

Fundamentos de la programación II

Programación modular

Facultad de Informática
Universidad Complutense

Ana Gil Luezas
(Adaptadas del original de Luis Hernández Yáñez)



Índice

Programas multiarchivo y compilación separada	2
Interfaz frente a implementación	6
Uso de módulos	9
Ejemplo I	12
El preprocesador	16
Cada cosa en su módulo	17
Ejemplo II	18
Dependencia entre módulos	23
El problema de las inclusiones múltiples	24
Compilación condicional	26
Ejemplo III	28
El problema de los círculos de inclusiones	36
Espacios de nombres	37
Calidad y reutilización del software	40



Programas multiarchivo

El código fuente del programa se reparte entre varios archivos (*módulos*).

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 struct {
    double elem[TM];
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(const 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(const tLista&);
double min(const tLista&);
double max(const tLista&);
double desv(const tLista&);
int minIndex(const tLista&);
int maxIndex(const tLista&);
double sum(const tLista&);
```

Archivos

```
bool cargar(tLista&, string);
bool guardar(tLista, string);
bool mezclar(string, string);
int size(const string&);
bool exportar(const string&);
```

Ejecutable



Compilación separada

Cada módulo se compila a código objeto de forma independiente:

Lista

```
const int TM = ...;

typedef struct {
    double elem[TM];
    int cont;
} tlista;

void init(tlista&);

bool insert(tlista&, double);

bool delete(tlista&, int);

int size(const tlista&);

void sort(tlista&);
```

Lista.obj

```
00101110101011001010010010101
0010101001010101111010101000
101001010101010100101010101
011001010101010101010101001
01010101010100000101010101
010010101010101010000101011
110010101010111100110010101
0110101010100100101001111
001010101010010101001010010
10100101010100101000010011110
100101010110010101001010100
10101010101001001010010101
010000101010111001010101010
01110101011101001101010101
010111111101010110011010111
000010010101001010101010110
```

Cálculos

```
double mean(const tlista&);

double min(const tlista&);

double max(const tlista&);

double desv(const tlista&);

int minIndex(const tlista&);

int maxIndex(const tlista&);

double sum(const tlista&);

...
```

Calculos.obj

```
010110010100100101010010100
101010111110101000101001010
1010101001010101010101001010
101010101010101010010101010
1010000010101011101010010101
01010101000010101011110010101
0101011100110010101011010101
01010010010101001111001010101
010010101001010010101001010
10100101000010011110100101010
11001010101001010100101010101
010100101010010101010000101
01011100101010010100011101010
111010011010101001010111111
101010110011010111000010010
1010010101010101010001111010
```

Archivos

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

bool guardar(tlista, string);

bool mezclar(string, string);

int size(const string&);

bool exportar(const string&);

...
```

