2 Diseño Modular

Contenido

1. Introducción.

Niklaus Wirth definió la programación como la suma de dos tareas: el diseño del algoritmo y el diseño de la estructura de datos:

Programa = Algoritmo + Estructura de Datos

El concepto de **programación modular** tiene diversas interpretaciones en función del concepto de **módulo** y de la tarea en la que se enfatice:

1. Énfasis en el procesamiento:
   a. **Módulo** como algoritmo autocontenido, es decir, un **subprograma**. Estos módulos son fruto de la descomposición de un problema al aplicar el diseño descendente (divide y vencerás). Dicho módulo puede ser diseñado "independientemente" del ámbito en el que va a ser usado. Los procedimientos y las funciones son los mecanismos más comunes que ofrecen los lenguajes de programación para permitir este tipo de modularidad.
   b. **Módulo** como agrupación de subprogramas relacionados lógicamente entre sí. En función de esta relación se pueden distinguir dos situaciones:
      i. ***Módulo de programa***: Agrupación de un programa principal y los subprogramas necesarios para resolver un determinado problema.
      ii. ***Módulo de biblioteca***: Agrupación de diferentes subalgoritmos diseñados bajo un ámbito de aplicación común. Estas bibliotecas son utilizadas por otros módulos.

2. Énfasis en los datos:
    a. **Módulo** como agrupación de una estructura de datos y las operaciones que la gestionan. Este tipo de módulos deben ser utilizados por otros, por lo que también se les puede aplicar la denominación de bibliotecas.
        i. Tipos transparentes: Se puede acceder a la estructura interna de los datos desde otro módulo.
        ii. Tipos Abstractos de Datos: No se puede acceder a la estructura interna de los datos desde otro módulo. Sólo se puede acceder a la estructura de los datos a través de las funciones definidas en el propio módulo.
        iii. Orientación a Objetos: Tiene propiedades de herencia y polimorfismo.

En los problemas resueltos hasta ahora, un programa siempre ha estado constituido por un único módulo del tipo módulo de programa. En cambio, en este tema abordaremos el diseño y la utilización de **bibliotecas** (tanto para la agrupación de distintos subalgoritmos como para diseñar tipos transparentes). Esto hará posible, por tanto, la construcción de programas complejos formados por la integración de uno o varios módulos de biblioteca y un módulo de programa.

En realidad, nuestros módulos de programas construidos hasta ahora, han utilizado módulos de biblioteca proporcionados por el sistema para realizar diversas tareas (stdio, string, etc.). Ahora se trata de diseñar nuestras propias bibliotecas, con objeto de dividir la solución algorítmica a un determinado problema en diferentes entidades que **agrupen partes lógicamente relacionadas**, al mismo tiempo que proporcionen un mecanismo adecuado de **reutilización de código**.

2. Ventajas de la modularización de programas.

Escribir un programa como una colección de módulos ofrece muchas ventajas:

- Los módulos son una herramienta potente para **crear grandes programas**. Un programador puede diseñar un programa grande como un conjunto de módulos, relacionados entre sí mediante interfaces definidas apropiadamente. Escribir y depurar un programa es más fácil porque el programador puede trabajar un módulo cada vez, usando servicios facilitados por otros módulos pero ignorando los detalles de su funcionamiento (principio de **abstracción**). Este tipo de **diseño modular** es particularmente necesario cuando el programa se está desarrollando entre un conjunto de programadores.
- Ayudar a que un programa sea más **fácil de modificar**. Puesto que los detalles de implementación de un módulo están ocultos, pueden ser modificados sin afectar al resto de los módulos, esto es, no es necesario efectuar modificaciones en el código de otros módulos y por lo tanto no hay que recompilar el sistema aunque, como es obvio, se debe enlazar de nuevo.
- Hacer los **programas más portables**. El programador puede ocultar los detalles dependientes de la máquina en un módulo, de forma que cuando se transporte a otro ordenador sólo debe preocuparse de modificar dichos aspectos.
- Hacer posible la **compilación separada**. El programa se divide en diferentes porciones que el compilador puede procesar separadamente. De esta forma, un cambio en un módulo únicamente requiere volver a compilar dicho módulo, no el programa completo. Esta ventaja se aprecia cuando el número de módulos que forman un programa llega a ser grande.

