Escuela Técnica Superior de Ingeniería y Diseño Industrial

POLITÉCNICA

"Ingeniamos el futuro"

C++ estándar

Apuntes de Informática Industrial y Comunicaciones.

Miguel Hernando Gutiérrez

Madrid, Febrero de 2020

C++ estándar

Apuntes de Informática Industrial y Comunicaciones.

MIGUEL HERNANDO GUTIÉRREZ

Departamento de Electrónica, Automática e Informática Industrial

Escuela Técnica Superior de Ingeniería y Diseño Industrial

UNIVERSIDAD POLITÉCNICA DE MADRID

MADRID, FEBRERO DE 2020

"El único modo de hacer un gran trabajo es amar lo que haces.
Si no lo has encontrado todavía, sigue buscando...
No te acomodes.
Como con todo lo que es propio del corazón,
lo sabrás cuando lo encuentres."

Steve Jobs.

Presentación

La asignatura de informática Industrial y comunicaciones se compone de dos partes algo diferenciadas como el propio nombre de la asignatura indica. Por un lado se pretende que el estudiante de esta asignatura comience a programar según una filosofía de programación orientada a objetos (POO), para lo cual se ha escogido el paradigma de este tipo de lenguajes que es C++. Por otro, se pretende informar y formar al estudiante en la programación específica de sistemas de comunicación, para lo cual es necesario introducir conceptos relativos a los sistemas distribuidos y las redes.

Estos apuntes pretenden de forma docente exponer el lenguaje de programación C++ pero centrándose especialmente en los mecanismos que este dispone para abordar la programación orientada a objetos. De esta forma, aunque se explican muchas de las posibilidades adicionales del lenguaje, se centran en explicar con detalle el modo con el que C++ pone a nuestra disposición el encapsulamiento, la herencia y el polimorfismo.

Por este motivo, aspectos como las excepciones o las plantillas son explicados al final de la parte relativa a la exposición de la sintaxis del lenguaje y en menor detalle, siendo para la asignatura un objetivo secundario.

Estos apuntes no tienen ningún interés comercial, sino puramente docente. Para su elaboración se han tomando muchos ejemplos e incluso explicaciones textuales de apuntes o cursos tanto escritos en papel como presentes en Internet. Salvo por error, se han ido incluyendo las referencias a estas fuentes. El orden y modo de exposición se ha pensado de forma que sean una herramienta eficiente de cara a explicar o seguir un curso presencial .

Por último, cualquier errata detectada por el lector o sugerencia constructiva que se considere oportuna agradecería que me fuera transmitida por correo electrónico a miguel.hernando@ upm.es.

Prof. Miguel Hernando

Índice de contenidos

1. Introducción a C++

Para poder comprender correctamente estos apuntes de programación en C++, se considera que el lector ya tiene unos conocimientos previos del lenguaje C. A la hora de diseñar el temario de las asignaturas de informática de la Escuela Técnica de Ingenieros Industriales, se planteó como objetivo principal el que el alumno fuera capaz de programar en cualquier lenguaje al finalizar los estudios. Para ello era necesario seleccionar un lenguaje modelo que permitiera plantear los distintos inconvenientes y técnicas que se dan en la programación. Por este motivo se seleccionó el lenguaje C en las asignaturas de primer año como paradigma de lenguaje procedural y como sistema de acercamiento al modo de funcionamiento de un ordenador, y C++ como el lenguaje más representativo de la programación orientada a objetos (POO). Es importante considerar que el Ingeniero Industrial Electrónico, tiene una clara orientación al dispositivo físico, de ahí que los lenguajes que mantienen la cercanía a los elementos constitutivos del ordenador sean más relevantes para su formación, que otros que aun dando una funcionalidad mayor, se han alejado del contacto directo con el Hardware o el sistema operativo. C a menudo es considerado un lenguaje de medio nivel por esta característica.

Sin embargo, es importante reflejar el hecho de que esta asignatura, Informática Industrial, ha sido dividida a su vez en dos partes: una centrada en el proceso de diseño o ingeniería del software, y otra con el enfoque esencialmente práctico de aprender un lenguaje con filosofía de POO como es el caso de C++. Como se verá a continuación, el proceso de desarrollo de un programa en un lenguaje fuertemente dependiente de la estructura como es el caso de C++, requiere de un proceso de análisis y de diseño teórico previo mucho más importante que en la programación procedural.

La programación orientada a objetos es una de las más modernas técnicas de programación buscando como principal objetivo la reducción del tiempo de desarrollo aumentando la eficiencia del proceso de generación de los programas. Como consecuencia, si el diseño previo es correcto, en la POO los programas tienen menos líneas de código escritas por el programador, menos bifurcaciones, y sobre todo la facilidad de introducir elementos de programas previos o escritos por otras personas, así como su actualización.

Sin embargo, para lograr estos resultados es necesario un esfuerzo del programador en las fases anteriores a la escritura del programa propiamente dicho. Si así no fuera, los resultados pueden ser francamente decepcionantes. Así, para no llevar a engaño, y con la idea de amenizar ligeramente unos apuntes que de por sí prometen ser densos, se incluye a continuación una entrevista -ficticia- que durante un tiempo circuló por los foros de programación:

> **Entrevista al padre del C++**
>
> *El 1 de Enero de 1998, Bjarne Stroustrup, padre del lenguaje C++, dio una entrevista a la revista de informática del IEEE. Naturalmente, los editores pensaron que estaba dando una visión retrospectiva de los siete años de diseño orientado a objetos, usando el lenguaje que él mismo había creado.*
>
> *Al finalizar la entrevista, el entrevistador consiguió más de lo que había pactado en un principio, y consecuentemente, el editor decidió suprimir los contenidos 'por el bien de la industria'. Pero como suele suceder, la información se filtró... Aquí está una completa transcripción de lo que se dijo, no editado, no ensayado, es decir que no es como las entrevistas planeadas... Lo encontraréis interesante...*
>
> **Int:** Bien, hace unos pocos años que cambió el mundo del diseño de software, ¿cómo se siente mirando atrás?
>
> **BS:** En este momento estaba pensando en aquella época, justo antes de que llegase. ¿La recuerdas? Todo el mundo escribía en C y el problema era que eran demasiado buenos... Las Universidades eran demasiado buenas enseñándolo también. Se estaban graduando programadores competentes a una velocidad de vértigo. Esa era la causa del problema.
>
> **Int:** ¿Problema?
>
> **BS:** Sí, problema. ¿Recuerdas cuando todos programaban en Cobol?
>
> **Int:** Desde luego. Yo también lo hice.

BS: Bien, al principio, esos tipos eran como semidioses. Sus salarios eran altos, y eran tratados como la realeza...

Int: Aquellos fueron buenos tiempos, ¿eh?

BS: Exacto. Pero, ¿que pasó? IBM se cansó de ello, e invirtió millones en entrenar a programadores, hasta el punto que podías comprar una docena por medio dólar...

Int: Eso es por lo que me fui. Los salarios bajaron en un año hasta el punto de que el trabajo de periodista esta mejor pagado.

BS: Exactamente. Bien, lo mismo pasó con los programadores de C...

Int: Ya veo, pero ¿a donde quiere llegar?

BS: Bien, un día, mientras estaba sentado en la oficina, pensaba en este pequeño esquema, que podría inclinar la balanza un poquito. Pensé '¿Qué ocurriría si existiese un lenguaje tan complicado, tan difícil de aprender, que nadie fuese capaz de inundar el mercado de programadores?' Empecé cogiendo varias ideas del X10, ya sabes, X Windows. Es una autentica pesadilla de sistemas gráficos, que sólo se ejecutaba en aquellas cosas Sun 3/60... tenía todos los ingredientes que yo buscaba. Una sintaxis ridículamente compleja, funciones oscuras y estructuras pseudo-OO. Incluso ahora nadie escribe en código nativo para las X-Windows. Motif es el único camino a seguir si quieres mantener la cordura.

Int: ¿Está bromeando?

BS: Ni un pelo. De hecho, existe otro problema... Unix está escrito en C, lo que significa que un programador en C puede convertirse fácilmente en un programador de sistemas. ¿Recuerdas el dinero que un programador de sistemas solía conseguir?

Int: Puede apostar por ello. Es lo que solía hacer yo...

BS: Ok, por lo tanto, este nuevo lenguaje tenía que divorciarse por sí mismo de Unix, ocultando las llamadas al sistema. Esto podría permitir a tipos que sólo conocían el DOS ganarse la vida decentemente...

Int: No me puedo creer que haya dicho eso...

BS: Bueno, ha llovido mucho desde entonces. Ahora creo que la mayoría de la gente se habrá figurado que C++ es una pérdida de tiempo, pero debo decir que han tardado más en darse cuenta de lo que pensaba.

Int: Entonces, ¿qué hizo exactamente?

BS: Se suponía que tenía que ser una broma, nunca pensé que la gente se tomase el libro en serio. Cualquiera con dos dedos de frente puede ver que la programación orientada a objetos es anti intuitiva, ilógica e ineficiente...

Int: ¡¿¡¿Qué?!?!

BS: Y como el código reutilizable... ¿cuándo has oído de una compañía que reutilice su código?

Int: Bien, nunca, pero...

BS: Entonces estás de acuerdo. Recuerda, algunos lo intentaron al principio. Había esa compañía de Oregon, creo que se llamaba Mentor Graphics, que reventó intentando reescribir todo en C++ en el 90 o 91. Lo siento realmente por ellos, pero pensé que los demás aprenderían de sus errores.

Int: Obviamente no lo hicieron, ¿verdad?

BS: Ni lo más mínimo. El problema es que la mayoría de las empresas se callaron sus mayores disparates, y explicar 30 millones de dólares de perdidas a los accionistas podría haber sido dificil... Démosles el reconocimiento que merecen, finalmente consiguieron hacer que funcionase

Int: ¿Lo hicieron? Bien eso demuestra que la OO funciona...

BS: Casi. El ejecutable era tan gigantesco que tardaba unos cinco minutos en cargar en una estación de trabajo de HP con 128 MB de RAM. Iba tan rápido como un triciclo. Creí que sería un escollo insalvable pero nadie se preocupo. SUN y HP estaban demasiado alegres de vender enormes y poderosas maquinas con gigantescos recursos para ejecutar programas triviales. Ya sabes, cuando hicimos nuestro primer compilador de C++, en AT&T, compile el clasico 'Hello World', y no me podía creer el tamaño del ejecutable. 2.1 MB.

Int: ¡¿¡¿ Qué ?!?!. Bueno, los compiladores han mejorado mucho desde entonces...

BS: ¿Lo han hecho? Inténtalo en la última versión de C++, la diferencia no será mayor que medio mega. Además existen multitud de ejemplos actuales en todo el mundo. British Telecom tuvo un desastre mayor en sus manos, pero, afortunadamente, se deshicieron de ello y comenzaron de nuevo. Tuvieron más suerte que Australian Telecom. Ahora he oído que Siemens está construyendo un dinosaurio y se empiezan a preocupar porque los recursos hardware no hacen más que crecer para hacer funcionar ejecutables típicos. ¿No es una delicia la herencia múltiple?

Int: Bien, pero C++ es un lenguaje avanzado...

BS: ¡¿Realmente crees eso ?! Te has sentado alguna vez y te has puesto a trabajar en un proyecto C++? Esto es lo que sucede: Primero he puesto las suficientes trampas para asegurarme de que solo los proyectos más triviales funcionen a la primera. Coge la sobrecarga de operadores. Al final del proyecto casi todos los módulos lo tienen, normalmente los programadores sienten que deberían hacerlo así porque es como les enseñaron en sus cursos de aprendizaje. El mismo operador entonces significa cosas diferentes en cada modulo. Intenta poner unos cuantos juntos, cuando tengas unos cientos de módulos. Y para la ocultación de datos. Dios, a veces no puedo parar de reírme cuando oigo los problemas que algunas empresas han tenido al hacer a sus módulos comunicarse entre sí. Creo que el término 'sinergetico' fue especialmente creado para retorcer un cuchillo en las costillas del director de proyecto...

Int: Tengo que decir que me siento bastante pasmado por todo esto. Dice que consiguió subir el salario de los programadores? Eso es inmoral.

BS: No del todo. Cada uno tiene su opción. Yo no esperaba que la cosa se me fuese tanto de las manos. De cualquier forma acerté. C++ se esta muriendo ahora, pero los programadores todavía conservan sus sueldos altos. Especialmente esos pobres diablos que tienen que mantener toda esta majadería. Comprendes que es imposible mantener un gran modulo en C++ si no lo has escrito tu mismo?

Int: ¿Como?

BS: Estas fuera de juego, ¿verdad? Recuerdas 'typedef'?

Int: Si, desde luego.

BS: ¿Recuerdas cuanto tiempo se perdía buscando a tientas en las cabeceras sola para darse cuenta de que 'RoofRaised' era un numero de doble precisión? Bien, imagina el tiempo que te puedes tirar para encontrar todos los typedefs implícitos en todas las clases en un gran proyecto.

Int: ¿En que se basa para creer que ha tenido éxito?

BS: Te acuerdas de la duración media de un proyecto en C?. Unos 6 meses. No mucho para que un tipo con una mujer e hijos pueda conseguir un nivel de vida decente. Coge el mismo

proyecto, realízalo en C++ y ¿qué obtienes? Te lo diré. Uno o dos años. ¿No es grandioso? Mucha más seguridad laboral solo por un error de juicio. Y una cosa más. Las universidades no han estado enseñando C desde hace mucho tiempo, lo que produce un descenso del número de buenos programadores en C. Especialmente de los que saben acerca de la programación en sistemas Unix. Cuantos tipos sabrían que hacer con un 'malloc', cuando han estado usando 'new' durante estos años y nunca se han preocupado de chequear el código de retorno?. De hecho la mayoría de los programadores en C++ pasan de los codigos que les devuelven las funciones. ¿Que paso con el '-1'? Al menos sabías que tenías un error, sin enredarte con 'throw', 'catch', 'try'...

Int: Pero seguramente la herencia salve un montón de tiempo..

BS: ¿Lo hace? Te has fijado en la diferencia entre un proyecto en C y el mismo en C++? La etapa en la que se desarrolla un plan en un proyecto en C++ es tres veces superior. Precisamente para asegurarse de que todo lo que deba heredarse, lo hace, lo que no, no. Y aun así sigue dando fallos. Quien ha oído hablar de la pérdida de memoria en un programa en C? Ahora se ha creado una autentica industria especializada en encontrarlas. Muchas empresas se rinden y sacan el producto, sabiendo que pierde como un colador, simplemente para reducir el gasto de buscar todas esas fugas de memoria.

Int: Hay herramientas...

BS: La mayoría escritas en C++.

Int: Si publicamos esto, probablemente le lincharan. ¿Se da cuenta?

BS: Lo dudo. Como dije, C++ esta en su fase descendente ahora y ninguna compañía en su sano juicio comenzaría un proyecto en C++ sin una prueba piloto. Eso debería convencerles de que es un camino al desastre. Si no lo hace, entonces se merecen todo lo que les pase. Ya sabes?, yo intente convencer a Dennis Ritchie a reescribir Unix en C++...

Int: Oh Dios. ¿Que dijo?

BS: Afortunadamente tiene un buen sentido del humor. Creo que tanto él cómo Brian se figuraban lo que estaba haciendo en aquellos días, y nunca empezaron el proyecto. Me dijo que me ayudaría a escribir una versión en C++ de DOS, si estaba interesado...

Int: Lo estaba?

BS: De hecho ya he escrito DOS en C++, te pasare una demo cuando pueda. Lo tengo ejecutándose en una Sparc 20 en la sala de ordenadores. Va como un cohete en 4 CPUs, y solo ocupa 70 megas de disco...

Int: ¿Como se comporta en un PC?

BS: Ahora estas bromeando. No has visto Windows '95? Creo que es mi mayor éxito. Casi acaba con la partida antes de que estuviese preparado

Int: Ya sabes, la idea de Unix++ me ha hecho pensar. Quizás haya alguien ahí fuera intentándolo.

BS: No después de leer esta entrevista.

Int: Lo siento, pero no nos veo capaces de publicar esto.

BS: Pero es la historia del siglo. Solo quiero ser recordado por mis compañeros programadores, por lo que he hecho por ellos. ¿Sabes cuanto puede conseguir un programador de C++ hoy día?

Int: Lo ultimo que oí fue algo como unos $70 - $80 la hora para uno realmente bueno...

BS: ¿Lo ves? Y se los gana a pulso. Seguir la pista de todo lo que he puesto en C++ no es fácil. Y como dije anteriormente, todo programador en C++ se siente impulsado por alguna promesa mistica a usar todos los elementos del lenguaje en cada proyecto. Eso ciertamente me molesta a veces, aunque sirva a mi propósito original. Casi me ha acabado gustando el lenguaje tras todo este tiempo.

Int: ¿Quiere decir que no era así antes?.

BS: Lo odiaba. Parece extraño, ¿no estas de acuerdo? Pero cuando los beneficios del libro empezaron a llegar... bien, te haces una idea...

Int: Solo un minuto. ¿Que hay de las referencias?. Debe admitir que mejoro los punteros de C...

BS: Hmm... Siempre me he preguntado por eso. Originalmente creí que lo había hecho. Entonces, un día estaba discutiendo esto con un tipo que escribe en C++ desde el principio. Dijo que no podía recordar cuales de sus variables estaban o no referenciadas, por lo que siempre usaba punteros. Dijo que el pequeño asterisco se lo recordaba.

Int: Bien, llegados a este punto suelo decir 'muchas gracias' pero hoy no parece muy adecuado.

BS: Prométeme que publicaras esto. Mi conciencia esta dando lo mejor de mi mismo en estos momentos.

Int: Se lo haré saber, pero creo que se lo que dirá mi editor...

BS: ¿Quien se lo creería de todas formas?... De todos modos, ¿puedes enviarme una copia de la cinta?.

Int: Descuide, lo haré.

¿Sorprendido?. No deja de ser más que una de las gracias que circulan por la red de cuando en cuando. Sin embargo, sí que hay algo de verdad en la entrevista por lo que se anima a que al finalizar el curso el lector vuelva a leerla para que pueda comprender mejor los riesgos de una programación orientada a objetos mal entendida, o de las posibles consecuencias que el mal uso de las potentes herramientas de C++ pueden generar.

1.1. Historia de C++

El lenguaje C++ proviene como su propio nombre indica del lenguaje C. Este lenguaje ya estudiado en primer curso, nació en los laboratorios Bell de AT&T de las mentes de Kernighan y Ritchie en los 70. Su eficiencia y claridad, y la posibilidad de realizar tanto acciones de bajo como de alto nivel han hecho de este lenguaje el principal tanto en el mundo del desarrollo de sistemas operativos como de aplicaciones tanto industriales como de ofimática.

Con el tiempo y la experiencia el lenguaje fue evolucionando para dotarlo de mayor eficiencia desde el punto de vista del programador. En 1980 se añaden al lenguaje C características como *las clases* como resultado de la evolución de las estructuras, chequeo del tipo de los argumentos de una función y su conversión si es posible etc, dando como resultado lo que en ese momento se llamó *C con clases*[1]. Su autor fue Bjarne Stroustrup, ya mencionado en la introducción, y se desarrolló en la AT&T Bell Laboratories.

En 1983 este lenguaje fue rediseñado y comenzó a utilizarse fuera de ATT. Se introdujeron las funciones virtuales, la sobrecarga de funciones y operadores. Tras refinamientos y modificaciones menores, este nuevo lenguaje apareció documentado y listo para su uso bajo el nombre de C++.

Posteriormente C++ ha sido ampliamente revisado lo que ha dado lugar a añadir nuevas características como la herencia múltiple, las funciones miembro *static y const*, miembros *protected*, tipos de datos genéricos o plantillas, y la manipulación de excepciones. Se revisaron aspectos de la sobrecarga y manejo de memoria, se incremento la compatibilidad con C, etc. El éxito alcanzado por el lenguaje fue arrollador por lo que la ATT se vió en la necesidad de promover su estandarización internacional. En 1989 se convocó el comité de la ANSI[2] que más tarde entró a formar parte de la estandarización ISO. El trabajo conjunto de estos comités permitió publicar en 1998 el estándar ISO C++ (ISO/IEC 14882) de forma que el lenguaje pasó a ser estable y el código generado utilizable por distintos compiladores y en distintas plataformas.

Dentro de esta estandarización se incluye un conjunto de clases, algoritmos y plantillas que constituyen la librería estandar de C++[3]. Esta librería introduce facilidades para manejar las entradas y salidas del programa, para la gestión de listas, pilas, colas, vectores, para tareas de búsqueda y ordenación de elementos, para el manejo de operaciones matemáticas complejas, gestión de cadenas de caracteres, tipos de datos genéricos, etc. A este conjunto de algoritmos, contenedores y plantillas, se los denomina habitualmente por las siglas *STL (Standard Template Library).*

En el mundo de los lenguajes de programación, para realizar una referencia a los distintos estándares se sigue una nomenclatura que introduce el sufijo de la estandarización

[1] Del nombre inglés *C with classes.*

[2] American Nacional Standards Institute.

[3] En su traducción al castellano del término *library*, la palabra más correcta es *biblioteca*. Sin embargo la mayoría de los programadores la han traducido por *librería*, por lo que en estos apuntes se adoptará esta última.

al nombre del lenguaje. De esta forma, al estándar resultante de la norma ISO/IEC 14882:1998 publicada en 1998, se le denomina como C++98.

Posteriormente existe una nueva especificación denominada como C++03, como consecuencia de la aprobación de la revisión INCITS/ISO/IEC 14882:2003. En esta, no se modifica la sintaxis y aspecto del lenguaje pero si se especifican aspectos necesarios para el desarrollo de compiladores, y librerías para el estándar.

En donde si se produce una variación del lenguaje que los compiladores han comenzado a integrar es en el estándar C++11 ISO/IEC 14882:2011 . Aprobado y publicado en Agosto de 2011 como el nuevo estándar. En este caso si que se incluyen variaciones importantes en el lenguaje, pero conservando casi prácticamente la compatibilidad total con el estándar C++98.

