Programación 2
Diseño Modular

Fernando Orejas

1. Introducción a la asignatura.

2. Modularidad

Nombre:

- Fernando Orejas (orejas@cs.upc.edu)

Despacho:

- 238 (Edificio Omega)

Consultas:

previa cita vía Google Meet

Consultas por correo electrónico (preferible a slack):

Sí, si la respuesta es poco compleja y no puede ser obtenida fácilmente por otros medios

Evaluación

- Control de Laboratorio (10%)
- Práctica (25%)
- Examen de la práctica (15%)
- 2 Exámenes parciales (25% cada uno)

Objetivo de la asignatura:

Aprender a programar

Objetivo de la asignatura:

Aprender a programar ... un poco más.

Objetivos concretos:

Aprender a construir programas (algo) grandes.

Aprender a asegurarnos de que nuestros programas son correctos

Programa:

1. Diseño modular
2. Estructuras de datos lineales
3. Árboles
4. Diseño iterativo: verificación y derivación
5. Diseño recursivo
6. Mejoras en la eficiencia
7. Tipos recursivos de datos

Diseño modular

Qué cualidades ha de tener un buen programa?

1. Corrección

2. Legibilidad.

3. Eficiencia.

Cómo construir programas grandes?

Dos principios básicos:

- Descomposición en *partes* (módulos)
- Especificación

Pero, ¿cómo?

Dos principios básicos:

- Descomposición en *partes* (módulos)
- Especificación
- Usando abstracción

En qué consiste la abstracción:

- *Olvidar* detalles
- Identificar cada parte con un *concepto* conocido.

Una (buena) descomposición modular

- Hace más comprensibles los programas
- Facilita el diseño y la corrección
- Facilita el análisis de la corrección, eficiencia,…
- Facilita la modificación posterior
- Incrementa la reutilización de software
- Facilita el trabajo en equipo

Descomposición modular

- Módulos con cohesión interna fuerte
  - Más fáciles de especificar
- Módulos con acoplamiento débil
  - independientes

Descomposición:

Módulos basados en abstracciones:

- Abstracciones funcionales: describen e implementan nuevas operaciones
- Abstracciones de datos: describen e implementan nuevos tipos de datos, incluyendo estructuras de datos
- En Pro1 la abstracción de datos es la definición de tipos. 👎👎

Contar palabras

Se desea diseñar un programa que lea un texto acabado en un punto y nos devuelva la lista de palabras que hay en el texto, ordenada por orden alfabético, indicando el número de veces que aparece cada palabra. Se supone que, como máximo, habrá 10.000 palabras distintas.

Contar palabras

Por ejemplo, dado el texto:

```
Mi mama me mima, mi mama me ama, ¿me ama
mi mama?.
```

La respuesta debería ser:

```
ama 2
mama 3
me 3
mi 3
mima 1
```

```
struct num_palabra{
    string p;
    int n; // numero de veces que aparece p
}

const int max_pals = 10000;

struct Lista_pal{
    vector <num_palabra> v;
    int sl; // 0 <= sl <= 10000

}
```

```
int main() {
    Lista_pal L;
    L.v = vector <num_palabra>(max_pals);
    L.sl = 0;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") guardar_palabra(L,S);
    }
    escribir(L)
}
```

```
// Pre:  --
// Post: lee la siguiente palabra del texto y devuelve un
//   booleano que nos dice si se ha acabado el texto
bool leer_palabra(string & S) ;

// Pre:  --
// Post: guarda la palabra en el vector L, si ya estaba
// incrementa su numero de apariciones, si no, la añade,
// indicando que ha aparecido una vez.
void guardar_palabra(Lista_pal& L, string & S) ;

// Pre:  --
// Post: Se han escrito las palabras de L
void escribir (const Lista_pal& L);
```