- Permiten desarrollar bibliotecas con **código reutilizable**. Ello conlleva no sólo un ahorro de trabajo, sino además un aumento en la fiabilidad del programa, pues dichas bibliotecas están probadas, cosa que no ocurriría si hubiera que recodificarlas.
- El código que se genera es de **más fácil comprensión** ya que se halla dividido en unidades pequeñas, autocontenidas y diseñadas independientemente del contexto de utilización. Por dichos motivos, es usual también que esté **mejor documentado**.

3. Criterios de modularización.

No existen algoritmos formales para determinar cómo descomponer un problema en módulos. Es una labor subjetiva. Existen algunos criterios que pueden guiarnos a la hora de modularizar:

- **Acoplamiento**
  El objetivo de la modularidad es obtener software que sea manejable, de modo que una modificación futura afecte a unos pocos módulos. Para conseguir esto es necesario maximizar la independencia de los módulos. Esto puede parecer contrario a la idea de que es imprescindible mantener algún tipo de conexión entre los módulos para que formen un **sistema coherente**. Por **acoplamiento** se entiende el grado de interconexión entre los módulos. El objetivo será, por tanto, maximizar la independencia de los módulos, es decir, minimizar el acoplamiento.
  El acoplamiento entre módulos se da de varias maneras: de **control** y de **datos**. El **acoplamiento de control** implica la transferencia del control de un módulo a otro (ej., llamada/retorno de subprogramas). El **acoplamiento de datos** se refiere al compartimiento de datos entre los módulos. Las herramientas que proporcionan los lenguajes de programación para implementar el acoplamiento de datos son las listas de parámetros de los subprogramas.
  Los **efectos laterales** se definen como un acoplamiento implícito (no deseado y que puede provocar errores) que se obtiene al utilizar datos globales dentro de varios módulos.

- **Cohesión**
  Tan importante como minimizar el acoplamiento es maximizar los vínculos internos en cada módulo. La **cohesión** se define como el grado de interrelación entre las partes internas de un módulo. Los módulos deben realizar acciones concretas y bien definidas. Mezclar varias acciones o operaciones dentro del mismo módulo disminuye su cohesión y puede dificultar su modificabilidad en futuras revisiones.
  Existen dos tipos de cohesión: **cohesión lógica**, que consiste en agrupar dentro del mismo módulo elementos que realizan operaciones de similar naturaleza (es una cohesión débil); **cohesión funcional**, que consiste en que todas las partes del módulo están encaminadas a realizar una sola actividad (cohesión más fuerte).

4. Módulos de Biblioteca.

Los módulos de biblioteca se usan para exportar e importar recursos a otros módulos. Un módulo de biblioteca consta de dos partes, la **parte de definición** y la **parte de implementación**.

Definición del módulo.

La parte de definición (o interfaz) de un módulo de biblioteca contiene las definiciones de constantes y tipos y las declaraciones de subprogramas (cabeceras seguidas de punto y coma, también denominadas prototipos) que el módulo exporta. En definitiva, se establece todo aquello que puede ser posteriormente utilizado por parte de otro módulo, ya sea un módulo de programa o bien otra biblioteca.

La especificación del interfaz se almacena en un fichero con extensión `".h"`. Este tipo de fichero se denomina también ***fichero de cabecera*** (*header* en inglés).

Implementación del módulo.

La implementación de un módulo de biblioteca contiene las definiciones de los subprogramas (implementación) declarados en el interfaz, así como cualquier otra declaración y/o definición necesaria para realizar la implementación. La implementación se almacena en un fichero con extensión `".c"` o `".cpp"`. Como norma, el nombre del fichero de implementación coincidirá con el nombre del fichero de cabecera.

El fichero de implementación incluirá al principio el fichero cabecera del módulo para poder acceder a las definiciones y declaraciones realizadas en el interfaz. En esta inclusión el nombre del fichero se enmarca entre comillas dobles en lugar de entre paréntesis angulares como ocurre para la inclusión de bibliotecas del sistema.

