2.3.2 Uso de plantillas

Introducción

El objetivo principal del programa es almacenar todos los datos para que el usuario pueda visualizarlos y modificarlos mediante los interfaces adecuados. El programa debe almacenar datos de naturaleza muy distinta, ya que se tiene información muy variada. Debe almacenar toda la información necesaria para reconstruir la infraestructura vial de la ciudad, es decir, toda la información sobre las calles, intersecciones, paradas y recorridos de líneas de autobús, etcétera. Y no solo información estática, sino también información concerniente al movimiento de los vehículos en la ciudad, como por ejemplo, en qué direcciones le está permitido girar a un vehículo cuando llega a una intersección y qué movimientos intermedios debe hacer para ello, cada cuánto tiempo pasa un autobús por una parada, teniendo en cuenta que esto depende de la hora del día, y otros aspectos de diversa índole.

Se puede entender, por tanto, que habrá que dedicar gran parte del esfuerzo a diseñar estructuras de datos básicas (clases de C++) que permitan almacenar esta información. Por ejemplo, si se quisiera guardar cierta información básica sobre un libro se podría definir la siguiente clase:

```
Class libro
{
public:
int alto;
int ancho;
int num_paginas;
};
```

Después se podrían crear objetos de esta clase, cada uno con sus características particulares:

```
libro  milibro;
milibro.alto= 230;
milibro.ancho= 135;
milibro.num_paginas= 1056;

libro sulibro;
sulibro.alto= 250;
sulibro.ancho= 140;
sulibro.num_paginas= 245;
```

Para crear los dos objetos de la clase libro, se ha tenido que añadir en el código fuente del programa las líneas *libro milibro;* y *libro sulibro;*. Sin embargo, será necesario poder crear tantos objetos como sea necesario, pues a priori no es posible saber cuántos se van a tener que almacenar en memoria.

La solución la proporciona la gestión de memoria dinámica. La gestión de memoria dinámica permite reservar memoria para almacenar un objeto de manera que no es necesario crear una instancia de una clase que se corresponda con ese objeto, como

ocurre en el ejemplo de la clase *libro*. Usando memoria dinámica se reserva una zona de memoria donde se almacena el objeto, lo único que hay que hacer es conservar la dirección de memoria donde está almacenado cada objeto que se cree dinámicamente para después poder acceder a él cuando sea necesario. La dirección de memoria donde finaliza la memoria reservada para el objeto no es necesaria porque al conocer de qué clase es instancia el objeto se sabe cuánto espacio ocupa al ser almacenado en memoria.

Además, como se manejarán muchas estructuras de datos distintas, será necesario diseñar estructuras más complejas que agrupen ordenadamente a estas estructuras más básicas, con el objetivo de optimizar los tiempos de búsqueda de un dato concreto y el tiempo necesario para añadir nuevos datos. Además, como ya se ha comentado, no se puede saber a priori el número de objetos (instancias de una clase) que se deben almacenar, por lo que se necesita una estructura que reserve la memoria del sistema dinámicamente y que permita añadir y eliminar objetos según sea necesario. C++ permite crear objetos dinámicamente, es decir, reservando memoria para ellos hasta que dejen de ser útiles, momento en el que se debe liberar dicha memoria.

Existen varias estructuras de datos que cumplen estas características, como los árboles y distintos tipos de listas: enlazada, doblemente enlazada, cola, etcétera. Sin embargo se decide usar listas enlazadas simples. En la lista enlazada, cada elemento (nodo de la lista) es un objeto de una clase a la que hemos añadido una nueva propiedad que es un puntero al siguiente objeto de la lista. Este puntero contiene la dirección de memoria donde está almacenado el siguiente nodo de la lista.

Así se pueden realizar búsquedas eficientemente, sólo hay que recorrer la lista adecuada hasta encontrar el objeto deseado. Si en el ejemplo anterior se hubiese reservado memoria dinámicamente, después de usar el objeto se tendría que haber liberado.

El ejemplo anterior tiene el inconveniente de que si se necesitan tres objetos de la clase *libro* en lugar de dos, el código del programa debe ser diferente, por lo que este esquema no es adecuado para el objetivo que se pretende alcanzar.

