Anexo II

Lenguaje C++

Guion-resumen

1. **Introducción**
2. **La función main ()**
3. **Tipos básicos de datos**
4. **Variables**
5. **Operadores, caracteres especiales, instrucciones, arrays, cadenas de caracteres, punteros, estructuras, uniones, enumeraciones y typedef**
6. **Tipos compuestos**
7. **Asignación dinámica de memoria**
8. **Sentencias de control en C++**
9. **Funciones**
10. **Programación eficiente**
11. **El preprocesador**
12. **Entrada / Salida de datos**
13. **Programación orientada a objetos**
14. **Herencia en C++**
15. **Sobrecarga en C++**
16. **Templates**
17. **Manejo de excepciones**

1. Introducción

El lenguaje de programación C++ se comenzó a desarrollar en 1980 como una ampliación del lenguaje C; de ahí el nombre de C++ que proviene del operador incremento ++. También es conocido como C con clases.

Entre las **características iniciales** que debemos tener en cuenta tenemos:

- Los ficheros fuente terminan con la extensión ***.cpp**
- Los ficheros de encabezado figuran con la extensión ***.h** y se incluyen en el fichero fuente usando la directriz **#include** del preprocesador (como sucedía en el lenguaje C),

  Ejemplo: *#include "iostream.h"* *ó* *#include <iostream.h>*

Nota1. La diferencia entre usar las comillas o usar <> estriba en que si el nombre de la biblioteca está entre comillas se busca el archivo de la manera definida en la implementación (generalmente primero en el directorio actual de trabajo y luego como si se hubiese puesto <>). Y si se usa <> se busca según se haya establecido en la creación del compilador. Esto también es aplicable al lenguaje C.

Nota2. Las líneas que empieza por # son una directiva. En este caso indica que se incluya el fichero «iostream.h», que contiene las definiciones para entrada/salida de datos en C++.

- Los **identificadores** válidos del C++ son los formados a partir de los caracteres del alfabeto (del inglés, no podemos usar ni la ñ ni palabras acentuadas), los dígitos "0 – 9" y el guión bajo "_"; la única restricción es que no podemos comenzar un identificador con un dígito. Hay que señalar que el C++ distingue entre mayúsculas y minúsculas, por lo que Lolo y lolo representan dos cosas diferentes. Hay que evitar el uso de identificadores que solo difieran en letras mayúsculas y minúsculas, porque inducen a error.
- Las **variables** se pueden declarar en cualquier lugar del programa, siempre y cuando tengamos cuidado de usarlas una vez que hayan sido declaradas.
- Los **comentarios** se pueden hacer de dos formas:
  - — Una sola línea (no afecta a la sentencia situada a la izquierda de las //):

    Ejemplo: x=2+3; // este es un comentario de una línea

  - — Un bloque:

    Ejemplo: x=2+3; / comentario de*

    un bloque/*

Los comentarios entre /* y */ pueden tener la longitud que queramos, pero no se anidan, es decir, si escribimos /* hola /* amigos */ de TAI. Los comentarios que comienzan por // solo son válidos hasta el final de la línea en la que aparecen.

- Los **agrupadores** de bloque o de funciones, igual que sucedía en C, son las llaves "{ }".
- Cada **sentencia** debe terminar en un punto y coma ";". Es por eso que en una misma línea se pueden poner varias sentencias, separadas por el ";" o incluso podríamos poner todo el programa en una sola línea.

  Ejemplo: *int x=2; float y=8; float z; z=x+y;*

- El soporte a la **programación modular** en C++ se consigue mediante el empleo de algunas palabras clave y de las directivas de compilación. Lo más habitual es definir cada módulo mediante una cabecera (un archivo con la terminación .h) y un cuerpo del módulo (un archivo con la terminación *.c, .cpp*, o algo similar; depende del compilador). En el archivo cabecera (header) ponemos las declaraciones de funciones, tipos y variables que queremos que sean accesibles desde el exterior y en el cuerpo o código definimos las funciones publicas o visibles desde el exterior, además de declarar y definir variables, tipos o funciones internas a nuestro módulo.

2. La función main ()

La función ***main()*** es la función principal del programa, por donde empieza la ejecución del mismo. En caso de no recibir argumentos se declara de la siguiente forma:

```
void main(void) {
  ......
  ......
}
```

Puede recibir argumentos desde la consola en el momento de la ejecución del programa:

```
void main (int argc, char *argv[]) {
  ......
  ......
}
```

- **argc** indica el número de argumentos que son pasados a la función main.

- "**argv**" es un "array" de caracteres a través de punteros que contiene el valor de los argumentos.

Todos los programas deben tener una función "main()" que es la que se ejecuta al comenzar el programa. Un programa será una secuencia de líneas que contendrán sentencias, directivas de compilación y comentarios. Las sentencias simples se separan por punto y coma, y las compuestas se agrupan en bloques mediante llaves. Las directivas serán instrucciones que le daremos al compilador para indicarle que realice alguna operación antes de compilar nuestro programa, las directivas comienzan con el símbolo # y no llevan punto y coma.

Nota. *El mínimo programa de C++ es: main() { }*

3. Tipos básicos de datos

(Ver tipos de datos en Lenguaje C)

Los tipos elementales definidos en C++ son: char, short, int, long, que representan enteros de distintos tamaños (los caracteres son enteros de 8 bits) float, double y long double, que representan números reales (en coma flotante). Para declarar variables de un tipo determinado escribimos el nombre del tipo seguido del de la variable. Por ejemplo:

int i; double d; char c;

Podemos declarar varias variables de un mismo tipo poniendo el nombre del tipo y las variables a declarar separadas por comas: *int i, j,k.* Además podemos inicializar una variable a un valor en el momento de su declaración: *int i=100*;

Cada tipo definido en el lenguaje (o definido por el usuario) tiene un nombre sobre el que se pueden emplear dos operadores: **sizeof**, que nos indica la memoria necesaria para almacenar un objeto del tipo, y **new**, que reserva espacio para almacenar un valor del tipo en memoria.

C++ solo define un nuevo tipo de dato al lenguaje C que es el tipo de dato "bool" que puede tomar solo dos valores: **true o false**.

Para especificar si los valores a los que se refieren tienen o no signo, empleamos las palabras **signed y unsigned** delante del nombre del tipo (por ejemplo **unsigned int** para enteros sin signo).

El tipo **void** es sintácticamente igual a los tipos elementales pero solo se emplea junto a los derivados y no hay objetos del tipo **void**. Se emplea para especificar que una función no devuelve nada o como base para punteros a objetos de tipo desconocido.

Por ejemplo: **void lolo (void)**; indica que la función lolo no tiene parámetros y no retorna nada.

3.1. Conversiones de tipos

• **Conversiones implícitas**

Cuando trabajamos con tipos elementales podemos mezclarlos en operaciones sin realizar conversiones de tipos, ya que el compilador se encarga de aproximar al valor de mayor precisión.

• **Conversiones explícitas (casting)**

Para indicar la conversión explícita de un tipo en otro usamos el nombre del tipo; por ejemplo, si tenemos "i" de tipo "int" y "j" de tipo "long", podemos hacer *i=(long)j* (sintaxis del C, válida en C++) o *i=long(j)* (sintaxis del C++).

4. Variables

(Ver variables en Lenguaje C).

4.1. Visibilidad y duración de las variables

La visibilidad o ámbito de vida de una variable es la parte del programa en la que esa variable está definida y puede ser utilizada. Podemos clasificar las variables en **locales y globales**:

- **Variables locales** son aquellas que son declaradas dentro de un bloque y son visibles en ese bloque y en los bloques anidados dentro de él. Ocultan a las variables globales con el mismo nombre.
- **Variables globales** son aquellas que son declaradas fuera de cualquier función. Son visibles durante toda la ejecución del programa y se pueden utilizar en todas las funciones del fichero que forma el programa.

La duración o tiempo de vida de una variable hace referencia al tiempo que transcurre entre la creación de una variable y el instante en que es destruida. Si son variables ***auto*** (locales), la variable se crea y se destruye cada vez que pasa por el bloque (esta es la opción por defecto). Si es una variable ***static***, la duración de dicha variable es hasta que finalice el programa.

Las variables existen solo dentro del bloque en el que se definen, es decir, se crean cuando se entra en el bloque al que pertenecen y se destruyen al salir de él. Para acceder a variables que se definen en otros módulos la declaramos en nuestro módulo precedida de la palabra **extern**.

Si queremos que una variable sea local a nuestro módulo la definimos *static*, de manera que es inaccesible desde el exterior de nuestro módulo y además permanece durante todo el tiempo que se ejecute el programa guardando su valor entre accesos al bloque.

Si queremos que una variable no pueda ser modificada la declaramos *const*; tenemos que inicializarla en su declaración y mantendrá su valor durante todo el programa. Estas variables se emplean para constantes que necesitan tener una dirección (para pasarlas por referencia).

Si queremos que una variable sea comprobada cada vez que la utilicemos la declararemos precedida de la palabra **volatile**; esto es útil cuando definimos variables que almacenan valores que no solo modifica nuestro programa.

Podemos intentar hacer más eficientes nuestros programas indicándole al compilador qué variables usamos más a menudo para que las coloque en los registros. Esto se hace declarando la variable precedida de la palabra **register**.

Podemos tener definida una variable local con el mismo nombre que una global; en este caso la local prevalece sobre la global. No obstante si queremos acceder a una variable global en un bloque donde exista una local del mismo nombre, utilizamos el operador "::".

Ejemplo:

```
int x=10;          //variable global
void main( ) {
        int x = 20; //variable local
        int y = ::x; // asigna a y el valor 10
        int t =  x;  // asigna a t el valor 20
}
```

5. Operadores, caracteres especiales, instrucciones, arrays, cadenas de caracteres, punteros, estructuras, uniones, enumeraciones y typedef

(Ver Lenguaje C).

5.1. Tipos derivados

De los tipos fundamentales podemos derivar otros mediante el uso de los siguientes operadores de declaración:

* Puntero & Referencia [ ] Vector (Array) () Función

Ejemplos:

```
int *punt; // puntero a un entero
int v[20]; // vector de 20 enteros
int *punt[20]; // vector de 20 punteros a entero
void f(int j); // función con un parámetro entero
```

int i; // declaración de un entero i

*i = *n; // almacenamos en i el valor al que apunta n*

i = v[2] // almacenamos en i el valor de el tercer elemento de v

*i = *v[2] // almacenamos en i el valor al que apunta el tercer puntero de v*

f(i) // llamamos a la función f y le enviamos el parámetro i

Para **declarar un puntero a un vector** necesitamos paréntesis:

*int *v[20]; // vector de 20 punteros a entero*

*int (*punt)[20] // puntero a vector de 20 enteros*

Al **declarar variables de tipos derivados**, el operador se asocia a la variable, no al nombre del tipo:

int x,y,z; // declaración de tres variables enteras

*int *i, j; // declaramos un puntero a entero (i) y un entero (j)*

*int v[10], *p; // declaramos un vector de 10 enteros y un puntero a entero*

- **Punteros**

Para cualquier tipo X, el puntero a ese tipo es X*. Una variable de tipo X* contendrá la dirección de un valor de tipo X. Los punteros a vectores y funciones necesitan el uso de paréntesis:

*int *punt_i // Puntero a entero*

*char **punt_c // Puntero a puntero a carácter*

*int (*punt_v)[10] // Puntero a vector de 10 enteros*

*int (*punt_f)(float) // Puntero a función que recibe un real y retorna un entero*

La operación fundamental sobre punteros es la de indirección (retornar el valor apuntado por él):

char c1 = 'a'; // c1 contiene el carácter 'a'

*char *p = &c1; // asignamos a p la dirección de c1 (& es el operador referencia)*

*char c2 = *p; // ahora c2 vale lo apuntado por p ('a')*

- **Vectores**

Para un tipo X, X[n] indica un tipo vector con n elementos. Los índices del vector empiezan en 0 y llegan hasta n-1. Podemos definir vectores multidimensionales como vectores de vectores:

int v1[10]; // vector de 10 enteros

*int v2[20][10]; // vector de 20 vectores de 10 enteros (matriz de 20*10)*

Accedemos a los elementos del vector a través de su índice (entre []):

v1[3] = 15; // el elemento con índice 3 vale 15
v2[8][3] = v1[3]; // el elemento 3 del vector 8 de v2 vale lo mismo que v[3]

El compilador no comprueba los límites del vector, es responsabilidad del programador. Para inicializar un vector podemos enumerar sus elementos entre llaves. Los vectores de caracteres se consideran cadenas, por lo que el compilador permite inicializarlos con una constante cadena (pero les añade el carácter nulo). Si no ponemos el tamaño del vector al inicializarlo el compilador le dará el tamaño que necesite para los valores que hemos definido. Ejemplos:

int v1[5] = {0, 1, 2, 3, 4};
char v2[2][3] = {{'t', 'a', 'i'}, {'T', 'A', 'I'} }; // vect. multidimensional
int v3[2] = {1, 2, 3, 4}; // error: solo tenemos espacio para 2 enteros
char c1[5] = {'h','o','l','a','\0'}; // cadena «hola»
char c2[5] = «holita»; // cadena holita
char c3[] = «hola»; // el compilador le da tamaño 5 al vector
char vs[3][] = {«hola», «aprobados», «tai»} // vector de 3 cadenas (3 punteros a carácter)

- **Referencias**

Una referencia es un nombre alternativo a un objeto; se emplea para el paso de argumentos y el retorno de funciones por referencia. X& significa referencia a tipo X. Las referencias tienen **restricciones**:

1. Se deben inicializar cuando se declaran (excepto cuando son parámetros por referencia o referencias externas).
2. Cuando se han inicializado no se pueden modificar.
3. No se pueden crear referencias a referencias ni punteros a referencias.

Ejemplos:

int a; // variable entera
int &r1 = a; // ref es sinónimo de a
int &r2; // error, no está inicializada
extern int &r3; // válido, la referencia es externa (estará inicializada en otro
// módulo)
int &&r4=r1; // error: referencia a referencia

6. Tipos compuestos

Existen cuatro tipos compuestos en C++:

Estructuras	Uniones	Campos de bits	Clases

- **Estructuras**

Las estructuras son el tipo equivalente a los registros de otros lenguajes. Se definen poniendo la palabra **struct** delante del nombre del tipo y colocando entre llaves los tipos y nombres de sus campos. Si después de cerrar la llave ponemos una lista de variables, las declaramos a la vez que definimos la estructura. Si no, luego podemos declarar variables poniendo *struct nombre_tipo* (ANSI C, C++) o *nombre_tipo* (C++).

Ejemplo:

```
struct persona {
        int edad;
        char nombre[50];
} empleado;
struct persona alumno; // declaramos la variable alumno de tipo persona (ANSI C)
persona profesor; // declaramos la variable profesor de tipo persona
persona *p; // declaramos un puntero a una variable persona
```

Podemos inicializar una estructura de la misma forma que un array:

```
persona lolo= {21, «Lolo Lolete»};
```

Para acceder a los campos de una estructura ponemos el nombre de la variable, un punto y el nombre del campo. Si trabajamos con punteros podemos poner -> en lugar de de referenciar el puntero y poner un punto (esto lo veremos en el punto de variables dinámicas):

```
alumno.edad = 20; // el campo edad de alumno vale 20
p->nombre = «Lolo»; // el nombre de la estructura apuntada por p vale «Lolo»
(*p).nombre = «Lolo»; // igual que antes
```

Empleando el operador **sizeof** a la estructura podemos saber cuántos bytes ocupa.

