

## 8

## Programación modular

Grados en Ingeniería Informática, Ingeniería del Software e Ingeniería de Computadores

Miguel Gómez-Zamalloa Gil  
(adaptadas del original de Luis Hernández Yáñez y Ana Gil)

Facultad de Informática  
Universidad Complutense



# Índice

---

Programas multiarchivo y compilación separada	2
Interfaz frente a implementación	7
Uso de módulos de biblioteca	13
Ejemplo: Gestión de una lista de datos ordenada I	14
Compilación de programas multiarchivo	22
El preprocesador	24
Cada cosa en su módulo	26
Ejemplo: Gestión de una lista de datos ordenada II	27
El problema de las inclusiones múltiples	33
Compilación condicional	38
Protección frente a inclusiones múltiples	39
Ejemplo: Gestión de una lista de datos ordenada III	40
Espacios de nombres	48
Implementaciones alternativas	56
Calidad y reutilización del software	64



# Fundamentos de la programación

---

## Programas multiarchivo y compilación separada



# Programación modular

## Programas multiarchivo

El código fuente del programa se reparte entre varios archivos (*módulos*), cada uno con las declaraciones y los subprogramas que tienen relación.

→ Módulos: archivos de código fuente con declaraciones y subprogramas de una unidad funcional: una estructura de datos, un conjunto de utilidades, ...

### Lista

```
const int TM = ...;
typedef double tTupla[TM];
typedef struct {
    tTupla elem;
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(tLista);

void sort(tLista&);
```

### Principal

```
int main() {
    tLista lista;
    init(lista);
    cargar(lista, "datos.txt");
    sort(lista);
    double dato;
    cout << "Dato: ";
    cin >> dato;
    insert(lista, dato);
    cout << min(lista) << endl;
    cout << max(lista) << endl;
    cout << sum(lista) << endl;
    guardar(lista, "datos.txt");

    return 0;
}
```

### Cálculos

```
double mean(tLista);

double min(tLista);

double max(tLista);

double desv(tLista);

int minIndex(tLista);

int maxIndex(tLista);

double sum(tLista);
```

### Archivos

```
bool cargar(tLista&, string);

bool guardar(tLista, string);

bool mezclar(string, string);

int size(string);

bool exportar(string);
```

Ejecutable



# Programación modular

## Compilación separada

Cada módulo se compila a código objeto de forma independiente:

### Lista

```
const int TM = ...;
typedef double tTupla[TM];
typedef struct {
    tTupla elem;
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(tLista);

void sort(tLista&);
```

### Lista.obj

```
001011101011001010010010101
0010101001010101111010101000
101001010101010010101010101
011001010101010101010101001
01010101010000010101010101
0100101010101010100010101011
110010101010111100110010101
011010101010100100101010111
00101010101010101010101010
101010101010010100010011110
10010101010010101001010100
101010101010010100101010101
010000101010111001010100100
01110101011101001010100101
01011111101010100101010111
0000100101001010101010110
```

### Cálculos

```
double mean(tLista);

double min(tLista);

double max(tLista);

double desv(tLista);

int minIndex(tLista);

int maxIndex(tLista);

double sum(tLista);

...
```

### Calculos.obj

```
010110010100100101001010100
101010111101010100010101010
101010100101010101010101010
101010101010101010101010101
1010000010101010101010010101
010101010000101011110010101
0101011110011001010101010101
0101001001010001111001010101
010010101010100101010101010
1010010100001001111010101010
1100101010101010010101010101
0101001010101010101010000101
0101110010101010001110101010
11101001101010101010111111
10101011001101010111000010010
101001010101010110001111010
```

### Archivos

```
bool cargar(tLista&, string);

bool guardar(tLista, string);

bool mezclar(string, string);

int size(string);

bool exportar(string);

...
```

### Archivos.obj

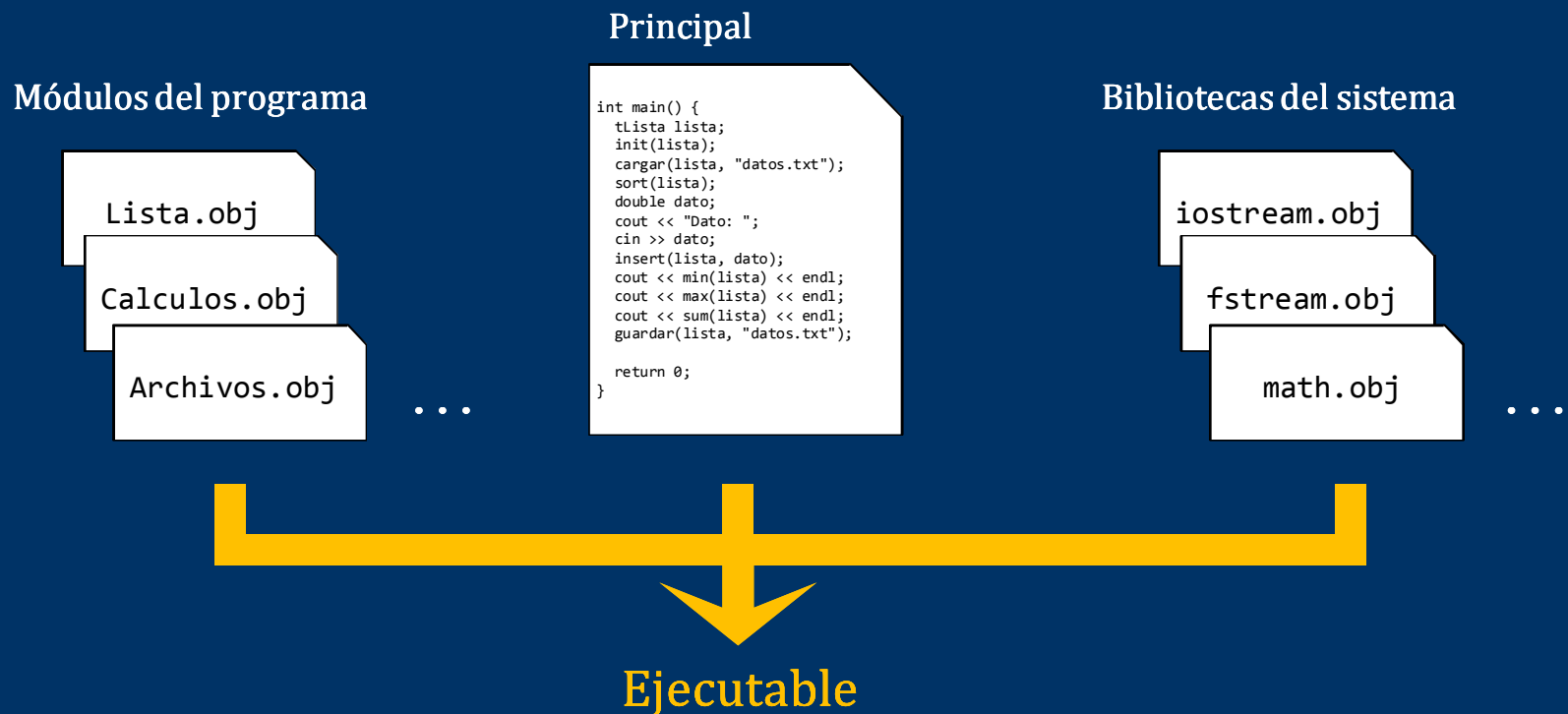
```
1110101011001001001010101010
10100101010111110101010001010
010101010100010101010101010
0101010101010101010101010101
0101010100000101010101010100
101010101010000101010111100
101010101111001100101010110
1010101010010101010111001010
1010100101001010101010101010
010101010001000100111101001
0101011001010100101010101010
101010100101010010101010100
00101010110010100101000111
01010111010011010101001010101
111110101011001101010110000
100101010010101010101101111
```



