Curso de programación en C++

EUI (UPV)

Valencia, 17 al 28 de Julio de 1995

Apuntes de clase

Sergio Talens Oliag

Contenidos

Bibliografía básica

— Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, 1988

— Bjarne Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, 1991

— Enrique Hernández Orallo, José Hernández Orallo, *Programación en C++*, Paraninfo, 1993

Bases teóricas. Introducción a la POO.

INTRODUCCIÓN

Para comenzar a estudiar cualquier lenguaje de programación se debe conocer cuales son los conceptos que soporta, es decir, el tipo de programación que vamos a poder realizar con él. Como el C++ incorpora características nuevas respecto a lenguajes como Pascal o C, en primer lugar daremos una descripción a los conceptos a los que este lenguaje da soporte, repasando los paradigmas de programación y centrándonos en la evolución desde la programación Funcional a la programación Orientada a Objetos. Más adelante estudiaremos el lenguaje de la misma manera, primero veremos sus características funcionales (realmente la parte que el lenguaje hereda de C) y después estudiaremos las extensiones que dan soporte a la programación orientada a objetos (el ++ de C++).

PARADIGMAS DE PROGRAMACIÓN

Según los conceptos en que se basa un lenguaje de programación tenemos distintas maneras de aproximarnos a la resolución de los problemas y diferentes estilos de programación. Podemos clasificar los lenguajes de programación en varios tipos:

— Imperativos

— Orientados a Objetos

— Funcionales

— Lógicos

Las dos primeras opciones se basan en la abstracción de los tipos de datos. Básicamente se trata de representar las características variables de los objetos mediante tipos que el ordenador pueda tratar, como por ejemplo números enteros o caracteres alfanuméricos. Nuestro programa será una colección de algoritmos que opere sobre los datos que hemos modelado. La diferencia entre las dos aproximaciones se verá en puntos posteriores.

Los lenguajes funcionales, al contrario que los imperativos, eliminan totalmente la idea de tipo de datos, se limitan a tratar todos los datos como símbolos y hacen hincapié en las operaciones que podemos aplicar sobre estos símbolos, agrupados en listas o árboles. Es importante indicar que en estos lenguajes se emplea únicamente el concepto de función aplicado a símbolos, siendo una de sus características principales el empleo de las funciones recursivas. Como ejemplo de este tipo de lenguajes podríamos citar el *LISP*.

Los lenguajes lógicos son los que trabajan directamente con la lógica formal, se trata de representar relaciones entre conjuntos, para luego poder determinar si se verifican determinados predicados. El lenguaje lógico más extendido es el *Prolog*.

PROGRAMACIÓN IMPERATIVA

Como ya hemos mencionado anteriormente, la programación imperativa trata con tipos de datos y algoritmos, los primeros representan la información utilizada por los programas, mientras que los segundos se refieren a la manera en que tratamos esa información.

En los puntos que siguen revisaremos de forma breve los conceptos fundamentales de la programación imperativa clásica, también llamada *programación procedural.* La idea básica de esta aproximación es la de definir los algoritmos o procedimientos más eficaces para tratar los datos de nuestro problema.

Tipos de datos

Cuando nos planteamos la resolución de problemas mediante computador lo más usual es que queramos tratar con datos que son variables y cuantificables, es decir, que toman un conjunto de valores distintos entre un conjunto de valores posibles, además de poder almacenar los valores de estos datos en alguna forma aceptable para el computador (ya sea en la memoria o en periféricos de almacenamiento externo).

En un lenguaje de programación el concepto de *tipo de datos* se refiere al conjunto de valores que puede tomar una variable. Esta idea es similar a la que se emplea en matemáticas, donde clasificamos las variables en función de determinadas características, distinguiendo entre números enteros, reales o complejos. Sin embargo, en matemáticas, nosotros somos capaces de diferenciar el tipo de las variables en función del contexto, pero para los compiladores esto resulta mucho más difícil. Por este motivo debemos declarar explícitamente cada variable como perteneciente a un tipo. Este mecanismo es útil para que el computador almacene la variable de la forma más adecuada, además de permitir verificar que tipo de operaciones se pueden realizar con ella.

Se suelen diferenciar los tipos de datos en varias categorías:

— *Tipos elementales*, que son aquellos cuyos valores son atómicos y, por tanto, no pueden ser descompuestos en valores más simples. Entre las variables de estos tipos siempre encontramos definidas una serie de operaciones básicas: asignación de un valor, copia de valores entre variables y operaciones relacionales de igualdad o de orden (por lo tanto, un tipo debe ser un conjunto *ordenado*).

Los tipos más característicos son:

booleanos = {verdadero, falso}

enteros = {… -2, -1, 0, +1, +2, …}

reales = {… -1.0, …, 0.0, …, +1.0, …}

caracteres = {… 'a', 'b', …, 'Z', …}

Generalmente existen mecanismos para que el usuario defina nuevos tipos elementales por *enumeración*, es decir, definiendo el conjunto de valores explícitamente. Por ejemplo podríamos definir el tipo *día* como {lunes, martes, miércoles, jueves, viernes, sábado, domingo}, las variables definidas como *día* sólo podrían tomar estos valores.

Por último mencionaremos otro tipo de datos elemental de características especiales, el *puntero*, que es el tipo que almacena las direcciones de las variables (la dirección de memoria en la que se almacena su valor). Analizaremos este tipo más adelante.

— *Tipos compuestos o estructurados*, que son los tipos formados a partir de los elementales. Existen varias formas de agrupar los datos de tipos elementales:

La más simple es la estructura *indexada*, muy similar a los vectores o matrices de matemáticas, en donde lo que hacemos es relacionar unos índices (pertenecientes a un tipo de datos) con los valores de un tipo determinado. Sobre estas estructuras se pueden realizar las operaciones de consulta o asignación de un valor (a través de su índice).

Otra estructura compuesta muy importante es el *registro*, que no es más que una sucesión de elementos de distintos tipos, denominados *campos*, que llevan asociados un identificador. Sobre estos tipos se definen las operaciones de asignación y de acceso a un campo. Algunos lenguajes también soportan la operación de asignación para toda la estructura (la copia de todos los campos en un solo paso).

El tipo cadena de caracteres es un caso especial de tipo de datos, ya que algunos lenguajes lo incorporan como tipo elemental (con un tamaño fijo), mientras que en otros lenguajes se define como un vector de caracteres (de longitud fija o variable) que es una estructura indexada. Como se ve, los campos de un registro pueden ser de otros tipos compuestos, no sólo de tipos elementales.

— *Tipos recursivos*, que son un caso especial de tipos compuestos, introduciendo la posibilidad de definir un tipo en función de sí mismo.

Para terminar nuestro análisis de los tipos de datos, hablaremos de la forma en que los lenguajes de programación relacionan las variables con sus tipos y las equivalencias entre distintos tipos.

Hemos dicho que el empleo de tipos de datos nos sirve para que el computador almacene los datos de la forma más adecuada y para poder verificar las operaciones que hacemos con ellos. Sería absurdo que intentáramos multiplicar un carácter por un real pero, si el compilador no comprueba los tipos de los operandos de la multiplicación, esto sería posible e incluso nos devolvería un valor, ya que un carácter se representa en binario como un número entre 0 y 255. Para evitar que sucedan estas cosas los compiladores incorporan mecanismos de *chequeo de tipos*, verificando antes de cada operación que sus operandos son de los tipos esperados. El chequeo se puede hacer estática o dinámicamente. El chequeo estático se realiza en tiempo de compilación, es decir, antes de que el programa sea ejecutable. Para realizar este chequeo es necesario que las variables y los parámetros tengan tipos fijos, elegidos por el programador. El chequeo dinámico se realiza durante la ejecución del programa, lo que permite que las variables puedan ser de distintos tipos en tiempo de ejecución.

Por último señalaremos que existe la posibilidad de que queramos realizar una operación definida sobre un tipo de datos, por ejemplo reales, aplicada a una variable de otro tipo, por ejemplo entera. Para que podamos realizar esta operación debe de existir algún mecanismo para compatibilizar los tipos, convirtiendo un operando que no es del tipo esperado en éste, por ejemplo transformando un entero en real, añadiendo una parte decimal nula, o transformando un real en entero por redondeo.

Operadores y expresiones

Como ya hemos dicho con los datos de un tipo podemos realizar determinadas operaciones pero, ¿cómo las expresamos en un lenguaje de programación? Para resolver este problema aparecen lo que llamamos *operadores*. Podemos decir que un *operador* es un símbolo o conjunto de símbolos que representa la aplicación de una función sobre unos operandos. Cuando hablamos de los operandos no sólo nos referimos a variables, sino que hablamos de cualquier elemento susceptible de ser evaluado en alguna forma. Por ejemplo, si definimos una *variable entera* podremos aplicarle operadores aritméticos (+, -, *, /), de asignación (=) o relacionales (>, <, …), si definimos una *variable compuesta* podremos aplicarle un operador de campo que determine a cual de sus componentes queremos acceder, si definimos un *tipo de datos* podemos aplicarle un operador que nos diga cual es el tamaño de su representación en memoria, etc.

Los operadores están directamente relacionados con los tipos de datos, puesto que se definen en función del tipo de operandos que aceptan y el tipo del valor que devuelven. En algunos casos es fácil olvidar esto, ya que llamamos igual a operadores que realizan operaciones distintas en función de los valores a los que se apliquen, por ejemplo, la división de enteros no es igual que la de reales, ya que la primera retorna un valor entero y se olvida del resto, mientras que la otra devuelve un real, que tiene decimales.

Un programa completo está compuesto por una serie de *sentencias*, que pueden ser de distintos tipos:

— *declarativas*, que son las que empleamos para definir los tipos de datos, declarar las variables o las funciones, etc., es decir, son aquellas que se emplean para definir de forma explícita los elementos que intervienen en nuestro programa,

— *ejecutables*, que son aquellas que se transforman en código ejecutable, y

— *compuestas*, que son aquellas formadas de la unión de sentencias de los tipos anteriores.

Llamaremos *expresión* a cualquier sentencia del programa que puede ser evaluada y devuelve un valor. Las expresiones más simples son los *literales*, que expresan un valor fijo explícitamente, como por ejemplo un número o una cadena de caracteres. Las expresiones compuestas son aquellas formadas por una secuencia de términos separados por operadores, donde los términos pueden ser literales, variables o llamadas a funciones (ya que devuelven un resultado).

Algoritmos y estructuras de control

Podemos definir un *algoritmo* de manera general como un conjunto de operaciones o reglas bien definidas que, aplicadas a un problema, lo resuelven en un número finito de pasos. Si nos referimos sólo a la informática podemos dar la siguiente definición:

> Un *procedimiento* es una secuencia de instrucciones que pueden realizarse mecánicamente. Un *procedimiento* que siempre termina se llama *algoritmo*.

Al diseñar algoritmos que resuelvan problemas complejos debemos emplear algún método de diseño, la aproximación más sencilla es la del *diseño descendente* (top-down). El método consiste en ir descomponiendo un problema en otros más sencillos (subproblemas) hasta llegar a una secuencia de instrucciones que se pueda expresar en un lenguaje de alto nivel. Lo que haremos será definir una serie de acciones complejas y dividiremos cada una en otras más simples. Para controlar el orden en que se van desarrollando las acciones, utilizaremos las *estructuras de control*, que pueden ser de distintos tipos:

— *condicionales o de selección*, que nos permiten elegir entre varias posibilidades en función de una o varias condiciones,

— *de repetición (bucles)*, que nos permiten repetir una serie de operaciones hasta que se verifique una condición o hayamos dado un número concreto de vueltas, y

— *de salto*, que nos permiten ir a una determinada línea de nuestro algoritmo directamente.

Funciones y procedimientos

En el punto anterior hemos definido los algoritmos como procedimientos que siempre terminan, y procedimiento como una secuencia de instrucciones que pueden realizarse mecánicamente, aquí consideraremos que un *procedimiento* es un algoritmo que recibe unos *parámetros de entrada*, y una *función* un *procedimiento* que, además de recibir unos parámetros, devuelve un valor de un tipo concreto. En lo que sigue emplearé los términos procedimiento y función indistintamente.

Lo más importante de estas abstracciones es saber como se pasan los parámetros, ya que según el mecanismo que se emplee se podrá o no modificar sus valores. Si los parámetros se pasan *por valor*, el procedimiento recibe una copia del valor que tiene la variable parámetro y por lo tanto no puede modificarla, sin embargo, si el parámetro se pasa *por referencia*, el procedimiento recibe una referencia a la variable que se le pasa como parámetro, no el valor que contiene, por lo que cualquier consulta o cambio que se haga al parámetro afectará directamente a la variable.

¿Por qué surgieron los procedimientos y las funciones? Sabemos que un programa según el paradigma clásico es una colección de algoritmos pero, si los escribiéramos todos seguidos, nuestro programa sería ilegible. Los procedimientos son un método para ordenar estos algoritmos de alguna manera, separando las tareas que realiza un programa. El hecho de escribir los algoritmos de manera independiente nos ayuda a aplicar el diseño descendente; podemos expresar cada subproblema como un procedimiento distinto, viendo en el programa cual ha sido el refinamiento realizado. Además algunos procedimientos se podrán reutilizar en problemas distintos.

Por último indicaremos que el concepto de procedimiento introduce un nivel de abstracción importante en la programación ya que, si queremos utilizar un procedimiento ya implementado para resolver un problema, sólo necesitamos saber cuáles son sus parámetros y cuál es el resultado que devuelve. De esta manera podemos mejorar o cambiar un procedimiento sin afectar a nuestro programa, siempre y cuando no cambie sus parámetros, haciendo mucho más fácil la verificación de los programas, ya que cuando sabemos que un procedimiento funciona correctamente no nos debemos volver a preocupar por él.

Constantes y variables

En los puntos anteriores hemos tratado las variables como algo que tiene un tipo y puede ser pasado como parámetro pero no hemos hablado de cómo o dónde se declaran, de cómo se almacenan en memoria o de si son accesibles desde cualquier punto de nuestro programa.

Podemos decir que un programa está compuesto por distintos bloques, uno de los cuales será el principal y que contendrá el procedimiento que será llamado al comenzar la ejecución del programa. Serán bloques el interior de las funciones, el interior de las estructuras de control, etc.

Diremos que el *campo* o *ámbito* de un identificador es el bloque en el que ha sido definido. Si el bloque contiene otros bloques también en estos el identificador será válido. Cuando hablo de identificador me refiero a su sentido más amplio: variables, constantes, funciones, tipos, etc. Fuera del *ámbito* de su definición ningún identificador tiene validez.

Clasificaremos las variables en función de su ámbito de definición en *globales* y *locales*. Dentro de un bloque una variable es local si ha sido definida en el interior del mismo, y es global si se ha definido fuera de el bloque pero podemos acceder a ella.

Como es lógico las variables ocupan memoria pero, como sólo son necesarias en el interior de los bloques donde se definen, durante la ejecución del programa serán creadas al entrar en su ámbito y eliminadas al salir de él. Así, habrá variables que existirán durante todo el programa (si son globales para todos los bloques) y otras que sólo existan en momentos muy concretos. Este mecanismo de creación y destrucción de variables permite que los programas aprovechen al máximo la memoria que les ha sido asignada.

Todo lo dicho anteriormente es válido para las variables declaradas estáticamente, pero existe otro tipo de variables cuya existencia es controlada por el programador, las denominadas *variables dinámicas*. Ya hablamos anteriormente de los punteros y dijimos entonces que eran las variables empleadas para apuntar a otras variables, pero ¿a qué nos referimos con apuntar? Sabemos que las variables se almacenan en memoria, luego habrá alguna dirección de memoria en la que encontremos su valor (que puede ocupar uno o varios bytes). Los punteros no son más que variables cuyo contenido es una dirección de memoria, que puede ser la de la posición del valor de otra variable.

Cuando deseamos crear variables de tipo dinámico el lenguaje de programación nos suele proporcionar alguna función estándar para reclamarle al S.O. espacio de memoria para almacenar datos, pero como no hemos definido variables que denoten a ese espacio, tendremos que trabajar con punteros. Es importante señalar que el espacio reservado de esta forma se considera ocupado durante todo el tiempo que se ejecuta el programa, a menos que el programador lo libere explícitamente, pero los punteros que contienen la dirección de ese espacio si son variables estáticas, luego dejan de existir al salir de un campo. Si salimos de un campo y no hemos liberado la memoria dinámica, no podremos acceder a ella (a menos que alguno de los punteros fuera global al ámbito abandonado), pero estaremos ocupando un espacio que no será utilizable hasta que termine nuestro programa.

Para terminar sólo diré que existen variables constantes (que ocupan memoria). Son aquellas que tienen un tipo y un identificador asociado, lo que puede ser útil para que se hagan chequeos de tipos o para que tengan una dirección de memoria por si algún procedimiento requiere un puntero a la constante.

PROGRAMACIÓN MODULAR

Con la programación procedural se consigue dar una estructura a los programas, pero no diferenciamos realmente entre los distintos aspectos del problema, ya que todos los algoritmos están en un mismo bloque, haciendo que algunas variables y procedimientos sean accesibles desde cualquier punto de nuestro programa.

