotar al TAD Lista de genericidad en el tipo base.
- Para ello, tenemos que escribir un único fichero .hpp que incluya tanto la declaración como la implementación de las operaciones del TAD Lista.
- El fichero .hpp no hay que compilarlo: se incluye (include) en el fuente donde va a ser usado.



# Plantilla del TAD Lista

```

// ****
// Fichero de cabecera TLista.hpp pra la definición del TAD lista, con
// implementación basada en arrays
// ****
// Para evitar múltiples inclusiones de ficheros de definición

#ifndef _TLISTA_HPP_
#define _TLISTA_HPP_
#include <array>

namespace {
 const unsigned MAX_LISTA = 100; // Máximo tamaño de la lista
}

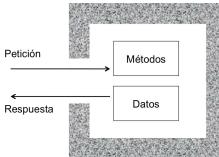
namespace bblProgII{
 template <typename TpElemLista>
 class CLista{
 public: La parte pública de la clase
 CLista(): longitud(0){
 } // Fin del constructor por defecto

 bool estaVacia() const{
 return longitud == 0;
 } // Fin de estaVacia
 };
}

```

Se indica el tipo abstracto  
antes de la palabra  
reservada `class`

**Implementamos los  
métodos en el .hpp**



# Plantilla del TAD Lista

```

unsigned obtenerLongitud() const{
 return longitud;
} // Fin de obtenerLongitud()

void insertar(unsigned indice, const TpElemLista &nuevoElemento, bool &exito) {
 exito = (indice >= 1) && (indice <= longitud+1) && (longitud < MAX_LISTA);

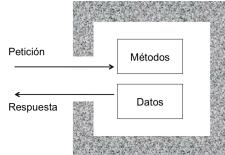
 if (exito) {
 abrirHueco(traducir(indice));
 elementos[traducir(indice)] = nuevoElemento; // Asignación debe estar definida
 ++longitud; // Un elemento más en la lista
 } // Fin if
} // Fin insertar

void eliminar(unsigned indice, bool &exito){
 exito = (indice >= 1) && (indice <= longitud);
 if (exito) {
 cerrarHueco(traducir(indice));
 --longitud;
 } // Fin if
} // Fin eliminar

```

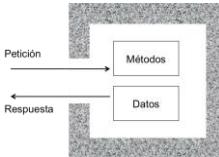
Continuación de la parte pública de la clase.

# Plantilla del TAD Lista



Continuación de la parte pública de la clase.

```
void consultar(unsigned indice, TpElemLista &elemento, bool &exito) const{
 exito = (indice >= 1) && (indice <= longitud);
 if (exito){
 elemento = elementos[traducir(indice)];
 }
} // Fin consultar
```



# Plantilla del TAD Lista

## La parte privada de la clase

**private:**

```

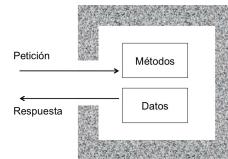
std::array <TpElemLista, MAX_LISTA> elementos; // Atributo: array de elementos
unsigned longitud; // Atributo: longitud de la lista

unsigned traducir(unsigned indice) const{
 return indice - 1;
} // Fin traducir

void abrirHueco(unsigned indice){
 for (unsigned pos = longitud; pos > indice; --pos) {
 elementos[pos] = elementos[pos-1];
 } // Fin for
} // Fin abrirHueco

void cerrarHueco(unsigned indice){
 for (unsigned pos = indice; pos < longitud - 1; ++pos) {
 elementos[pos] = elementos[pos+1];
 } // Fin for
} // Fin cerrarHueco
}; // Fin de CLista
} // Fin de bblProgII
#endif