Al usar una lista enlazada, se puede ir creando objetos dinámicamente y enlazarlos a la lista. Enlazarlos quiere decir que el que hasta ahora era el último elemento de la lista va a contener un puntero a la dirección de memoria del elemento que se acaba de añadir. De esta manera se tiene acceso a todos los elementos de la lista, y cuando se termine de usarla se pueden liberar todos los elementos de la misma, pues en todo instante se tiene constancia de las direcciones de memoria donde está guardado cada nodo gracias a los punteros. La única modificación que requiere la clase libro para poder almacenar los objetos de la clase libro en una lista es añadir el campo “puntero al siguiente elemento”, como se muestra a continuación:

```
Class libro
{
public:
int alto;
int ancho;
int num_paginas;
libro * siguiente_libro;
```

};

De esta manera se pueden crear dinámicamente múltiples objetos de la clase *libro*, cada uno en una zona de memoria, pero, al estar enlazados en una lista, cualquier nodo (instancia de la clase libro) puede ser accedido o eliminado en cualquier momento ya que el nodo anterior indica en qué zona de memoria está almacenado. Así, una lista enlazada podría representarse de la siguiente manera:

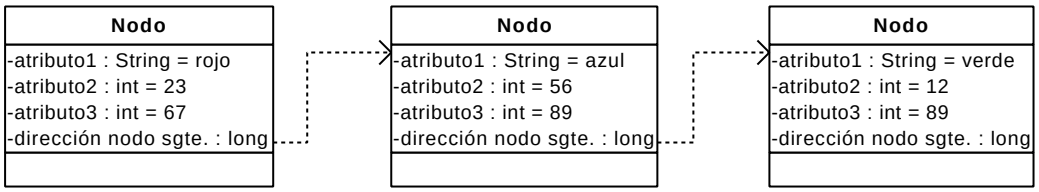

Figura 2.3.2.1: Lista enlazada

Este tipo de estructuras también tiene algunos problemas, pero pueden solucionarse poniendo un poco de atención. En efecto, si se borra un nodo intermedio de la lista, se pierden las direcciones de memoria donde comienzan todos los nodos por detrás de él, con lo que no podremos acceder a dichos objetos.

Este enfoque es fácil de entender si se compara el programa con una base de datos. La base de datos almacena los datos agrupados en tablas. En el programa, cada una de las listas es una tabla. En las bases de datos, cada tabla tiene una serie de campos, y cada fila de la tabla consta simplemente de un valor determinado para cada campo. En el programa, cada una de las propiedades de una clase es uno de esos campos y cada objeto de la lista (instancia de la clase) es una fila de la tabla, con la única particularidad de que contiene un campo adicional que nos indica en qué zona de la memoria está la siguiente fila. Para ver más clara la analogía en la tabla 1 se representa una tabla de una base de datos que contiene la misma información que la anterior lista enlazada:

String	**Int**	**Int**
Rojo	23	67
Azul	56	89
Verde	12	89

Tabla 2.3.2.1: Representación de la información en una base de datos

El entorno de programación Borland dispone de una clase llamada *TList* que permite implementar las listas enlazadas sin necesidad de tener que escribir todo el código necesario para ello y sin tener que añadir un campo adicional para el puntero al siguiente elemento a la clase de los nodos. No obstante, se tendrán que implementar un gran número de listas, debido a la enorme cantidad de datos que deben almacenarse. Además se tendrán que realizar muchas operaciones sobre las listas, como por ejemplo, buscar ciertos elementos, que se tendrían que codificar para cada una de las listas.

Afortunadamente, C++ dispone de un elemento que va a simplificar el trabajo. Se trata de las *plantillas* o *templates*. Mediante las plantillas se pueden crear clases genéricas y funciones genéricas. En una clase o función genérica, el tipo de dato con el que opera la

clase o función queda especificado como un parámetro. Esto permite crear una lista genérica, es decir, crear una sola lista, con las funciones que se necesiten también genéricas, y cuyos elementos pueden ser cualquier clase. Por lo tanto se puede utilizar una función o una clase con distintos tipos de datos sin tener que volver a codificar versiones específicas para cada uno de estos tipos. Para tener una lista de objetos de una cierta clase bastaría con crear una instancia de una clase lista genérica para esa clase en concreto.