Utilización de Bibliotecas.

Para que un módulo de biblioteca pueda ser utilizado por parte de otro módulo, este último debe incluir el fichero cabecera de la biblioteca, donde están las definiciones y declaraciones necesarias para su utilización. Si se trata de una biblioteca del sistema la inclusión se realizará de la siguiente forma:

```
#include <nombre_fich_cabecera>
```

mientras que en otro caso la forma de la inclusión será:

```
#include “nombre_fich_cabecera”
```

De esta segunda forma, el compilador buscará el fichero cabecera en el directorio de trabajo actual.

Ejemplo: Los parámetros de las funciones trigonométricas en la biblioteca <math.h> deben ser expresados en radianes, no en grados. Los programas que miden ángulos en grados necesitan convertir de grados a radianes y viceversa (llamaremos a estas funciones gradosAradianes y radianesAgrados). Debido a que estas funciones pueden ser útiles en más de un programa, tiene sentido crear un módulo de biblioteca llamado conversiónAngulos que exporte las dos funciones. El fichero de cabecera sería:

conversionAngulos.h

```
/* declara funciones para la conversion entre angulos */
#ifndef _conversionAngulos_h_    // para evitar inclusión duplicada
#define _conversionAngulos_h_

/* convierte un ángulo de grados a radianes */
double gradosAradianes (double grad);
/* convierte un ángulo de radianes a grados */
double radianesAgrados (double rad);

#endif
```

El fichero de cabecera especifica qué servicios proporciona el módulo, en nuestro caso, las funciones gradosAradianes y radianesAgrados.

Las líneas:
#ifndef _conversionAngulos_h_
#define _conversionAngulos_h_
#endif

son directivas de compilación, es decir, no son sentencias ejecutables sino información adicional que se le proporciona al compilador. En este caso, lo que se está diciendo es:

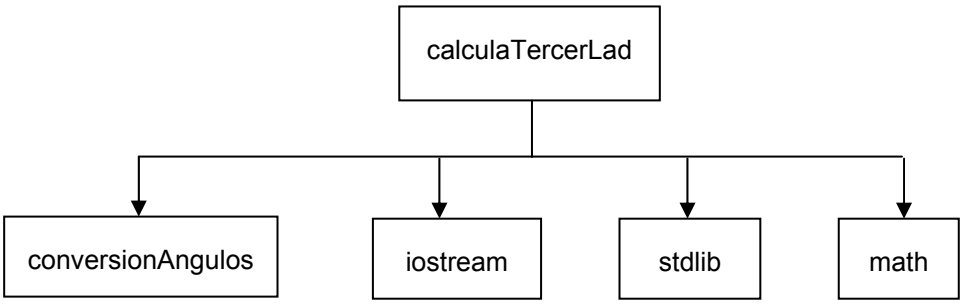
> *Si no está definida la "macro" _conversionAngulos_h_ defínela y también las cabeceras.*
> *Si ya lo estaba no hagas nada (puesto que no hay nada después de la directiva #endif).*

Esta macro suele consistir en el nombre del fichero pero empezando y acabando por '_' y sustituyendo el '.' del nombre del fichero por otro '_ '.

Este tipo de sentencias se suele incluir en todos los ficheros de cabecera para impedir que se produzcan errores de compilación cuando un fichero de cabecera es importado dos veces por el mismo módulo (esto suele ocurrir cuando un módulo importa un fichero de cabecera que a su vez importa otro fichero de cabecera). De esta forma, la segunda vez que se incluye el fichero de cabecera el compilador se salta todas las definiciones puesto que ya está declarada la macro correspondiente.

A un módulo que importa una entidad exportada por un módulo de biblioteca se le denomina *cliente* del módulo de biblioteca. El siguiente programa, calculaTercerLado, es un cliente del módulo de biblioteca conversionAngulos, del cual importa la función gradosAradianes (calculaTercerLado es también cliente de los módulos de biblioteca iostream, stdlib y math).

Esto se puede representar gráficamente de la siguiente forma:



Donde las flechas indican “utiliza”. Sin embargo, el uso de las bibliotecas estándar no se suele representar.