# Programación modular

## Compilación separada

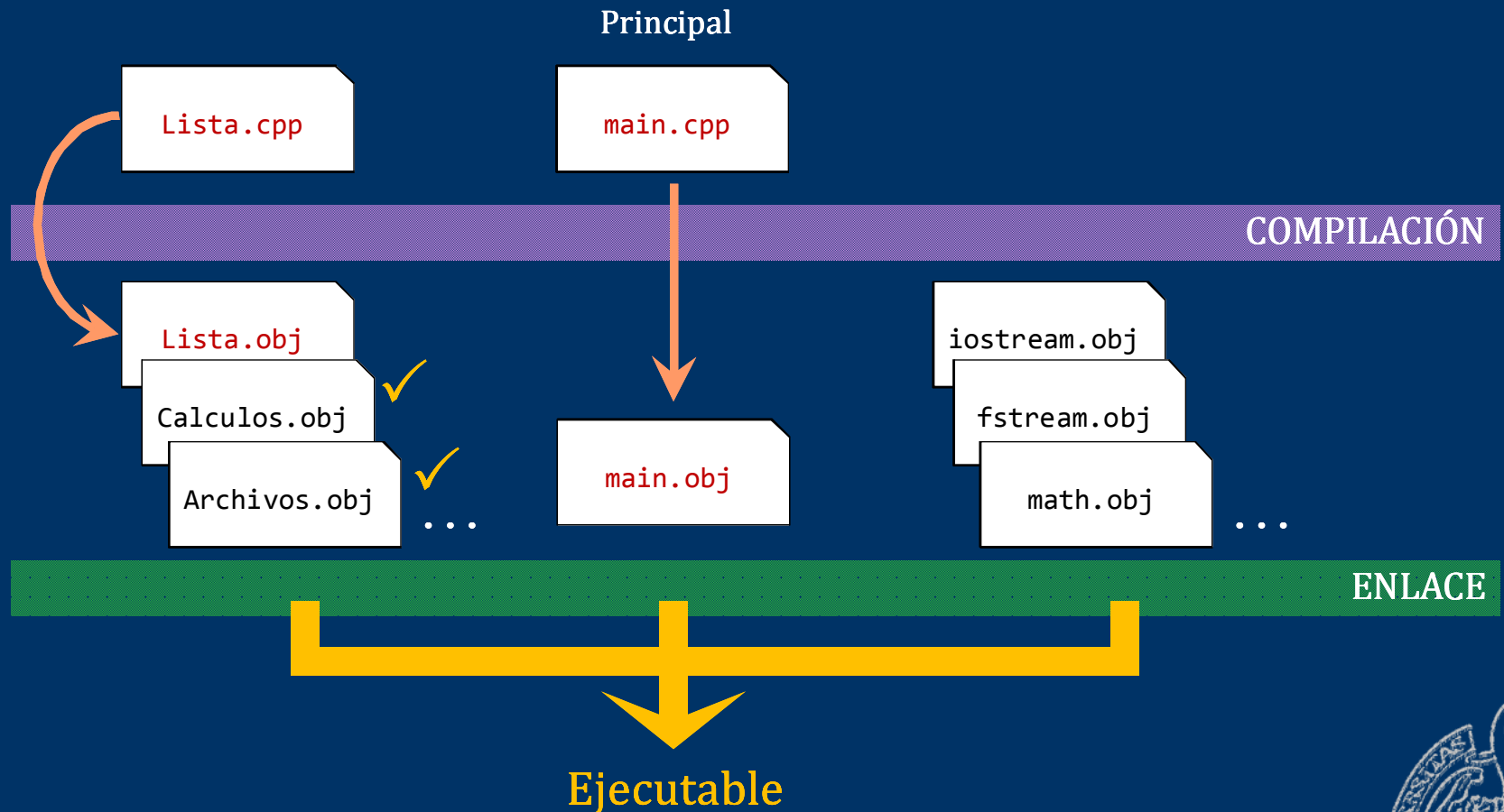
Al generar el ejecutable (del programa principal), se enlazan los módulos compilados:



# Programación modular

## Compilación separada

*¡Sólo los archivos de código fuente modificados necesitan ser recompilados!*



# Fundamentos de la programación

---

## Interfaz frente a implementación





# Interfaz frente a implementación

## *Creación de módulos de biblioteca*

En el código de un programa de un único archivo tenemos:

- ✓ Definiciones de constantes.
- ✓ Declaraciones de tipos de datos.
- ✓ Prototipos de los subprogramas.
- ✓ Implementación de los subprogramas.
- ✓ Implementación de la función `main()`.

Las constantes, tipos y prototipos de subprogramas que tienen que ver con alguna unidad funcional indican *cómo se usa* ésta: **interfaz**.

- ✓ Estructura de datos con los subprogramas que la gestionan.
- ✓ Conjunto de utilidades (subprogramas) de uso general.
- ✓ Etcétera.

La implementación de los subprogramas es eso, **implementación**.



# Interfaz frente a implementación

## *Creación de módulos de biblioteca*

**Interfaz:** Declaraciones de datos y subprogramas (prototipos) documentados.

¡Es todo lo que el usuario de esa unidad funcional necesita saber!

**Implementación:** Código de los subprogramas de la interfaz.

No necesita conocerse para utilizarlo: ¡Se da por sentado que es correcto!

Pueden contener declaraciones adicionales.