A pesar de que uno de los principio de diseño de C++ siempre ha sido la modificación o ampliación de las librerías frente a el núcleo del lenguaje, en este estándar se hace un esfuerzo importante de actualización. C++11 incluye varias adiciones al núcleo del lenguaje y extiende la biblioteca estándar de C++, incorporando unas cuantas librerías como parte del estándar. Las áreas del núcleo del lenguaje que han sido significativamente mejoradas incluyen el soporte multithreading (multihilo), el soporte de programación genérica, la inicialización uniforme, y mejoras del rendimiento.

En el diseño de C++11-14-17 el comité encargado del mismo establece unos principios de diseño que sirven de guía. Estos son:

- Mantener compatibilidad hacia atrás (con C++98 y si es posible con C).
- Mejor extender la librería estandar que extender el núcleo del lenguaje.
- Tender a incluir mejoras que busquen la evolución de la técnica/modo de programar.
- Facilitar el diseño de sistemas y librerías (en vez de incluir especializaciones concretas para aplicaciones específicas).
- Mejorar la seguridad de los *tipos* dando alternativas seguras a los procedimiento inseguros heredados de versiones anteriores del lenguaje.
- Mejorar el rendimiento y la interacción con el hardware.
- Implementar el principio de zero-overhead ((soporte adicional requerido por algunas utilidades debe ser usado solo si la utilidad es usada).
- Hacer C++ más fácil de enseñar y aprender

En los siguientes apuntes nos centraremos en el estándar más extendido (C++98-C++03) aunque se anotarán algunas variaciones que se consideren relevantes introducidas por C++11. En el momento de escribir estas líneas se ha aprobado ya el estándar C++17, que

también incluye nuevas mejoras. Como puede observarse, desde la aparición de C++11 y como reacción a un declive claro del lenguaje se ha realizado un esfuerzo en actualizarlo y mejorarlo cada 3 años. De ahí que el siguiente estándar previsto sea C++20.

El resultado de todo ello es un potente lenguaje de programación que admite la programación procedural (como C) y la programación orientada a objetos. El resultado es mucho mejor que C, soportando la abstracción de datos, y la programación genérica gracias a las plantillas.

Los tres índices que más se utilizan para medir la popularidad de un lenguaje en la actualidad son el TIOBE Programming Community Index, el Languaje Popularity Index, y el PYPL, este último soportado por el análisis realizado por Google Trends.

En el siguiente gráfico se incluye una captura del índice TIOBE con fecha Enero de 2020. Este índice está pensado para decidir que lenguaje debe saber un programador a fecha actual, y se calcula no por el número de líneas de código sino más bien por el número de programadores y estudiantes de programación que en la actualidad siguen, programan o aprenden un lenguaje.

Se observa como el mundo de C, C++, C# y (cada vez en menore medida) Objective-C constituyen casi el 30% del índice. Si a eso añadimos que un programador de C++ es rápidamente convertible en programador de JAVA por su gran similitud en cuanto a muchos

Jan 2020	Jan 2019	Change	Programming Language	Ratings	Change
1	1		Java	16.896%	-0.01%
2	2		C	15.773%	+2.44%
3	3		Python	9.704%	+1.41%
4	4		C++	5.574%	-2.58%
5	7	^	C#	5.349%	+2.07%
6	5	v	Visual Basic .NET	5.287%	-1.17%
7	6	v	JavaScript	2.451%	-0.85%
8	8		PHP	2.405%	-0.28%
9	15	^^	Swift	1.795%	+0.61%
10	9	v	SQL	1.504%	-0.77%
11	18	^^	Ruby	1.063%	-0.03%
12	17	^^	Delphi/Object Pascal	0.997%	-0.10%
13	10	v	Objective-C	0.929%	-0.85%
14	16	^	Go	0.900%	-0.22%
15	14	v	Assembly language	0.877%	-0.32%

aspectos de la sintaxis y de la estructuración del programa, se puede deducir que en la actualidad, no se puede considerar seriamente un programador que no sepa al menos alguno de los lenguajes de la familia de C, y en concreto al menos uno bajo el paradigma de la POO.

Tabla 1. Indice TIOBE 2020 de los principales lenguajes de programación

La siguiente gráfica muestra la tendencia que estos lenguajes han seguido según este índice durante los últimos 10 años:

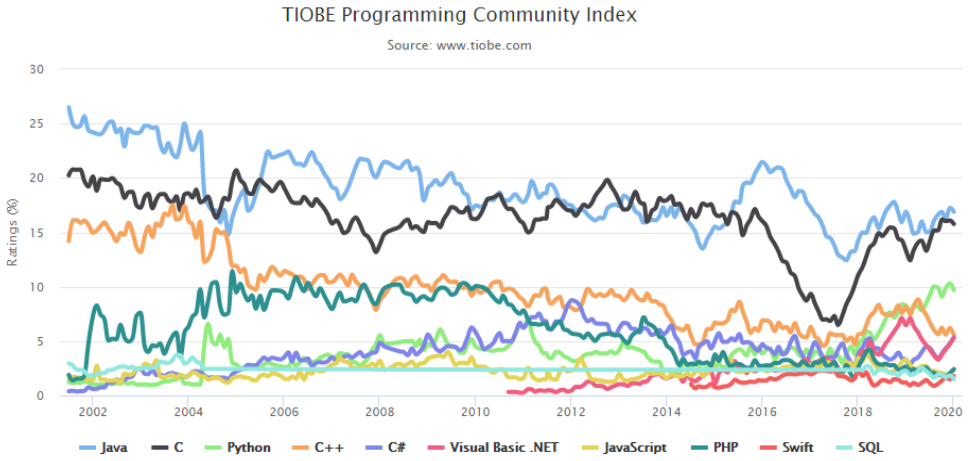
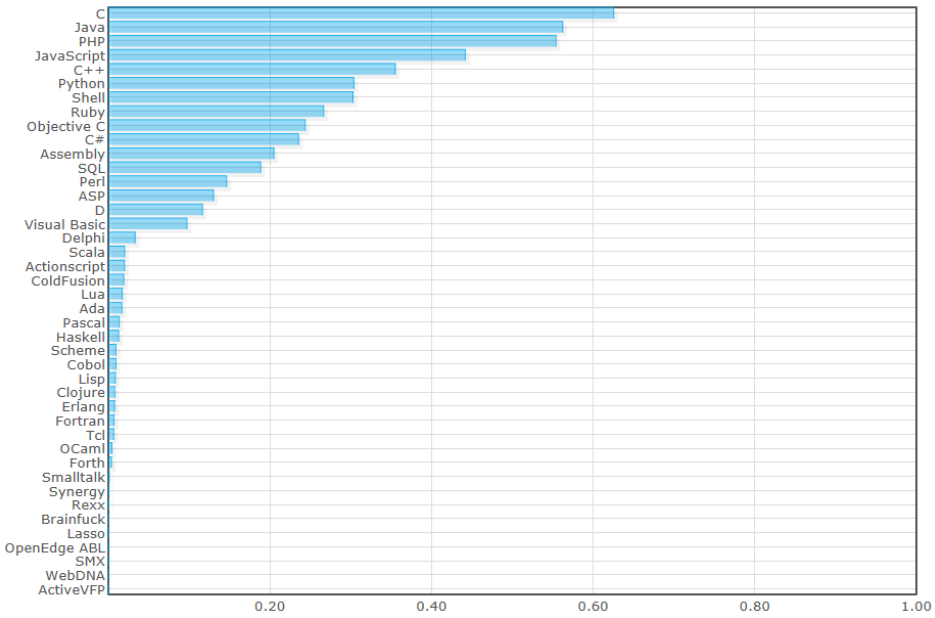
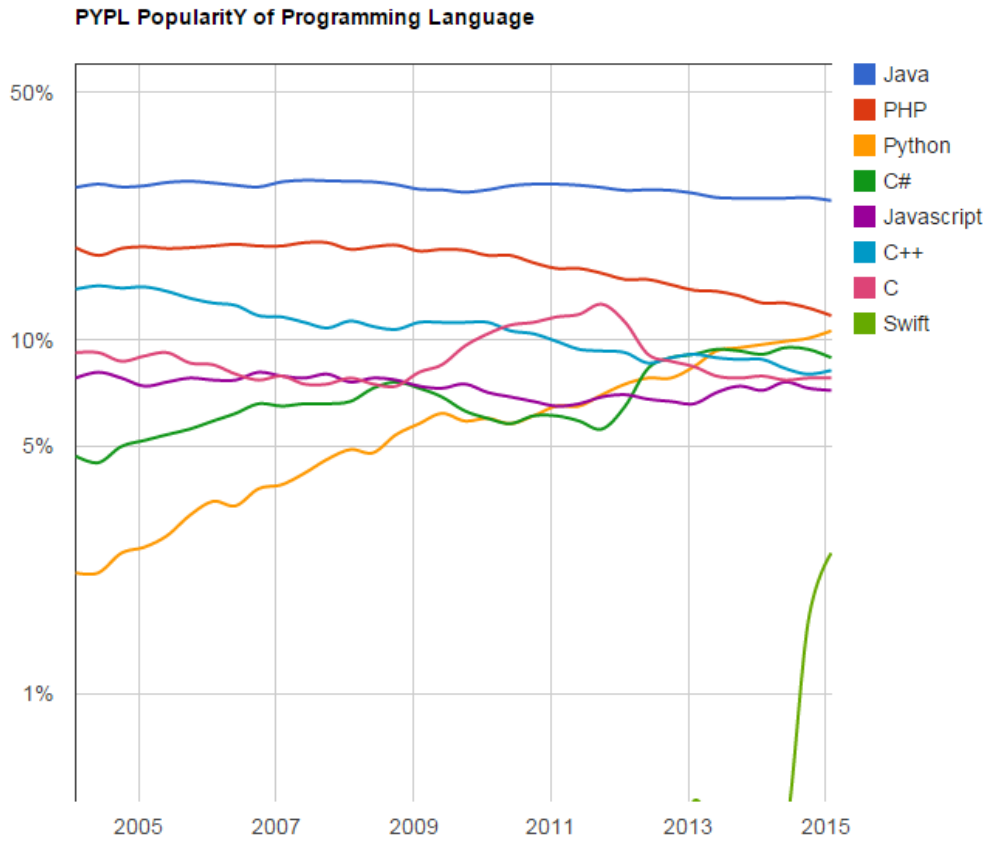


Tabla 2. Evolución del indice TIOBE 2020 de los principales lenguajes

De forma análoga se puede observar un gráfico que refleja la popularidad de los lenguajes según el Languaje Popularity Index combinando el número de repositorios públicos, discusiones, número de búsquedas etc. Unos índices detallados se pueden ver en http://langpop.com . El resumen que se realiza en dicha página viene a confirmar lo mismo:

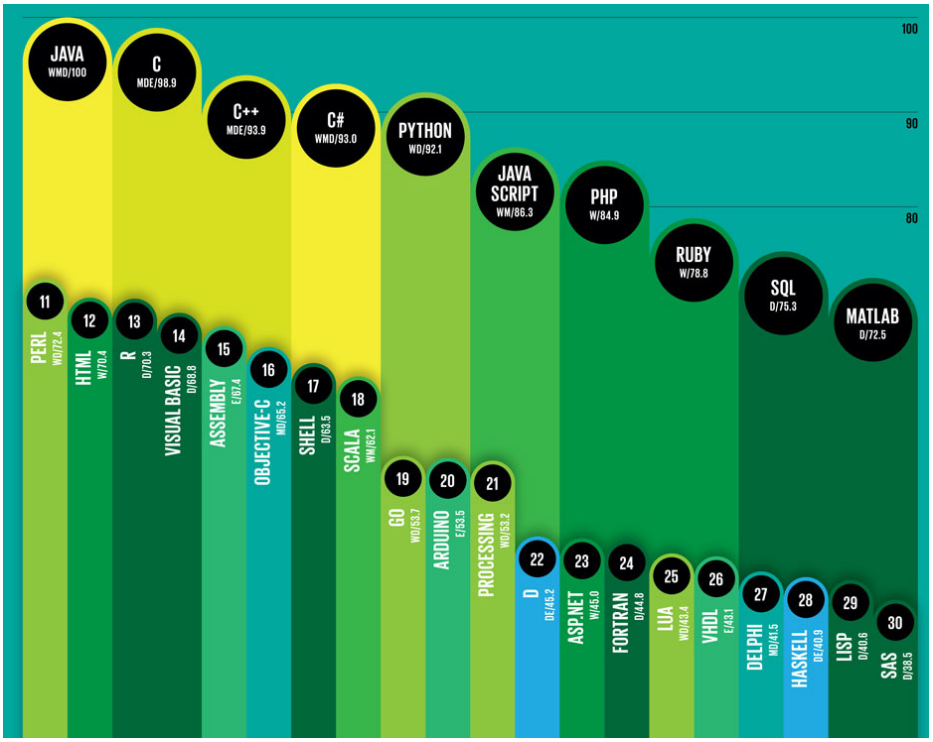


Estos datos discrepan ligeramente con el mostrado por PYPL. En cualquier caso, de nuevo, el trío C, C++, C# junto con Java son predominantes. La diferencia entre TIOBE y PYPL básicamente se debe a que TIOBE cuenta el número de paginas web que hablan del lenguaje (simplificando mucho) mientras que PYPL se centra en el número de visitas y búsquedas arrojadas por GoogleTrends. Un caso curioso es el de Objective-C que tiene unos 20 millones páginas en la web frente a los 11 millones de C, sin embargo las lecturas y búsquedas de C son actualmente 30 veces más en C que en Objective-C.



Si es cierto que durante la última época, se ha observado un crecimiento constante y aceptación de Python como lenguaje de programación pero aún es pronto para saber si realmente es un lenguaje consolidado, particularmente por el hecho de que varía áun excesivamente entre versión y versión y de que aún carece de una especificación estándar. Se observa que es un lenguaje muy usado para el prototipado rápido y para trabajar como sustitución de java por su característica multiplataforma y la facilidad de utilización de funcionalidades externas. De igual forma se observa un importante crecimiento de JavaScript, básicamente por convertirse en el lenguaje de los navegadores, permitiendo a una página web ejecutar auténticas aplicaciones en el entorno local de la máquina.

Se incluye finalmente el análisis realizado a finales de 2014 por el IEEE sobre la popularidad de los lenguajes combinando índices y además destacando las plataformas para las que fundamentalmente se utiliza el lenguaje (W: web; M: Móviles; D: Escritorio; E: Sistemas embebidos).



Finalmente, para dar por cerrada la introducción se reflejan algunos términos comúnmente adoptados, pero que facilitarán la estructura de la exposición. La parte de C incluida en C++ es conocida como C- y puede compilarse en C++ sin ningún problema. La sintaxis y las reglas de edición en general son iguales, por lo que a las modificaciones introducidas por C++ a la edición de código procedural (es decir, como si se tratase de C) se las denomina modificaciones menores ya que son fácilmente asumibles por un programador de C.

A las modificaciones introducidas para soportar la programación orientada a objetos y la programación genérica, se las denomina modificaciones mayores, puesto que requieren para su uso de un cambio radical en la filosofía de programación.

Para terminar este capítulo de introducción, se procederá a dar unas primeras pinceladas sobre la programación orientada a objetos que permitan un simple marco sobre el que ir vinculando los conceptos que se vayan explicando.

1.2. Lenguajes más relevantes en la actualidad

- **JAVA:** es un lenguaje basado en clases, orientado a objetos desarrollado por Sun Microsystems en 1990. Actualmente soportado por Oracle es uno de los lenguajes más demandados y se puede considerar un estándar en la empresa, en contenido

web en juegos y en aplicaciones de móviles. Basicamente ha tenido un especial impulso por utilizar Android un pseudojava como lenguaje nativo. Lo más importante de Java es que está diseñado para trabajar en multiples platarformas, de forma que no importa en que sistema operativo se ejecuta el programa.

- **C**: Es un leguaje de propósito general, procedural, desarrollado a comienzos de los 70, y se puede considerar el más antiguo y extensamente utilizado. En el fondo es la herramienta básica que se utiliza para construir otros lenguajes como es Java, C#, JavaScript o Python. Se utiliza extensamente para construir sistemas operativos y para sistemas embebidos en donde no existe competidor. Es el lenguaje de las máquinas. Dado que es la base de casi todos los lengajes es importante aprender C y C++ antes que moverse a otros más populares.
- **C++**: A C++ como veremos se lo considera un lenguaje intermedio con características de Orientación a Objetos. De hecho, como se verá, la razón de ser del lenguaje es impulsar C al paradigma de la programación orientada a objetos. Muchisimo software está construido en C++, en general todos los que requieran de una alta eficiencia. Se usa básicamente para software de sistemas, de aplicaciones, videojuegos y aplicaciones tanto de servidores como de clientes en particular las que requieran de alta potencia.
- **C#**: (c-sharp) es un lenguaje desarrollado por Microsoft que intenta combinar los principios de C, C++ y Java, y se usa básicamente en el desarrollo de programas para Windows. Su mayor inconveniente es precisamente el constituir un lenguaje asociado una plataforma.
- **Objetive-C** : es un lenguaje de propósito general orientado a objetos y utilizado principalmente por los sistemas operativos de Apple. Tanto OS X como iOS así como sus correspondientes APIs están en este lenguaje lo cual a provocado su moda en consonancia con el avance de iphone y los productos Apple. Es un lenguaje de la década de los 80 en contra de lo que mucha gente piensa, y podría decirse que es más primitivo en cuanto a sintaxis y concepto que C++. Es un superconjunto estricto de C por lo que es 100% compatible el código C con este.
- **PHP:** Su nombre proviene del acrónimo Procesador de Hypertexto. Es un lenguaje interpretado pensado para correr en el servidor, libre, que fue diseñado específicamente para el desarrollo de páginas web dinámicas (es decir generación de código html en base a unos contenidos cambiantes) y desarrollo de aplicaciones. Es un código directamente incluible en una página de código HTML, sin necesidad de residir en una serie de ficheros externos, lo cual lo ha convertido en especialmente popular como lenguaje de desarrollo de páginas web. Entre otros, es la base de Wordpress, Codeacademy o Facebook.

- **Python** es un lenguaje de alto nivel, interpretado, en principio pensado para trabajar en la parte del servidor de los sitios web o en aplicaciones de móviles. Se le considera un lenguaje sencillo para principiantes debido a su facil lectura y su sintaxis compacta, de tal forma que en muy pocas lineas se pueden hacer muchas cosas que en otros de los lenguajes anteriores conllevarían mucho más código. Actualmente destaca su uso en las webs y aplicaciones de Instagram, Pinterest, Biicode, y hay desarrollado un potente y conocido entorno de desarrollo web Django. Es utilizado por Google, Yahoo y la Nasa como parte de su código.
- **Ruby**. Es un lenguaje dinámico, interpretado, orientado a objetos, pensado para el desarrollo de páginas web dinámicas. Su diseño busca la sencillez y facilidad de escritura. Es la base del entorno Ruby on Rails, que se utiliza en Spotify, Github y Scribd. Al igual que Python, es un buen lenguaje para aquellos que comienzan.
- **JavaScript**. Es un lenguaje interpretado tanto para el cliente como para el servidor, desarrollado por Netscape y que deriva mucha de su sintaxis de C. En contra de lo que indica el nombre, no es Java y su desarrollo es paralelo. Puede usarse al igual que este ultimo para multiples plataformas por medio del intérprete del navegador, y se considera esencial para el desarrollo de la parte más interactiva y autónoma de las funciones web. Ultimamente ha sufrido un fuerte incremento dado que permite por la potencia de los navegadores el desarrollo de juegos que por su naturaleza funcionan tanto en ordenadores de escritorio como en dispositivos móviles. Está embebido en Chrome, en Safari, y en los intérpretes de Adobe

1.3. Primeras nociones sobre la Programación Orientada a Objetos.

Como ya se ha ido comentando, la programación orientada objetos (POO) no constituye un nuevo sistema de programación en cuanto a tecnología sino más bien una nueva filosofía de programación que posteriormente se ha visto reflejada en distintos lenguajes y de distintas formas.

Evidentemente la razón principal de la aparición de C++ es precisamente la adaptación de C a esta nueva forma de programación. El principal objetivo de la POO es lograr un código más reutilizable por lo que se mejora notablemente la eficiencia a la hora de programar.

Es importante destacar que la aparición de este modo de programación no responde a un diseño de laboratorio en el que un conjunto de cabezas pensantes se han dedicado a diseñar “el lenguaje de programación perfecto”, sino que proviene de la extracción de los

modos de programar que poco a poco se fueron estableciendo entre los programadores de los lenguajes estructurados para lograr sistemas más eficientes y robustos. Es decir, la mayoría de los mecanismos que se vana presentar responden a necesidades específicas de la programación práctica.

Antes de proceder a describir algunos de los términos básicos utilizados en cualquier lenguaje de programación con filosofía de POO se va a introducir un poco la idea de en qué consiste.

En los lenguajes estrictamente procedurales (C, PASCAL, BASIC) como se indica en su propia descripción, el elemento principal lo constituye el algoritmo o procedimiento. Es decir, de alguna manera, los datos son un elemento externo al programa y la función principal de este es manejarlos. Así de forma natural se tiende hacia un sistema centralizado en donde existe un motor principal dividido en distintos elementos, denominados funciones, que se encargan de ir manejando y modificando una serie de datos que van pasando de un lado a otro.

Así podríamos pensar en una librería en donde los datos corresponden a los libros, y el encargado de ordenarlos forrarlos, destruirlos, etc. es el librero con todos sus ayudantes. Es claro que ni el local, ni las estanterías, ni los libros y hojas son responsables de decidir en donde han de situarse ni de cuál es su precio etc. a pesar de que son los contenedores de la información que determina estas características. Será el librero el que al observarlo y examinarlo decidirá su ubicación, clasificación, precio, el forro, etc.

En la programación orientada a objetos el protagonismo pasa de residir en la algorítmica a los datos. Es decir, el elemento importante en el momento de programar lo constituyen las unidades de información. Cada una de estas unidades es capaz de realizar por sí misma las operaciones que se refieren precisamente a su ámbito de información. Así, hipotéticamente cada libro es capaz de identificar su género, su valía su precio, la cantidad de forro necesario para ser recubierto, el tiempo que cada tipo de lector tardará en leerlo, etc. A su vez estos libros se encuentran en estantes que son responsables de mantener los libros ordenados alfabéticamente, o de realizar operaciones de inserción y extracción de libros manteniendo el orden. Un conjunto de estanterías formarán un armario que es capaz de establecer la distribución de libros por estantes, etc. Finalmente el local decidirá el modo en que las librerías se disponen y los tipos de libros que albergará cada una de estas estanterías. Aunque no es más que un ejemplo mediocre, se muestra ahora alguna de las ventajas de este sistema.