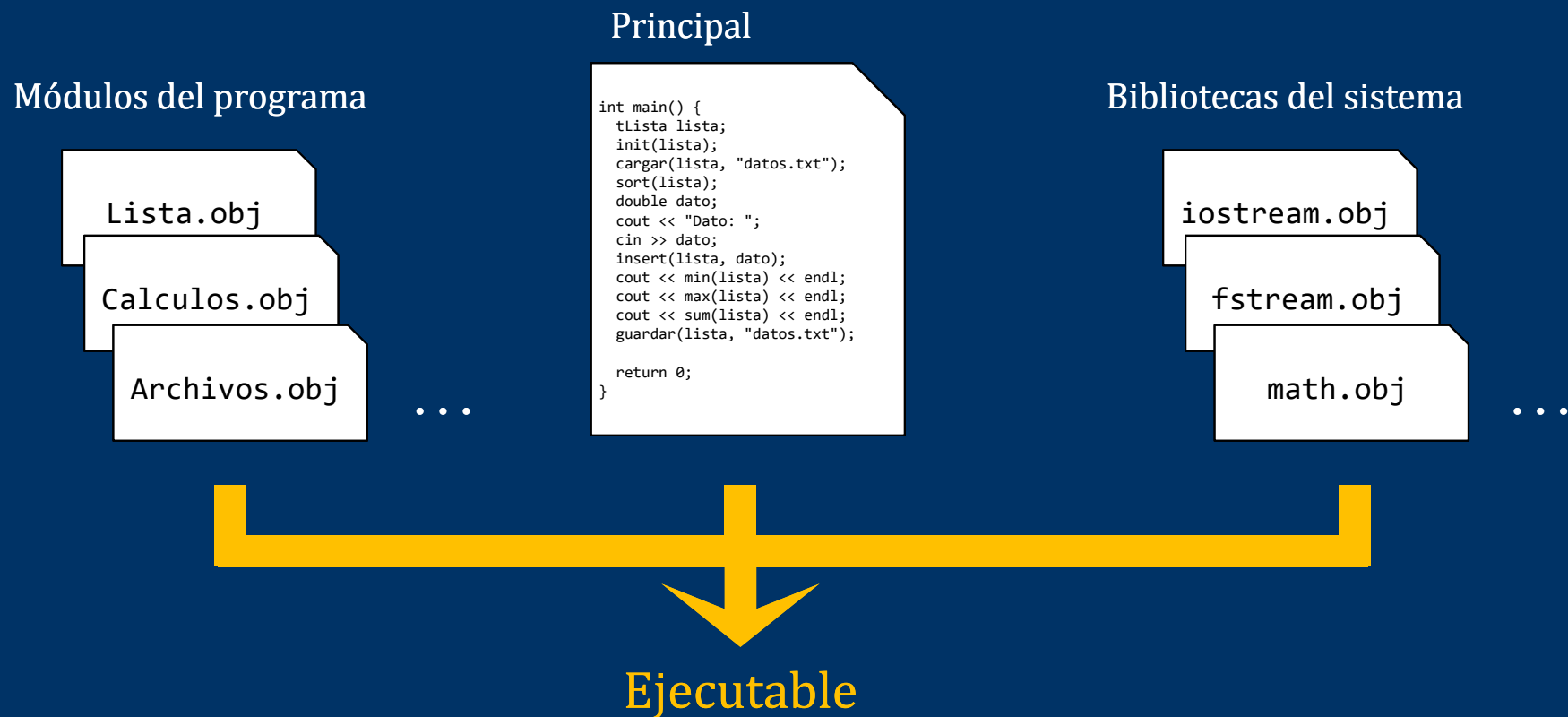
Archivos.obj

```
11101010110010100100101010010
1010010101011110101010001010
01010101010001010101010110
010101010101010101010010101
010101010000010101011010100
10101010101000001010111100
1010101010111001100101010110
101010100100101010011110010
101010010101001010010101010
010101001010000100111101001
010101100101010010101001010
1010101001010010101010100
00101010111001010100101000111
01010111010011010101010101
11111101010100110101110000
100101010010101010101101111
```