Para poder utilizar las funciones que ofrece el fichero de cabecera, el programa cliente debe incluir dicho fichero de cabecera. Esto se hace de forma similar a como lo habíamos hecho hasta ahora con los ficheros de biblioteca estándar como iostream o stdlib pero sustituyendo los paréntesis angulares por comillas dobles. Esto indica al compilador que ese fichero de cabecera es nuestro y que, por tanto, debe buscarlo en otros directorios distintos a los directorios en los que busca los ficheros de cabecera estándar. En general, los ficheros de cabecera se buscan en el mismo directorio donde esté el archivo que los incluya excepto que se diga lo contrario en las opciones de compilación del proyecto.

El programa principal de la aplicación es el que se muestra a continuación:

calculaTercerLado.cpp

```
/*calcula la longitud del tercer lado de un triángulo a partir de la
longitud de los otros dos lados y el ángulo formado entre ellos*/
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include "conversionAngulos.h"

using namespace std;

int main ()
{
 double lado1, lado2, lado3, angulo;
 cout << "Introduce la longitud de un lado ";
 cin >> lado1;
 cout << "Introduce la longitud del otro lado ";
 cin >> lado2;
 cout << "Introduce el angulo que forman en grados ";
 cin >> angulo;

 lado3 = sqrt (lado1*lado1 + lado2*lado2
         -2.0 * lado1*lado2 * cos (gradosAradianes(angulo)));
 cout << "La longitud del lado 3 es " << lado3 << endl;
 return 0;
}
```

Hemos de darnos cuenta de que calculaTercerLado no sabe nada de como funciona gradosAradianes. La parte de implementación de un módulo de biblioteca "llena los huecos" dejados por la parte de definición. Los procedimientos cuyas definiciones aparecen en la parte de definición deben tener declaraciones completas en la parte de implementación. La parte de implementación de conversionAngulos sería:

conversionAngulos.cpp

```
#include "conversionAngulos.h"
const double pi=3.14159;

double gradosAradianes (double grad)
   /* convierte un ángulo de grados a radianes */
{
        return (pi * grad) / 180.0;
}

double radianesAgrados (double rad)
/* convierte un ángulo de radianes a grados */
{
        return 180 * rad / pi;
}
```

Lo primero que se debe hacer al escribir la parte de implementación es incluir el fichero de cabecera mediante la directiva #include "conversionAngulos.h". De esta forma nos aseguramos que todas las definiciones del fichero de cabecera estén accesibles en la parte de implementación. Además de "llenar los huecos" dejados en la parte de definición del módulo, la parte de implementación puede declarar entidades no mencionadas en la parte de definición (en nuestro caso la constante pi). Estas entidades son locales al módulo de implementación: no son visibles fuera del módulo.

Creación de módulos de biblioteca en C++.

Para crear un módulo de biblioteca en C++ utilizamos todos los conceptos vistos anteriormente junto con el concepto de espacio de nombres.

Espacio de nombres.

Un *espacio de nombres* agrupa bajo un mismo ámbito un conjunto de definiciones y declaraciones lógicamente relacionadas entre sí con algún criterio. Sintácticamente, para definir un espacio de nombres se utilizará la palabra `namespace` seguida del identificador que queramos asignarle. A continuación encerramos entre llaves las diferentes declaraciones y definiciones que deseamos agrupar bajo el espacio de nombres.

```
namespace nombre {
declaraciones y definiciones
}
```

Un espacio de nombres es un ámbito, por lo cual, si un nombre se declara dentro de un espacio de nombres podrá ser visible dentro de ese espacio de nombres con las reglas habituales de ámbitos.

Sin embargo, para usar ese nombre fuera del espacio de nombres hay que utilizar el nombre del espacio de nombres seguido de la expresión "::" a la cual seguirá el nombre que queremos utilizar. Esto es muy útil para programas grandes donde puede ser que tengamos varios subprogramas con el mismo nombre. Poner delante el nombre del espacio de nombres (lo que se denomina, cualificar el nombre) nos evitaría entrar en conflictos.