Separamos la interfaz y la implementación en dos archivos:

Archivos de cabecera: extensión .h

Archivos de implementación: extensión .cpp

} Mismo nombre (x.h / x.cpp)

- ✓ Archivo de cabecera: Declaraciones de datos y subprogramas (prototipos).  
Se corresponde con la **Interfaz** del módulo (unidad funcional).
- ✓ Archivo de implementación: Implementación de los subprogramas.  
Se corresponde con la **Implementación** de la interfaz del módulo.



# Interfaz frente a implementación

## Creación de módulos de biblioteca

### Interfaz frente a implementación

#### Lista.h

```
const int TM = ...;
typedef double tTupla[TM];
typedef struct {
    tTupla elem;
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(tLista);

void sort(tLista&);
```

#### Lista.cpp

```
#include "Lista.h"

void init(tLista& lista) {
    lista.cont = 0;
}

bool insert(tLista& lista,
double valor) {
    if (lista.cont == TM)
        return false;
    else {
        lista.elem[lista.cont] =
        valor;
        lista.cont++;
    }
}
```

Módulo  
Unidad  
Biblioteca

Si otro módulo (o el programa principal) quiere usar algo de esa biblioteca:  
Deberá incluir el archivo de cabecera.

#### main.cpp

```
#include "Lista.h"
...
```

Los nombres de archivos de cabecera  
propios (no del sistema) se encierran  
entre dobles comillas, no entre ángulos.



# Interfaz frente a implementación

## Creación de módulos de biblioteca

### Interfaz

El archivo de cabecera (.h) tiene todo lo que necesita conocer cualquier otro módulo (o programa principal) que quiera utilizar los servicios (tipos, subprogramas, ...) de ese módulo de biblioteca.

La directiva `#include` añade las declaraciones del archivo de cabecera en el código del módulo (*preprocesamiento*):

main.cpp

```
#include "Lista.h"
...
```

Preprocesador



main.cpp

```
const int TM = ...;
typedef double tTupla[TM];
typedef struct {
    tTupla elem;
    int cont;
} tLista;

void init(tLista&);
bool insert(tLista&, double);
bool delete(tLista&, int);
int size(tLista);
...
```

Lista.h

```
const int TM = ...;
typedef double tTupla[TM];
typedef struct {
    tTupla elem;
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);
bool delete(tLista&, int);

int size(tLista);

void sort(tLista&);
```

Es todo lo que se necesita saber para comprobar si el código de main.cpp hace un uso correcto de la lista (declaraciones y llamadas a funciones).



# Interfaz frente a implementación

## Creación de módulos de biblioteca

### Implementación

Compilar el módulo significa compilar su archivo de implementación (.cpp).

También necesita conocer sus propias declaraciones:

Lista.cpp

```
#include "Lista.h"
...
```

Lista.cpp

```
#include "Lista.h"

void init(tLista& lista) {
    lista.cont = 0;
}

bool insert(tLista& lista, double valor)
{
    if (lista.cont == TM)
        return false;
    else {
        lista.elem[lista.cont]=valor;
        lista.cont++;
    }
}
```



Lista.obj

```
00101110101011001010010101
0010101001010111110101000
1010010101010100101010101
0110010101010101010101001
0101010101000001010101101
0100101010101010000101011
110010101010111100110010101
0110101010100100101010111
001010101001010100101010010
1010010101010010100010011110
1001010101001010101001010100
101010101010010101001010101
010000101010111001010010100
011101010111010011010100101
010111111101010110011010111
0000100101010010101010110
```

Cuando se compila el módulo se genera el código objeto.

Mientras no se modifique el código fuente no hay necesidad de recompilar el módulo.

## Uso de módulos de biblioteca

- ✓ Necesita sólo el archivo de cabecera para compilar.
- ✓ Se adjunta el código objeto del módulo durante el enlace.



# Fundamentos de la programación

---

## Uso de módulos de biblioteca



# Programación modular

## *Uso de módulos de biblioteca*

Ejemplo: Gestión de una lista de datos ordenada (Tema 7)

Todo lo que tenga que ver con la lista en sí estará en su propio módulo.

Ahora el código estará repartido en tres archivos:

- ✓ `lista.h`: archivo de cabecera del módulo de lista de datos
- ✓ `lista.cpp`: archivo de implementación del módulo de lista de datos
- ✓ `datosMain.cpp`: programa principal que usa el módulo de lista de datos

Tanto `lista.cpp` como `datosMain.cpp` deben incluir al principio `lista.h`.

Como es un módulo propio de la aplicación, en la directiva `#include` usamos dobles comillas:

```
#include "lista.h"
```

Los archivos de cabecera de las bibliotecas del sistema siempre se encierran entre ángulos y no tienen que llevar extensión `.h`.




## *Módulo: Gestión de una lista de datos ordenada I*

lista.h

```
// lista.h
#include <string>
using namespace std;

const int TM = ...;
typedef struct {
    int codigo;
    string nombre;      // requiere la librería string
    double sueldo;
} tRegistro;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
...
```





¡Documenta bien el código!

```
void mostrar(const tRegistro & registro);
void mostrar(const tLista & lista);
void solicitar(tRegistro & registro);

bool operator<(const tRegistro & opIzq, const tRegistro & opDer);

bool insertar(tLista &lista, const tRegistro & registro);
bool eliminar(tLista &lista, int pos);
bool buscar(const tLista & lista, const string & nombre, int & pos);
bool cargar(const string & nombArchivo, tLista &lista);
    // requieren la librería string → #include <string>
bool guardar(const string & nombreArchivo, const tLista & lista);
```

Cada prototipo irá con un comentario que explique la utilidad del subprograma, los datos de E/S, particularidades, ...  
(Aquí se omiten por cuestión de espacio.)