Supongamos que ahora queremos añadir un estante a un armario. En el primer caso (programación procedural) habrá que reprogramar a todos los “empleados” y al “librero” para que reconozcan a este nuevo elemento. En el segundo caso bastará con agregar una balda “autoordenable” a uno de los armarios. Este armario informado de que tiene una

librería más será el único que tendrá que reconocerla, sin que el resto de sistemas se vea afectado. Simplemente cuando le lleguen los libros tendrá en cuenta esta nueva estantería. Añadir un nuevo armario al local requiere cambiar bastantes cosas en el código procedural, especialmente si este tipo de modificaciones no estaba previsto. En el caso de la POO será tan sencillo como crear una nueva variable y asignarle el tipo de libros que contendrá… el mismo armario se encargará de su organización. Supóngase que se quiere montar ahora una sucursal de la librería en la planta baja de "El Corte Inglés"… en el primer caso habrá que modificar el "código" de El Corte Inglés para albergar todos los algoritmos y estructuras de datos que mantienen los sistemas para la gestión de la librería. En POO bastará con agregar un objeto "local de librería" que ya se encarga de autogestionarse.

El ejemplo no da más de sí, pero la POO va mucho más lejos, puesto que si ahora en vez de tener una librería lo que se tiene es una frutería, la cantidad de código que hay que modificar es de nuevo mínima, puesto que muchas de las operaciones de organización son análogas aunque el elemento básico haya cambiado. Un frutero tendrá que clasificar su mercancía y cuidarla de forma parecida a un librero, y eso lo aprovecha la POO como se irá viendo.

Elementos básicos de la POO.

Algunos de los conceptos básicos que se manejarán a partir de ahora son los siguientes:

- **Objetos**. Un objeto es una entidad que tiene unos atributos particulares (datos) y unas formas de operar sobre ellos (los métodos o funciones miembro). Es decir, un objeto incluye, por una parte una serie de operaciones que definen su comportamiento, y una serie de variables manipuladas por esas funciones que definen su estado. Por ejemplo, una ventana Windows contendrá operaciones como "maximizar" y variables como "ancho" y "alto" de la ventana.
- **Clases**. Una clase es la definición de un tipo de objetos. De esta manera, una clase "Empleado" representaría todos los empleados de una empresa, mientras que un objeto de esa clase (también denominado instancia) representaría a uno de esos empleados en particular.
- **Métodos**. Un método es una función implementada dentro de un objeto con la finalidad de realizar una serie de operaciones sobre los datos que contiene. Así, en el ejemplo anterior, las ventanas de Windows tienen el método maximizar, minimizar, cerrar, etc. que modifican el aspecto de la ventana al dibujarse.
- **Mensajes**. Un programa basado en objetos, utiliza como mecanismo de comunicación entre los mismos una serie de métodos que pueden ser llamados

por agentes externos al objeto. A la acción de ejecutar métodos externamente a un objeto se le denomina mensaje. Por ejemplo, cuando decimos a Windows que deseamos ver el escritorio, y que por tanto todas las ventanas abiertas deben minimizarse, Windows se encarga de enviar mensajes de minimización a todas las ventanas que en ese momento está gestionando. Cada una de estas ventanas ejecutará como consecuencia su método de minimización lográndose de esta forma el resultado buscado.

Características principales de la POO

Las principales características de la POO son: la abstracción, el encapsulamiento, la herencia y el polimorfismo:

- **Abstracción**. Es el mecanismo de diseño en la POO. Nos permite extraer de un conjunto de entidades datos y comportamientos comunes para almacenarlos en clases. El ejemplo clásico para entender estas características se realiza con los vehículos. Fácilmente se puede establecer una definición de vehículo como todo aquello que es capaz de transportar personas. En este caso tendremos que aviones, coches, barcos, triciclos y bicicletas tienen algo en común que debemos poder representar en un sistema de programación orientado a objetos.

- **Encapsulamiento**. Mediante esta técnica conseguiremos que cada clase de objetos sea una caja negra, de tal manera que los objetos de esa clase se puedan manipular como unidades básicas. Los detalles de la implementación se encuentran dentro de la clase, mientras que desde el exterior, un objeto será simplemente una entidad que responde a una serie de mensajes públicos (también denominados interfaz de la clase). Así para el caso de la clase vehículo, nos interesa saber inicialmente cuantas personas caben, su autonomía, y la posición y velocidad que tiene en un determinado momento y su modificación. Nos da igual que esté utilice gasolina, energía nuclear o tracción animal.

- **Herencia**. Es el mecanismo que nos permite crear clases derivadas (especialización) a partir de clases bases (generalización). Es decir, podríamos tener la clase “vehículo” (clase base) y la clase “avión” derivando de la anterior. Al especializarse, se agregarán características y métodos a la clase vehículo generando como consecuencia un nuevo tipo de objeto “el avión”, sin dejar por ello de ser un vehículo. Es decir, un avión, además de tener personas y saber donde está, nos puede informar del número de motores que tiene, de la altitud a la que se encuentra, etc. Una librería de clases (como la MFC) no es más que un conjunto de definiciones de clases interconectadas por múltiples relaciones de herencia.

- **Polimorfismo**. Esta característica nos permite disponer de múltiples implementaciones de un mismo método de clase, dependiendo de la clase en la que se realice. Es decir, podemos acceder a una variedad de métodos distintos (con el mismo nombre) mediante el mismo mecanismo de acceso. Por ejemplo, podremos decirle a cualquier vehículo que se mueva sin importarnos como lo hace. El coche arrancará el motor, meterá una marcha y soltando el embrague comenzará a moverse. El avión encenderá las turbinas, bajará los flaps y procederá a realizar la operación de despegue, etc. El caso es que para el que está considerando el avion y el coche como vehículos, la operación muévete es la misma, y es el objeto el que, en función del tipo específico al que pertenece, decidirá cómo realizará el movimiento. En C++ el polimorfismo se consigue mediante la definición de clases derivadas, funciones virtuales y el uso de punteros a objetos.

Otros dos conceptos muy importantes en la POO son relativos a la creación y destrucción de objetos. En lenguajes estructurados convencionales, cuando se define una variable se le reserva espacio en memoria y, si no se inicializa expresamente, se hace por defecto (por ejemplo, en C una variable global siempre se inicializa a 0, pero una automática no, por lo que si no se inicializa expresamente su contenido inicial será basura); por otra parte, cuando se destruye una variable (porque se abandona el ámbito de su definición -scope -) se libera la memoria que estaba ocupando. Si ahora hacemos el paralelismo obligado entre variables y objetos para los lenguajes POO nos daremos cuenta de que deben existir procedimientos especiales de construcción y destrucción de objetos. En concreto, cada clase tiene dos funciones miembro especiales denominadas constructor y destructor.

- **Constructor**: Función miembro que es automáticamente invocada cada vez que se define un objeto, su objetivo es la inicialización del mismo. Toma el mismo nombre que la clase, puede recibir parámetros y podemos tener varios constructores definidos.

- **Destructor**: Función miembro invocada automáticamente cada vez que se destruye un objeto. Su objetivo es realizar operaciones como liberación de memoria, cerrar ficheros abiertos, etc. Toma el mismo nombre de la clase comenzado primero por el carácter "~", no toma parámetros y no admite la sobrecarga (sólo puede existir uno en cada clase).

En muchos casos, para las clases más sencillas, podemos encontrar clases que no tiene constructor o destructor, o ninguno de los dos. En C++, siempre existen constructores y destructores por defecto que realizan una inicialización/liberación estándar.

1.4. Organización de los apuntes

En primer lugar se dedicará un capítulo a introducir las modificaciones menores de C que aparecen en C++. Es decir, algunas de las características que utilizables en una programación procedural aporta C++. Seguidamente se dedicará el resto de capítulos a ir explicando ordenadamente los distintos mecanismos que tiene C++ para realizar una programación orientada a objetos. Para ello se comenzará explicando en más detalle el concepto de clase y de objetos. Una vez establecida la base de lo que es una clase, se dedicará un capítulo a explicar el mecanismo de la herencia tan importante para lograr un código eficiente y reutilizable. Este concepto se completará con el desarrollo del polimorfismo en C++ que se aborda en el quinto capítulo, especialmente interesante si se observa las consecuencias que este tiene sobre Windows. Uno de los aspectos más desconocidos en C++ (para los programadores nóveles) se aborda en el capítulo 6 al introducir el concepto de plantilla.

Hasta este punto los conceptos desarrollados son comunes a los lenguajes de POO. En el capítulo siete se procederá a exponer el modo en que C++ realiza las operaciones de entrada y salida, así como algunos de los objetos y funciones básicas disponibles en las librerías estandar con esta finalidad. Para finalizar la parte de los apuntes dedicada a la exposición del lenguaje C++ se realizará una breve descripción del mecanismo para tratar los errores: las excepciones.

A lo largo del contenido de este curso se irán intercalando ejemplos y aplicaciones de los conceptos explicados que permitirán a su vez ilustrar aspectos auxiliares de C++. Un elemento importante y que requerirá una exposición detallada en una asignatura posterior, es la representación gráfica de los elementos que integran un sistema informático. Por ello se han ido integrando de forma natural estos modos de representar gráficamente los elementos y se incluye un breve resumen de los mismos en el capítulo 8.

2. Modificaciones menores a C en C++

Antes de proceder a enumerar los cambios introducidos en C al realizar programación procedural en C++, es necesario destacar que alguna de estas modificaciones están incluidas en los compiladores actuales de C, puesto que al convivir durante su desarrollo con la aparición de C++ se introdujeron para mejorar la eficiencia y legibilidad del lenguaje.

2.1. Extensión de los ficheros

El primer cambio que tiene que conocer cualquier programador es que los ficheros fuente de C++ tienen la extensión ***.cpp** (de C plus plus, que es la forma oral de llamar al lenguaje en inglés), en lugar de ***.c**. Esta distinción es muy importante, pues determina ni más ni menos el que se utilice el compilador de C o el de C++. La utilización de nombres incorrectos en los ficheros puede dar lugar a errores durante el proceso de compilación.

En C++ los ficheros de código se terminan con la extensión ***cpp*** de forma que el entorno de desarrollo puede decidir que compilador va a utilizar.

2.2. Palabras reservadas

2.2.1. Los elementos de C/C++

Cuando el compilador analiza un fichero de texto (código) lo primero que hace es romperlo en elementos significativos (TOKENS), lo que en los compiladores se denomina como PARSE. Estos elementos básicos son los siguientes:

- Punctuators: son los caracteres especiales que se utilizan como separadores de las distintas construcciones de un lenguaje de programación: # [ ] { } ; : * … = ( )
- Keywords: que son palabras reservadas que solo se pueden utilizar para lo que está previsto por el lenguaje salvo que sean parte de un literal (cadena de caracteres).
- Identificators: son palabras que utilizaremos para denominar algo dentro del código, como por ejemplo el nombre de una variable. Los identificadores en C y C++ son sensibles a mayúsculas y minúsculas[4], deben comenzar por una letra del alfabeto inglés o el carácter '_' , y a continuación se podrán agregar letras y números, incluyendo la barra baja hasta un máximo de 31 en ANSI C. Obviamente no podrán ser nunca iguales a una palabra reservada.
- Literals: Se denomina así a la especificación de un valor concreto de un tipo de dato.
    - Numeros: 3 3.1416 0.314e1 0.314e+1 .3141e1f
      Mediante sufijos podemos modificar el tipo básico que es int para las constantes sin punto, y double para las que tienen (F para float) (L para long) (U para unsigned).
      Mediante prefijos podemos expresar un modo de codificación (0x hexadecimal) (0 para octal)
    - Caracteres: 'a' '0' '\n' '\\' '\''
    - Cadenas de caracteres. Las cadenas de caracteres están constituidas por un vecto de caracteres explícitos más un carácter terminador que es el '\0' que corresponde al valor 0. "Hola Mundo" "\"Hola Mundo entre comillas\""

[4] Case sensitive

- Operators: Igual que en matemáticas son combinaciones de símbolos o un símbolo que realizan una acción específica sobre uno o más operandos y que habitualmente devuelven un valor: + * / % >> etc.

De cara a la escritura de código se suelen seguir unas recomendaciones en el uso de identificadores. Se exponen aquí brevemente antes de pasar a ver las nuevas palabras clave y operadores introducidos en C++:

- MACROS y DEFINICIONES: todo en mayúsculas. Si tienen varias palabras separarlas con _

```
#define RADIO 3.0F
#define MAX_NUM 3453
```

- Estructuras y tipos de datos definidos por el usuario: se recomienda empezarlos con mayúscula y poner en mayúscula el comienzo de cada palabra explicativa

```
struct PoligonoRegular { …} ;
```

- Funciones y métodos: comenzar su identificador con un verbo en minúscula habitualmente en infinitivo o imperativo, a continuación cada palabra en mayúscula

```
void imprimeContenido( … )
```

- Variables: todo en minúsculas. Si hay varias palabras separalas por _.

```
int contador = 0, cuenta_cuentos = 0;
```

2.2.2. Nuevas palabras clave incluidas por C++

Mientras que el conjunto de palabras clave de C , y por tanto reservadas, era de 32:

if	while	char	unsigned	sizeof
auto	long	extern	continue	volatile
double	switch	return	for	do
int	case	union	signed	static
struct	enum	const	void	goto
break	register	float	default	short
else	typedef			

En la actualidad, C++20 define 64 más. Eso significa que hay un total de 96 palabras clave, y que como consecuencia no pueden ser utilizadas como identificadores. Algunas de ellas han sido redefinidas o han quedado en desuso. En cualquier caso, para mantener la compatibilidad se incluyen entre la lista de palabras de uso exclusivo:

Palabras clave de C++ (		
alignas (desde C++11)	default (*11)	register (*17)
alignof (desde C++11)	delete (*11)	reinterpret_cast
and	do	requires (desde C++20)
and_eq	double	return
asm	dynamic_cast	short
atomic_cancel (TM TS)	else	signed
atomic_commit (TM TS)	enum	sizeof (*11)
atomic_noexcept (TM TS)	explicit	static
auto (*11)	export (*11) (*20)	static_assert (desde C++11)
bitand	extern (*11)	static_cast
bitor	false	struct (*11)
bool	float	switch
break	for	synchronized (TM TS)
case	friend	template
catch	goto	this
char	if	thread_local (desde C++11)
char8_t (desde C++20)	inline (*11)	throw
char16_t (desde C++11)	int	true
char32_t (desde C++11)	long	try
class (*11)	mutable (*11)	typedef
compl	namespace	typeid
concept (desde C++20)	new	typename
const	noexcept (desde C++11)	union
consteval (desde C++20)	not	unsigned
constexpr (desde C++11)	not_eq	using (*11)
constinit (desde C++20)	nullptr (desde C++11)	virtual
const_cast	operator	void
continue	or	volatile
co_await (desde C++20)	or_eq	wchar_t
co_return (desde C++20)	private	while
co_yield (desde C++20)	protected	xor
decltype (desde C++11)	public	xor_eq

(TMTS): Transactional Memory Technical Specification ISO/IEC TS 19841:2015
**11, *17, *20 : El significado ha cambiado en C++11, C++17, C++20 respectivamente*

El significado de las mismas se irá viendo a lo largo del curso. Algunas de las palabras clave se incluyen para facilitar la escritura e interpretación del código, y por tanto constituyen una alternativa a la representación. En C (y en C++ hasta su eliminación definitiva con C++17)

existía este mismo concepto mediante los Trigrafos. En C++ aparecen las palabras alternativas y los digrafos, todos ellos resumidos en la siguiente tabla:

operador	Palabra clave	simbolo	digrafo
&&	and	{	<%
&=	and_eq	}	%>
&	bitand	[	<: (1)
|	bitor	]	:>
~	compl	#	%:
!	not	##	%:%:
!=	not_eq		
||	or		
|=	or_eq		
^	xor		
^=	xor_eq		

(1) Si el preprocesador se encuentra la secuencia <:: y el siguiente carácter no es ni : ni >, entonces no se aplica el digrafo.

2.3. Nuevos operadores

En C++ aparecen nuevos operadores. La utilidad de los mismos se irá viendo a lo largo del curso puesto que muchos de ellos requieren nociones de la POO:

Operador	Significado
`::`	Operador de resolución de ámbito de una variable o función miembro de una clase. Denominado como operador ***scope*.**
`this`	Toma el valor de la dirección del elemento desde el que ha sido invocado.
`&`	Adicionalmente a los significados en C, se utilizará para definir ***referencias*** y ***alias***.
`&&`	Adicionalmente a los significados de C, se usará para definir referencias a valores r. Elementos que solo se pueden encontrar a la derecha de una signacion.
`new`	Crea un objeto del tipo indicado a continuación. Modo habitual de reserva

	dinámica de memoria en C++
delete	Destruye un objeto creado dinámicamente con el operador ***new***
.*	Accede al miembro de una clase cuando el miembro es referenciado por un puntero.
->*	Accede al miembro de una clase cuando tanto el miembro como la instancia de la clase están referenciados por un puntero.
typeid	Identificación de tipo
???_cast	Conversiones forzadas del tipo de una expresión.

Los dos modos de acceso nuevos (.* y ->*), no aparecerán más en estos apuntes por lo que se incluye a continuación un breve ejemplo y explicación de los mismos.

```
#include <cstdio>
struct Complex{
    float real;
    float imag;
};

void main()
{
    struct Complex a={0,0}, *b; //creo dos variables de tipo Complex
    float Complex::*valor; //defino un puntero a floats de Complex
    b=&a;
    //le asigno a valor la direccion relativa de la parte real
    valor= &Complex::real;
    a.*valor=5.0F;
    printf("\n%f + %fj",a.real, a.imag);
    //le asigno a valor la direccion relativa de la parte imag
    valor = &Complex::imag;
    b->*valor=3.0F;
    printf("\n%f + %fj",a.real, a.imag);
}
```

Para intentar entenderlo, pensemos que a diferencia de un puntero normal en C, valor representará puntero relativo. Es decir contiene la dirección de un float relativo a la dirección de un contenedor. Una estructura de tipo Complex, estará en una posición de memoria , digamos 1000, por lo que en 1000 estará también la parte real (float) y en 1004 la parte imaginaria. Un puntero a un miembro de una estructura (y mas delante de una clase) se

puede entender como un desplazamiento respecto de la dirección de la estructura. Así, el valor de &Complex::real podemos decir que es 0, mientras que el de &Complex::imag es 4.

Los dos operadores nuevos, nos permitirán aplicar estos tipos de punteros que codifican direcciones relativas, a las instancias de las estructuras o a los objetos.

Como consecuencia de la aparición de los nuevos operadores, es necesario revisar la precedencia y asociatividad de todos ellos en C++.

La precedencia nos indica el orden de asociación de un operador con sus operadorandos. De forma que un operador con precedencia (mas alto en la tabla que otro) sobre otro, se agrupará antes con sus operandos. Esto es lo que intuitivamente hacemos en matemáticas cuando encontramos sumas y productos mezclados: 5+4*3+8 lo hacemos de la forma siguiente: (5+(4*3))+8 . Es decir, el primero en asociarse a sus operandos es el *.

Cuando dos operadores tienen la misma asociatividad (por ejemplo por que fueran el mismo) entonces se sigue la regla de asociatividad, que es de izquierda a derecha o de derecha a izquierda: 3+5+2-4+8 lo interpretaremos como (((3+5)+2)-4)+8 es decir de izquiera a derecha.

Prec.	**Operador**	**Asoc.**
1	`::`	>>
2	`() [] . -> v++ v-- ???_cast typeid`	>>
3	`-a +a ~ ! & * ++v –v sizeof new delete (tipo)`	<<
4	`->* .*`	>>
5	`a*b a/b a%b`	>>
6	`a+b a-b`	>>
7	`<< >>`	>>
8	`< <= > >=`	>>
9	`== !=`	>>
10	`a&b`	>>
11	`a^b`	>>
12	`a|b`	>>
13	`&&`	>>
14	`||`	>>

15	a?b:c	<<
16	= *= /= %= += -= <<= >>= &= |= ^=	<<
17	,	>>

NOTA IMPORTANTE: Tanto en C como en C++ se establece el modo en que se interpretan las expresiones, sin embargo no se establece el orden de ejecución dentro de un operador de estas expresiones.. Así en la expresión a+b , el estándar deja libertad al compilador para decidir si obtener primero el valor de a y luego el de b y sumarlos o al revés. Lo mismo se aplica a la evaluación de los parámetros de una función.

Las únicas excepciones son para el operador ternario, y en los operadores lógicos && y || (siendo el orden de izquierda a derecha). Todo este tema es un poco más complejo dado que además hay que analizar si los efectos de la expresión se evalúan antes o después, pero para el punto en el que nos encontramos es suficiente con estas nociones.

2.4. Comentarios

En el C original, la inclusión de comentarios en los ficheros de código venía delimitado por una combinación de caracteres de comienzo (**/***) y otra de finalización de comentario (***/**). De tal forma que todo el texto contenido entre estos dos delimitadores era eliminado por el preprocesador antes de proceder a realizar cualquier operación de interpretación.

En C++ adicionalmente se pueden introducir comentarios de una sola línea. Así, desde el momento en que en una línea aparece e conjunto de caracteres (//) hasta el final de esa línea, el texto abarcado es considerado como un comentario.

El siguiente ejemplo muestra la combinación de ambos tipos de comentarios.

```
/* Este es un comentario típico de C,
en el que se incluyen varias líneas anuladas
por los delimitadores */

void main() // y este es un comentario de C++
{
int i;
for(i=0;i<10;i++)
      {
      //si quisiera introducir varias lineas
      //bastará con introducir la doble barra
      // en cada linea, o utilizar /* */
      printf("%d",i);
      }
}
```

Es interesante destacara que los comentarios en // han sido introducidos en la mayoría de los compiladores de C dada la gran comodidad que suponen a la hora de anular lineas de código durante el desarrollo.