Para introducir una organización en el tratamiento de los datos apareció el concepto de *módulo*, que es un conjunto de procedimientos y datos interrelacionados. Aparece el denominado *principio de ocultación de información*, los datos contenidos en un módulo no podrán ser tratados directamente, ya que no serán accesibles desde el exterior del mismo, sólo permitiremos que otro módulo se comunique con el nuestro a través de una serie de procedimientos públicos definidos por nosotros. Esto proporciona ventajas como poder modificar la forma de almacenar algunos de los datos sin que el resto del programa sea alterado o poder compilar distintos módulos de manera independiente. Además, un módulo bien definido podrá ser reutilizado y su depuración será más sencilla al tratarlo de manera independiente.

TIPOS ABSTRACTOS DE DATOS

Con los mecanismos de definición de tipos estructurados podíamos crear tipos de datos más complejos que los primitivos, pero no podíamos realizar más que unas cuantas operaciones simples sobre ellos. Sin embargo, los procedimientos nos permiten generalizar el concepto de operador. En lugar de limitarnos a las operaciones incorporadas a un lenguaje, podemos definir nuestros propios operadores y aplicarlos a operandos que no son de un tipo fundamental (por ejemplo, podemos implementar una rutina que multiplique matrices y utilizarla como si fuera un operador sobre variables del tipo matriz). Además, la estructura modular vista en el apartado anterior nos permite reunir en un solo bloque las estructuras que componen nuestro tipo y los procedimientos que operan sobre él. Surgen los denominados Tipos Abstractos de Datos (TAD).

Un TAD es una *encapsulación* de un tipo abstracto de datos, que contiene la definición del tipo y todas las operaciones que se pueden realizar con él (en teoría, algún operando o el resultado de las operaciones debe pertenecer al tipo que estamos definiendo). Esto permite tener localizada toda la información relativa a un tipo de datos, con lo que las modificaciones son mucho más sencillas, ya que desde el resto del programa tratamos nuestro TAD como un tipo elemental, accediendo a él sólo a través de los operadores que hemos definido.

El problema de esta idea está en que los lenguajes que soportan módulos pero no están preparados para trabajar con TAD sólo permiten que definamos los procedimientos de forma independiente, es decir, sin asociarlos al TAD más que por la pertenencia al módulo. Las

variables del tipo únicamente se podrán declarar como estructuras, por lo que los procedimientos necesitarán que las incluyamos como parámetro. La solución adoptada por los nuevos lenguajes es incorporar mecanismos de definición de tipos de usuario que se comportan casi igual que los tipos del lenguaje.

PROGRAMACIÓN ORIENTADA A OBJETOS

La diferencia fundamental entre la programación pocedural y la orientada a objetos está en la forma de tratar los datos y las acciones. En la primera aproximación ambos conceptos son cosas distintas, se definen unas estructuras de datos y luego se define una serie de rutinas que operan sobre ellas. Para cada estructura de datos se necesita un nuevo conjunto de rutinas. En la programación orientada a objetos los datos y las acciones están muy relacionadas. Cuando definimos los datos (*objetos*) también definimos sus acciones. En lugar de un conjunto de rutinas que operan sobre unos datos tenemos objetos que interactuan entre sí.

Objetos y mensajes

Un *Objeto* es una entidad que contiene información y un conjunto de acciones que operan sobre los datos. Para que un objeto realice una de sus acciones se le manda un *mensaje*. Por tanto, la primera ventaja de la programación orientada a objetos es la encapsulación de datos y operaciones, es decir, la posibilidad de definir Tipos Abstractos de Datos.

De cualquier forma la *encapsulación* es una ventaja mínima de la programación orientada a objetos. Una característica mucho más importante es la posibilidad de que los objetos puedan heredar características de otros objetos. Este concepto se incorpora gracias a la idea de*clase*.

Clases

Cada objeto pertenece a una *clase*, que define la implementación de un tipo concreto de objetos. Una clase describe la información de un objeto y los mensajes a los que responde. La declaración de una clase es muy parecida a la definición de un registro, pero aquí los campos se llaman *instancias de variables* o *datos miembro* (aunque utilizaré el término *atributo*, que no suena tan mal en castellano). Cuando le mandamos un mensaje a un objeto, este invoca una rutina que implementa las acciones relacionadas con el mensaje. Estas rutinas se denominan *métodos* o *funciones miembro*. La definición de la clase también incluye las implementaciones de los métodos.

Se puede pensar en las clases como plantillas para crear objetos. Se dice que un objeto es una *instancia* de una clase. También se puede decir que un objeto es *miembro* de una clase.

Herencia y polimorfismo

Podemos definir clases en base a otra clase ya existente. La nueva clase se dice que es una *subclase* o *clase derivada*, mientras que la que ya existía se denomina *superclase* o *clase base*. Una clase que no tiene superclase se denomina *clase raíz*.

Una subclase *hereda* todos los métodos y atributos de su superclase, además de poder definir miembros adicionales (ya sean datos o funciones). Las subclases también pueden redefinir (override) métodos definidos por su superclase. Redefinir se refiere a que las subclase responde al mismo mensaje que su superclase, pero utiliza su propio método para hacerlo.

Gracias al mecanismo de redefinición podremos mandar el mismo mensaje a objetos de diferentes clases, esta capacidad se denomina *polimorfismo*.

Programación con objetos

A la hora de programar con objetos nos enfrentamos a una serie de problemas: ¿Qué clases debo crear?, ¿Cuándo debo crear una subclase? ¿Qué debe ser un método o un atributo?

Lo usual es crear una jerarquía de clases, definiendo una clase raíz que da a todos los objetos un comportamiento común. La clase raíz será una clase *abstracta*, ya que no crearemos instancias de la misma. En general se debe definir una clase (o subclase de la clase raíz) para cada concepto tratado en la aplicación. Cuando necesitemos añadir atributos o métodos a una clase definimos una subclase. Los atributos deben ser siempre privados, y deberemos proporcionar métodos para acceder a ellos desde el exterior. Todo lo que se pueda hacer con un objeto debe ser un método.

El lenguaje C++

INTRODUCCIÓN

En este bloque introduciremos el lenguaje de programación C++. Comenzaremos por dar una visión general del lenguaje y después trataremos de forma práctica todos los conceptos estudiados en el bloque anterior, viendo como se implementan en el C++.

Trataremos de que el tema sea fundamentalmente práctico, por lo que estos apuntes se deben considerar como una pequeña introducción al lenguaje, no como una referencia completa. Los alumnos interesados en conocer los detalles del lenguaje pueden consultar la bibliografía.

CONCEPTOS BÁSICOS

Comenzaremos estudiando el soporte del C++ a la programación imperativa, es decir, la forma de definir y utilizar los tipos de datos, las variables, las operaciones aritméticas, las estructuras de control y las funciones. Es interesante remarcar que toda esta parte está heredada del C, por lo que también sirve de introducción a este lenguaje.

Estructura de los programas

El mínimo programa de C++ es:

```
main() { }
```

Lo único que hemos hecho es definir una función (`main`) que no tiene argumentos y no hace nada. Las llaves { } delimitan un bloque en C++, en este caso el cuerpo de la función `main`. 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.

Los comentarios se introducirán en el programa separados por `/*` y `*/` o comenzándolos con `//`. Los comentarios entre `/*` y `*/` pueden tener la longitud que queramos, pero no se anidan, es decir, si escribimos `/* hola /* amigo */ mío */`, el compilador interpretará que el comentario termina antes de mío, y dará un error. Los comentarios que comienzan por `//` sólo son válidos hasta el final de la línea en la que aparecen.

Un programa simple que muestra todo lo que hemos visto puede ser el siguiente:

```
/*
  Este es un programa mínimo en C++, lo único que hace es escribir una frase
  en la pantalla
*/

#include <iostream.h>

int main()
{
  cout << "Hola guapo\n";  // imprime en la pantalla la frase "hola guapo"
}
```

La primera parte separada entre `/*` y `*/` es un comentario. Es recomendable que se comenten los programas, explicando que es lo que estamos haciendo en cada caso, para que cuando se lean sean más comprensibles.

La línea que empieza por `#` es una directiva. En este caso indica que se incluya el fichero "iostream.h", que contiene las definiciones para entrada/salida de datos en C++.

En la declaración de `main()` hemos incluido la palabra `int`, que indica que la función devuelve un entero. Este valor se le entrega al sistema operativo al terminar el programa. Si no se devuelve ningún valor el sistema recibe un valor aleatorio.

La sentencia separada ente llaves indica que se escriba la frase "Hola guapo". El operador `<<` ("poner en") escribe el segundo argumento en el primero. En este caso la cadena `"Hola guapo\n"` se escribe en la salida estándar (`cout`). El carácter `\` seguido de otro carácter indica un solo carácter especial, en este caso el salto de línea (`\n`).

Veremos el tema de la entrada salida estándar más adelante. Hay que indicar que las operaciones de E/S se gestionan de forma diferente en C y C++, mientras que el C proporciona una serie de funciones (declaradas en el fichero "stdio.h"), el C++ utiliza el concepto de *stream*, que se refiere al flujo de la información (tenemos un flujo de entrada que proviene de `cin` y uno de salida que se dirige a `cout`) que se maneja mediante operadores de E/S.

Por último señalar que debemos seguir ciertas reglas al nombrar tipos de datos, variables, funciones, etc. Los identificadores válidos del C++ son los formados a partir de los caracteres del alfabeto (el inglés, no podemos usar ni la ñ ni palabras acentuadas), los dígitos (0..9) y el subrayado (_), la única restricción es que no podemos comenzar un identificador con un dígito (es así porque se podrían confundir con literales numéricos). Hay que señalar que el C++ distingue entre mayúsculas y minúsculas, por lo que `Hola` y `hola` representan dos cosas diferentes. Hay que evitar el uso de identificadores que sólo difieran en letras mayúsculas y minúsculas, porque inducen a error.

Tipos de datos y operadores

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;
```

Sobre los tipos elementales se pueden emplear los siguientes operadores aritméticos:

```
+     (más, como signo o como operación suma)
-     (menos, como signo o como operación resta)
*     (multiplicación)
/     (división)
%     (resto)
```

Y los siguientes operadores relacionales:

```
==    (igual)
!=    (distinto)
<     (menor que)
>     (mayor que)
<=    (menor o igual que)
>=    (mayor o igual que)
```

El operador de asignación se representa por `=`.

En la bibliografía del C++ se suelen considerar como tipos *derivados* los construidos mediante la aplicación de un operador a un tipo elemental o compuesto en su declaración. Estos operadores son:

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

Los tipos compuestos son las estructuras (`struct`), las uniones (`unión`) y las clases (`class`).

Estructuras de control

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)

Veremos como se manejan todas ellas en el punto dedicado a las estructuras de control

Funciones

Una función es una parte con nombre de un programa que puede ser invocada o llamada desde cualquier otra parte del programa cuando haga falta. La sintaxis de las funciones depende de si las declaramos o las definimos.

La declaración se escribe poniendo el tipo que retorna la función seguido de su nombre y de una lista de parámetros entre paréntesis (los parámetros deben ser de la forma `tipo-param [nom_param]`, donde los corchetes indican que el nombre es opcional), para terminar la declaración ponemos punto y coma (recordar que una declaración es una sentencia).

Para definir una función se escribe el tipo que retorna, el nombre de la función y una lista de parámetros entre paréntesis (igual que antes, pero aquí si que es necesario que los parámetros tengan nombre). A continuación se abre una llave, se escriben las sentencias que se ejecutan en la función y se cierra la llave.

Un ejemplo de declaración de función sería:

```
int eleva_a_n (int, int);
```

Y su definición sería:

```
int eleva_a_n (int x, int n)
{
  if (n<0) error ("exponente negativo");

  switch (n) {
  case 0:  return 1;
  case 1:  return x;
  default: return eleva_a_n (x, n-1);
  }
}
```

Por defecto los parámetros se pasan por valor, para pasarlos por referencia usaremos punteros y referencias. Veremos más ampliamente las funciones en el punto dedicado a ellas.

Soporte a la programación modular

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 modulo (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.

Si queremos utilizar en un módulo las declaraciones de una cabecera incluimos el archivo cabecera mediante la directiva `#include`. También en el fichero que empleamos para definir las funciones de una cabecera se debe incluir el .h que define.

Existe además la posibilidad de definir una variable o función externa a un módulo mediante el empleo de la palabra `extern` delante de su declaración.

Por último indicar que cuando incluimos una cabecera estándar (como por ejemplo "iostream.h") el nombre del fichero se escribe entre menor y mayor (`#include <iostream.h>`), pero cuando incluimos una cabecera definida por nosotros se escribe entre comillas (`#include "mi_cabecera.h"`). Esto le sirve al compilador para saber donde debe buscar los ficheros.

Soporte a los Tipos de Datos Abstractos

Para soportar los tipos de datos se proporcionan mecanismos para definir operadores y funciones sobre un tipo definido por nosotros y para restringir el acceso a las operaciones a los objetos de este tipo. Además se proporcionan mecanismos para redefinir operadores e incluso se soporta el concepto de tipos genéricos mediante las *templates* (plantillas).

También se define un mecanismo para manejo de excepciones que permite que controlemos de forma explícita los errores de nuestro programa.

Soporte a la programación Orientada a Objetos

Para soportar el concepto de programación orientada a objetos se incluyen diversos mecanismos:

— declaración de clases (usando la palabra class)

— de paso de mensajes a los objetos (realmente son llamadas a funciones)

— protección de los métodos y atributos de las clases (definición de métodos privados, protegidos y públicos)

— soporte a la herencia y polimorfismo (incluso se permite la herencia múltiple, es decir, la definición de una clase que herede características de dos clases distintas)

Es interesante señalar que el soporte a objetos complementa a los mecanismos de programación modular, encapsulando aun más los programas.

TIPOS DE DATOS, OPERADORES Y EXPRESIONES

Tipos de datos

Para declarar una variable ponemos el nombre del tipo seguido del de la variable. 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.

Tipos elementales

El C++ tiene un conjunto de tipos elementales correspondientes a las unidades de almacenamiento típicas de un computador y a las distintas maneras de utilizarlos:

— enteros:

```
char
short int
int
long int
```

— reales (números en coma flotante):

```
float
double
long double
```

La diferencia entre los distintos tipos enteros (o entre los tipos reales) está en la memoria que ocupan las variables de ese tipo y en los rangos que pueden representar. A mayor tamaño, mayor cantidad de valores podemos representar. Con el operador `sizeof` podemos saber cuanto ocupa cada tipo en memoria.

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).

Para tener una notación más compacta la palabra `int` se puede eliminar de un nombre de tipo de más de una palabra, por ejemplo `short int` se puede escribir como `short`, `unsigned` es equivalente a `unsigned int`, etc.

El tipo entero `char` es el que se utiliza normalmente para almacenar y manipular caracteres en la mayoría de los computadores, generalmente ocupa 8 bits (1byte), y es el tipo que se utiliza como base para medir el tamaño de los demás tipos del C++.

Un tipo especial del C++ es el denominado `void` (vacío). Este tipo tiene características muy peculiares, ya que es sintácticamente igual a los tipos elementales pero sólo se emplea junto a los derivados, es decir, 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 (esto lo veremos al estudiar los punteros). Por ejemplo:

```
void BorraPantalla (void);
```

indica que la función BorraPantalla no tiene parámetros y no retorna nada.

Tipos enumerados

Un tipo especial de tipos enteros son los tipos enumerados. Estos tipos sirven para definir un tipo que sólo puede tomar valores dentro de un conjunto limitado de valores. Estos valores tienen nombre, luego lo que hacemos es dar una lista de constantes asociadas a un tipo.

La sintaxis es:

```
enum booleano {FALSE, TRUE};  // definimos el tipo booleano
```

Aquí hemos definido el tipo booleano que puede tomar los valores FALSE o TRUE. En realidad hemos asociado la constante FALSE con el número 0, la constante TRUE con 1, y si hubiera más constantes seguiríamos con 2, 3, etc. Si por alguna razón nos interesa dar un número concreto a cada valor podemos hacerlo en la declaración:

```
enum colores {rojo = 4, azul, verde = 3, negro = 1};
```

azul tomará el valor 5 (4+1), ya que no hemos puesto nada. También se pueden usar números negativos o constantes ya definidas.

Para declarar un variable de un tipo enumerado hacemos:

```
enum booleano test;  // sintaxis de ANSI C
booleano test;       // sintaxis de C++
```

En ANSI C los enumerados son compatibles con los enteros, en C++ hay que convertir los enteros a enumerado:

```
booleano test = (booleano) 1;  // asigna TRUE a test (valor 1)
```

Si al definir un tipo enumerado no le damos nombre al tipo declaramos una serie de constantes:

```
enum { CERO, UNO, DOS };
```

Hemos definido las constantes CERO, UNO y DOS con los valores 0, 1 y 2.

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

El problema más grave con los tipos derivados es la forma de declararlos: los punteros y las referencias utilizan notación prefija y los vectores y funciones usan notación postfija. La idea es que las declaraciones se asemejan al uso de los operadores de derivación. Para los ejemplos anteriores haríamos lo siguiente:

```
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
             // (recordad que los vectores empiezan en 0)
i = *v[2]    // almacenamos en i el valor al que apunta el tercer puntero de v
f(i)         // llamamos a la función f con el parámetro i
```

Hay que indicar que otra fuente de errores es la asociación de operadores, ya que a veces es necesario el uso de paréntesis. Por ejemplo, en la declaración de c anterior hemos declarado un vector de punteros, pero para declarar un puntero a un vector necesitamos paréntesis:

```
int *c[20];     // vector de 20 punteros a entero
int (*p)[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
```

Este tipo de declaraciones deben evitarse para hacer más legibles los programas.

A continuación veremos los tipos derivados con más detalle, exceptuando las funciones, a las que dedicaremos un punto completo del bloque más adelante.

Punteros

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

```
int  *pi           // Puntero a entero
char **pc          // Puntero a puntero a carácter
int  (*pv)[10]     // Puntero a vector de 10 enteros
int  (*pf)(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')
```

Veremos más ampliamente los punteros al hablar de variables dinámicas.

Vectores

Para un tipo `T`, `T[n]` indica un tipo vector con `n` elementos. Los índices del vector empiezan en `0`, luego 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] = {2, 3, 4, 7, 8};
char v2[2][3] = {{'a', 'b', 'c'}, {'A', 'B', 'C'} };  // vect. multidimensional
int v3[2] = {1, 2, 3, 4};             // error: sólo tenemos espacio para 2 enteros
char c1[5] = {'h','o','l','a','\0'};  // cadena "hola"
char c2[5] = "hola";                  // lo mismo
char c3[] = "hola";                   // el compilador le da tamaño 5 al vector
char vs[3][] = {"uno", "dos", "tres"} // 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. `T&` significa referencia a tipo `T`.

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
```

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 nomtipo` (ANSI C, C++) o `nomtipo` (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 juan= {21, "Juan Pérez"};
```

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 dereferenciar 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 = "Pepe";  // el nombre de la estructura apuntada por p vale "Pepe"
  (*p).nombre = "Pepe";  // igual que antes
```

Empleando el operador `sizeof` a la estructura podemos saber cuantos 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 que 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)
```

Por último diremos que 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;          // Podemos usar la variable 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 en un punto del bloque.

Constantes (literales)

Hay cuatro tipos de literales en C++:

Literales enteros

Octales (en base ocho), si empiezan por cero, p. ej. 023 equivale a 10011 en binario o a 19 en decimal

Hexadecimales (en base dieciséis), si empiezan por 0x, p.ej. 0x2F que equivale a 101111 en binario o a 47 en decimal. En hexadecimal los valores del 10 al 15 se representan por A, B, C, D, E, y F (en mayúsculas o minúsculas).

Decimales, que son los que no empiezan por 0.

A cada uno de estos literales les podemos añadir un sufijo para indicar que son sin signo (sufijo `u` o `U`) o para forzar a que sean de tipo `long` (sufijo `l` o `L`), por ejemplo `23L` es el entero 23 de tipo `long`, `0xFu` es el entero 15 sin signo. También podemos mezclar los sufijos: `12Lu` entero 12 sin signo de tipo `long`.

Literales reales

Una constante en coma flotante se escribe con la parte entera y la decimal separadas por punto, y opcionalmente se puede escribir la letra e o E y un exponente.

El tipo de constante depende del sufijo:

```
Sin sufijo: double
```

```
f, F : float
l, L : long double
```

Ejemplos:

```
1.8e3        // 1.8 * 10^3 == 1800 de tipo double
0.1L         // valor 0.1 long double
-1e-3f       // -0.001 float
4.           // 4.0 double
.12          // 0.12 double
```

Literales carácter

Los caracteres se delimitan entre dos apóstrofes `''`. Dentro de los apóstrofes sólo podemos escribir un carácter, excepto cuando son caracteres especiales, que se codifican mediante el carácter de escape `\` seguido de otro carácter .

Ejemplos:

```
'a'    // carácter a
' '    // espacio en blanco (es un carácter)
'\\'   // carácter \ (es un carácter especial)
''     // error: debe haber un carácter entre los apóstrofes
'ab'   // error: pero sólo uno
```

Caracteres especiales más utilizados:

```
\n    // salto de línea
\t    // tabulador horizontal
\v    // tabulador vertical
\b    // retroceso
\r    // retorno de carro
\f    // salto de página
\a    // alerta (campana)
\\    // carácter \
\?    // interrogante
\'    // comilla simple
\"    // comilla doble
\ooo  // carácter ASCII dado por tres dígitos octales (ooo serán dígitos)
\xhh  // carácter ASCII dado por dos dígitos hexadecimales (hh serán dígitos)
\0    // carácter nulo
```

Literales cadena

Una cadena es un a secuencia de caracteres escrita entre comillas dobles y terminada con el carácter nulo (`\0`).

Ejemplos:

```
"hola"   // equivale a 'h','o','l','a','\0'
""       // equivale a '\0'
"'"      // equivale a '\'', '\0'
"\'"     // equivale a '\'', '\0'
"""      // error, para escribir " dentro de una cadena lo correcto sería "\""
"ho""la" // se concatenan, equivale a 'h','o','l','a','\0'
```

Variables

Alcance o ámbito de las variables

Las variables existen sólo 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 (no se crea al entrar en el módulo ni se destruye al salir, sino que permanece todo el tiempo) 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).

El operador ::

Dentro de un bloque podemos emplear el operador :: para acceder a variables declaradas en un bloque superior. Este operador sólo es útil cuando en un bloque interno tenemos una variable con un nombre igual a otro externo (la variable accesible será la interna, pero con :: accederemos a la externa). Veremos que el operador :: se usa fundamentalmente para clases.

Ejemplo:

```
main () {
  int v;
  …
    {
      char v;
      v = 5;  // asigna 5 a la variable char (interna)
      ::v=9 + v;  // asigna 9 + 5 (valor de v interna) a la variable int más externa
    }
}
```

Variables volátiles

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 sólo modifica nuestro programa (por ejemplo una variable que utiliza el Hardware o el SO).

Variables register

Podemos intentar hacer más eficientes nuestros programas indicándole al compilador que variables usamos más a menudo para que las coloque en los registros. Esto se hace declarando la variable precedida de la palabra `register`. No tenemos ninguna garantía de que el compilador nos haga caso, depende del entorno de desarrollo que empleemos.

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. De cualquier forma debemos ser cuidadosos a la hora de mezclar tipos, ya que la conversión que realiza el compilador puede no ser la que esperamos.

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++). Hay que tener en cuenta que hay conversiones que no tienen sentido, como pasar un long a short si el valor que contiene no cabe, o convertir un puntero a entero, por ejemplo.

Más adelante veremos que para los tipos definidos por nosotros podemos crear conversores a otros tipos.

Operadores y expresiones

El C++ tiene gran cantidad de operadores de todo tipo, en este punto veremos de forma resumida todos ellos, para luego dar una tabla con su precedencia y su orden de asociatividad. Antes de pasar a listar los operadores por tipos daremos unas definiciones útiles para la descripción de la sintaxis de los operadores:

Operandos LValue (left value)

Son aquellos operandos que cumplen las siguientes características:

— Pueden ser fuente de una operación.

— Pueden ser destino de una operación.

— Se puede tomar su dirección.

En definitiva un LValue es cualquier operando que este almacenado en memoria y se pueda modificar.

Expresiones

Como ya dijimos en el bloque anterior, una expresión es cualquier sentencia del programa que puede ser evaluada y devuelve un valor.

Lista de operadores según su tipo

Operadores aritméticos:

```
+           Suma
-           Resta
*           Producto
/           División: entera para escalares, real para reales
%           Módulo: retorna el resto de una división entre enteros
-(unario)   Cambio de signo
+(unario)   No hace nada
```

Operadores de incremento y decremento:

```
++          incremento en uno del operando LValue al que se aplica.
--          decremento en uno del operando LValue al que se aplica.
```

Estos operadores pueden ser prefijos o postfijos. Si son prefijos el operando se incrementa (decrementa) antes de ser evaluado en una expresión, si son postfijos el operando se incrementa (decrementa) después de la evaluación.

Operadores relacionales:

```
>           Mayor
<           Menor
>=          Mayor o igual
<=          Menor o igual
==          Igual
!=          Distinto
```

Como el C++ no tiene definido el tipo booleano el resultado de una comparación es 0 si no tiene éxito (FALSE) o distinto de cero (TRUE, generalmente 1) si si que lo tiene.

Operadores lógicos:

```
&&          AND lógico
||          OR lógico
!           NOT lógico
```

Toman expresiones como operandos, retornando verdadero o falso como en los operadores relacionales

Operadores de bit:

```
&           Producto binario de bits (AND).
|           Suma binaria de bits (OR).
^           Suma binaria exclusiva de bits (XOR).
<<          Desplazamiento hacia la izquierda del primer operando tantos bits como
            indique el segundo operando.
<<          Desplazamiento hacia la derecha del primer operando tantos bits como
            indique el segundo operando.
~ (unario)  Complemento a uno (cambia unos por ceros).
- (unario)  Complemento a dos (cambio de signo).
```

Estos operadores tratan los operandos como palabras de tantos bits como tenga su tipo, su tipo sólo puede ser entero (char, short, int, long).

Operadores de asignación:

```
=           Asignación de la expresión de la derecha al operando de la izquierda.
```

Si delante del operador de asignación colocamos cualquiera de los siguientes operadores asignaremos al lado izquierdo el resultado de aplicar el operador al lado izquierdo como primer operando y el lado derecho como segundo operando:

```
+     -     *     /     %     <<    >>    &     ^     |
```

Ejemplos:

```
int i = 25;
int v[20];

i /= 2;            // equivale a poner i = i / 2
v[i+3] += i * 8;   // equivale a v[i+3] = v[i+3] + (i * 8)
```

Operador de evaluación:

Sirve para evaluar distintas expresiones seguidas, pero sólo se queda con el resultado de la última.

Ejemplo:

```
int i = (1, 2, 3);               // i toma el valor 3
int v = (Randomize (), Rand());  // El Randomize se ejecuta, pero v toma el valor
                                 // retornado por Rand()
```

Operador de campo:

```
::          Visto en el punto de las variables, también se usa en las clases
```

Operadores de indirección:

```
*           Indirección
&           Dirección (referencia)
->          Puntero selector de miembro o campo
.           Selector de miembro o campo
[]          Subíndice
->*         Puntero selector de puntero a miembro
.*          Selector de puntero a miembro
```

Los usos de estos operadores que aún no hemos visto se estudiarán más adelante

Operador condicional:

```
?:          Expresión condicional
```

Este operador es equivalente a una expresión condicional, lo que hacemos es evaluar el primer operando y si es cierto retornamos el segundo operando, en caso contrario retornamos el tercer operando.

Ejemplo:

```
max = (a>b) ? a : b;  // si a es mayor que b max vale a, sino max vale b
```

Operadores sizeof:

```
sizeof var       Nos da el tamaño de una variable
sizeof (tipo)    Nos da el tamaño de un tipo de datos
```

Estos operadores nos devuelven el tamaño en bytes.

Precedencia, asociatividad y orden de evaluación

La precedencia de los operadores nos permite evitar el uso de paréntesis (que es lo que más precedencia tiene) en expresiones sencillas. Veremos la precedencia de los operadores en función del orden de la tabla, los primeros bloques serán los de mayor precedencia.

También indicaremos en la tabla el orden en que se asocian los operandos (si se pueden asociar).

Operador	Sintaxis	Descripción	Asociatividad
::	nom_clase::miembro	resolución de campos	NO
::	::nombre	operador de campo	NO
.	estructura.miembro	selector de miembro	Izquierda a derecha
->	puntero->miembro	puntero a miembro	Izquierda a derecha
[]	puntero[expr]	subíndice	Izquierda a derecha
()	nom_funcion()	llamada a función	Izquierda a derecha
()	tipo (expr)	conversión de tipos	NO
sizeof	sizeof expr	tamaño de un objeto	NO
sizeof	sizeof (tipo)	tamaño de un tipo	NO
++	lvalue ++	postincremento	NO
++	++ lvalue	preincremento	NO
--	lvalue --	postdecremento	NO
--	-- lvalue	predecremento	NO
~	~ expr	complemento	NO
!	! expr	negación	NO
-	- expr	negativo	NO
+	+ expr	positivo	NO
&	& lvalue	dirección	NO
*	* expr	indirección	NO
new	new tipo	crear objeto	NO
delete	delete puntero	borrar objeto	NO
delete[]	delete [] puntero	borrar array	NO
()	(tipo) expr	conversión de tipo	Derecha a izquierda
.*	objeto.ptr_miembro	selección miembro	Izquierda a derecha
->*	ptr->ptr_miembro	puntero miembro	Izquierda a derecha
*	expr * expr	multiplicación	Izquierda a derecha
/	expr / expr	división	Izquierda a derecha
%	expr % expr	resto	Izquierda a derecha
+	expr + expr	suma	Izquierda a derecha
-	expr - expr	resta	Izquierda a derecha
<<	expr << expr	desplazamiento izquierda	Izquierda a derecha
>>	expr >> expr	desplazamiento derecha	Izquierda a derecha
<	expr < expr	menor que	Izquierda a derecha
<=	expr <= expr	menor o igual que	Izquierda a derecha
>	expr > expr	mayor que	Izquierda a derecha
>=	expr >= expr	mayor o igual que	Izquierda a derecha

== !=	expr == expr expr != expr	igual que distinto que	Izquierda a derecha Izquierda a derecha
&	expr & expr	AND a nivel de bits	Izquierda a derecha
^	expr ^ expr	XOR a nivel de bits	Izquierda a derecha
|	expr | expr	OR a nivel de bits	Izquierda a derecha
&&	expr && expr	AND lógico	Izquierda a derecha
||	expr || expr	OR lógico	Izquierda a derecha
?:	expr ? expr : expr	expresión condicional	Derecha a izquierda
= *= /= %= += -= <<= >>= &= |= ^=	lvalue = expr lvalue *= expr lvalue /= expr lvalue %= expr lvalue += expr lvalue -= expr lvalue <<= expr lvalue >>= expr lvalue &= expr lvalue |= expr lvalue ^= expr	asignación simple producto y asignación división y asignación resto y asignación suma y asignación resta y asignación despl. izq. y asignación despl. der. y asignación AND y asignación OR y asignación XOR y asignación	Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda Derecha a izquierda
,	expr, expr	coma (secuencia)	Derecha a izquierda

Para terminar con los operadores indicaremos que el orden de evaluación de los operandos no está predefinido, excepto en los casos siguientes:

```
a || b     Si se cumple a no se evalúa b
a && b     Si no se cumple a no se evalúa b
,          La coma evalúa de izquierda a derecha
a ? b : c  Si se cumple a se evalúa b, sino se evalúa c
```

ESTRUCTURAS DE CONTROL

El C++, como todo lenguaje de programación basado en la algorítmica, posee una serie de estructuras de control para gobernar el flujo de los programas. Aquí estudiaremos las estructuras de control de la misma manera que las vimos en el bloque anterior, es decir, separando entre estructuras condicionales o de selección, de repetición y de salto.

Antes de comenzar debemos recordar que la evaluación de una condición producirá como resultado un cero si es falsa y un número cualquiera distinto de cero si es cierta, este hecho es importante a la hora de leer los programas, ya que una operación matemática, por ejemplo, es una condición válida en una estructura de control.

Estructuras de selección

Dentro de las estructuras de selección encontramos dos modelos en el C++, las de condición simple (sentencias if else) y las de condición múltiple (switch). A continuación estudiaremos ambos tipos de sentencias.

La sentencia if

Se emplea para elegir en función de una condición. Su sintaxis es:

```
if (expresión)
  sentencia 1
else
  sentencia 2
```

Los paréntesis de la expresión a evaluar son obligatorios, la sentencia 1 puede ser una sola instrucción (que no necesita ir entre llaves) o un bloque de instrucciones (entre llaves pero sin punto y coma después de cerrar). El `else` es opcional, cuando aparece determina las acciones a tomar si la expresión es falsa.

El único problema que puede surgir con estas sentencias es el anidamiento de if y else: Cada else se empareja con el if más cercano:

```
if (expr1)
  if (expr2)
    acción 1
  else          // este else corresponde al if de expr 2
    acción 2
else            // este corresponde al if de expr1
  acción 3
```

Para diferenciar bien unas expresiones de otras (el anidamiento), es recomendable tabular correctamente y hacer buen uso de las llaves:

```
if (expr1) {
  if (expr2)
    acción 1
}              // Notar que la llave no lleva punto y coma después, si lo pusiéramos
               // habríamos terminado la sentencia y el else se quedaría suelto
else           // Este else corresponde al if de expr1
  acción 3
```

Por último indicaremos que cuando anidamos else - if se suele escribir:

```
if (e1)
  a1
else if (e2)
  a2
else if (e3)
  a3
…
else
  an
```

de esta manera evitamos el exceso de tabulación.

La sentencia switch

Esta sentencia nos permite seleccionar en función de condiciones múltiples. Su sintaxis es:

```
switch (expresión) {
  case valor_1: sentencia 11;
                sentencia 12;
                …
                sentencia 1n;
                break;
  case valor_2: sentencia 21;
                sentencia 22;
                …
                sentencia 2m;
                break;
  …
  default:      sentencia d1;
                sentencia d2;
                …
                sentencia dp
}
```

El paréntesis en la expresión es obligatorio. El funcionamiento es el siguiente, si al evaluar la expresión se obtiene uno de los valores indicados por `case valor_i` se ejecutan todas las sentencias que encontremos hasta llegar a un `break` (o al cierre de las llaves). Si no se verifica ningún case pasamos a las sentencias `default`, si existe (default es opcional) y si no existe no hacemos nada.

Indicaremos que si queremos hacer lo mismo para distintos valores podemos escribir los case seguidos sin poner break en ninguno de ellos y se ejecutará lo mismo para todos ellos.

Ejemplo:

```
void main() {
  int i;
```

```
  cin >> i;
  switch (i) {
    case 0:
    case 2:
    case 4:
    case 6:
    case 8:
      cout << " El número " << i << " es par\n";
      break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
      cout << " El número " << i << " es impar\n";
      break;
    default:
      cout << " El número " << i << " no está reconocido\n";
  }
}
```

Estructuras de repetición

Dentro de las estructuras de repetición diferenciábamos 3 tipos: con condición inicial, con condición final y con contador.

La sentencia do-while

Es una estructura de repetición con condición final. Su sintaxis es:

```
do
  sentencia
while (expresión);
```

El funcionamiento es simple, entramos en el `do` y ejecutamos la sentencia, evaluamos la expresión y si es cierta volvemos al `do`, si es falsa salimos.

La sentencia while

Es una estructura de repetición con condición inicial. Su sintaxis es:

```
while (expresión)
  sentencia
```

El funcionamiento es simple evaluamos la expresión y si es cierta ejecutamos la sentencia y volvemos a evaluar, si es falsa salimos.

La sentencia for

Aunque en la mayoría de los lenguajes la sentencia `for` es una estructura de repetición con contador, en C++ es muy similar a un `while` pero con características similares. Su sintaxis es:

```
for (expr1; expr2; expr3)
  sentencia
```

Que es equivalente a:

```
expr1
while (expr2) {
  sentencia
  expr3
}
```

Es decir, la primera expresión se ejecuta una vez antes de entrar en el bucle, después se comprueba la verdad o falsedad de la segunda expresión y si es cierta ejecutamos la sentencia y luego la expresión 3.

En el siguiente ejemplo veremos como esto se puede interpretar fácilmente como una repetición con contador:

```
int i;
…
for (i=0; i<10; i++) {
  cout << " Voy por la vuelta " << i << endl;
}
```

Este fragmento de código inicializa la variable i a 0, comprueba que el valor de i es menor que 10, ejecuta la sentencia e incrementa i. A continuación vuelve a comprobar que i es menor que 10, ejecuta la sentencia e incrementa i, y sigue así hasta que i es mayor o igual que 10.

Hay que indicar que no es necesario poner ninguna de las expresiones en un `for`, la primera y tercera expresión son realmente sentencias ejecutables, por lo que no importa si las ponemos dentro o fuera del `for`, pero si no ponemos la segunda el compilador asume que la condición es verdadera y ejecuta un bucle infinito.

Por último diremos que una característica interesante de esta estructura es que los índices no modifican su valor al salir del bucle, por lo que se pueden utilizar después con el valor que les ha hecho salir del bucle.

Estructuras de salto

El C++ pretende ser un lenguaje eficiente, por lo que nos da la posibilidad de romper la secuencia de los algoritmos de una forma rápida, sin necesidad de testear infinitas variables para salir de un bucle. Disponemos de varias sentencias de ruptura de secuencia que se listan a continuación.

La sentencia break

Es una sentencia muy útil, se emplea para salir de los bucles (`do-while`, `while` y `for`) o de un `switch`. De cualquier forma esta sentencia sólo sale del bucle o switch más interior, si tenemos varios bucles anidados sólo salimos de aquel que ejecuta el `break` .

La sentencia continue

Esta sentencia se emplea para saltar directamente a evaluar la condición de un bucle desde cualquier punto de su interior. Esto es útil cuando sabemos que después de una sentencia no vamos a hacer nada más en esa iteración.

Hay que señalar que en los `for` lo que hace es saltar a la expresión 3 y luego evaluar la expresión 2

La sentencia goto

Sentencia maldita de la programación, es el salto incondicional. Para emplearla basta definir una etiqueta en cualquier punto del programa (un identificador seguido de dos puntos) y escribir `goto etiqueta`.

Ejemplo:

```
for (;;) {
  do {
    do {
      if (terminado)
        goto OUT;
    } while (…);
  } while (…);
}
OUT : cout << "He salido" << endl;
```

El goto sólo se puede usar dentro de la función donde se define la etiqueta y no se puede poner antes de una declaración de variables.

La sentencia return

Esta sentencia se emplea en las funciones, para retornar un valor. En el punto en el que aparece el return la función termina y devuelve el valor. Si una función no devuelve nada podemos poner return sin parámetros para terminar (si no lo ponemos la función retorna al terminar su bloque de sentencias).

Ejemplo:

```
int min (int a, int b) {
  if (a<b)
    return a;           // si entramos aquí la función retorna a y no sigue
                        // ejecutando lo que hay debajo
  return b;             // si no hemos salido antes b es el mínimo.
  // si pusiéramos más sentencias nunca se ejecutarían
}
```

FUNCIONES

Declaración de funciones

La declaración de una función nos da el nombre de la función, el tipo del valor que retorna y el número y tipo de parámetros que deben pasársele. La sintaxis es:

```
tipo_retorno nom_funcion (lista_tipos_param);
lista_tipos_param = tipo_param_1, tipo_param_2, … , tipo_param_n
```

Donde los paréntesis y el punto y coma son obligatorios. El tipo de retorno se puede omitir pero el compilador asume que es int. En C una función sin lista de parámetros se considera que tiene un número de parámetros indefinidos, mientras que en C++ se entiende que no se le pasa nada (para pasar un número de parámetros indefinido se ponen tres puntos (...) en la lista de parámetros). Lo más recomendable para evitar confusiones es poner siempre void como parámetro cuando no vamos ha pasar nada.

En la lista de parámetros podemos ponerle nombre a los parámetros, pero el compilador los ignorará.

Definición de funciones

Una definición de una función es una declaración en la que se incluye el cuerpo de la misma y se da nombre a los parámetros. La sintaxis es:

```
tipo_retorno nom_funcion (lista_param) {
  cuerpo de la función
}

lista_param = tipo_param_1 nom_param_1, … , tipo_param_n nom_param_2;
```

Donde los nombres de los parámetros son obligatorios.

Hay que indicar que una lista de parámetros debe indicar explícitamente el tipo de cada uno de ellos, así:

```
int producto (int a, b) { return (a*b); }
```

sería incorrecto, ya que el segundo parámetro no tiene tipo, habría que escribir:

```
int producto (int a, int b) { return (a*b); }
```

Es decir, una lista de parámetros no es una declaración de variables.

Si no deseamos usar algún parámetro podemos indicarlo no poniéndole nombre en la definición de la función.

Dentro del cuerpo de la función podemos declarar variables, pero no funciones, es decir, no se permiten funciones anidadas.

Paso de parámetros

En C++ todos los parámetros se pasan por valor, es decir, se hace una copia de los parámetros cuando llamamos a la función. En la llamada, antes de hacer la copia, se chequea que los parámetros actuales (los valores o variables especificados en la llamada) son del mismo tipo que los parámetros formales (los que declaramos en la lista de parámetros de la declaración de la función). Si los tipos son iguales o existe una conversión implícita la llamada puede realizarse, si no son iguales deberemos realizar conversiones explícitas.

Veamos un ejemplo de paso de parámetros:

```
#include <iostream.h>

void f (int val, int& ref) {
  val++;
  ref++;
}

main (){
  int i=1;
  int j=1;

  f(i.j);

  cout << "i vale " << i << " y j " << j << endl;
}
```

El resultado de ejecutar el programa será i vale 1 y j vale 2. Esto se debe a que cuando pasamos el parámetro val, esperamos un entero y lo que recibe la función es una copia del valor de i(1), al incrementar val en 1 modificamos la copia y la i queda igual. Sin embargo, el segundo parámetro es una referencia (la dirección de una variable entera), y al llamar a la función con j lo que se copia es su dirección, luego las modificaciones de ref se producen en la posición en la que se encuentra j. Como se ve, podemos considerar que al poner un parámetro de tipo referencia estamos realmente pasando el parámetro por referencia.

Hay que indicar que las referencias se usan igual que las variables del tipo, es decir, el ref++ no modifica la dirección, sino que modifica el valor referenciado.

Parámetros array

Hay un caso especial en el que aparentemente no se copia el parámetro, que es al pasar vectores. Por ejemplo:

```
void multiplica_matriz (int a[5], int val) {
  for (int i=0; i<5; i++)
    a[i] *= val;
}

main () {
  int m[5] = {1, 2, 3, 4, 5};

  multiplica_matriz(m, 2);
}
```

Después de la llamada a multiplica_matriz m valdría {2,4,6,8,10}. ¿Qué ha sucedido?

El problema está en la forma de tratar los vectores del C++, ya que el nombre de un vector o una matriz es en realidad un puntero al primer elemento de la misma, y utilizando el operador [] lo que hacemos es sumarle a ese puntero el tamaño de los elementos del array multiplicado por el valor entre corchetes. Esta es la razón de que los vectores comiencen en 0, ya que el primer elemento es el apuntado por el nombre del vector.

Todo esto implica que realmente su hayamos copiado el parámetro, sólo que el parámetro es el puntero al primer elemento, no el contenido del vector, y por tanto las modificaciones del contenido se hagan sobre los valores de la matriz pasada como parámetro.

Veremos en el punto dedicado a variables dinámicas que realmente los tipos vector y puntero son equivalentes (podremos usarlos indistintamente).

Retorno de valores

Para retornar valores una función emplea la instrucción return, que como ya vimos se puede colocar en cualquier punto de la misma y más de una vez. Cuando se ejecuta un return la función sale.

Hay que indicar que también se chequea el tipo de los retornos, es decir, lo que retornemos debe ser del mismo tipo (o convertible implícitamente) que el declarado como de retorno para la función.

Cuando una función se declara de manera que retorna algo, es un error no poner ningún return en la misma.

Sobrecarga de funciones

Una de las características más interesantes del C++ en cuanto a funciones se refiere es la posibilidad de definir distintas funciones con el mismo nombre aunque con distintos parámetros.

Esta capacidad se denomina sobrecarga de funciones y es útil para llamar de la misma forma a funciones que realizan operaciones similares pero sobre operandos distintos. En está frase hemos dado una clave para la definición de los TAD en C++, los operadores son funciones y como tales también pueden ser sobrecargadas. Este aspecto se estudiará una vez hayamos visto las clases, ya que tienen su mayor aplicación en estas.

El tipo de retorno debe ser igual, ya que de lo contrario se pueden provocar ambigüedades como que llamemos a una función esperando un real y tengamos dos funciones idénticas, una que retorna reales y otra enteros, ¿Cuál debemos usar? Si no hubiera conversiones implícitas estaría claro, pero podemos querer usar la que retorna enteros y que se transforme en real. La solución es no permitir retornos diferentes, lo único que debemos hacer es darles nombres diferentes a las funciones.

Parámetros por defecto

Algunas veces una función necesita determinados parámetros en condiciones excepcionales pero estos suelen tener unos valores fijos en los casos normales. Para no tener que dar estos parámetros en los casos normales el C++ permite el uso de parámetros por defecto. La sintaxis es muy simple, al declarar una función ponemos lo mismo que antes sólo que los parámetros por defecto van al final y se escriben no sólo poniendo el tipo sino también un signo igual y un valor:

```
tipo nom_funcion (lista_param, lista_param_por_defecto);

lista_param_por_defecto = tipo_pd_1 = vd1, …, tipo_pd_n = vd_n
```

Para usar la función podemos llamarla especificando sólo los parámetros indefinidos o poniendo estos y algunos parámetros por defecto con un valor.

Ejemplo:

```
void imprime_int (int valor, int base = 10);
// también se puede declarar como
// void imprime_int (int , int = 10);
// los nombres no importan

void imprime_int (int valor, int base) {
  switch (base) {
    case 8:
      cout << oct << i;
      break;
    case 10:
      cout << dec << i;
      break;
    case 16:
      cout << hex << i;
      break;
    default:
      cerr << "Función imprime_int (): Representación en base" << base \
           << " indefinida\n";
  }
}
```

Para imprimir un número en decimal haremos:

```
imprime_int (num);
```

Si queremos cambiar la base hacemos:

```
imprime_int (num, 8);    // imprime en octal
imprime_int (num,16);    // imprime en hexadecimal
imprime_int (num, 4);    // imprime un error.
```

Parámetros indefinidos

Para determinadas funciones puede ser imposible especificar el número y tipo de todos los parámetros esperados, por lo que el C++ define un mecanismo para utilizar funciones con un número de parámetros indefinido. La sintaxis para declarar las funciones es:

```
tipo nom_funcion (lista_args ...);  // al terminar la lista de args no ponemos coma
```

de esta manera, podemos declarar funciones que reciban siempre una serie de parámetros de forma normal y luego otros indefinidos.

Para acceder a los parámetros variables empleamos un conjunto de macros definidas en la cabecera estándar `<stdarg.h>`. Haremos lo siguiente:

1. Declaramos una variable del tipo `va_list` que es el tipo que asignamos a la lista de parámetros indefinidos.

2. Para inicializar la lista de argumentos empleamos la macro `va_start` que toma como argumentos la variable de tipo `va_list` y el último parámetro formal.

3. Para ir accediendo a los distintos elementos utilizamos la macro `va_arg`, que va leyendo los parámetros en orden y toma como argumentos la lista de tipo `va_list` y el tipo de la siguiente variable.

4. Antes de salir de una función que ha llamado a `va_start` debemos llamar a `va_end` con la variable de tipo `va_list` como parámetro.

El problema fundamental es saber que tipo asignar a cada variable y cuantas hay, la solución depende del tipo de parámetros que pasemos y lo que haga la función. Una forma fácil de saber

el tipo y número de parámetros variables es pasarle a la función una cadena de caracteres con los tipos.

Veamos como se usa con un ejemplo:

```
#include <iostream.h>
#include <stdarg.h>   // macros parámetros variables
#include <string.h>   // función strlen

void param_test  (char * ...);

void param_test (char *tipos ...) {
  int i;
  va_list ap;  // ap es la lista de parámetros

  va_start (ap, tipos);  // inicialización lista de param.
  i = strlen (tipos);    // la longitud de la cadena es el nº de param

// nuestra función reconoce tipos enteros y tipos reales (los reales sólo de tipo
// double).

  for (int j=0; j<i; j++) {
    switch (tipos[j]) {
     case 'e':
       int iv = va_arg(ap, int);
       cout << "Parámetro " << j << " = " << iv << " de tipo entero\n";
       break;
     case 'r':
       double dv = va_arg(ap, double);
       cout << "Parámetro " << j << " = " << dv << " de tipo real\n";
       break;
     default:
       cout << "Parámetro " << j << " de tipo desconocido\n";
       return;
    } // end switch
  } // end for

  va_end(ap);          // terminamos con la lista de param.
}

void main (void) {
  param_test ("eer", 12, 5, 5.35);
};
```

La salida de este ejemplo es:

```
Parámetro 0 = 12 de tipo entero
Parámetro 1 = 5 de tipo entero
Parámetro 2 = 5.35 de tipo real
```

Hay que decir que en el programa salimos abruptamente cuando no conocemos el tipo porque el resto de parámetros se leerán mal si realmente nos hemos equivocado. Por otro lado, si le decimos que el parámetro es de un tipo y lo pasado como parámetro es de otro lo más normal es que la impresión sea incorrecta, ya que no hacemos chequeo de tipos y si pasamos un entero (que, por ejemplo, ocupa 2 bytes) y leemos un real (por ejemplo de 4 bytes), habremos leído dos bytes de más, que le harán perder el sentido al real y además consumirán 2 bytes del siguiente parámetro. Es decir, si pasamos un parámetro erróneo lo más probable es que los demás también se pierdan.

Recursividad

Las funciones del C++ pueden ser recursivas, es decir, se pueden llamar a sí mismas. Lo único interesante de las funciones recursivas es que las variables declaradas dentro del cuerpo de la función que sean estáticas (`static`) no pierden su valor entre llamadas, es decir, no se crean y se destruyen en la pila, sino que ocupan siempre la misma posición, y por tanto cada modificación que se haga de la variable será valida entre sucesivas llamadas. Otra ventaja de esta aproximación es que si no importa para la recursividad que la variable mantenga su valor

después de una llamada recursiva (por ejemplo una variable temporal para intercambios), al declararla estática no tenemos que reservar espacio para ella en cada llamada (las llamadas recursivas consumen menos pila).

Aunque comento esto para funciones recursivas, la verdad es que esto se cumple para todas las funciones, luego podemos tener una variable estática que se use para contar el número de veces que se llama a una función (por ejemplo).

Punteros a funciones

Con las funciones sólo podemos hacer dos cosas, llamarlas u obtener su dirección, es decir, podemos definir punteros a funciones (que luego nos sirven para llamarlas).

La declaración de un puntero a función se hace:

```
tipo_retorno (*nom_var) (lista_tipos_argumentos);
```

El paréntesis es necesario, ya que el operador de función tiene más preferencia que el puntero, por lo que si escribimos:

```
tipo_retorno *nom_var (lista_tipos_argumentos);
```

el compilador interpretará que tenemos una función que retorna un puntero a un elemento de tipo `tipo_retorno`.

Para llamar a la función sólo tenemos que escribir:

```
(*nom_var) (param_actuales);
```

si ponemos:

```
nom_var (param_actuales);
```

el compilador seguramente identificará que `nom_var` es una función y llamará correctamente a la función, aunque es mejor no fiarse de eso.

Para asignar valor a un puntero a función sólo tenemos que escribir:

```
nom_var= &nom_funcion;
```

donde `nom_función` corresponde a una función con parámetros y retorno idénticos a los definidos en el puntero a función.

La función main()

Para terminar con las funciones hablaremos de la función principal de los programas de C++, la función `main()`.

La función `main()` se puede definir de varias formas distintas:

```
1. void main ();  // no recibe parámetros ni retorna nada
2. int  main ();  // no recibe parámetros y retorna un entero al SO (un código de
                  // error (generalmente negativo) o 0 si no hay errores)
       main ();  // igual que la anterior
3. void main (int argc, char *argv[]);  // recibe un array con 'argc' cadenas de
                                        // caracteres y no retorna nada
4. int  main (int argc, char *argv[]);  // igual que la anterior pero retorna un
                                        // código de error al SO
```

La tercera y cuarta formas reciben parámetros desde la línea de comandos en Sistemas Operativos como UNIX o MS-DOS. Es decir, cuando en MS-DOS escribimos un comando como:

```
C:\> COPY A:\MIO\PROGRAM.C C:
```

lo que en realidad hacemos es llamar al programa COPY pasándole una serie de parámetros. Lo que recibe la función `main()` es la línea de comandos, es decir, el nombre de la función seguido de los parámetros, cada palabra será una de las cadenas del array `argv[]`. Para el ejemplo, `main()` recibirá:

```
argc = 3
argv[1] = "COPY"
argv[1] = "A:\MIO\PROGRAM.C"
argv[2] = "C:"
```

Por último, para salir de un programa en C++ tenemos dos opciones, retornando un valor al SO en la función `main()` (o cuando esta termine si no retorna nada) o empleando la función `exit()`, declarada en la cabecera estándar `<stdlib.h>`:

```
void exit (int);
```

El entero que recibe la función `exit()` será el valor que se retorne al SO. Generalmente la salida con `exit()` se utiliza cuando se produce un error.

VARIABLES DINÁMICAS

En la mayoría de los lenguajes de alto nivel actuales existe la posibilidad de trabajar con variables dinámicas, que son aquellas que se crean en tiempo de ejecución. Para soportar el empleo de estas variables aparecen los conceptos de puntero y referencia que están íntimamente relacionados con el concepto de dirección física de una variable.

Punteros y direcciones

Ya hemos mencionado que el C++ es un lenguaje que pretende acercarse mucho al nivel de máquina por razones de eficiencia, tanto temporal como espacial. Por está razón el C++ nos permite controlar casi todos los aspectos de la ocupación y gestión de memoria de nuestros programas (sabemos lo que ocupan las variables, podemos trabajar con direcciones, etc.).

Uno de los conceptos fundamentales en este sentido es el de puntero. Un puntero es una variable que apunta a la dirección de memoria donde se encuentra otra variable. La clave aquí está en la idea de que un puntero es una dirección de memoria.

Pero, ¿cómo conocemos la dirección de una variable declarada en nuestro programa? La solución a esto está en el uso del operador de referencia (&), ya mencionado al hablar de los operadores de indirección. Para obtener la dirección de una variable solo hay aplicarle el operador de referencia (escribiendo el símbolo & seguido de la variable), por ejemplo:

```
int i = 2;
int *pi = &i;  // ahora pi contiene la dirección de la variable i.
```

El operador de indirección sólo se puede ser aplicado a variables y funciones, es decir, a LValues. Por tanto, sería un error aplicarlo a una expresión (ya que no tiene dirección).

Por otro lado, para acceder a la variable apuntada por un puntero se emplea el operador de indirección (*) poniendo el * y después el nombre del puntero:

```
int j = *pi;  // j tomaría el valor 2, que es el contenido de la variable i anterior
```

Para declarar variables puntero ponemos el tipo de variables a las que va a apuntar y el nombre del puntero precedido de un asterisco. Hay que tener cuidado al definir varios punteros en una misma línea, ya que el asterisco se asocia al nombre de la variable y no al tipo. Veamos esto con un ejemplo:

```
char *c, d, *e;  // c y e son punteros a carácter, pero d es una variable carácter
```

Como los punteros son variables podemos emplearlos en asignaciones y operaciones, pero hay que diferenciar claramente entre los punteros como dato (direcciones) y el valor al que apuntan. Veremos esto con un ejemplo, si tenemos los punteros:

```
int *i, *j;
```

La operación:

```
i= j;
```

hace que i y j apunten a la misma dirección de memoria, pero la operación:

```
*i = *j;
```

hace que lo apuntado por i pase a valer lo mismo que lo apuntado por j, es decir, los punteros no han cambiado, pero lo que contiene la dirección a la que apunta i vale lo mismo que lo que contiene la dirección a la que apunta j. Es decir, si i y j apuntaran a las variables enteras a y b respectivamente:

```
int a, b;
int *i = &a;
int *j = &b;
```

lo anterior sería equivalente a:

```
a= b;
```

Por último indicaremos que hay que tener mucho cuidado para no utilizar punteros sin inicializar, ya que no sabemos cuál puede ser el contenido de la dirección indefinida que contiene una variable puntero sin inicializar.

El puntero NULL

Siempre que trabajamos con punteros solemos necesitar un valor que nos indique que el puntero es nulo (es decir, que no apuntamos a nada). Esto se consigue dándole al puntero el valor 0 o NULL. NULL no es una palabra reservada, sino que se define como una macro en las cabeceras estándar `<stddef.h>` y `<stdlib.h>`, y por tanto será necesario incluirlas para usarlo. Si no queremos usar las cabeceras podemos definirlo nosotros de alguna de las siguientes formas:

```
#define NULL (0)
#define NULL (0L)
#define NULL ((void *) 0)
```

la primera y segunda formas son válidas porque 0 y 0L tienen conversión implícita a puntero, y la tercera es válida porque convertimos explícitamente el 0 a puntero `void`. Una forma adecuada de definir NULL es escribiendo:

```
#ifndef NULL
  #define NULL ((void *) 0)
#endif
```

que define NULL sólo si no está definido (podría darse el caso de que nosotros no incluyéramos las cabeceras que definen NULL, pero si se hiciese desde alguna otra cabecera que si hemos incluido).

Cualquier indirección al puntero NULL se transforma en un error de ejecución.

Punteros void

Ya hemos mencionado que el C++ define un tipo especial denominado `void` (vacío), que utilizábamos para indicar que una función no retorna nada o no toma parámetros. Además el tipo `void` se emplea como base para declarar punteros a variables de tipo desconocido.

Debemos recordar que no se pueden declarar variables de tipo `void`, por lo que estos punteros tendrán una serie de restricciones de uso.

Un puntero `void` se declara de la forma normal:

```
void *ptr;
```

y se usa sólo en asignaciones de punteros. Para trabajar con los datos apuntados por un puntero tendremos que realizar conversiones de tipo explícitas (casts):

```
char a;
char *p = (char *) ptr;

a = *p;
```

¿Cuál es la utilidad de estos punteros si sólo se pueden usar en asignaciones? Realmente se emplean para operaciones en las que no nos importa el contenido de la memoria apuntada, sino sólo la dirección, como por ejemplo en las funciones estándar de C para manejo de memoria dinámica (que también son válidas en C++, aunque este lenguaje ha introducido un operador que se encarga de lo mismo y que es más cómodo de utilizar). La definición de algunas de estas funciones es:

```
void *malloc (size_t N);  // reserva N bytes de memoria
void free (void *);       // libera la memoria reservada con malloc

/*
 size_t es un tipo que se usa para almacenar tamaños de memoria definido en las
cabeceras estándar mediante un typedef, generalmente lo consideraremos un entero sin
más
*/
```

y para usarlas hacemos:

```
void *p;
p= malloc (1000);      // reservamos 1000 bytes y asignamos a p la dirección del
                       // primer byte
free(p);               // liberamos la memoria asignada con malloc.
```

Aunque podemos ahorrarnos el puntero `void` haciendo una conversión explícita del retorno de la función:

```
long *c;
c=(long *)malloc(1000);  // aunque se transforma en puntero a long sigue reservando
                         // 1000 bytes, luego si cada long ocupa 4 bytes sólo nos
                         // cabrán 250 variables de tipo long
```

Por último señalar que en todas las ocasiones en las que hemos hecho conversiones con punteros hemos usado él método de C y no el de C++:

```
c = long *(malloc (1000));
```

ya que esto último da error de compilación. La solución a este problema es definir tipos puntero usando `typedef`:

```
typedef long *ptr_long;
c = ptr_long(malloc(1000));
```

Aritmética con punteros

Las variables de tipo puntero contienen direcciones, por lo que todas ellas ocuparan la misma memoria: tantos bytes como sean necesarios para almacenar una dirección en el computador sobre el que trabajemos. De todas formas, no podemos declarar variables puntero sin especificar a qué tipo apuntan (excepto en el caso de los punteros `void`, ya mencionados), ya que no es sólo la dirección lo que nos interesa, sino que también debemos saber que es lo apuntado para los chequeos de tipos cuando dereferenciamos (al tomar el valor apuntado para

usarlo en una expresión) y para saber que cantidades debemos sumar o restar a un puntero a la hora de incrementar o decrementar su valor.

Es decir, el incremento o decremento de un puntero en una unidad se traduce en el incremento o decremento de la dirección que contiene en tantas unidades como bytes ocupan las variables del tipo al que apunta.

Además de en sumas y restas los punteros se pueden usar en comparaciones (siempre y cuando los punteros a comparar apunten al mismo tipo). No podemos multiplicar ni dividir punteros.

Punteros y parámetros de funciones

Lo único que hay que indicar aquí es que los punteros se tratan como las demás variables al ser empleadas como parámetro en una función, pero tienen la ventaja de que podemos poner como parámetro actual una variable del tipo al que apunta el puntero a la que aplicamos el operador de referencia. Esta frase tan complicada se comprende mejor con un ejemplo, si tenemos una función declarada como:

```
void f (int *);
```

es decir, una función que no retorna nada y recibe como parámetro un puntero a entero, podemos llamarla con:

```
int i;
f(&i);
```

Lo que f recibirá será el puntero a la variable i.

Punteros y arrays

La relación entre punteros y arrays en C++ es tan grande que muchas veces se emplean indistintamente punteros y vectores. La relación entre unos y otros se basa en la forma de tratar los vectores. En realidad, lo que hacemos cuando definimos un vector como:

```
int v[100];
```

es reservar espacio para 100 enteros. Para poder acceder a cada uno de los elementos ponemos el nombre del vector y el índice del elemento al que queremos acceder:

```
v[8] = 100;
```

Pero, ¿cómo gestiona el compilador los vectores?. En realidad, el compilador reserva un espacio contiguo en memoria de tamaño igual al número de elementos del vector por el número de bytes que ocupan los elementos del array y guarda en una variable la dirección del primer elemento del vector. Para acceder al elemento i lo que hace el compilador es sumarle a la primera dirección el número de índice multiplicado por el tamaño de los elementos del vector. Esta es la razón de que los vectores comiencen en 0, ya que la primera dirección más cero es la dirección del primer elemento.

Hemos comentado todo esto porque en realidad esa variable que contiene la dirección del primer elemento de un vector es en realidad un puntero a los elementos del vector y se puede utilizar como tal. La variable puntero se usa escribiendo el nombre de la variable array sin el operador de indexado (los corchetes):

```
v [0] = 1100;
```

es igual que:

```
*v = 1100;
```

Para acceder al elemento 8 hacemos:

```
*(v + 8)
```

Es decir, la única diferencia entre declarar un vector y un puntero es que la primera opción hace que el compilador reserve memoria para almacenar elementos del tipo en tiempo de compilación, mientras que al declarar un puntero no reservamos memoria para los datos que va a apuntar y sólo lo podremos hacer en tiempo de ejecución (con los operadores new y delete). Excepto por la reserva de memoria y porque no podemos modificar el valor de la variable de tipo vector (no podemos hacer que apunte a una distinta a la que se ha reservado para ella), los vectores y punteros son idénticos.

Todo el tiempo hemos hablado sobre vectores, pero refiriéndonos a vectores de una dimensión, los vectores de más de una dimensión se acceden sumando el valor del índice más a la derecha con el segundo índice de la derecha por el número de elementos de la derecha, etc. Veamos como acceder mediante punteros a elementos de un vector de dos dimensiones:

```
int mat[4][8];

*(mat + 0*8 + 0)  // accedemos a mat[0][0]
*(mat + 1*8 + 3)  // accedemos a mat[1][3]
*(mat + 3*8 + 7)  // accedemos a mat[3][7] (último elemento de la matriz)
```

Por último mencionar que podemos mezclar vectores con punteros (el operador de vector tiene más precedencia que el de puntero, para declarar punteros a vectores hacen falta paréntesis).

Ejemplos:

```
int (*p)[20];  // puntero a un vector de 20 enteros
int p[][20];   // igual que antes, pero p no se puede modificar
int *p[20];    // vector de 20 punteros a entero
```

Operadores new y delete

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 base a 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 como 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

/*
  En las versiones del ANSI C++ 2.0 y anteriores el delete se debía poner como:

  delete [10]v;         // Libera espacio para 10 elementos del tipo de v
*/
```

Hay que tener cuidado con el `delete`, si ponemos:

```
delete v;
```

sólo 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
```

En caso de que se produzca un error al asignar memoria con `new`, se genera una llamada a la función apuntada por

```
void (* _new_handler)();  // puntero a función que no retorna nada y no tiene
                          // parámetros
```

Este puntero se define en la cabecera `<new.h>` y podemos modificarlo para que apunte a una función nuestra que trate el error. Por ejemplo:

```
#include <new.h>

void f() {
...
  cout << "Error asignando memoria" << endl;
...
}

void main () {
...
_new_handler = f;
...
}
```

Punteros y estructuras

Podemos realizar todas las combinaciones que queramos con punteros y estructuras: podemos definir punteros a estructuras, campos de estructuras pueden ser punteros, campos de estructuras pueden ser punteros a la misma estructura, etc.

La particularidad de los punteros a estructuras está en que el C++ define un operador que a la vez que indirecciona el puntero a la estructura accede a un campo de la misma. Este operador es el signo menos seguido de un mayor que (->). Veamos un ejemplo:

```
struct dos_enteros {
  int i1;
  int i2;
};

dos_enteros *p;

(*p).i1 = 10;  // asignamos 10 al campo i1 de la estructura apuntada por p
p->i2 = 20;    // asignamos 20 al campo i2 de la estructura apuntada por p
```

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

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)
```

Como se ve, el uso de punteros a punteros puede llegar a hacerse muy complicado, sobre todo teniendo en cuenta que en el ejemplo sólo hemos hecho asignaciones y no incrementos o decrementos (para eso hay que mirar la precedencia de los operadores).

PROGRAMACIÓN EFICIENTE

En este punto veremos una serie de mecanismos del C++ útiles para hacer que nuestros programas sean más eficientes. Comenzaremos viendo como se organizan y compilan los programas, y luego veremos que construcciones nos permiten optimizar los programas.

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 sólo 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 sólo 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 sólo 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).