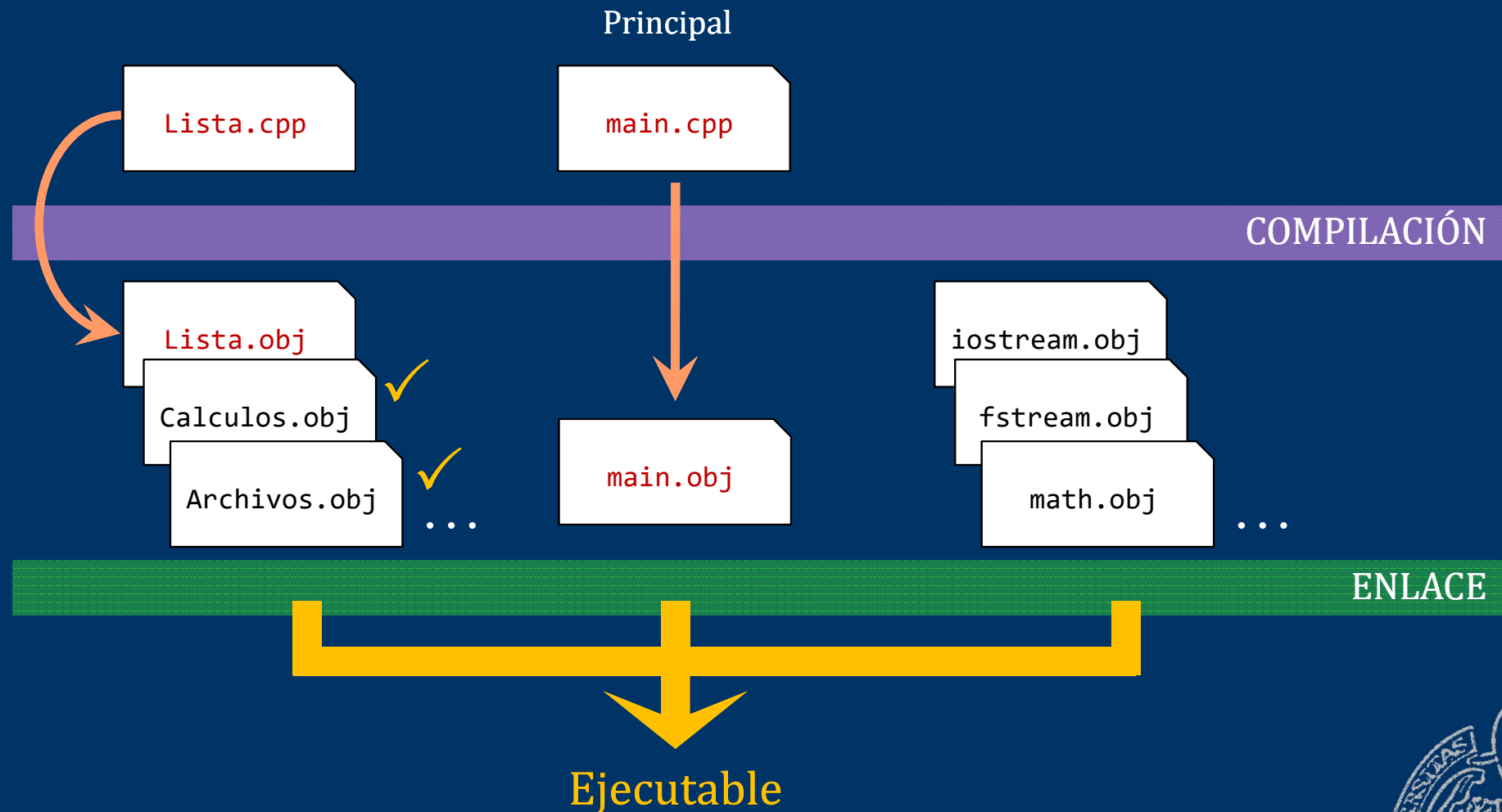
Compilación separada

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



Compilación separada

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



Interfaz frente a implementación

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**.

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



Interfaz frente a implementación

Interfaz: Declaraciones de datos y prototipos de subprogramas
¡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 y subprogramas adicionales)

Separamos la interfaz y la implementación en **dos archivos**
(con el mismo nombre XXX.h / XXX.cpp)

Archivos de **cabecera** (header): XXX.h

Se corresponde con la **Interfaz** del módulo.

Archivos de **implementación**: XXX.cpp

Se corresponde con la **Implementación** de la interfaz del módulo.



Uso de módulos

- ✓ Para compilar se necesita sólo el archivo de cabecera
- ✓ Durante el enlace se adjunta el código objeto del módulo obtenido del cpp

Lista.h

```
const int TM = ...;

typedef struct {
    double elem[TM];
    int cont;
} tLista;

void init(tLista&);
bool insert(tLista&, double);
bool delete(tLista&, int);
int size(const 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
Librería

- ✓ Si otro módulo (o el programa principal) quiere usar algo de una biblioteca beberá incluir el archivo de cabecera con la directiva `#include`
main.cpp

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

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



Uso de módulos

La directiva `#include` añade las declaraciones del archivo de cabecera en el código:

Si incluimos 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`.

`main.cpp`

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

El compilador tiene que comprobar si el código de `main.cpp` hace un uso correcto de la lista. Para ello necesita las declaraciones y prototipos (interfaz)

Preprocesador

`Lista.h`

```
const int TM = ...;

typedef struct {
    double elem[TM];
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(tLista);

void sort(tLista&);
```

`main.cpp`

```
const int TM = ...;

typedef struct {
    double elem[TM];
    int cont;
} tLista;

void init(tLista&);

bool insert(tLista&, double);

bool delete(tLista&, int);

int size(const tLista&);

...
```



Uso de módulos

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

También necesita conocer sus propias declaraciones:

Un archivo .cpp debe comenzar incluyendo su archivo .h

Lista.h

```
const int TM = ...;

typedef struct {
    double elem[TM];
    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++;
    }
}
```

Lista.obj

```
00101110101011001010010010101
0010101001010101111010101000
101001010101010100101010101
01100101010101010101010101001
010101010100000101010101101
01001010101010101000010101011
11001010101010111100110010101
011010101010010010101001111
001010101001010100101010010
101001010100101000010011110
1001010110010101001010100
1010101010100101010010101
0100001010111001010010100
011101011101001101010100101
010111111010101100110101011
0000100101001010101010110
```

Cuando se compila el módulo se genera su código objeto (en el archivo .obj).

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



Ejemplo I

Módulo: Imagen RGB (I)

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

Ahora el código estará repartido en tres archivos:

- ✓ `ImagenRGB.h`: archivo de cabecera del módulo
- ✓ `ImagenRGB.cpp`: archivo de implementación del módulo
- ✓ `ImagenMain.cpp`: archivo del programa principal (main) que usa el módulo de la imagen

Tanto `ImagenRGB.cpp` como `ImagenMain.cpp` deben incluir al principio el archivo de cabecera