- **Uniones**

Las uniones son idénticas a las estructuras en su declaración (poniendo *union* en lugar de *struct*), con la particularidad de que todos sus campos comparten la misma memoria (el tamaño de la unión será el del campo con un tipo mayor). Es responsabilidad del programador saber qué está haciendo con

las uniones, es decir, podemos emplear el campo que queramos, pero si usamos dos campos a la vez uno machacará al otro. Ejemplo:

```
union codigo {
    int i;
    float f;
} cod;
cod.i = 10; // i vale 10
cod.f = 25e3f; // f vale 25 * 1000, i indefinida (ya no vale 10)
```

Podemos declarar uniones o estructuras sin tipo siempre y cuando declaremos alguna variable en la definición. Si no declaramos variables la estructura sin nombre no tiene sentido, pero la unión permite que dos variables compartan memoria. Ejemplo:

```
struct {
    int i;
    char n[20]
} reg;
union {
    int i;
    float f;
}; // i y f son variables, pero se almacenan en la misma memoria
```

- **Campos de bits**

Un campo de bits es una estructura en la que cada campo ocupa un numero determinado de bits, de forma que podemos tratar distintos bits como campos independientes, aunque estén juntos en una misma palabra de la máquina. Ejemplo:

```
struct fichero {
    :3 // nos saltamos 3 bits
    unsigned int lectura : 1; // reservamos un bit para lectura
    unsigned int escritura : 1;
    unsigned int ejecución : 1;
    :0 // pasamos a la siguiente palabra
    unsigned int directorio: 8;
} flags;
flags.lectura = 1; // ponemos a 1 el bit de lectura
```

Los campos siempre son de tipo discreto (enteros) y no se puede tomar su dirección.

- **Clases**

Las clases son estructuras con una serie de características especiales; las estudiaremos en profundidad más adelante.

7. Asignación dinámica de memoria

Hay dos formas principales por medio de las cuales un programa puede almacenar información en la memoria del ordenador:

- **Por medio del uso de variables**, en cuyo caso la cantidad de memoria necesaria queda fijada en tiempo de compilación y no puede ser cambiada durante la ejecución del programa. Esto es, si se ha reservado memoria para un array de 10 elementos de tipo int, este array no puede cambiar durante la ejecución.
- La segunda forma de almacenamiento de información es **por medio del sistema de asignación dinámica de memoria** de C++. Por medio de este método se va asignando memoria en tiempo de ejecución según se vaya necesitando.

C++ tiene 2 operadores para la asignación dinámica de memoria: **new** y **delete**. Sus formas generales son:

```
var_puntero = new tipo_variable;
delete var_puntero;
```

Usando **new** para un vector, el tamaño del vector se sitúa entre corchetes. Con **delete** el contenido al que apunta el puntero es borrado. Para asignar memoria dinámicamente a un vector de "**n**" elementos, se utiliza el siguiente formato:

```
var_puntero = new tipo_variable[n];
delete [] var_puntero;
int n=100;
```

o

```
int *obj = new int[n]; //asigna
delete [] obj; //libera
```

Ejemplo:

```
int *obj = new int;
*obj=7.2;
delete obj;
```

Hemos mencionado que en C se usaban las funciones malloc() y free() para el manejo de memoria dinámica, pero dijimos que en C++ se suelen emplear los operadores new y delete. El operador new se encarga de reservar memoria y delete de liberarla. Estos operadores se emplean con todos los tipos del C++, sobre todo con los tipos definidos por nosotros (las clases). La ventaja sobre las funciones de C de estos operadores está en que utilizan los tipos como operandos, por lo que reservan el número de bytes necesarios para cada tipo y cuando reservamos más de una posición no lo hacemos en virtud de un número de bytes, sino en función del número de elementos del tipo que deseemos. El resultado de un new es un puntero al tipo indicado como operando y el operando de un delete debe ser un puntero obtenido con new. Veamos con ejemplos cómo se usan estos operadores:

```
int * i = new int; // reservamos espacio para un entero, i apunta a él
delete i; // liberamos el espacio reservado para i
int * v = new int[10]; // reservamos espacio contiguo para 10 enteros, v apunta
// al primero
delete []v; // Liberamos el espacio reservado para v
```

Hay que tener cuidado con el delete. Si ponemos: delete v; solo liberamos la memoria ocupada por el primer elemento del vector, no la de los 10 elementos. Con el operador new también podemos inicializar la variable a la vez que reservamos la memoria:

```
int *i = new int (5); // reserva espacio para un entero y le asigna el valor
```

Para asignar memoria dinámicamente a una estructura, hay que definir una variable puntero a dicha estructura y luego utilizar el operador **new** para la asignación. Para acceder a los elementos de dicha estructura se utiliza el operador "**->**" Para la asignación dinámica de memoria de arrays de estructuras, se utiliza la forma general de los arrays, es decir, hay que declarar un puntero a la estructura y luego asignar con el operador **new** el tipo y número de elementos. Para acceder a cada elemento del array se utiliza el índice y para acceder a cada miembro de la estructura se utiliza el operador ".":

```
nombre[indice]. miembro;
```

Ejemplo:

```
struct empleado{
        char nombre[20];
        int hijos;
        float sueldo
};
int n=10;
empleado *obj2 = new empleado[n];
obj1[0].nombre="Olga";
```

```
obj1[0].hijos=1;
obj1[0].sueldo=1200;
delete [] obj1;
empleado *obj2 = new empleado[n];
obj2->nombre="Yolanda";
obj2->hijos=1;
obj1->sueldo=1200;
delete obj2;
```

A la hora de usar el operador -> lo único que hay que tener en cuenta es la precedencia de operadores. Ejemplo:

```
++p->i1; // preincremento del campo i1, es como poner ++ (p->i1)
(++p)->i1; // preincremento de p, luego acceso a i1 del nuevo p.
```

Por último diremos que la posibilidad de definir campos de una estructura como punteros a elementos de esa misma estructura es la que nos permite definir los tipos recursivos como los nodos de colas, listas, árboles, etc.

- **Punteros a punteros (indirección múltiple)**

Además de definir punteros a tipos de datos elementales o compuestos también podemos definir punteros a punteros. La forma de hacerlo es poner el tipo y luego tantos asteriscos como niveles de indirección:

```
int *p1;        // puntero a entero
int **p2;       // puntero a puntero a entero
char *c[];      // vector de punteros a carácter
```

Para usar las variables puntero a puntero hacemos lo mismo que en la declaración, es decir, poner tantos asteriscos como niveles queramos acceder:

```
int ***p3;      // puntero a puntero a puntero a entero
p3 = &p2;       // trabajamos a nivel de puntero a puntero a puntero a entero
                // no hay indirecciones, a p3 se le asigna un valor de su mismo tipo
*p3 = &p1;      // el contenido de p2 (puntero a puntero a entero) toma la dirección de
                // p1 (puntero a entero). Hay una indirección, accedemos a lo apuntado por p3
p1 = **p3;      // p1 pasa a valer lo apuntado por lo apuntado por p3 (es decir, lo apuntado por p2). En //nuestro caso, no cambia su valor, ya que p2 apuntaba a p1 desde la operación anterior
```

```
***p3 = 5      // El entero apuntado por p1 toma el valor 5 (ya que p3 apunta a p2
               que apunta a p1)
```

8. Sentencias de control en C++

(Ver sentencias de control en Lenguaje C)

A modo de resumen:

Como estructuras de control el C++ incluye las siguientes construcciones:

— *condicionales:*

- **if** → instrucción de selección simple.
- **switch** → instrucción de selección múltiple.

— *bucles:*

- **do-while** → instrucción de iteración con condición final.
- **while** → instrucción de iteración con condición inicial.
- **for** → instrucción de iteración especial (similar a las de repetición con contador).

— *de salto:*

- **break** → instrucción de ruptura de secuencia (sale del bloque de un bucle o instrucción condicional).
- **continue** → instrucción de salto a la siguiente iteración (se emplea en bucles para saltar a la posición donde se comprueban las condiciones).
- **goto** → instrucción de salto incondicional (salta a una etiqueta).
- **return** → instrucción de retorno de un valor (se emplea en las funciones).

9. Funciones

Una función es una porción de código, un conjunto de sentencias, agrupadas por separado, generalmente enfocadas a realizar una tarea específica. También se suelen denominar subrutinas o subprogramas.

9.1. Definición

La definición de una función consta de la cabecera de la función y del cuerpo. Su forma general es:

```
tipo_valor_retorno nombre_funcion(tipo arg1, tipo arg2, ... ,tipo argn)  {
  ......
  // CUERPO DE LA FUNCION
  ......
}
```

La lista de argumentos, también llamados argumentos formales, es una lista de declaraciones de variables, precedidas de su tipo correspondiente y separadas por comas (,). Los argumentos formales son la forma más natural y directa para que una función reciba valores desde el programa que le llama. El "tipo_valor_retorno" indica el tipo del valor devuelto al programa que le ha llamado. Si no se desea que devuelva nada, el tipo de retorno debe ser **void**. La sentencia **return** permite devolver el valor.

9.2. Declaración

Toda función debe ser declarada antes de ser utilizada en el programa que realiza la llamada. Esta se hace mediante el **prototipo** de la función. La forma general del **prototipo** coincide con la primera línea de la definición –el encabezamiento–, con tres pequeñas diferencias:

- En vez de la lista de argumentos formales o parámetros, basta incluir solo los tipos de dichos argumentos.
- El prototipo termina con un carácter ";".
- Los valores pueden ser inicializados si se desea.

Ejemplo: int func (int, char, float);

La llamada a una función se hace incluyendo su nombre en una expresión o sentencia del programa principal o de otra función. Este nombre debe ir seguido de una lista de argumentos separados por comas y encerrados entre paréntesis. A los argumentos incluidos en la llamada se les llama **argumentos actuales.**

9.3. Paso de argumentos por valor y por referencia

Por defecto los parámetros se pasan por valor, para pasarlos por referencia usaremos punteros y referencias. **Sigue el mismo mecanismo visto en el lenguaje C**.

9.4. Funciones recursivas

Son funciones que pueden llamarse a sí mismas. Cuando una función es llamada por sí misma, se crea un nuevo juego de parámetros y variables locales, pero el código ejecutable es el mismo.

9.5. Funciones Inline

Las funciones **inline** son funciones que no son llamadas sino que son expandidas en línea, en el punto de cada llamada.

Las **ventajas** de estas funciones es que no representan un retardo vinculado con la llamada a la función ni con los mecanismos de vuelta de esta. Esto significa que las funciones **inline** son ejecutadas de forma mucho más rápida que las normales.

Las **desventajas** de estas funciones es que si son demasiado grandes y son llamadas con demasiada frecuencia, el programa se hace más grande.

Para declarar una función **inline** basta con anteponer el especificador **inline** a la definición de la función. Estas deben ser declaradas antes de ser usadas.

Una función inline es igual que una función normal que no genera código de llamada a función, sino que sustituye las llamadas a la misma por el código de su definición. La principal ventaja frente a las macros es que estas funciones sí que comprueban el tipo de los parámetros. No se pueden definir funciones inline recursivas.

10. Programación eficiente

Veremos en este módulo una serie de mecanismos del C++ útiles para hacer que nuestros programas sean más eficientes.

10.1. Estructura de los programas

El código de los programas se almacena en ficheros, pero el papel de los ficheros no se limita al de mero almacén, también tienen un papel en el lenguaje: son un ámbito para determinadas funciones (estáticas y en línea) y variables (estáticas y constantes) siempre que se declaren en el fichero fuera de una función.

Además de definir un ámbito, los ficheros nos permiten la compilación independiente de los archivos del programa, aunque para ello es necesario proporcionar declaraciones con la información necesaria para analizar el archivo de forma aislada.

Una vez compilados los distintos ficheros fuente (que son los que terminan en .c, .cpp, etc.), es el **linker** el que se encarga de enlazarlos para generar un único fichero fuente ejecutable.

En general, los nombres que no son locales a funciones o a clases se deben referir al mismo tipo, valor, función u objeto en cada parte de un programa.

Si en un fichero queremos declarar una variable que está definida en otro fichero, podemos hacerlo declarándola en nuestro fichero precedida de la palabra **extern**.

Si queremos que una variable o función solo pertenezca a nuestro fichero la declaramos **static**.

Si declaramos funciones o variables con los mismos nombres en distintos ficheros producimos un error (para las funciones el error solo se produce cuando la declaración es igual, incluyendo los tipos de los parámetros).

Las funciones y variables cuyo ámbito es el fichero tienen enlazado interno (es decir, el **linker** no las tiene en cuenta).

10.2. Los ficheros cabecera

Una forma fácil y cómoda de que todas las declaraciones de un objeto sean consistentes es emplear los denominados ficheros cabecera, que contienen código ejecutable y/o definiciones de datos. Estas definiciones o código se corresponderán con la parte que queremos utilizar en distintos archivos.

Para incluir la información de estos ficheros en nuestro fichero .C empleamos la directiva **include**, que le servirá al preprocesador para leer el fichero cabecera cuando compile nuestro código.

Un fichero cabecera debe contener:

- Definición de tipos struct punto { int x, y; };
- Templates template <class T> class V { ... }
- Declaración de funciones extern int strlen (const char *);
- Definición de funciones inline inline char get { return *p++ ;}
- Declaración de variables extern int a;
- Definiciones constantes const float pi = 3.141593;
- Enumeraciones enum bool { false, true };
- Declaración de nombres class Matriz;
- Directivas include #include <iostream.h>
- Definición de macros #define Case break;case
- Comentarios /* cabecera de mi_prog.c */

Y no debe contener:

- Definición de funciones ordinarias char get () { return *p++}
- Definición de variables int a;
- Definición de agregados constantes const tabla[ ] = { ... }

Si nuestro programa es corto, lo más usual es crear un solo fichero cabecera que contenga los tipos que necesitan los diferentes ficheros para comunicarse y poner en estos ficheros solo las funciones y definiciones de datos que necesiten e incluir la cabecera global. Si el programa es largo o usamos ficheros que pueden ser reutilizados, lo más lógico es crear varios ficheros cabecera e incluirlos cuando sean necesarios.

Por último, indicaremos que las funciones de biblioteca suelen estar declaradas en ficheros cabecera que incluimos en nuestro programa para que luego el linker las enlace con nuestro programa. Las bibliotecas estándar son:

Bibliotecas de C:

assert.h	Define la macro assert()
ctype.h	Manejo de caracteres
errno.h	Tratamiento de errores
float.h	Define valores en coma flotante dependientes de la implementación
limits.h	Define los límites de los tipos dependientes de la implementación
locale.h	Define la función setlocale()
math.h	Definiciones y funciones matemáticas
setjmp.h	Permite saltos no locales
signal.h	Manejo de señales
stdarg.h	Manejo de listas de argumentos de longitud variable
stddef.h	Algunas constantes de uso común
stdio.h	Soporte de E/S
sdlib.h	Algunas declaraciones estándar
string.h	Funciones de manipulación de cadenas
time.h	Funciones de tiempo del sistema

Bibliotecas de C++:

fstream.h	Streams fichero
iostream.h	Soporte de E/S orientada a objetos (streams)
new.h	Definición de _new_handler
strstream.h	Definición de streams cadena

11. El preprocesador

El preprocesador es un programa que se aplica a los ficheros fuente del C++ antes de compilarlos. Realiza diversas tareas, algunas de las cuales se pueden controlar mediante el uso de directivas de preprocesado. Como veremos, estas directivas permiten definir macros como las de los lenguajes ensambladores (en realidad no se trata más que de una sustitución). A continuación veremos las fases de preprocesado y las directivas, así como una serie de macros predefinidas. Por último explicaremos lo que son las secuencias trigrafo.