Por ejemplo, para tener una lista de la clase libro, habría que crear una instancia (un objeto) de la clase lista genérica pero pasándole como parámetro la clase libro. De esta forma, sólo hay que crear una instancia de la clase genérica para cada lista que se necesite.

Uso de las plantillas

Como ya se ha dicho, se pueden usar plantillas para crear funciones y clases genéricas. En los puntos siguientes se verá en primer lugar la creación de funciones genéricas, por ser más sencillo que las clases genéricas. El formato general de una función que incluye un template sería:

```
template <class Tipo> tipo nombre_funcion (parametros separados por comas)
{
// Código
}
```

Tipo es un nombre que se usa para el tipo de datos que utiliza la función. Este nombre se puede usar dentro de la función. El compilador reemplazará este nombre por el tipo de dato concreto cuando cree una versión específica de la función.

A continuación se incluye un ejemplo sencillo con una función que intercambia los valores de dos variables. Las variables pueden ser de cualquier tipo:

```
#include <iostream.h>

template <class X> void intercambio (X &a, X &b)
{
X aux;
aux=a;
a=b;
b=aux;
// Esta función realiza la operación deseada porque se han pasado las variables
// por referencia, como denota el símbolo &.
}

int main ()
{
int i=5, j=34;
float x=13.5, y=24.6;
```

```
char a='c', b='d';


intercambio (i,j);
intercambio (x,y);
intercambio (a,b);
cout <<"i="<<i<<" j="<<j<<endl;
cout <<"x="<<x<<" y="<<y<<endl;
cout <<"a="<<a<<" b="<<b<<endl;
return (0);
}
```

También pueden crearse funciones genéricas con más de un tipo genérico, simplemente enumerando los tipos separados por comas entre los símbolos < y >. Por ejemplo:

```
template <class A, class B> void unafuncion (A a, B b)
{
// Código
}
```

Como ya se ha comentado, también se pueden crear clases genéricas. Esto es muy interesante ya que se pretende tener una clase lista genérica. El formato general de la declaración de una clase genérica es el siguiente:

```
template <class Tipo> class nombre_clase
{
// propiedades y métodos

};
```

En este caso *Tipo* es el nombre del tipo que se especificará cuando se cree una instancia de la clase. De nuevo se podrían definir varios tipos insertándolos entre los símbolos < y > y separándolos por comas.

Una vez creada la clase genérica, se puede crear una instancia específica de la misma mediante el formato general:

```
nombre_clase <tipo> ob;
```

Aquí *tipo* es el nombre del tipo de los datos con que operará la clase. Las funciones miembro de una clase genérica son funciones genéricas, por definición. Por ello, no es preciso especificar este punto usando *template.*

Puede observarse como las clases genéricas son más engorrosas que las funciones genéricas. El siguiente ejemplo ayudará a clarificar algo más la idea:

Se trata de construir una pila (disciplina FIFO) para cualquier tipo de objeto, es decir, la clase pila será genérica. Este es el código necesario:

```
#include <iostream.h>
int TAMANO=100;

// clase generica pila
template <class Tipo> class pila
{
Tipo pil[TAMANO]; // tabla de TAMANO objetos del tipo de datos Tipo
// es el espacio de almacenamiento de la pila, dependerá de la longitud en bytes que
// ocupe en memoria dicho objeto

int indice;
public:
pila();
~pila();
void introducir (Tipo i);
// puede observarse como introducir es una función genérica, pero al ser miembro de la
// clase genérica pila no necesita especificarse usando template
Tipo obtener();
// lo mismo ocurre con obtener
};

// Ahora sí tendremos que especificar los templates, estamos fuera de la clase

template <class Tipo> pila <Tipo>::pila()
{
indice=0;
cout << "Inicialización de pila"<<endl;
}

template <class Tipo> pila <Tipo>::~pila()
{
cout << "Pila destruida"<<endl;
}

template <class Tipo> void pila<Tipo>::introducir ( Tipo i)
 {
if (indice==TAMANO)
        {
        cout << "La pila está llena"<<endl;
        return;
        }
else
        {
        pil[indice++]=i;
        }
}

template <class Tipo> Tipo pila<Tipo>::obtener()
{
if (indice==0)
```

```
        {
        cout << "La pila está vacía"<<endl;
        }
else
        {
        return pil[--indice];
        }
}
```