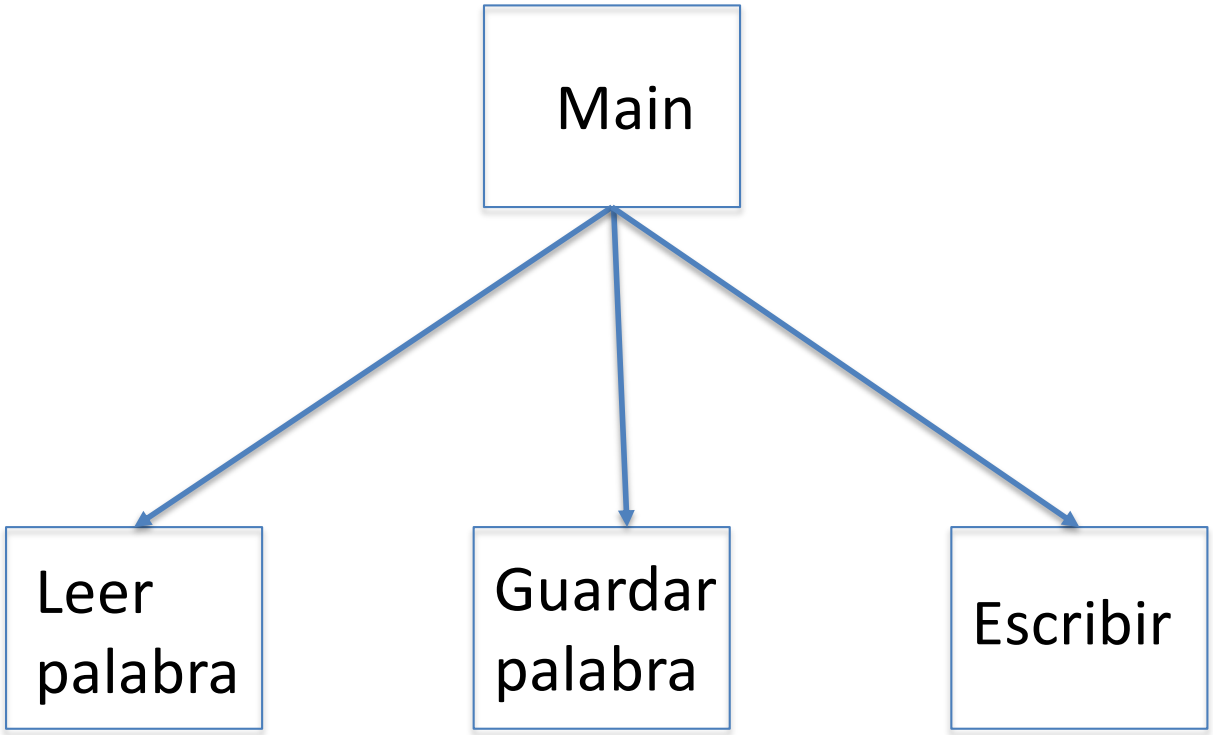
```
bool leer_palabra(string & S){
    bool fin = false;
    bool ini_pal = false;
    bool fin_pal = false;
    string S=""; char c;
    while ((not fin) and (not fin_pal) and (cin>>c)){
        if (c == '.') fin = true;
        else if (not es_letra(c) and ini_pal)
            fin_pal = true;
        else if (es_letra(c)){
            S.push_back(minusc(c)); ini_pal = true;
        }
    }
    return fin;
}
```

```
void guardar_palabra(Lista_pal& L,
                     string & S) {
   int i = 0;
   while (i < L.sl) {
      if (S == L.v[i].p) {
         ++L.v[i].n;
         return;
      }
      ++i;
   }
   L.v[L.sl] = {S,1};
   ++L.sl;
}
```

```
bool comp(num_palabra np1, num_palabra np2) {
    return np1.S < np2.S;
}

void escribir(const Lista_pal& L) {
    sort(L.v.begin(), L.v.begin()+L.sl, comp);
    for (int i = 0; i < L.sl; ++i) {
        cout << L.v[i].S << "  " << L.v[i].n << endl;
    }
}
```

Diagrama modular



Problemas:

- Acoplamiento fuerte
- Modificabilidad
- Poca estructura
- Cómo se distribuye el trabajo en un equipo
- Protección de la implementación

La alternativa:

Usar módulos que definen abstracciones de datos

Tipos Abstractos de Datos (TADs)

Un tipo se define dando:

- El nombre del tipo y una descripción de lo que es.
- Sus operaciones, incluida su descripción (qué hace, no cómo lo hace)
- Un tipo puede tener varias implementaciones. El tipo es su especificación, **no** su implementación.