Ejemplo: De esta forma, el ejemplo anterior se escribiría como vemos a continuación:

Fichero de cabecera: Todas las declaraciones y definiciones aparecerán agrupadas bajo el espacio de nombres elegido para el módulo. En general, el nombre del fichero cabecera coincidirá con el nombre del espacio de nombres que contiene.

conversionangulos.h

```
/* declara funciones para la conversión entre ángulos */
#ifndef _conversionangulos_h_     // para evitar inclusión duplicada
#define _conversionangulos_h_

namespace conversionangulos {

double gradosAradianes (double grad);
   /* convierte un ángulo de grados a radianes */
double radianesAgrados (double rad);
/* convierte un ángulo de radianes a grados */
}
#endif
```

Hemos introducido el espacio de nombres conversionangulos para indicar que es el espacio de nombres donde se encuentran las declaraciones del módulo conversionangulos.

Fichero de implementación: se define el mismo espacio de nombres para implementar cada una de las operaciones definidas en el fichero de cabecera. Por norma, el nombre del fichero de implementación coincidirá con el nombre del espacio de nombres que contiene.

conversionangulos.cpp

```
#include "conversionangulos.h"
namespace Mconversionangulos{
const double pi=3.14159;

double gradosAradianes (double grad){
   /* convierte un ángulo de grados a radianes */
       return (pi * grad) / 180.0;
}
double radianesAgrados (double rad){
/* convierte un ángulo de radianes a grados */
       return 180 * rad / pi;
}
}   // fin del módulo
```

Utilización de la biblioteca:

Para poder utilizar las diferentes entidades exportadas por un módulo de biblioteca, será necesario especificar el espacio de nombres en el que se definieron. Esto se puede hacer de diversas formas:

- ✓ Utilización explícita. Se antepone el nombre del espacio de nombres seguido de la notación "::" a la entidad que se desea utilizar. A esta notación se la denomina "cualificación". Por ejemplo, la sentencia que calcula la longitud del tercer lado en el programa anterior:

```
lado3 = sqrt (lado1*lado1 + lado2*lado2
  -2.0 * lado1*lado2 *cos (conversionangulos::gradosAradianes(angulo)));
```

- ✓ Utilización implícita. Se harán visibles sin necesidad de cualificación todas las entidades exportadas por el módulo biblioteca mediante la siguiente directiva:

```
using namespace conversionangulos;
```

También se puede importar sólo alguna de las entidades de la biblioteca, haciendo una declaración de uso mediante la instrucción using:

```
using conversionangulos::gradosAradianes;
```

Con lo cual, ya podemos utilizar la función sin tener que cualificarla. Esto es particularmente útil cuando el nombre que se está usando se utiliza muchas veces.

Nótese que la biblioteca estándar de C++ se encuentra definida dentro del espacio de nombres `std`, de ahí que en todos los programas se utiliza dicho espacio de nombres mediante la notación implícita (`using namespace std;`). El uso de espacios de nombres en C++ nos permite usar bibliotecas que tengan funciones con el mismo nombre.

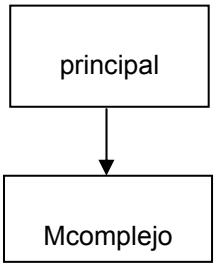
Un módulo biblioteca puede ser utilizado tanto por un módulo de programa como por otro módulo biblioteca (tanto su fichero cabecera como su fichero de implementación). Cuando el módulo que utiliza una biblioteca es un módulo de programa o uno de implementación, se optará por una utilización implícita del espacio de nombres, por su mayor comodidad. Sin embargo, si un módulo utiliza dos bibliotecas diferentes en las que sus espacios de nombres contienen alguna entidad cuyo nombre coincide en ambas (colisión), será necesario utilizar mediante cualificación dicha entidad.

En cambio, si la utilización de la biblioteca se realiza en un fichero de cabecera, es preferible usar la utilización explícita del espacio de nombres, con objeto de restringir el uso sin cualificar de las entidades del espacio de nombres por parte de otros módulos (potencialmente muchos) que incluyan este fichero cabecera. Esto es también aplicable al uso del espacio de nombres estándar `std`.

