Diferencia entre clases y structs

```
struct punto{
     double x,y;
}
```

Diferencia entre clases y structs

```
class punto {
private:
    double x, y;

public

punto (){
    x = 0;
    y = 0;
}

double coord_x (){
    return x;

}
```

Diferencia entre clases y structs

```
double coord_y (){
    return y;
}

void mover_x (double z){
    x = x+z;
}

void mover_y (double z){
    y = y+z;
}


...
}
```

Diferencia entre clases y structs

```
// Si punto es una struct

void mover_xy (punto &p, double z1, double z2){
    x = x+z1;
    y = y+z2;
}

// Si punto es una clase

void mover_xy (punto &p, double z1, double z2){
    p.mover_x(z_1);
    p.mover_y(z_2);
}
```

Diferencia entre clases y structs

```
// Coordenadas polares
class punto {
private:
    double x, y;
public

punto (){
    x = 0;
    y = 0;
}

double coord_x (){
    return x*cos(y);

}
```

Diferencia entre clases y structs

```
double coord_y (){
    return x*sin(y);
}

void mover_x (double z){
    ...
}

void mover_y (double z){
    ...
}


...
}
```

Especificación y uso de las clases

La clase Estudiante

```
Class Estudiante{
/* Define el TAD estudiante caracterizado por un DNI y una
nota (opcional*/

private:
// Cómo representamos los objetos de la clase

public:
// Constructoras

Estudiante ();
// Pre: Cierto
// Post: El resultado es un estudiante sin nota con DNI = 0

Estudiante (int dni);
// Pre: Cierto
// Post: El resultado es un estudiante sin nota con DNI = dni
```

Operaciones creadoras o constructoras

Son operaciones para construir objetos nuevos: y se llaman en la declaración de un objeto.

- Tienen el nombre de la clase
- Crean un objeto nuevo.
- Puede haber varias constructoras. Se distinguen por sus parámetros
- La ***constructora por defecto*** (sin parámetros) crea un objeto nuevo, dando valores por defecto a sus atributos.
- Se llaman al declarar un objeto:

```
Estudiante est1;  Estudiante est2(12345678);
```

La clase Estudiante

```
// Modificadoras

void poner_nota (double n);
// Pre: El estudiante no tiene nota y n es una nota válida.
// Post: El estudiante pasa a tener nota n

void cambiar_nota (double n);
// Pre: El estudiante tiene un nota que es válida.
// Post: El estudiante pasa a tener nota n
```

La clase Estudiante

```
// Consultoras

int consultar_DNI() const;
// Pre: Cierto
// Post: Retorna el DNI del estudiante

bool tiene_nota() const;
// Pre: Cierto
// Post: Retorna si el estudiante tiene nota

double consultar_nota() const;
// Pre: El estudiante tiene nota
// Post: Retorna la nota del estudiante

// Operación de la clase

static double nota_maxima() const;
// Pre: Cierto
// Post: Retorna la nota máxima que puede tener un estudiante
```

Métodos de una Clase

Las operaciones de una clase se dividen en

- Constructoras.
- Destructoras: destruyen los objetos (los eliminan de la memoria)
- Modificadoras: modifican el parámetro implícito
- Consultoras: suministran información contenida en el objeto.
- Entrada/Salida: Escriben información contenida en el objeto.

Los métodos de clase

Se declaran al definir el método como **`static`**.

- Pertenecen a la clase y no a los objetos
- No tienen parámetros implícitos.

Por ejemplo:

```
cin >> nota;
if (nota >= 0 && nota <= Estudiante::nota_maxima())
  e.cambiar_nota(nota);
else
  cout << "la nota introducida no es valida" << 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 poner_nota (double n);

bool tiene_nota() const;
```

en vez de

```
void poner_nota (Estudiante &e, double n);

void tiene_nota(const Estudiante &e);
```

Clases y Objetos en C++

- Al usar los métodos:

```
e.poner_nota(7.5);
bool b = e.tiene_nota();
```

en vez de

```
poner_nota(e, 7.5);
bool b = tiene_nota(e);
```

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

Clases y Objetos en C++

Dentro de la clase, los atributos y métodos del parámetro implícito no se cualifican, los de otros objetos, sí.

```
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;
}
```

Ejemplo de uso de la clase Estudiante

```
bool cambia_NP(vector <Estudiant>. & v, int dni);
// Pre: Todos los estudiantes de v tienen DNI diferente.

/* Post: Si el estudiante de v, con DNI = dni, no tiene nota,
pasa a tener 0 y el resto no cambia. La respuesta es true si
se ha podido hacer la modificación, es false si no hay un
estudiante con ese DNI*/
```

Ejemplo de uso de la clase Estudiante

```
bool cambia_NP(vector <Estudiant> & v, int dni){
  int i = 0;
  while (i < v.size()) {
     if (v[i].consultar_DNI() == dni){
        if (not v[i].tiene_nota())
           v[i].poner_nota(0);
        return true;
     }
     ++i;
  }
  return false;
}
```

La clase Conjunto de Estudiantes

```
#include "Estudiante.hh"
Class Cjt_estudiantes{
/* Define el TAD conjunto de estudiantes. Los conjuntos
tienen definido un tamaño máximo */

private:
// Cómo representamos los objetos de la clase

public:
// Constructoras

Cjt_estudiantes ();
// Pre: Cierto
// Post: El resultado es un conjunto vacío de estudiantes
```

La clase Conjunto de Estudiantes

```
// Modificadoras

void incluir_estudiante(const Estudiante &e);
/* Pre: En el conjunto no hay otro estudiante con el mismo
DNI. El tamaño del conjunto es menor que el máximo */
// Post: Se ha añadido el estudiante e al conjunto


void modificar_estudiante(const Estudiante &e);
/* Pre: En el conjunto hay otro estudiante con el mismo DNI
que e */
/* Post: Se ha cambiado en el conjunto el estudiante con el
mismo DNI por e */
```