```
#include "ImagenRGB.h"
```



Ejemplo I

```
// ImagenRGB.h
#include <string>
typedef unsigned short int uint;
typedef unsigned char uint8; // byte
typedef struct {
    uint8 rojo, verde, azul;
} tRGB;

const uint Max_Res = 24; // máximo nº de filas y columnas
typedef struct {
    uint numFilas, numCols; // resolución de la imagen
    tRGB bmp[Max_Res][Max_Res]; // Max_Res*Max_Res colores
} tImagen;

bool cargar(tImagen & img, std::string const& nombre);
// requieren la librería string
bool guardar(tLista const& lista, std::string const& nombreArchivo);
bool buscar(tImagen const& bmp, tRGB const& color);
...
```

Dependencia
entre
módulos



Ejemplo I

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

bool cargar(tImagen & img, std::string const& nombre) {
    ifstream flujo;
    flujo.open(nombre);
    if (!flujo.is_open()) return false;
    else {
        ...
        flujo.close();
        return true;
    }
}

// requieren la librería fstream
bool guardar(tLista const& lista, std::string const& nombreArchivo) {
    ofstream flujo;
    ...
}
...
```

Dependencia
entre
módulos



Ejemplo I

Programa principal

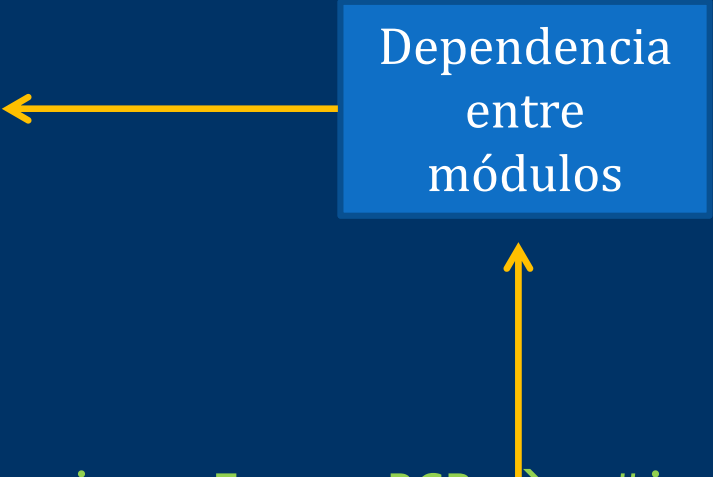
```
// ImagenMain.cpp (main)
#include <iostream>
#include <string>
#include "ImagenRGB.h"
using namespace std;

...

int main() {
    tImagenRGB imagen; // requiere ImagenRGB → #include
    string archivo;     // requiere string
    cout << "Introduce el archivo: "; // requieren iostream
    cin >> archivo;

    if (!cargar(imagen, archivo)) cout<< "Error al cargar el ...";
    else {
        ...
        guardar(imagen, archivo);
    }
}
```

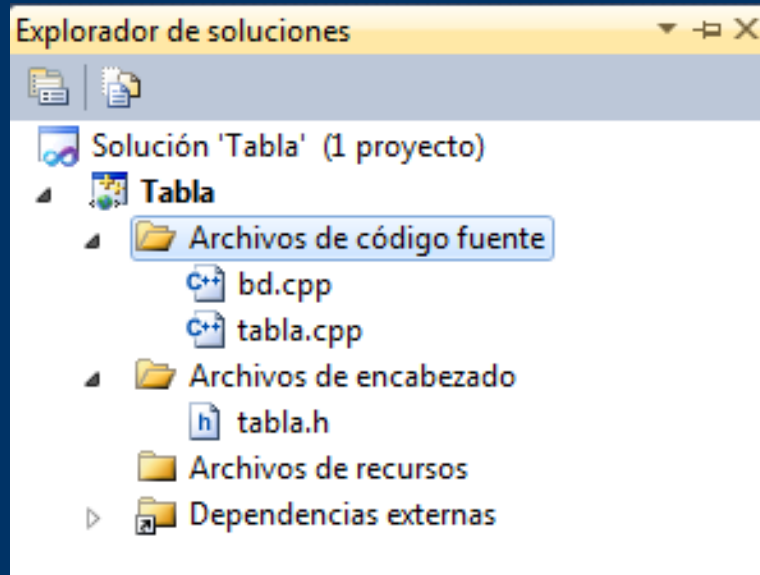
Dependencia entre módulos



El Proyecto

Compilación de programas multiarchivo

Proyecto Visual Studio C++: Los archivos de cabecera y de implementación que forman el proyecto se muestran organizados en:



Archivos de código fuente y
Archivos de encabezado.

Con la opción *Compilar* se compilan sólo los archivos del proyecto que han cambiado desde la compilación más reciente.

Con la opción *Recompilar* se compilan todos.

Recuerda que sólo se compilan los .cpp.



