UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

TIPOS ABSTRACTOS DE DATOS

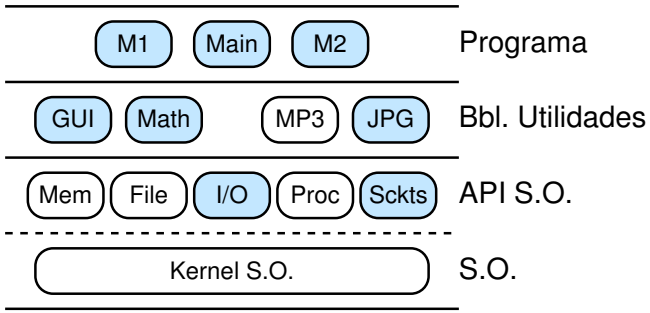
Tema 2

Programación II

Tema 2: TIPOS ABSTRACTOS DE DATOS

PROGRAMACIÓN MODULAR

- La división del código fuente en modulos presenta numerosas ventajas:
  - Aumenta la localidad, cohesión y aislamiento del código
    - Facilita la corrección de errores, la abstracción y modificación del código
  - Proporciona compilación separada de módulos
  - Fundamental para la *reutilización* de código.
    - Posibilita la creación, distribución y utilización de *bibliotecas*



INTERFAZ, IMPLEMENTACIÓN Y UTILIZACIÓN DE MÓDULOS

- Usualmente, en C++, un módulo de se compone de dos ficheros:
  - Interfaz (público): contiene las definiciones de tipos, constantes y prototipos de los subprogramas que el módulo ofrece (guardas previenen inclusión duplicada)
  - Implementación (privado): código que resuelve un determinado problema. Implementa los subprogramas que el módulo ofrece
- Un programa completo se compone de varios módulos, cada uno con su fichero de interfaz y de implementación, y de un módulo principal donde reside la función principal `main`.
  - El fichero de implementación deberá incluir el fichero de encabezamiento (interfaz) para acceder a las definiciones (`"..."`)
  - Para utilizar las facilidades que proporciona un módulo, se deberá incluir el fichero de encabezamiento (interfaz) del módulo que vaya a utilizar (`"..."`)

main.cpp (Principal)

```
#include "complejo.hpp"
int main()
{
   // Utilización
   // de Complejo
}
```

complejo.hpp (Interfaz)

```
#ifndef complejo_hpp_
#define complejo_hpp_
// Constantes
// Tipos
// Prototipos
#endif
```

complejo.cpp (Implementación)

```
#include "complejo.hpp"
// Constantes Auxiliares
// Tipos Auxiliares
// Subprogramas Auxiliares
// Subprogramas Públicos
...
```

COMPILACIÓN SEPARADA Y ENLAZADO

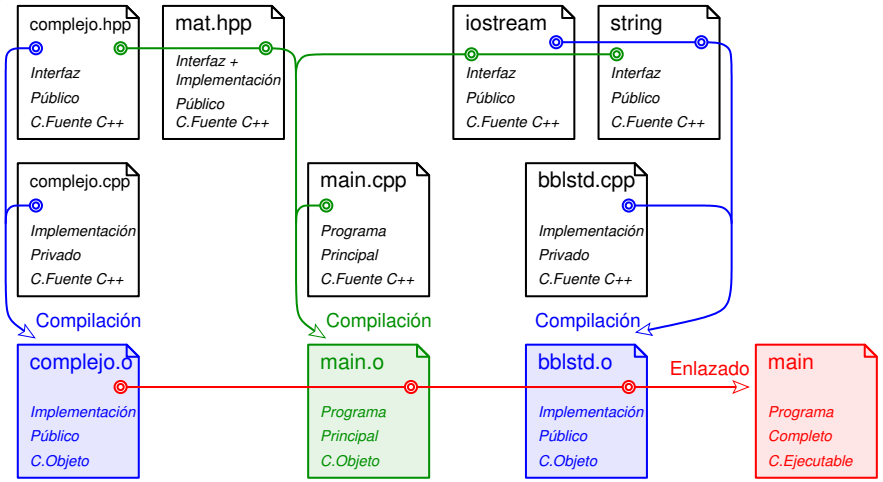
- Generación del programa ejecutable: dos procesos

  1. Compilación separada: (generación de código objeto)

     ```
     g++ -ansi -Wall -Wextra -Werror -c complejo.cpp
     g++ -ansi -Wall -Wextra -Werror -c main.cpp
     ```

  2. Enlazado: (generación de código ejecutable)

     ```
     g++ -ansi -Wall ... -o main main.o complejo.o
     ```



- Es posible realizar ambos procesos al mismo tiempo:

  ```
  g++ -ansi -Wall ... -o main main.cpp complejo.cpp
  ```

- Es posible combinar ambos procesos:

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

- La biblioteca estándar de C++ se enlaza automáticamente (no es necesario especificarla en el enlazado)

- Es posible enlazar con bibliotecas externas:

  ```
  g++ -ansi -Wall -Wextra -Werror -o main main.cpp complejo.cpp -ljpeg
  ```

ESPACIOS DE NOMBRES: DEFINICIÓN

- Para evitar posibles colisiones entre identificadores pertenencientes a distintos módulos existen los *espacios de nombres* (*namespace*)
- Permiten agrupar bajo una misma denominación un conjunto de declaraciones y definiciones
- Esta denominación será necesaria para identificar y diferenciar cada entidad declarada en un determinado espacio de nombres
- La inclusión de ficheros se debe realizar fuera del espacio de nombres
- En cada cpp se puede definir un espacio de nombres **Anónimo**, para la implementación privada de entidades auxiliares en el módulo de implementación

main.cpp (Principal)

```
#include <iostream>
#include "complejo.hpp"
using namespace std ;
using namespace umalcc ;
int main()
{
   // Utilización
   // del tipo
   // Complejo
}
```

complejo.hpp (Interfaz)

```
#ifndef complejo_hpp_
#define complejo_hpp_
#include <...otros...>
namespace umalcc {
   // Definición de
   // * Constantes
   // * Tipos
   // * Prototipos
}
#endif
```

complejo.cpp (Implementación)

```
#include "complejo.hpp"
#include <...otros...>
namespace {
   // Constantes Auxiliares
   // Tipos Auxiliares
   // Subprogramas Auxiliares
}
namespace umalcc {
   // Subprogramas Públicos
}
```