La clase Lista_pal

```
class Lista_pal {
/* Implementa una estructura de datos que contiene las
palabras que han aparecido en una secuencia de palabras y
cuántas veces han aparecido */
   private:
     static const int max_pals = 10000;
     vector <num_palabra> v;
     int sl;
// 0 <= sl <= 10000
// Todas las posiciones de v[0:sl-1] están ocupadas con las
// palabras tratadas. El resto de v está libre.

     static bool comp(num_palabra np1, num_palabra np2);
// Nos dice si np11 es menor que np2
```

La clase Lista_pal

```
public:
    Lista_pal ();
    // Crea una lista de palabras vacía

    void guardar_palabra(string & S);
    // Guarda la palabra S en la lista de palabras

    void escribir();
    // Escribe la lista de palabras tal como se pide

}
```

```
int main() {
    Lista_pal L;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") L.guardar_palabra(S);
    }
    L.escribir()
}
```

La clase Lista_pal

```
public:
    // Constructora
    // Pre: --
    // Post: crea una lista de palabras vacía
    Lista_pal (){
        v = vector <num_palabra>(max_pals);
        sl = 0;
    }
```

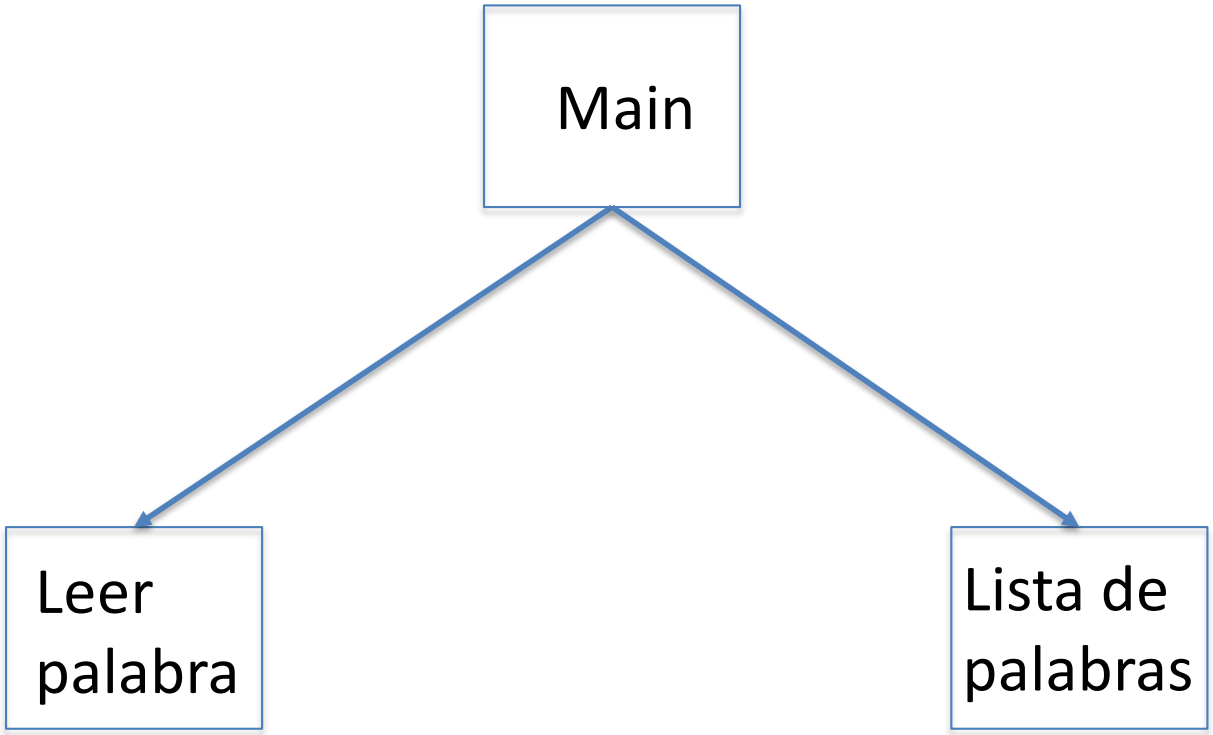
Clase Lista_pal

```
    // Modificadora
     void guardar_palabra(string & S) {
        int i = 0;
        while (i < sl) {
           if (S == v[i].p) {
              ++v[i].n;
              return;
           }
           ++i;
        }
        v[sl] = {S,1};
        ++sl;
}
```

Clase Lista_pal

```
    // Operación de escritura
     void escribir() {
        sort(v.begin(), v.begin()+sl, comp);
        for (int i = 0; i < sl; ++i) {
           cout << v[i].S << "  " << v[i].n << endl;
        }
     }
}
```

Diagrama modular



Sin problemas:

- Acoplamiento débil
- Podemos cambiar la implementación de las listas de palabras y el programa seguirá funcionando.
- La clase añade estructura al programa
- En un equipo, podríamos asignar la implementación de la estructura a una person, la lectura de palabras a otra y el programa principal a otra.
- La implementación está protegida.