## *Módulo: Gestión de una lista de datos ordenada I*

lista.cpp

```
// lista.cpp
#include "lista.h"    // lista.cpp implementa lista.h
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

void solicitar(tRegistro & registro) {
    cout << "Introduce el codigo: "; cin >> registro.codigo;
    cout << "Introduce el nombre: "; cin >> registro.nombre;
    cout << "Introduce el sueldo: "; cin >> registro.sueldo;
    // requieren la librería <iostream> -> #include
}

void mostrar(const tLista & lista){
    ...
}
```



```
bool insertar(tLista & lista, const tRegistro & registro) {
    bool ok = true;
    if (lista.cont == TM)
        ok = false; // lista llena
    else {
        int pos;
        bool enc= buscar(lista, registro.nombre, pos);
        if(enc) ok= false; // Duplicado
        else { // Insertamos en la posición pos
            desplazarDerecha(lista, pos);
            lista.registros[pos] = registro;
            lista.cont++;
            ok = true;
        }
        return ok;
    }
    ...
}
```



```
bool eliminar(tLista &lista, int pos) {
    bool ok = true;
    if ((pos < 0) || (pos >= lista.cont))
        ok = false; // Posición inexistente
    else {
        desplazarIzquierda(lista, pos);
        lista.cont--;
        ok = true;
    }
    return ok;
}
```

```
bool buscar(const tLista &lista, const string & nombre, int & pos) {
    int ini = 0, fin = lista.cont - 1, mitad;
    bool encontrado = false;
    while ((ini <= fin) && !encontrado) {
        ...
    }
}
```



## *Módulo: Gestión de una lista de datos ordenada I*

**datosMain.cpp**

```
// datosMain.cpp (main)
#include <iostream>
using namespace std;
#include "lista.h"
const char BD[] = "datos.txt";
int menu();

int menu() {
    cout << endl;
    cout << "1 - Insertar" << endl;
    cout << "2 - Eliminar" << endl;
    cout << "3 - Buscar" << endl;
    cout << "0 - Salir" << endl;
    int op;
    do {
        cout << "Elige: ";
        cin >> op;
    } while ((op < 0) || (op > 3));
    return op;
}
```



```
int main() {
    tLista lista; // requiere lista → #include
    if (!cargar(BD, lista)) cout<< "Error al cargar el archivo!"<< endl;
    else {
        mostrar(lista);
        int op = menu();
        while (op != 0) {
            op = menu();
            if (op == 1) { tRegistro registro;
                solicitar(registro);
                if (!insertar(lista, registro))
                    cout << "Error: lista llena!" << endl;
            }
            else if (op == 2) { int pos;
                cout << "Posicion: "; cin >> pos;
                if (!eliminar(lista, pos-1))
                    cout << "Error: Posicion inexistente!" << endl;
            }
            ...
        }
    }
}
```



# Fundamentos de la programación

---

## Compilación de programas multiarchivo



# Programación modular

## *Compilación de programas multiarchivo*

G++

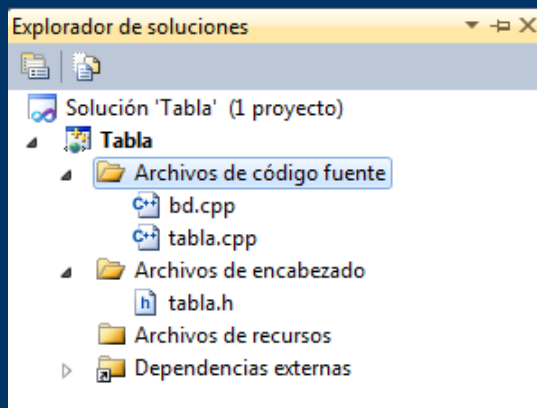
Si están todos los archivos de cabecera y de implementación en el mismo directorio simplemente listamos todos los .cpp en la orden g++:

```
D:\FP\Tema8>g++ -o datosMain.exe lista.cpp datosMain.cpp
```

Recuerda que sólo se compilan los .cpp.

Visual C++/Studio

Muestra los archivos de cabecera y de implementación organizados en grupos distintos:



A los archivos de cabecera los llama de encabezado.

Con la opción Generar solución se compilan todos los .cpp.





# Fundamentos de la programación

---

## El preprocesador



# Programación modular

## El Preprocesador

Directivas: `#...`

Antes de realizar la compilación se pone en marcha el *preprocesador*.

Interpreta las directivas y genera un único archivo temporal con todo el código del módulo o programa principal. Como en la inclusión (directiva `#include`):

```
// lista.h

#include <string>
using namespace std;

const int TM = ...;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;

typedef tRegistro tTupla[TM];

typedef struct {
    tTupla registros;
    int cont;
} tLista;

...
```

```
// datosMain.cpp (main)

#include "lista.h"

int menu();
...
```

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

const int TM = ...;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;

typedef tRegistro tTupla[TM];

typedef struct {
    tTupla registros;
    int cont;
} tLista;

...

int menu();
...
```



# Fundamentos de la programación

---

Cada cosa en su módulo



# Programación modular

## *Distribuir la funcionalidad del programa en módulos*

Un módulo encapsula un conjunto de subprogramas que tienen relación entre sí.

La relación puede venir dada por una estructura de datos sobre la que trabajan.

O por ser todos subprogramas de un determinado contexto de aplicación (bibliotecas de funciones matemáticas, de cadenas, etcétera).

A menudo las estructuras de datos contienen otras estructuras:

```
const int TM = ...;
typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;

typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
```

Una lista es una tupla de registros.

- ✓ Estructura tRegistro
- ✓ Estructura tLista  
(contiene datos de tipo tRegistro)

La gestión de cada estructura, en su módulo.




## *Gestión de una lista de datos ordenada II*

registro.h

```
// registro.h
#include <string>
using namespace std;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;

void solicitar(tRegistro & registro);
void mostrar(const tRegistro & registro);
bool operator<(const tRegistro & opIzq, const tRegistro & opDer);
...
```



## Gestión de una lista de datos ordenada II

registro.cpp

```
// registro.cpp
#include "registro.h"    // registro.cpp implementa registro.h
#include <iostream>
#include <iomanip>
using namespace std;

void solicitar(tRegistro & registro);
{
    cout << "Introduce el codigo: "; cin >> registro.codigo;
    cout << "Introduce el nombre: "; cin >> registro.nombre;
    cout << "Introduce el sueldo: "; cin >> registro.sueldo;
}
// requieren la librería <iostream>

void mostrar(const tRegistro& registro) {
    cout << setw(10) << registro.codigo
         << setw(20) << registro.nombre
         << setw(12) << fixed << setprecision(2)
         << registro.sueldo << endl;
}
// requiere la librería <iomanip>

...
```



## Gestión de una lista de datos ordenada II

lista2.h

```
// lista2.h
#include <string>
using namespace std;
#include "registro.h"

const int TM = ...;
typedef tRegistro tTupla[TM]; // requiere registro
typedef struct {
    tTupla registros;
    int cont;
} tLista;

void mostrar(const tLista& lista);
bool insertar(tLista& lista, const tRegistro& registro);
bool eliminar(tLista& lista, tPos pos);
bool buscar(const tLista& lista, const string & nombre, tPos & pos);
bool cargar(const string & nombreArch, tLista& lista);
void guardar(const string & nombreArch, const tLista & lista);
```



## *Gestión de una lista de datos ordenada II*

lista2.cpp