ESPACIOS DE NOMBRES: UTILIZACIÓN

- Para utilizar un determinado identificador definido dentro de un espacio de nombres es necesario algún tipo de *cualificación*
  - Desde *dentro del mismo espacio de nombres* no es necesario utilizar ningún tipo de cualificación para elementos pertenecientes al mismo espacio de nombres
  - En un fichero de *implementación* (`cpp`), la directiva `using namespace` permite utilizar todos los identificadores del espacio de nombres sin necesidad de cualificación (*cualificación implícita*)
  - En un fichero de *interfaz* (`hpp`), se debe utilizar la *cualificación explícita* (`::`) para referenciar a cualquier identificador perteneciente a otro espacio de nombres

main.cpp (Principal)

```
#include <iostream>
#include <string>
#include "datos.hpp"
using namespace std ;
using namespace umalcc ;
int main()
{
   string nombre ;
   Vector v ;
}
```

datos.hpp (Interfaz)

```
#ifndef datos_hpp_
#define datos_hpp_
#include <string>
#include <array>
#include <...otros...>
namespace umalcc {
   typedef std::array<int, 9> Vector ;
   void leer(std::string& nm, Vector& v) ;
}
#endif
```

ESPACIOS DE NOMBRES: COLISIONES

- Cuando se utilizan varios espacios de nombres de forma implícita, pueden suceder *colisiones* de identificadores definidos en ambos espacios de nombres
  - En este caso, se puede utilizar la cualificación explícita para deshacer la ambigüedad

main.cpp (Principal)

```
#include <iostream>
#include <string>
#include "datos.hpp"
using namespace std ;
using namespace umalcc ;
int main()
{
   string colision ;
   std::string nombre_1 ;
   umalcc::string nombre_2 ;
   ...
}
```

datos.hpp (Interfaz)

```
#ifndef datos_hpp_
#define datos_hpp_
#include <array>
#include <...otros...>
namespace umalcc {
   struct string {
      std::array<char, 50> datos ;
      int size ;
   } ;
   ...
}
#endif
```

PROGRAMACIÓN MODULAR. EJEMPLO

```
//- fichero: complejos.hpp -------------
#ifndef complejos_hpp_
#define complejos_hpp_
namespace umalcc {
    struct Complejo {
        double real ;
        double imag ;
    } ;
    void sumar(Complejo& r, const Complejo& a,
               const Complejo& b) ;
}
#endif
```

```
//- fichero: complejos.cpp -------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
namespace umalcc {
    void sumar(Complejo& r, const Complejo& a,
               const Complejo& b)
    {
        r.real = a.real + b.real ;
        r.imag = a.imag + b.imag ;
    }
}
//- fin: complejos.cpp -----------------
```

```
//- fichero: main.cpp ------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;
int main()
{
    Complejo c1 = { 3.4, 5.6 } ;
    Complejo c2 = { 1.3, 7.2 } ;
    Complejo c3 ;
    sumar(c3, c1, c2) ;
    cout << c3.real << " " << c3.imag << endl ;
}
//- fin: main.cpp ----------------------
```

Tema 2: TIPOS ABSTRACTOS DE DATOS

TIPOS ABSTRACTOS DE DATOS (TAD)

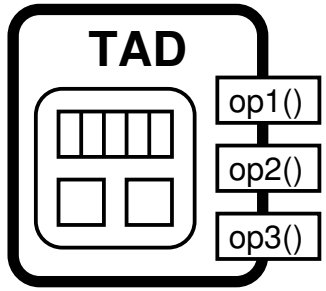
- Complejidad del problema a resolver

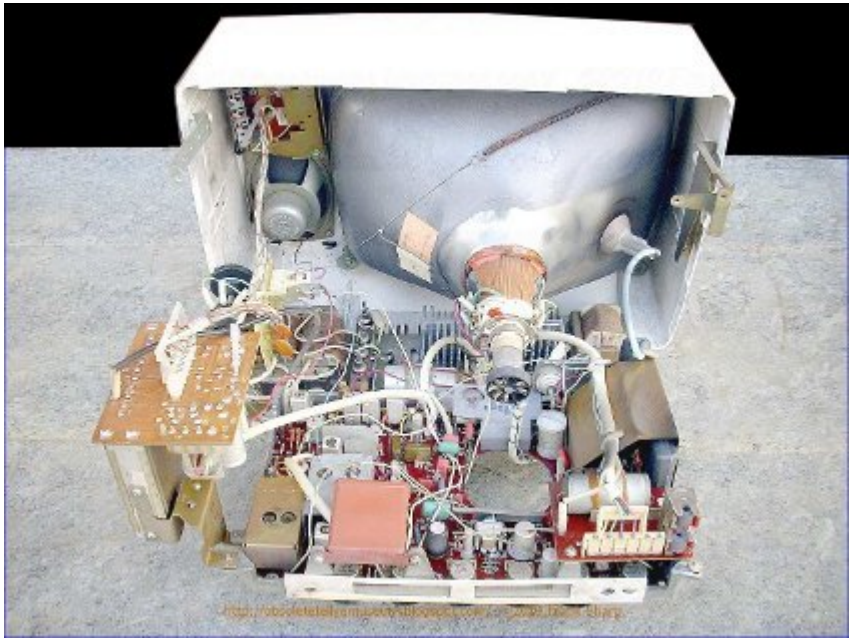
  $\Rightarrow$ Complejidad Algorítmica: Abstracción Procedimental/Modular

  $\Rightarrow$ Complejidad Estructuras de Datos: Tipos Abstractos de Datos

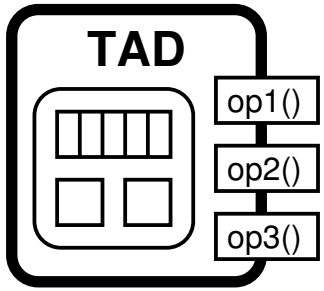
- Tipo Abstracto de Datos (TAD):

  - Especifica el concepto abstracto que representa
  - Especifica la semántica de las operaciones
  - Representación e implementación inaccesibles
  - Utilización independiente de la implementación





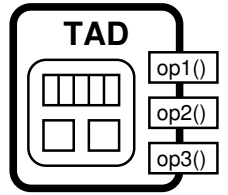

HYPER
http://obsoletetelevisionmuseum.blogspot.com/ © 2010 Frank Sharp


TAD
op1()
op2()
op3()

TIPOS ABSTRACTOS DE DATOS (TAD)

- Los Tipos Abstractos de Datos definen un paradigma de programación que nos permite diseñar programas definiendo **abstracciones** que modelan los **datos** que representan el problema que queremos resolver.