TADs e independencia de los módulos

Los módulos que implementan TADs, están fuertemente cohesionados y su acoplamiento es débil.

Como consecuencia su uso es independiente de su implementación: Si cambiamos la implementación de un TAD, esto no afecta a. los programas que lo usan.

Programación orientada a objetos

POO = TADs + Herencia

Programación orientada a objetos

En los lenguajes orientados a objetos:

- Los módulos que definen TADs se llaman ***clases***
- Los elementos (constantes y variables) del tipo que define una clase se llaman ***objetos***
- Las operaciones del tipo definido por una clase se llaman ***métodos***
- los campos de una clase se llaman ***atributos***

Clases y Objetos

- Las clases tienen una parte privada y una pública
- Las clases pueden ser predefinidas o definidas por nosotros.
- En C++, los métodos son las operaciones (no estáticas) de una clase.

Métodos de una Clase

Las operaciones de una clase se dividen en

- Constructoras: son operaciones para construir los objetos básicos. Tienen el nombre de la clase y se llaman en la declaración de un objeto. Puede haber varias constructoras.

```
Lista_pal ();
```

- Destructoras: destruyen los objetos (los eliminan de la memoria)

```
~Lista_pal()
```

Métodos de una Clase

```
   Lista_pal (){
      v = vector <num_palabra>(max_pals);
      sl = 0;
   }
   Lista_pal (Lista_pal L){
      v = L.v;
      sl = L.sl;
   }
...
Lista_pal L1;
Lista_pal L2 (L3);
```

Métodos de una Clase

- Modificadoras: modifican el parámetro implícito
- Consultoras: suministran información contenida en el objeto.
- Entrada/Salida: Escriben información contenida en el objeto.

La clase Lista_pal

```
class Lista_pal {
/* Implementa una estructura de datos que contiene las
palabras que han aparecido en una secuencia de palabras y
cuántas veces han aparecido */
   private:
      static const int max_pals = 10000;
      vector <num_palabra> v;
      int sl;
// 0 <= sl <= 10000
// Todas las posiciones de v[0:sl-1] están ocupadas con las
// palabras tratadas. El resto de v está libre.
      static bool comp(num_palabra np1, num_palabra np2){
         return np1.S < np2.S;
      }
```

Clase Lista_pal

```
    // Modificadora
     void guardar_palabra(string & S) {
        int i = 0;
        while (i < sl) {
           if (S == v[i].p) {
              ++v[i].n;
              return;
           }
           ++i;
        }
        v[sl] = {S,1};
        ++sl;
}
```

Clase Lista_pal

```
    // Operación de entrada/salida
     void escribir() {
        sort(v.begin(), v.begin()+sl, comp);
        for (int i = 0; i < sl; ++i) {
           cout << v[i].S << "  " << v[i].n << endl;
        }
     }
}
```

Clases y Objetos en C++

- Cada objeto de una clase *posee* todos los atributos y métodos de la clase (salvo que sean estáticos). Los métodos tienen como *parámetro implícito* al objeto al que "pertenecen". Por ejemplo, escribimos:

```
void guardar_palabra(string & S)
```

en vez de

```
void guardar_palabra(Lista_pal& L, string & S)
```

y al usarla:

```
L.guardar_palabra(s)
```

en vez de

```
guardar_palabra(L,s)
```

- Las otras operaciones que trabajen sobre ese tipo, pero que no estén en la clase o sean estáticas no se consideran métodos.

```
int main() {
    Lista_pal L;
//    L.v = vector <num_palabra>(max_pals);
//    L.sl = 0;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") L.guardar_palabra(S);
    }
    L.escribir()
}
```

Clases y Objetos en C++

Dentro de una clase, si nos queremos referir a un atributo o a un método, lo hacemos directamente, sin cualificarlo. En cambio, si nos queremos referir a otro objeto de la misma o de otra clase, hemos de cualificarlo con el nombre del objeto:

```
class punto {
private:
float x, y, color;
public
...
bool mismo_color(punto p) const{
return color == p.color;
}
```

Clases y Objetos en C++

Si nos queremos referir al P.I. podemos usar **`this`**:

```
class C {
private:
...
public
...
void copia(C x) const {
   if (this != x){
      ...
   }
}
```

¡¡Ojo!! No es buen estilo:

```
bool mismo_color(punto p) const {
return this->color == p.color;
}
```

Diseño Orientado a Objetos

1. Identificamos las clases que juegan un papel en el programa

El propio enunciado es una buena fuente de información (abstracciones de datos). También el esquema de implementación que tengamos en la cabeza.