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 sólo 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.hDefinición de streams cadena
```

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 podemos controlarlas nosotros mediante el uso de directivas de preprocesado. Como veremos, estas directivas nos 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.

Fases de preprocesado

1. Traduce los caracteres de fin de línea del fichero fuente a un formato que reconozca el compilador. Convierte los trigrafos en caracteres simples.

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          Incluye un fichero del directorio actual
"fichero"
#include          Incluye un fichero del directorio por defecto
<fichero>
#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 expr        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 errorea a ID. Modifica el valor de los macros
                  predefinidos __LINE__ y __FILE__
#pragma OPCION    Especifica al compilador opciones especificas 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.

Macros predefinidos

Existe un conjunto de macros predefinidos que contienen cierta información actualizada durante el preprocesado:

```
__LINE__     Constante decimal con el número de línea del fichero fuente donde se utiliza
__FILE__     Constante literal con el nombre del fichero fuente que se está compilando
__DATE__     Constante literal con la fecha de la compilación. Es de la forma "Mmm dd yyy"
__TIME__     Constante literal con la hora de compilación. Es de la forma "hh:mm:ss"
__cplusplus  Está definida cuando compilamos un programa en C++. En C no.
```

Secuencias trigrafo

Se definen en el estándar para poder escribir programas C/C++ en máquinas con código de caracteres reducido (7 bits). Consiste en reemplazar caracteres normales por secuencias de 3 caracteres que representan lo mismo. Las equivalencias son:

```
??=   #
??/   \
??'   ~
??(   {
??)   }
??!   |
```

Prácticamente no se usan.

Funciones inline

Como ya hemos visto, las macros permiten que resumamos una serie de operaciones con una sola palabra, pero cuando las usamos como funciones corremos el peligro de pasar parámetros erróneos. Por este motivo el C++ introdujo el concepto de función inline. Una función inline es igual que una función normal (excepto que su declaración viene precedida por la palabra inline), 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 si que comprueban el tipo de los parámetros.

No se pueden definir funciones inline recursivas (evidentemente, no podemos sustituir infinitas veces su código).

Inclusión de rutinas en ensamblador

Para hacer más eficientes determinadas partes del código podemos incluir rutinas en ensamblador dentro de nuestro código C++ usando la palabra `asm`. La sintaxis es:

```
asm { … }
```

Dentro de las llaves escribimos código ensamblador de la máquina destino. El uso de esta facilidad debe estudiarse en el manual del compilador que utilicemos.

Eficiencia y claridad de los programas

Utilizando macros, funciones inline, rutinas en ensamblador o declarando variables de tipo register conseguimos una serie de optimizaciones sobre el código. El empleo de estas facilidades debe realizarse siempre que tenga sentido, es decir, mejore la velocidad de nuestro programa.

De todas formas, utilizando muchas técnicas de programación podemos mejorar programas en cualquier lenguaje de alto nivel (mejora de bucles sacando variables, eliminación de la recursividad en procedimientos recursivos, etc.).

Algunas de estas optimizaciones las pueden hacer los compiladores (aunque mejor no fiarse), y es recomendable aplicarlas sólo cuando vayamos a compilar la versión definitiva de un programa, ya que consumen mucho tiempo.

También es recomendable que las optimizaciones que realicemos en alto nivel sean claras, es decir, que el resultado de la optimización no haga el programa ilegible. Si esto no es posible no hay ningún problema siempre y cuando el código este comentado y sepamos de donde viene (como era el algoritmo antes de optimizar). Lo mejor es escribir los programas dos veces, una vez con algoritmos simples (que suelen ser poco eficientes pero claros) y después reescribirlo optimizando los algoritmos más cruciales, manteniendo la primera versión como documentación. Si escribís unos programas muy eficientes pero no los documentáis es fácil que se llegue ha hacer imposible su modificación o corrección por parte de otros (o de vosotros mismos después de un tiempo, y si no echarles un vistazo a vuestros programas de fundamentos).

CLASES

Introducción

Comenzamos en este punto con las características específicas del C++ que realmente lo hacen merecedor de ese postincremento que lo diferencia del C: las clases.

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 sólo se diferencian de los elementales en la forma de crearlos, no en la de usarlos.

Pero las clases no sólo 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 sólo 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 pueden 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).

Clases y miembros

Comenzaremos el estudio de las clases partiendo de la siguiente idea: 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 como 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 sólo 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.

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 pepe;  // declaramos un objeto de tipo empleado

pepe.sueldo = 500;          // asignamos el valor 500 al campo sueldo
pepe.modificar_sueldo(600)  // le decimos a pepe 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 sólo debe ser vista en la definición de la clase. Para que lo que contiene la clase sólo sea accesible desde la definición empleamos la palabra `class` en lugar de `struct` para definir el tipo:

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

Acceso a miembros: etiquetas de control de acceso

El uso de `class` da lugar a un problema ¿cómo accedemos a los miembros de la clase que si deben ser vistos desde fuera? La solución está en emplear la etiqueta `public` delante de los miembros que queremos que sean accesibles:

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

pepe.sueldo = 500;            // ERROR, sueldo es un miembro privado
pepe.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`). Mi consejo es 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 sólo permite el acceso a un miembro cada vez (es lo mismo que sucedía con las uniones como tipo de datos compuesto).

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;
  }
  ...
};
```

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) contenidos 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.

En realidad, los objetos de una clase tienen un atributo específico asociado, su dirección. La dirección del objeto nos permitirá saber que 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.

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;
}
```

Y ahora un ejemplo del segundo:

```
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.

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).

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 sólo 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.

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
```

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 (que se estudiarán más adelante).

Métodos estáticos y funciones amigas

Dentro de las peculiaridades de las clases encontramos dos tipos de funciones especiales: los métodos estáticos y las funciones amigas. Los comentamos separados del bloque relativo a clases y miembros por su similitud y por la importancia de las funciones amigas en la sobrecarga de operadores.

Su característica común es que no poseen parámetro implícito `this`. Aunque las expliquemos juntas sus aplicaciones son muy diferentes.

Métodos estáticos

Al igual que los atributos estáticos mencionados en el 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).

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`.

Construcción y destrucción

Hasta ahora hemos hablado de la declaración y definición de clases, pero hemos utilizado los objetos sin saber como se crean o se destruyen. En este punto veremos como las clases se crean y destruyen de distintas maneras y que podemos controlar que 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 el 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.

Los constructores

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 sólo podremos crear objetos de esa clase utilizando funciones amigas. A las clases que sólo tienen constructores privados se las suele denominar *privadas*.

Los constructores se pueden declarar virtuales (ya lo veremos).

El destructor

Para cada clase sólo 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;
}
```

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.

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 esta 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.

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);  // Creamos un complejo llamando al constructor con
                                // un parámetro.
```

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.

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

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 crearan 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 llamara 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
}
```

HERENCIA Y POLIMORFISMO

Hasta ahora hemos visto como definir clases en C++ y una serie de características de estas últimas, como la forma de crear objetos, de declarar miembros con distintos tipos de acceso, etcétera. Pero todas estas características son tan sólo una parte de la historia. Dijimos que el uso de objetos se introducía para representar conceptos del mundo de nuestro programa en una forma cómoda y que permitían el uso de las clases como tipos del lenguaje, pero, ¿cómo representamos las relaciones entre los objetos?, es decir, ¿cómo indicamos la relación entre las personas y los empleados, por ejemplo?.

Esto se consigue definiendo una serie de relaciones de *parentesco* entre las clases. 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 sólo 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 sólo 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.

Clases derivadas o subclases

Clases derivadas

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 tipos dinámico (no tiene forma de saber que un puntero a un tipo base realmente apunta a un objeto de la clase derivada).

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
```

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)
{
  ...
}
```

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.

A parte de el 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).

Por último mencionaremos que 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.

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 sólo 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 si pueden ser declarados virtuales.

Las funciones virtuales necesitan el parámetro `this` para saber que 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 porque saber si esta es virtual o no.

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 sólo 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 como 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 sólo 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 porque 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 sólo necesitarán implementar las funciones virtuales puras que su clase padre (la derivada de la abstracta original) no haya definido.

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.

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 porque 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 como podemos resolverlas.

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.

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 sólo 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 sólo 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?).

Necesidad de la herencia múltiple

Como hemos visto, la herencia múltiple le da mucha potencia al C++, pero por otro lado introduce una gran complejidad a la hora de definir las clases derivadas de más de una clase. En realidad no existe (que yo sepa), ninguna cosa que se pueda hacer con herencia múltiple que no se pueda hacer con herencia simple escribiendo más código.

Ha habido muchas discusiones por culpa de la incorporación de esta característica al C++, ya que complica mucho la escritura de los compiladores, pero como se ha incorporado al estándar, lo lógico es que todos los compiladores escritos a partir del estándar la incorporen. Por tanto, el uso o no de la herencia múltiple depende de las ventajas que nos reporte a la hora de hacer un programa. Lo más normal es que ni tan siquiera lleguéis a utilizarla.

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 sólo 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 quien 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`, sólo 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. Sólo 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.

SOBRECARGA DE OPERADORES

Ya hemos mencionado que el C++ permite la sobrecarga de operadores. 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 dereferencia `->` 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.

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 sólo es para simplificar la escritura. Por ejemplo:

```
int c = a + b;
```

es lo mismo que:

```
int c = operator+ (a, b);
```

Operadores unarios y binarios

En C++ tenemos operadores binarios (con dos operandos) y unarios (con un operando).

Los operadores binarios se pueden definir como una función miembro con un argumento (el otro será el objeto de la clase que lo emplee) o como funciones globales con dos argumentos. De esta forma, para cualquier operador binario `@`, `a@b` se interpretará como o como . Si las dos funciones están definidas, se aplican una serie de reglas para saber que función utilizar (si es posible decidirlo).

Los operadores unarios, ya sean prefijos o postfijos, se pueden definir como función miembro sin argumentos o como función global con un argumento. Igual que para operadores binarios, si definimos un operador de las dos formas habrá que determinar según unas reglas que operador emplear.

Veremos al hablar del incremento y decremento como se trabaja con operadores prefijos y postfijos.

Los operadores sólo se pueden definir del tipo (unario o binario) que tienen en C++, no podemos usar operadores binarios como unarios ni viceversa y no podemos definir operadores con más de 2 operandos.

Significados predefinidos de los operadores

Sobre los operadores definidos por el usuario sólo se hacen algunas consideraciones, en particular los operadores `operator=`, `operator[]`, `operator()` y `operator->` no pueden ser funciones miembro estáticas (esto asegura que sus primeros operandos serán LValues).

Los significados de algunos operadores predefinidos por el lenguaje están relacionados, pero esto no implica que cuando redefinamos un operador los relacionados con él mantengan esta relación. Por ejemplo, el operador `+=` y el operador `+` están relacionados, pero redefiniendo `+` no hacemos que `+=` emplee la nueva suma antes de la asignación. Si quisiéramos que esto fuera así deberíamos redefinir también el operador `+=`.

Operadores y clases

Todos los operadores sobrecargados deben ser miembros de una clase o tener como argumento un objeto de una clase (excepto para los operadores `new` y `delete`). En particular, no pueden definirse operadores que operen exclusivamente con punteros. Esto sirve para asegurar que los significados predefinidos del C++ no se pueden alterar (una expresión no puede modificar su significado a menos que intervengan objetos de clases definidas por el usuario).

Un operador que acepte como primer operando un tipo básico no puede ser una función miembro, ya que no podríamos interpretar el operador como miembro de una clase, puesto que no existe la definición de clase para los tipos básicos.

Una cosa importante a este respecto es tener en cuenta que la definición de un operador no es conmutativa, es decir, si queremos aplicar un operador al tipo A y al tipo B en ambos sentidos debemos definir dos funciones para cada operación.

Conversiones de tipos

Cuando queremos definir una serie de operadores para trabajar con una clase tenemos que redefinir cada operación para emplearla con los objetos de esa clase y después redefinirla también para todas las operaciones con otros tipos (y además en ambos sentidos, para operadores conmutativos). Para evitar tener que definir las funciones que operan con objetos de nuestra clase y objetos de otras clases podemos emplear un truco bastante simple: definimos un conversor de los tipos para pasar objetos de otros tipos a objetos de nuestra clase.

Uso de constructores

Un tipo de conversión de tipos es la realizada mediante constructores que aceptan como parámetro un objeto de un tipo y crean un objeto de nuestra clase usando el objeto parámetro. Por ejemplo, si tenemos un tipo complejo con la siguiente definición:

```
class complejo {
private:
  double re, im;
public:
  complejo(double r, double i=0) {   // constructor / conversor de double a complejo
    re = r; im = i;
  }
  // operadores como funciones amigas
  friend complejo operator+ (complejo, complejo);  // suma de complejos
  friend complejo operator* (complejo, complejo);  // producto de complejos
  ...
  // operadores como funciones miembro
  complejo operator+=(complejo);  // suma y asignación
  complejo operator*=(complejo);  // producto y asignación
  ...
};
```

El constructor de la clase complejo nos sirve como conversor de variables `double` a complejos, y además como los constructores con un sólo parámetro no necesitan ser invocados explícitamente, la sintaxis de la conversión es mucho más amigable:

```
complejo z1 = complejo (23);  // z1 toma el valor (23 + i*0)
complejo z2 = 23;             // z2 toma el valor (23 + i*0), llamamos
                              // implícitamente al constructor
```

Con la definición de la clase anterior, cualquier constante será convertida a `double` y generará el número complejo correcto. Si usáramos los dos parámetros del constructor, la conversión de constantes a `double` también se realizaría. Si en un operador se necesita un complejo como operando y empleamos un `double`, el operador llamara al constructor de complejos y transformará el operando en complejo. Sólo deberemos implementar operadores con parámetros de tipos distintos a nuestra clase si es necesaria la máxima eficiencia (nos evitaremos la construcción del objeto temporal).

Si el operador crea objetos temporales automáticamente, los destruirá en cuanto pueda (generalmente después de emplearlos en la operación). La conversión implícita sólo se realiza si el conversor definido por el usuario es único.

Operadores de conversión

La conversión de tipos usando constructores tiene algunos problemas:

1. No puede haber conversión implícita de un objeto de una clase a un tipo básico, ya que los tipos básicos no son clases.

2. No podemos especificar la conversión de un tipo nuevo a uno viejo sin modificar la vieja clase.

3. No es posible tener un constructor sin tener además un conversor.

El último problema no es realmente grave, ya que el empleo del constructor como conversor suele tener siempre un sentido, y los dos primeros problemas se solucionan definiendo operadores de conversión para el tipo fuente.

Una función miembro `X::operatorT()`, donde `T` es el nombre de un tipo, define una conversión de `X` a `T`. Este tipo de conversiones se deben definir sólo si son realmente necesarias, si se usan poco es preferible definir una función miembro normal para hacer las conversiones, ya que hay que llamarla explícitamente y puede evitar errores no intencionados.

Problemas de ambigüedad

Una asignación o inicialización de un objeto de una clase `X` es legal si, o bien el valor asignado es de tipo `X` o sólo hay una conversión de el valor asignado al tipo `X`. En algunos casos una conversión necesita el uso repetido de constructores o operadores de conversión, sólo se usará conversión implícita de usuario en un primer nivel, si son necesarias varias conversiones de usuario hay que hacerlas explícitamente. Si existe más de un conversor de tipos, la conversión implícita es ilegal.

Operadores y objetos grandes

Cuando una clase define objetos pequeños, la utilización de copias de los objetos en las conversiones o en las operaciones no causa mucho problema, pero su la clase define objetos de gran tamaño, la copia puede ser excesivamente costosa (ineficiente). Para evitar el empleo de copias podemos definir los argumentos (y retornos) de una función como referencias (recordemos que los punteros no se pueden usar porque no se puede modificar el significado de un operador cuando se aplica a punteros).

Los parámetros referencia no causan ningún problema, pero los retornos referencia deben se usados con cuidado: si hay que crear el objeto resultado es preferible retornar un objeto y que se copie, ya que la gestión de memoria para retornar referencias a objetos creados en la función del operador puede resultar muy complicada.