```



# Usando el TAD Lista (plantilla)

```

#include <iostream>
#include "TLista.hpp"
using namespace std;
using namespace bblProgII;

int main(){
 CLista<int> listaEnteros; // Declaramos una lista de enteros
 int elem;
 bool exito;

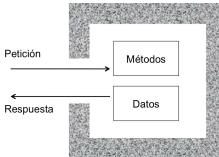
 cout << "Longitud inicial: " << listaEnteros.obtenerLongitud() << endl;
 if (listaEnteros.estaVacia()) {cout << "La lista está inicialmente vacía." << endl; }

 listaEnteros.insertar(1, 10, exito);
 listaEnteros.insertar(1, 5, exito);
 listaEnteros.insertar(1, -5, exito);

 cout << "Nueva longitud de la lista: " << listaEnteros.obtenerLongitud() << endl;
 for (unsigned int pos = 1; pos <= listaEnteros.obtenerLongitud(); ++pos) {
 listaEnteros.consultar(pos, elem, exito);
 if (exito) {cout << elem << " ";}
 }
 cout << endl;
}

```

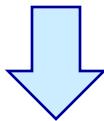
usoTLista.cpp



# Compilando y enlazando

- **Compilando el cliente (programa principal):**

```
g++ -c -o usoTLista.o usoTLista.cpp -Wall -Werror -Wextra -ansi
```



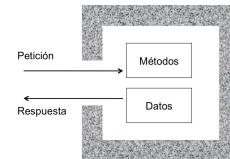
- **Enlazando y generando el ejecutable:**

```
g++ -o usoTLista usoTLista.o -Wall -Werror -Wextra -ansi
```



## Anexo 2

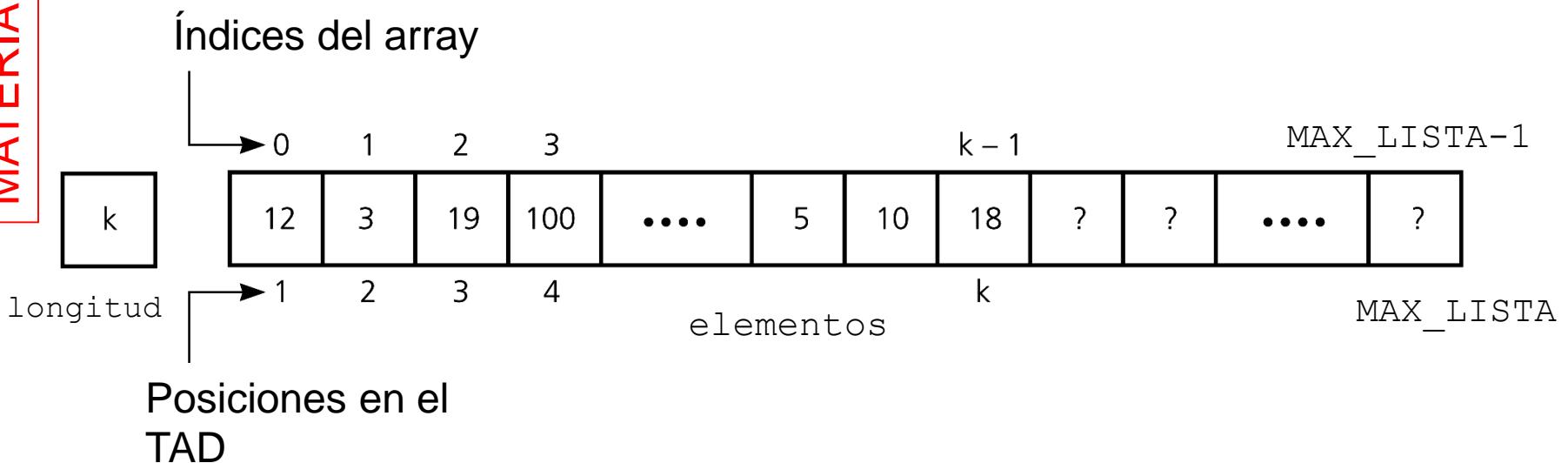
# Implementación del módulo *lista* con módulos y espacios de nombres