2. Especificamos dichas clases
3. Implementamos el programa principal en términos de las operaciones y objetos definidos por las clases.
4. Implementamos las clases especificadas.
5. Para implementar las clases puede ser necesario definir, especificar e implementar nuevas clases

...

Especificación

- La especificación es muy importante:

Sin especificación no sabemos qué hacen los módulos que usamos.

- La especificación es un contrato

Cualquier cambio en la implementación, si respeta la especificación no afecta al resto del programa

Especificación de una Clase

1. Describimos qué son los objetos de la clase
2. Para cada operación de la clase definimos su Pre y su Post

La clase Lista_pal

```
class Lista_pal {
/* Implementa una estructura de datos que contiene las
palabras que han aparecido en una secuencia de palabras y
cuántas veces han aparecido */
   private:
      static const int max_pals = 10000;
      vector <num_palabra> v;
      int sl;
// 0 <= sl <= 10000
// Todas las posiciones de v[0:sl-1] están ocupadas con las
// palabras tratadas. El resto de v está libre.
      static bool comp(num_palabra np1, num_palabra np2){
         return np1.S < np2.S;
      }
```

La clase Lista_pal

```
public:
    // Constructora
    // Pre: --
    // Post: crea una lista de palabras vacía
    Lista_pal (){
       v = vector <num_palabra>(max_pals);
       sl = 0;
    }
```

Implementación

- La implementación debe de satisfacer la especificación
- Cualquier implementación que satisfaga la especificación no ha de afectar al resto del programa.

Implementación de una Clase

1. Elegimos una representación para los objetos
2. Describimos su invariante
3. Implementamos sus operaciones
4. Si hace falta, utilizamos funciones auxiliares (privadas)

Ficheros de una Clase

Es conveniente separar en ficheros diferentes la *especificación* de la implementación de una clase:

- .hh : Contiene la especificación de la clase, aunque también las cabeceras y atributos de la parte privada.
- .cc : Contiene la implementación de las operaciones de la clase

Especificación de una clase (fichero .hh)

```
class Lista_pal {
/* Implementa una estructura de datos que contiene las
palabras que han aparecido en una secuencia de palabras y
cuántas veces han aparecido */
   private:
     static const int max_pals = 10000;
     vector <num_palabra> v;
     int sl;
// 0 <= sl <= 10000
// Todas las posiciones de v[0:sl-1] están ocupadas con las
// palabras tratadas. El resto de v está libre.
     static bool comp(num_palabra np1, num_palabra np2){
        return np1.S < np2.S;
     }
```

Especificación de una clase (fichero .hh)

```
    public:
    // Pre: --
    // Post: crea una lista de palabras vacía
    Lista_pal ();

    // Pre: --
    // Post: Añade la aparición de una palabra a la lista
     void guardar_palabra(string & S);

    // Pre: --
    // Post: Escribe por orden alfabético las palabras de la
    //lista junto con el número de veces que han aparecido
     void escribir();
}
```

En la implementación de una Clase

- En la cabecera de las operaciones, cualificamos sus nombres con el nombre de la clase. Por ejemplo:

```
Lista_pal::guardar_palabra
```

- Nos referimos a los atributos de la clase directamente. Por ejemplo:

```
v[sl]
```

Implementación de una clase (fichero .cc)