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 sólo construimos un objeto, pero destruimos dos. El operador de asignación definido por el usuario no se aplica a un objeto sin inicializar, 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);
}
```

Subíndices

El operador `operator[]` puede redefinirse para dar un significado a los subíndices para los objetos de una clase. Lo bueno es que el segundo operando (el subíndice) puede ser de cualquier tipo.

Para redefinir el operador de subíndice debemos definirlo como función miembro. Por ejemplo, para acceder a los elementos de un conjunto de enteros almacenado en una lista podemos redefinir el operador de subíndice:

```
class cjto {
private:
  nodo_lista *elem;
  ...
public:
  ...
  int operator[] (int);
  ...
}

int cjto::operator[] (int i) {
  nodo_lista *n = elem;           // puntero al primer elem. de la lista
  for (int k=0; k<i; k++)         // recorremos la lista hasta llegar al elem i
    if (!(n = n->sig)) return 0;  // siempre y cuando este exista
  return n->val;                  // retornamos el contenido de la posición i
}
```

Llamadas a función

La llamada a función, esto es, la notación `expresión(lista_expresiones)`, puede ser interpretada como una operación binaria con `expresión` como primer argumento y `lista_expresiones` como segundo. La llamada `operator()` puede ser sobrecargada como los otros operadores. La lista de expresiones se chequea como en las llamadas a función.

La sobrecarga de la llamada a función se redefine para emplear los objetos como llamadas a función (sobre todo para definir iteradores sobre clases), la ventaja de usar objetos y no funciones está en que los objetos tienen sus propios datos para guardar información sobre la

aplicación sucesiva de la función, mientras que las funciones normales no pueden hacerlo. Otro uso de la sobrecarga de la llamada a función está en su empleo como operador de subíndice, sobre todo para arrays multidimensionales.

Dereferencia

El operador de dereferencia `->` puede ser considerado como un operador unario postfijo. Dada una clase:

```
class Ptr {
  ...
  X* operator->();
};
```

podemos usar objetos de la clase `Ptr` para acceder a objetos de la clase `X` como si accediéramos a través de punteros:

```
Ptr p;

p->m = 7;  // (p.operator->)()->m = 7;
```

Como se ve, se aplica el operador dereferencia y luego con el puntero resultado se accede a un miembro. La transformación de `p` en el puntero `p.operator->()` no depende del miembro `m` al que se accede. En este sentido el operador es postfijo.

La utilidad de esta sobrecarga está en la definición de punteros inteligentes, objetos que sirven de punteros pero que realizan alguna función cuando accedemos a un objeto a través de ellos. La posibilidad de esta sobrecarga es importante para una clase interesante de programas, la razón es que la *indirección* es un concepto clave, y la sobrecarga de `->` proporciona una buena forma de representar la indirección en los programas.

Incremento y decremento

Los operadores de incremento y decremento son muy interesantes a la hora de sobrecargarlos por varias razones: pueden ser prefijos y postfijos, son ideales para representar los accesos a estructuras ordenadas (listas, arrays, pilas, etc.), y pueden definirse de forma que verifiquen rangos en tiempo de ejecución.

Para sobrecargar estos operadores en forma prefija hacemos lo de siempre, pero para indicar que el operador es postfijo lo definimos con un argumento entero (como el operador es unario, está claro que no se usará el parámetro, es un parámetro vacío, pero con la declaración el compilador distingue entre uso prefijo y postfijo)::

```
class Puntero_seguro_a_T {
  T*  p;     // puntero a T, valor inicial del array
  int tam;   // tamaño vector apuntado por T
  ...
  T* operator++ ();     // Prefijo
  T* operator++ (int);  // Postfijo
  ...
};
```

Definiendo los operadores de incremento y decremento de esta forma podremos verificar si accedemos a un valor superior o inferior del array, impidiendo errores por interferencia con otras variables.

Sobrecarga de new y delete

Al igual que el resto de operadores, los operadores `operator new` y `operator 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.

Funciones amigas o métodos

Una pregunta importante es: ¿Cuándo debo sobrecargar un operador como miembro o como función amiga?

En general, siempre es mejor usar miembros porque no introducen nuevos nombres globales. Cuando queremos definir operandos que afectan al estado de la clase debemos definirlos como miembros o como funciones amigas que toman una referencia no constante a un objeto de la clase. Si queremos emplear conversiones implícitas para todos los operandos de una operación, la función que sobrecarga el operador deberá ser global y recibir como parámetros una referencia constante o un argumento que no sea una referencia (esto permite la conversión de constantes).

Si no hay ninguna razón que nos incline a usar una cosa u otra lo mejor es usar miembros. Son más cómodos de definir y más claros a la hora de leer el programa. Es mucho más evidente que un operador puede modificar al objeto si es un miembro que si la función que lo implementa recibe una referencia a un objeto.

TEMPLATES

Genericidad

El 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.

El 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, sustituimos 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-funciones 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) el 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 este 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.

Funciones genéricas

Para definir una función genérica sólo 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>                          // sólo un tipo parámetro
T max (T a, T b) { return (a>b) ? a : b }  // función genérica máximo
```

Los tipos parámetro no sólo 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.

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 sólo 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 sólo 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 sólo 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 sólo queremos especializar un método, lo declaramos como siempre pero con el
// tipo para el que especializamos indicado
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
  ...
};
```

MANEJO DE EXCEPCIONES

Programación y errores

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.

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.

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.

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
  }
  ...
}
```

El bloque de prueba: try

El tratamiento de errores visto hasta ahora es muy limitado, ya que no tenemos forma de especificar donde se pueden producir errores (en que bloques del programa). La forma de especificar donde 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();
}
```

En realidad sólo 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 origino, sino que se sigue con el código que hay después del último `catch` asociado al `try` donde se acepto el error. En el ejemplo se ejecutaría la función `z()`.

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 sólo lanza cadenas y enteros
```

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

Funciones terminate() y unexpected()

Existen situaciones en las que un programa debe terminar abruptamente por que el manejo de excepciones no puede encontrar un manejador para una excepción lanzada, cuando la pila está corrompida (y no podemos ejecutar los mecanismos de excepción) o cuando un destructor llamado por una excepción provoca otra excepción.

En estos casos el programa llama automáticamente a una función llamada `terminate()` que no retorna nada y no tiene parámetros. Esta función llama a otra que podemos especificar nosotros mediante la llamada a una función denominada `set_terminate()`. Esta función acepta como parámetro punteros a funciones del tipo:

```
f () {  // función sin parámetros que no retorna nada (ni void)
  ...
}

set_terminate (&f);  // f() es la función que llamará terminate.
```

La función por defecto de `terminate()` es `abort()` que termina la ejecución sin hacer nada.

La función `unexpected()` se llama cuando una función lanza una excepción que no está en su lista `throw`, y hace lo mismo que `terminate()`, es decir, llama a una función. Podemos especificar a cual usando la función `set_unexpected()` que acepta punteros al mismo tipo de funciones que `set_terminate()`. La función por defecto de `unexpected()` es `terminate()`.

ENTRADA Y SALIDA

Introducción

Casi todos los lenguajes de alto nivel disponen de bibliotecas estándar de funciones para gestionar la Entrada/Salida de datos, tanto para teclado/pantalla como ficheros. El C++ no emplea esta estrategia, es decir, no define una biblioteca de funciones, sino que define una biblioteca de clases que se puede expandir y mejorar si la aplicación lo requiere.

La idea es que las operaciones de entrada y salida se aplican a objetos de unas clases determinadas, empleando la sobrecarga de operadores como método para indicar la forma de introducir y extraer datos hacia o desde la E/S a nuestro programa.

La forma de trabajar con la E/S hace que sea posible el chequeo de tipos de entrada y de salida, que tengamos una forma uniforme de leer y escribir variables de todos los tipos (incluso clases) e incluso que podamos tratar de forma similar la entrada salida para distintos dispositivos.

El concepto fundamental en C++ para tratar la entrada/salida es la noción de *stream* que se puede traducir como flujo o corriente. La idea es que existe un flujo de datos entre nuestro programa y el exterior, y los *streams* son los encargados de transportar la información, serán como un canal por el que mandamos y recibimos información.

El C++ define *streams* para gestionar la E/S de teclado y pantalla (entrada y salida estándar), la E/S de ficheros e incluso la gestión de E/S de cadenas de caracteres.

Primero estudiaremos la E/S entre nosotros y la máquina (teclado y pantalla) y luego veremos la gestión de ficheros y cadenas. Hablaremos primero de la entrada y la salida simples y luego comentaremos las posibilidades de formateo y el uso de los manipuladores de E/S.

Objetos Stream

El C++ define (al incluir la cabecera `<iostream.h>`) una serie de clases `stream`, las más importantes son `istream`, `ostream` y `iostream`, que definen `streams` de entrada, de salida y de entrada/salida respectivamente. Además de definir las clases, en esta cabecera se declaran una serie de objetos estándar que serán los que utilizaremos con más frecuencia:

```
cin   Objeto que recibe la entrada por teclado, pertenece a la clase istream
cout  Objeto que genera la salida por pantalla, pertenece a ostream
cerr  Objeto para salida de errores, es un ostream que inicialmente saca sus
       mensajes por pantalla, aunque se puede redirigir.
clog  Es igual que cerr, pero gestiona los buffers de forma diferente
```

Entrada y salida

En este punto describiremos la clase ios, que es la clase empleada para definir objetos de tipo stream para manejar la E/S. Primero veremos la descripción de la clase y luego veremos que cosas podemos utilizar para conocer el estado del stream, las posibilidades de formateo de E/S y una lista de funciones especiales de acceso a Streams.

La clase ios

```
class ios {
  ostream* tie(ostream* s);   // Liga dos streams
                              // podemos ligar entrada con salida
```

```
  ostream* tie();

  int width(int w);           // Pone la longitud de campo
  int width() const;          // Devuelve la longitud
  char fill(char);            // Pone carácter de relleno
  char fill() const;          // Devuelve carácter de relleno
   long flags(long f);         // Pone los flags del stream
  long flags() const;         // Devuelve los flags del stream
  setf(long setbits, long field);
  setf(long);
  unsetf(long);
  int precision(int);         // Pone la precisión de los reales
  int precision() const;      // Devuelve la precisión de los reales

  // Funciones de estado del stream
  int rdstate() const;
  int eof() const;
  int fail() const;
  int bad() const;
  int good() const;
  void clear(int i=0);

  operator void *();  // Retorna NULL si failbit, badbit o hardfail están a uno
  int operator !();   // Retorna verdadero si failbit, badbit o hardfail están a uno
};
```

Estado de la E/S.

```
enum io_state {
  goodbit = OxO0,    // Estado normal
  eofbit = OxOl,     // Al final del stream
  failbit = Ox02,    /* La última operación de E/S ha fallado. El stream se puede volver a
                        usar si se recupera el error. */
  badbit = Ox04,     /* La última operación es inválida. El stream se puede volver a usar
                        si se recupera el error */
  hardfail = Ox08    // Error irrecuperable
};
```

Flags

Podemos modificarlos con las funciones `setf()` y `unsetf()`.

FLAG	ESTADO	SIGNIFICADO
skipws	SI	Ignorar caracteres blancos en la entrada
left	NO	Salida justificada a la izquierda
right	NO	Salida justificada a la derecha
internal	NO	Usar 'padding' después del signo o de la indicación de base. Esto quiere decir que el signo se pone justificado a la izquierda y el número justificado a la derecha. El espacio del medio se llena con el carácter de relleno.
dec	SI	Base en decimal. Equivalente a la función dec
oct	NO	Base en octal. Equivalente a la función oct
hex	NO	Base en hexadecimal. Equivalente a la función hex
showbase	NO	Usar indicador de base en la salida. Por ejemplo si hex está ON, un número saldrá precedido de 0x y en hexadecimal. En octal saldría precedido únicamente de un 0
showpoint	NO	Poner en la salida el punto decimal y ceros a la derecha del punto (formato de coma) aunque no sea necesario para números reales
uppercase	NO	Usar mayúsculas para los números hexadecimales en la salida y la E del exponente
showpos	NO	Poner un '+' a todos los números positivos en la salida
scientific	NO	Usar notación exponencial en los números reales
fixed	NO	Usar notación fija en los números reales
unitbuf	NO	Volcar todos los streams después de la salida
stdio	NO	Volcar cout y cerr después de la salida

Manipuladores

MANIPULADOR	SENTIDO	ACCIÓN
dec	E/S	Pone la base a decimal hasta nueva orden
oct	E/S	Lo mismo a octal
hex	E/S	Lo mismo a hexadecimal
ws	E	Extrae los caracteres blancos del stream. Se usa cuando skipws está a 0
endl	S	Inserta una nueva línea. La única diferencia con el carácter '\n' es que endl vuelca el stream
ends	S	Inserta un carácter nulo. Completamente equivalente a ' \ 0 ' y vuelca el string
flush	S	Vuelca el stream
setw(int)	E/S	Pone el ancho del campo. (Equivalente a width()). Tope máximo para la entrada y mínimo para la salida
setfill(char)	S	Pone el carácter de relleno. Por defecto es el carácter espacio ' '. (Equivalente a fill())
setprecision(int)	S	Pone la precisión a N dígitos después del punto decimal. (Equivalente a precision())
setbase(int)	E/S	Pone la base (0, 8, 10, 16). 8 10 y 16 son equivalentes a oct, dec y hex que hemos visto antes. 0 implica dec en la salida y en la entrada significa que podemos usar las convenciones del C++ para introducir los números en distintas bases (0N: octal. 0xN: hexadecimal, N: decimal)
resetiosflags(long)		Equivalente a unsetf()
setiosflags(long)		Equivalente a setf()

Otras funciones de acceso a Streams

ostream& put(char);
Inserta un carácter en el stream de salida. Retorna el stream de salida.

int get();
Extrae el siguiente carácter del stream de entrada y lo retorna. Se retorna EOF si está vacío.

int peek();
Lo mismo que get() pero sin extraer el carácter.

istream& putback(char);
Pone de vuelta un carácter en el stream de entrada. En caso de querer meter otro carácter dará error. Retorna el stream de entrada.

istream& qet(char &):
Extrae el siguiente carácter del stream de entrada. Retorna el stream de entrada.

istream& get(char *s, int n, char t= '\n');
Extrae hasta n caracteres en s parando cuando se encuentra el carácter t o bien cuando se llega a fin de fichero o hasta que se han leído (n-1) caracteres . El carácter t no se almacena en s. pero sí un '\0' final en s. Retorna el stream de entrada. Falla sólo si no se extrae ningún carácter.

istream& getline(char *s, int n, char t= '\n');
Igual que la anterior pero en el caso de que se encuentre t se extrae y se añade en s.

istream& ignore(lnt n, int t= EOF);
Extrae y descarta hasta n caracteres o hasta que el carácter t se encuentre. El carácter t se saca del stream. Se retorna el stream de entrada.

int gcount();
Retorna el número de caracteres extraídos en la última extracción.

ostream& flush();
Vuelca el contenido del stream. Esto es vacía el buffer en la salida.

itream& read(char *s, int n);

Extrae n caracteres y los copia en s. Utilizar gcount() para ver cuantos caracteres han sido extraídos si la lectura termina en error.

ostream& seekp(streampos);
istream& seekg(streampos);
Posicionan el buffer de entrada o salida a la posición absoluta pasada como parámetro.

ostream& seekp(streamoff, seek_dir);
istream& seekg(streamoff, seek_dir);
Posiciona el buffer de entrada o salida relativamente en función del parámetro seek_dir:

```
enum seek_dir {
  beg=0,   // relativo al principio
  cur=l,   // relativo a la posición actual
  end=2    // relativo al fin de fichero
}
```

streampos tellp();
Devuelve la posición actual del buffer de salida.

streampos tellg();
Posición actual del buffer de entrada.

ostream& write(const char* s, int n);
Inserta n caracteres en el stream de salida (caracteres nulos incluidos).

Ficheros

Apertura y cierre

fstream::fstream()
Constructor por defecto. Inicializa un objeto de stream sin abrir un fichero.

fstream::fstream(const char *f, int ap, int p= S_IREAD | S_IWRITE);
Constructor que crea un objeto fstream y abre un fichero f con el modo de apertura ap y el modo de protección p.

fstream::~fstream()
Destructor que vuelca el buffer del fichero y cierra el fichero (si no se ha cerrado ya).

fstream::open(const char *f, int ap, int p= S_IREAD | S_IWRITE);
Abre el fichero f con el modo de apertura ap y con el modo de protección p.

int fstream::is_open();
Retorna distinto de cero si el fichero está abierto.

fstream::close();
Cierra el fichero si no está ya cerrado.

Modos de apertura

MODO	SIGNIFICADO
in	Abierto para lectura
out	Abierto para escritura
ate	Colocarse al final del fichero
app	Modo append. Toda la escritura ocurre al final del fichero
trunc	Borra el contenido del fichero al abrir si ya existe. Es el valor por defecto si sólo se especifica out
nocreate	El fichero debe existir en el momento de la apertura, si no, falla
noreplace	El fichero no debe existir en el momento de la apertura, si no, falla
binary	Los caracteres '\r' y '\n' no son convertidos. Activar cuando se trabaje con ficheros de datos binarios. Cuando se trabaje con textos dejarlo por defecto que es desactivado

Modos de protección

Los modos de protección dependen del sistema operativo y por tanto no existe una definición estándar de los mismos. Sólo se definen:

```
S_IREAD        Permiso de lectura
```

```
     S_IWRITE       Permiso de escritura
```

Otras funciones de gestión de ficheros

```
fstream (int fh)
Construye un stream usando un descriptor de fichero abierto existente descrito por fh.

attach(int fh);
Liga un stream con el descriptor fh. Si el stream ya está ligado da un error failbit.

fstream(int fh, char *p, int l);
Permite construir un stream con buffer que se liga a fh. p apunta a un buffer de l byte de
longitud. Si p==NULL o l==O el stream no utilizará buffer.

setbuf(char *p, int l);
Permite cambiar el buffer y la longitud. Si p==NULL o l==O el stream pasará a no tener
buffer.

istream& seekg(long offset, seek_dir mode= ios::beg);
ostream& seekp(long offset, seek_dir mode= ios::beg);
long tellg();
long tellp();
istream& read(signed char *s, int nbytes);
istream& read(unsigned char *s, int nbytes);
istream& read(void *p, int nbytes);
ostream& write(const signed char *s, int nbytes);
ostream& write(const unsigned char *s, int nbytes);
ostream& write(void p, int nbytes);
```

PROGRAMACIÓN EN C++

A continuación daremos unas nociones sobre lo que debe ser la programación en C++. Estas ideas no son las únicas aceptables, sólo pretendo que os sirvan como referencia inicial hasta que encontréis lo que más se acomode a vuestra forma de trabajar. Hay una frase que leí una vez que resume esto último: "Los estándares son buenos, cada uno debería tener el suyo".