2.5. Tipo de datos bool

La palabra clave **bool** declara un nuevo tipo especial de variable, denominada booleana (0.1) que solo puede tener dos valores: cierto (**true**) y falso (**false**).

```
bool val = false; // declara val variable Booleana y la inicia
val = true;       // ahora se cambia su valor (nueva asignación)
```

Las palabras clave false y true son literales que tienen valores predefinidos (podemos considerar que son objetos de tipo booleano de valor constante predefinido). **false** tiene el valor falso (numéricamente es cero), **true** tiene el valor cierto (numéricamente es uno).

Tenga en cuenta que los bool y los int son tipos distintos. Sin embargo, cuando es necesario, el compilador realiza una conversión automática de forma que pueden utilizarse libremente valores bool (true y false) junto con valores int sin utilizar una conversión explícita.

```
int x = 0, y = 5;
if (x == false) printf("x falso.\n");        // Ok.
if (y == true) printf("y cierto.\n");        // Ok.
if ((bool) x == false) printf("x falso.\n"); // cast innecesario
```

Estas conversiones entre tipos lógicos y numéricos, son simplemente una concesión del C++ para poder aprovechar la gran cantidad de código C existente. Tradicionalmente en C se hacía corresponder falso con el valor cero y cierto con el valor uno (o distinto de cero).

Para convertir tipos aritméticos (enumeraciones, punteros o punteros a miembros de clases) a un tipo **bool**, la regla es que cualquier valor cero; puntero nulo, o puntero a miembro de clase nulo, se convierte a **false**. Cualquier otro valor se convierte a true. En principio el estándar establece que solo los valores enteros son convertidos automáticamente siendo necesario una conversión explícita en otro caso, sin embargo, a menudo, los compiladores no requieren de este cast por comodidad.

2.6. Tipos de datos definidos por el usuario

Simplificación en la declaración de estructuras y uniones

En C las palabras **struct** y **union** forman parte del identificador del tipo de datos definido por el usuario. En C++ esto se simplifica eliminando estas palabras en el momento de declarar las variables o referirse al tipo de datos definido.

El siguiente extracto de código está escrito en C:

```
/*declaración del patrón de la estructura complejo*/
struct complejo
{
float x;
float y;
};

void main()
{
/*petición de variables*/
int j;
struct complejo micomplejo;
/*código*/
j=sizeof(struct complejo);
}
```

El mismo código en C++ quedaría:

```
//declaración del patrón de la estructura complejo
struct complejo
{
float x;
float y;
};

void main()
{
//petición de variables
int j;
complejo micomplejo;
//código
j=sizeof(complejo);
}
```

Se observa como desde el punto de vista del código, queda más claro y cómodo.

Las enumeraciones como tipo de datos

En C++ el tipo de datos **enum** es un nuevo tipo puesto que en C las enumeraciones eran de tipo de datos **int**. A diferencia de lo que ocurría entonces, en C++ no se podrá asignar directamente un entero a una variable de timo **enum**. Para poder realizar esta asignación será necesario indicar la conversión explícitamente.

Por otro lado, al igual que ocurría con las estructuras y las uniones, en las enumeraciones se simplifica la indicación del tipo durante su uso prescindiendo de la palabra **enum**.

El siguiente extracto de código muestra un ejemplo de una enumeración en C++:

```
enum notas {suspenso, aprobado, bien, notable, sobresaliente};
notas nota1 = bien;
notas nota2;
nota2 = notas(2); //conversión explícita.
// nota2 = 2; sería en C pero que en C++ es incorrecto
```

Sin embargo, si es posible realizar la asignación contraria. Es decir, una variable entera puede recibir el valor de una variable de tipo enumeración. Por defecto, el número entero asignado a cada uno de los posibles valores de una variable **enum** van por orden comenzando desde 0. De esta forma, *suspenso* vale 0, *aprobado* vale 1, *bien* vale 2, etc. Sin embargo, el programador puede establecer estos valores en la definición de la enumeración. El siguiente código muestra estas dos características:

```
enum colores {rojo=3,verde=5, azul, amarillo=8};
int i = verde;
```

En este caso, *rojo* vale 3, *verde* vale 5, *azul* vale 6 y *amarillo* vale 8. Es decir, cuando explícitamente no se da un valor, se sigue la numeración del valor anterior.

Para ayudar a entender el nivel de fuerza que este tipo de datos tiene, las siguientes propiedades se cumplen a la hora de evaluar expresiones con tipos enum:

No habrá error si a una variable de tipo enum le asigno (indicando la conversión) un valor no definido: nota2=notas(100) sería una expresión válida.

Se puede asignar el mismo valor numérico a dos elementos de la enumeración distintos.

Una enumeración es promocionable directamente a entero en las expresiones aritméticas pero realmente no hay definida una aritmética específica. Por eso será posible utilizar expresiones del tipo *entero=entero+enum* , *entero = enum+entero,* o *entero = enum+enum* . Puesto que la operación suma no está definida para las enumeraciones, y no es

posible asignar a una enumeración un valor entero de forma directa, el resultado de estas operaciones nunca puede ser directamente recibido por una variable de tipo enum sin una conversión explícita.

Uniones anónimas

Aunque su aplicación más inmediata se da en el interior de estructuras más complejas, es conveniente mencionar ya que C++ admite la definición de uniones anónimas. Su finalidad es la de definir un conjunto de campos (miembros en el caso de clases) ubicados en la misma zona de memoria. Simplifica el acceso a estos campos evitando el tener que utilizar, como ocurre en C, el indicador del tipo de acceso sobre la unión. Para visualizar la variación, se mostrará el mismo segmento de programa escrito en C y en C++:

El siguiente programa en C:

```
struct datos /*definición del patrón de estructuras*/
       {
       int tipo;
       union _dato /*union dentro de la estructura*/
              {
              char caracter;
              int entero;
              float decimal;
              }dato;
       };
int main()
{
struct datos midato;
midato.tipo=2;
midato.dato.entero=5;
}
```

Este mismo código escrito en C++ quedaría como sigue:

```
struct datos /*definición del patrón de estructuras*/
       {
       int tipo;
       union  /*union dentro de la estructura*/
              {
              char caracter;
              int entero;
              float decimal;
              };
       };
int main()
{
```

```
datos midato;
midato.tipo=2;
midato.entero=5;
}
```

2.7. Flexibilidad en la declaración de variables

Las variables locales en C debían ser declaradas al comienzo del bloque de una función, de forma que el código quedaba siembre estructurado en dos zonas: la petición de variables y el conjunto de sentencias ejecutables de código.

En C++ se permite la declaración de variables en cualquier lugar de un bloque, entendiéndose como bloque el conjunto de sentencias encerradas entre llaves {}.

Esto incluye la posibilidad de declarar variables en el interior de una sentencia como se mostrará en los ejemplos siguientes:

```
void  funcion(void)
{
int i=8;
for(int j=0;j<i;j++)
      {
      printf("Hola mundo");
      int k=3;
      printf("valor de k %d",k);
      }
if(--i==0)return;
}
```

En C quedaba claro cual era el ámbito de una variable, puesto que una variable local era visible desde todo el cuerpo de la función en la que estaba definida, y una variable global lo era desde el punto en el que estaba en el código en adelante.

En C++, el ámbito de una variable es desde el punto en el que esta está declarada hasta el final del bloque en el que se encuentra.

En el caso de las variables globales el criterio es el mismo que en C. El siguiente ejemplo puede aclarar esto:

```
int val=2;
printf("%d",val); //se imprime 2
for(int i=0;i<10;i++)
      {
      printf("%d",val); //se imprime 2
      int val=3;
```

```
        printf("%d",val); //se imprime 3
        }
printf("%d",val);  // se imprime 2
```

Una variable local oculta al igual que en C a una variable más global relativamente.

Es decir, bajo un mismo nombre, las variable que se considerará será la más local entre las visibles.

C++ dispone del operador (::), llamado operador de resolución de visibilidad. Este operador, antepuesto al nombre de una variable global que está oculta por una variable local del mismo nombre, permite acceder al valor de la variable global. No es posible, sin embargo, acceder a una variable local que esté oculta por otra variable local. El siguiente ejemplo muestra el modo de uso de el operador **scope**.

```
int i=0;
void main()
{
int i;
while(::i<10)
       {
       ++::i;
       for(i=0;i<10;i++)
              printf("%d : %d\n",::i,i);
       }
return 1;
}
```

En este programa se ha realizado la anidación de dos bucles con dos iteradores del mismo nombre, uno global y otro local.

Por útimo, la duración de las variables locales (**auto**) será el bloque en el que están definidas. Es decir, que la variable es destruida en el momento en que se finaliza la ejecución del bloque en el que se encuentra su declaración. Es posible definir variables estáticas al igual que en C, de forma que una variable local definida como estática (**static**) no es destruida hasta el final del programa, y por tanto sólo es creada la primera vez que se ejecuta el bloque en el que se encuentra su declaración .

Hay que tener precaución a la hora de utilizar declaraciones de variables en el interior de un **switch**, puesto que en este caso no existen bloques diferenciados sino distintos

puntos de acceso. Esto provoca el que no se realice la inicialización de una variable dentro de un bloque, o que se produzca una doble definición de una variable. El siguiente ejemplo muestra una declaración errónea, y su solución:

```
int main()
{
int i=2;
float v;
switch (i)
	{
	case 3:
		int i=5;
	break;
	case 2:
		v=3.5F;
		for(int i=0;i<10;i++)v=1.0F;
	default:
		v=3.0F;
	break;
	}
}
```

El compilador arroja los siguientes errores:

1. La inicialización de ***i*** es evitada por las etiquetas case.
2. La inicialización de ***i*** es evitada por las etiqueta default.
3. Redefinición de ***i***.

La forma más sencilla de evitar este error sin modificar el número de variables locales del programa, es establecer en el interior de cada caso un bloque:

```
int main()
{
int i=2;
float v;
switch (i)
	{
	case 3:
		{int i=5;}
	break;
	case 2:
		{
		v=3.5F;
```

```
                for(int i=0;i<10;i++)v=1.0F;
                }
        default:
                v=3.0F;
        break;
        }
}
```

Revisión de los tipos de almacenamiento[5]

C++ dispone de los siguientes especificadores de tipo de almacenamiento: auto, extern, static y register. Además algunos modificadores: const, volatile y mutable. Como ya se ha visto, todos ellos son palabras clave del lenguaje.

Almacenamiento automático: Por defecto las variables locales son auto. Estas variables se crean durante la ejecución, y se elige el tipo de memoria a utilizar en función del ámbito temporal de la variable. Una vez cumplido el ámbito, la variable es destruida. Es decir, una variable automática local de una función se creará cuando sea declarada, y se destruirá al terminar la función. Una variable local automática de un bucle será destruida cuando el bucle termine.

Almacenamiento estático: El especificador static se utiliza tanto para variables como para funciones:

```
static <tipo> <nombre_variable>;
static <tipo> <nombre_de_función>(<lista_parámetros>);
```

Cuando se usa en la petición de una variable, este especificador hace que se asigne una dirección de memoria fija para el objeto mientras el programa se esté ejecutando. Es decir, su duración es la del programa. En cuanto la visibilidad conserva el que le corresponde según el punto del código en que aparezca la declaración. Debido a que tiene una posición de memoria fija, su valor permanece, aunque se trate de una variable declarada de forma local, entre distintas reentradas en el ámbito del objeto. Los objetos estáticos no inicializados toman por defecto un valor nulo, aunque conviene siempre indicar el inicializador com buena practica a la hora de programar.

Esta inicialización sólo afectará al momento de creación de la variable.

[5] Este apartado ha sido extraido y reformateado de la siguiente página web: http://c.conclase.net/curso/index.php. Los cursos de C y C++ con clase son altamente recomendados por lo bien que están explicados y el carácter docente de esta página web.

Curiosamente si utilizamos el modificador static sobre una variable global (fuera de cualquier bloque) estaremos indicando que esa variable no debe ser accesible desde otros ficheros del programa, sino solo desde el fichero que las está declarando.

Almacenamiento externo: para indicarlo se hace uso de la palabra clave extern y es aplicable tanto a variables como a funciones:

```
extern <tipo> <nombre_variable>;
[extern] <tipo> <nombre_de_función>(<lista_parámetros>);
```

Este especificador se usa para indicar que el almacenamiento y valor de una variable o la definición de una función están definidos en otro módulo o fichero fuente. Las funciones declaradas con **extern** son visibles por todos los ficheros fuente del programa, salvo que se defina la función como static.

El especificador extern sólo puede usarse con objetos y funciones globales.

Las declaraciones de prototipos son declaraciones externas por defecto, de ahí que al igual que ocurría con la especificación auto, normalmente no veamos esta palabra clave en las declaraciones de funciones. Se puede usar `extern "c"` con el fin de prevenir que algún nombre de función escrita en C pueda ser ocultado por funciones de programas C++. Este especificador no se refiere al tipo de almacenamiento, ya que sabemos que en el caso de prototipos de funciones es el especificador por defecto. En realidad es una directiva que está destinada al enlazador, y le instruye para que haga un enlazado "C", distinto del que se usa para funciones en C++.

Lógicamente, este especificador lo usaremos con programas que usen varios ficheros fuente, que será lo más normal con aplicaciones que no sean ejemplos o aplicaciones simples.

Almacenamiento en registro: Para especificar este tipo de almacenamiento se usa el especificador register.

```
register <tipo> <nombre_variable>;
```

Indica al compilador una preferencia para que la variable se almacene en un registro de la CPU, si es posible, con el fin de optimizar su acceso, consiguiendo una mayor velocidad de ejecución. Los datos declarados con el especificador register tienen el mismo ámbito que las automáticas. De hecho, sólo se puede usar este especificador con parámetros y con objetos locales.

El compilador puede ignorar la petición de almacenamiento en registro, que se acepte o no estará basado en el análisis que realice el compilador sobre cómo se usa la variable. Una variable de este tipo no reside en memoria, y por lo tanto no tiene una dirección de memoria, es decir, no es posible obtener la dirección a un objeto declarado con el tipo de almacenamiento *register*.

Se puede usar un registro para almacenar objetos de tipo char, int, float, y punteros. En general, objetos que quepan en un registro

2.8. Modificaciones a las funciones

Funciones inline

C++ permite introducir el modificador **inline** en la declaración de las funciones. Con este modificador indicamos al compilador que consideramos conveniente que las llamadas realizadas a esta función sean sustituidas por el cuerpo de código de la función. El efecto respecto de la funcionalidad del programa es el mismo en ambos casos, pero en caso de realizarse esta sustitución, el programa resultante es más rápido, al evitarse el salto a la función, a costa de un código ejecutable más extenso.

Es recomendable utilizar el modificador **inline** en funciones pequeñas, que son llamadas en pocos lugares (que es diferente a que sean llamadas pocas veces).

Para poder asignar el modificador **inline** a una función, dicha función debe estar definida antes de que sea invocada, de lo contrario el compilador no lo tendrá en cuenta. Por este motivo, las funciones **inline** son normalmente definidas en los ficheros de cabecera. Modificar una función para que sea de este tipo implica anteponer **inline** al tipo del valor retornado por la función.

```
inline int maximo(int a, int b)
{
return((a>b)?a:b);
}
int main()
{
int a,b;
scanf("%d%d",&a,&b);
printf("El máximo de %d y %d es %d",a,b,maximo(a,b));
}
```

El comportamiento de las funciones **inline** recuerda a las macros de C. Sin embargo las macros de C definidas por medio de la directiva al preprocesador **#define**, pueden fácilmente llevar a errores de código debido a que se realiza una sustitución textual de los parámetros. Esto queda perfectamente resuelto con esta nueva herramienta aportada por C++.

Por ejemplo, en C, la macro correspondiente a la función anterior, se escribiría correctamente de la forma siguiente:

```
#define MAXIMO(a,b) (((a)>(b))?(a):(b))
```

En el siguiente código, se observa un efecto no deseado en el valor de las variables para el caso de uso de las macros:

```
void main()
{
int a=2,b=4,max;
max=MAXIMO(++a,--b);
printf("el máximo de %d y %d es %d",a,b,max);
}
```

El mensaje de salida del ejemplo sería:

El máximo de 3 y 2 es 2

Esto es debido a que *a* ha sido incrementada una vez y *b* dos veces, una en la evaluación de la expresión de la condición, y otra en la evaluación de la expresión del resultado. Este tipo de errores, así como las precauciones clásicas tomadas por medio de los paréntesis quedan solventadas por medio del uso de las funciones **inline**.

Adicionalmente, cualquier función definida en la declaración en el interior de una clase, se asume como función **inline**.

Así, en el fichero de cabecera se podría incluir la siguiente línea, siendo entonces la función máximo considerada como **inline**:

```
//cabecera.h
class MiClase
{
      int maximo(int a,int b){return((a>b)?a:b);};
};
```

A diferencia de lo habitual, el modificador **inline se asocia con la definición** de la función en vez de con su declaración. En el fondo esto es porque una función definida como

inline no será utilizada en la fase de enlazado y por tanto podremos tener distintas versiones de una función en las distintas unidades de compilación.

Funciones sobrecargadas

La sobrecarga de funciones es una característica de C++ que hace que los programas sean más legibles.

Consiste en declarar y definir varias funciones distintas que tienen un mismo nombre. En el momento de la compilación se decide si se llama a una u otra función dependiendo del número y/o tipo de los argumentos actuales de la llamada a la función.

La sobrecarga de funciones no admite funciones que difieran sólo en el tipo del valor de retorno, pero con el mismo número y tipo de argumentos.

De hecho, el valor de retorno no influye en la determinación de la función que es llamada; sólo influyen el número y tipo de los argumentos.

El siguiente ejemplo muestra como sobrecargar la función suma para números reales y números complejos:

```
struct complejo
 {
 float real, imaginario;
 }
float suma(float a, float b)
 {
 return (a+b);
 }
complejo suma(complejo a, complejo b)
 {
 complejo c;
 c.real=a.real+b.real;
 c.imaginario=a.imaginario+b.imaginario;
 return c;
 }
void main()
 {
 complejo a={1.0F,1.5F},b={0.0F,1.1F},c;
 float d=3.0F,e=8.2F,f;
 c=suma(a,b); //utiliza la funcion suma de complejos
 f=suma(d,e); //utiliza la funcion suma de enteros
 }
```

En el uso de la sobrecarga de funciones hay que tener cuidado con la posibilidad de que se produzcan ambigüedades en la decisión de qué función va a ser ejecutada.

En caso de que pueda darse esta ambigüedad por ser los tipos de datos promocionables o equivalentes se puede romper la ambigüedad mediante una conversión explícita. Por ejemplo:

```
void imprime(double a);
void imprime(long a);
void main()
{
int b=8;
imprime(b);
imprime(static_cast<long>(b));
imprime((double)b);
}
```

En este ejemplo, la primera llamada a la función imprime es ambigua puesto que *b* puede ser promocionado tanto a **double** como a **long**. Esta ambigüedad es resuelta en los dos casos siguientes. En la segunda llamada se ha utilizado uno de los nuevos operadores de C++ que es la conversión estática al tipo indicado entre llaves. Este nuevo operador aparece especialmente por la posibilidad de realizar conversiones con verificación de tipo durante la ejecución por medio de su dual **dynamic_cast**. El matiz entre los ditintos tipos de conversión se verá más adelante, baste ahora con mencionarlos.

Parámetros por defecto en una función

Una de las primeras cosas que uno aprende en el manejo de funciones en C es que el número de argumentos formales de una función (parámetros establecidos en la definición) debe coincidir con el número de argumentos actuales (parámetros introducidos en la ejecución). En C++ esto ya no es así, puesto que es posible definir valores por defecto para cada uno de los argumentos, de forma que si en el uso de una función no son indicados por el programador, dichos parámetros tomaran el valor previsto.

Es decir, en la declaración de una función, o en su definición se especificarán los valores que deberán asumir los parámetros cuando se produzca una llamada a la función y estos se omitan. El siguiente ejemplo, muestra una función que imprime el contenido de un vector con un indicador de tamaño que por defecto vale 3, y un indicador de si hay cambio de línea que vale por defecto **false**.

```
void imprimeVector(float v[], int tam=3,bool linea=false)
{
printf("(");
for(int i=0;i<tam;i++)printf(" %f",v[i]);
if(linea)printf(" )\n");
```

```
 else printf(" ) ");
}
void main()
{
float vector[5]={1.0F,2.0F,3.0F,4.0F,5.0F};
imprimeVector(vector);
imprimeVector(vector,5);
imprimeVector(vector,5,true);
}
```

Es importante destacar que una vez omitido un argumento en una llamada, hay que omitir todos los posteriors. Es decir, no es posible escoger omitir un parámetro sí y otro no, sólo es posible omitir los parámetros desde un punto determinado.

Otro aspecto interesante a tener en cuenta para evitar errores en la compilación, es que en caso de existir un prototipo de definición de la función (bien en el fichero de código o en una cabecera), los parámetros por defecto deben situarse en el prototipo y no en la definición. Así, el ejemplo anterior quedaría como sigue:

```
void imprimeVector(float v[], int tam=3,bool linea=false);

void main()
{
float vector[5]={1.0F,2.0F,3.0F,4.0F,5.0F};
imprimeVector(vector);
imprimeVector(vector,5);
imprimeVector(vector,5,true);
}
void imprimeVector(float v[], int tam,bool linea)
{
printf("(");
for(int i=0;i<tam;i++)printf(" %f",v[i]);
if(linea)printf(" )\n");
 else printf(" ) ");
}
```

2.9. Variables de tipo referencia.

Si se recuerda la entrevista al padre del C++ incluida en la introducción de los apuntes, uno de los aspectos que el entrevistador destacaba como importante aportación del lenguaje C++ era este. ¿Qué es pues una referencia?.

> Una referencia es un nombre alternativo o *alias* para una variable.

Según esta definición, una referencia no es una copia de la variable referenciada, sino que es la misma variable con un nombre diferente. Su principal aplicación está en el paso de parámetros por variable a las funciones, así como la posibilidad de encadenar resultados de operaciones como se verá a continuación.

Antes de comenzar sin embargo con todas las posibles aplicaciones y peculiaridades de este nuevo "tipo de variables", la regla práctica para no terminar liándose es la ya expuesta antes. Una referencia no es un puntero o algo externo a la variable referenciada… es la misma variable con un nombre distinto.

Se verá ahora la aplicación más sencilla pero con menos utilidad: la creación de un alias de una variable.