- Un tipo abstracto de datos **encapsula** una determinada estructura abstracta de datos



  - Impide su manipulación directa
  - Permite solamente su manipulación a través de las operaciones especificadas
  - Proporciona un mecanismo adecuado para el diseño y reutilización de software fiable y robusto.

- Para un TAD, se distinguen tres niveles:

Utilización de TAD
Especificación de TAD
Implementación de TAD

  - Utilización: basada en la especificación, independiente de su implementación. Manipulación mediante la invocación a operaciones públicas.
  - Especificación: (interfaz público) especifica el concepto abstracto que representa, y la semántica y restricciones de las operaciones.
  - Implementación: (privada) define las estructuras de datos internas y la implementación de las operaciones.

DEFINICIÓN DE TAD en C++: CLASES Y OBJETOS

- Las clases en C++ proporcionan un soporte adecuado a la utilización, definición, e implementación de tipos abstractos de datos
- Una Clase define una abstracción, y los métodos definen su comportamiento.
- Las clases permiten definir la representación interna (atributos) y las operaciones (métodos) que definen el TAD, así como su utilización
- Una *clase* define un TAD, y un *objeto* es una instancia de una clase (de igual forma que una variable es una instancia de un tipo de datos)
- Objetos: entidades activas que encapsulan datos y algoritmos.
  - Atributos (datos): contienen el estado y la representación interna del objeto, cuyo acceso está restringido.
  - Métodos (algoritmos): permiten la manipulación e interacción entre objetos. Definen el comportamiento del objeto.
- Definiremos las clases utilizando programación modular:
  - Definición de la clase en el fichero interfaz del módulo (`hpp`)
  - Implementación de la clase en el fichero de implementación del módulo (`cpp`)
  - Utilización de la clase de igual forma que la utilización de módulos.

DEFINICIÓN DE TAD en C++: Clase y Objeto

- Una **Clase** representa una abstracción de datos, especifica las características de unos **objetos**: su estado y su comportamiento.
- Un **objeto** es una *instancia* de una determinada *Clase*.
- Las características del *objeto* (estado y comportamiento) están determinadas por la *Clase* a la que pertenece.
- Puede haber muchos objetos **distintos** que sean de la misma *Clase* (y tambien de distintas Clases).
- Cada *objeto* almacena y contiene su propio estado interno (*atributos*), de forma *independiente* de los otros *objetos*.
- El objeto podrá ser manipulado e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.

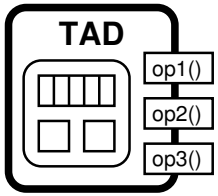
DEFINICIÓN DE TAD en C++: Métodos y Atributos

- La clase representa una abstracción de datos, y los métodos definen su comportamiento.
- Los **métodos** son algoritmos *especiales* definidos por la *Clase*, y se aplican sobre los objetos.
- Los métodos manipulan el estado interno del objeto sobre el que se aplican.
- Los métodos se invocan aplicándolos sobre un objeto.
- Los objetos responden a las invocaciones de los métodos dependiendo de su estado interno.
- La invocación a métodos puede llevar parámetros asociados, así como producir un resultado, además de manipular el estado interno del objeto sobre el que se aplica.
- Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.
- Los **atributos** almacenan los valores del estado interno del objeto.
- Cada objeto tiene un estado interno asociado, independiente de los otros objetos.
- Los atributos estan protegidos. Sólo se permite su acceso y manipulación a través de los métodos.

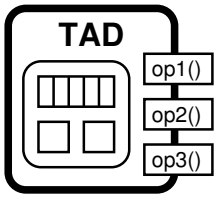
DEFINICIÓN DE CLASES EN C++

- Definición de clase en fichero `hpp`, con *guarda* y *espacio de nombres*
- `class` *Nombre* `{ ... } ;`
- Dos ámbitos de accesibilidad:
  - `public:` acceso externo permitido
  - `private:` acceso restringido (sólo interno)
- Atributos: representación interna (instancias independientes para cada objeto) (equivalentes a los campos de un registro)
- Constructor por defecto (construye e inicializa un objeto)
- Métodos (manipulan el objeto)
- Métodos const (**NO modifican** el objeto)
- Parámetros por *referencia* (normal o const)

```
//- fichero: complejos.hpp -------------
#ifndef complejos_hpp_
#define complejos_hpp_
namespace umalcc {

    class Complejo {

    public:     // zona pública

        Complejo() ;    // CTOR por Defecto

        void sumar(const Complejo& a,
                   const Complejo& b) ;

        void leer() ;

        void escribir() const ;

    private:    // zona privada

        double real ;   // ATRIBUTO

        double imag ;   // ATRIBUTO
    } ;
}
#endif
//- fin: complejos.hpp -----------------
```

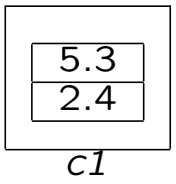


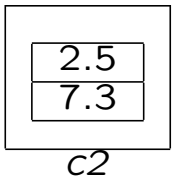
UTILIZACIÓN: OBJETOS COMO INSTANCIAS DE CLASES