11.1. Fases de preprocesado

1. Traduce los caracteres de fin de línea del fichero fuente a un formato que reconozca el compilador.
2. Concatena cada línea terminada con la barra invertida (\) con la siguiente.
3. Elimina los comentarios. Divide cada línea lógica en símbolos de preprocesado y espacios en blanco.
4. Ejecuta las directivas de preprocesado y expande los macros.
5. Reemplaza las secuencias de escape dentro de constantes de caracteres y cadenas de literales por sus caracteres individuales equivalentes.
6. Concatena cadenas de literales adyacentes.
7. Convierte los símbolos de preprocesado en símbolos de C++ para formar una unidad de compilación.

Estas fases se ejecutan exactamente en este orden.

Directivas del preprocesador:

#define ID VAL	Define la macro ID con valor VAL.
#include «fichero»	Incluye un fichero del directorio actual.
#include <fichero>	Incluye un fichero del directorio por defecto.

#defined id	Devuelve 1 si id está definido.
#defined (id)	Lo mismo que el anterior.
#if expr	Si la expresión se cumple se compila todo lo que sigue. Si no, se pasa hasta un #else o un #endif.
#ifdef id	Si el macro id ha sido definido con un #define, la condición se cumple y ocurre lo del caso anterior. Es equivalente a if defined id.
#ifndef id	Si el macro id no ha sido definido con un #define, la condición se cumple. es equivalente a if !defined id.
#else	Si el #if, #ifdef o #ifndef más reciente se ha cumplido todo lo que haya después del #else hasta #endif no se compila. Si no se ha cumplido si se compila.
#elif exp.	Contracción de #else if expr.
#endif	Termina una condición.
#line CONST ID	Cambia el número de línea según la constante CONST y el nombre del fichero de salida de error a ID. Modifica el valor de los macros predefinidos __LINE__ y __FILE__
#pragma OPCION	Especifica al compilador opciones específicas de la implementación.
#error CADENA	Causa la generación de un mensaje de error con la cadena dada.

Ejemplo de macros:

```
#define MIN(a,b) (((a) < (b)) ? (a) : (b) )
main () {
      int i, j=6, k=8;
      i = MIN(j*3, k-1);
}
```

Después del preprocesado tendremos:

```
main () {
      int i, j=6, k=8;
      i = (((j*3) < (k-1)) ? (j*3) : (k-1));
}
```

Si no hubiéramos puesto paréntesis en la definición de la macro, al sustituir a y b podíamos haber introducido operadores con mayor precedencia que ?: y haber obtenido un resultado erróneo al ejecutar la macro. Notar que la macro no hace ninguna comprobación en los parámetros simplemente sustituye, por lo que a veces puede producir resultados erróneos.

12. Entrada / Salida de datos

12.1. Entrada / Salida a través de Consola en C++

C++ incorpora su propio archivo de cabecera, denominado **iostream.h** que implementa su propio conjunto de funciones de entrada/salida. La E/S es un flujo de C++ se describe como un conjunto de clases en **iostream.h**. Estas clases sobrecargan los operadores put to y get from, << y >>. En C++, la clase proporciona soluciones modulares a las necesidades de manipulación de datos. La biblioteca estándar de C++ ofrece tres clases de E/S como alternativa a las funciones de E/S de propósito general de C. Estas clases contienen definiciones para el mismo par de operadores (>> y <<) que se optimizan para todos los tipos de datos.

cin,cout y cerr.

El equivalente para los flujos en C++ de stdin, stdout y stderr, descritos en STDIO.H, son cin, cout y cerr, que se describen en IOSTREAM.H. Estos tres flujos se abren automáticamente cuando comienza la ejecución del programa y pasan a ser la interfaz entre el programa y el usuario. El flujo cin se asocia con el teclado del terminal. Los flujos *cout y cerr* se asocian con la pantalla de visualización.

Los operadores >> para extracción y << para inserción.

Las entradas y salidas en C++ han mejorado de forma significativa y se han simplificado debido a los operadores de la biblioteca de flujo >> (get from o extracción) y << (put to o inserción). Una de las principales mejoras que presenta C++ respecto a C es la sobrecarga de operadores. La sobrecarga de operadores permite al compilador determinar qué función u operador va a ser ejecutado basándose en los tipos de datos de las variables asociadas.

Ejemplo:

```
int ivalor=10;   float fvalor=7.2;
count << "Valor entero: " << ivalor << ", Valor en coma flotante: " << fvalor;
cin >> ivalor >> fvalor >> c;
```

Ya no es necesario preceder a las variables de entrada con el operador de dirección &. Cuando se quiere introducir información, se extrae >> del flujo de entrada, **cin**, y se coloca la información en una variable, por ejemplo ivalor. Para extraer información, se coge una copia de la información de la variable fvalor y se inserta << en el flujo de salida, **cout**. El operador de extracción >> lee hasta el carácter nueva línea pero no lo ignora.

Cuando se compara el código fuente en C++ con la salida del programa, una de las cosas que se observa inmediatamente es que el operador de inserción << no genera automáticamente una nueva línea. Puede controlar cuándo ocurre esto incluyendo el símbolo de nueva línea *\n* o *endl* cuando sea necesario; endl es muy útil para la salida de datos en un programa interactivo porque no solo inserta una nueva línea en un flujo, sino que además vuelca el buffer de salida. También puede utilizar **flush**; sin embargo, este no inserta una nueva línea.

La función **cin.get**() leerá todo, incluyendo el espacio en blanco, hasta el número máximo de caracteres especificado que se hayan leído o hasta el siguiente carácter de nueva línea. El tercer parámetro opcional, que no se ha mostrado, identifica un símbolo de terminación. Por ejemplo, la siguiente línea leería N caracteres en nombre o todos los caracteres escritos antes de un símbolo * o un carácter de nueva línea:

*cin.get (nombre, N, *);*

La clase **istream** mantiene las operaciones de entrada básicas, mientras que la salida básica la mantiene la clase **ostream**. La E/S bidireccional es mantenida por la clase **iostream**, que se deriva de **istream** y **ostream**. Hay cuatro objetos de flujo predefinidos para el usuario:

- **cin** → Objeto de la clase **istream** asociado a la entrada estándar.
- **cout** → Objeto de la clase **ostream** asociado a la salida estándar.
- **cerr** → Objeto de la clase **ostream** sin buffer de salida asociado al error estándar.
- **clog** → Objeto de la clase **ostream** con buffer de salida asociado al error estándar.

12.2. Entrada / Salida de datos a través de ficheros en C++

Los ficheros se utilizan para la lectura y/o escritura de datos en unidades de almacenamiento permanente como los disquetes, discos duros, etc. En los ficheros de **acceso secuencial** se lee o escribe desde el inicio del fichero o se escribe a partir del final. Para trabajar con este tipo de ficheros las clases necesarias son **ifstream, ofstream y fstream**, que derivan de **istream** y **ostream**, que a su vez derivan de la clase ios. Para utilizarlas se debe incluir el archivo de cabecera **fstream.h**.

Antes de abrir un fichero hay que crear un **flujo** o **stream**, es decir un objeto de las clases **ifstream**, **ofstream** o **fstream** e indicar el modo de apertura (lectura, escritura, etc.).

- **Clase ofstream**

Es una clase derivada de **ostream**, especializada en manipular ficheros en el disco abiertos **para escribir.** Al construir un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **filebuf** (un buffer). La funcionalidad de esta clase está soportada por las siguientes funciones miembro:

ofstream (const char **nombre_fichero,* int *modo*=ios::out, int *proteccion*=filebuf:: openprot);

void open (const char **nombre_fichero*, int *modo*=ios::out, int *proteccion*= filebuf::openprot);

void close(); //esta función cierra el fichero

int is_open(); //verifica si el fichero está abierto(=1). Si no lo está devuelve un 0.

Para **escribir** en el fichero se utiliza el operador de inserción << sobrecargado. Para **leer** del fichero se usa el operador de extracción >>. Esta forma de escritura es solo en formato texto.

- **Clase ifstream**

Es una clase derivada de **istream**, especializada en manipular ficheros en el disco abiertos **para leer**. Al construir un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **filebuf** (un buffer). La funcionalidad de esta clase está soportada por las siguientes funciones miembro:

ifstream (const char **nombre_fichero,* int *modo*=ios::in, int *proteccion*=filebuf::openprot);

void open (const char **nombre_fichero,* int *modo*=ios::in, int *proteccion*=filebuf::openprot);

void close(); //esta función cierra el fichero

int is_open(); //verifica si el fichero está abierto(=1). Si no lo está devuelve un 0.

- **Clase fstream**

Es una clase derivada de **iostream**, especializada en manipular ficheros en el disco abiertos **para leer y/o escribir**. Al construir un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **filebuf** (un buffer). La funcionalidad de esta clase está soportada por las siguientes funciones miembro:

fstream (const char **nombre_fichero*, int *modo*, int *proteccion*=filebuf::openprot);

void open (const char **nombre_fichero*, int *modo*, int *proteccion*=filebuf::openprot);

void close(); //esta función cierra el fichero

int is_open(); //verifica si el fichero está abierto(=1). Si no lo está devuelve un 0.

Si el fichero se abre con el modo: **ios::app** entonces, todo lo que se escriba se agregará a partir del final del fichero. Otras posibilidades de leer y escribir en un fichero son:

— **getline():** lee de fichero un número de caracteres especificado en la variable **nCount** o hasta que encuentre el carácter fin de línea \n. Devuelve un NULL cuando encuentra el final del fichero. Su prototipo es:

istream& **getline(**unsigned char* puch, int nCount, char dlm = '\n');

— **read() y write():** leen y escriben, respectivamente, bloques de bytes o **datos binarios.** Sus prototipos son:

istream& **read**(unsigned char* bif, int num);
ostream& **write**(unsigned char* bif, int num);

En el **acceso aleatorio** de ficheros permite leer o escribir a partir de una determinada posición del fichero. Esto tiene una gran ventaja, ya que se pueden modificar algunos de los valores contenidos en el fichero. C++ nos da unas funciones para el acceso aleatorio:

— Para la **clase istream**:

- istream &seekg(streamoff **desp**, ios::seek_dir **pos**);
- streampos tellg();

— Para la **clase ostream**:

- ostream &seekp(streamoff desp, ios::seek_dir pos);
- streampos tellp();

Donde:

streampos es un typedef de long.

desp es la nueva posición, desplazada.

desp bytes, desde la posición dada por **pos**, el cual puede ser:

ios::beg Principio del fichero.

ios::cur Posición actual del puntero del stream.

ios::end Final del stream.

seekg se usa para desplazarse en un fichero para lectura.

seekp se usa para desplazarse en un fichero para escritura.

tellg, tellp dan la posición actual del puntero de lectura y escritura, respectivamente.

Para escribir en un fichero de acceso aleatorio, este debe ser abierto de modo lectura/escritura, usando para ello: **ios::in | ios::out**

Leer y escribir objetos. Para leer y escribir objetos en formato binario, se deben sobrecargar los operadores de extracción >> e inserción <<, en los cuales pondremos el **código** necesario para usar las funciones **read** y **write** de las clases **ofstream**, **ifstream** o fstream.

13. Programación orientada a objetos

13.1. Concepto de clase en C++

Una clase es un tipo de datos definido por el usuario. Es una agrupación de datos (**variables**) y de funciones (**métodos**) que operan sobre esos datos. La definición de una clase consta de dos partes:

- La primera está formada por el nombre de la clase precedido por la palabra reservada **class**.
- La segunda parte es el cuerpo de la clase, encerrado entre llaves y seguido por "**;**":

```
class nombre
{
    //cuerpo
};
```

El cuerpo de la clase consta de:

- Especificadores de acceso: **public, protected y private.**
- **Atributos**: datos miembro de la clase (variables).
- **Métodos**: definiciones de funciones miembro de la clase.

Tanto las variables como los métodos pueden ser declarados public, protected y private, controlando de esta forma el acceso y evitando un uso inadecuado. La idea de clase junto con la sobrecarga de operadores, que estudiaremos más adelante, permite al usuario diseñar tipos de datos que se comporten de manera similar a los tipos estándar del lenguaje. Esto significa que debemos poder declararlos y usarlos en diversas operaciones como si fueran tipos elementales, siempre y cuando esto sea necesario. La idea central es que los tipos del usuario solo se diferencian de los elementales en la forma de crearlos, no en la de usarlos. Pero las clases no solo nos permiten crear tipos de datos, sino que nos dan la posibilidad de definir tipos de datos abstractos y definir sobre estos nuevas operaciones sin relación con los operadores estándar del lenguaje. Introducimos un nuevo nivel de abstracción y comenzamos a ver los tipos como representaciones de ideas o conceptos que no necesariamente tienen que tener una contrapartida a nivel matemático, como sucedía hasta ahora, sino que pueden ser conceptos relacionados con el mundo de nuestro programa. Así, podremos definir un tipo coche y un tipo motor en un programa de mecánica o un tipo unidad funcional en un programa de diseño de arquitectura de computadores, etcétera. Sobre estos tipos definiremos una serie de operaciones que definirán la interacción de las variables (objetos) de estos tipos con el resto de entidades de nuestro programa.

Otra idea fundamental a la hora de trabajar con clases es la distinción clara entre la definición o interface de nuestros tipos de datos y su implementación, esto es, la distinción entre qué hace y cómo lo hace. Una buena definición debe permitir el cambio de la estructura interna de nuestras clases sin que esto afecte al resto de nuestro programa. Deberemos definir los tipos de manera que el acceso a la información y operaciones que manejan sea sencillo, pero el acceso a las estructuras y algoritmos utilizados en la implementación sea restringido, de manera que una modificación en estos últimos solo sea percibido en la parte de nuestro programa que implementa la clase.

Por último, es interesante tener presentes las ideas ya mencionadas en el bloque anterior de objeto y paso de mensajes entre objetos. Estas ideas pue-

den resultar de gran ayuda a la hora del diseño e implementación de programas, ya que podemos basarnos en el comportamiento de los objetos de interés en nuestro programa para abstraer clases y métodos (mensajes).

Nota. *Por omisión, los datos miembro de la clase son private.*

13.2. Miembros de una clase

Los miembros de una clase pueden ser variables de cualquier tipo y funciones. En C++ a las variables se las denomina datos miembro, y a las funciones, funciones miembro de la clase.