El Preprocesador

Directivas: #...

Antes de realizar la compilación se realiza el *preprocesado*.

Interpreta las directivas y genera un único archivo temporal con todo el código del módulo o programa principal. Directiva `#include`:

```
// lista.h

#include <string>
using namespace std;

const int TM = ...;

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

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

...

int menu();
...
```



Cada cosa en su módulo

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 (librerías de funciones matemáticas, de cadenas, etcétera).

A menudo las estructuras de datos contienen otras estructuras:

```
typedef struct {
    uint8 rojo, verde, azul;
} tRGB;

const uint Max_Res = 24;
typedef struct {
    uint numFilas, numCols;
    tRGB bmp[Max_Res][Max_Res];
} tImagen;
```

Una imagen es una matriz de colores.

- ✓ Estructura tRGB
- ✓ Estructura tImagen
(contiene datos de tipo tRGB)

La gestión de cada estructura, en su módulo.



ImagenRGB (II)

```
// ColorRGB.h
#include "Utilidades.h"

typedef struct {
    uint8 rojo, verde, azul; // uint8 se define en utilidades
} tRGB;

bool operator == (tRGB const& c1, tRGB const& c2);
bool operator != (tRGB const& c1, tRGB const& c2);
tRGB operator * (tRGB const& c1, tRGB const& c2);
tRGB operator + (tRGB const& c1, tRGB const& c2);
void mostrar(tRGB const& color);
...
```



Ejemplo II

```
// ColorRGB.cpp
#include "ColorRGB.h" // ColorRGB.cpp implementa ColorRGB.h
using namespace std;
bool operator == (tRGB const& c1, tRGB const& c2) {
    return c1.azul == c2.azul && c1.rojo == c2.rojo && c1.verde == c2.verde;
}
bool operator != (tRGB const& c1, tRGB const& c2) {
    return !(c1 == c2);
}
void mostrar(tRGB const& color) {
    cout << "Rojo: " << uint(color.rojo) << '\n';
    cout << "Verde: " << uint(color.verde) << '\n';
    cout << "Azul: " << uint(color.azul) << endl;
}
tRGB operator * (tRGB const& c1, tRGB const& c2) {
    tRGB rgb;
    rgb.rojo = c1.rojo * c2.rojo; ...
    return rgb;
}
tRGB operator + (tRGB const& c1, tRGB const& c2) { ... }
```

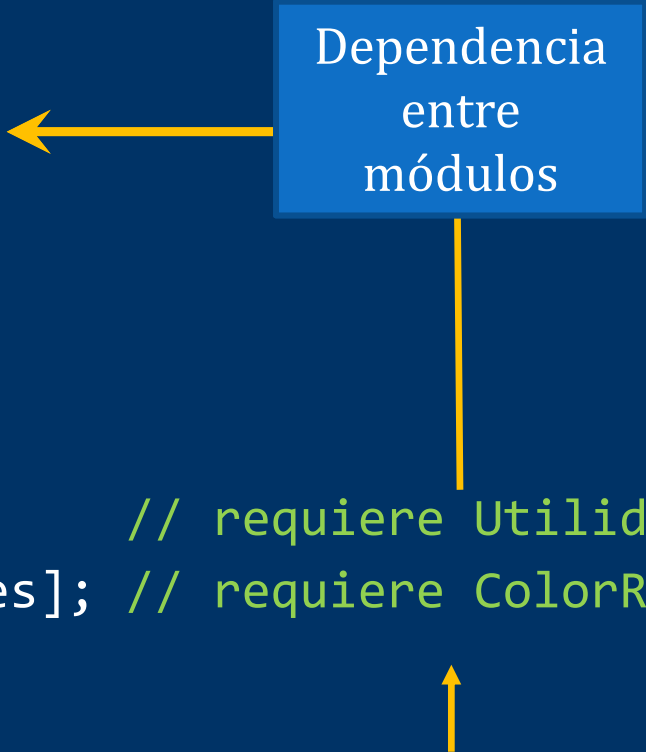


Ejemplo II

```
// ImagenRGB.h
#include <string>
#include "ColorRGB.h"
#include "Utilidades.h"

const uint Max_Res = 24;
typedef struct {
    uint numFilas, numCols;    // requiere Utilidades
    tRGB bmp[Max_Res][Max_Res]; // requiere ColorRGB
} tImagen;

bool cargar(tImagen & bmp, std::string const& nombre);
// requieren la librería string
bool guardar(tImagen const& bmp, std::string const& nombreArchivo);
bool buscar(tImagen const& bmp, tRGB const& color);
...
```