Esta clase genérica podría usarse en un programa como el siguiente:

```
int main ()
{
pila<int> a; // crea pila de enteros
pila<double> b; // crea pila de decimales con doble precisión
pila <char> c; // crea pila de caracteres

a.introducir (3);
b.introducir (234.67);
c.introducir ('x');
a.introducir (4);

cout << a.obtener()<<endl;
cout << a.obtener()<<endl;
cout << b.obtener()<<endl;
cout << c.obtener()<<endl;

}
```

Puede observarse como la clase genérica con sus correspondientes funciones genéricas ha eliminado la necesidad de codificar distintas clases pila según el tipo de dato que se quiera almacenar. También ha ahorrado el esfuerzo de codificar cada una de las funciones genéricas *obtener* e *introducir*, e incluso los constructores, para cada tipo de dato.

Uso de las plantillas en la aplicación

En este punto se verá cómo se ha implementado la lista genérica que se necesita y cómo se han creado listas para cada clase a partir de esta clase lista genérica.

La lista genérica se llamó *Lista_inf* y se define así:

```
template <class Y> class Lista_inf
{
 TList *lista;
// Se basará en la clas TList de Borland

public:
 void Crea_lista() {lista=new TList;}
 void Add_elemento (Y inf);
```

```
TList * Lista() {return lista;}
void Insertar_elemento (Y posicion,Y elemento);
void Insertar_elemento_detras (Y elemento,Y posicion);
void Borrar_elemento (Y ele);
void Borrar_elemento_indice (unsigned int indice);
void Borrar_ultimo_elemento ();
void Modificar_elemento (Y ele);
void Modificar_elemento_indice (Y ele, unsigned int i);
void Borra_lista();
void Destruir_lista();
Y Elemento (unsigned int i);
unsigned int Buscar (Y el);
Y Buscar (Y elemento,bool *encontrado);
unsigned int Numero_elementos () {return lista->Count;}
Lista_inf operator= (Lista_inf l);
void Grabar_Informacion(char *nombre);
void Cargar_Informacion(char *nombre);
void Grabar_Informacion_cont(FILE *F1);
void Cargar_Informacion_cont(FILE *F1);
};
```

Se ha definido la clase genérica *Lista_inf*. Todas las funciones que aparecen como funciones miembro (métodos) se pueden aplicar sobre cualquier lista de cualquier clase de objetos que se creen usando la plantilla *Lista_inf*. Algunas de ellas, como se puede observar, son funciones genéricas, ya que usan como parámetro Y.

Una de las clases que se tienen definidas en el código es la clase *Datos_tramos*. Esta clase almacena información relativa a las calles de la ciudad y por tanto es interesante tener una lista de objetos *Datos_tramo*, en concreto un elemento para cada tramo de la ciudad. Hay muchos tipos de datos que es interesante tener almacenados, pero se va a usar *Datos_tramo* para mostrar la implementación de la clase genérica *Lista_inf* en el código del programa.

Para tener una lista de *Datos_tramos* lo único que tenemos que hacer es:

```
template class Lista_inf<Datos_tramos>;
```

A partir de ahora se pueden crear listas de objetos de la clase *Datos_tramos* basadas en la plantilla *Lista_inf*. La que se usa en el código es la siguiente:

```
Lista_inf <Datos_tramos> Lista_Datos_tramos;
```

Como se podrá ver a continuación en el código, se ha creado la lista pero aún no se ha reservado memoria para ella. Esto debe hacerse con la función miembro *Crea_lista.*

Será interesante poder usar las funciones miembro, algunas genéricas, de la clase *Lista_inf* sobre la lista de *Datos_tramos*, ya que ésta es precisamente la potencia que ofrecen las plantillas. A continuación se explican algunas de las funciones más importantes.