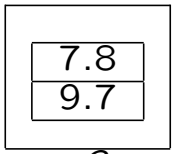


- Incluir `hpp` y uso de *espacio de nombres*
- Objeto como instancia de una clase (propia representación interna independiente) (equiv. a variable como instancia de tipo)
- Tiempo de vida: *construcción* y *destrucción*
- La parte privada del objeto está protegida (no es posible su acceso externo)
- Manipulación del objeto a través de sus métodos públicos
- Invocación a métodos: el operador punto (aplicación a objeto concreto)
- Parámetros por *referencia* (normal o const)
- Sólo los métodos constantes se pueden aplicar a los objetos constantes

```
//- fichero: main.cpp -----------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2, c3 ; // Ctor Defecto

    c1.leer() ;

    c2.leer() ;

    c3.sumar(c1, c2) ;

    c3.escribir() ;
} // invocación implícita al Destructor
//- fin: main.cpp ---------------------
```

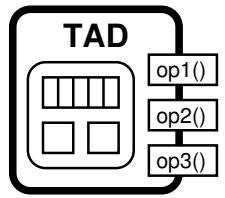






IMPLEMENTACIÓN DE CLASES

- Implementación en fichero `cpp`, en el mismo *espacio de nombres* que la definición (hpp)
- Definición de constantes, tipos y subprogramas *auxiliares* en espacio de nombres anónimo
- Implementación de Constructores y Métodos: *cualificación explícita* con identif. de la clase
- *Acceso directo* a **atributos** del propio objeto
- *Operador punto* (.) para acceder a **atributos** de otros objetos de la misma clase
- Invocación *directa* a **métodos** aplicados sobre el propio objeto invocado
- *Operador punto* (.) para invocar a **métodos** sobre otros objetos
- Constructor:
  - lista de inicialización: invocación de constructores sobre cada atributo (en orden). Se recomienda **invocación explícita** a los constructores. La invocación implícita deja los tipos simples sin inicializar.
  - cuerpo: código para inicialización extra




```
//- fichero: complejos.cpp -------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
namespace { // Anónimo
    double vabs(double x) { /* ... */}
}
namespace umalcc {
    Complejo::Complejo()
        : real(0.0), imag(0.0) { }
    void Complejo::sumar(const Complejo& a,
                         const Complejo& b)
    {
        real = a.real + b.real ;
        imag = a.imag + b.imag ;
    }
    void Complejo::escribir() const
    {
        cout << "{ " << real << ", " << imag << " }" ;
    }
    void Complejo::leer()
    {
        cin >> real >> imag ;
    }
}
//- fin: complejos.cpp -----------------
```

MÉTODOS PRIVADOS

- Es posible definir métodos privados dentro de una clase. Ayudan a la resolución de otros métodos de la clase
- Se definen en la parte privada de la clase
- Se implementan de igual forma a los métodos públicos de la clase
- Sólo pueden ser invocados desde los métodos propios de la clase.
- Invocación *directa* a *métodos privados* aplicados sobre el propio objeto invocado
- *Operador punto* (.) para invocar a *métodos privados* sobre otros objetos de la misma clase
- En el espacio de nombres **anónimo** (cpp) se pueden definir **subprogramas** auxiliares privados.
- Metodo público: invocación pública. Accede al estado interno del objeto.
- Metodo privado: invocación restringida a la clase. Accede al estado interno del objeto.
- Subprograma privado: invocación restringida al cpp. **NO** accede al estado interno del objeto.

```
//-- complejo.hpp ----------------
namespace umalcc {
  class Complejo {
  public:
      void metodo(Complejo& a) ;
  private:
      void normalizar() ;
  };
}
//-- complejo.cpp ----------------
namespace {  // Anónimo
  void vabs(double& x) { /* ... */}
}
namespace umalcc {
  void Complejo::normalizar() { /* ... */ }
  void Complejo::metodo(Complejo& a) {
      a.normalizar() ; // invoca sobre el objeto a
      normalizar() ;   // invoca sobre el propio objeto
      vabs(real);      // invoca sin objeto
  }
}
//-- main.cpp --------------------
#include "complejo.hpp"
using namespace umalcc;
int main() {
    Complejo c1, c2 ;
    c1.metodo(c2) ;
}
```

CONSTANTES Y TIPOS DE ÁMBITO DE CLASE

- Constantes y Tipos de ámbito de clase. Usualmente Privados. A veces Públicos.
- Las constantes integrales (ordinales) se deben especificar como `static const`
- Sin embargo, es preferible definir las constantes de tipo real (`double`) en el ámbito externo a la clase (en el `hpp` si debe ser pública, o en el `cpp` si debe ser privada)
- Los Tipos deben ser públicos si forman parte de los parámetros de métodos públicos
- Para utilizar **externamente** tanto los tipos como las constantes públicas de una clase, se deben cualificar explícitamente con el nombre de la clase (notación `::`)

```
//-- main.cpp -----------------------
#include "vector.hpp"
int main()
{
    double err = ERROR;
    Vector v ;
    Vector::Elemento e = { /* ... */ } ;
    v.anyadir(e) ;
    if (v.size() < Vector::LIMITE) { /* ... */ }
}
```

```
//-- vector.hpp ----------------------
const double ERROR = 1e-10;
class Vector {
public:
    static const int LIMITE = 100 ;
    struct Elemento { /* ... */ } ;
    Vector() ;
    int size() const;
    void anyadir(const Elemento& e) ;
private:
    static const int MAX = 256 ;
    typedef std::array<Elemento, MAX> Datos ;
    //----------
    int sz;
    Datos dat;
};
//-- vector.cpp ----------------------
Vector::Vector() : sz(0), dat() { }
int Vector::size() const { return sz ; }
void Vector::anyadir(const Elemento& e)
{
    if (sz < MAX) {
        dat[sz] = e ;
        ++sz ;
    }
}
```

EL DESTRUCTOR

- Se invoca **implícitamente** cuando se destruye un objeto al expirar su tiempo de vida
- Libera los recursos asociados al objeto
- No tiene parámetros
- Si es necesario, el cuerpo del destructor especifica código extra de liberación
- Automáticamente invoca al destructor sobre cada atributo del objeto
- Si no lo define el programador, entonces lo define automáticamente el compilador *(invocando implícitamente al destructor sobre cada atributo del objeto)*

```
//-- complejo.hpp ----------------
class Complejo {
public:
    ~Complejo() ;
    // ...
};

//-- complejo.cpp ----------------
Complejo::~Complejo() { }

//-- main.cpp --------------------
#include "complejo.hpp"
int main()
{
    Complejo c1 ;  // Invocación al Ctor Defecto
    // ...
} // Invocación implícita del destructor sobre c1
```