Ejemplo II

```
// ImagenRGB.cpp
#include "lista.h"    ← // ImagenRGB.cpp implementa Imagen.h
#include <fstream>
using namespace std;

bool cargar(tImagen & img,  std::string const& nombre) {
    ifstream flujo;
    flujo.open(nombre);
    if (!flujo.is_open()) return false;
    else {
        ...
        flujo.close();
        return true;
    }
}

// requieren la librería fstream
bool guardar(tLista const& lista,  std::string const& nombreArchivo) {
    ofstream flujo;
    ...
}
...
```



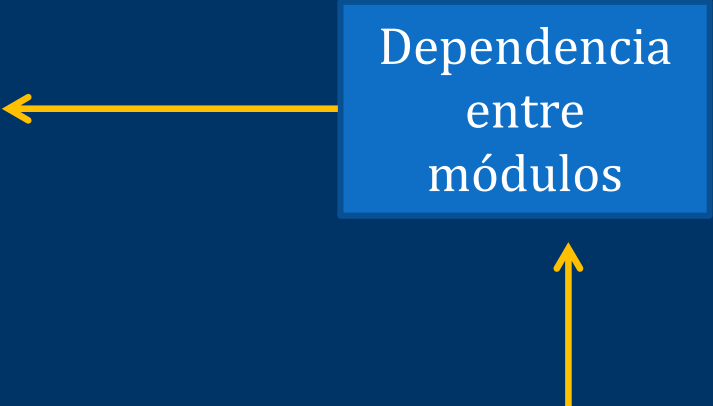
Ejemplo II

Programa principal

```
// ImagenMain.cpp (main)
#include <iostream>
#include <string>
#include "ImagenRGB.h"
#include "ColorRGB.h"
using namespace std;
...

int main() {
    tImagen imagen; // requiere ImagenRGB → #include
    tColor color;    // requiere ImagenRGB
    string archivo;  // requiere string
    cout << "Introduce el archivo: "; // requieren iostream
    cin >> archivo;
    if (!cargar(imagen, archivo)) cout<< "Error al cargar el ...";
    else {
        cout << "Introduce el color: ";
        cin >> color;
        if (buscar(imagen, color) cout << "...";
        else ...
    }
    ...
}
```

Dependencia
entre
módulos

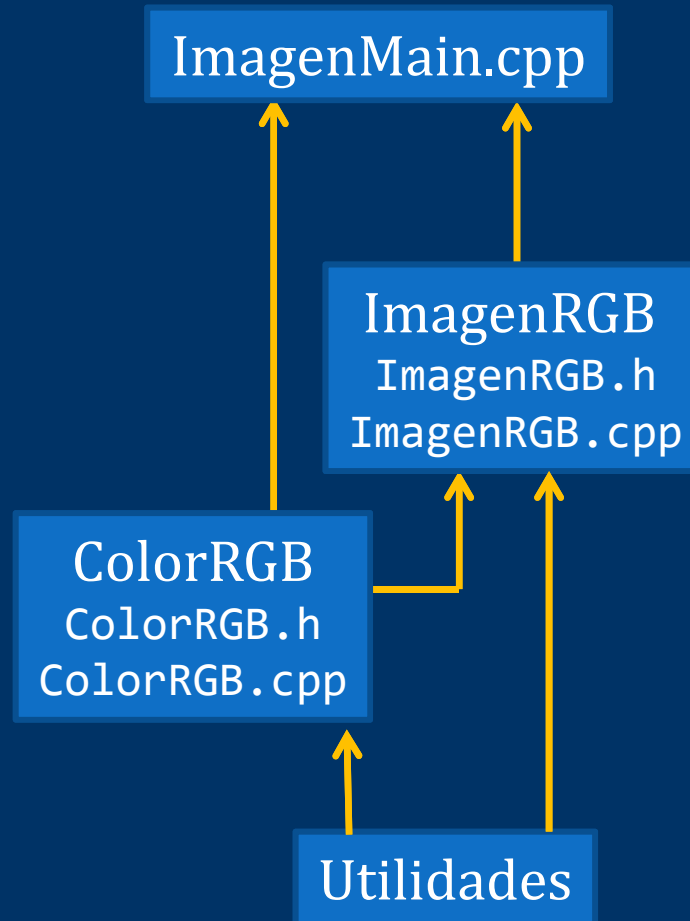


*Este ejemplo tiene errores de compilación
por inclusiones múltiples*



Modularización del programa

Grafo de dependencia entre módulos



El problema de las inclusiones múltiples

Preprocesamiento de `#include`:

```
// ImagenMain.cpp
```

```
...  
#include "ColorRGB.h"  
#include "ImagenRGB.h"
```

```
...  
  
int main()  
{  
    ...  
}  
...
```

```
...  
typedef struct {  
    ...  
} tRGB;  
...
```

```
#include "ColorRGB.h"  
...  
  
typedef struct {  
    uint numFilas, numCols;  
    tRGB bmp[Max_Res][Max_Res];  
} tImagen;  
...
```

```
...  
typedef struct {  
    ...  
} tRGB;  
...
```