```
// lista2.cpp
#include "lista2.h"          // lista2.cpp implementa lista2.h
#include <iostream>
#include <fstream>
using namespace std;

void mostrar(const tLista& lista) {
    for(tPos p=0; p<lista.cont; p++){
        cout << setw(3) << p+1 << ": ";
        mostrar(lista.registros[p]);
    }
}

bool insertar(tLista& lista, const tRegistro & registro) {
    bool ok = true;
    if (lista.cont == TM) ok = false; // lista llena
    else {
        ...
    }
    return ok;
}

...
```





## Gestión de una lista de datos ordenada II

datosMain2.cpp

```
// datosMain2.cpp (main)
#include <iostream>
using namespace std;
#include "registro.h"
#include "lista2.h"

const char BD[] = "datos.txt";

int menu();

int menu() {
    cout << endl;
    cout << "1 - Insertar" << endl;
    cout << "2 - Eliminar" << endl;
    cout << "3 - Buscar" << endl;
    cout << "0 - Salir" << endl;
    int op;
    do {
        cout << "Elige: ";
        cin >> op;
    } while ((op < 0) || (op > 3));
    return op;
}
...
```



*Este ejemplo tiene errores de compilación  
por inclusiones múltiples*



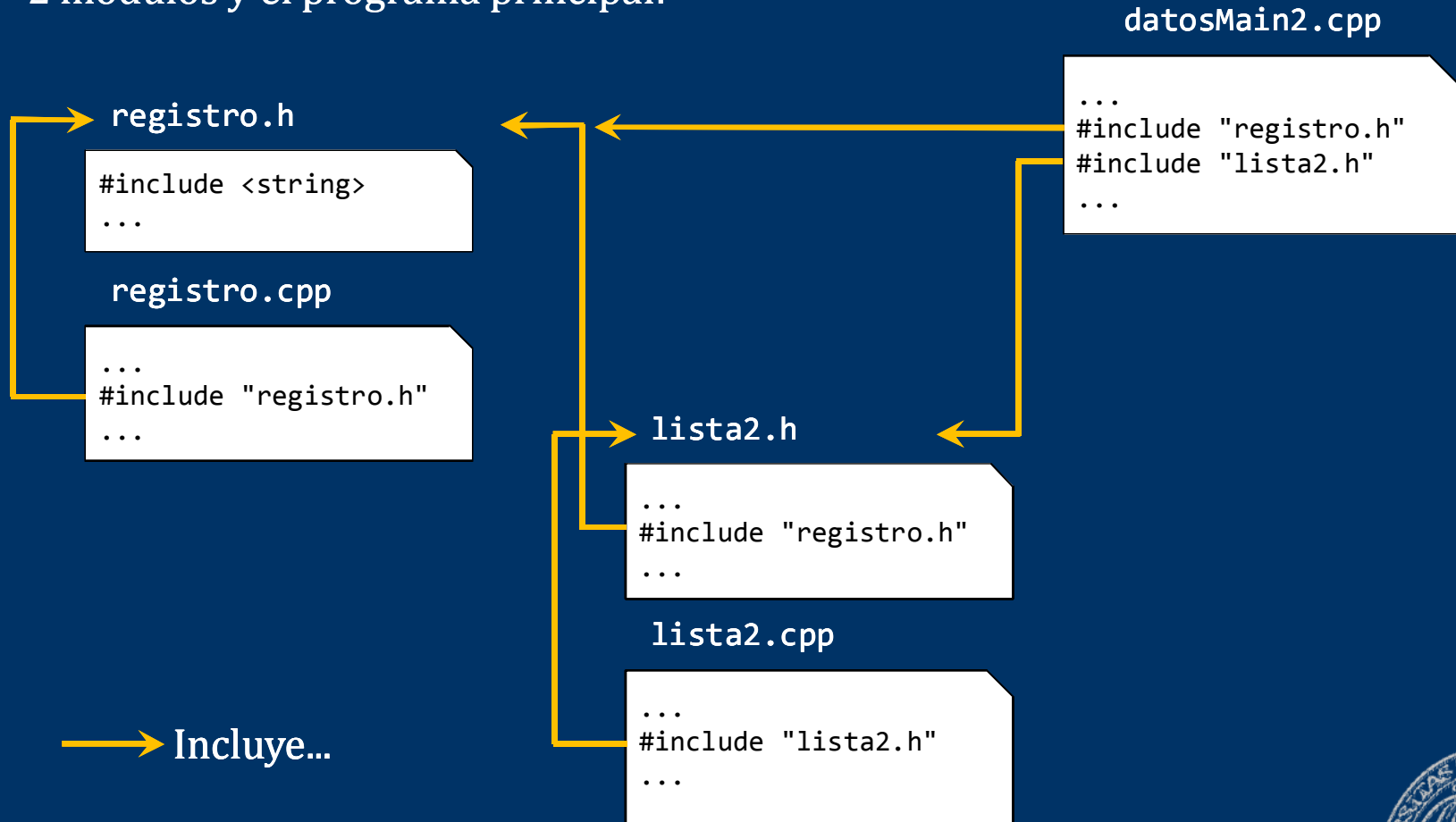
## El problema de las inclusiones múltiples



# Programación modular

## *Gestión de una lista de datos ordenada II*

2 módulos y el programa principal:



# Programación modular

## Gestión de una lista de datos ordenada II

Preprocesamiento de `#include`:

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

```
#include "registro.h"
```

```
#include "lista2.h"
```

```
const char BD[] = "
```

```
int menu();
```

```
...
```

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

```
typedef struct {
    ...
} tRegistro;
...
```

```
#include <string>
using namespace std;
#include "registro.h"

const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
...
```

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

```
typedef struct {
    ...
} tRegistro;
...
```



# Programación modular

## Gestión de una lista de datos ordenada II

Preprocesamiento de `#include`:

 Sustituido

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

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

```
typedef struct {
    ...
} tRegistro;
...
```

```
#include "lista2.h" ←
```

```
const char BD[] = "datos.txt"
```

```
int menu();
```

```
...
```

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

```
typedef struct {
    ...
} tRegistro;
...
```

```
const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
...
```



# Programación modular

## Gestión de una lista de datos ordenada II

Preprocesamiento de `#include`:

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

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

```
typedef struct {
    ...
} tRegistro;
...
```

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

typedef struct {
    ...
} tRegistro;
...
```

*¡Identificador duplicado!*

```
const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
...

int menu();

...
```



# Programación modular

## *Compilación condicional*

Directivas `#ifdef`, `#ifndef`, `#else` y `#endif`.

Se usan en conjunción con la directiva `#define`.