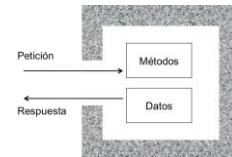


# Implementación del módulo Lista

- Una lista basada en arrays
  - ¤ El elemento  $k$ -ésimo de la lista es almacenado en  $\text{elementos}[k-1]$

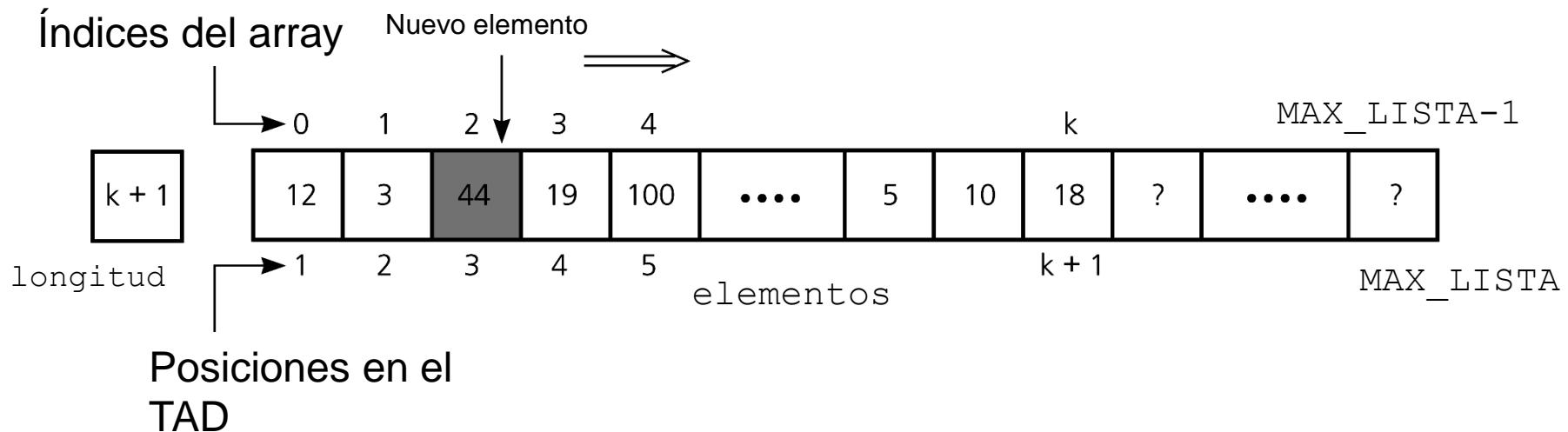
MATERIAL OPCIONAL



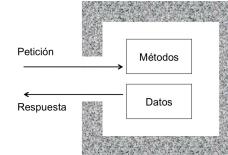


# Implementación del módulo lista

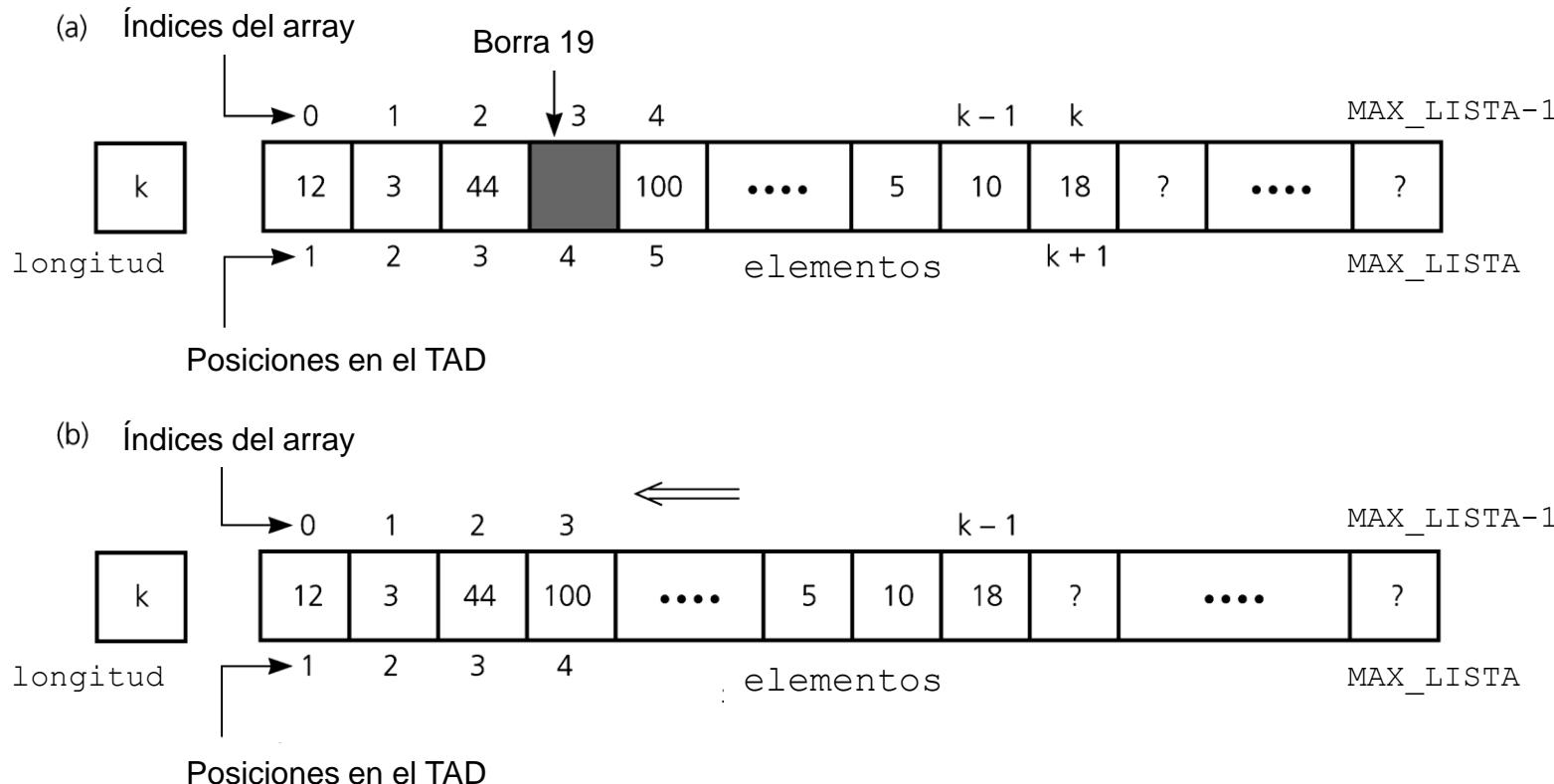
- Para insertar un elemento, hay que abrir un hueco:

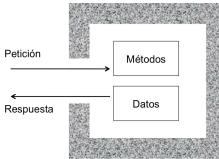


# Implementación del módulo lista



- Para eliminar un elemento, hay que cerrar hueco:





# Implementación del módulo lista

```

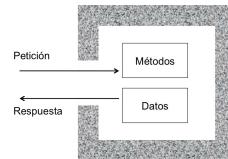
// Para evitar múltiples inclusiones de ficheros de definición
#ifndef _MLISTA_HPP_
#define _MLISTA_HPP_
#include <tr1/array>
namespace bblProgII{

 const unsigned MAX_LISTA = 100; // Máximo tamaño de la lista
 typedef int TpElemLista; // Lista de enteros
 typedef std::tr1::array <TpElemLista, MAX_LISTA> TpElementos; // Array de elementos
 struct TpLista{
 TpElementos elementos;
 unsigned longitud;
 };

 // Operaciones con listas:
 void crear(TpLista &lista); // Crea una lista de elementos vacía
 bool estaVacia(const TpLista &lista);
 unsigned obtenerLongitud(const TpLista &lista);
 void insertar(TpLista &lista, unsigned indice,
 const TpElemLista &nuevoElemento, bool &exito);
 void eliminar(TpLista &lista, unsigned indice, bool &exito);
 void consultar(const TpLista &lista, unsigned indice,
 TpElemLista &elemento, bool &exito);
} // Fin bblProgII
#endif