La forma de declarar una referencia a un objeto en general es:

```
tipo &alias = variable
```

Al igual que en los punteros, en caso de tener varias declaraciones seguidas, se adjuntará el & al identificador de la referencia en vez de al tipo básico.

```
int a,b;
int &c=a,d,&e=b;
```

Toda referencia, excepto las declaradas como parámetros formales en una función, debe ser siempre inicializada. Además una referencia no puede alterar el objeto al que se refiere una vez inicializada de igual forma que no es posible mover una variable de un sitio a otro de la memoria, sino que una vez creada permanece en el mismo sitio.

El siguiente ejemplo muestra la creación de un *alias* de otra variable:

```
void main()
{
int i;
int& j=i; //j es un alias de i
for(i=0;i<3;i++)printf("%d",j); //imprime 012
for(j=0;j<3;j++)printf("%d",i); //imprime 012
for(j=0;i<3;j++)printf("%d",j); //imprime 012
}
```

El efecto es el mismo que si sustituimos todas las *j* por *i*, ¡son la misma variable!, y por tanto los tres bucles **for** hacen exactamente lo mismo. Se observa por tanto, que la aplicación de las referencias dentro de un mismo bloque tiene poca utilidad. El hecho de que se haya mencionado esta aplicación es que ayuda al entendimiento del modo de proceder de las referencias.

La importancia de este nuevo mecanismo aportado por C++ reside fundamentalmente en las funciones, tanto en el paso de parámetros como en los valores de retorno.

Las referencias como parámetros de una función.

Una de las reglas básicas que se establecen en C en parte para diferenciarlo de conceptos derivados del PASCAL, es que los parámetros de las funciones son siempre pasos por valor. Es decir, al ejecutarse una función, los argumentos actuales de la llamada son copiados sobre los argumentos formales, de forma que una función siempre trabaja con una copia de los valores con los que fue llamada.

El efecto práctico derivado de esta regla era que cada vez que se quería realizar una función que tenía que modificar alguna de las variables introducidas como argumento, era necesario pasar la dirección física de la variable en vez de su valor. El clásico ejemplo que se suele utilizar es el de la función permutar, que lo que tiene que realizar es intercambiar el contenido de dos variables. El código en C de esta función, así como su aplicación podría ser el siguiente:

```
void permutar(int *a,int *b)
{
int c;
c=*a;
*a=*b;
*b=c;
}
void main()
{
int x=1,y=2;
printf("contenido de x e y: %d, %d\n",x,y); /*imprime 1, 2 */
permutar(&x,&y); /*llamada a la función con la dirección de
                   las variables x e y*/
printf("contenido de x e y: %d, %d\n",x,y); /*imprime 2, 1 */
}
```

Aunque desde el punto de vista del modo de funcionamiento real del ordenador, es mucho más correcto el modo en que este código se escribe en C, en C++ las referencias permiten realizar esta operación mediante la creación de alias de los argumentos actuales, consiguiendo desde el punto de vista del programador una notación mucho más cómoda:

```
void permutar(int &a, int &b)
{
int c;
c=a;
```

```
a=b;
b=c;
}
void main()
{
int x=1,y=2;
printf("contenido de x e y: %d, %d\n",x,y); //imprime 1, 2
permutar(x,y);      //llamada a la función directamente con
                    //las variables x e y
printf("contenido de x e y: %d, %d\n",x,y); /*imprime 2, 1 */
}
```

Lo que ocurre es que indicamos que las referencias argumentos formales de la función se inicializan con los argumentos actuales de la llamada a la función, por lo que **las referencias pasan a ser las mismas variables pero con un nombre y ámbito local a la función**. El efecto es equivalente a pasar estos argumentos por variable en vez de por valor.

Evidentemente, para los programadores aún no experimentados esto es un soplo de aire fresco. Sin embargo hay que tener cierta precaución en el uso de las referencias, puesto que se están utilizando indistintamente variables de un ámbito superior a una función y variables propias de la función como si fueran iguales. Esto puede provocar peligrosas confusiones.

Internamente, el compilador transforma las llamadas a funciones con parámetros por referencia a la utilización de punteros y contenidos propia de C. Por este motivo, las reglas que llevaban a utilizar punteros en vez de valores en C se siguen aplicando para el caso de las referencias. Es decir, en el caso de que estemos manejando variables de gran tamaño (estruturas o clases de tamaño mediano o grande), es conveniente utilizar bien el paso por dirección o por referencia, puesto que el código es mucho más eficiente.

La referencia como valor de retorno

Cuando en C se establece el tipo del valor de retorno de una función, lo que realmente estamos diciendo es de que tipo es la función, o a que tipo de datos es equivalente una llamada a una función. Esto permitía introducir la llamada a una función en cualquier sitio en donde se acepte un valor del tipo de datos de dicha función. El siguiente ejemplo muestra esta idea:

```
struct vector2D
{
double x,y;
};
```

```
vector2D *normaliza(vector2D *a)
{
double modulo;
modulo=sqrt((a->x)*(a->x)+(a->y)*(a->y));
a->x/=modulo;
a->y/=modulo;
return a;
}
void main()
{
vector2D vector={3.0,4.0};
printf("la x normalizada es %d",normaliza(&vector)->x);
}
```

Pues bien, en C++ se pueden utilizar las referencias como valor de retorno para realizar anidaciones de este tipo con notación más cómoda, o incluso ¡para poder realizar asignaciones a una variable retornada por referencia tras la llamada de la función!. Esto ya comienza a sonar a chino para un experto programador de C, pero la idea es la misma que antes… con una referencia podemos utilizar un nombre distinto para una variable. Ahora ese nombre distinto será la llamada a la función.

Se muestra a continuación las dos aplicaciones más inmediatas utilizando la estructura vector anteriormente definida: la concatenación de operaciones, y la asignación de valores a las variables retornadas por referencia por una función:

```
vector2D& normaliza(vector2D& a)
{
double modulo;
modulo=sqrt((a.x)*(a.x)+(a.y)*(a.y));
a.x/=modulo;
a.y/=modulo;
return a;
}

double& componenteX(vector2D& a)
{
return (a.x);
}

void main()
{
vector2D vector={3.0,4.0};
printf("la x normalizada es %d",normaliza(vector).x);
componenteX(vector)=8.0;
```

```
...
}
```

Nótese que gracias a las referencias, ahora es posible utilizar la llamada a una función en el lado izquierdo de una asignación. No asignamos a la función el valor situado a la derecha, sino que asignamos dicho valor a la variable retornada por referencia por la función, puesto que la expresión de llamar a la función se ha convertido en un *alias* de la variable retornada.

Es importante considerar que sólo es posible retornar referencias a variables cuya vida supere a la duración de la función, por el mismo motivo que en C no se retornan punteros a variables declaradas en el interior de la función: cuando se termina la ejecución de la función, las variables **auto** definidas en su interior son destruidas.

El siguiente código sería por tanto erróneo:

```
vector2D& normaliza(vector2D a)
{
double modulo;
modulo=sqrt((a.x)*(a.x)+(a.y)*(a.y));
a.x/=modulo;
a.y/=modulo;
return a;
}
void main()
{
vector2D vector={3.0,4.0;
printf("la x normalizada es %d",normaliza(vector).x);
}
```

El compilador generaría el siguiente error:

"Returning address of local variable or temporary"

Esto es debido a que los argumentos formales son variables locales de la función inicializadas con el valor de los argumentos actuales. Como consecuencia, la variable que se retorna por referencia es una variable local copia de la introducida como parámetro.

Por último, cabe destacar en el mensaje de error, que se señala que se está retornando la dirección de una variable local. Hasta este punto se ha evitado mencionar la relación existente entre punteros y referencias para evitar confusiones en su uso. Sin embargo hay que destacar que en el fondo las referencias no son más que triquiñuelas del compilador para hacer el código más legible, siendo en verdad direcciones de las variables (es decir punteros) lo que se pasa tanto en los argumentos como en el retorno.

2.10. Reserva dinámica de memoria: operadores new y delete.

Cualquier programador de C sabe que existe una diferencia cualitativa importante en el momento en que se comienza a programar con variables creadas dinámicamente. Es decir, lugares de la memoria para almacenar datos que son solicitados al sistema operativo durante la ejecución de un programa en función de las necesidades del mismo. Puesto que la memoria es uno de los principales recursos gestionados por el sistema operativo, para obtener estas variables durante la ejecución, en ANSI C era necesario realizar la petición generalmente por medio de la función **malloc** definida en la librería estándar **stdlib**.

Esta memoria solicitada durante la ejecución, puesto que había sido directamente solicitada al sistema, permanecía reservada para su uso por parte del programa de forma indefinida. Si se quería dejar libre una zona anteriormente reservada, entonces había que realizar la correspondiente petición al sistema operativo por medio de la función **free** también definida en **stdlib**.

Existen muchas aplicaciones directas en la reserva dinámica de memoria, tales como la generación de listas, pilas, árboles, vectores de tamaño variable, etc. Simplemente para ayudar a recordar estas ideas, se expone a continuación un programa en C que permite almacenar tantos números distintos de cero como quepan en la memoria (realmente 50% de la memoria libre que actualmente es mucho) por medio de un vector de caracteres de tamaño variable:

```
#include <stdlib.h>
#include <stdio.h>
#include <memory.h> /*necesario para usar memcpy*/
int main()
{
long tamanio=0,capacidad=10,i;
int valor;
int *vector,*aux;
vector=(int *)malloc(capacidad*sizeof(int));
while(scanf("%d",&valor)&&(valor!=0))
      {
      vector[tamanio++]=valor;
      if(tamanio==capacidad)
            {
            capacidad+=10;
            aux=(int *)malloc(capacidad*sizeof(int));
            if(aux==NULL)
                  {
                  printf("Error en la reserva de memoria");
                  free(vector);
                  return -1;
```

```
                    }
                memcpy(aux,vector,tamanio*sizeof(int));
                free(vector);
                vector=aux;
                }
        }
printf("número de números almacenado: %d\n",tamanio);
for(i=0;i<tamanio;i++)printf("%d, ",vector[i]);
free(vector);
return 1;
}
```

Ciertamente este ejemplo no es muy didáctico, puesto que se ha optado por una programación compacta. La finalidad del mismo es la de concienciar al lector de que a partir de ahora se requerirán al menos los conocimientos de C reflejados en este ejercicio. Si no fuera el caso, se recomienda repasar C antes de continuar. Por tanto:

> No es conveniente seguir sin tener los conocimientos de C necesarios para comprender el ejercicio anterior.

En C++ se introducen dos operadores más completos que realizan la reserva y liberación de memoria dinámica: los operadores **new** y **delete**.

El operador new.

El operador **new** es semejante a la función **malloc** aunque como se verá a lo largo del curso no se limita exclusivamente a realizar la reserva de memoria. Por ahora se considerará que permite asignar memoria perteneciente al área de almacenamiento libre para un objeto o para una matriz de objetos –de momento entiéndase por objeto una variable.

Cuando se reserva memoria para un solo objeto, el tamaño de memoria necesario para realizar dicho almacenamiento se determina directamente en función del tipo de dicho objeto. Por este motivo, la sintaxis es la siguiente:

```
int *a,*b;
a=new int;
b=new (int);
```

El ejemplo muestra dos posibles sintaxis para la misma operación. Ambas son equivalentes. Es posible, como se ha visto, realizar la declaración de un puntero y la reserva de memoria en la misma sentencia:

```
int *a = new int;
```

La llamada al operador **new** devuelve un puntero al espacio de memoria reservado, siendo el puntero del tipo especificado. Por este motivo, a diferencia de la función **malloc** no es necesario realizar el **cast** (conversión forzada) al puntero retornado. Si no hubiera espacio para la reserva de memoria, el operador devuelve un puntero NULL si no se utilizan las excepciones (se verá más adelante).

Cuando el operador new reserva memoria para una matriz, el tipo de datos que devuelve es del tipo de la dirección al primer elemento de la matriz. Se suele considerar por este motivo como un operador distinto, aunque en el fondo es el mismo modo de funcionamiento que en la instrucción **malloc** de C.

Así, los siguientes ejemplos muestran como crear vectores y matrices dinámicamente con el operador **new**:

```
1   int num=6,i;
2   int *a = new int;
3   int *b = new int[num];
4   int (*c)[3] = new int[8][3];
5   int *d[3];
6   for(i=0;i<3;i++)d[i]=new int[8];
```

En este ejemplo se ha establecido la comparación entre la reserva de memoria para un solo objeto o variable (línea 2), y la reserva de memoria de vectores. En el caso de los vectores el modo de proceder es exactamente igual que en C. Un vector al final es un puntero que apunta a una zona de memoria en donde comienzan a situarse los elementos del vector de forma ordenada. De tal forma, que la dirección corresponde con la del primer elemento y tiene por tanto este tipo. Recordemos, que la razón principal para proceder así, esta en la aritmética de punteros y en los sistemas de indirección, logrando de esta forma que uno de los mecanismos básicos del lenguaje tenga correspondencia directa con el modo de funcionamiento de un microprocesador.

Obsérvese a su vez como al igual que en C internamente no existen matrices de elementos de más de una dimensión, sino que todo se considera como vector. En el caso de una matriz de dos dimensiones en C y C++ se consideran como un vector de vectores. Esto es lo que aparece reflejado en la línea 4. Aunque a la derecha la petición aparece con el aspecto de una matriz de 8 por 3, lo que entiende C (y eso es lo que se refleja a la izquierda) es que se está solicitando un vector de vectores de tres enteros, por lo que la dirección de retorno es la de vector de tres enteros. Este modo de lectura, ampliamente recomendado, se logra si se siguen las reglas de asociatividad de los corchetes. El operador new, considera sólo los primeros corchetes, lo demás corresponde al tipo básico del vector.

Puesto que los corchetes tienen preferencia sobre el operador *, es necesario poner los paréntesis, para que el compilador entienda que c –línea 4- es un puntero (primero se lee lo que hay entre paréntesis) a vectores de tres enteros.

Por el contrario, en la línea 5, el compilador entiende que d es un vector –tienen preferencia los corchetes frente al asterisco- de tres punteros a entero. Que evidentemente es distinto al caso anterior.

Para finalizar este operador al que más adelante se irá haciendo referencia, se comentarán para darle una forma más completa, los aspectos que lo hacen diferente y mejor a la función estándar **malloc** de C. Las dos últimas características, corresponden a aspectos más relacionados con la POO:

-El operador **new** no necesita de la conversión forzada.

- En la generación de un objeto, se realiza una llamada al constructor, e incluso se permite la inicialización.

- La gestión de errores es mucho mejor, puesto que lanza una excepción (bad_alloc) cuando hay error, o incluso permite la gestión de los errores de reserva de memoria por medio de la función **set_new_handle**.

El operador delete

El operador **delete** destruye un objeto creado por el operador **new**, liuberando el sistema operativo la memoria ocupada por dicho objeto. A diferencia de lo que ocurría en C con la función free, **delete** puede ser utilizado sobre un puntero nulo (apunta a cero) en cuyo caso no realiza ninguna operación. Es importante recordar que nunca se puede pedir la liberación de una memoria que no ha sido reservada dinámicamente. O como regla práctica,

> El operador delete solo puede ser aplicado a zonas de memoria reservadas mediante un new.

A diferencia de C, en caso de querer eliminar un vector, se utilizara una sintaxis distinta del operador **delete**. La razón de que se proceda de esta forma viene del uso o no de los destructores del tipo de datos básico –ya se verá el significado de destructor. En el caso de los vectores, se realiza la llamada al destructor del tipo de datos del puntero por cada uno de los elementos eliminados. Evidentemente para el caso de un solo objeto también se realiza la llamada. Es importante considerar que el valor de la variable puntero utilizado para eliminar el/los objetos, no es modificado. De forma que lo habitual es que una vez realizada la llamada a *delete* se proceda a continuación a asignar cero al puntero utilizado.

Según lo indicado, las operaciones que hay que realizar para eliminar los objetos reservados para el anterior ejemplo son:

```
1   delete a;
3   delete [] b;
4   delete [] c; //observese que se considera c como vector
5   for(i=0;i<3;i++)delete [] d[i]; //para cada new un delete
```

Donde se observa en la línea 1 el modo en el que **delete** es utilizado en la eliminación de un solo objeto. Y en las líneas siguientes, el formato en la llamada a **delete** para proceder a la liberación de la memoria reservada para distintos vectores de elementos.

2.11. Espacios de nombres.

Un espacio de nombres (*namespace)* es un concepto sencillo, similar al de ámbito. La idea es que para evitar conflictos de nombres durante la fase de enlazado, se puedan definir subconjuntos o ámbitos de identificadores. Un espacio de nombres, como indica su denominación, es una zona separada donde se pueden declarar y definir objetos, funciones y en general, cualquier identicador de tipo, clase, estructura, etc; al que se asigna un nombre o identificador propio. Hasta ahora, en nuestros programas, siempre habíamos declarado y definido nuestros identificadores fuera de cualquier espacio con nombre, en lo que se denomina el espacio global. Este mecanismo permite reutilizar el código en forma de librerías, que de otro modo no podría usarse. Es frecuente que diferentes diseñadores de librerías usen los mismos nombres para cosas diferentes, de modo que resulta imposible integrar estas en la misma aplicación.

Por ejemplo un diseñador crea una biblioteca matemática con una clase llamada "Conjunto" y otro una biblioteca gráfica que también contenga una clase con ese nombre. Si nuestra aplicación incluye las dos bibliotecas, obtendremos un error al intentar declarar dos clases con el mismo nombre.

Para entender el concepto, supongamos que queremos crear una variable llamada "Pepe", y que el código que estamos integrando con más gente también contiene posiblemente la variable "Pepe" distinta. Así que lo que decidimos es que yo voy a utilizar "Pepe" del ámbito "Almería" y mi compañero "Pepe" del ámbito de "Toledo". Dentro de un ámbito, siempre que indique "Pepe" se hará referencia implícita al "Pepe" de dicho ámbito, pero desde fuera siempre se podrá decir "Pepe el de Almería" o "Pepe el de Toledo".

Además se puede indicar, por medio de la palabra clave *using* el uso de un ámbito externo por defecto. Esto nos permite subdividir el espacio global de nombres en espacios personalizados evitando las redefiniciones o el agotamiento de identificadores en proyectos grandes.

Por último, es interesante aclarar que se pueden añadir elementos a un espacio de nombres en cualquier parte, y en muchos ficheros distintos. Lo habitual es que una librería en C++ por ejemplo tenga todos sus elementos definidos en un namespace específico.

```
namespace Almeria{
  int Pepe;
  void foo(){
    Pepe=73;
  }
}
namespace Toledo{
  int Pepe;
}
void foo(){
  Almeria::Pepe=3;
  Toledo::Pepe=5;
}

void foo2()
{
  using namespace Almeria;
  Pepe=8;
  Toledo::Pepe=3;
  Almeria::foo(); //hay foo() y Almería::foo() es necesario
}
```

Es posible crear alias de un espacio de nombre, para evitar tener que usar una notación excesivamente larga e incómoda:

```
namespace <alias_de_espacio> = <nombre_de_espacio>;
```

Por ejemplo:

```
namespace nombredemasiadolargoycomplicado {
...
declaraciones
...
}
...
namespace ndlyc = nombredemasiadolargoycomplicado; // Alias
...
```

Espacios de nombre anónimos. Si nos fijamos en la sintaxis de la definición de un espacio con nombre, vemos que el nombre es opcional, es decir, podemos crear espacios con nombre anónimos.

Pero, ¿para qué crear un espacio anónimo? Su uso es útil para crear identificadores accesibles sólo en determinadas zonas del código. Por ejemplo, si creamos una variable en uno de estos espacios en un fichero fuente concreto, la variable sólo será accesible desde ese

punto hasta el final del fichero. Este mecanismo nos permite restringir el ámbito de objetos y funciones a un fichero determinado dentro de un proyecto con varios ficheros fuente. Si recordamos el capítulo anterior, esto se podía hacer con el especificador static aplicado a funciones o a objetos globales.

De hecho, la especificación de C++ aconseja usar espacios con nombre anónimos para esto, en lugar del especificador static, con el fin de evitar la confusión que puede producir este doble uso del especificador.

Por último, es interesante aclarar que se pueden añadir elementos a un espacio de nombres en cualquier parte, y en muchos ficheros distintos, y que no hay problema en anidar espacios de nombres dentro de otros espacions de nombre. Lo habitual es que una librería en C++ por ejemplo tenga todos sus elementos definidos en un namespace específico.

2.12. Operaciones de entrada y salida

En la mayoría de los programas expuestos anteriormente se ha hecho uso de las funciones **printf** y **scanf** ya utilizadas en C. Mediante dichas funciones –perfectamente utilizables en C++- logramos que el programa envíe y reciba datos de periféricos del sistema. (la pantalla y el teclado). En C++ esta operación se simplifica y generaliza introduciéndose el concepto de flujo. Es decir, dado que es muy habitual tener que realizar operaciones de transmisión y recepción de datos desde o hacia ficheros, impresoras, escáneres, sitios de internet, etc. Lo que se ha hecho es unificar y simplificar estas comunicaciones por medio de lo que se llaman flujos. Así, un flujo de salida sería el envío de datos desde el programa hacia la pantalla (mostrar un mensaje o unos valores) y un flujo de entrada puede ser el conjunto de teclas pulsadas en un teclado.

Por tanto, un flujo es un objeto que hace de intermedio entre el programa y el destino u origen de la información. En la mayoría de los casos no es relevante para el programa la naturaleza física del sistema que envía o recibe los datos, sino que lo que interesa es el carácter secuencial de los datos enviados o recibidos.

De hecho, al igual que en C, las operaciones de entrada y salida (lectura y escritura) siguen en general siempre el mismo esquema: inicialmente se abre el canal de comunicación o flujo (apertura de un fichero o un puerto), se envía o recibe información durante el tiempo que sea necesario, y finalmente, cuando ya no es necesario este canal, se cierra para no sobrecargar al sistema operativo.