- El destructor de un tipo simple no hace nada
- El destructor del tipo `std::string` libera la memoria asociada
- El destructor del tipo `std::array` invoca al destructor sobre cada elemento del array.
- El destructor de un registro invoca al destructor sobre cada campo del registro

EL CONSTRUCTOR POR DEFECTO

- Si no se especifica explícitamente ningún otro mecanismo de construcción del objeto, se invoca **implícitamente** cuando se construye un objeto al comenzar su tiempo de vida
- Construye e inicializa un objeto
- No tiene parámetros
- La lista de inicialización invoca a los constructores sobre cada atributo del objeto (siguiendo el orden de declaración)
- Si es necesario, el cuerpo del constructor especifica código extra de inicialización
- Si el programador no define explícitamente ningún constructor, entonces lo define automáticamente el compilador *(invocando implícitamente al constructor por defecto sobre cada atributo del objeto)*

- La invocación implícita al constructor por defecto de los tipos simples queda **sin inicializar**. Sin embargo, la invocación explícita (directa o indirecta) inicializa con el valor **cero**.
- El constructor por defecto del tipo `std::string` inicializa el valor a vacía (`""`).

```
//-- complejo.hpp ----------------
class Complejo {
public:
    Complejo() ;
    // ...
};
//-- complejo.cpp ----------------
Complejo::Complejo() : real(), imag() { }
//-- main.cpp --------------------
#include "complejo.hpp"
int main()
{
    Complejo c1 ;     // Invocación al Ctor Defecto
    c1 = Complejo() ; // Invocación explícita
    // ...
} // Invocación implícita del destructor sobre c1
```

- El constructor por defecto del tipo `std::array` invoca implícitamente al constructor por defecto sobre cada elemento del array.
- El constructor por defecto de un registro invoca implícitamente al constructor por defecto sobre cada campo del registro

EL CONSTRUCTOR DE COPIA

- Se invoca **explícitamente** cuando se construye un objeto, al comenzar su tiempo de vida, copiando el valor de otro objeto de la misma clase.
- Se invoca **implícitamente** cuando un objeto se pasa como *parámetro por valor*, y cuando un objeto se *devuelve* desde una función.
- Construye e inicializa un objeto por copia
- Un parámetro de la misma clase (Ref-Cte)
- La lista de inicialización invoca a los constructores sobre cada atributo del objeto (siguiendo el orden de declaración)
- Si es necesario, el cuerpo del constructor especifica código extra de inicialización
- Si no lo define el programador, entonces lo define automáticamente el compilador *(invocando al constructor de copia sobre cada atributo del objeto)*

- El constructor de copia de los tipos simples copia el valor especificado como parámetro
- El constructor de copia del tipo `std::string` copia el valor especificado como parámetro

```
//-- complejo.hpp ----------------
class Complejo {
public:
    Complejo(const Complejo& o) ;
};
//-- complejo.cpp ----------------
Complejo::Complejo(const Complejo& o)
    : real(o.real), imag(o.imag) { }
//-- main.cpp --------------------
#include "complejo.hpp"
int main()
{
    Complejo c1 ;      // Invocación al Ctor Defecto
    Complejo c2(c1) ;  // Invocación al Ctor de Copia
    Complejo c3 = c1 ; // Invocación al Ctor de Copia
    // ...
} // Invocación del destructor sobre c3, c2 y c1
```

- El constructor de copia del tipo `std::array` invoca al constructor de copia sobre cada elemento del array.
- El constructor de copia de un registro invoca al constructor de copia sobre cada campo del registro

CONSTRUCTORES ESPECÍFICOS

- Se invoca **explícitamente** cuando se construye un objeto, al comenzar su tiempo de vida, utilizando los parámetros para inicializar el objeto
- Construye e inicializa un objeto utilizando los valores de los parámetros
- Uno o varios parámetros de tipos diversos
- Pueden definirse múltiples constructores específicos, diferenciándose por los parámetros
- La lista de inicialización invoca a los constructores sobre cada atributo del objeto (siguiendo el orden de declaración)
- Si es necesario, el cuerpo del constructor especifica código extra de inicialización

```
//-- complejo.hpp ----------------
class Complejo {
public:
    Complejo(double p_real, double p_imag) ;
};

//-- complejo.cpp ----------------
Complejo::Complejo(double p_real, double p_imag)
    : real(p_real), imag(p_imag) { }

//-- main.cpp --------------------
#include "complejo.hpp"
int main()
{
    Complejo c1(2.5, 7.3) ;   // Ctor Especifico
    c1 = Complejo(3.4, 5.6) ; // Invocación explícita
    // ...
} // Invocación del destructor sobre c1
```

EL OPERADOR DE ASIGNACIÓN

- Se invoca **explícitamente** mediante el operador de asignación (=), asignando el valor de otro objeto de la misma clase.
- Asigna al objeto receptor de la asignación un nuevo valor, como copia del valor de otro objeto, destruyendo previamente, si es preciso, el valor almacenado.
- Un parámetro de la misma clase (Ref-Cte)
- Antes de realizar la asignación, se debe comprobar que no sea una autoasignación `if (this != &o) { /* ... */ }`
  - `this` es la dirección en memoria donde está almacenado el propio objeto (receptor de la asignación)
  - `&o` es la dirección en memoria donde está almacenado el objeto recibido como parámetro
- Devuelve una referencia al propio objeto `*this` (que nosotros **no utilizamos**)
- Si no lo define el programador, entonces lo define automáticamente el compilador *(invocando al operador de asignación sobre cada atributo del objeto)*

```
//-- complejo.hpp ----------------
class Complejo {
public:
    Complejo& operator=(const Complejo& o) ;
};
//-- complejo.cpp ----------------
Complejo& Complejo::operator=(const Complejo& o)
{
    // compara direcciones de memoria de los objetos
    if (this != &o) {
        // destruir valor anterior y asignar nuevo
        real = o.real ;
        imag = o.imag ;
    }
    return *this ;   // devuelve el propio objeto
}
//-- main.cpp --------------------
#include "complejo.hpp"
int main()
{
    Complejo c1(2.5, 7.3) ;
    Complejo c2 ;
    c2 = c1 ;                  // Operador Asignación
    c2 = Complejo(3.4, 5.6) ; // Operador Asignación
    // ...
}
```

Tema 2: TIPOS ABSTRACTOS DE DATOS

EJEMPLO: TAD PUNTO DEL PLANO CARTESIANO (I)

- Un *Punto* describe una determinada posición en el *plano cartesiano*, especificada por el valor de las componentes X e Y de las coordenadas cartesianas.