```

**MLista.hpp**



# Implementación del módulo lista

```
#include "MLista.hpp"
#include <tr1/array>
using namespace std; using namespace std::tr1;
// Parte privada del módulo
namespace {
 using namespace bblProgII;

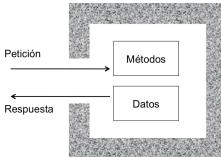
 unsigned int traducir(unsigned indice) {
 return indice - 1;
 } // Fin traducir

 void abrirHueco(TpLista &lista, unsigned indice) {
 for (unsigned pos = lista.longitud; pos > indice; --pos) {
 lista.elementos[pos] = lista.elementos[pos-1];
 } // Fin for
 } // Fin abrirHueco

 void cerrarHueco(TpLista &lista, unsigned indice) {
 for (unsigned pos = indice; pos < lista.longitud - 1; ++pos) {
 lista.elementos[pos] = lista.elementos[pos+1];
 } // Fin for
 } // Fin cerrarHueco
} // Del namespace

```

MLista.cpp



# Implementación del módulo lista

```

namespace bblProgII{
 void crear(TpLista &lista) {
 lista.longitud = 0;
 } // Fin del constructor

 bool estaVacia(const TpLista &lista) {
 return lista.longitud == 0;
 } // Fin de estaVacia

 unsigned int obtenerLongitud(const TpLista &lista) {
 return lista.longitud;
 } // Fin de obtenerLongitud()

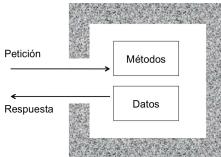
 void insertar(TpLista &lista, unsigned indice, const TpElemLista &nuevoElemento,
 bool &exito) {
 exito = (indice >= 1) &&
 (indice <= lista.longitud+1) &&
 (lista.longitud < MAX_LISTA);

 if (exito){
 abrirHueco(lista, traducir(indice));

 lista.elementos[traducir(indice)] = nuevoElemento; // Asignación debe estar definida
 ++lista.longitud; // Un elemento más en la lista
 } // Fin if
 } // Fin insertar
}

```

**MLista.cpp**



# Implementación del módulo lista

```

void eliminar(TpLista &lista, unsigned indice, bool &exito){
 exito = (indice >= 1) && (indice <= lista.longitud);

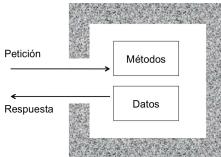
 if (exito){
 // Cerramos hueco desplazando a la izquierda los elementos a partir
 // de indice + 1
 cerrarHueco(lista, traducir(indice));
 --lista.longitud;
 } // Fin if
} // Fin eliminar

void consultar(const TpLista &lista, unsigned indice,
 TpElemLista &elemento, bool &exito){
 exito = (indice >= 1) && (indice <= lista.longitud);

 if (exito)
 elemento = lista.elementos[traducir(indice)];
 } // Fin consultar
} // Fin bblProgII

```

MLista.cpp



# Usando el módulo Lista

```

#include <iostream>
#include "MLista.hpp"

using namespace std;
using namespace bblProgII;

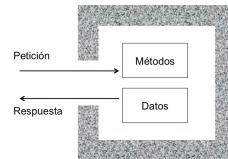
int main(){
 TpLista listaEnteros;
 int elem;
 bool exito;

 crear(listaEnteros); // Inicializamos la lista a lista vacía
 cout << "Longitud inicial: " << obtenerLongitud(listaEnteros) << endl;
 if (estaVacia(listaEnteros)){
 cout << "La lista está inicialmente vacía." << endl;
 }
 insertar(listaEnteros, 1, 10, exito);
 insertar(listaEnteros, 1, 5, exito);
 insertar(listaEnteros, 1, -5, exito);

 cout << "Nueva longitud de la lista: " << obtenerLongitud(listaEnteros) << endl;
 for (unsigned int pos = 1; pos <= obtenerLongitud(listaEnteros); ++pos){
 consultar(listaEnteros, pos, elem, exito);
 if (exito){
 cout << elem << " ";
 }
 } // Del for
 cout << endl;
} // Del main