En primer lugar, la función *Crea_lista,* la primera que debe ser llamada. Es una función *inline* que se define dentro de la clase, por definición. El código es simplemente:

```
void Crea_lista() {lista=new TList;}
```

Ya se ha creado la lista *Datos_tramos* completamente y se puede empezar a añadir elementos. Hasta ahora el proceso ha sido general, es decir, no ha influido nada el hecho de que los objetos de la lista vayan a ser de la clase *Datos_tramo*. Esto ya no es así en la función que se debe usar para añadir un elemento:

```
template <class Y> void Lista_inf<Y>::Add_elemento (Y elemento)
{
   Y *el;
   el=new Y;
  *el=elemento;
    lista->Add (el);
}
```

Esta función crea un puntero a la clase Y. La clase Y es la que se ha recibido como parámetro, es decir, *Datos_tramos*, porque a esta función se la llamará de la siguiente manera:

```
// Suponiendo que existe Datos_tramos nuevo_tramo;
//Y suponiendo que existe Lista_Datos_tramos  y que hemos llamado a
// Lista_Datos_tramos.Crea_lista();

Lista_Datos_tramos.Add_elemento (nuevo_tramo);
```

A continuación se reserva memoria dinámica para el elemento, de manera que ésta sólo se eliminará cuando se realice una llamada a *free().* Hay funciones de *Lista_inf* que liberan toda la memoria reservada para la lista. Ahora hay que tener cuidado, el siguiente paso es copiar en todos los datos miembros de *el* todos los datos miembros de *elemento.* En realidad se podía haber realizado cualquier otra operación, ya que lo único que se hace es usar la sobrecarga del operador asignación (=). Para ello se sobrecarga el operador = para la clase *Datos_tramos* para que haga lo que se desea. Podrá verse el código correspondiente algo después. Siguiente función:

```
template <class Y> unsigned int Lista_inf<Y>::Buscar (Y elemento)
{
 unsigned int i,numero_elementos;
 bool Salida=false;
 Y e;
       numero_elementos=lista->Count;
       for (i=0;i<numero_elementos && !Salida;i++)
        {
          e=Elemento(i);
          if (e==elemento)
             Salida=true;
```

```
        }
if (Salida)
 return i-1;
else
 return numero_elementos;
}
```

Esta función devuelve el índice dentro de la lista del elemento que se busca, si lo encuentra. Si no, devuelve el número de elementos de la lista. No hay posible confusión entre el número de elementos de la lista y el índice del último elemento porque los elementos se numeran empezando en el cero.

Pero hay una línea interesante, *if (e==elemento).* Lo que hace esta línea es comparar dos objetos de la clase *Datos_tramos*. En realidad, esto es lo que ocurre en este caso, puesto que la función es genérica y lo que realmente ocurre es la que se aplica la sobrecarga del operador == para un objeto de la clase *Datos_tramos*, al igual que se hizo con *Add_elemento* con el operador =. Por tanto, para poder usar esta función con la lista de objetos de la clase *Datos_tramo*, debe estar adecuadamente sobrecargado el operador == para dicha clase. A continuación puede verse cómo se ha hecho:

Habiendo declarado las funciones dentro de la clase *Datos_tramo*:

```
bool Datos_tramos::operator== (Datos_tramos d)
{
 if (cod_linea==d.Codigo())
  return true;
 else
  return false;
}
```

Es decir, dos objetos de la clase *Datos_tramo* son iguales si tienen el mismo valor en la propiedad *cod_linea*. Esto es lógico, ya que *cod_linea* es un identificador único del tramo. Volviendo a la analogía de las bases de datos, sería la Clave Principal (Primary Key).

Y la sobrecarga para el operador asignación (=):

```
Datos_tramos Datos_tramos::operator= (Datos_tramos d)
{
 cod_linea=d.Codigo();
 codigo_funcion=d.Funcion();
 Actualiza_descripcion(d.Descripcion());
 return (*this);
}
```

Copia en el objeto de la clase *Datos_tramos* que aparece a la izquierda del operador las propiedades *cod_linea*, *codigo_funcion* y *Descripcion* del objeto *d* que se le pasa como parámetro, es decir, que aparece en la derecha del operador y devuelve aquel objeto. Éste es el comportamiento deseado para el operador =.