- El TAD Punto (*nsp* `umalcc`) proporciona las siguientes operaciones públicas:
  - Constructor por defecto: construye un objeto *Punto* en el origen de coordenadas.
  - Constructor de copia: construye un objeto *Punto* copiando sus valores de otro objeto *Punto*.
  - Operador de asignación: asigna a un objeto *Punto* el valor de otro objeto *Punto*.
  - Constructor específico: construye un objeto *Punto* según los valores de las coordenadas especificadas como parámetros.
  - Desplazar el objeto *Punto* desde la posición actual una determinada distancia especificada como parámetros en ambos ejes.
  - Calcular la distancia absoluta entre dos objetos *Punto* en el plano cartesiano.
  - Acceder a los valores de ambas componentes X e Y del objeto *Punto*.
  - Destructor: destruye el objeto *Punto* y todos sus recursos asociados.

EJEMPLO: TAD PUNTO DEL PLANO CARTESIANO (II)

```
#ifndef punto_hpp_
#define punto_hpp_
namespace umalcc {
    class Punto {
    public:
        ~Punto();
        Punto();
        Punto(double x, double y);
        Punto(const Punto& o);
        Punto& operator=(const Punto& o);
        void desplazar(double dx, double dy);
        double distancia(const Punto& o) const;
        double coord_x() const;
        double coord_y() const;
    private:
        double x;
        double y;
    } ;
}
#endif
```

```
#include "punto.hpp"
#include <cmath>
using namespace std;
namespace {
    inline double sq(double x) {
        return x * x;
    }
}
namespace umalcc {
    Punto::~Punto() {}
    Punto::Punto() : x(0), y(0) {}
    Punto::Punto(double a, double b) : x(a), y(b) {}
    Punto::Punto(const Punto& o) : x(o.x), y(o.y) {}
    Punto& Punto::operator=(const Punto& o) {
        if (this != &o) {
            x = o.x;
            y = o.y;
        }
        return *this;
    }
    void Punto::desplazar(double dx, double dy) {
        x += dx;
        y += dy;
    }
    double Punto::distancia(const Punto& o) const {
        return sqrt( sq(o.y - y) + sq(o.x - x) );
    }
    double Punto::coord_x() const { return x; }
    double Punto::coord_y() const { return y; }
}
```

EJEMPLO: TAD PUNTO DEL PLANO CARTESIANO (III)

```
#include <iostream>
#include "punto.hpp"
using namespace std;
using namespace umalcc;
void leer(Punto& p) {
    double x, y;
    cout << "Introduce las coordenadas X e Y: ";
    cin >> x >> y;
                      //ALT// Punto aux(x, y);
    p = Punto(x, y);  //ALT// p = aux;
}
void mostrar(const Punto& p) {
    cout << "(" << p.coord_x() << ", "
                << p.coord_y() << ")";
}
int main() {
    Punto malaga(-4.4750, 36.7166);
    Punto pepe(malaga);
    pepe.desplazar(-0.0344, -0.1171);
    cout << "Pepe: posicion: ";
    mostrar(pepe);
    cout << endl;
    cout << pepe.distancia(malaga) << endl;
    leer(pepe);
    mostrar(pepe);
    cout << endl;
    cout << pepe.distancia(malaga) << endl;
}
```

- El subprograma `leer` pretende modificar un objeto *punto* (`p`) que ya está creado (no se pueden invocar a los constructores sobre él).
- Sin embargo, la clase `Punto` **no** ofrece métodos públicos suficientes para modificar los atributos del objeto `p` de la forma requerida.
- Por lo tanto, en este caso, es necesario **crear** un nuevo objeto auxiliar (de la clase `Punto`) con lo nuevos valores, y **asignar** el nuevo objeto auxiliar al objeto original `p`, modificando de esta forma los valores de sus atributos.
  - Implementación Alternativa 1:

    ```
    Punto aux(x, y);
    p = aux;
    ```

  - Implementación Alternativa 2:

    ```
    p = Punto(x, y);
    ```

EJEMPLO: TAD SEGMENTO DEL PLANO CARTESIANO (I)

- Un *Segmento* es un fragmento de una recta que se encuentra comprendido entre dos *Puntos* en el *plano cartesiano*.

- El TAD Segmento (*nsp* `umadpt`) proporciona las siguientes operaciones públicas:
  - Constructor por defecto: construye un objeto *Segmento* nulo, donde ambos puntos que lo delimitan se encuentran en el origen de coordenadas.
  - Constructor de copia: construye un objeto *Segmento* copiando sus valores de otro objeto *Segmento*.
  - Operador de asignación: asigna a un objeto *Segmento* el valor de otro objeto *Segmento*.
  - Constructor específico: construye un objeto *Segmento* según los valores de dos puntos especificados como parámetros.
  - Constructor específico: construye un objeto *Segmento* según los valores de las 4 coordenadas de dos puntos especificados como parámetros.
  - Desplazar el objeto *Segmento* desde la posición actual una determinada distancia especificada como parámetros en ambos ejes.
  - Calcular la longitud del objeto *Segmento*.
  - Acceder a los valores de ambas puntos que definen al objeto *Segmento*.
  - Destructor: destruye el objeto *Segmento* y todos sus recursos asociados.