La clase Conjunto de Estudiantes

```
void modificar_i_esimo(int i, const Estudiante &e);
/* Pre: 1 ≤ i ≤ el tamaño del conjunto */
/* Post: En el conjunto, se ha substituido por e, el
estudiante que ocupa el i-ésimo puesto en orden creciente de
DNIs */


// Consultoras

int mida() const;
// Pre: Cierto
/* Post: Retorna el tamaño del conjunto */

bool existe_estudiante(int dni) const;
// Pre: dni > 0
/* Post: Retorna si hay un estudiante con ese DNI en el
conjunto */
```

La clase Conjunto de Estudiantes

```
Estudiante consultar_i_esimo(int i) const;
// Pre: 1 ≤ i ≤ el tamaño del conjunto
/* Post: Retorna el el estudiante i-ésimo en orden creciente
de DNIs */

// Operación de la clase

static int mida_maxima () const;
// Pre: Cierto
// Post: Retorna el tamaño máximo que puede tener un conjunto
```

La clase Conjunto de Estudiantes

```
// Operaciones de entrada/salida

void leer() const;
// Pre: Cierto
/* Post: El parámetro implícito pasa a contener todos los
estudiantes que aparecen en el canal de entrada */


void escribir() const;
// Pre: Cierto
/* Post: Se han escrito, por orden ascendente de DNIs, los
estudiantes que hay en el conjunto */
```

Redondear notas

```
// Ejemplo de uso de Cjt_estudiantes

void redondear_notas(cjt_estudiantes &c);
// Pre: cierto
/* Post: Se han resondeado las notas de todos los estudiantes d
c que tienen nota */

// función auxiliar

void redondear (Estudiante &e){
// Pre: e tiene nota
// Post: e pasa a tener su nota original redondeada
   e.modificar_mota( (int)est.consultar_nota()+0.5)
}
```

Redondear notas

```
// Ejemplo de uso de Cjt_estudiantes

void redondear_notas(cjt_estudiantes &c){

   for (int i = 1; i <= c.mida(); ++mida){
      Estudiante e;
       e = c.consultar_i_esimo(i);
       if (e.tiene_nota()
          e.redondear_nota();
          c.modificar_i_esimo(i,e);
       }
   }
}
```

Implementación de las clases

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 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

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)

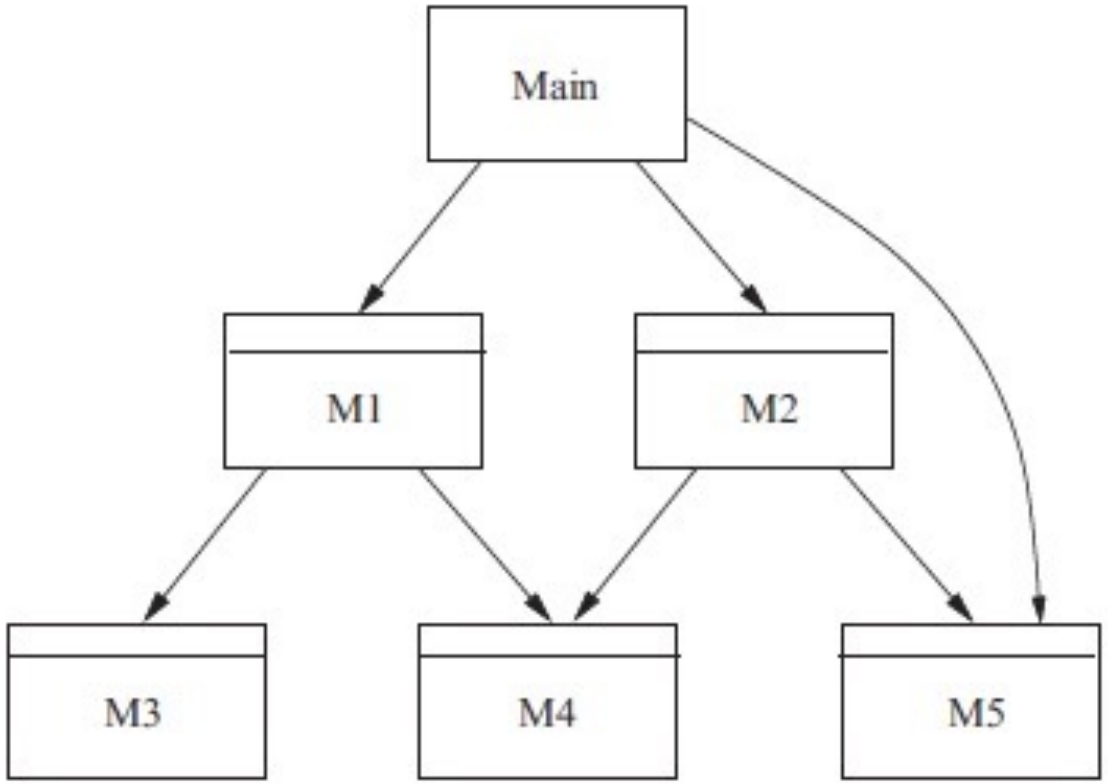
```
#include "Lista_pal.hh"
void Lista_pal::guardar_palabra(string & S) {
   int i = 0;
   while (i < sl) {
      if (S == v[i].p) {
         ++v[i].n;
         return;
      }
      ++i;
   }
   v[sl] = {S,1};
   ++sl;
}
```

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

Extensión de una clase

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

...