<code>#define X</code>	<code>#define X</code>
<code>#ifdef X</code>	<code>#ifndef X</code>
<code>... // Código if</code>	<code>... // Código if</code>
<code>[#else</code>	<code>[#else</code>
<code>... // Código else</code>	<code>... // Código else</code>
<code>]</code>	<code>]</code>
<code>#endif</code>	<code>#endif</code>

La directiva `#define`, como su nombre indica, define un símbolo (identificador).

En el caso de la izquierda se compilará el "Código if" y no el "Código else", en caso de que lo haya. En el caso de la derecha, al revés, o nada si no hay else.

Las cláusulas else son opcionales.

Directivas útiles para configurar distintas versiones.



# Programación modular

## *Protección frente a inclusiones múltiples*

lista.cpp incluye registro.h y datosMain.cpp también incluye registro.h → ¡Identificadores duplicados!

Cada módulo debe incluirse una y sólo una vez.

Protección frente a inclusiones múltiples:

```
#ifndef X
```

```
#define X
```

```
... // Código del módulo
```

```
#endif
```

El símbolo *X* debe ser único para cada módulo.

La primera vez que se encuentra ese código el preprocesador, incluye el código del módulo por no estar definido el símbolo *X*, pero al mismo tiempo define ese símbolo. De esta forma, la siguiente vez que se encuentre ese `#ifndef` ya no vuelve a incluir el código del módulo, pues ya ha sido definido el símbolo *X*.

Para cada módulo elegimos como símbolo *X* el nombre del archivo, sustituyendo el punto por un subrayado: REGISTRO\_H, LISTA\_H, ...





## *Gestión de una lista de datos ordenada III*

registrofin.h

```
#ifndef REGISTROFIN_H
#define REGISTROFIN_H

#include <string>
using namespace std;

typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;

void mostrar(const tRegistro & registro);
bool operator<(const tRegistro &opIzq, const tRegistro &opDer);
void solicitar(tRegistro & registro);
...

#endif
```



## Gestión de una lista de datos ordenada III

registrofin.cpp

```
#include "registrofin.h"           // registrofin.cpp implementa registrofin.h
#include <iostream>
#include <fstream>
using namespace std;
#include <iomanip>

void solicitar(tRegistro & registro){
    cout << "Introduce el codigo: ";
    cin >> registro.codigo;
    cout << "Introduce el nombre: ";
    cin >> registro.nombre;
    cout << "Introduce el sueldo: ";
    cin >> registro.sueldo;
}

bool operator<(const tRegistro & opIzq, const tRegistro & opDer) {
    return opIzq.nombre < opDer.nombre;
}

...
```



## Gestión de una lista de datos ordenada III

listafin.h

```
#ifndef LISTAFIN_H
#define LISTAFIN_H
#include <string>
using namespace std;
#include "registrofin.h"

const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
const char BD[] = "datos.txt";

void mostrar(const tLista &lista);
bool insertar(tLista &lista, const tRegistro & registro);
bool eliminar(tLista &lista, int pos);
bool buscar(const tLista & lista, const string & nombre, int & pos);
bool cargar(const string & nombreArch, tLista &lista);
void guardar(const string & nombreArch, const tLista & lista);
#endif
```



## Gestión de una lista de datos ordenada III

listafin.cpp

```
#include "listafin.h"           // listafin.cpp implementa listafin.h
#include <iostream>
#include <fstream>
using namespace std;

bool insertar(tLista &lista, const tRegistro & registro) {
    bool ok = true;
    if (lista.cont == TM)
        ok = false; // lista llena
    else {
        int i = lista.cont;
        while ((i > 0) && (registro < lista.registros[i - 1])){
            lista.registros[i] = lista.registros[i - 1];
            i--;
        }
        lista.registros[i] = registro;
        lista.cont++;
    }
    return ok;
} ...
```



## Gestión de una lista de datos ordenada III

datosMainfin.cpp

```
#include <iostream>
using namespace std;
#include "registrofin.h"
#include "listafin.h"

int menu();

int menu() {
    cout << endl;
    cout << "1 - Insertar" << endl;
    cout << "2 - Eliminar" << endl;
    cout << "3 - Buscar" << endl;
    cout << "0 - Salir" << endl;
    int op;
    do {
        cout << "Elige: ";
        cin >> op;
    } while ((op < 0) || (op > 3));
    return op;
}
...
```

*¡Ahora ya puedes compilarlo!*



# Programación modular

## Gestión de una lista de datos ordenada III

Preprocesamiento de `#include` en `datosMainfin.cpp`:

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

#include "registrofin.h" ←
#include "listafin.h"

int menu();

...
```

```
#ifndef REGISTROFIN_H
#define REGISTROFIN_H
#include <string>
using namespace std;

typedef struct {
    ...
} tRegistro;

...
```

REGISTROFIN\_H no se ha definido todavía



# Programación modular

## Gestión de una lista de datos ordenada III

Preprocesamiento de `#include` en `datosMainfin.cpp`:

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

```
#define REGISTROFIN_H
#include <string>
using namespace std;

typedef struct {
    ...
} tRegistro;
...
```

```
#include "listafin.h"
```

```
int menu();
```

```
...
```

```
#ifndef LISTAFIN_H
#define LISTAFIN_H
#include <string>
using namespace std;
#include "registrofin.h"
```

```
const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
...
```

LISTAFIN\_H no se ha definido todavía



# Programación modular

## Gestión de una lista de datos ordenada III

Preprocesamiento de `#include` en `datosMainfin.cpp`:

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

```
#define REGISTROFIN_H
#include <string>
using namespace std;
```

```
typedef struct {
    ...
} tRegistro;
...
```

```
#define LISTAFIN_H
#include <string>
using namespace std;
#include "registrofin.h"

...

int menu();

...
```

```
#ifndef REGISTROFIN_H
#define REGISTROFIN_H
#include <string>
using namespace std;

typedef struct {
    ...
} tRegistro;
...
```

*¡REGISTROFIN\_H  
ya se ha definido!*





# Fundamentos de la programación

---

## Espacios de nombres



# Espacios de nombres

## *Agrupaciones lógicas de declaraciones*

Un *espacio de nombres* permite agrupar entidades (tipos, variables, funciones) bajo un nombre distintivo. Se puede considerar que el ámbito global del programa completo está dividido en subámbitos, cada uno con su propio nombre.

Forma de un espacio de nombres:

```
namespace nombre {  
    // Declaraciones (entidades)  
}
```

Por ejemplo:

```
namespace miEspacio {  
    int i;  
    double d;  
}
```

Se declaran las variables *i* y *d* dentro del espacio de nombres *miEspacio*.



# Espacios de nombres

## *Acceso a miembros de un espacio de nombres*

Para acceder a las entidades declaradas dentro de un espacio de nombres hay que utilizar el *operador de resolución de ámbito* (::).

Por ejemplo, para acceder a las variables declaradas en el espacio de nombres `miEspacio` cualificamos esos identificadores con el nombre del espacio y el operador de resolución de ámbito:

```
miEspacio::i  
miEspacio::d
```

Los espacios de nombres resultan útiles en aquellos casos en los que pueda haber entidades con el mismo identificador en distintos módulos o en ámbitos distintos de un mismo módulo.

Encerraremos cada declaración en un espacio de nombres distinto:

```
namespace primero {  
    int x = 5;  
}  
  
namespace segundo {  
    double x = 3.1416;  
}
```

Ahora se distingue entre `primero::x` y `segundo::x`.



# Espacios de nombres

## *using namespace*

La instrucción `using namespace` se utiliza para introducir todos los nombres de un espacio de nombres en el ámbito de declaraciones actual:

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

```
namespace primero {
    int x = 5;
    int y = 10;
}
namespace segundo {
    double x = 3.1416;
    double y = 2.7183;
}
int main() {
```

`using namespace`  
sólo tiene efecto  
en el bloque  
en que se encuentra.

```
5
10
3.1416
2.7183
```

```
    using namespace primero;
    cout << x << endl; // x es primero::x
    cout << y << endl; // y es primero::y
    cout << segundo::x << endl; // espacio de nombres explícito
    cout << segundo::y << endl; // espacio de nombres explícito
    return 0;
}
```



# Espacios de nombres

*Ejemplo de uso de espacios de nombres*

```
// listaES.h
#ifndef LISTAEN_H
#define LISTAEN_H
#include <string>
#include "registrofin.h"
namespace ord { // Lista ordenada
    const int TM = ...;
    typedef tRegistro tTupla[TM];
    typedef struct {
        tTupla registros;
        int cont;
    } tLista;
    const char BD[] = "datos.txt";

    void mostrar(const tLista &lista);
    bool insertar(tLista &lista, const tRegistro & registro);
    bool eliminar(tLista &lista, int pos);
    bool buscar(const tLista &lista, const string & nombre, int & pos);
    bool cargar(const string & nombreArch, tLista &lista);
    void guardar(const string & nombreArch, const tLista &lista);
} // namespace
#endif
```



# Espacios de nombres

## *Archivo de implementación*

```
#include "listaEN.h"           // listaEN.cpp implementa listaEN.h
#include <iostream>
#include <fstream>
using namespace std;

// IMPLEMENTACIÓN DE LAS FUNCIONES DEL ESPACIO DE NOMBRES ord

bool ord::insertar(tLista &lista, const tRegistro & registro) { // ...
}

bool ord::eliminar(tLista &lista, int pos) { // ...
}

bool ord::buscar(const tLista &lista, const string & nombre, int & pos) { // ...
}

void ord::mostrar(const tLista &lista) { // ...
}

bool ord::cargar(const string & nombreArch, tLista &lista) { // ...
}

void ord::guardar(const string & nombreArch, const tLista &lista) { // ...
}
```



# Espacios de nombres

## *Uso del espacio de nombres*

Cualquier módulo que utilice `listaEN.h` debe cualificar cada nombre de esa biblioteca con el nombre del espacio de nombres en que se encuentra:

```
#include <iostream>
using namespace std;
#include "registrofin.h"
#include "listaEN.h"
```

...

```
int main() {
    ord::tLista lista;

    if (!ord::cargar(lista)) {
        cout << "Error al abrir el archivo!" << endl;
    }
    else {
        ord::mostrar(lista);
        ...
    }
}
```

O utilizar una instrucción `using namespace ord;`



# Espacios de nombres

## *Uso de espacios de nombres*

```
#include <iostream>
using namespace std;
#include "registrofin.h"
#include "listaEN.h"
using namespace ord; ←

...

int main() {
    tlista lista;
    if (!cargar(lista)) {
        cout << "Error al abrir el archivo!" << endl;
    }
    else {
        mostrar(lista);
        ...
    }
}
```





# Implementaciones alternativas

## *Misma interfaz, implementación alternativa*

lista.h

```
#include <string>
using namespace std;
#include "registrofin.h"

const int TM = ...;
typedef tRegistro tTupla[TM];
typedef struct {
    tTupla registros;
    int cont;
} tLista;
```

```
bool insertar(tLista &lista, const tRegistro & registro);
```

Lista  
ordenada

```
bool insertar(tLista &lista, const tRegistro & registro) {
    bool ok = true;
    if (lista.cont == TM) ok = false; // lista llena
    else {
        int i = lista.cont;
        while ((i > 0) && (registro < lista.registros[i - 1])) {
            lista.registros[i] = lista.registros[i - 1];
            i--;
        }
        lista.registros[i] = registro;
        lista.cont++;
    }
    return ok;
}
```

Lista  
no ordenada

```
bool insertar(tLista &lista, const tRegistro & registro)
{
    bool ok = true;
    if (lista.cont == TM) ok = false; // lista llena
    else {
        lista.registros[lista.cont] = registro;
        lista.cont++;
    }
    return ok;
}
```



# Espacios de nombres

## *Implementaciones alternativas*

Distintos espacios de nombres para distintas implementaciones.

¿Lista ordenada o lista desordenada?

```
namespace ord { // Lista ordenada
    const int TM = ...;
    typedef tRegistro tTupla[TM];
    ...
    void mostrar(const tLista& lista);
    bool insertar(tLista& lista, const tRegistro & registro);
    ...
} // namespace
```

```
namespace des { // Lista desordenada
    const int TM = ...;
    typedef tRegistro tTupla[TM];
    ...
    void mostrar(const tLista& lista);
    bool insertar(tLista& lista, const tRegistro & registro);
    ...
} // namespace
```



## Implementaciones alternativas

listaEN.h

Todo lo que sea común puede estar fuera de la estructura `namespace`:

```
#ifndef LISTAEN_H
#define LISTAEN_H

#include "registrofin.h"

const int TM = ...;

typedef tRegistro tTupla[TM];
typedef struct {
    tLista registros;
    int cont;
} tLista;

void mostrar(const tLista &lista);
bool eliminar(tLista& lista, int pos);

...
```



# Espacios de nombres

## *Implementaciones alternativas*

```
namespace ord { // Lista ordenada
```

```
    bool insertar(tLista &lista, const tRegistro & registro);  
    bool buscar(const tLista &lista, const string & nombre, int & pos);  
    bool cargar(const string & nombreArch, tLista &lista);  
    void guardar(const string & nombreArch, const tLista &lista);  
} // namespace
```

```
namespace des { // Lista desordenada
```

```
    bool insertar(tLista &lista, const tRegistro & registro);  
    bool buscar(const tLista &lista, const string & nombre, int & pos);  
    bool cargar(const string & nombreArch, tLista &lista);  
    void guardar(const string & nombreArch, const tLista &lista);  
} // namespace
```

```
#endif
```



## *Implementaciones alternativas*

listaEN.cpp

```
#include "listaEN.h"    // listaEN.cpp implementa listaEN.h
#include <iostream>
#include <fstream>
using namespace std;

// parte común

bool eliminar(tLista &lista, int pos) {

    // ...