El proceso de desarrollo

Dentro de la metodología de programación clásica se definen una serie de fases en el proceso de desarrollo de aplicaciones. No es mi intención repetirlas ahora, sólo quiero indicar que las nuevas metodologías de programación orientadas a objetos han modificado la forma de trabajar en estas etapas. El llamado ciclo de vida del software exigía una serie de etapas a la hora de corregir o modificar los programas, trabajando sobre todas las etapas del proceso de desarrollo. A mi modo de ver estas etapas siguen existiendo de una manera u otra, pero el trabajo sobre el análisis y diseño (que antes eran textos y diagramas, no código) es ahora posible realizarlo sobre la codificación: la idea de clase y objeto obliga a que los programas tengan una estructura muy similar a la descrita en las fases de análisis y diseño, por lo que un código bien documentado junto con herramientas que trabajan sobre el código (el browser, por ejemplo, nos muestra la estructura de las jerarquías de clases de nuestro programa) puede considerarse un modelo del análisis y el diseño (sobre todo del diseño, pero las clases nos dan idea del tipo de análisis realizado).

Mantenibilidad y documentación

Para mantener programas en C++ de forma adecuada debemos tener varias cosas en cuenta mientras programamos: es imprescindible un análisis y un diseño antes de implementar las clases, y todo este trabajo debe estar reflejado en la documentación del programa. Además, la estructura de clases nos permite una prueba de código mucho más fácil, podemos verificar clase a clase y método a método sin que ello afecte al resto del programa.

Debemos documentar el código abundantemente, es decir, debemos comentar todo lo que podamos los programas, explicando el sentido de las variables y objetos o la forma de implementar determinados algoritmos.

Diseño e implementación

Debemos diseñar las aplicaciones en una serie de niveles diferentes: diseño de gestión de la información, diseño de la interface de la aplicación, etc.

A la hora de hacer programas es importante separar la parte de interface con el usuario de la parte realmente computacional de nuestra aplicación. Todo lo que hagamos a nivel de gestión de ficheros y datos debe ser lo más independiente posible de la interface de usuario en la que trabajamos. El hacer así las cosas nos permite realizar clases reutilizables y transportables a distintos entornos. Una vez tenemos bien definidas las clases de forma independiente de la interface con el usuario podemos definir esta e integrar una cosa y otra de la forma más simple y elegante posible.

Esta separación nos permitirá diseñar programas para SO con interface textual y transportarla a SO con ventanas y menús sin cambios en la funcionalidad de la aplicación.

Elección de clases

Un buen diseño en C++ (o en cualquier lenguaje orientado a objetos), pasa por un buen análisis de las clases que deben crearse para resolver nuestro programa. Una idea para identificar que deben ser clases, que deben ser objetos, que deben ser atributos o métodos de una clase y como deben relacionarse unas clases con otras es estudiar una descripción textual del problema a resolver. Por norma general los conceptos abstractos representarán clases, las características de estos conceptos (nombres) serán atributos, y las acciones (verbos) serán métodos. Las características que se refieran a más de un concepto nos definirán de alguna manera las relaciones de parentesco entre las clases y los conceptos relativos a casos concretos definirán los objetos de necesitamos.

Todo esto es muy vago, existen metodologías que pretenden ser sistemáticas a la hora de elegir clases y definir sus miembros, pero yo no acabo de ver claro como se pueden aplicar a casos concretos. Quizás la elección de clases y jerarquías tenga un poco de intuitivo. De cualquier forma, es fácil ver como definir las clases una vez tenemos un primer modelo de las clases para tratar un problema e intentamos bosquejar que tipo de flujo de control necesitamos para resolverlo.

Interfaces e implementación

Cuando definamos las clases de nuestra aplicación debemos intentar separar muy bien lo que es la interface externa de nuestra clase (la declaración de su parte protegida y pública) de la implementación de la clase (declaración de miembros privados y definición de métodos). Incluyo en la parte de implementación los miembros privados porque estos sólo son importantes para los métodos y funciones amigas de la clase, no para los usuarios de la clase. La correcta separación entre una cosa y otra permite que nuestra clase sea fácil de usar, de modificar y de transportar.

LIBRERÍAS DE CLASES

Como hemos visto, el C++ nos permite crear clases y jerarquías de clases reutilizables, es decir, las clases que definimos para programas concretos pueden utilizarse en otros programas si la definición es lo suficientemente general. Lo cierto es que existen una serie de clases que se pueden reutilizar siempre: las clases que definen aspectos de la interface con el SO (ventanas, menús, gestión de eventos, etc.) y las clases contenedor (pilas, colas, árboles, etc.). Existe otra serie de clases que pueden reutilizarse en aplicaciones concretas (clases para definir figuras geométricas en 2 y 3 dimensiones para aplicaciones de dibujo, clases para gestionar documentos de texto con formato en editores de texto, etc.).

En este bloque comentaremos algunas cosas a tener en cuenta a la hora de diseñar y trabajar con bibliotecas de clases.

Diseño de librerías

Lo primero que debemos plantearnos a la hora de diseñar una biblioteca de clases es si es necesario hacerlo. Por ejemplo, si queremos diseñar una biblioteca de clases de uso general para la gestión del SO, lo más normal es que estemos perdiendo el tiempo, ya que deben existir varias bibliotecas comerciales que hagan lo mismo con la ventaja de que deben estar probadas y lo único que nosotros debemos hacer es aprender a manejarlas.

Si nos decidimos a utilizar una biblioteca comercial lo más importante es saber cual es el soporte que esta biblioteca tiene, es decir, saber si la biblioteca tiene un futuro y si ese futuro pasa por la compatibilidad. Es habitual que las compañías que comercializan una biblioteca de clases vayan actualizando y mejorando sus clases, sacando al mercado sucesivas versiones de la misma. Lo más importante en estos casos es que las nuevas versiones añadan cosas o mejoren implementaciones, pero no modifiquen las interfaces de las clases antiguas, ya que esto puede hacer que nuestros viejos programas tengan que reescribirse para cada versión de una biblioteca. De todas formas, tenemos pocas garantías de que una compañía mantenga la compatibilidad en una biblioteca de software, aunque conforme vaya pasando el tiempo el mercado generará unos estándares que todo el mundo empleará.

Si la biblioteca que queremos escribir no existe (o las disponibles son malas), en primer lugar deberemos saber cuál es el alcance y potencia que queremos que tenga. Si lo que nos interesa es una biblioteca para uso personal podremos definirla a nuestro aire, pero es difícil que le sirva de mucho a otras personas.

Si por el contrario queremos que tenga un uso relativamente amplio (que la usen varias personas o grupos de personas) tendremos que comenzar pensando que el diseño debe documentarse y razonarse, definiendo las jerarquías y clases de la forma más simple y flexible posible. Es decir, tenemos que identificar que posibles clases se pueden definir en el ámbito que trata la biblioteca y definir las relaciones entre ellas. Debemos intentar que la biblioteca tenga el número mínimo de clases posibles de manera que sus declaraciones (su estructura externa, es decir, su interface con el usuario) sean fáciles de comprender y de ampliar (mediante herencia).

También es muy importante intentar que hagan el menor uso posible de las facilidades no estándar del C++, es decir, que no intenten aprovechar una arquitectura o un sistema operativo concretos. El aprovechamiento de estos recursos siempre se puede incorporar después en la implementación, pero el diseño pretende ser lo más general posible para que las implementaciones se puedan transportar de unas máquinas a otras. Sería una buena idea implementar una biblioteca sin optimizaciones (algoritmos sencillos, independencia de la máquina, poca gestión de memoria, etc.) y guardarla como primera versión. A partir de esta biblioteca iremos refinando (y documentando los refinamientos) para llegar a una versión definitiva probada y eficiente. Si hemos documentado todos los pasos de nuestro diseño e implementación el manejo de nuestra biblioteca será rápido de aprender y las modificaciones sencillas.

Otra cosa importante es considerar los errores que se pueden producir al usar las clases, para dotarlas de una gestión de errores adecuada.

Se que todas estas indicaciones son fáciles de dar pero difíciles de llevar a la práctica, lo fundamental es saber para qué estamos programando, si es para luego emplear la biblioteca muy a menudo es preferible trabajarla bien al principio en el diseño (comprobando que es el adecuado para lo que nos proponemos) y la implementación (comprobando la corrección y robustez del código), al final nos ahorrará tiempo.

Clases Contenedor

Un tipo de clases muy empleado es el de las clases contenedor. En la actualidad, la mayoría de compiladores incorporan las templates y una biblioteca específica de clases contenedor genéricas. Lo importante al emplear estas clases es tener en cuenta que cosas debemos de incorporar a nuestras clases para que trabajen adecuadamente con ellas. Es muy habitual que sea

necesario definir alguna relación de orden en nuestras clases (sobrecarga de operadores relacionales).

Si queremos diseñar clases contenedor deberemos tener en cuenta que clase de objetos han de contener, he intentar que dependan lo menos posible de ellos. Lo habitual es que sólo necesitemos relaciones de orden (para árboles ordenados, por ejemplo) y de igualdad (para comprobar si un objeto está dentro de un contenedor). En la actualidad lo más razonable es emplear plantillas para definir este tipo de clases (cuando estas no existían se trabajaba con contenedores de punteros a objetos). También es importante considerar que es lo que queremos almacenar: objetos, punteros a objetos o referencias a objetos.

También hay que considerar si estas clases deben pertenecer o no a una jerarquía (es decir, si las queremos definir como objetos relacionados con los demás o sólo como almacenes de datos).

Por último hay que saber que existen muchas posibilidades alternativas para la gestión e implementación de contenedores, con diferente niveles de eficiencia temporal y espacial. La idea es intentar llegar a un compromiso entre ambas cosas, pero también puede ser útil definir varias alternativas para una misma clase y emplear en cada caso la que más convenga al programa por velocidad y espacio que necesita.

Clases para aplicaciones

En la actualidad existen bibliotecas comerciales que nos permiten escribir programas completos en entornos complejos como Windows, en pocas líneas. Esto se consigue gracias a la idea de clases de aplicación y de interacción con el SO. Si nosotros tenemos que escribir programas de gestión de bases de datos, por ejemplo, sabemos que lo habitual es que todos los programas tengan la misma estructura interna (implementación) y externa (interfaz con el usuario de la aplicación). Pues bien, hay bibliotecas que hacen uso de ese hecho y definen una jerarquía que permite controlar y definir estos objetos comunes, para implementar la aplicación bastará con usar objetos de las clases de la biblioteca y quizás definir unas pocas clases derivadas redefiniendo algunos métodos.

Yo creo que no falta mucho para que aparezca una o varias bibliotecas estándar para la construcción de determinados tipos de aplicaciones. Puede que así desaparezcan algunos lenguajes específicos bastante desafortunados (pero desgraciadamente muy extendidos).

Clases de Interface

Dentro de las clases para aplicación lo que si han aparecido son bibliotecas para definir la interface de las aplicaciones, pero la mayoría de estas bibliotecas son dependientes del Sistema Operativo, por lo que no son realmente estándar. Lo ideal sería que se definiera una interface gráfica estándar orientada a objetos (como las que de alguna manera definen los lenguajes SmallTalk y Oberon, este último implementado con un pequeño sistema operativo propio).

Hasta que no existan estas bibliotecas estándar tendremos que seguir estudiando bibliotecas distintas para cada sistema operativo y cada compilador concreto.

No es mala idea crearse una pequeña biblioteca de interface para nuestras aplicaciones de entorno textual, ya que estas suelen ser muy transportables.

Eficiencia temporal y gestión de memoria

A la hora de diseñar o utilizar una biblioteca son fundamentales dos cosas, la eficiencia temporal de las operaciones con los objetos de las clases y la gestión de memoria que se haga. El C++ es un lenguaje que pretende ser muy eficaz en estos aspectos, por lo que las bibliotecas deberían aprovechar al máximo las posibilidades disponibles.

Estandarización

En la actualidad se está refinando y ampliando el estándar del C++. En el último borrador del comité (28 Abril de 1995) se incorporan una serie de mejoras del lenguaje como la defínción de espacios de nombres (asignar un ámbito a los identificadores para evitar conflictos en las biliotecas) o de operadores de conversión (casts) mucho más refinados.

Además, la bibloteca de clases está muy ampliada (en realidad sólo hemos visto la biblioteca de E/S, pero es que no había nada más estandarizado). El borrador divide la biblioteca en diez componentes:

1. *Soporte al lenguaje*: declara y define tipos y funciones que son usadas implicitamente por los programas escritos en C++.

2. *Diagnósticos*:define componentes que puden ser usados para detectar e informar de errores.

3. *Utilidades generales*: componentes usados por otros componentes de la biblioteca y que también se pueden usar en nuestros programas.

4. *Cadenas(strings)*: Componentes para manipular secuencias de caracteres (los caracteres pueden ser tipos definidos por el usuario o char y w_char, que es un nuevo tipo de la biblioteca).

5. *Localización*: Componentes para soporte internacional, incluye facilidades para la gestión de formatos de fecha, unidades monetarias, orden de los caractéres, etc.

6. *Contenedores*: Componentes que se pueden emplear para manejar colecciones de información.

7. *Iteradores*: Componentes que los programas pueden emplear para recorrer contenedores, streams(E/S) y stream buffers (E/S).

8. *Algoritmos*:Componentes para realizar operaciones algorítmicas sobre contenedores y otras secuencias.

9. *Numéricos*:Componentes que se pueden emplear para realizar operaciones semi-numéricas. Define los complejos, operaciones con matrices y vectores, etc.

10. *E/S*:Componentes para realizar tareas de entrada/salida.

De momento parece que no se va a incorporar la E/S gráfica, pero es relativamente lógico, ya que el diseño de una biblioteca de ese tipo podría limitar mucho a la hora de aprovechar las capacidades de un SO concreto.

RELACIÓN C/C++

No se puede usar en ANSI C

(1) Las clases y la programación orientada o objetos. Esta es la faceta más importante del C++; simularla en C es posible pero bastante complicada

(2) Los templates. En C se puede hacer algo parecido usando macros

(3) El tratamiento de errores: `try`, `catch`, `throw`. Intentar simular esto es muy complicado

(4) La sobrecarga y el "name-mangling". En C dos funciones con el mismo nombre deben tener el mismo tipo de parámetros. La única solución es usar macros o simplemente dar nombres diferentes a las funciones

(5) La sobrecarga de operadores

(6) Las funciones `inline`. En C se solían usar macros para hacer operaciones eficientes

(7) Los operadores `new` y `delete`. La gestión de memoria en C se hacía con funciones

(8) Los parámetros por defecto

(9) Los comentarios estilo C++: `//`

(10) Los casts estilo C++: `cast ()`

(11) El uso de `asm`

(12) El operador de campo: `::`

(13) Las uniones y enumeraciones anónimas

(14) Las referencias y por ello, el pase de parámetros por referencia. Para pasar parámetros por referencia en C se usan punteros

(15) Flexibilidad de declaraciones. En C las variables se deben declarar y definir al principio del bloque actual. En C++ se pueden definir en cualquier parte

Diferencias entre C y C++

(1) Compatibilidad de tipos En ANSI C un puntero `void` es compatible con todos los punteros. En C++ es válido asignarle a un puntero no `void` un puntero `void` pero no lo contrario.

(2) Flexibilidad de constantes: Las constantes en C++ pueden ser usadas en cualquier lugar. En ANSI C en cambio lo siguiente seria incorrecto:

```
const int Len= 1000;
int Vector[Len];
```

En C++ sería perfectamente correcto.

(3) La longitud (`sizeof`) de un carácter en C es 2 o 4 como un entero (dependiendo del tamaño de palabra). En C++ es siempre 1.

(4) En C se necesita poner la palabra `struct` o `union` delante de cualquier estructura o unión que se haya definido:

```
struct point {
  int x, y;
};
struct point p; // Obligatoria en C, opcional en C++
```

(5) En C++ está prohibido saltar sobre una declaración En C si que se puede saltar

(6) En C++ una declaración de variable sin `extern` es considera siempre la definición de una variable. En cambio en ANSI C es considerado una tentativa y se pueden definir varias veces una variable en un fichero y será convertida a una sola definición

(7) En C++ el enlazado por defecto de una constante es `static`. En ANSI C es `extern`

(8) C y C++ interpretan la lista vacía de argumentos de forma diferente. En C la lista vacía suspende el chequeo de tipos. Así `f()` en C puede tomar cero o más argumentos de cualquier tipo. En C++ la lista vacía significa que no tiene parámetros. En C se utilizaría el parámetro `void` (`f(void)`)

(9) En C está permitido el uso de una función no declarada, que se toma como una función si chequeo de tipos. En C++ toda función tiene que estar declarada

(10) En C++ no se puede hacer un `return` sin valor para una función que devuelva algo.

(11) C++ y ANSI C interpretan de forma diferente el tipo de cadena constante usada para inicializar un vector de caracteres. En C++ lo siguiente es ilegal mientras que en C está permitido:

```
char vocales[5] ="aeiou";
```

En ANSI C esto es interpretado como la asignación de cada uno de los componente del vector de vocales. sin tener en cuenta el cero final.