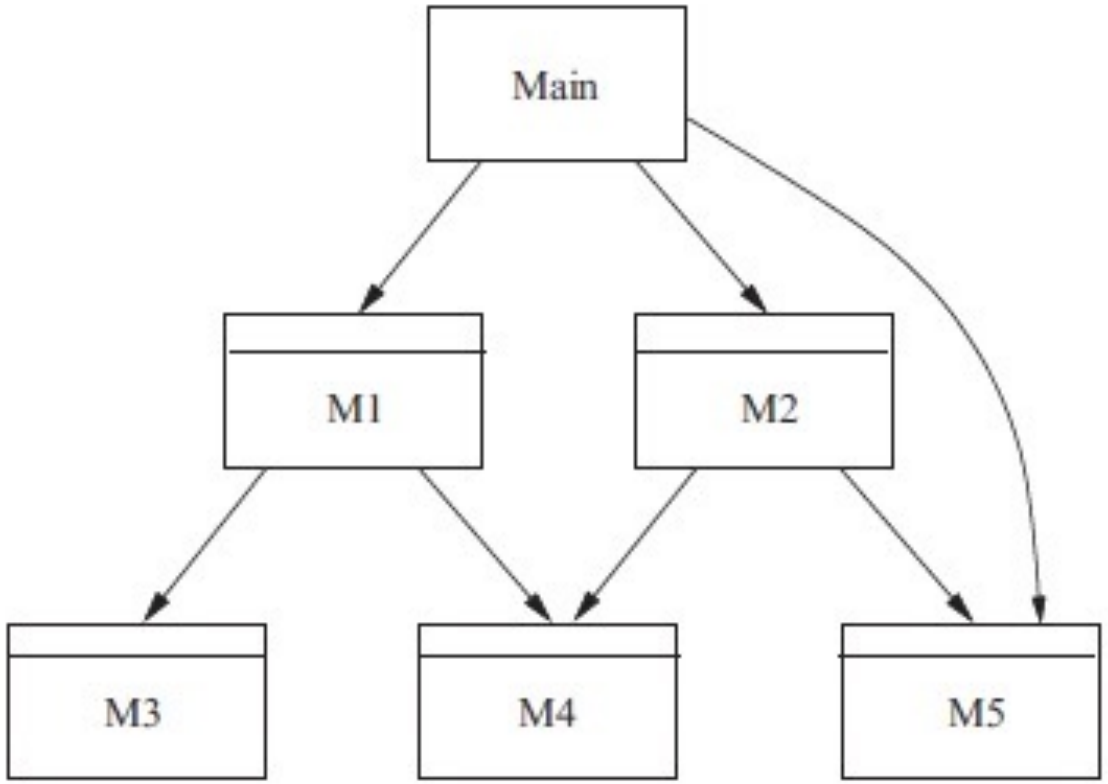
```
#include "Lista_pal.hh"
void Lista_pal::guardar_palabra(string & S) {
   int i = 0;
   while (i < sl) {
      if (S == v[i].p) {
         ++v[i].n;
         return;
      }
      ++i;
   }
   v[sl] = {S,1};
   ++sl;
}
```

Implementación de una clase (fichero .cc)

```
Lista_pal::Lista_pal (){
   v = vector <num_palabra>(max_pals);
   sl = 0;
}

void Lista_pal::escribir() {
    sort(v.begin(), v.begin()+sl, comp);
    for (int i = 0; i < sl; ++i) {
       cout << v[i].S << "  " << v[i].n << endl;
    }
}
```

Diagramas modulares



Relaciones entre módulos

Programa = conjunto de módulos relacionados/dependientes

Un módulo puede:

- Definir un nuevo tipo de datos
- Enriquecer o ampliar tipos con nuevas operaciones (módulos funcionales)

Las relaciones de uso pueden ser:

- Visibles (en la especificación)
- Ocultas para una implementación concreta

Ampliación de tipos de datos

Si queremos añadir nuevas operaciones a un tipo de datos, podemos hacer tres cosas:

- Definir las nuevas operaciones fuera de la clase
- Añadir los nuevos métodos a la clase.

  Si necesitamos acceder a la parte privada de la clase
- Usar el mecanismo de herencia (no en P2)

Solución 1

- No se modifica ni la especificación, ni la implementación de la clase.
- Se puede hacer en un módulo nuevo o en una clase que la utilice.
- Sin parámetro implícito
- Puede ser ineficiente

Solución 2

- Hay que poder modificar la clase y entender cómo está implementada.
- Hay que añadir las cabeceras (o el código) en los ficheros de la clase.
- Puede ser una solución más eficiente
- Adicionalmente, se puede cambiar la implementación de la clase, lo que implicará modificar las operaciones

Genericidad

Una clase genérica es una clase que tiene un *tipo parámetro*. En C++, a las clases genéricas se les llama templates:

```
template <class T> class Lista {
private:
    vector <T> v;
    int sl;
....
```

Después podemos usar esta declaración para crear listas de diferentes tipos:

```
Lista <int> L1;
Lista <string> L2;
```

Bibliotecas

En C++ tenemos una biblioteca standard de clases, la Standard C++ Library (std) y una biblioteca standard de templates (STL).

La STL contiene una serie de templates standard como:

- vector
- stack
- queue
- list

...