El problema de las inclusiones múltiples

Preprocesamiento de `#include`:

```
...
//#include "ColorRGB.h"
...
typedef struct {
} tRGB;
...

//#include "ImagenRGB.h"
//#include "ColorRGB.h"
...
typedef struct {
} tRGB;
...

typedef struct {
    uint numFilas, numCols;
    tRGB bmp[Max_Res][Max_Res];
} tImagen;
...

...

int main()
{
    ...
}

...
```

`// ImagenMain.cpp`

*Error de compilación:
¡Identificador duplicado!*



Compilación condicional

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

Se usan en conjunción con la directiva `#define X`, que define un identificador (X)

```
#ifdef X
... // Código if
[#else
... // Código else
]
#endif
```

```
#ifndef X
... // Código if
[#else
... // Código else
]
#endif
```

Suponiendo X definido:

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.



Compilación condicional

ImagenRGB.cpp incluye ColorRGB.h y

ImagenMain.cpp incluye ColorRGB.h → ¡Identificadores duplicados!

Cada módulo debe incluirse una y sólo una vez.

Protección frente a inclusiones múltiples (en los .h):

```
#ifndef X
#define X
... // Declaraciones del módulo
#endif
```

IMPORTANTE: 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, ...



ImagenRGB (III)

// ColorRGB.h

```
#ifndef COLORRGB_H
#define COLORRGB_H

#include "Utilidades.h"

typedef struct {
    uint8 rojo, verde, azul;    // uint8 se define en utilidades
} tRGB;

bool operator == (tRGB const& c1, tRGB const& c2);
bool operator != (tRGB const& c1, tRGB const& c2);
tRGB operator * (tRGB const& c1, tRGB const& c2);
tRGB operator + (tRGB const& c1, tRGB const& c2);
void mostrar(tRGB const& color);
...

#endif
```



Ejemplo III

```
// ColorRGB.cpp
#include "ColorRGB.h"    ← // ColorRGB.cpp implementa ColorRGB.h

using namespace std;
bool operator == (tRGB const& c1, tRGB const& c2) {
    return c1.azul == c2.azul && c1.rojo == c2.rojo && c1.verde ==
c2.verde;
}
bool operator != (tRGB const& c1, tRGB const& c2) {
    return !(c1 == c2);
}
void mostrar(tRGB const& color) {
    cout << "Rojo: " << uint(color.rojo) << '\n';
    cout << "Verde: " << uint(color.verde) << '\n';
    cout << "Azul: " << uint(color.azul) << endl;
}
tRGB operator * (tRGB const& c1, tRGB const& c2) {
    tRGB rgb;
    rgb.rojo = c1.rojo * c2.rojo; ...
    return rgb;
}
tRGB operator + (tRGB const& c1, tRGB const& c2) { ... }
```



Ejemplo III

```
// ImagenRGB.h
```

```
#ifndef IMAGENRGB_H  
#define IMAGENRGB_H
```

```
#include <string>  
#include "ColorRGB.h" ←  
#include "Utilidades.h"
```

```
const uint Max_Res = 24;  
typedef struct {  
    uint numFilas, numCols;  
    tRGB bmp[Max_Res][Max_Res];  
} tImagen;
```

```
bool cargar(tImagen & bmp, std::string const& nombre);  
bool guardar(tImagen const& bmp, std::string const& nombreArchivo);  
bool buscar(tImagen const& bmp, tRGB const& color)  
...  
#endif
```



Ejemplo III

```
// ImagenRGB.cpp
#include "lista.h"      ← // ImagenRGB.cpp implementa Imagen.h
#include <fstream>
using namespace std;

bool cargar(tImagen & img,  std::string const& nombre) {
    ifstream flujo;
    flujo.open(nombre);
    if (!flujo.is_open()) return false;
    else {
        ...
        flujo.close();
        return true;
    }
}
// requieren la librería fstream
bool guardar(tLista const& lista, std::string const& nombreArchivo) {
    ofstream flujo;
    ...
}
...
```




```
// ImagenMain.cpp (main)
```

```
#include <iostream>
#include <string>
#include "ImagenRGB.h"
#include "ColorRGB.h"
using namespace std;
...
```



¡Ahora ya puedes compilarlo!

```
int main() {
    tImagen imagen; // requiere ImagenRGB → #include
    tColor color;    // requiere ImagenRGB
    string archivo;  // requiere string
    cout << "Introduce el archivo: "; // requieren iostream
    cin >> archivo;
    if (!cargar(imagen, archivo)) cout<< "Error al cargar el ...";
    else {
        cout << "Introduce el color: ";
        cin >> color;
        if (buscar(imagen, color) cout << "...";
        else ...
    }
    ...
}
```