Las funciones que se han visto hasta ahora eran muy interesantes porque ilustran la necesidad de sobrecargar los operadores = y ==, pero para la plantilla *Lista_inf* hay definidas más funciones. A continuación se presenta un resumen con el cometido de cada una de ellas:

- *void Insertar_elemento (Y posicion,Y elemento);*

Inserta el objeto *elemento* en la lista en la posición del objeto *posicion*.

- *void Insertar_elemento_detras (Y elemento,Y posicion);*

Igual que la anterior pero el objeto *elemento* se inserta tras el objeto *posicion*.

- *void Borrar_elemento (Y ele);*

Borra el elemento *ele* de la lista.

- *void Borrar_elemento_indice (unsigned int indice);*

Borra de la lista el elemento que ocupa la posición *indice*. Se recuerda que el primer elemento de la lista tiene índice 0.

- *void Borrar_ultimo_elemento ();*

Borra el último elemento de la lista

- *void Modificar_elemento (Y ele);*

Esta función busca en la lista el elemento *ele* y copia el objeto *ele* en esa posición. Dicho así parece no tener mucho sentido, pero tiene mucha utilidad si se tiene un objeto que ha cambiado alguna de sus propiedades pero no la que se utiliza con el operador == para distinguirlo de otros. Si la sobrecarga del operador == implica comparar todas las propiedades del objeto entonces esta función no tiene sentido. Ejemplo:
El objeto *Tramo* tiene varias propiedades, una de ellas es el código de tramo, que es único para cada tramo (clave primaria) y otras son características del tramo, como longitud, número de carriles, etcétera. En este caso, seguramente será conveniente tener el operador == definido de manera que se obtenga que dos tramos sean iguales si lo es su código de tramo. En este caso, si varían ciertas propiedades de un tramo (excepto, claro está, al código, pues si no sería otro tramo diferente), se pueden actualizar fácilmente todas sus propiedades usando esta función (lo que se hace en realidad es aplicar la sobrecarga del operador =). El comportamiento depende totalmente de la sobrecarga de los operadores == y =.

- *void Modificar_elemento_indice (Y ele, unsigned int i);*

Asigna al objeto que tiene como índice *i* el objeto *ele.* La asignación está determinada por el operador =.

- *void Borra_lista();*

Borra la lista.

- *void Destruir_lista();*

Borra la lista y libera la memoria reservada.

- *Y Elemento (unsigned int i);*

Devuelve el elemento de la lista que tiene como índice *i.*

- *unsigned int Numero_elementos ();*

Devuelve el número de elementos de la lista.

- *Lista_inf operator= (Lista_inf l);*

Sobrecarga el operador asignación para listas. Añade a la lista todos los objetos de la lista l (sin borrar los que ya existían).

- *void Grabar_Informacion(char *nombre);*

Graba toda la información de la lista en el fichero *nombre* en formato binario. En primer lugar guarda el número de elementos de la lista y a continuación cada uno de los objetos de la misma.

- *void Cargar_Informacion(char *nombre);*

Extrae todos los objetos del fichero *nombre* y los añade a la lista, sin borrar los ya existentes. El fichero debe estar en el formato especificado en la explicación de la función *Grabar_Informacion*.

- *void Grabar_Informacion_cont(FILE *F1);*

Es igual que *Grabar_Informacion* pero recibe como parámetro un manejador de fichero en lugar del nombre del mismo.

- *void Cargar_Informacion_cont(FILE *F1);*

Es igual que *Cargar_Informacion* pero recibe como parámetro un manejador de fichero en lugar del nombre del mismo.

Las cuatro funciones que tienen como objetivo cargar los datos desde disco y grabar los datos a disco, *Cargar_Informacion*, *Cargar_Informacion_cont*, *Grabar_Informacion* y *Grabar_Informacion_cont*, se discutirán con más detalle en otra sección debido a la importancia que tienen de cara a poder recuperar el trabajo realizado con la aplicación.