Programación 2

Fernando Orejas

1. Introducción a la asignatura.

2. Modularidad

Nombre:

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

Despacho:

- 238 (Edificio Omega)

Consultas:

- Despues de las clases en mi despacho
- previa cita vía en mi despacho o con Google Meet
- por correo electrónico, si la respuesta es poco compleja

Evaluación

- Práctica (35%)
- Examen de la práctica (15%)
- 1er examen parcial (20%)
- 2º examen parcial (30%)

Fechas clave

- 24/4 Primer parcial
- 06/6. Examen de la práctica
- 16/6 Segundo parcial

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 recursivo
5. Diseño iterativo: verificación y derivación
6. Mejoras en la eficiencia
7. Tipos recursivos de datos

Diseño modular

1. Abstracción y especificación: diseño modular
2. Diseño basado en objetos
3. Especificación y uso de las clases
4. Implementación de las clases
5. Extensión de una clase

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

Buena descomposición modular

- Módulos con cohesión interna fuerte
  - Los módulos tienen significado por si mismos e interactuan con otros módulos de forma simple

- Módulos con acoplamiento débil
  - independientes: Los cambios en un módulo no afectan al resto de los módulos

Descomposición:

Módulos basados en abstracciones:

- Abstracciones funcionales: uso de funciones (auxiliares)
- Especificaciones pre/post

Especificación vs Implementación:

Regla básica: Un cambio de implementación de una función que respete su especificación, no puede modificar la corrección del programa.

- Especificación: contrato entre implementador y el usuario
- Especificación: abstracción de la implementación

Contar palabras

Se desea diseñar un programa que lea un texto 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
```

```
int main() {
    struct num_palabra{
        string p;
        int n; // numero de veces que ha aparecido p
    };
    vector <num_palabra> v;
    string S;
    bool fin = false;
    while (not fin) {
        bool ini_pal = false;
        bool fin_pal = false;
        string S="";
        char c;
```

```
while ((not fin_pal) and (cin>>c)){
   if (not es_letra(c) and ini_pal)
      fin_pal = true;
   else if (es_letra(c)){
      S.push_back(minusc(c)); ini_pal = true;
   }
}
if (S == "") fin = true;
else {
```

```
        int i = 0;
        bool finadd = false;
        while (i < v.size() && (not fin_add)) {
          if (S == v[i].p) {
             ++v[i].n; finadd = true;
          }
          ++i;
        }
        num_palabra np;
        np.p = S; np.n = 1;
        v.push_back(v,np);
    }
}
```

```
    sort(v.begin(), v.end(), comp);
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i].p << "  " << v[i].n << endl;
    }
}

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

Problemas:

- Difícil de entender
- Probabilidad alta de errores
- Modificabilidad difícil

```
int main() {
    struct num_palabra{
        string p;
        int n; // numero de veces que ha aparecido p
    };
    vector <num_palabra> v;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") guardar_palabra(v,S);
    }
    escribir(v)
}
```

```
// 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 v, si ya estaba
// incrementa su numero de apariciones, si no, la añade,
// indicando que ha aparecido una vez.
void guardar_palabra(vector <num_palabra>& v, string & S) ;

// Pre:  --
// Post: Se han escrito las palabras de v
void escribir (const vector <num_palabra>& v);
```

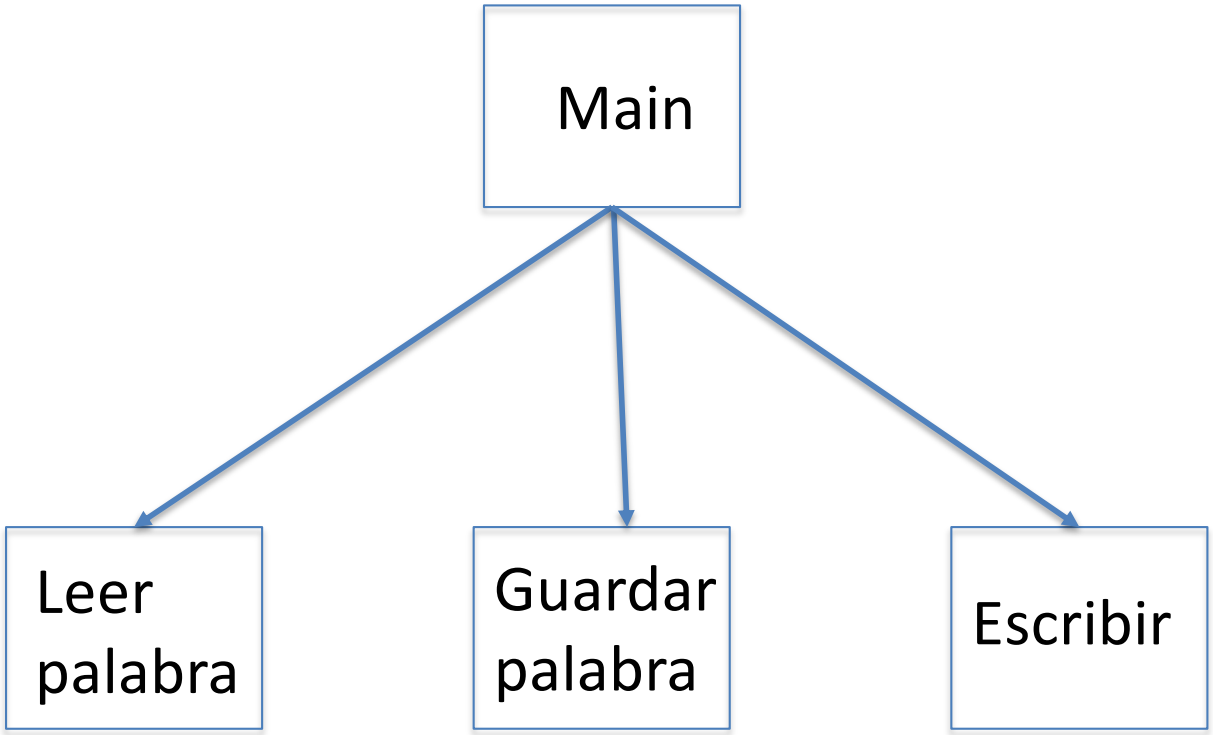
```
bool leer_palabra(string & S){
    bool ini_pal = false;
    bool fin_pal = false;
    string S=""; char c;
    while ((not fin_pal) and (cin>>c)){
        if (not es_letra(c) and ini_pal)
            fin_pal = true;
        else if (es_letra(c)){
            S.push_back(minusc(c)); ini_pal = true;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