A) Datos miembro de una clase

Para declarar un dato miembro se procede de la misma forma que para declarar cualquier variable. Por ejemplo:

```
class Lolo
{
        public:
                float real;
};
```

- Los datos miembro **no** pueden ser inicializados durante la declaración.
- Si no se pone un especificador de acceso (como es nuestro ejemplo public), por defecto los datos miembro serán private.
- En una clase cada dato miembro debe tener un nombre único.
- También podemos declarar como datos miembro de una clase, **objetos** de otra clase, siendo necesario que esta haya sido previamente definida.

```
class datos
{
        Lolo obj_lolo; //Declaración de un objeto de la clase dolo
};
```

- Para acceder a un dato miembro (siempre y cuando sea declarado como **público**) de una clase desde un objeto, se utilizará el operador Punto ".". Por ejemplo: ***obj_lolo.real = 5.4;***

13.3. Funciones miembro de una clase

Las funciones miembro de una clase definen las operaciones que se pueden realizar con sus datos miembro. Las funciones miembro también pueden ser públicas o privadas, lo cual se hace con los respectivos especificadores. Al

igual que los datos miembro, si no se especifica serán privadas (*private*) por defecto. Para declarar una función miembro de una clase se procede de la misma forma que para declarar una función cualquiera. Por ejemplo en el fichero **lolo.h**:

```
class Lolo {
        private:
        float real,imaginario;
        public: //funcione miembro públicas
        void Asignar (float x, float y);
};
```

El cuerpo de una clase solo contiene los prototipos de las funciones miembro (las declaraciones). La definición de la función se hace en los ficheros fuente (***.cpp**).

Para definir la función miembro en el fichero fuente se utiliza el nombre de la clase seguido por el **operador de resolución de ámbito** "::". Por ejemplo, en el fichero **lolo.cpp:**

```
void Lolo::Asignar(int x, float y) {
        real = x; imaginario = y; //acceso a los datos miembro
        ......
        ......
}
```

Las funciones miembro que tengan poco código, o las que se desee, también pueden ser definidas en el cuerpo de la clase. Dentro del cuerpo de la clase no hay necesidad de anteponer el nombre de la clase con el operador "::", como se hace generalmente en el fichero fuente, puesto que el nombre de la clase es conocido. Para acceder a una función miembro (siempre y cuando sea declarada como ***pública***) de una clase desde un objeto, se utilizará el operador Punto ".". Por ejemplo: ***obj_lolo.Asignar (10, 7.2)***;

A) Control de acceso a los miembros de la clase

El concepto de clase incluye la idea de ocultación de datos, que, básicamente, consiste en que no se puedan acceder a los datos miembro directamente, sino que hay que hacerlo a través de las funciones miembro públicas de la clase. Para controlar el acceso a los miembros (datos y funciones) de una clase, C++ provee de tres especificadores:

- **Public**. Un miembro declarado público es accesible en cualquier parte del programa donde el **objeto** de la clase en cuestión es accesible.
- **Private**. Un miembro declarado privado puede ser accedido solamente por las funciones miembro de su propia clase o por funciones amigas (**friend**) de su clase. En cambio, **no puede ser accedido** por funciones globales o por las funciones propias de una clase derivada (**herencia**).

- **Protected**. Un miembro declarado protegido se comporta exactamente igual que uno privado para las funciones globales, pero **actúa como un miembro público** para las funciones miembro de una clase derivada.

Una clase es un tipo de datos que se define mediante una serie de miembros que representan atributos y operaciones sobre los objetos de ese tipo. Hasta ahora conocemos la forma de definir un tipo de datos por los atributos que posee, lo hacemos mediante el uso de las estructuras. Pensemos, por ejemplo, en cómo definimos un tipo de datos *empleado* en un programa de gestión de personal:

```
struct empleado {
        char * nombre;
        long DNI;
        float sueldo;
        ...
};
```

Con esta definición conseguimos que todas las características que nos interesan del empleado se puedan almacenar conjuntamente, pero nos vemos obligados a definir funciones que tomen como parámetro variables de tipo empleado para trabajar con estos datos:

```
void modificar_sueldo (empleado *e, float nuevo_sueldo);
...
```

Pero el C++ nos da una nueva posibilidad, incluir esas funciones como miembros del tipo empleado:

```
struct empleado {
        char * nombre;
        long DNI;
        float sueldo;
        ...
        void modificar_sueldo (float nuevo_sueldo);
        ...
};
```

A estas funciones se les denomina **miembros función o métodos**, y tienen la peculiaridad de que solo se pueden utilizar junto con variables del tipo definido. Es interesante señalar, aunque sea anticipar acontecimientos, que la función miembro no necesita que se le pase la estructura como parámetro, ya que al estar definida dentro de ella tiene acceso a los datos que contiene.

Como distintas clases pueden emplear el mismo nombre para los miembros, a la hora de definir las funciones miembro debemos especificar el nombre de la estructura a la que pertenecen:

```
void empleado::modificar_sueldo (float nuevo_sueldo) {
sueldo = nuevo_sueldo;
};
```

Si definimos la función dentro de la estructura esto último no es necesario, ya que no hay lugar para la confusión.

B) Acceso a miembros: la palabra class

Hasta ahora hemos empleado la palabra struct para definir las clases; este uso es correcto, pero tiene una connotación específica: todos los miembros del tipo son accesibles desde el exterior del tipo, es decir, podemos modificar los datos o invocar a las funciones del mismo desde el exterior de la definición:

```
empleado lolo; // declaramos un objeto de tipo empleado
lolo.sueldo = 500; // asignamos el valor 500 al campo sueldo
lolo.modificar_sueldo(600) // le decimos a lolo que cambie su sueldo a 600
```

En el caso del ejemplo puede parecer poco importante que se pueda acceder a los datos del tipo, pero hemos dicho que lo que nos interesa es que la forma de representar los datos o de implementar los algoritmos solo debe ser vista en la definición de la clase. Para que lo que contiene la clase solo sea accesible desde la definición empleamos la palabra class en lugar de struct para definir el tipo:

```
class empleado {
...
}
```

C) Acceso a miembros: etiquetas de control de acceso

Con public delante de los miembros de la clase éstos sí deben ser vistos desde fuera:

```
class empleado {
        char * nombre;
        long DNI;
        float sueldo;
        ...
public:
        void modificar_sueldo (float nuevo_sueldo);
        ...
```

```
} lolo;
lolo.sueldo = 500; // ERROR, sueldo es un miembro privado
lolo.modificar_sueldo (600); // CORRECTO, modificar_sueldo() es un método público
```

Además de public también podemos emplear las etiquetas protected y private dentro de la declaración de la clase. Todo lo que aparezca después de una etiqueta será accesible (o inaccesible) hasta que encontremos otra etiqueta que cambie la accesibilidad o inaccesibilidad. La etiqueta protected tiene una utilidad especial que veremos cuando hablemos de herencia; de momento la usaremos de la misma forma que private, es decir, los miembros declarados después de la etiqueta serán inaccesibles desde fuera de la clase. Utilizando las etiquetas podemos emplear indistintamente la palabra struct o class para definir clases, la única diferencia es que si no ponemos nada con struct se asume acceso público y con class se asume acceso privado (con el sentido de la etiqueta private, no protected). Es mejor usar siempre la palabra class y especificar siempre las etiquetas de permiso de acceso, aunque podamos tener en cuenta el hecho de que por defecto el acceso es privado es más claro especificarlo. Hemos de indicar que también se puede definir una clase como union, que implica acceso público pero solo permite el acceso a un miembro cada vez (es lo mismo que sucedía con las uniones como tipo de datos compuesto).

D) Operadores de acceso a miembros

El acceso a los miembros de una clase tiene la misma sintaxis que para estructuras (el operador . y el operador ->), aunque también se emplea muy a menudo el operador de campo (::) para acceder a los miembros de la clase. Por ejemplo se emplea el operador de campo para distinguir entre variables de un método y miembros de la clase:

```
class empleado {
        ...
        float sueldo;
        ...
public:
        void modificar_sueldo (float sueldo) {
        empleado::sueldo = sueldo;
}
...
};
```

E) El puntero this

En uno de los puntos anteriores comentábamos que un método perteneciente a una clase tenía acceso a los miembros de su propia clase sin necesidad de pasar como parámetro el objeto con el que se estaba trabajando. Esto no es tan sencillo, puesto que es lógico pensar que los atributos (datos) con-

tenidos en la clase son diferentes para cada objeto de la clase, es decir, se reserva memoria para los miembros de datos, pero no es lógico que cada objeto ocupe memoria con una copia de los métodos, ya que replicaríamos mucho código.

Los objetos de una clase tienen un atributo específico asociado: su dirección. La dirección del objeto nos permitirá saber qué variables debemos modificar cuando accedemos a un miembro de datos. Esta dirección se pasa como parámetro (implícito) a todas las funciones miembro de la clase y se llama this. Si en alguna función miembro queremos utilizar nuestra propia dirección podemos utilizar el puntero como si lo hubiéramos recibido como parámetro. Por ejemplo, para retornar el valor de un atributo escribimos:

```
float empleado::cuanto_cobra (void) {
return sueldo;
}
```

Pero también podríamos haber hecho lo siguiente:

```
float empleado::cuanto_cobra (void) {
return this->sueldo;
}
```

Utilizar el puntero dentro de una clase suele ser redundante, aunque a veces es útil cuando trabajamos con punteros directamente.

F) Funciones miembro constantes

Un método de una clase se puede declarar de forma que nos impida modificar el contenido del objeto (es decir, como si para la función el parámetro this fuera constante). Para hacer esto basta escribir la palabra después de la declaración de la función:

```
class empleado {
        ...
        float cuanto_cobra (void) const;
        ...
};
float empleado::cuanto_cobra (void) const {
        return sueldo;
}
```

Las funciones miembro constantes se pueden utilizar con objetos constantes, mientras que las que no lo son no pueden ser utilizadas (ya que podrían modificar el objeto). De cualquier forma, existen maneras de modificar un objeto desde un método constante: el empleo de cast sobre el parámetro this o el uso de miembros puntero a datos no constantes. Veamos un ejemplo para el primer caso:

```
class empleado {
        private:
                ...
                long num_accesos_a_empleado;
                ...
        public:
                ...
                float cuanto_cobra (void) const
                ...
};
float empleado::cuanto_cobra (void) const {
        ((empleado *)this)->num_accesos_a_empleado += 1; // hemos accedido una
        vez más a
        // la clase empleado
        return sueldo;
}
```

Otro ejemplo:

```
struct contabilidad {
        long num_accesos_a_clase;
};
class empleado {
        private:
        ...
        contabilidad *conta;
        ...
        public:
        ...
        float cuanto_cobra (void) const
        ...
};
float empleado::cuanto_cobra (void) const {
        conta->num_accesos_a_clase += 1; // hemos accedido una vez más a
        // la clase empleado
        return sueldo;
}
```

Esta posibilidad de modificar objetos desde métodos constantes se permite en el lenguaje por una cuestión conceptual: un método constante no debe modificar los objetos desde el punto de vista del usuario, y declarándolo como tal el usuario lo sabe, pero, por otro lado, puede ser interesante que algo que, para el que llama a una función miembro, no modifica al objeto si lo haga realmente con variables internas (no visibles para el usuario) para llevar contabilidades o modificar estados. Esto es especialmente útil cuando declaramos objetos constantes de una clase, ya que podemos modificar variables mediante funciones constantes.

G) Funciones miembro inline

Al igual que se podían declarar funciones de tipo inline generales, también se pueden definir funciones miembro inline. La idea es la misma; que no se genere llamada a función. Para hacer esto en C++ existen dos posibilidades: definir la función en la declaración de la clase (por defecto implica que la función miembro es inline), o definir la función fuera de la clase precedida de la palabra inline:

```
inline float empleado::cuanto_cobra {
        return sueldo;
}
```

Lo único que hay que indicar es que no podemos definir la misma función inline dos veces (en dos ficheros diferentes).

H) Atributos estáticos

Cuando en la declaración de una clase ponemos atributos (datos) estáticos, queremos indicar que ese atributo es compartido por todos los objetos de la clase. Para declararlo estático solo hay que escribir la palabra static antes de su declaración:

```
class empleado {
        ...
        static long num_total_empleados;
        ...
};
```

Con esto conseguimos que el atributo tenga características de variable global para los miembros de la clase, pero que permanezca en el ámbito de la misma. Hay que tener presente que los atributos estáticos ocupan memoria aunque no declaremos ningún objeto.

Si un atributo se declara público para acceder a él desde el exterior de la clase debemos identificarlo con el operador de campo: *empleado::num_total_empleados = 1000;*

El acceso desde los miembros de la clase es igual que siempre. Los atributos estáticos se deben definir fuera del ámbito de la clase, aunque al hacerlo no se debe poner la palabra static (podrían producirse conflictos con el empleo de static para variables y funciones globales). Si no se inicializan en su definición toman valor 0:

```
long empleado::num_total_empleados; // definición, toma valor 0
```

El uso de atributos estáticos es más recomendable que el de las variables globales.

I) Tipos anidados

Dentro de las clases podemos definir nuevos tipos (enumerados, estructuras, clases...), pero para utilizarlos tendremos las mismas restricciones que para usar los miembros, es decir, serán accesibles según el tipo de acceso en el que se encuentren y para declarar variables de esos tipos tendremos que emplear la clase y el operador de campo:

```
class lista {
    private:
        struct nodo {
            int val;
            nodo *sig;
        };
        nodo *primero;
    public:
        enum tipo_lista { FIFO, LIFO };
        void inserta (int);
        int siguiente ();
    ...
};
nodo n1;                // ERROR, nodo no es un tipo definido, está en otro ámbito
tipo_lista tl1;         // ERROR, tipo_lista no definido
lista::nodo n2;         // ERROR, tipo nodo privado
lista::tipo_lista tl2;  // CORRECTO
```

J) Punteros a miembros

Cuando dimos la lista de operadores de indirección mencionamos dos de ellos que aún no se han visto: el operador de puntero selector de puntero a miembro (->*) y el operador selector de puntero a miembro (.*). Estos operadores están directamente relacionados con los punteros a miembros de una clase (como sus nombres indican). Suele ser especialmente interesante tomar la dirección de

los métodos por la misma razón que era interesante tomar la dirección de funciones, aunque en clases se utiliza más a menudo (de hecho las llamadas a métodos de una clase hacen uso de punteros a funciones, aunque sea implícitamente). Para tomar la dirección de un miembro de la clase X escribimos &X::miembro. Una variable del tipo puntero a miembro de la clase X se obtiene declarándolo de la forma X::*. Por ejemplo si tenemos la clase:

```
class empleado {
        ...
        void imprime_sueldo (void);
        ...
};
```

Podemos definir una variable que apunte a un método de la clase que retorna void y no tiene parámetros:

```
void (empleado::*ptr_a_metodo) (void);
```

o usando typedef:

```
typedef void (empleado::*PMVV) (void);
PMVV ptr_a_metodo;
```

Para usar la variable podemos hacer varias cosas:

```
empleado e;
empleado *pe;
PMVV ptr_a_metodo = &empleado::imprime_sueldo;
e.imprime_sueldo();          // llamada normal
(e.*ptr_a_metodo)();         // acceso a miembro apuntado por puntero a través de un objeto
(pe->*ptr_a_metodo)();       // acceso a miembro apuntado por puntero a través de un puntero a objeto
```

En el ejemplo se usan paréntesis porque .* y ->* tienen menos precedencia que el operador de función. En realidad el uso de estos punteros es poco usual, ya que se puede evitar usando funciones virtuales.