Puede ocurrir que, dentro de un mismo programa, un fichero cabecera sea incluido varias veces de forma indirecta. Para evitar los problemas de duplicación de definiciones que surgirían, se usan las "guardas" de fichero de cabecera vistas anteriormente (con `#ifndef` ....).

Ejemplo: El módulo Mcomplejo.

A continuación se muestra un programa que utiliza un módulo de biblioteca denominado Mcomplejo que exporta el tipo de datos complejo así como algunas operaciones que se pueden realizar con este tipo de datos. La estructura del programa sería la siguiente:



La parte de definición del módulo complejo (Mcomplejo.h) sería:

```
// declara el tipo de datos Complejo y algunas operaciones
#ifndef _Mcomplejo_h_
#define _Mcomplejo_h_
namespace Mcomplejo{

 struct Complejo {
         float real, imag;
 };         // números complejos
 //lee un número complejo de teclado
 void leer (Complejo& c);
 // escribe un número complejo por pantalla
 void escribir (Complejo x);
 // asigna a c el número complejo formado por real e imag
 void crear (Complejo& c, float real, float imag);
 // suma x + y asignado el resultado a res
 void sumar (Complejo& res, Complejo x, Complejo y);
 // resta dos números complejos
 void restar (Complejo& res, Complejo x, Complejo y);

}
#endif
```

La parte de implementación (Mcomplejo.cpp) quedaría como:

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

// implementa las operaciones sobre complejos

namespace Mcomplejo{

//lee un número complejo de teclado
void leer (Complejo& c){
  cin >> c.real >> c.imag;
}
// escribe un número complejo por pantalla
void escribir (Complejo x){
  cout << x.real << " + " << x.imag << "i";
}
// asigna a c el número complejo formado por real e imag
void crear (Complejo& c, float real, float imag){
  c.real = real;
  c.imag = imag;
}
// suma x + y asignado el resultado a res
void sumar (Complejo& res, Complejo x, Complejo y){
  res.real = x.real + y.real;
  res.imag = x.imag + y.imag;
}
// resta dos números complejos
void restar (Complejo& res, Complejo x, Complejo y){
  res.real = x.real - y.real;
  res.imag = x.imag - y.imag;
}

} // fin del módulo
```

El programa principal (principal.cpp) sería el siguiente:

```
#include <stdlib.h>
#include <iostream>
#include "Mcomplejo.h"

// programa cliente del modulo Mcomplejo.

// Lista de importación:
using Mcomplejo::Complejo;
using Mcomplejo::leer;
using Mcomplejo::sumar;
using Mcomplejo::escribir;
 //  equivalente a using namespace Mcomplejo;
using namespace std;

int main()
{
  Complejo c1, c2, c3;  // declara 3 números complejos
  leer(c1);
  leer(c2);
  sumar (c3, c1, c2);   // c3 = c1 + c2
  escribir (c3);
  cout << endl;

  return 0;
}
```

5. Modularización en Visual C++

En primer lugar creamos un proyecto. Para ello se utiliza la opción "Nuevo Proyecto" del menú "Archivo". Para nuestros ejemplos utilizamos la opción "Proyecto vacío". Elegimos el nombre del proyecto y el directorio donde se desea guardar el fichero del proyecto y pulsaremos Aceptar. Este fichero es el que usa el compilador para guardar toda la información sobre el proyecto (ficheros fuente que lo componen, opciones de compilación del proyecto, etc.).

Un proyecto contendrá un fichero fuente para el módulo de programa (que contiene la función `main()`) y dos ficheros para cada biblioteca que queramos crear. El módulo de definición de la biblioteca (con extensión .h) se incluirá en el directorio de "ficheros de cabecera" del proyecto y el módulo de implementación de la biblioteca (con extensión .c o .cpp) se incluirá en el directorio de "ficheros fuente" del proyecto.

Una vez creado el proyecto se puede compilar tal como lo habíamos hecho hasta ahora cuando sólo usábamos un fichero fuente. Una vez compilados todos los ficheros, se procede al enlazado automático de los distintos ficheros objeto que se han generado para crear el ejecutable.