```
void guardar_palabra(vector <num_palabra>& v,
                     string & S) {
   int i = 0;
   while (i < v.size()) {
      if (S == v[i].p) {
         ++v[i].n; return;
      }
      ++i;
   }
    num_palabra np;
    np.p = S; np.n = 1;
    v.push_back(v,np);
}
```

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

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

Diagrama modular



Problemas:

- Mala modificabilidad
- Protección de la implementación

Diseño basado en objetos

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. 👎👎

La alternativa:

Usar módulos que definen abstracciones de datos (***Tipos abstractos 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 y su frecuencia */
private:
    vector <num_palabra> v;
// Las posiciones de v contienen las palabras tratadas.
public:
...
void guardar_palabra(string & S);
// Pre: Cierto
// Post: Si S está en la lista de palabras suma 1 a su
// frecuencia, si no la añade con frecuencia 1
void escribir();
// Pre: Cierto
// Post: Escribe la lista de palabras tal como se pide
}
```

Clase Lista_pal

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

Clase Lista_pal

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

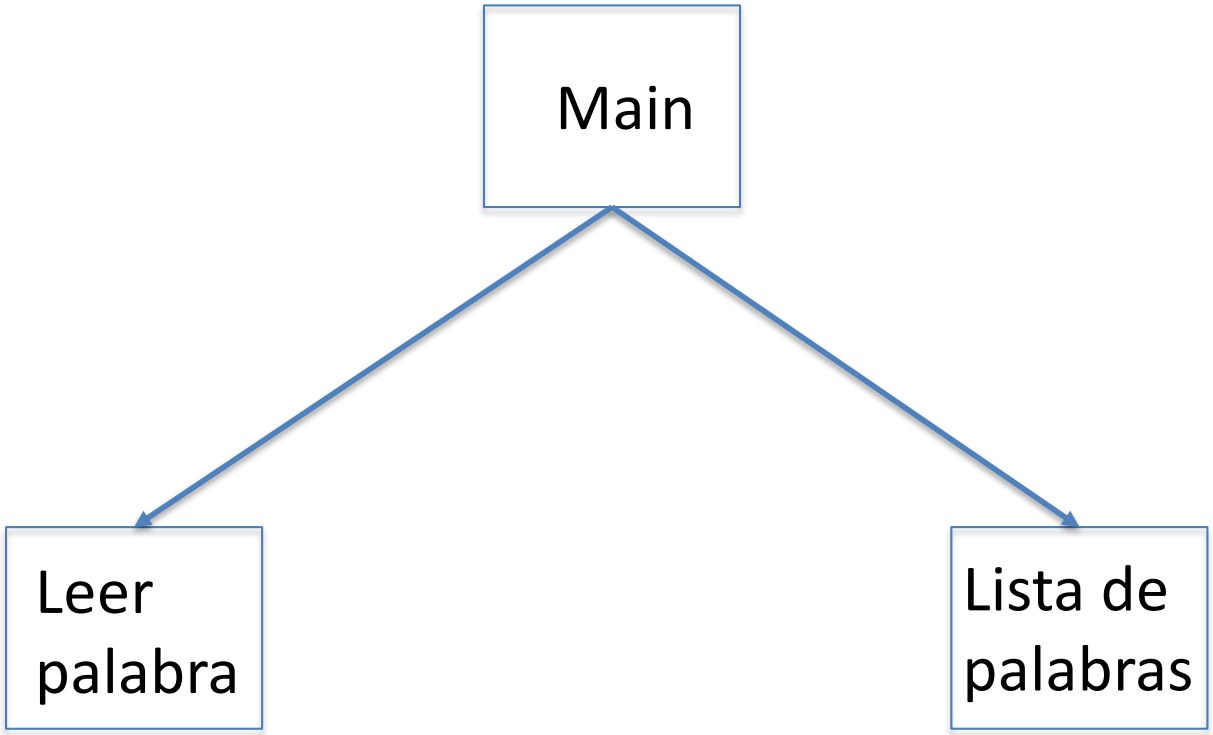
La clase num_palabra

```
class num_palabra{
  string p;
  int n; // numero de veces que ha aparecido p

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

```
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()
}
```

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.

TADs e independencia de los módulos

Fase de especificación: Se deciden las operaciones de los TADs y sus especificaciones

Fase de implementación: Se decide una representación y se implementan las operaciones

Programación orientada a objetos

POO = TADs + Herencia

Clases y 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.
- Cada objeto es ***propietario*** de sus atributos y sus métodos
- Cada método tiene un ***parámetro implícito***: su propietario sobre el que se aplica la operación.
- Podemos tener otras funciones que operen sobre el tipo, pero si no están en la clase, no son métodos

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 <= Estudiant::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_studiantes{
/* Define el TAD conjunto de estudiantes. Los conjuntos
tienen definido un tamaño máximo */

private:
// Cómo representamos los objetos de la clase

public:
// Constructoras

Cjt_studiantes ();
// 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 */
```