EJEMPLO: TAD SEGMENTO DEL PLANO CARTESIANO (II)

```
#ifndef segmento_hpp_
#define segmento_hpp_
#include "punto.hpp"
namespace umadpt {
    class Segmento {
    public:
        ~Segmento();
        Segmento();
        Segmento(double x1, double y1,
                 double x2, double y2);
        Segmento(const umalcc::Punto& a,
                 const umalcc::Punto& b);
        Segmento(const Segmento& o);
        Segmento& operator=(const Segmento& o);
        void desplazar(double dx, double dy);
        double longitud() const;
        umalcc::Punto origen() const;
        umalcc::Punto destino() const;
    private:
        umalcc::Punto org;
        umalcc::Punto dst;
    } ;
}
#endif
```

```
#include "segmento.hpp"
using namespace umalcc;
namespace umadpt {
    Segmento::~Segmento() {}
    Segmento::Segmento() : org(), dst() {}
    Segmento::Segmento(double x1, double y1,
                       double x2, double y2)
        : org(x1, y1), dst(x2, y2) {}
    Segmento::Segmento(const Punto& a, const Punto& b)
        : org(a), dst(b) {}
    Segmento::Segmento(const Segmento& o)
        : org(o.org), dst(o.dst) {}
    Segmento& Segmento::operator=(const Segmento& o) {
        if (this != &o) {
            org = o.org;
            dst = o.dst;
        }
        return *this;
    }
    void Segmento::desplazar(double dx, double dy) {
        org.desplazar(dx, dy);
        dst.desplazar(dx, dy);
    }
    double Segmento::longitud() const {
        return org.distancia(dst);
    }
    Punto Segmento::origen() const { return org; }
    Punto Segmento::destino() const { return dst; }
}
```

EJEMPLO: TAD SEGMENTO DEL PLANO CARTESIANO (III)

```
#include <iostream>
#include "segmento.hpp"
using namespace std;
using namespace umalcc;
using namespace umadpt;
void mostrar(const Punto& p) {
    cout << "(" << p.coord_x() << ", " << p.coord_y() << ")";
}
int main() {
    Punto malaga(-4.4750, 36.7166);
    Punto benalmadena(-4.5094, 36.5995);
    Segmento camino(malaga, benalmadena);
    camino.desplazar(-0.0344, -0.1171);
    cout << "Origen: ";
    mostrar(camino.origen());
    cout << endl;
    cout << "Destino: ";
    mostrar(camino.destino());
    cout << endl;
    cout << "Longitud: " << camino.longitud() << endl;
}
```

EJEMPLO: TAD POLÍGONO DEL PLANO CARTESIANO (I)

- Un *Polígono* es una figura plana compuesta por una secuencia finita de *lados* consecutivos determinados por *vértices* (*Puntos*) en el *plano cartesiano*.
- El TAD Polígono (*nsp* `umalcc`) proporciona las siguientes operaciones públicas:
  - Constructor por defecto: construye un objeto *Polígono* vacío, sin vértices ni lados.
  - Constructor de copia: construye un objeto *Polígono* copiando sus valores de otro objeto *Polígono*.
  - Operador de asignación: asigna a un objeto *Polígono* el valor de otro objeto *Polígono*.
  - Añadir un nuevo vértice al objeto *Polígono*.
  - Cambiar la posición de un determinado vértice del objeto *Polígono*.
  - Desplazar el objeto *Polígono* desde la posición actual una determinada distancia especificada como parámetros en ambos ejes.
  - Calcular el perímetro del objeto *Polígono*.
  - Acceder a los valores de los vértices y de los lados que definen al objeto *Polígono*.
  - Destructor: destruye el objeto *Polígono* y todos sus recursos asociados.

EJEMPLO: TAD POLÍGONO DEL PLANO CARTESIANO (II)

```
#ifndef plgn_hpp_
#define plgn_hpp_
#include "punto.hpp"
#include "segmento.hpp"
#include <array>
namespace umalcc {
    class Plgn {
    public:
        ~Plgn();
        Plgn();
        Plgn(const Plgn& o);
        Plgn& operator=(const Plgn& o);
        void nuevo_vertice(const Punto& v);
        void mover_vertice(int pos, const Punto& v);
        void desplazar(double dx, double dy);
        double perimetro() const;
        Punto vertice(int pos) const;
        umadpt::Segmento lado(int pos) const;
        int n_vertices() const;
    private:
        static const int MAX = 32;
        typedef std::array<Punto, MAX> Vertices;
        //---------------
        int n_vert;
        Vertices vert;
    } ;
}
#endif
```

EJEMPLO: TAD POLÍGONO DEL PLANO CARTESIANO (III)

```
#include "plgn.hpp"
using namespace std;
using namespace umadpt;
namespace umalcc {
    Plgn::~Plgn() {}
    Plgn::Plgn() : n_vert(0), vert() {}
    Plgn::Plgn(const Plgn& o)
        : n_vert(o.n_vert), vert(o.vert) {}
    Plgn& Plgn::operator=(const Plgn& o) {
        if (this != &o) {
            n_vert = o.n_vert;
            vert = o.vert;
        }
        return *this;
    }
    void Plgn::nuevo_vertice(const Punto& v) {
        if (n_vert < int(vert.size())) {
            vert[n_vert] = v;
            ++n_vert;
        }
    }
    void Plgn::mover_vertice(int pos,
                             const Punto& v) {
        if (0 <= pos && pos < n_vert) {
            vert[pos] = v;
        }
    }                                          }
    int Plgn::n_vertices() const { return n_vert; }
    void Plgn::desplazar(double dx, double dy) {
        for (int i = 0; i < n_vert; ++i) {
            vert[i].desplazar(dx, dy);
        }
    }
    double Plgn::perimetro() const {
        double suma = 0;
        for (int i = 0; i < n_vert - 1; ++i) {
            suma += vert[i].distancia(vert[i+1]);
        }
        return suma;
    }
    Punto Plgn::vertice(int pos) const {
        Punto p;
        if (0 <= pos && pos < n_vert) {
            p = vert[pos];
        }
        return p;
    }
    Segmento Plgn::lado(int pos) const {
        Segmento s;
        if (0 <= pos && pos < n_vert - 1) {
            s = Segmento(vert[pos], vert[pos+1]);
        }
        return s;
    }
```