Se puede objetar a lo anterior que para el uso de **printf** no ha sido nunca necesario realizar la apertura del canal. Es cierto, pero es que en C, todas las operaciones de entrada y salida estándar se realizan por medio de ficheros que en algunos casos están ocultos. Al comenzar cualquier programa en C se produce la apertura automática de tres ficheros: **stdin,**

stdout, y **stderr**, que constituyen respectivamente la entrada estándar de datos, la salida estándar y la salida para mensajes de error estándar. De forma que es el sistema el que realiza la apertura de ficheros por nosotros. Cuando uno escribe una intrucción **printf(…)** realmente está escribiendo **fprintf(stdout,…).**

Volviendo entonces a C++, el comportamiento es análogo, pero mucho más cómodo. Cuando un programa en C++ se ejecuta, se crean automáticamente tres flujos identificados por los siguientes objetos:

1. Un flujo de entrada estándar (normalmente el teclado): **cin.**
2. Un flujo de salida estándar (habitualmente la pantalla): **cout.**
3. Dos flujos hacia la salida estándar de error (pantalla): **cerr,** y **clog**.

Aunque según avance el curso, se irán viendo aspectos más avanzados sobre los flujos de datos, a continuación se muestra el modo en que se realizan las operaciones de entrada y salida en C++.

Los flujos cin y cout.

En primer lugar, para poder utilizar los flujos anteriormente expuestos, es necesario incluir la librería estándar **iostream.h** .

La gran ventaja o comodidad del uso de flujos, reside en que cada dato sabe inicialmente como se debe transmitir (es decir, el juego de caracteres que lo representa), y en caso de ser un tipo de datos definido por el usuario, es posible indicar como se realiza su escritura o su lectura gracias a la posibilidad de sobrecargar los operadores **>>** y **<<**.

El siguiente ejemplo ilustra como se imprimen una serie de datos por pantalla mediante el uso del flujo estándar **cout**.

```
#include<iostream.h>
int main()
{
int i=2,j=3;
double dato=5.3;
char a='a',b[]="hola";

cout << i;
cout << i << j << endl;
cout << "el valor de dato es " << dato << endl;
cout << "el carácter a=" << a << "y la cadena b" << b << endl;
}
```

La primera línea indica que se imprima *i* siendo el compilador el encargado de convertir el valor de *i* a su representación como una cadena de caracteres. Es decir, ya no es necesario indicar el formato con el que un dato debe ser impreso, sino que el propio sistema puede hacerlo atendiendo al tipo de ese dato.

Si se desean escribir varios datos seguidos, bastará con concatenar las operaciones una detrás de otra tal y como aprece reflejado en las líneas siguientes. El identificador endl es equivalente a imprimir por pantalla el carácter '\n', es decir un cambio de línea.

Si por cualquier motivo se desease imprimir un dato con un formato distinto al de su tipo, bastará con realizar una conversión forzada al formato con el que se desea imprimit:

```
cout << static_cast<int>('A');
```

En este caso se imprime el valor entero correspondiente al character *A*, es decir su código ASCII.

Al operador **<<** se le denomina como operador *inserción*.

Las operaciones de entrada se realizan de forma análoga, pero cambiando **cout** por **cin** y el operador **<<** por el operador **>>**. Al operador >> se le denomina como operador de extracción. El modo de funcionamiento es equivalente al de **scanf** pero en este caso no es necesario ni la especificación de tipos ni el paso de direcciones:

```
double b;
int i;
char c, d[50];
cout << "introduzca un número real :";
cin >> b; //al teclear 3.5, en b se almacena este valor
cout << "el valor introducido es: " << b << endl;
cout << "introduzca un real seguido de un entero y un char:";
cin >> b >> i >> c;
cout << "introduzca su nombre:";
cin >> d;
```

Al igual que en **scanf** la separación entre cada valor introducido se realiza por medio de el espacio en blanco según el lenguaje C (espacio, tabulador, cambio de línea, …).

Un aspecto importante que se abordará en el capítulo correspondiente es el de la gestión de errores y el estado de los flujos, así como la especificación de formatos (por ejemplo, el número de decimales que se utilizarán en la impresión de un **double**). De momento se mencionará que son operaciones fácilmente realizables pero que escapan al enfoque de este capítulo.

Es interesante considerar que la operación de insertar en un flujo tanto si es una pantalla como si es un fichero es la misma. Aunque ya se verá a lo largo del libro, el acceso a

estos elementos está realizado mediante el uso de una clase base común, por lo que será realmente equivalente hacer una operación u otra salvo que especifiquemos lo contrario. A continuación de modo ilustrativo se incluye un breve ejemplo de escritura de un texto en un fichero de datos. No se realizan las comprobaciones oportunas ni se cierra el fichero para ilustrar la equivalencia. En principio, el fichero al destruir su objeto, cerrará correctamente el recurso del sistema (pero lo mas correcto es comprobar la apertura e indicar el cierre).

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{
	ofstream miFichero("example.txt");
	miFichero<<"Hola Mundo!"<<endl;
}
```

2.13. EJERCICIOS

EJERCICIO 2.1

Dada la siguiente estructura de datos:

```
**********************************/
#define MAX_CADENA 100

//valores posibles de las cosas
enum TipoCosa {T_complejo, T_libro};

//datos para un complejo
struct Complejo{
   double real;
   double imag;
   };
//datos para un libro
struct Libro{
   char Titulo[MAX_CADENA];
   char Autor[MAX_CADENA];
   };
//datos para cualquier cosa
struct Cosa{
   TipoCosa tipo;
   union{
         Complejo complejo;
         Libro libro;
```

```
        };
};
```

Escribir un programa que permita almacenar una lista de hasta diez cosas, tanto libros como complejos. A parte de introducir los datos, el programa permite imprimir tanto la lista de complejos, como la lista de libros o la lista de cosas en el mismo orden en que fueron introducidas.

EJERCICIO 2.2

Diseñar e implementar una función de nombre **norma** que permita obtener la norma $\sqrt[2]{\sum_{i=1}^{n} x_i^2}$ de un vector de *float* de dimensión *n* y que en función del parámetro adicional *bnormaliza* de tipo *bool* permita normalizar (*true)* o no (*false)* el vector entrante. Además por defecto, se considerará que los vectores son de dimensión 3 y que no se desea modificar el vector introducido.

EJERCICIO 2.3

Una función contiene las siguientes sentencias de C++. Indíquese la impresión por pantalla que dichas instrucciones provocan.

```
int a,b,c=1;
a=3,2+3;
b=(3,2+3);
c+=c+=2;
printf("a=%d  b=%d c=%d",a,b,c);
```

EJERCICIO 2.4

Indíquese la impresión por pantalla del siguiente programa:

```
#include <stdio.h>
void main()
{

int i=0;
for(i=0;i<30;i++){
   if(!(i%2)||!(i%3))continue;
   printf("%d ", i);
   }
}
```

EJERCICIO 2.5

Indíque cual de las siguientes líneas marcadas con un rectánculo es correcta (el compilador la acepta):

```
enum notas {suspenso, aprobado, bien, notable, sobresaliente};

int var;
notas nota1, nota2;
```

☐ `nota1=bien;`

☐ `nota1=1;`

☐ `nota1=(notas)1;`

☐ `nota1=suspenso+1;`

☐ `nota1=suspenso+aprobado;`

☐ `var=5+bien;`

☐ `var=bien+5;`

☐ `var=notable+bien;`

☐ `var=nota1+bien;`

☐ `nota1++;`

☐
```
for(var=suspenso;var<sobresaliente;var++)
printf("%d",var);
```

☐
```
for(nota2=suspenso;nota2<sobresaliente;nota2++)
printf("%d",(int)nota2);
```

EJERCICIO 2.6

Indique la impresión por pantalla del código siguiente, si la secuencia de números introducida por el usuario es la siguiente: 12, 13, 14, 15, 16, 17, 18, 12, 13 y 14.

```
#include "stdio.h"

int &menor(int *vector, int num){
   int j=0;
   for(int i=0;i<num;i++)if(vector[i]<vector[j])j=I;
   return vector[j];
   }
void main(){
   int lista[3]={0,0,0};
   for(int i=0;i<10;i++)scanf("%d",&menor(lista,3));
   printf("Valores: %d %d %d",lista[0],lista[1],lista[2]);
   }
```

3. El Concepto de Clase

Desde el comienzo de estos apuntes, se ha destacado la importancia del cambio de mentalidad o filosofía a la hora de programar que introduce el lenguaje C++. Sin embargo, hasta este capítulo, apenas se ha realizado ninguna variación sobre el modo de programar de C. Bien, la parte introductoria o de situación se terminó en el capítulo anterior y por tanto, con el concepto de clase, objeto y encapsulamiento se comienza a trabajar en la idea de la Programación Orientada a Objetos, precisamente con la introducción del elemento básico… el objeto.

Como ya se había comentado en el capítulo introductorio un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Hasta ahora se han realizado programas en los que los datos y las funciones estaban perfectamente separados. Cuando se programa con objetos esto no es así, cada objeto contiene datos y funciones. Y un programa se construye como un conjunto de objetos, o incluso como un único objeto. Los objetos se agrupaban en clases, de la misma forma que en la realidad, existiendo muchas mesas distintas (la de mi cuarto, la del comedor, la del

laboratorio, la del compañero o la mía) agrupamos todos esos objetos reales en un concepto más abstracto denominado mesa. De tal forma que podemos decir que un objeto específico es una realización o instancia de una determinada clase.

Por ejemplo, en los programas de Windows, cada botón clásico de una interfaz es un objeto. Todos los botones son **instancias** o realizaciones de la clase **CButton** de Windows. Luego lo que realmente existe son los objetos, mientras que las clases se pueden considerar como patrones para crear objetos. Pero además, estos botones pertenecen a un objeto más grande que es el tapiz sobre el que están pintados. Este objeto de windows habitualmente no es más que una instancia de la clase **CDialog** que permite crear objetos que contienen objetos que son capaces de interaccionar con el usuario de forma gráfica. Muchas de las interfaces que estamos acostumbrados a menejar en windows son instancias de esta clase de objetos.

¿Por qué se trabaja de esta forma?¿Que ventajas tiene?... al programar basándose en objetos, es posible utilizar con facilidad el código realizado por otras personas. Así, cuando creamos un botón en Windows prácticamente lo único de lo que nos tenemos que preocupar es de indicar el título o texto que aparece en el centro del botón. Sin embargo, toda la gestión de dibujar el rectángulo, la sombra, de reaccionar ante un clic del ratón, etc… no es necesario que lo programemos puesto que ya lo ha hecho alguien por nosotros al constituir características comunes a todos los botones.

3.1. Las clases en C++

En el fondo, se puede decir que una clase en C++ es un mecanismo que permite al programador definir sus propios tipos de objetos. Un ejemplo clásico es el de los números complejos. Es bien conocido que en C no existe un tipo de datos para definir variables de tipo complejo. Por ello lo que se hacía era definir estructuras que contuviesen un campo que representara la parte real como un número de tipo float, y una parte imaginaria también de tipo float que representara la parte imaginaria. De esta forma se podía agrupar bajo un mismo identificador el conjunto de datos necesarios para definir un complejo.

En C, esto se realizaba por medio del uso de estructuras con estos dos campos:

```
typedef struct _complex{
      float real;
      float imag;
      }complex;
void main()
{
complex a;
a.real=5.0F;
a.imag=3.0F;
}
```

Posteriormente, si se quisiera trabajar cómodamente con este nuevo tipo de datos se definían una serie de funciones pensadas para manipular o extraer información de los mismos. Así, sería posible definir una función que permitiera extraer el módulo de cualquier complejo mediante el siguiente código:

```
float moduloComplejo(complex a)
{
float m;
m=sqrt(a.real*a.real+a.imag*a.imag);
return m;
}
```

Se observa que mediante este sistema por un lado están los datos y por otro las funciones, aunque estas funciones estén definidas exclusivamente para trabajar con el tipo de datos complex.

Las clases en C++ se pueden considerar como la evolución de las estructuras. Las clases permiten no sólo agrupar los datos –como ocurre en las estructuras- sino que además nos permiten incluir las funciones que operan con estos datos.

Las clases en C++ tienen los siguientes elementos o atributos:

- Un conjunto de datos miembro. En el caso de los complejos, el número real que representa la parte real y el número real que representa la parte imaginaria, serán datos miembro de la clase complex. La información guardada en estos datos son lo que harán distinto un objeto de otro dentro de una misma clase. Aunque normalmente siempre tenemos datos en una clase, no es necesario que existan.
- Un conjunto de métodos o funciones miembro. Como se ha mencionado antes, serán un conjunto de funciones que operarán con objetos de la clase. Puede haber desde cero hasta tantos métodos como se consideren necesarios.
- Unos niveles de acceso a los datos y métodos de la clase. Supongamos que se ha creado una clase que permite almacenar un conjunto de números, de tal forma que es como una especie de saco, en donde se van agregando números, y después se pueden ir extrayendo de uno en uno o se puede preguntar cuantos números hay almacenados. En la clase existirán por tanto un conjunto de datos que permitirá ir almacenando los números que se introducen (*data*) y un dato que nos permite conocer cuántos números se han introducido hasta ese momento (*num*). No sería conveniente que en esta clase, el programador pudiera modificar libremente el valor de *num*,

puesto que este valor indica el número de objetos que se han introducido y no un valor arbitrario. Por ello en C++ se puede establecer que algunos de los datos y funciones miembro de una clase no puedan ser utilizados por el programador, sino que están protegidos o son datos privados de la clase ya que sirven para su funcionamiento interno, mientras que otros son públicos o totalmente accesibles al programador. Esta cualidad aplicable a los datos o a las funciones miembro es lo que se denomina como nivel de acceso.

- Un identificador o nombre asociado a la clase. Al igual que en las estructuras, será necesario asignar un nombre genérico a la clase para poder hacer referencia a ella. Las palabras **CButton, CDialog, ó complex** utilizadas en los ejemplos anteriores serán los identificadores de las clases que permiten crear objetos de tipo botón, diálogo o complejo.

3.2. Definición de una clase

Esbozado el concepto de lo que se entiende en C++ por clase, se procede ahora a establecer la forma con que se crean las clases, y se utilizan.

Inicialmente consideraremos que existen tres momentos en la definición y utilización de una clase: la declaración, la definición y la instanciación. Es posible realizar la declaración y la definición en el mismo segmento de código, sin embargo de momento se realizarán siempre de forma separada para facilitar su comprensión.

La declaración de una clase lo constituye la descripción de los datos y funciones que pertenecen a la clase. En la declaración no es necesario indicar el código correspondiente a los métodos de la clase, sino que basta con indicar el nombre del método y sus parámetros de entrada y salida. El concepto es análogo al de la declaración de funciones en C, en donde lo que interesa es comunicar al compilador la existencia de una función, el modo en que esta se debe usar (argumentos y valor de retorno), y no se escribe el código.

La definición de una clase, es la parte o partes del programa que describen el código contenido en los métodos de la clase. La idea es análoga a la definición de las funciones en C.

La instanciación, es el proceso por el cual se crea un objeto de una clase.

Estos tres conceptos quedan ilustrados en una primera versión y utilización de la clase *complex* para los objetos que representan números complejos.

```
//comienzo de la declaración
class complex
 {
private:
 double real;
 double imag;
public:
 void estableceValor(float re,float im) ;
 float obtenModulo(void) ;
 void imprime() ;
};
//fin de la declaración

//comienzo de las definiciones
void complex::estableceValor(double re, double im)
{
real=re ;
imag=im ;
}
double complex::obtenModulo(void)
{
return (sqrt(real*real+imag*imag)) ;
}
void complex::imprime(void)
{
cout<<real<<`+`<<imag<<`i` ;
}
//fin de las definiciones

//comienzo del programa de ejemplo
void main()
{
complex micomplejo; //creación de un objeto de la clase complex
//asigno a micomplejo el valor 3.2+1.8i
micomplejo.estableceValor(3.2,1.8);
//imprimo el modulo
cout<<"El modulo de ";
micomplejo.imprime();
cout<<" es  "<<micomplejo.obtenModulo();
}
```

Se verá con detalle el significado que tienen todas estas nuevas palabras y porque se usan de la forma en la que se han utilizado:

Declaración de la clase

La sintaxis más simple de declaración de una clase es igual que la utilizada para la declaración de las estructuras en C, pero cambiando la palabra clave **struct** por **class** y con la posibilidad de introducir funciones y niveles de acceso entre las llaves:

```
class <identificador>
 {
 [<nivel de acceso a>:]
 <lista de miembros de la clase>
 [<nivel de acceso b>:
 <lista de miembros de la clase>]
 [<…>]
 }[lista de objetos];
```

Son múltiples las posibilidades en la declaración de una clase, sin embargo de momento se comienza con el formato más sencillo. Al igual que en las estructuras es posible la creación de objetos de una clase en su misma declaración, aunque no es lo más habitual. Sin embargo hay que destacar que esta posibilidad nos obliga a introducir un punto y coma (;) al final de la declaración de la clase tanto si se crean objetos como si no, en contra de lo que ocurre habitualmente con las llaves en C.

La primera línea establece el nombre con el que vamos a identificar la clase por medio de la palabra introducida en el campo <identificador>. En el ejemplo, el identificador es la palabra complex.

Una vez establecido el nombre se procede a indicar los datos y los métodos de la clase. Aunque no existe una especificación en el orden en que los datos y funciones deben ser declarados, es habitual proceder a introducir en primer lugar los datos miembro con sus niveles de acceso y después el conjunto de métodos tanto públicos como privados.

En el ejemplo se indica en primer lugar que los elementos que se van a declarar posteriormente serán de carácter privado. Esto de ha realizado por medio de la palabra clave **private**. De esta forma, los datos *real* e *imag* de tipo **double** sólo podrán ser modificados y consultados por medio de métodos de la clase. Veremos con más detalle la utilidad de todo esto al hablar del concepto de encapsulamiento en el siguiente apartado.

Después se indica que a partir del punto en el que parece la palabra **public** se declaran las funciones y datos miembro de la clase que son accesibles desde fuera de la clase, y que por tanto constituyen la interfaz. En el ejemplo, tres funciones tienen este nivel de acceso: *estableceValor, obtenModulo* e *imprime*. Con esto se finaliza la declaración de la clase por lo que se procede a definirla.

Definición o implementación de una clase

Al igual que ocurre con los prototipos en C, en la declaración sólo hemos informado al compilador sobre el modo de uso y especio necesario para el tipo de objetos definidos por la clase. En la definición se procederá a explicitar la funcionalidad de cada uno de los métodos por medio de código C++.

La definición puede situarse en un fichero distinto –de hecho es lo más habitual- a la declaración de la clase. Incluso es posible que varias definiciones de distintas clases estén en un mismo fichero… para el compilador es indiferente. Sin embargo, es importante indicar cuando se define el contenido de un método de una clase, la clase a la que se hace referencia. Para definir un método fuera del cuerpo de declaración de una clase, es necesario indicar siempre a qué clase pertenece dicho método. Para ello hay que especificar el nombre de la clase antes del nombre del método, separado del mismo por medio del operador de ámbito (**::**).

La sintaxis será la siguiente:

```
<tipo> <iden_clase>::<iden_metodo>(<argumentos>)
 {
 [código del método]
 }
```

Se observa que ya no es necesario indicar si el método es público o privado, puesto que esta información se había establecido ya en la declaración.

También hay que destacar que los métodos de una clase, acceden directamente a los datos y métodos de la propia clase, mientras que desde *fuera* como en el caso de la función *main*, es necesario utilizar el operador "**.**" para poder ejecutarlos.

El puntero implícito this.

Lo que hace distinto a dos objetos de una misma clase es precisamente los datos que contienen. Por ello, los datos miembro o atributos de un objeto son distintos para cada objeto, sin embargo las funciones o métodos son comunes a todos los objetos de una misma clase. Es decir, cada objeto almacena sus propios datos, pero para acceder y operar con ellos, todos comparten los mismos métodos definidos en su clase. Por lo tanto, Para que un método conozca la identidad del objeto en particular para el que ha sido invocado, C++ proporciona un puntero al objeto denominado **this**. De hecho, el compilador reescribe el código de los métodos que hemos definido, anteponiendo ante todos los identificadores de los atributos la expresión (**this->).** Implícitamente, todos los métodos no estáticos de una clase tienen como argumento implícito un puntero al objeto que llama al método, y este es accesible por medio de la palabra **this**.

Por ejemplo, la funcion *estableceValor* podría reescribirse de la forma siguiente:

```
void complex::estableceValor(float re, float im)
{
this->real=re ;
this->imag=im ;
}
```

Al igual que ocurre en C, para acceder a los miembros de una estructura a partir de un puntero a una variable de este tipo, se utiliza el operador flecha.

Instanciación de un objeto de la clase

Una vez definido un tipo de objetos por medio de una clase, la creación y utilización de objetos de esta clase es muy sencilla. Se realiza de forma análoga a como se construye cualquier otra variable de un tipo predefinido. Así, en la función *main* del ejemplo, para crear un objeto de tipo *complex* basta con escribir la línea:

```
complex micomplejo;
```

Y para ejecutar los distintos métodos que se han definido, se utiliza el operador "**.**" que indica que se tratará de un miembro del objeto que lo antecede. A través del operador punto, sólo se podrá acceder a los atributos y métodos públicos, quedando los elementos privados protegidos por el compilador. De esta forma, la siguiente expresión daría un error de compilación:

```
micomplejo.real=2.0;
```

Por lo que para establecer el valor del número complejo sólo se puede hacer uso del método previsto *estableceValor* que obliga a introducir tanto la parte real como la imaginaria.

Todo esto tiene mucho que ver con el concepto de encapsulamiento que se verá a continuación.

3.3. Encapsulamiento

Ya se ha comentado que para una clase, los únicos elementos accesibles desde el exterior son aquellos que han sido declarados como públicos. Por tanto, los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

En la programación orientada a Objetos, los niveles de acceso son el medio que se utiliza para lograr el encapsulamiento que no es más que cada objeto se comporte de forma

autónoma y lo que pase en su interior sea invisible para el resto de objetos. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

El siguiente ejemplo ilustra como este concepto es muy importante de cara a obtener un código reutilizable y seguro que es la principal finalidad de la POO. La siguiente clase lista de números, nos permite almacenar un máximo determinado de números en memoria estática. Dicha clase, evita que el usuario cometa errores de ejecución tales como acceder a un número que no existe o introducir un conjunto de números que supere la capacidad de almacenamiento de esta lista estática.

El precio que hay que pagar por esta protección es el tener que utilizar para todas las operaciones métodos y por tanto funciones, aunque estas operaciones sean muy sencillas. Sin embargo, el resultado es robusto, y fácilmente trasladable a varios programas.