K) Objetos de una clase

Un **objeto** es un ejemplar concreto de una clase. Las **clases** son como tipos de variables, mientras que los **objetos** son como variables concretas de un tipo determinado. Los **objetos** constan de una estructura interna (los datos) y de una interfaz que permite manipular tal estructura (las funciones). Un objeto se crea de la misma forma que como se crea una variable.

Ejemplo.- Lolo milolo;

Un objeto no se inicializa como las variables:

Ejemplo: Lolo milolo=5; *//Error*

13.4. Constructores y destructores

Hasta ahora hemos hablado de la declaración y definición de clases, pero hemos utilizado los objetos sin saber cómo se crean o se destruyen. En este punto veremos cómo las clases se crean y destruyen de distintas maneras y qué cosas se hacen al crear o destruir un objeto.

- **Creación de objetos**

Podemos clasificar los objetos en cuatro tipos diferentes según la forma en que se crean:

1. **Objetos automáticos:** son los que se crean al encontrar la declaración del objeto y se destruyen al salir del ámbito en que se declaran.
2. **Objetos estáticos:** se crean al empezar la ejecución del programa y se destruyen al terminar la ejecución.
3. **Objetos dinámicos:** son los que se crean empleando el operador new y se destruyen con el operador delete.
4. **Objetos miembro:** se crean como miembros de otra clase o como un elemento de un array.

Los objetos también se pueden crear con el uso explícito del constructor (lo vemos en seguida) o como objetos temporales. En ambos casos son objetos automáticos. Hay que notar que estos modelos de creación de objetos también es aplicable a las variables de los tipos estándar del C++, aunque no tenemos tanto control sobre ellos.

- **Inicialización y limpieza de objetos**

Con lo que sabemos hasta ahora sería lógico pensar que si deseamos inicializar un objeto de una clase debemos definir una función que tome como parámetros los valores que nos interesan para la inicialización y llamar a esa función nada más declarar la función. De igual forma, nos interesará tener una función de limpieza de memoria si nuestro objeto utiliza memoria dinámica, que deberíamos llamar antes de la destrucción del objeto. Bien, esto se puede hacer así, explícitamente, con funciones definidas por nosotros, pero las llamadas a esos métodos de inicialización y limpieza pueden resultar pesadas y hasta difíciles de localizar en el caso de la limpieza de memoria.

Para evitar el tener que llamar a nuestras funciones, C++ define dos funciones especiales para todas las clases: los métodos constructor y destructor. La función **constructor** se invoca automáticamente cuando creamos un objeto y la **destructor** cuando lo destruimos. Nosotros podemos implementar o no estas funciones, pero es importante saber que si no lo hacemos el C++ utiliza un constructor y destructor por defecto. Estos métodos tienen una serie de **características comunes** muy importantes:

- No retornan ningún valor, ni siquiera de tipo void. Por lo tanto, cuando las declaramos, no debemos poner ningún tipo de retorno.
- Como ya hemos dicho, si no se definen se utilizan los métodos por defecto.
- No pueden ser declarados constantes, volátiles ni estáticos.
- No se puede tomar su dirección.
- Un objeto con constructores o destructores no puede ser miembro de una unión.
- El orden de ejecución de constructores y destructores es inverso, es decir, los objetos que se construyen primero se destruyen los últimos. Ya veremos que esto es especialmente importante al trabajar con la herencia.

A) Constructor

En C++, una forma de asegurar que los objetos siempre contengan valores válidos y que puedan ser inicializados al momento de la declaración es escribiendo un **constructor**. Un constructor es una función miembro especial de una clase que es llamada de forma automática siempre que se declara un objeto de esa clase. Su función es crear e inicializar un objeto de su clase. Dado que un constructor es una función miembro, admite argumentos al igual que estas. El constructor se puede distinguir claramente, con respecto a las demás funciones miembro de la clase, porque tiene el mismo nombre que el de la clase. Un constructor no retorna ningún valor ni se hereda. Si el usuario no ha creado uno, el compilador crea uno por omisión, sin argumentos. Pueden existir varios constructores, siempre y cuando difieran en los argumentos. Se puede crear un objeto de cualquiera de las formas siguientes:

- Declarando un objeto local o temporal: **Lolo obj_lolo;**
- Invocando al operador new: **Lolo *obj_lolo = new Lolo;**

  *Ejemplo: Lolo *obj_lolo = new Lolo(3,4);*

De esta forma, se debe llamar al operador **delete** al finalizar.

- Llamando explícitamente al constructor: **Lolo obj_lolo(4 , 5.0)**;
- Cuando se utiliza el operador **new** para crear el objeto, el acceso a las variables y funciones miembro de la clase se hace a través del operador (->), siempre y cuando sean públicas. Por ejemplo:

```
obj_lolo->real = 4.5; //llamada a una variable perteneciente a la clase
obj_lolo->funcion(); //llamada a una funcion perteneciente a la clase
```

Los constructores se pueden considerar como funciones de inicialización y, como tales, pueden tomar cualquier tipo de parámetros, incluso por defecto. Los constructores se pueden sobrecargar, por lo que podemos tener muchos constructores para una misma clase (como ya sabemos, cada constructor debe tomar parámetros distintos). Existe un constructor especial (que podemos definir) o no definir que tiene una función muy específica: copiar atributos entre objetos de una misma clase. Si no lo definimos se usa uno por defecto que copia todos los atributos de los objetos, pero si lo definimos se usa el nuestro.

Este constructor se usa cuando inicializamos un objeto por asignación de otro objeto. Para declarar un constructor lo único que hay que hacer es declarar una función miembro sin tipo de retorno y con el mismo nombre que la clase, como ya hemos dicho los parámetros pueden ser los que queramos:

```
class Complejo {
  private:
      float re;
      float im;
  public:
     Complejo (float = 0, float = 0);      // constructor con dos parámetros por defecto
                                           // Lo podremos usar con 0, 1, o 2 parámetros.
     Complejo (&Complejo);                 // constructor copia
     ...
};
                                           // Definición de los constructores
                                           // Inicialización
Complejo::Complejo (float pr, float pi) { re = pr; im = pi; }
                                           // Constructor copia
Complejo::Complejo (Complejo& c) { re= c.re; im= c.im; }
```

Los constructores se suelen declarar públicos, pero si todos los constructores de una clase son privados solo podremos crear objetos de esa clase utilizando funciones amigas. A las clases que solo tienen constructores privados se las suele denominar privadas. Los constructores se pueden declarar virtuales.

B) Destructor

De la misma forma que existe una función para **construir** cada unos de los objetos que declaramos, también existe una función para **destruir** cada objeto construido, liberando así la memoria que ocupa. Esta función recibe el

nombre de destructor. Un objeto es destruido automáticamente al salir del ámbito en el que ha sido definido. Sin embargo, si se ha creado con el operador new, se debe utilizar el operador delete para destruirlo. El destructor es una función miembro especial de una clase que se utiliza para eliminar un objeto de esa clase. El destructor se distingue fácilmente del resto de funciones porque tiene el mismo nombre que la clase precedido del operador tilde "~". El destructor no es heredado, no tiene argumentos ni puede retornar ningún valor. No hace falta llamar al destructor, ya que este es invocado de forma automática cuando se destruye el objeto por los medios mencionados anteriormente. Si hiciera falta, el destructor también puede ser invocado de forma explícita:

```
Objeto.nombre_clase::~nombre_clase;    //llamada al destructor
Objeto->nombre_clase::~nombre_clase;   //cuando el objeto ha sido creado con new
    class Lolo{                        //lolo.h
      public:
        float real,imaginario;
      public:
           Lolo();                      //constructor
        ~Lolo();                       //destructor
    };
    Lolo::Lolo(){                      //lolo.cpp
                                       //codigo necesario para inicial.
        real = imaginario = 0;
    }
    Lolo::~Lolo(){
                                       //codigo necesario para liberar memoria.
        cout << «Se ha llamado al destructor\n»;
      }
```

Para cada clase solo se puede definir un destructor, ya que el destructor no puede recibir parámetros y por tanto no se puede sobrecargar. Ya hemos dicho que los destructores no pueden ser constantes, volátiles ni estáticos, pero si pueden ser declarados virtuales (ya veremos más adelante que quiere decir esto). Para declarar un destructor escribimos dentro de la declaración de la clase el símbolo ~ seguido del nombre de la clase. Se emplea el símbolo ~ para indicar que el destructor es el complemento del constructor. Veamos un ejemplo:

```
class X {
    private:
    int *ptr;
    public:
```

```
        X(int =1);                    // constructor
        ~X();                         // destructor
};

                                      // declaración del constructor
X::X(int i){
        ptr = new int [i];
}
                                       // declaración del destructor
X::~X() {
        delete []ptr;
}
```

13.5. Miembros static de una clase

Cada objeto tiene una copia de los datos miembro de la clase. Un dato miembro de una clase declarado como **static** implica que solo existirá una copia de ese dato miembro para todos los objetos y existe aunque no existan objetos de esa clase, con lo cual se concluye que es un dato asociado con la clase y no con el objeto. Un dato miembro **static**:

- Puede ser declarado static, private o public.
- Tiene que ser inicializado a nivel global (ámbito de fichero, no de clase).
- Cuando sea accedido a través de una función miembro, dicha función también tiene que ser declarada static.
- Existe aunque no haya ningún objeto de la clase declarado.

Al igual que los atributos estáticos mencionados en un punto anterior, las funciones miembro estáticas son globales para los miembros de la clase y deben ser definidas fuera del ámbito de la declaración de la clase. Estos métodos son siempre públicos, se declaren donde se declaren. Al no tener parámetro this no pueden acceder a los miembros no estáticos de la clase (al menos directamente, ya que se le podría pasar un puntero al objeto para que modificara lo que fuera).

13.6. Funciones amigas (friend)

Son funciones que tienen acceso a los miembros privados de una clase sin ser miembros de la misma. Se emplean para evitar la ineficiencia que supone el tener que acceder a los miembros privados de una clase a través de métodos. Como son funciones independientes de la clase no tienen parámetro this, por lo que el acceso a objetos de una clase se consigue pasándoles como parámetro una

referencia al objeto (una referencia como tipo implica pasar el objeto sin copiar, aunque se trata como si fuera el objeto y no un puntero), un puntero o el mismo objeto. Por la misma razón, no tienen limitación de acceso, ya que se definen fuera de la clase. Para hacer amiga de una clase a una función debemos declararla dentro de la declaración de la clase precedida de la palabra friend:

```
class X {
   private:
   int i;
   ...
   friend int f(X&, int);      // función amiga que toma como parámetros una referencia
                               // a un objeto del tipo X y un entero y retorna un entero
}
```

En la definición de la función (que se hace fuera de la clase como las funciones normales) podremos usar y modificar los miembros privados de la clase amiga sin ningún problema:

```
int f(X& objeto, int i) {
   int j = objeto.i;
   objeto.i = i;
   return j;
}
```

Es importante ver que aunque las funciones amigas no pertenecen a la clase se declaran explícitamente en la misma, por lo que forman parte de la **interface** de la clase. Una función miembro de una clase puede ser amiga de otra:

```
class X {
   ...
   void f();
   ...
};
class Y {
   ...
   friend void X::f();
};
```

Si queremos que todas las funciones de una clase sean amigas de una clase podemos poner:

```
class X {
   friend class Y;
   ...
};
```

En el ejemplo todas las funciones de la clase Y son amigas de la clase X, es decir, todos los métodos de Y tienen acceso a los miembros privados de X.

Resumiendo, los miembros de una clase declarados como privados solamente pueden ser accedidos mediante las funciones miembro de la clase, garantizando así la integridad de los datos. Una función declarada **friend** (amiga) de una clase puede acceder a los miembros privados y protegidos de la clase C++. Para declarar una función amiga, basta con escribir la palabra clave **friend** delante del nombre de la función.

13.7. Variables locales

El constructor de una variable local se ejecuta cada vez que encontramos la declaración de la variable local y su destructor se ejecuta cuando salimos del ámbito de la variable. Para ejecutar un constructor distinto del constructor por defecto al declarar una variable hacemos:

```
Complejo c (1, -1); // Crea el complejo c llamando al constructor
// Complejo (float, float)
```

Y para emplear el constructor copia para inicializar un objeto hacemos:

```
Complejo d = c; // crea el objeto d usando el constructor copia
// Complejo(Complejo&)
```

Si definimos c y luego d, al salir del bloque de la variable primero llamaremos al destructor de d, y luego al de c.

13.8. Almacenamiento estático

Cuando declaramos objetos de tipo estático su constructor se invoca al arrancar el programa y su destructor al terminar. Un ejemplo de esto está en los objetos cin, cout y cerr. Estos objetos se crean al arrancar el programa y se destruyen al acabar. Como siempre, constructores y destructores se ejecutan en orden inverso. El único problema con los objetos estáticos está en el uso de la función exit(). Cuando llamamos a exit() se ejecutan los destructores de los objetos estáticos, luego usar exit() en uno de ellos provocaría una recursión infinita. Si terminamos un programa con la función abort() los destructores no se llaman.

13.9. Almacenamiento dinámico

Cuando creamos objetos dinámicos con new ejecutamos el constructor utilizado para el objeto. Para liberar la memoria ocupada debemos emplear el operador delete, que se encargará de llamar al destructor. Si no usamos delete no tenemos ninguna garantía de que se llame al destructor del objeto.

Para crear un objeto con new ponemos: *Complejo *c= new Complejo (1); y para destruirlo: delete c;*

El usuario puede redefinir los operadores new y delete y puede modificar la forma de interacción de los constructores y destructores con estos operadores. Veremos todo esto al hablar de sobrecarga de operadores. La creación de arrays de objetos se discute más adelante.

13.10. Objetos como miembros

Cuando definimos una clase podemos emplear objetos como miembros, pero lo que no sabemos es como se construyen estos objetos miembro. Si no hacemos nada los objetos se construyen llamando a su constructor por defecto (aquel que no toma parámetros). Esto no es ningún problema, pero puede ser interesante construir los objetos miembro con parámetros del constructor del objeto de la clase que los define. Para hacer esto lo único que tenemos que hacer es poner en la *definición* del constructor los constructores de objetos miembro que queramos invocar. La sintaxis es poner dos puntos después del prototipo de la función constructora (en la definición, es decir, cuando implementamos la función) seguidos de una lista de constructores (invocados con el nombre del objeto, no el de la clase) separados por comas. El cuerpo de la definición de la función se pone después.

Estos constructores se llamaran en el orden en el que los pongamos y antes de ejecutar el constructor de la clase que los invoca. Veamos un ejemplo:

```
class cjto_de_tablas {
   private:
      tabla elementos;              // objeto de clase tabla
      tabla componentes;            // objeto de clase tabla
      int tam_tablas;
      ...
   public:
      cjto_de_tablas (int tam);     // constructor
      ~cjto_de_tablas ();           // destructor
      ...
};
cjto_de_tablas::cjto_de_tablas (int tam)
:elementos (tam), componentes(tam), tam_tablas(tam)
{
   ... // Cuerpo del constructor
}
```

Como se ve en el ejemplo podemos invocar incluso a los constructores de los objetos de tipos estándar. Si en el ejemplo no inicializáramos componentes el objeto se crearía invocando al constructor por defecto (el que no tiene parámetros, que puede ser un constructor nuestro con parámetros por defecto).