```

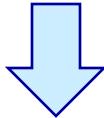
usoMLista.cpp



# Compilando y enlazando

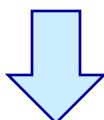
- **Compilando el módulo lista:**

```
g++ -c -o MLista.o MLista.cpp -Wall -Werror -Wextra -ansi
```



- **Compilando el cliente (programa principal):**

```
g++ -c -o usoMLista.o usoMLista.cpp -Wall -Werror -Wextra -ansi
```



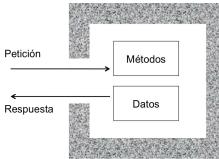
- **Enlazando y generando el ejecutable:**

```
g++ -o usoMLista MLista.o usoMLista.o -Wall -Werror -Wextra -ansi
```



## Anexo 3

# Sobrecarga de otros operadores



# Sobrecarga de oper. ++ y --

```

#ifndef __CLASS_PUNTO2D__
#define __CLASS_PUNTO2D__

namespace bblProgII{
 class CPunto2D{
 public:
 ... // Todos los métodos antes vistos
 // OPERADORES DE INCREMENTO Y DECREMENTO
 // Incremento prefijo (++p)
 CPunto2D & operator++(); // Devuelve una referencia a un objeto

 // Incremento postfijo (p++)
 CPunto2D operator++(int); // Devuelve un objeto

 // Decremento prefijo (--p)
 CPunto2D & operator--(); // Devuelve una referencia a un objeto

 // Decremento postfijo (p--)
 CPunto2D operator--(int); // Devuelve un objeto
 private:
 double x, y;
 }; // De CPunto2D
} // de bblProgII
#endif

```

CPunto2D.hpp

**private:**

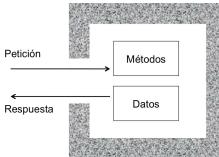
double x, y;

}; // De CPunto2D

} // de bblProgII

#endif

El empleo de un parámetro “instrumental” o “mudo”, de tipo entero, permite al compilador diferenciar entre la versión prefija (sin parámetro) y la postfija (con parámetro): el parámetro no sirve para otra cosa.



# Sobrecarga de oper. ++ y --

```
CPunto2D & CPunto2D::operator++() {
 x += 1.0; y += 1.0;
 return *this;
}
```

CPunto2D.cpp

Objeto actual: this apunta al objeto actual

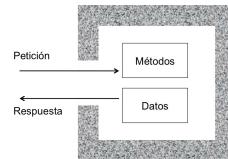
```
CPunto2D CPunto2D::operator++(int) {
 CPunto2D punto(x, y);
 x += 1.0; y += 1.0;
 return punto;
}
```

La referencia evita una llamada al constructor de copia

```
CPunto2D & CPunto2D::operator--() {
 x -= 1.0; y -= 1.0;
 return *this;
}
```

En cambio, en la versión postfija del operador no puede devolverse una referencia a punto, puesto que es un objeto local que se destruye al finalizar la función miembro.

```
CPunto2D CPunto2D::operator--(int) {
 CPunto2D punto(x, y);
 x -= 1.0; y -= 1.0;
 return punto;
}
```