Ejemplo III

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

```
// ImagenMain.cpp (main)
```

```
#include <iostream>
```

```
#include "ColorRGB.h"
```

```
#include "ImagenRGB.h"
```

```
using namespace std;
```

```
...
```

```
int main() {
```

```
...
```

```
}
```

```
...
```

```
#ifndef COLORRGB_H
```

```
#define COLORRGB_H
```

```
...
```

```
typedef struct {
```

```
...
```

```
} tRGB;
```

```
...
```

```
#endif
```

COLORRGB_H no se ha definido todavía



Ejemplo III

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

```
// ImagenMain.cpp (main)
```

```
#include <iostream>
```

```
#define COLORRGB_H
...
typedef struct {
    ...
} tRGB;
...
```

```
#include "ImagenRGB.h"
```

```
using namespace std;
```

```
int main() {
    ...
}
...
```

```
#ifndef IMAGENRGB_H
#define IMAGENRGB_H
...
#include "ColorRGB.h"
...
typedef struct {
    ...
} tImagen;
...
```

IMAGENRGB_H no se ha definido todavía



Ejemplo III

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

```
// ImagenMain.cpp (main)
```

```
#include <iostream>
```

```
#define COLORRGB_H
```

```
...  
typedef struct {
```

```
...  
} tRGB;
```

```
...
```

```
#define IMAGENRGB_H
```

```
#include "ColorRGB.h"
```

```
...
```

```
using namespace std;
```

```
int main() {
```

```
...
```

```
}
```

```
...
```

```
#ifndef COLORRGB_H
```

```
#define COLORRGB_H
```

```
...
```

```
typedef struct {
```

```
...
```

```
} tRGB;
```

```
...
```

```
#endif
```

¡COLORRGB_H ya se ha definido!



El problema de los círculos de inclusiones

- 3 módulos que generan un ciclo (en los .h):

```
//Unit1.h
#ifndef UNIT1_H
#define UNIT1_H
#include "Unit2.h"
... //se usa Unit2
#endif
```

```
//Unit2.h
#ifndef UNIT2_H
#define UNIT2_H
#include "Unit3.h"
...//no se usa Unit3
#endif
```

```
//Unit3.h
#ifndef UNIT3_H
#define UNIT3_H
#include "Unit1.h"
... // se usa Unit1
#endif
```

- Solución: incluir solo los módulos necesario en el archivo que se necesitan

```
//Unit2.h
#ifndef UNIT2_H
#define UNIT2_H
... //no se usa Unit3
#endif
```

```
//Unit2.cpp
#include "Unit2.h"
#include "Unit3.h"
... // se usa Unit3
```

- Solución: ...



Espacios de nombres

Agrupaciones lógicas de declaraciones

Un *espacio de nombres* permite agrupar entidades (tipos, variables, funciones) bajo un nombre distintivo.

Forma de un espacio de nombres:

```
namespace nombre {  
    // Declaraciones  
}
```

Por ejemplo:

```
namespace miBMP24 {  
    const uint8 MAX_VALOR = 255;  
    typedef struct { ... } tRGB;  
}
```

Se declaran los identificadores tRGB y MAX_VALOR dentro del espacio de nombres bmpEspacio



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:

```
miBMP24::MAX_VALOR  
miBMP24::tRGB
```

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 miBMP24 {  
    const uint8 MAX_VALOR = 255;  
    typedef struct {uint8 r,g,b;} tRGB;  
}
```

```
namespace miBMPfloat {  
    const float MAX_VALOR = 1;  
    typedef struct {float r,g,b;} tRGB;  
}
```

Ahora se distingue entre `miBMP24::tRGB` y `miBMPfloat::tRGB`



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:

```
using namespace std;
```

```
namespace miBMP24 {  
    const uint8 MAX_VALOR = 255;  
    typedef struct {uint8 r,g,b;} tRGB;}  
}
```

```
namespace miBMPfloat {  
    const float MAX_VALOR = 1;  
    typedef struct {float r,g,b;} tRGB;  
}
```

```
int main() {  
    using namespace miBMP24;  
    cout << MAX_VALOR << endl;  
    cout << miBMPfloat::MAX_VALOR << endl;  
    return 0;  
}
```

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

// MAX_VALOR es miBMP24::MAX_VALOR
// espacio de nombres explícito

255

1



Calidad y reutilización del software

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 proyectos 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.

Debemos desarrollar el software pensando en su posible reutilización.

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






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.