Este método es mejor que emplear punteros a objetos y construirlos con new en el constructor y liberarlos con delete en el destructor, ya que el uso de objetos dinámicos consume más memoria que los objetos estáticos (ya que usan un puntero y precisan llamadas al sistema para reservar y liberar memoria). Si dentro de una clase necesitamos miembros objeto pero no necesitamos que sean dinámicos emplearemos objetos miembro con la inicialización en el constructor de la clase.

- **Arrays de objetos**

  Al igual que con los tipos estándar, también es posible crear **arrays** de objetos:

  - Forma estática, utilizando los corchetes:

    Nombre_clase Nombre_array[num_objetos];

  - Forma dinámica, usando ***new***:

    *Nombre_clase *Nom_Puntero = new Nombre_clase[num];*

  Notas.

  - Para crear un puntero a objeto se utiliza **new**.
  - Para acceder a los miembros se usa el operador (->).
  - Hemos de acordarnos de llamar a delete cuando ya no necesitemos el array.

  Para declarar un array de objetos de una clase determinada es imprescindible que la clase tenga un constructor por defecto (que como ya hemos dicho es uno que no recibe parámetros pero puede tener parámetros por defecto). Al declarar el array se crearán tantos objetos como indiquen los índices llamando al constructor por defecto. La destrucción de los elementos para arrays estáticos se realiza por defecto al salir del bloque de la declaración (igual que con cualquier tipo de objetos estáticos), pero cuando creamos un array dinámicamente se siguen las mismas reglas explicadas al hablar de new y delete, es decir, debemos llamar a delete indicándole que queremos liberar un array. Veamos varios ejemplos:

```
tabla at[20];                  // array de 20 tablas, se llama a los constructores por defecto
void f(int tam) {
tabla *t1 = new tabla;         // puntero a un elemento de tipo tabla
tabla *t2 = new tabla [tam];         // puntero a un array de 'tam' tablas
...
delete t1;                     // destrucción de un objeto
detele []t2;                   // destrucción de un array
}
```

14. Herencia en C++

Una **clase derivada** puede definirse a partir de otra clase ya existente (**clase base**), de la que **hereda** sus variables y funciones miembro. La clase derivada puede añadir y/o redefinir nuevas variables y/o funciones miembro. La clase base suele ser más general que la clase derivada. Esta añade nuevas determinaciones o especificaciones (nuevas variables y/o funciones miembro). A su vez, la clase derivada puede ser clase base de una nueva clase derivada, que hereda sus variables y funciones miembro. Se puede constituir una jerarquía de clases. Además de public y private, C++ permite también definir miembros protected. Los miembros protected, al igual que los private, no son accesibles desde fuera de la clase. En una clase base, los miembros protected se diferencian de los private en que sí pueden ser accesibles para las clases derivadas de dicha clase base.

Para la clase derivada, la clase base se puede heredar como pública o como privada:

- La clase derivada no tiene acceso a los miembros private de la clase base. Sí tiene acceso a los miembros public y protected.
- Si la clase base se hereda como public, la clase derivada hereda los miembros public y protected de la clase base como miembros public y protected, respectivamente.
- Si la clase base se hereda como private, la clase derivada hereda todos los miembros de la clase base como private.

14.1. Constructores de clases derivadas

Un objeto de una clase derivada contiene todos los miembros de la clase base. El constructor de la clase derivada debe llamar al de la clase base. Cuando se define un constructor para una clase derivada, se debe especificar un inicializador base (llamada al constructor de la clase base).

El inicializador base se especifica poniendo, a continuación de los argumentos del constructor, el carácter ":" y un constructor de la clase base seguido de una lista de argumentos entre paréntesis. Al declarar un objeto de la clase derivada, se ejecuta primero el constructor de la clase base y luego el de la clase derivada. El inicializador base puede ser omitido si la clase base tiene un constructor por defecto. El constructor de una clase derivada debe disponer de valores para sus propias variables y para el constructor de la clase base.

```
popo::popo(const char *nombre) : lolo (nombre) {
.....
.....
};
```

Definimos las clases como antes, pero intentamos dar unas clases base o clases padre para representar las características comunes de las clases y luego

definimos unas clases derivadas o subclases que definen tan solo las características diferenciadoras de los objetos de esa clase. Por ejemplo, si queremos representar empleados y clientes podemos definir una clase base persona que contenga las características comunes de ambas clases (nombre, DNI, etc.) y después declararemos las clases empleado y cliente como derivadas de persona, y solo definiremos los miembros que son nuevos respecto a las personas o los que tienen características diferentes en la clase derivada, por ejemplo un empleado puede ser despedido, tiene un sueldo, puede firmar un contrato, etc., mientras que un cliente puede tener una cuenta, una lista de pedidos, puede firmar un contrato, etc. Como se ha mencionado ambos tipos pueden firmar contratos, pero los métodos serán diferentes, ya que la acción es la misma pero tiene significados distintos. En definitiva, introducimos los mecanismos de la **herencia** y **polimorfismo** para implementar las relaciones entre las clases. La herencia consiste en la definición de clases a partir de otras clases, de tal forma que la clase derivada hereda las características de la clase base, mientras que el polimorfismo nos permite que métodos declarados de la misma manera en una clase base y una derivada se comporten de forma distinta en función de la clase del objeto que la invoque, el método es **polimórfico**, tiene varias formas.

14.2. Clases derivadas o subclases

Una clase derivada es una clase que se define en función de otra clase. La sintaxis es muy simple: declaramos la clase como siempre, pero después de nombrar la clase escribimos dos puntos y el nombre de su clase base. Esto le indica al compilador que todos los miembros de la clase base se heredan en la nueva clase. Por ejemplo, si tenemos la clase empleado (derivada de persona) y queremos definir la clase directivo podemos declarar esta última como derivada de la primera. Así, un directivo tendrá las características de persona y de empleado, pero definirá además unos nuevos atributos y métodos propios de su clase:

```
class directivo : empleado {
    private:
        long num_empleados;
        long num_acciones;
        ...
    public:
        ...
        void despide_a (empleado *e);
        void reunion_con (directivo *d);
        ...
};
```

Como un objeto de tipo directivo es un empleado, se podrá usar en los lugares en los que se trate a los empleados, pero no al revés (un empleado no puede usarse cuando necesitamos un directivo). Esto es cierto cuando trabajamos con punteros a objetos, no con objetos:

```
directivo d1, d2;
empleado e1;
lista_empleados *le;
le= &d1; // inserta un directivo en la lista de empleados
d1.next = &e1; // el siguiente empleado es e1
e1.next = &d2; // el siguiente empleado es el directivo 2
d1.despide_a (&e1); // el directivo puede despedir a un empleado
d1.despide_a (&d2); // o a otro directivo, ya que también es un empleado
e1.despide_a (&d1); // ERROR, un empleado no tiene definido el método despide a
d1.reunion_con (&d2); // Un directivo se reúne con otro
d1.reunion_con (&e); // ERROR, un empleado no se reúne con un directivo
empleado *e2 = &d2; // CORRECTO, un directivo es un empleado
directivo *d3 = &e; // ERROR, no todos los empleados son directivos
d3->num_empleados =3; // Puede provocar un error, ya que e1 no tiene espacio
// reservado para num_empleados
d3 = (directivo *)e2. // CORRECTO, e2 apunta a un directivo
d3->num_empleados =3; // CORRECTO, d3 apunta a un directivo
```

En definitiva, un objeto de una clase derivada se puede usar como objeto de la clase base si se maneja con punteros, pero hay que tener cuidado ya que el C++ no realiza chequeo de tipo dinámico (no tiene forma de saber que un puntero a un tipo base realmente apunta a un objeto de la clase derivada).

A) Funciones miembro en clases derivadas

En el ejemplo del punto anterior hemos definido nuevos miembros (podemos definir nuevos atributos y métodos, e incluso atributos de la clase derivada con los mismos nombres que atributos de la clase base de igual o distinto tipo) para la clase derivada, pero, ¿cómo accedemos a los miembros de la clase base desde la derivada? Si no se redefinen, podemos acceder a los atributos de la forma habitual y llamar a los métodos como si estuvieran definidos en la clase derivada, pero si se redefinen para acceder al miembro de la clase base debemos emplear el operador de campo aplicado al nombre de la clase base (en caso contrario accedemos al miembro de la clase derivada):

```
class empleado {
   ...
   void imprime_sueldo();
   void imprime_ficha ();
   ...
```

```
}
class directivo : empleado {
   ...
   void imprime_ficha () {
   imprime_sueldo();
   empleado::imprime_ficha();
   ...
}
...
};
directivo d;
d.imprime_sueldo ();              // se llama al método implementado para empleado, ya
                                  // que la clase directivo no define el método
d.imprime_ficha ();               // se llama al método definido en directivo
d.empleado::imprime_ficha ();     // llamamos al método de la clase base empleado
```

14.3. Constructores y destructores

Algunas clases derivadas necesitan constructores, y si la clase base de una clase derivada tiene un constructor este debe ser llamado proporcionándole los parámetros que necesite. En realidad, la gestión de las llamadas a los constructores de una clase base se gestionan igual que cuando definimos objetos miembro, es decir, se llaman en el constructor de la clase derivada de forma implícita si no ponemos nada (cuando la clase base tiene un constructor por defecto) o de forma explícita siempre que queramos llamar a un constructor con parámetros (o cuando esto es necesario). La única diferencia con la llamada al constructor respecto al caso de los objetos miembro es que en este caso llamamos al constructor con el nombre de la clase y no del objeto, ya que aquí no existe. Veamos un ejemplo:

```
class X {
   ...
   X();      // constructor sin param
   X (int); // constructor que recibe un entero
   ~X();     // destructor
};
class Y : X {
   ...
   Y();      // constructor sin param
```

```
    Y(int);  // constructor con un parámetro entero
    Y (int, int)        ; // constructor con dos parámetros enteros
    ...
};
    // constructor sin param, invoca al constructor por defecto de X
Y::Y() {
    ...
}
    // constructor con un parámetro entero, invoca al constructor que recibe un entero  de la clase X
Y::Y(int i) : X(i) {
    ...
}
    // constructor con dos parámetros enteros, invoca al constructor por defecto de X
Y::Y (int i , int j) {
...
}
```

14.4. Las jerarquías de clases

Como ya hemos visto las clases derivadas pueden a su vez ser clases base de otras clases, por lo que es lógico pensar que las aplicaciones en las que definamos varias clases acabemos teniendo una estructura en árbol de clases y subclases. En realidad esto es lo habitual, construir una jerarquía de clases en las que la clase base es el tipo objeto y a partir de él cuelgan todas las clases. Esta estructura tiene la ventaja de que podemos aplicar determinadas operaciones sobre todos los objetos de la clase, como por ejemplo, mantener una estructura de punteros a objeto de todos los objetos dinámicos de nuestro programa o declarar una serie de variables globales en la clase raíz de nuestra jerarquía que sean accesibles para todas las clases pero no para funciones definidas fuera de las clases.

Aparte del diseño en árbol se utiliza también la estructura de bosque: definimos una serie de clases sin descendencia común, pero que crean sus propios árboles de clases. Generalmente, se utiliza un árbol principal y luego una serie de clases contenedor que no están en la jerarquía principal y, por tanto, pueden almacenar objetos de cualquier tipo sin pertenecer realmente a la jerarquía (si están junto con el árbol principal podemos llegar a hacer programas muy complejos de forma innecesaria, ya que una pila podría almacenarse a sí misma, causando problemas a la hora de destruir objetos).

No siempre la estructura es un árbol, ya que la idea de herencia múltiple provoca la posibilidad de interdependencia entre nodos de ramas distintas, por lo que sería más correcto hablar de grafos en vez de árboles.

14.5. Los métodos virtuales

El C++ permite el empleo de funciones polimórficas, que son aquellas que se declaran de la misma manera en distintas clases y se definen de forma diferente. En función del objeto que invoque a una función polimórfica se utilizará una función u otra. En definitiva, una función polimórfica será aquella que tendrá formas distintas según el objeto que la emplee. Los métodos virtuales son un mecanismo proporcionado por el C++ que nos permiten declarar funciones polimórficas. Cuando definimos un objeto de una clase e invocamos a una función virtual, el compilador llamará a la función correspondiente a la de la clase del objeto. Para declarar una función como virtual basta poner la palabra virtual antes de la declaración de la función en la declaración de la clase. Una función declarada como virtual debe ser definida en la clase base que la declara (excepto si la función es virtual pura), y podrá ser empleada aunque no haya ninguna clase derivada. Las funciones virtuales solo se redefinen cuando una clase derivada necesita modificar la de su clase base.

Una vez se declara un método como virtual esa función sigue siéndolo en todas las clases derivadas que lo definen, aunque no lo indiquemos. Es recomendable poner siempre que la función es virtual, ya que si tenemos una jerarquía grande se nos puede olvidar que la función fue declarada como virtual.

Para gestionar las funciones virtuales el compilador crea una tabla de punteros a función para las funciones virtuales de la clase, y luego cada objeto de esa clase contendrá un puntero a dicha tabla. De esta manera tenemos dos niveles de indirección, pero el acceso es rápido y el incremento de memoria escaso. Al emplear el puntero a la tabla el compilador utiliza la función asociada al objeto, no la función de la clase que tenga el objeto en el momento de invocarla.

Empleando funciones virtuales nos aseguramos que los objetos de una clase usarán sus propias funciones virtuales aunque se estén accediendo a través de punteros a objetos de un tipo base. Ejemplo:

```
class empleado {
   ...
   virtual void imprime_sueldo() const;
   virtual void imprime_ficha () const;
   ...
}
class directivo : empleado {
   ...
   virtual void imprime_ficha () const;      // no es necesario poner virtual
...
};
directivo d;
```

```
empleado e;
d.imprime_ficha ();          // llamamos a la función de directivo
e.imprime_ficha ();          // llamamos a la función de empleado
d.imprime_sueldo();          // llamamos a la función de empleado, ya que aunque es
                             // virtual, la clase directivo no la redefine
empleado *pe = &d;
pe->imprime_sueldo();        // pe apunta a un directivo, llamamos a la función de la
                             // clase directivo, que es la asociada al objeto d
```

La tabla se crea al construir el objeto por lo que los constructores no podrán ser virtuales, ya que no disponemos del puntero a la tabla hasta terminar con el constructor. Por esa misma razón hay que tener cuidado al llamar a funciones virtuales desde un constructor: llamaremos a la función de la clase base, no a la que redefina nuestra clase. Los destructores sí pueden ser declarados virtuales. Las funciones virtuales necesitan el parámetro this para saber qué objeto las utiliza y, por tanto, no pueden ser declaradas static ni friend. Una función friend no es un método de la clase que la declara como amiga, por lo que tampoco tendría sentido definirla como virtual. De cualquier forma, dijimos que una clase puede tener como amigos métodos de otras clases. Pues bien, estos métodos amigos pueden ser virtuales; si nos fijamos un poco, la clase que declara una función como amiga no tiene por qué saber si esta es virtual o no.