# Sobrecarga de oper. ++ y --

```
#include "CPunto2D.hpp"
#include <iostream>

using namespace std;
using namespace bblProgII;

int main(){
 CPunto2D a(1.0, 2.0), b(-1.0, 4.0), c;

 c = a++; // Equivalente: c = a.operator++(0);
 cout << "c = a++ = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

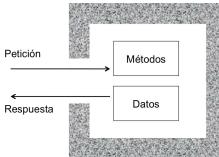
 c = ++a; // Equivalente: c = a.operator++();
 cout << "c = ++a = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

 c = b--; // Equivalente: c = a.operator--(0);
 cout << "c = b-- = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

 c = --b; // Equivalente: c = a.operator--();
 cout << "c = --b = (" << c.obtenerX() << ", " << c.obtenerY() << ")" << endl;

 cout << "a = (" << a.obtenerX() << ", " << a.obtenerY() << ")" << endl;
 cout << "b = (" << b.obtenerX() << ", " << b.obtenerY() << ")" << endl;
}
```

usoCPunto2D.cpp



# Sobrecarga de oprs. relacionales

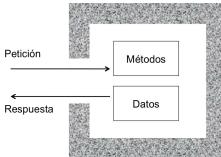
```

#ifndef __CLASS__PUNTO2D__
#define __CLASS__PUNTO2D__

namespace bblProgII{
 class CPunto2D{
 public:
 ... // Todos los métodos anteriormente vistos
 bool operator==(const CPunto2D &otro) const;
 bool operator!=(const CPunto2D &otro) const;
 bool operator<(const CPunto2D &otro) const;
 bool operator>(const CPunto2D &otro) const;
 bool operator<=(const CPunto2D &otro) const;
 bool operator>=(const CPunto2D &otro) const;
 private:
 double x, y;
 }; // De CPunto2D
} // de bblProgII
#endif

```

CPunto2D.hpp



# Sobrecarga de oprs. relacionales

```
// Comparación de igualdad ==
bool CPunto2D::operator==(const CPunto2D &otro) const{
 return (x == otro.x && y == otro.y);
}

// Comparación de desigualdad !=
bool CPunto2D::operator!=(const CPunto2D &otro) const{
 return (x != otro.x || y != otro.y);
}

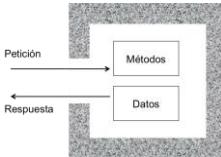
// Operador <: en función de la distancia al origen
bool CPunto2D::operator<(const CPunto2D &otro) const{
 CPunto2D origen;

 return (this->Distancia(origen) < otro.Distancia(origen));
}

// Operador >: en función de la distancia al origen
bool CPunto2D::operator>(const CPunto2D &otro) const{
 CPunto2D origen;

 return (this->Distancia(origen) > otro.Distancia(origen));
}
```

CPunto2D.cpp



# Sobrecarga de oprs. relacionales

CPunto2D.cpp

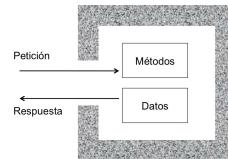
MATERIAL OPCIONAL

```
// Operador <=: en función de la distancia al origen
bool CPunto2D::operator<=(const CPunto2D &otro) const{
 CPunto2D origen;

 return (this->Distancia(origen) <= otro.Distancia(origen));
}

// Operador >=: en función de la distancia al origen
bool CPunto2D::operator>=(const CPunto2D &otro) const{
 CPunto2D origen;

 return (this->Distancia(origen) >= otro.Distancia(origen));
}
```



# Sobrecarga de oprs. relacionales

```

#include "CPunto2D.hpp"
#include <iostream>

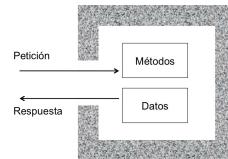
using namespace std;
using namespace bblProgII;

int main(){
 CPunto2D a(1.0, 2.0), b(-1.0, 4.0), c;
 ...

 if (a == c)
 cout << "a y c son iguales" << endl;
 if (a != c)
 cout << "a y c son distintos" << endl;
 cout << "Asigno a = c" << endl;
 a = c;
 cout << "Ahora..." << endl;
 if (a == c)
 cout << "a y c son iguales" << endl;
 if (a != c)
 cout << "a y c son distintos" << endl;
 if (a < b)
 cout << "a está más cerca del origen que b" << endl;
 else
 cout << "b está más cerca del origen que a" << endl;
}