}

void mostrar(const tLista &lista) {

    // ...

}
```



# Espacios de nombres

## *Implementaciones alternativas*

```
// IMPLEMENTACIÓN DE LAS FUNCIONES DEL ESPACIO DE NOMBRES ord

bool ord::insertar(tLista &lista, const tRegistro & registro) {
    // insertar ordenadamente
}

bool ord::buscar(const tLista &lista, const string & nombre, int & pos){
    // búsqueda binaria
}
...
// IMPLEMENTACIÓN DE LAS FUNCIONES DEL ESPACIO DE NOMBRES des

bool des::insertar(tLista &lista, const tRegistro & registro) {
    // insertar al final
}

bool des::buscar(const tLista &lista, const string & nombre, int & pos){
    // búsqueda secuencial
}
...
```



## Implementaciones alternativas

```
#include <iostream>
using namespace std;
#include "registrofin.h"
#include "listaEN.h"
using namespace ord;

int menu();

int main() {
    tList lista;
    ...
    tRegistro registro;
    solicitar(registro);
    if (!insertar(lista, registro))
    ...
}
```

```
D:\FP\Tema9>tablaEN
1: 12345 Alvarez 120000.00
2: 11111 Benitez 100000.00
3: 21112 Dominguez 90000.00
4: 11111 Duran 120000.00
5: 22222 Fernandez 120000.00
6: 12345 Gomez 100000.00
7: 10000 Hernandez 150000.00
8: 21112 Jimenez 100000.00
9: 54321 Manzano 95000.00
10: 11111 Perez 90000.00
11: 12345 Sanchez 90000.00
12: 10000 Sergei 100000.00
13: 33333 Tarazona 120000.00
14: 12345 Turegano 100000.00
15: 11111 Urpiano 90000.00

1 - Insertar
2 - Eliminar
3 - Buscar
0 - Salir
Elige: 1
Introduce el codigo: 33333
Introduce el nombre: Calvo
Introduce el sueldo: 95000
1: 12345 Alvarez 120000.00
2: 11111 Benitez 100000.00
3: 33333 Calvo 95000.00
4: 21112 Dominguez 90000.00
5: 11111 Duran 120000.00
6: 22222 Fernandez 120000.00
7: 12345 Gomez 100000.00
8: 10000 Hernandez 150000.00
9: 21112 Jimenez 100000.00
10: 54321 Manzano 95000.00
11: 11111 Perez 90000.00
12: 12345 Sanchez 90000.00
13: 10000 Sergei 100000.00
14: 33333 Tarazona 120000.00
15: 12345 Turegano 100000.00
16: 11111 Urpiano 90000.00
```



## Implementaciones alternativas

```
#include <iostream>
using namespace std;
#include "registrofin.h"
#include "listaEN.h"
using namespace des;

int menu();

int main() {
    tList lista;
    ...
    tRegistro registro;
    solicitar(registro);
    if (!insertar(lista, registro))
    ...
}
```

```
D:\FP\Tema9>tablaEN
1:      12345      Alvarez      120000.00
2:      11111      Benitez      100000.00
3:      21112      Dominguez     90000.00
4:      11111      Duran          120000.00
5:      22222      Fernandez     120000.00
6:      12345      Gomez          100000.00
7:      10000      Hernandez     150000.00
8:      21112      Jimenez       100000.00
9:      54321      Manzano       95000.00
10:     11111      Perez          90000.00
11:     12345      Sanchez        90000.00
12:     10000      Sergei         100000.00
13:     33333      Tarazona       120000.00
14:     12345      Turegano       100000.00
15:     11111      Urpiano        90000.00

1 - Insertar
2 - Eliminar
3 - Buscar
0 - Salir
Elige: 1
Introduce el codigo: 33333
Introduce el nombre: Calvo
Introduce el sueldo: 95000
1:      12345      Alvarez      120000.00
2:      11111      Benitez      100000.00
3:      21112      Dominguez     90000.00
4:      11111      Duran          120000.00
5:      22222      Fernandez     120000.00
6:      12345      Gomez          100000.00
7:      10000      Hernandez     150000.00
8:      21112      Jimenez       100000.00
9:      54321      Manzano       95000.00
10:     11111      Perez          90000.00
11:     12345      Sanchez        90000.00
12:     10000      Sergei         100000.00
13:     33333      Tarazona       120000.00
14:     12345      Turegano       100000.00
15:     11111      Urpiano        90000.00
16:     33333      Calvo          95000.00
```





# Fundamentos de la programación

---

## Calidad y reutilización del software



# Calidad del software

---

## *Software de calidad*

El software debe ser desarrollado con buenas prácticas de ingeniería del software que aseguren un buen nivel de calidad.

Los distintos módulos de la aplicación deben ser probados exhaustivamente, tanto de forma independiente como en su relación con los demás módulos.

El proceso de prueba y depuración es muy importante y todos los equipos deberán seguir buenas pautas para asegurar la calidad.

Los módulos deben ser igualmente bien documentados, de forma que otros desarrolladores puedan aprovecharlo en otros proyectos.



# Reutilización del software

---

## *No reinventemos la rueda*

Debemos desarrollar el software pensando en su posible reutilización.

Un software de calidad bien documentado debe poder ser fácilmente reutilizado.

Los módulos de nuestra aplicación deben poder ser fácilmente usados, quizá con modificaciones, en otras aplicaciones con necesidades parecidas.

Por ejemplo, si necesitamos una aplicación que gestione una lista de longitud variable de registros con NIF, nombre, apellidos y edad, partiríamos de los módulos `registro` y `lista` de la aplicación anterior. Las modificaciones básicamente afectarían al módulo `registro`.



# Referencias bibliográficas

---



- ✓ *El lenguaje de programación C++* (Edición especial)  
B. Stroustrup. Addison-Wesley, 2002






# Acerca de *Creative Commons*



## Licencia CC (*Creative Commons*)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