14.6. Clases abstractas

Ya hemos mencionado lo que son las jerarquías de clases, pero hemos dicho que se pueden declarar objetos de cualquiera de las clases de la jerarquía. Esto tienen un problema importante: al definir una jerarquía es habitual definir clases que no queremos que se puedan instanciar, es decir, clases que solo sirven para definir el tipo de atributos y mensajes comunes para sus clases derivadas: son las denominadas clases abstractas.

En estas clases es típico definir métodos virtuales sin implementar, es decir, métodos que dicen cómo debe ser el mensaje pero no qué se debe hacer cuando se emplean con objetos del tipo base. Este mecanismo nos obliga a implementar estos métodos en todas las clases derivadas, haciendo más fácil la consistencia de las clases.

Pues bien, el C++ define un mecanismo para hacer esto (ya que si no lo hiciera deberíamos definir esos métodos virtuales con un código vacío, lo que no impediría que declaráramos subclases que no definieran el método y además permitiría que definiéramos objetos del tipo base abstracto).

La idea es que podemos definir uno o varios métodos como virtuales puros o abstractos (sin implementación), y esto nos obliga a redeclararlos en todas las clases derivadas (siempre que queramos definir objetos de estas subclases). Además, una clase con métodos abstractos se considera una clase abstracta y por tanto no podemos definir objetos de esa clase.

Para declarar un método como abstracto solo tenemos que igualarlo a cero en la declaración de la clase (escribimos un igual a cero después del prototipo del método, justo antes del punto y coma, como cuando inicializamos variables):

```
class X {
   private:
      ...
   public:
      X();
      ~X();
      virtual void f(int) = 0;        // método abstracto, no debemos definir la función para esta clase
      ...
}
class Y : public X {
   ...
   virtual void f(int);        // volvemos a declarar f, deberemos definir el método para la clase Y
   ...
}
```

Lo único que resta por mencionar de las funciones virtuales puras es que no tenemos por qué definirlas en una subclase de una clase abstracta si no queremos instanciar objetos de esa subclase. Esto se puede producir cuando de una clase abstracta derivan subclases para las que nos interesa definir objetos y también subclases que van a servir de clases base abstractas para nuevas clases derivadas.

Una subclase de una clase abstracta será abstracta siempre que no redefinamos *todas* las funciones virtuales puras de la clase padre. Si redefinimos algunas de ellas, las clases que deriven de la subclase abstracta solo necesitarán implementar las funciones virtuales puras que su clase padre (la derivada de la abstracta original) no haya definido.

14.7. Herencia múltiple

La idea de la herencia múltiple es bastante simple, aunque tiene algunos problemas a nivel de uso. Igual que decíamos que una clase podía heredar características de otra, se nos puede ocurrir que una clase podría heredar características de más de una clase. El ejemplo típico es la definición de la clase de vehículos *anfibios*; como sabemos los anfibios son vehículos que pueden circular por tierra o por mar. Por tanto, podríamos definir los anfibios como elementos que heredan características de los vehículos terrestres y los vehículos marinos. La sintaxis para expresar que una clase deriva de más de una clase base es simple, ponemos el nombre de la nueva clase, dos puntos y la lista de clases padre:

```
class anfibio : terrestre, marino {
...
};
```

Los objetos de la clase derivada podrán usar métodos de sus clases padre y se podrán asignar a punteros a objetos de esas clases. Las funciones virtuales se tratan igual, etc. Todo lo que hemos comentado hasta ahora es que la herencia múltiple es como la simple, excepto por el hecho de que tomamos (heredamos) características de dos clases. Pero no todo es tan sencillo, existen una serie de problemas que comentaremos en los puntos siguientes.

14.8. Ocurrencias múltiples de una base

Con la posibilidad de que una clase derive de varias clases es fácil que se presente el caso de que una clase tenga una clase como clase más de una vez. Por ejemplo, en el caso del anfibio tenemos como base las clases *terrestre* y *marino*, pero ambas clases podrían derivar de una misma clase base *vehículo*. Esto no tiene por qué crear problemas, ya que podemos considerar que los objetos de la clase anfibio contienen objetos de las clases *terrestre* y *marino*, que a su vez contienen objetos diferentes de la clase *vehículo*. De todas formas, si intentamos acceder a miembros de la clase *vehículo*, aparecerán ambigüedades. A continuación veremos cómo podemos resolverlas.

14.9. Resolución de ambigüedades

Evidentemente, dos clases pueden tener miembros con el mismo nombre, pero cuando trabajamos con herencia múltiple esto puede crear ambigüedades que deben ser resueltas. El método para acceder a miembros con el mismo nombre en dos clases base desde una clase derivada es emplear el operador de campo, indicando cuál es la clase del miembro al que accedemos:

```
class terrestre : vehiculo {
    ...
    char *Tipo_Motor;
    ...
    virtual void imprime_tipo_motor() { cout << Tipo_Motor; }
    ...
};
class marino : vehiculo {
    ...
    char *Tipo_Motor;
    ...
```

```
    virtual void imprime_tipo_motor(); { cout << Tipo_Motor; }
    ...
    };
    class anfibio : terrestre, marino {
    ...
    virtual void imprime_tipo_motor();
    ...
};
void anfibio::imprime_tipo_motor () {
    cout << «Motor terrestre : «;
    terrestre::imprime_tipo_motor ();
    cout << «Motor acuático : «;
    marino::imprime_tipo_motor ();
}
```

Lo habitual es que la ambigüedad se produzca al usar métodos (ya que los atributos suelen ser privados y, por tanto, no accesibles para la clase derivada), y la mejor solución es hacer lo que se ve en el ejemplo: redefinir la función conflictiva para que utilice las de las clases base. De esta forma los problemas de ambigüedad se resuelven en la clase y no tenemos que emplear el operador de campo desde fuera de esta (al llamar al método desde un objeto de la clase derivada). Si intentamos acceder a miembros ambiguos el compilador no generará código hasta que resolvamos la ambigüedad.

14.10. Clases base virtuales

Las clases base que hemos empleado hasta ahora con herencia múltiple tienen la suficiente entidad como para que se declararen objetos de esas clases, es decir, heredábamos de dos o más clases porque en realidad los objetos de la nueva clase se componían o formaban a partir de otros objetos. Esto está muy bien, y suele ser lo habitual, pero existe otra forma de emplear la herencia múltiple: el **hermanado de clases**.

El mecanismo de hermanado se basa en lo siguiente: para definir clases que toman varias características de clases derivadas de una misma clase. Es decir, definimos una clase base y derivamos clases que le añaden características y luego queremos usar objetos que tengan varias de las características que nos han originado clases derivadas. En lugar de derivar una clase de la base que reúna las características, podemos derivar una clase de las subclases que las incorporen. Por ejemplo, si definimos una clase ventana y derivamos las clases *ventana_con_borde y ventana_con_menu, en lugar de derivar de la clase ventana una clase ventana_con_menu_y_borde*, la derivamos de las dos subclases. En realidad lo que queremos es emplear un mismo objeto de la clase base *ventana*, por lo que nos interesa que las dos subclases generen sus objetos a partir de un mismo objeto *ventana*. Esto se consigue declarando la herencia de la clase

base como virtual en todas las subclases que quieran compartir su padre con otras subclases al ser empleadas como clase base, y también en las subclases que la hereden desde varias clases distintas:

```
class ventana { };
class ventana_con_borde : public virtual ventana { };
class ventana_con_menu : public virtual ventana { };
class ventana_con_menu_y_borde
: public virtual ventana,
public ventana_con_borde,
public ventana_con_menu { };
```

El problema que surge en estas clases es que los métodos de la clase base común pueden ser invocados por dos métodos de las clases derivadas y que, al agruparse en la nueva clase, generen dos llamadas al mismo método de la clase base inicial. Por ejemplo, en el caso de la clase *ventana*, supongamos que definimos un método *dibujar*, que es invocado por los métodos dibujar de las clases *ventana_con_borde y ventana_con_menu*.

Para definir el método *dibujar* de la nueva clase *ventana_con_menu_y_ borde* lo lógico sería llamar a los métodos de sus funciones padre, pero esto provocaría que llamáramos dos veces al método *dibujar* de la clase *ventana*, provocando no solo ineficiencia, sino incluso errores (ya que el redibujado de la ventana puede borrar algo que no debe borrar, por ejemplo el menú). La solución pasaría por definir dos funciones de dibujo, una virtual y otra no virtual: usaremos la virtual para dibujar objetos de la clase (por ejemplo, ventanas con marco) y la no virtual para dibujar solo lo característico de nuestra clase. Al definir la clase que agrupa características llamaremos a las funciones no virtuales de las clases padre, evitando que se repitan llamadas. Otro problema con estas clases es que si dos funciones hermanas redefinen un método de la clase padre (como el método *dibujar* anterior), la clase que herede de ambas deberá redefinirla para evitar ambigüedades (¿a qué función se llama si la subclase no redefine el método?).

14.11. Control de acceso

Como ya comentamos en puntos anteriores, los miembros de una clase pueden ser privados, protegidos o públicos (private, protected, public). El acceso a los miembros privados está limitado a funciones miembro y amigas de la clase; el acceso protegido es igual que el privado, pero también permite que accedan a ellos las clases derivadas; y los miembros públicos son accesibles desde cualquier sitio en el que la clase sea accesible.

El único modelo de acceso que no hemos estudiado es el protegido. Cuando implementamos una clase base podemos querer definir funciones que puedan utilizar las clases derivadas pero que no se puedan usar desde fuera de la clase. Si definimos miembros como privados tenemos el problema de que la clase derivada tampoco puede acceder a ellos. La solución es definir esos métodos como protected.

Estos niveles de acceso reflejan los tipos de funciones que acceden a las clases: las funciones que la implementan, las que implementan clases derivadas y el resto. Ya se ha mencionado que dentro de la clase podemos definir prácticamente cualquier cosa (tipos, variables, funciones, constantes, etc.). El nivel de acceso se aplica a los nombres, por lo que lo que podemos definir como privados, públicos o protegidos no solo los atributos, sino todo lo que puede formar parte de la clase.

Aunque los miembros de una clase tienen definido un nivel de acceso, también podemos especificar un nivel de acceso a las clases base desde clases derivadas. El nivel de acceso a clases base se emplea para saber quién puede convertir punteros a la clase derivada en punteros a la clase base (de forma implícita, ya que con casts siempre se puede) y acceder a miembros de la clase base heredados en la derivada. Es decir, una clase con acceso private a su clase base puede acceder a su clase base, pero ni sus clases derivadas ni otras funciones tienen acceso a la misma, es como si definiéramos todos los miembros de la clase base como private en la clase derivada. Si el acceso a la clase base es protected, solo los miembros de la clase derivada y los de las clases derivadas de esta última tienen acceso a la clase base. Y si el acceso es público, el acceso a los miembros de la clase base es el especificado en ella. Para especificar el nivel de acceso a la clase base ponemos la etiqueta de nivel de acceso antes de escribir su nombre en la definición de una clase derivada. Si la clase tiene herencia múltiple, debemos especificar el acceso de todas las clases base. Si no ponemos nada, el acceso a las clases base se asume public. Ejemplo:

```
class anfibio : public terrestre, protected marino {
...
};
```

- **Gestión de memoria**

Cuando creamos objetos de una clase derivada se llama a los constructores de sus clases base antes de ejecutar el de la clase, y luego se ejecuta el suyo. El orden de llamada a los destructores es el inverso, primero el de la clase derivada y luego el de sus padres. Comentamos al hablar de métodos virtuales que los destructores podían ser declarados como tales; la utilidad de esto es clara: si queremos destruir un objeto de una clase derivada usando un puntero a una clase base y el destructor no es virtual la destrucción será errónea, con los problemas que esto puede traer. De hecho, casi todos los compiladores definen un flag para que los destructores sean virtuales por defecto. Lo más típico es declarar los destructores como virtuales siempre que en una clase se defina un método virtual, ya que es muy posible que se manejen punteros a objetos de esa clase.

Además de comentar la forma de llamar a constructores y destructores, en este punto se podría hablar de las posibilidades de sobrecarga de los operadores new y delete para clases, ya que esta sobrecarga nos permite modificar el modo en que se gestiona la memoria al crear objetos. Como el siguiente punto es la sobrecarga de operadores estudiaremos esta posibilidad en ella. Solo decir que la sobrecarga de la gestión de memoria es especialmente interesante en las clases base, ya que si ahorramos memoria al trabajar con objetos de la clase base es evidente que la ahorraremos siempre que creemos objetos de clases derivadas.

15. Sobrecarga en C++

- **Sobrecarga de funciones**

La sobrecarga de funciones consiste en definir varias funciones con el mismo nombre diferenciándolas por los argumentos que son de distinto tipo y será el detalle que permite al compilador llamar a una u otra función.

```
#include <iostream.h>
void funcion(int x);   void funcion(double x);
void main(void)  {
   int y=10;       double x=7.2;
   funcion(x);    funcion(y);
}
void funcion(int a) {
   cout << «Valor entero: « << a << endl;
}
void funcion(double a) {
   cout << «Valor real: « << a << endl;
}
```

- **Sobrecarga de operadores**

Los operadores, al igual que las funciones, pueden ser sobrecargados:

- La mayor parte de los operadores de C++ pueden ser redefinidos para actuar sobre objetos de una clase.
- Se puede cambiar la definición de un operador, pero no su gramática: número de operandos, precedencia y asociatividad.
- El **tipo** de los **operandos** determina **qué definición** del operador se va a utilizar.
- Al menos uno de los **operandos** debe ser un objeto de la clase.

La sintaxis para declarar un operador sobrecargado es la siguiente:

Tipo_operator operador([argumentos]);

Donde:

tipo indica el tipo del valor retornado por la función

operador es unos de los siguientes: +,-,*,/,%,&,!,>,<,=,[ ],new, delete, ...

Sobrecarga del operador de indexación "[ ]":

- El operador de indexación, **operator[]**, permite manipular los objetos de clases igual que si fuesen arrays.
- La llamada a la función **operator[]** de una clase se hace escribiendo el nombre de un objeto de la clase para el cual se quiere invocar dicha función, seguido de un valor encerrado entre corchetes.
- La forma de sobrecargar el operador es:

  int& operator[](int i);

Esta capacidad se traduce en poder definir un significado para los operadores cuando los aplicamos a objetos de una clase específica. Además de los operadores aritméticos, lógicos y relacionales, también la llamada (), el subíndice [] y la de referencia -> se pueden definir, e incluso la asignación y la inicialización pueden redefinirse. También es posible definir la conversión implícita y explícita de tipos entre clases de usuario y tipos del lenguaje.

15.1. Funciones operador

Se pueden declarar funciones para definir significados para los siguientes operadores:

+ - * / % ^ & | ~ !