```

usoCPunto2D.cpp



# Sobrecarga del operador [ ]

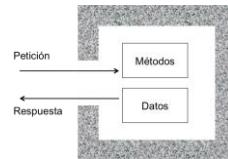
- Ejemplo para la clase CLista:

En el .hpp (CLista+.hpp):

```
TpElemLista &operator[] (unsigned indice);
// Devuelve una referencia a un elemento
// de la lista: permitirá el acceso con [] al
// elemento para consulta o modificación
```

En el .cpp (CLista+.cpp):

```
TpElemLista &CLista::operator[] (unsigned indice) {
 return elementos[indice];
}
```



# Sobrecarga del operador []

- Uso del operador [] en la clase CLista:

En usoCLista+.cpp:

```

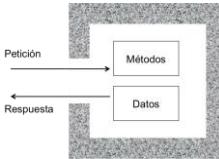
#include <iostream>
#include "CLista+.hpp"

using namespace std;
using namespace bblProgII;

int main(){
 CLista listaEnteros;
 bool exito;

 listaEnteros.insertar(1, 10, exito);
 listaEnteros.insertar(1, 5, exito);
 listaEnteros.insertar(1, -5, exito);
 listaEnteros[0] = 3;
 cout << "Longitud de la lista: " << listaEnteros.obtenerLongitud() << endl;
 for (unsigned pos = 0; pos < listaEnteros.obtenerLongitud(); ++pos){
 cout << listaEnteros[pos] << endl;
 }
 cout << endl;
}

```



# Sobrecarga de >> y <<

```
public:
...
// OPERADORES DE LECTURA/ESCRITURA
// Lectura por teclado:
friend istream & operator>>(istream &ent, CPunto2D &p);
```

CPunto2D.hpp

// Escritura por pantalla:

```
friend ostream & operator<<(ostream &sal, const CPunto2D &p);
```

**MATERIAL OPCIONAL**

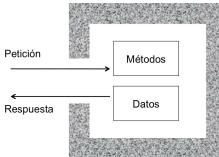
Las funciones **friend** no son miembros de la clase: se emplean cuando es necesario acceder a los miembros privados pero el operador actua como un función global: el objeto se pasa como parámetro y la función, al ser amiga, tiene acceso a la parte privada y pública.

Los operadores devuelven una referencia a un flujo (de entrada o salida): esto permitirá encadenar llamadas:  
cout << a << b...

cout << punto; equivale a:  
**operator<<(cout, punto);**

Obsérvense los dos parámetros necesarios.

La función amiga debe recibir dos parámetros: el flujo y el objeto que se va a leer/escribir. El objeto es necesario pasarlo porque la función amiga no es un método de la clase.



# Sobrecarga de >> y <<

CPunto2D.cpp

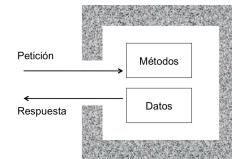
MATERIAL OPCIONAL

```

// OPERADORES DE ENTRADA/SALIDA
// Lectura por teclado:
istream & operator>>(istream &ent, CPunto2D &p) {
 cout << endl << "Coordenada X?: ";
 ent >> p.x;
 cout << "Coordenada Y?: ";
 ent >> p.y;
 return ent; // El flujo resultante se devuelve
}

// Escritura por pantalla:
ostream & operator<<(ostream &sal, const CPunto2D &p) {
 sal << "(" << p.x << ", " << p.y << ")";
 return sal; // El flujo resultante se devuelve
}

```



# Sobrecarga de >> y <<

usoCPunto2D.cpp

```

#include "CPunto2D.hpp"
#include <iostream>

using namespace std;
using namespace bblProgII;

int main() {
 CPunto2D a, b, c;
 ...

 cout << "Introduzca los puntos: " << endl;
 cin >> a >> b >> c; // Obsérvese el encadenamiento del operador >>
 cout << "Puntos introducidos: " << endl;
 cout << a << endl << b << endl << c << endl;
}

```