EJEMPLO: TAD POLÍGONO DEL PLANO CARTESIANO (IV)

```
#include <iostream>
#include "punto.hpp"
#include "plgn.hpp"
using namespace std;
using namespace umalcc;
void mostrar(const Punto& p) {
    cout << "(" << p.coord_x() << ", " << p.coord_y() << ")";
}
int main() {
    Punto malaga(-4.4750, 36.7166);
    Punto benalmadena(-4.5094, 36.5995);
    Punto mijas(-4.6497, 36.5959);
    Punto cartama(-4.6350, 36.7124);
    Plgn camino;
    camino.nuevo_vertice(malaga);
    camino.nuevo_vertice(benalmadena);
    camino.nuevo_vertice(mijas);
    camino.nuevo_vertice(cartama);
    camino.nuevo_vertice(malaga);
    cout << "Origen: ";
    mostrar(camino.vertice(0));
    cout << endl;
    cout << "Perimetro: " << camino.perimetro() << endl;
}
```

EJEMPLO: TAD LISTA DE PUNTOS (I)

- Diseñe un TAD lista posicional (en *namespace* `umadpt`) que contiene una secuencia de elementos homogéneos (de tipo `Punto`) accesibles por su posición, que defina los siguientes métodos:
  - Destructor: destruye y libera los recursos asociados a la lista
  - Constructor por defecto: construye una lista vacía
  - Constructor de copia: copia el contenido de otra lista
  - Operador de asignación: asigna el contenido de otra lista
  - Llena: método constante que devuelve si el número de elementos almacenados en la lista alcanzó su máxima capacidad
  - Size: método constante que devuelve el número de elementos almacenados en la lista
  - Clear: elimina los elementos almacenados, dejando la lista vacía
  - Insertar: inserta el elemento recibido como parámetro en la posición indicada como parámetro *PRECOND:* `( ! llena() && 0 <= pos && pos <= size() )`
  - Eliminar: elimina el elemento que se encuentra en la posición indicada como parámetro *PRECOND:* `( 0 <= pos && pos < size() )`
  - Acceder: método constante que devuelve el elemento almacenado en la posición indicada como parámetro *PRECOND:* `( 0 <= pos && pos < size() )`
  - Modificar: asigna el elemento recibido como parámetro al elemento que se encuentra en la posición indicada como parámetro *PRECOND:* `( 0 <= pos && pos < size() )`
  - Buscar: devuelve la posición donde se encuentra el elemento recibido como parámetro, o `-1` si no lo encuentra

EJEMPLO: TAD LISTA DE PUNTOS (II)

```
#ifndef listapuntos_hpp_
#define listapuntos_hpp_
#include <array>
#include "punto.hpp"
namespace umadpt {
  class ListaPuntos {
  public:
    // ~ListaPuntos() ;                    // Destructor Automático
    ListaPuntos() ;
    ListaPuntos(const ListaPuntos& o) ;
    ListaPuntos& operator = (const ListaPuntos& o) ;
    bool llena() const ;
    int size() const ;
    void clear() ;
    void insertar(int pos, const umalcc::Punto& dato) ;
    void eliminar(int pos) ;
    umalcc::Punto acceder(int pos) const ;
    void modificar(int pos, const umalcc::Punto& dato);
    int buscar(const umalcc::Punto& dato) ;
  private:
    static const int MAX = 100;
    typedef std::array<umalcc::Punto, MAX> Datos;
    //----------------------------
    void abrir_hueco(int pos) ;
    void cerrar_hueco(int pos) ;
    //----------------------------
    int sz;     // numero de elementos de la lista
    Datos v;    // contiene los elementos de la lista
  };
}
#endif
```

EJEMPLO: TAD LISTA DE PUNTOS (III)

```
#include "listapuntos.hpp"
#include <cassert>
using namespace umalcc ;
namespace {
  bool iguales(const Punto& p1, const Punto& p2)
  {
    return (p1.coord_x() == p2.coord_x())
        && (p1.coord_y() == p2.coord_y());
  }
}
namespace umadpt {
  // ListaPuntos::~ListaPuntos() {}
  ListaPuntos::ListaPuntos() : sz(0), v() {}
  ListaPuntos::ListaPuntos(const ListaPuntos& o)
    : sz(o.sz), v()
  {
    for (int i = 0; i < sz; ++i) {
      v[i] = o.v[i] ;
    }
  }
  ListaPuntos&
  ListaPuntos::operator=(const ListaPuntos& o)
  {
    if (this != &o) {
      sz = o.sz ;
      for (int i = 0; i < sz; ++i) {
        v[i] = o.v[i] ;
      }
    }
    return *this ;
  }
```

```
bool ListaPuntos::llena() const
{
  return sz == int(v.size());
}
int ListaPuntos::size() const
{
  return sz ;
}
void ListaPuntos::clear()
{
  sz = 0 ;
}
void ListaPuntos::insertar(int pos,
                           const Punto& dato)
{
  assert( ! llena() && pos >= 0 && pos <= size()) ;
  abrir_hueco(pos) ;
  v[pos] = dato ;
}
void ListaPuntos::eliminar(int pos)
{
  assert(pos >= 0 && pos < size()) ;
  cerrar_hueco(pos) ;
}
```

EJEMPLO: TAD LISTA DE PUNTOS (IV)

```
Punto ListaPuntos::acceder(int pos) const
{
  assert(pos >= 0 && pos < size()) ;
  return v[pos] ;
}
void ListaPuntos::modificar(int pos,
                            const Punto& dato)
{
  assert(pos >= 0 && pos < size()) ;
  v[pos] = dato;
}
int ListaPuntos::buscar(const Punto& dato)
{
  int i = 0;
  while ((i < sz)&& ! iguales(v[i], dato)) {
    ++i;
  }
  if (i == sz) {
    i = -1;
  }
  return i;
}
```

```
void ListaPuntos::abrir_hueco(int pos)
{
  assert(sz < int(v.size())) ;
  for (int i = sz; i > pos; --i) {
    v[i] = v[i-1];
  }
  ++sz;
}
void ListaPuntos::cerrar_hueco(int pos)
{
  assert(sz > 0) ;
  --sz;
  for (int i = pos; i < sz; ++i) {
    v[i] = v[i+1];
  }
}
}
```

Tema 2: TIPOS ABSTRACTOS DE DATOS