= < > += -= *= /= %= ^= &=

|= << >> <<= >>= == != <= >= &&

|| ++ — ->* , -> [] () new delete

No podemos modificar ni las precedencias ni la sintaxis de las expresiones para los operadores, ya que podríamos provocar ambigüedades. Tampoco podemos definir nuevos operadores. El nombre de una función operador es la palabra clave operator seguida del operador, por ejemplo operator+. Al emplear un operador podemos llamar a su función o poner el operador, el uso del operador solo es para simplificar la escritura. Por ejemplo:

int c = a + b;

es lo mismo que:

int c = operator+ (a, b);

15.2. Asignación e inicialización

La asignación entre objetos de un mismo tipo definido por el usuario puede crear problemas; por ejemplo, si tenemos la clase cadena:

```
class cadena {
    private:
        char *p; // puntero a cadena
        int tam; // tamaño de la cadena apuntada por p
    public:
        cadena (int t) { p = new char [tam =t] }
        ~cadena () { delete []p; }
}
la operación:
cadena c1(10);
cadena c2(20);
c2 = c1;
```

asignará a c2 el puntero de c1, por lo que al destruir los objetos dejaremos la cadena c2 original sin tocar y llamaremos dos veces al destructor de c1. Esto se puede resolver redefiniendo el operador de asignación:

```
class cadena {
    ...
    cadena& operator= (const cadena&); // operador de asignación
}
cadena& cadena::operator= (const cadena& a) {
    if (this != &a) { // si no igualamos una cadena a si misma
        delete []p;
        p = new char[tam = a.tam];
        strncpy (p, a.p);
    }
    return *this; // nos retornamos a nosotros mismos
}
```

Con esta definición resolvemos el problema anterior, pero aparece un nuevo problema: hemos dado por supuesto que la asignación se hace para objetos inicializados pero, ¿qué pasa si en lugar de una asignación estamos haciendo una inicialización? Por ejemplo:

```
cadena c1(10);
cadena c2 = c1;
```

En esta situación solo construimos un objeto, pero destruimos dos. El operador de asignación definido por el usuario no se aplica a un objeto sin ini-

cializar; en realidad debemos definir un constructor copia para objetos de un mismo tipo. Este constructor es el que se llama en la inicialización:

```
class cadena {
   ...
   cadena (const cadena&); // constructor copia
}
cadena::cadena (const cadena& a) {
   p = new char[tam = a.tam];
   strncpy (p, a.p);
}
```

15.3. Sobrecarga de new y delete

Al igual que el resto de operadores, los operadores new y delete se pueden sobrecargar. Esto se emplea para crear y destruir objetos de formas distintas a las habituales: reservando el espacio de forma diferente o en posiciones de memoria que no están libres en el heap, inicializando la memoria a un valor concreto, etc.

El operador new tiene un parámetro obligatorio de tipo size_t y luego podemos poner todo tipo y número de parámetros. Su retorno debe ser un puntero void. El parámetro size_t es el tamaño en bytes de la memoria a reservar, si la llamada a new es para crear un vector size_t debe ser el número de elementos por el tamaño de la clase de los objetos del array.

Es muy importante tener claro lo que hacemos cuando redefinimos la gestión de memoria, y siempre que sobrecarguemos el new o el delete tener presente que ambos operadores están relacionados y ambos deben ser sobrecargados a la vez para reservar y liberar memoria de formas extrañas.

16. Templates

C++ es un lenguaje muy potente tal y como lo hemos definido hasta ahora, pero al ir incorporándole características se ha tendido a que no se perdiera eficiencia (dentro de unos márgenes) a cambio de una mayor comodidad y potencia a la hora de programar. C introdujo en su momento un mecanismo sencillo para facilitar la escritura de código: las macros. Una **macro** es una forma de representar expresiones; se trata en realidad de evitar la repetición de la escritura de código mediante el empleo de abreviaturas, sustituyendo una expresión por un nombre o un nombre con aspecto de función que luego se expande y sustituye las abreviaturas por código.

El mecanismo de las macros no estaba mal, pero tenía un grave defecto: el uso y la definición de macros se hace a ciegas en lo que al compilador se refiere. El mecanismo de sustitución que nos permite definir pseudo-funcio-

nes no realiza ningún tipo de chequeos y es por tanto poco seguro. Además, la potencia de las macros es muy limitada.

Para evitar que cada vez que definamos una función o una clase tengamos que replicar código en función de los tipos que manejemos (como parámetros en funciones o como miembros y retornos y parámetros de funciones miembro en clases), C++ introduce el concepto de **funciones** y **clases genéricas**.

Una función genérica es realmente como una plantilla de una función: lo que representa es lo que tenemos que hacer con unos datos sin especificar el tipo de algunos de ellos. Por ejemplo una función máximo se puede implementar igual para enteros, para reales o para complejos, siempre y cuando esté definido el operador de relación <. Pues bien, la idea de las funciones genéricas es definir la operación de forma general, sin indicar los tipos de las variables que intervienen en la operación. Una vez dada una definición general, para usar la función con diferentes tipos de datos, la llamaremos indicando el tipo (o los tipos de datos) que intervienen en ella. En realidad es como si le pasáramos a la función los tipos junto con los datos. Al igual que sucede con las funciones, las clases contenedor son estructuras que almacenan información de un tipo determinado, lo que implica que cada clase contenedor debe ser reescrita para contener objetos de un tipo concreto. Si definimos la clase de forma general, sin considerar el tipo que tiene lo que vamos a almacenar y luego le pasamos a la clase el tipo o los tipos que le faltan para definir la estructura, ahorraremos tiempo y código al escribir nuestros programas.

16.1. Funciones genéricas

Para definir una función genérica solo tenemos que poner delante de la función la palabra template seguida de una lista de nombres de tipos (precedidos de la palabra class) y separados por comas, entre los signos de menor y mayor. Los nombres de los tipos no se deben referir a tipos existentes, sino que deben ser como los nombres de las variables, identificadores.

Los tipos definidos entre mayor y menor se utilizan dentro de la clase como si de tipos de datos normales se tratara. Al llamar a la función el compilador sustituirá los tipos parametrizados en función de los parámetros actuales (por eso, todos los tipos parametrizados deben aparecer al menos una vez en la lista de parámetros de la función). Ejemplo:

template <class T> // solo un tipo parámetro

T max (T a, T b) { return (a>b) ? a : b } // función genérica máximo

Los tipos parámetro no solo se pueden usar para especificar tipos de variables o de retornos, también podemos usarlos dentro de la función para lo que queramos (definir variables, punteros, asignar memoria dinámica, etc.). En definitiva, los podemos usar para lo mismo que los tipos normales.

Todos lo modificadores de una función (inline, static, etc.) van después de template < ... >.

Las funciones genéricas se pueden sobrecargar y también especializar. Para sobrecargar una función genérica lo único que debemos hacer es redefinirla con distinto tipo de parámetros (haremos que emplee más tipos o que tome distinto número o en distinto orden los parámetros), y para especializar una función debemos implementarla con los tipos parámetro especificados (algunos de ellos al menos):

```
template <class T>
T max (T a, T b) { ... }      // función máximo para dos parámetros de tipo T,  sobrecarga de la función
template <class T>
T max (int *p, T a) { ... }   // función máximo para punteros a entero y valores de tipo T
                              // sobrecarga de la función
template <class T>
T max (T a[]) { ... }         // función genérica máximo para vectores de tipo T,  especialización
                              // función máximo para cadenas como punteros a carácter
const char* max(const char *c1, const char *c2) {
return (strncmp(c1, c2) >=1) ? c1 : c2;
}
                              // ejemplos de uso
int i1 = 9, i2 = 12;
cout << max (i1, i2);         // se llama a máximo con dos enteros, T=int
int *p = &i2;
cout << max (p, i1);          //llamamos a la función que recibe puntero y tipo T (T=entero)
cout << max («HOLA», «ADIOS»);    // se llama a la función especializada para trabajar  con cadenas.
```

Con las funciones especializadas lo que sucede es muy simple: si llamamos a la función y existe una versión que especifica los tipos, usamos esa. Si no encuentra la función, busca una función **template** de la que se pueda instanciar una función con los tipos de la llamada. Si las funciones están sobrecargadas resuelve como siempre, si no encuentra ninguna función aceptable, da un error.

16.2. Clases genéricas

También podemos definir clases genéricas de una forma muy similar a las funciones. Esto es especialmente útil para definir las clases contenedor, ya que los tipos que contienen solo nos interesan para almacenarlos y podemos definir las estructuras de una forma más o menos genérica sin ningún problema. Hay que indicar que si las clases necesitan comparar u operar de alguna forma con los objetos de la clase parámetro, las clases que usemos como parámetros actuales de la clase deberán tener sobrecargados los operadores que necesitemos. Para declarar una clase paramétrica hacemos lo mismo de antes:

```
template <class T>          // podríamos poner más de un tipo
class vector {
   T* v;    // puntero a tipo T
   int tam;
   public:
   vector (int);
   T& operator[] (int);     // el operador devuelve objetos de tipo T
   ...
}
```

Pero para declarar objetos de la clase debemos especificar los tipos (no hay otra forma de saber por que debemos sustituirlos hasta no usar el objeto):

```
vector<int> v(100); // vector de 100 elementos de tipo T = int
```

Una vez declarados los objetos se usan como los de una clase normal.

Para definir los métodos de la clase solo debemos poner la palabra template con la lista de tipos y al poner el nombre de la clase adjuntarle su lista de identificadores de tipo (igual que lo que ponemos en template pero sin poner class):

```
template <class T>
vector<T>::vector (int i) {
...
}
template <class T>
T& vector<T>::operator[] (int i) {
...
}
...
```

Al igual que las funciones genéricas, las clases genéricas se pueden especializar, es decir, podemos definir una clase especifica para unos tipos determinados e incluso especializar solo métodos de una clase. Lo único a tener en cuenta es que debemos poner la lista de tipos parámetro especificando los tipos al especificar una clase o un método:

```
// especializamos la clase para char *, podemos modificar totalmente la def. de la  clase
class vector <char *> {
        char *feo;
    public:
        vector ();
        void hola ();
}
        // Si solo queremos especializar un método
vector<float>::vector (int i) {
        ... // constructor especial para float
}
```

Además de lo visto el C++ permite que las clases genéricas admitan constantes en la lista de tipos parámetro:

```
template <class T, int SZ>
class pila {
    T bloque[SZ]; // vector de SZ elementos de tipo T
    ...
};
```

La única limitación para estas constantes es que deben ser conocidas en tiempo de compilación. Otra facilidad es la de poder emplear la herencia con clases parametrizadas, tanto para definir nuevas clases genéricas como para definir clases no genéricas. En ambos casos debemos indicar los tipos de la clase base, aunque para clases genéricas derivadas de clases genéricas podemos emplear tipos de nuestra lista de parámetros. Ejemplo:

```
template <class T, int SZ>
class pila {
   ...
}
   // clase template derivada
template <class T, class V>
class pilita : public pila<T, 20> {     // la clase base usa el tipo T y SZ vale 20
   ...
};
   // clase no template derivada
class pilita_chars : public pila<char, 50> {    // heredamos de la clase pila con  T=char y SZ=50
   ...
};
```

17. Manejo de excepciones

Existen varios tipos de errores a la hora de programar: los errores sintácticos y los errores de uso de funciones o clase y los errores del usuario del programa. Los primeros los debe detectar el compilador, pero el resto se deben detectar en tiempo de ejecución, es decir, debemos tener código para detectarlos y tomar las acciones oportunas. Ejemplos típicos de errores son el salirse del rango de un vector, divisiones por cero, desbordamiento de la pila, etc. Para facilitarnos el manejo de estos errores el C++ incorpora un mecanismo de tratamiento de errores más potente que el simple uso de códigos de error y funciones para tratarlos.

17.1. Tratamiento de excepciones en C++ (throw - catch - try)

La idea es la siguiente: en una cadena de llamadas a funciones los errores no se suelen tratar donde se producen, por lo que la idea es lanzar un mensaje de error desde el sitio donde se produce uno y ir pasándolo hasta que alguien se encargue de él. Si una función llama a otra y la función llamada detecta un error lo lanza y termina. La función llamante recibirá el error, si no lo trata, lo pasará a la función que la ha llamado a ella. Si la función recoge la excepción ejecuta una función de tratamiento del error. Además de poder lanzar y recibir errores, debemos definir un bloque como aceptor de errores. La idea es que probamos a ejecutar un bloque y si se producen errores los recogemos. En el resto de bloques del programa no se podrán recoger errores.

17.2. Lanzamiento de excepciones: throw

Si dentro de una función detectamos un error lanzamos una excepción poniendo la palabra **throw** y un parámetro de un tipo determinado, es como si ejecutáramos un return de un objeto (una cadena, un entero o una clase definida por nosotros). Por ejemplo:

```
f() {
   ...
   int *i;
   if ((i= new int) == NULL)
   throw «Error al reservar la memoria para i»;   // no hacen falta paréntesis,
   // es como en return
   ...
}
```

Si la función f() fue invocada desde g() y esta, a su vez, desde h(), el error se irá pasando entre ellas hasta que se recoja.

17.3. Recogida: catch

Para recoger un error empleamos la pseudofunción **catch**; esta instrucción se pone como si fuera una función, con catch y un parámetro de un tipo determinado entre paréntesis, después abrimos llave, escribimos el código de gestión del error y cerramos la llave. Por ejemplo si la función h() trataba el error anterior:

```
h() {
   ...
   catch (char *ce) {
      cout << «He recibido un error que dice : « << ce;
   }
...
}
```

Podemos poner varios bloques catch seguidos, cada uno recogerá un error de un tipo distinto. El orden de los bloques es el orden en el que se recogen las excepciones:

```
h() {
   ...
   catch (char *ce) {
      ... // tratamos errores que lanzan cadenas
   }
   catch (int ee) {
      ... // tratamos errores que lanzan enteros
}
...
}
```

Si queremos que un catch trate más de un tipo de errores, podemos poner tres puntos (parámetros indefinidos):

```
h() {
   ...
   catch (char *ce) {
      ... // tratamos errores que lanzan cadenas
   }
   catch (...) {
      ... // tratamos el resto de errores
   }
   ...
}
```

17.4. El bloque de prueba: try

El tratamiento de errores visto hasta ahora es muy limitado, ya que no tenemos forma de especificar dónde se pueden producir errores (en qué bloques del programa). La forma de especificar dónde se pueden producir errores que queremos recoger es emplear bloques **try**, que son bloques delimitados poniendo la palabra try y luego poniendo entre llaves el código que queremos probar. Después del bloque try se ponen los bloques catch para tratar los errores que se hayan podido producir:

```
h() {
   ...
   g();          // si produce un error, se le pasa al que llamo a h()
   try {
      g();       // si produce un error lo tratamos nosotros
   }
   catch (int i){
   ...
      }
   catch (...){
      ...
   }
   z();
}
```

Solo podemos recoger errores después de un bloque try, por lo que los catch siempre van asociados a los try. Si una función que no está dentro de un bloque de prueba recibe un error la pasa a su nivel superior hasta que llegue a una llamada producida dentro de un bloque de prueba que trate el error o salga del programa principal. Si en un bloque try se produce un error que no es tratado por sus catch, también pasamos el error hacia arriba. Cuando se recoge un error con un catch no se retorna al sitio que lo originó, sino que se sigue con el código que hay después del último catch asociado al try donde se aceptó el error. En el ejemplo se ejecutaría la función z().

17.5. La lista throw

Podemos especificar los tipos de excepciones que puede lanzar una función, poniendo después del prototipo de la función la lista throw, que no es más que la palabra throw seguida de una lista de tipos separada por comas y entre paréntesis:

```
void f () throw (char*, int); // f solo lanza cadenas y enteros
```

Si una función lanza una excepción que no esté en su lista de tipos se produce un error de ejecución. Si ponemos una lista vacía la función no puede lanzar excepciones.