```
class listaNumeros
{
private:
 int num;
 int lista[100];
public:
 int agregarNumero(int val);
 int extraerNumero(int ind);
 int numeroNumeros(void);
 listaNumeros(void);
};
//constructor de la clase… en breve se verá su utilidad
listaNumeros::listaNumeros(void){
  num=0;
  }
int listaNumeros::agregarNumero(int val){
  if(num<100)lista[num++]=val;
  else return -1;
  return num;
  }
int listaNumeros::extraerNumero(int ind){
  if((ind<num)&&(ind>=0))return lista[ind];
  return 0;
  }
int listaNumeros::numeroNumeros(){
  return num;
  }
```

```
#include <iostream.h>
void main()
{
	listaNumeros milista;
	int i,val=1;
	while(val!=0)
		{
		cout<<"introduzca un numero (finalizar con 0):";
		cin>>val;
		if(val!=0)val=milista.agregarNumero(val);
		}
	cout<<"\nLa lista de números es la siguiente:\n";
	for(i=0;i<milista.numeroNumeros();i++)
		cout<<milista.extraerNumero(i)<<" ";
	cout<<"\n*********FIN DEL PROGRAMA**********\n";
}
```

Las funciones diseñadas exclusivamente para servir de interfaz de la parte privada son habitualmente pequeñas, y ocupan muy poquitas líneas. De hecho es muy habitual que existan una serie de métodos cuya única función es devolver el valor de un atributo, y que por tanto solo contienen una sentencia: **return** del atributo.

En este caso es habitual escribir el contenido de estas funciones directamente en la declaración de la clase. Cuando un método se define en la declaración de una clase este método pasa ser de tipo **inline**. Lo que significa que cuando sea llamado por cualquier parte del programa, en vez de realizar el proceso de llamada a un método, el compilador realiza una sustitución de la sentencia de llamada por el contenido interpretado del método. Por tanto, será un proceso rapidísimo en ser ejecutado, a costa de que el código de la función se repetirá tantas veces como llamadas explícitas se escriben en el código.

Por ello, aunque cualquier función puede ser definida como **inline**, sólo se utiliza para aquellas funciones pequeñas en las que el proceso de ejecución de un método es mayor que la ejecución del método en sí. Esto es lo que ocurrirá con todas las funciones de consulta o modificación de atributos no públicos de una clase.

El código anterior podría ser reescrito entonces de la siguiente forma:

```
class listaNumeros
{
private:
 int num;
 int lista[100];
public:
 int agregarNumero(int val);
 int extraerNumero(int ind)
```

```
        {
        if((ind<100)&&(ind>=0))return lista[ind];
        return 0;
        }
 int numeroNumeros(void){return num;};
 listaNumeros(void){num=0;};
};

int listaNumeros::agregarNumero(int val)
{
        if(num<100)lista[num++]=val;
        else return -1;
        return num;
}
```

Aunque el método *agregarNumero* podría perfectamente haberse definido en la declaración de la clase, y por tanto haberse declarado como **inline**, se ha preferido dejarlo de la forma inicialmente descrita para mostrar como en una clase lo normal es mezclar ambos modos de definir los métodos.

Cuando el método es tan breve que puede ser escrito en la misma línea de la propia declaración del método, se opta por la brevedad para evitar declaraciones de clase excesivamente extensas.

La importancia de la encapsulación puede ponerse de manifiesto en esta pequeña y poco útil clase. Tal y como está diseñada, no es posible que exista incoherencia entre el contenido del vector de enteros (*lista*) y el indicador del número de enteros introducido (*num)*, puesto que los valores almacenados en ambos, son modificados por los métodos de la clase, los cuales se aseguran de que los valores introducidos son válidos. Al impedir que el usuario modifique directamente el valor de estos atributos, se evita la aparición de incoherencias y por tanto se elimina una de las mayores fuentes de error.

La clase además gestiona la propia capacidad de almacenamiento, de forma que se asegura de que nunca se introduzcan más de cien números. Aunque el usuario pida introducir doscientos números sólo se almacenaran cien, puesto que sólo se dispone de espacio para cien.

Por todo esto es muy interesante que los atributos que afectan al funcionamiento interno de una clase se declaren protegidos ante posibles modificaciones externas y que siempre se compruebe la validez de los valores solicitados por medio de las distintas funciones de interfaz.

Sin embargo, en ciertas ocasiones, se querrá tener acceso a determinados miembros privados de un objeto de una clase desde otros objetos de clases diferentes. Para esos casos

(que deberían ser excepcionales) C++ proporciona un mecanismo para sortear el sistema de protección. Más adelante se verá la utilidad de esta técnica, ahora se limitará a explicar en que consiste. El mecanismo del que se dispone en C++ es el denominado de amistad (*friend)*.

Clases y métodos friend

La palabra clave **friend** es un modificador que puede aplicarse tanto a clases como a funciones para inhibir el sistema de protección de atributos y funciones de una clase. La idea general es que la parte privada de una clase solo puede ser modificada o consultada por la propia clase o por los *amigos* de la clase (tanto métodos como funciones).

Se puede establecer un cierto paralelismo con el concepto de amistad entre personas para facilitar algunas ideas básicas sobre como funciona este mecanismo en C++. Algunas de estas ideas se pueden resumir en las siguientes afirmaciones:

- **La amistad no puede transferirse**: si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C. (La famosa frase: "los amigos de mis amigos son mis amigos" es falsa en C++ y en la vida real, puesto que es fácil deducir entonces que todos somos amigos de todos y menuda amistad es esa en la que muere tanta gente a manos de otra gente).
- **La amistad no puede heredarse** –ya veremos el concepto de herencia-. Si A es migo de B, y C es una clase hija de B, A no tiene que ser amigo de C. (Los hijos de mis amigos, no tienen porqué ser amigos míos. De nuevo, el símil es casi perfecto… hay cada hijo de mis amigos…).
- **La amistad no es simétrica**. Si A es amigo de B, B no tiene por que ser amigo de A. (En la vida real una situación como esta hará peligrar la amistad de A con B, pero de nuevo y por desgracia se trata de una situación muy frecuente en la que A no sabe lo que B piensa de él… basta con observar lo que algunos son capaces de decir de sus "amig@s" a su espalda).

Funciones amigas de una clase

El siguiente ejemplo ilustra el concepto de amistad, aunque en este programa no es funcionalmente útil. Para que una función sea amiga de una clase, y por tanto tenga acceso a la parte privada de los objetos de dicha clase, sabe ser definida como amiga (anteponiendo la palabra friend al prototipo) en la declaración de la clase. Lógicamente, es la clase la que define quién es amiga suya. Se observa que la función *Ver* no tiene que indicar en ningún momento quien la considera amiga, de tal forma que podríamos tener millares de clases amigas de dicha función sin que por ello esta vea afectada su definición. Cada una de las

clases sin embargo deberá incluir dentro de su declaración la copia del prototipo de la función precedida de la palabra friend.

```
#include <iostream>
class A
 {
 public:
 A(int i) {a=i;};
 void Ver() { cout << a << endl; }
 private:
 int a;
 friend void Visualiza(A); //Visualiza es amiga de la clase A
 };

// La función Visualiza puede acceder a miembros privados
// de la clase A, ya que ha sido declarada "amiga" de A
void Visualiza(A Xa)
 {
 cout << Xa.a << endl;
 }

void main()
 {
 A Na(10);
 Visualiza(Na);  // imprime el valor de Na.a
 Na.Ver(); // Equivalente a la anterior
 }
```

Se observa como la función visualiza es capaz de acceder directamente al miembro privado *a*, del objeto *Na* gracias a su condición de función amiga de la clase. Si en este mismo ejemplo eliminásemos la línea de la declaración de la clase *A* en la que se define Visualiza como función amiga, el compilador daría un error por intentar acceder a un miembro privado del objeto pasado en *Xa*.

Al final de este capítulo se introduce un código extenso que ilustra la utilidad de este mecanismo.

Es importante destacar para finalizar que una función amiga de una clase, no es miembro de esa clase.

Métodos de una clase amigos de otra clase

En este caso se precisa que un método de una clase sea capaz de acceder a los datos y métodos privados de otra clase.

```
#include <iostream>
class A; // Declaración previa (forward)
```

```
class B
{
public:
 B(int i){b=i;};
 void ver() { cout << b << endl; };
 bool esMayor(A Xa); // Compara b con a
private:
 int b;
};
class A
{
public:
 A(int i=0) : a(i) {}
 void ver() { cout << a << endl; }
private:
 // Función amiga tiene acceso
 // a miembros privados de la clase A
 friend bool B::esMayor(A Xa);
 int a;
};
bool B::esMayor(A Xa)
 {
 return (b > Xa.a);
 }
void main(void)
{
A Na(10);
B Nb(12);
Na.ver();
Nb.ver();
if(Nb.esMayor(Na)) cout << "Nb es mayor que Na" << endl;
else cout << "Nb no es mayor que Na" << endl;
}
```

La función EsMayor pertenece a la clase B, pero en la declaración de la clase A se indica que dicho método es capaz de acceder a la parte privada de A. Por ello, es indiferente el lugar dentro de la declaración de A en el que se incluye la declaración de amistad del método de la claseB. En el ejemplo, se ha introducido en la parte privada de la declaración, pero podría haberse situado en cualquier otra parte de la declaración. Puesto que ambas clases deben hacerse referencia entre sí, es necesario realizar una declaración anticipada de la existencia de la clase situada en segundo lugar de forma que el compilador no de un error durante la interpretación. Esta es la finalidad de la segunda línea.

Desde el punto de vista sintáctico se observa que la declaración de amistad de un método y una función son prácticamente equivalentes, puesto que la inclusión de **B::** indica que se trata de una función miembro de B.

Este mecanismo es bastante más utilizado que el anterior debido a que los métodos amigos de una clase son capaces de acceder a la parte privada de su propia clase y de la clase respecto de la cual tienen una relación de amistad.

Clase amiga de otra clase

Cuando lo que se desea es que todos los métodos de una clase tengan la capacidad de acceder a la parte privada de otra, entonces, en vex de ir definiendo la relación de amistad para cada uno de los métodos de la clase, bastará con definir que toda la clase es amiga.

Así, por ejemplo, si se quiere que todos los métodos de la clase C1 sean amigos de la clase C2, y por tanto tengan acceso a la parte privada de los objetos de la clase C2, entonces esquemáticamente esta declaración se realizaría de la forma siguiente:

```
class C1
 {
 …
 };
class C2
 {
 …
 friend class C1;
 …
 };
```

3.4. Constructores y destructores

Existen dos tipos de funciones miembro de una clase especiales cuya función es asegurar que la creación y destrucción de los objetos se realiza de forma correcta. Estas funciones que tienen una sintaxis específica se denominan como constructores –los encargados de la creación del objeto- y el destructor –encargado de su destrucción.

Constructores de una clase

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.

Los constructores tienen el mismo nombre que la clase, no retornan ningún valor y no pueden ser heredados. Además suelen ser públicos, dado que habitualmente se usan desde el exterior de la clase. Algunos patrones más avanzados de lo visto en este curso, como el “singleton” hacen uso a menudo de constructores privados o protegidos, pero en nuestro caso en general los declararemos como públicos.

Los constructores suelen ser públicos.

La siguiente clase es puramente ilustrativa, y sirve para almacenar parejas de números. Para la misma se ha definido un constructor y se ha utilizado en el cuerpo del programa:

```
#include <iostream>
class pareja
{
public:
 // Constructor
 pareja(int a2, int b2);
 // Funciones miembro de la clase "pareja"
 void lee(int &a2, int &b2);
 void guarda(int a2, int b2);
private:
 // Datos miembro de la clase "pareja"
 int a, b;
};
//definición del constructor
pareja::pareja(int a2, int b2)
{
a = a2;
b = b2;
}
void pareja::lee(int &a2, int &b2) {
a2 = a;
b2 = b;
}
void pareja::guarda(int a2, int b2) {
a = a2;
b = b2;
}
void main(void)
{
pareja par1(12, 32);
int x, y;
par1.lee(x, y);
cout << "Valor de par1.a: " << x << endl;
cout << "Valor de par1.b: " << y << endl;
}
```

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase, y si requiere argumentos, es obligatorio suministrarlos. Por ejemplo, las siguientes declaraciones son ilegales:

```
pareja par1;
pareja par1();
```

La primera porque el constructor de "pareja" requiere dos parámetros, y no se suministran. La segunda es ilegal por otro motivo más complejo. Aunque existiese un constructor sin parámetros, no se debe usar esta forma para declarar el objeto, ya que el compilador lo considera como la declaración de un prototipo de una función que devuelve un objeto de tipo "pareja" y no admite parámetros. Cuando se use un constructor sin parámetros para declarar un objeto no se deben escribir los paréntesis.

Las siguientes declaraciones son válidas:

```
pareja par1(12,43);
pareja par2(45,34);
```

Cuando no se especifica un constructor para una clase, el compilador crea uno por defecto sin argumentos al que se denomina como **constructor de oficio**. Por eso los ejemplos anteriores funcionaban correctamente. Cuando se crean objetos locales, los datos miembros no se inicializan si el programador no se preocupa de hacerlo. Contendrían la "basura" que hubiese en la memoria asignada al objeto. Curiosamente, si se trata de objetos globales, los datos miembros se inicializan a cero. Esto se realiza de esta forma porque el proceso de inicialización lleva un tiempo. Los objetos locales son por propia definición temporales y pueden ser creados muchas veces durante la ejecución de un programa, por lo que al final, el tiempo de inicialización puede ser significativo. Por el contrario, los objetos globales son inicializados al comenzar la ejecución del programa, por lo que esto se hace una sóla vez y durante la preparación de la ejecución del código.

Para declarar objetos usando el constructor por defecto o un constructor que se haya declarado sin parámetros no se debe usar el paréntesis:

```
pareja par2();
```

Se trata de un error frecuente cuando se empiezan a usar clases, lo correcto es declarar el objeto sin usar los paréntesis:

```
pareja par2;
```

Inicialización de objetos

Existe un modo simplificado de inicializar los datos miembros de los objetos en los constructores. Se basa en la idea de que en C++ todo son objetos, incluso las variables de tipos básicos como **int**, **char** o **float**.

Según esto, cualquier variable (u objeto) tiene un constructor por defecto, incluso aquellos que son de un tipo básico. Sólo los constructores admiten inicializadores. Cada inicializador consiste en el nombre de la variable miembro a inicializar, seguida de la expresión que se usará para inicializarla entre paréntesis. Los inicializadores se añadirán a continuación del paréntesis cerrado que encierra a los parámetros del constructor, antes del cuerpo del constructor y separado del paréntesis por dos puntos "**:**".

Por ejemplo, en el caso anterior de la clase *pareja*:

```
pareja::pareja(int a2, int b2)
{
a = a2;
b = b2;
}
```

Se puede sustituir el constructor por:

```
pareja::pareja(int a2, int b2) : a(a2), b(b2) {}
```

Por supuesto, también pueden usarse inicializadores en línea, dentro de la declaración de la clase. Ciertos miembros es obligatorio inicializarlos, ya que no pueden ser asignados. Este es el caso de las constantes y las referencias. Es muy recomendable usar la inicialización siempre que sea posible en lugar de asignaciones, ya que se desde el punto de vista de C++ es mucho más seguro.

Al igual que ocurre con las funciones globales, y como ya veremos con los métodos y operadores, es posible sobrecargar el constructor. Recuérdese que la sobrecarga significa que bajo un mismo identificador se ejecutan códigos distintos en función de los argumentos que se pasen para la ejecución. Por eso, la única restricción en la sobrecarga es que no pueden definirse dos constuctores que no se diferencien en el número y tipo de los argumentos que utilizan.

Para la clase palabra, en el siguiente ejemplo, se añaden dos nuevos constructores, el primero sin argumentos que inicializa la pareja a cero, al que se denomina constructor por defecto. El segundo, inicializará la pareja por medio de un número real del cual extrae el primer dígito de la parte decimal y la parte entera.

```
class pareja
```

```
{
public:
// Constructor anterior
pareja(int a2, int b2) : a(a2), b(b2) {}
//contructor por defecto
pareja() : a(0), b(0) {}
//otro constructor
pareja(double num);
// Funciones miembro de la clase "pareja"
void Lee(int &a2, int &b2);
void Guarda(int a2, int b2);
private:
// Datos miembro de la clase "pareja"
int a, b;
};
pareja::pareja(double num)
{
a=(int)num;
b=((int)(num*10))%10;
}
void main()
{
pareja p1,p2(12,3),p3(2.8);
…
}
```

La pareja p1 será entonces inicializada con los valores (a=0,b=0), p2 con (a=12,b=3) y p3 con (a=2, b=8). De este modo se pueden definir muchas maneras de inicializar los objetos de una clase.

Al igual que cualquier función, es posible definir valores por defecto para los parámetros. El inconveniente es que si existen dos funciones que tienen definidos valores por defecto de todos sus argumentos, se producirá una ambigüedad a la hora de definir cual de las dos se tiene que ejecutar. El compilador informará de esta situación, y en caso de que sea necesario utilizar el constructor por defecto en alguna parte del código –es posible que aunque se haya definido no se haga uso de él- entonces generará un error de compilación.

Por ejemplo, el compilador no permitiría la ejecución del siguiente programa:

```
#include <iostream.h>
#include <math.h>
class pareja
{
public:
 // Constructor anterior con valores por defecto
 pareja(int a2=0, int b2=0) : a(a2), b(b2) {}
```

```
  //contructor por defecto
  pareja() : a(0), b(0) {}
  void Lee(int &a2, int &b2);
  void Guarda(int a2, int b2);
private:
 int a, b;
};

void main()
{
pareja p1;
pareja p2(12,3),p3(2.8);
int a,b;
p1.Lee(a,b);
cout<<a<<','<<b;
}
```

e indicaría el siguiente error en la línea de declaración de p1:

'pareja::pareja' : ambiguous call to overloaded function

El constructor de copia

Existe otro tipo de constructor especial que en caso de que l programador no defina el compilador asigna uno de oficio. Este es el llamado constructor de copia, y que como su propio nombre indica sirve para inicializar un objeto como copia de otro.

El constructor de copia de oficio actuará de la misma forma que una igualdad no sobrecargada. Realizará una copia literal de los atributos o datos miembro de un objeto sobre el otro. Habitualmente esto será suficiente, pero hay algunos casos en el que esto no es conveniente. La misma razón y ejemplo que se utilizará para explicar la necesidad de sobrecargar la operación de igualdad servirá para ilustrar la necesidad de definir en algunos casos un constructor de copia no de oficio.

La sintaxis del constructor de copia será habitualmente la siguiente:

```
ident_clase::ident_clase(const tipo_clase &obj);
```

Para el ejemplo pareja se podría escribir lo siguiente:

```
class pareja
```

```
{
…
pareja(const pareja &p):a(p.a),b(p.b){}
…
}
void main(void)
{
pareja p1(12,32);
pareja p2(p1); //Uso del constructor de copia
…
}
```

El destructor

El complemento a los constructores de una clase es el destructor. Así como el constructor se llama al declarar o crear un objeto, el destructor es llamado cuando el objeto va a dejar de existir por haber llegado al final de su vida. En el caso de que un objeto local haya sido definido dentro de un bloque {…}, el destructor es llamado cuando el programa llega al final de ese bloque.

Si el objeto es global o static su duración es la misma que la del programa, y por tanto el destructor es llamado al terminar la ejecución del programa. Los objetos creados con reserva dinámica de memoria (en general, los creados con el operador **new**) no están sometidos a las reglas de duración habituales, y existen hasta que el programa termina o hasta que son explícitamente destruidos con el operador **delete**. En este caso la responsabilidad es del programador, y no del compilador o del sistema operativo. El operador **delete** llama al destructor del objeto eliminado antes de proceder a liberar la memoria ocupada por el mismo.

A diferencia del constructor, el destructor es siempre único (no puede estar sobrecargado) y no tiene argumentos en ningún caso. Tampoco tiene valor de retorno. Su nombre es el mismo que el de la clase precedido por el carácter tilde (~), carácter que se consigue con Alt+126 en el teclado del PC o mediante el uso de los denominados trigrafos (??-). Los trigrafos son conjuntos de caracteres precedidos por ?? que el preprocesador convierte a un carácter a veces no presente en algunos teclados o idiomas. Por tanto, para el ordenador es lo mismo llegar a escribir el carácter ~ que el conjunto de caracteres ??-.

En el caso de que el programador no defina un destructor, el compilador de C++ proporciona un destructor de oficio, que es casi siempre plenamente adecuado (excepto para liberar memoria de vectores y matrices).

En el caso de que la clase *pareja* necesitase un destructor, la declaración sería así:

```
~pareja();
```

El siguiente ejemplo muestra un caso en el que es necesario almacenar una cadena de caracteres de tamaño variable, por lo que al ser eliminado el objeto es necesario liberar la memoria reservada de forma dinámica:

```
class persona
{
public:
 persona(const char *nom, long dni);
 persona():nombre(NULL),DNI(0){};
 ??-persona(); //podría haberse escrito ~persona();
 void mostrar();
private:
 char *nombre;
 long DNI;
};
persona::persona(const char *nom,long dni)
{
nombre=new char[strlen(nom)+1];
strcpy(nombre,nom);
DNI=dni;
cout<<"Construyendo "<<nombre<<endl;
}

persona::~persona() // ??- y ~ son lo mismo
{
cout<<"Destruyendo "<<nombre<<endl;
delete [] nombre;
}
void persona::mostrar()
{
cout<<"almacenado:"<<nombre<<" DNI:"<<DNI<<endl;
}
void main(void)
{
persona p1("dura todo el main",10);
for(int i=0;i<2;i++)
      {
      persona p2("Dura el ciclo del for ",i);
      p2.mostrar();
      }
}
```

El resultado de ejecutar este programa es el siguiente:

```
Construyendo dura todo el main
Construyendo Dura el ciclo del for
almacenado:Dura el ciclo del for  DNI:0
Destruyendo Dura el ciclo del for
Construyendo Dura el ciclo del for
almacenado:Dura el ciclo del for  DNI:1
Destruyendo Dura el ciclo del for
Destruyendo dura todo el main
```

Lo que permite visualizar como han sido llamados los constructores y destructores de los objetos que se han creado.

3.5. Métodos y operadores sobrecargados

Métodos sobrecargados

En el capítulo de modificaciones menores a C, uno se los puntos que se vio era la posibilidad que ofrece C++ para escribir varias funciones que teniendo el mismo identificador eran diferenciadas por el tipo de parámetros utilizado. De esta forma se facilitaba la comprensión del código y la facilidad de programación al evitar el tener que utilizar nombres distintos para tipos de argumentos distintos cuando la funcionalidad es parecida.

Bueno, pues este mismo concepto es aplicable a los métodos de una clase. Baste para ilustrarlo el siguiente ejemplo:

```
#include <iostream>
struct punto3D
 {
 float x, y, z;
 };
class punto
 {
 public:
 void Asignar(float xi, float yi, float zi)
      {
      x = xi;
      y = yi;
      z = zi;
      }
 void Asignar(punto3D p)
      {
      Asignar(p.x, p.y, p.z);
      }
```

```
 void Ver()
      {
      cout << "(" << x << "," << y
      << "," << z << ")" << endl;
      }
private:
 float x, y, z;
};
void main(void)
 {
 punto3D p3d = {32,45,74};
 P.Asignar(p3d);
 P.Ver();
 P.Asignar(12,35,12);
 P.Ver();
 }
```

Operadores sobrecargados

En C++ los operadores pasan a ser unas funciones como otra cualquiera, pero que permiten para su ejecución una notación especial. Todos ellos tienen asignada una función por defecto que es la que hasta ahora habíamos utilizado en C. Por ejemplo, el operador '+' realizará la suma aritmética o la suma de punteros de los operandos que pongamos a ambos lados.

A partir de ahora, podremos hacer que el operador '+' realice acciones distintas en función de la naturaleza de los operandos sobre los que trabaja. De hecho, en realidad, la mayoría de los operandos en C++ están sobrecargados. El ordenador realiza operaciones distintas cuando divide enteros que cuando divide números en coma flotante. Un ejemplo más claro sería el del operador *, que en unos casos trabaja como multiplicador y en otros realiza lo operación de indirección.

En C++ el programador puede sobrecargar casi todos los operadores para adaptarlos a sus propios usos. Para ello disponemos de una sintaxis específica que nos permite definirlos o declararlos al igual que ocurre con las funciones:

```
Prototipo para los operadores:
<tipo> operator <operador> (<argumentos>);

Definición para los operadores:
<tipo> operator <operador> (<argumentos>)
 {
 <sentencias>;
```

```
}
```

Antes de ver lo interesante de su aplicación como operadores miembros de una clase, es necesario mostrar algunas limitaciones en la sobrecarga de operadores, así como su uso como funciones externas a una clase.

Limitaciones:

- Se pueden sobrecargar todos los operadores excepto "**.** " , "**.***" , "**::**" ,"**?:**".
- Los operadores "=","[ ]","->","new" y "delete", sólo pueden ser sobrecargados cuando se definen como miembros de una clase.
- Al menos uno de los argumentos para los operadores externos deben ser tipos enumerados o estructurados: struct, enum, union o class.
- No se pueden cambiar la precedencia, asociatividad o el número de operandos de un operador, pero sí el velor de retorno salvo para los conversores de tipo.
- La sobrecarga de "&&", "||" y "," hacen que pierdan sus propiedades especiales de evaluación ordenada con efecto colateral. Pasarán a comportase como cualquier otro operador o función.
- El operador "->" debe retornar un puntero sobre el que se pueda aplicar el acceso al campo correspondiente.

Para ilustrar cómo se pueden sobrecargar los operadores en C++, se va a definir la operación suma para las estructuras de complejos que se pusieron como ejemplo al comienzo del capítulo:

```
typedef struct _complex{
      float real;
      float imag;
      }complex;

complex operator +(complex x,complex y)
{
complex z;
z.real=x.real+y.real;
z.imag=x.imag+y.imag;
return z;
}
#include <iostream.h>

void main()
```

```
{
complex a={5.0F,3.0F},b={3.0F,1.2F},c;
c=a+b;
cout<<c.real<<"+"<<c.imag<<"i"<<endl;
}
```

Puesto que lo que se ha sobrecargado es el operador suma con dos operandos, es necesario que aparezcan tanto en el prototipo como en la definición estos dos operandos con su tipo. Es fácil deducir que con esta sintaxis se podrán generar operadores suma que sean capaces de sumar vectores y complejos y que su resultado sea una matriz. Como siempre, C++ nos da una herramienta que clarifica el código, pero sigue dependiendo del programador que su uso sea lógico.

Sin embargo, donde más aplicación y uso tiene la sobrecarga de operadores es cuando el operador es miembro de una clase. Se va a distinguir a continuación los operadores binarios (trabajan con dos operandos) de los unarios (trabajan con uno solo).

Sobrecarga de operadores binarios

Como su propio nombre indica, los operadores binarios son aquellos que requieren de dos operandos para calcular el valor de la operación. Esto aparece muy claro en los operadores definidos en el exterior de una clase.

Sin embargo, cuando un operador binario es a su vez miembro de una clase, se asume que el primer operando de la operación es el propio objeto de la clase donde se define el operador. Por ello, en la sobrecarga de operadores binarios sólo será necesario especificar un operando, puesto que el otro es el propio objeto.

Por tanto, en la declaración de la clase se incluirá una línea con la siguiente sintaxis:

```
<tipo> operator<operador binario>(<tipo> <identificador>);
```

Habitualmente <tipo> será para un objeto de la misma clase, pero no es necesario. Así el caso más habitual es que si estamos sumando complejos, el primer y segundo operador sean de tipo complejo y el resultado un complejo.

Sin embargo no tiene por que ser de esta manera. Por ejemplo en el caso de operar con matrices, es perfectamente válido definir una operación producto entre una matriz y un vector, objetos de dos clases distintas, y cuyo resultado es un vector. Si la operación producto

está definida en el interior de la clase matriz, el argumento será de tipo vector y el valor de retorno también.

Se mostrará un ejemplo sencillo, que permite operar con valores de tiempo de forma sencilla. En esta clase se definirá el operador suma de tal forma que se puedan sumar periodos de duración medidos en horas y minutos.

```
#include <iostream.h>
class Tiempo
{
public:
 Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}
 void Mostrar(){cout << hora << ":" << minuto << endl;};
 Tiempo operator+(Tiempo h);
private:
 int hora;
 int minuto;
};

Tiempo Tiempo::operator+(Tiempo h)
{
Tiempo temp;
temp.minuto = minuto + h.minuto;
temp.hora = hora + h.hora;
if(temp.minuto >= 60){
  temp.minuto -= 60;
  temp.hora++;
  }
return temp;
}
void  main(void)
 {
 Tiempo Ahora(12,24), T1(4,45);
 T1 = Ahora + T1;
 T1.Mostrar();
 (Ahora + Tiempo(4,45)).Mostrar(); //(1)
 }
```

En este ejemplo se ha introducido una de las posibilidades que ofrece C++ hasta ahora no comentada ni utilizada. En el punto del código marcado con (1), se utiliza una instancia de la clase sin haberle dado un nombre. Es decir, Tiempo(4,5) crea un objeto que no podremos referenciar posteriormente una vez ejecutada la sentencia puesto que no le hemos asignado ningún identificador. Sin embargo el objeto existirá y podrá ser utilizado, de forma que en (1) se está diciendo que se sume al tiempo guardado en Ahora un total de 4 horas y 45 minutos.

Igual es mucho de golpe, pero es momento de comenzar a ir viendo código más real.

A continuación se realizará una sobrecarga del operador suma de tal forma que a la hora le vamos a sumar sólo minutos. En ese caso, el operador suma recibirá como segundo operando un valor entero, y dará como resultado un objeto Tiempo.

```
#include <iostream.h>
class Tiempo
{
public:
 Tiempo(int h=0, int m=0) : hora(h), minuto(m) {};
 void Mostrar(){cout << hora << ":" << minuto << endl;};
 Tiempo operator+(Tiempo h);
 Tiempo operator+(int mins);
private:
 int hora;
 int minuto;
};

Tiempo Tiempo::operator+(Tiempo h)
{
 Tiempo temp;
 temp.minuto = minuto + h.minuto;
 temp.hora = hora + h.hora;
 if(temp.minuto >= 60)
    {
    temp.minuto -= 60;
    temp.hora++;
    }
 return temp;
}
Tiempo Tiempo::operator +(int mins)
{
 Tiempo temp;
 temp.minuto=minuto+mins;
 temp.hora=hora+temp.minuto/60;
 temp.minuto=temp.minuto%60;
 return temp;
}

void  main(void)
 {
 Tiempo Ahora(12,24), T1(4,45);
 T1 = Ahora + T1;
 T1.Mostrar();
 (Ahora+45).Mostrar();
 }
```

De nuevo se ha utilizado un objeto temporal que es el retornado por el operador +. El compilador diferencia a cual de las dos operaciones suma hacemos referencia gracias al distinto tipo de operandos utilizado en cada caso. Así, en la primera suma se ejecuta el operador para sumar dos tiempo, mientras que en la segunda se ejecutará el operador que permite la suma de un tiempo y unos minutos.

Obsérvese –aunque esto es tema de C- cómo se aprovecha el modo en que C opera distinto con los números en coma flotante y los enteros para poder transformar los minutos en las horas y minutos correspondientes.

Sobrecarga del operador igual.

Un operador que a menudo es necesario sobrecargar es el operador igual. Si no se sobrecarga este operador, el compilador asignará uno por defecto (como en el caso del ejemplo) que realizará una copia literal de lo que hay en la memoria de un objeto sobre el otro. Este operador por defecto es válido siempre que no se esté utilizando memoria dinámica, pero puede crear auténticos quebraderos de cabeza en caso contrario.

Un ejemplo típico de esta situación es el de una clase con capacidad de almacenar frases tan largas como se quiera: la clase cadena. El siguiente código no funcionará como deseamos, y posiblemente nos de muchos problemas:

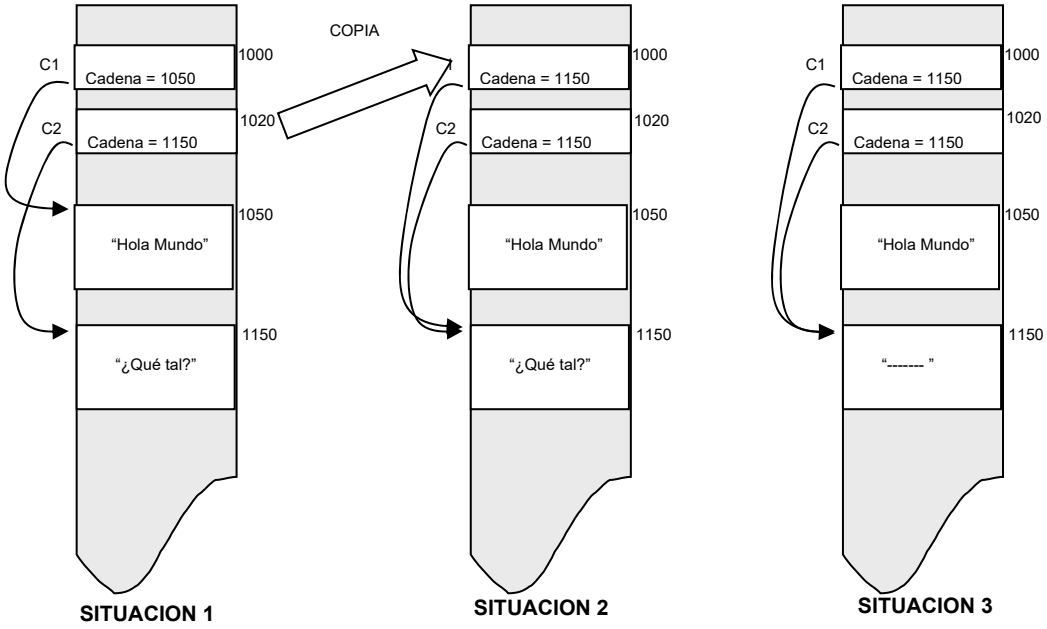
```
class Cadena
{
public:
 Cadena(char *cad);
 Cadena() { cadena=NULL;} ;
 ~Cadena() { delete[] cadena; };
 void Mostrar(){cout << cadena << endl;};
 void RellenarDeGuiones();
private:
 char *cadena;
};
Cadena::Cadena(char *cad)
{
cadena = new char[strlen(cad)+1];
strcpy(cadena, cad);
}
void Cadena::RellenarDeGuiones()
{
for(int i=0;i<strlen(cadena);i++)
      cadena[i]='-';
}
```

Si ejecutamos el siguiente programa, se observa que el resultado dista mucho de ser el esperado además de generar un error al finalizarse la ejecución:

```
void main(void)
{
Cadena C1("Hola Mundo"), C2("¿Qué tal?"); //(1)
C1.Mostrar();
C2.Mostrar();
C1=C2; //(2)
C2.RellenarDeGuines(); //(3)
C1.Mostrar();
C2.Mostrar();
}
```

En contra de lo que pueda parecer, en la impresión segunda impresión de C1 y C2, en pantalla aparecen solo guiones para ambas cadenas… ¿Por qué, si en el código solo se ha indicado el relleno de guiones para C2?¿Por qué es modificado C1?.

Las tres columnas siguientes muestran el estado de C1 y C2 en (1), (2), y (3) en la memoria, en donde como es habitual el contenido del puntero es representado por una flecha.



Al hacer la asignación no se ha copiado el contenido de la cadena, sino que se ha copiado el valor almacenado en el puntero. Ese puntero era el que señalaba la zona de la memoria en donde por medio del operador **new** el sistema había reservado el espacio para almacenar la frase. Tras la asignación, tanto C1 como C2 tienen un puntero que apunta al mismo sitio (el lugar que originalmente se había reservado para C2), mientras que hay una zona de memoria que se había reservado y que no queda apuntada por ninguno de los dos.

Por eso al modificar tanto C1 como C2, el resultado se observa en ambos, puesto que ambos están apuntando el mismo sitio. Además se produce un error en la ejecución del programa, puesto que al eliminarse C1 se eliminará la memoria apuntada por su puntero cadena, y el sistema, al intentar eliminar C2, se encontrará con que el sitio apuntado ya no pertenece al programa pues ha sido liberado. Recuérdese que no se puede llamar a **delete** sobre una dirección que no nos pertenece.

Para evitar este problema se puede sobrecargar el operador de asignación, de forma que al hacer el igual lo que se realiza en la memoria es una duplicación del espacio y una copia carácter a carácter de una cadena sobre la otra.

El código para realizar este operador sería el siguiente:

```
Cadena &Cadena::operator=(const Cadena &c)
 {
 if(this != &c)
              {
       delete[] cadena;
              if(c.cadena)
                     {
                     cadena = new char[strlen(c.cadena)+1];
                     strcpy(cadena, c.cadena);
                     }
              else cadena = NULL;
              }
 return *this;
 }
```

Al manejar punteros hay que tener una serie de precauciones que aparecen de manifiesto en este ejemplo. Por ejemplo, inicialmente se comprueba que no se está realizando una asignación de una Cadena sobre sí misma. Se comprueba antes si existe una cadena en el objeto cadena pasado, y no hubiera estado de más comprobar que existe un espacio reservado en la misma cadena antes de proceder a eliminarla. Gracias al puntero **this** estas operaciones se han podido realizar sin problema.

Es muy habitual que el operador '=' retorne una referencia al propio objeto para permitir realizar asignaciones concatenadas sin necesidad de crear un objeto temporal. Gracias a esta referencia será posible escribir sentencias del estilo: C1=C2=C3… tal y como sucede con las asignaciones en los tipos básicos.

Además del operador **+** pueden sobrecargarse prácticamente todos los operadores:

+, -, *, /, %, ^, &, |, (,), <, >, <=, >=, <<, >>, ==, !=, &&, ||, =, +=. -=, *=, /=, %=, ^=, &=, |=, <<=,>>=, [ ], (),->, new y delete.

Los operadores **=, [], ()** y **->** sólo pueden sobrecargarse en el interior de clases.

Por ejemplo, el operador **>** para la clase Tiempo podría declararse y definirse así:

```
class Tiempo
 {
 ...
 bool operator>(Tiempo h);
 ...
 };
bool Tiempo::operator>(Tiempo h)
 {
 return (hora > h.hora ||
       (hora == h.hora && minuto > h.minuto));
 }
void main(void)
 {
 ...
 if(Tiempo(1,32) > Tiempo(1,12))
       cout << "1:32 es mayor que 1:12" << endl;
 else cout << "1:32 es menor o igual que 1:12" << endl;
 ...
 }
```

Lógicamente, los operadores de comparación suelen retornar un valor **bool**, ya que estamos comprobando si algo es cierto o falso.

Otro ejemplo para la clase Tiempo sería el del operador +=, que al igual que con los tipos básicos realizará la suma sobre el primer operando del segundo.

```
class Tiempo
 {
 ...
 void operator+=(Tiempo h);
 ...
```

```
};
void Tiempo::operator+=(Tiempo h)
 {
 minuto += h.minuto;
 hora += h.hora;
 while(minuto >= 60)
       {
       minuto -= 60;
       hora++;
       }
 }
void main(void)
 {
 ...
 Ahora += Tiempo(1,32);
 Ahora.Mostrar();
 ...
 }
```

No es imprescindible mantener el significado de los operadores. Por ejemplo, para la clase Tiempo no tiene sentido sobrecargar el operadores >>, <<, * ó /, pero se puede de todos modos, y olvidar el significado que tengan habitualmente.

De igual modo se podría haber sobrecargado el operador + y hacer que no sumara los tiempos sino que, por ejemplo, los restara. En última instancia, es el programador el que decide el significado de los operadores.

Sin embargo, como norma general, es conveniente de cara a obtener un código más legible que la funcionalidad del operador sea análoga a su significado.

Sobrecarga del operador >> y <<.

Por su aplicación más habitual se considerará brevemente el caso de los operadores binarios de desplazamiento o de inserción y extracción en un flujo de datos. En primer lugar hay que aclarar que no se trata de un operador nuevo, sino que es el que se ha escogido para insertar información en los iostream. Por tanto se puede utilizar para cualquier otra operación que consideremos oportuna. Sin embargo lo más habitual es respetar este uso realizado por la librería estándar, y además en muchos casos adherirnos a este modo de transmitir o recibir la información de un objeto.

Cuando insertamos un objeto en un flujo de datos normalmente lo hacemos de la forma siguiente:

```
cout<<miObjeto;
```

Lo cual implica que a la izquierda del operador esta cout y a las derecha nuestro objeto. Por tanto si sobrecargamos el operador dentro de una clase, obligatoriamente debería ser dentro de la clase del tipo de cout. Lógicamente esta clase no la podemos modificar

puesto que pertenece a la STL. Por ello, la sobrecarga del operador de inserción se realiza como una función externa, y además, como habitualmente será necesario acceder a la parte provada del objeto para poder imprimirlo, se declara como función amiga de la clase.

El aspecto que tiene la sobrecarga tanto del operador >> como <<, siguiendo la forma más común de realizarse es el siguiente:

```
ostream & operator<< (ostream &os, const MI_CLASE &obj)
istream & operator>> (istream &is, MI_CLASE &obj)
```

Los flujos de salida, gracias a la herencia, podremos agruparlos de forma genérica dentro del concepto *ostream.* Por tanto cout, es un objeto de tipo *ostream*. Los flujos de entrada serán del tipo *istream*. Para poder concatenar inserciones y extracciones, el resultado de la operación es el mismo flujo sobre el que se ha extraido o insertado. Pongamos como ejemplo estos operadores para la clase Tiempo:

```
class Tiempo
 {
 ...
 friend ostream & operator<< (ostream &os, const Tiempo &t)
 friend istream & operator>> (istream &is, Tirmpo &t)
 ...
 };
ostream & operator<< (ostream &os, const Tiempo &t)
 {
  os<<t.hora<<':'<<t.minuto;
  return os;
 }
istream & operator>> (istream &is, Tiempo &t)
 {
  string a;
  is>>a;
  int index=a.find_first_of(":");
  if(index!=string::nppos){
    t.hora=stoi(a.substring(0,index);
    t.minuto=stoi(a.substring(index+1));
  }
  return is;
 }


void main()
{
 Tiempo t1;
 cin>>t1;
 cout<<"Tiempo leído = "<<t1<<endl;
}
```

Sobrecarga de operadores unarios

Como su propio nombre indica, los operadores unarios son aquellos que requieren de un solo operando para calcular su valor. Tal es el caso del operador incremento o el operador negación.

Cuando se sobrecargan operadores unitarios en una clase el operando es el propio objeto de la clase donde se define el operador. Por lo tanto los operadores unitarios dentro de las clases en un principio no requieren de operandos. De ahí que la sintaxis se simplifique respecto de los operadores binarios y quede como sigue:

```
<tipo> operator<operador unitario>();
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador, pero al igual que ocurría con los operadores binarios, esto es algo que queda a la libre elección del programador.

Como ejemplo se realizará el operador de incremento para la clase Tiempo que se ha venido definiendo hasta ahora.

```
class Tiempo
 {
 ...
 Tiempo operator++();
 ...
 };
Tiempo Tiempo::operator++(){
 minuto++;
 while(minuto >= 60)
      {
      minuto -= 60;
      hora++;
      }
 return *this;
 }
void main(void){
 ...
 T1.Mostrar();
 ++T1;
 T1.Mostrar();
 ...
 }
```

Sin embargo, existen dos versiones para el operador incremento, el pre incremento y el post incremento. El ejemplo mostrado es la sobrecarga del operador pre incremento para la clase Tiempo, ¿cómo se sobrecarga el operador de post incremento?

En realidad no hay forma de decirle al compilador cuál de las dos modalidades del operador se está sobrecargando, así que los compiladores usan una regla: si se declara un parámetro para un operador ++ ó -- se sobrecargará la forma sufija del operador.

El parámetro se ignorará, así que bastará con indicar el tipo.

También tenemos que tener en cuenta el peculiar funcionamiento de los operadores sufijos, cuando se sobrecarguen, al menos si se quiere mantener el comportamiento que tienen normalmente.

Cuando se usa un operador en la forma sufijo dentro de una expresión, primero se usa el valor actual del objeto, y una vez evaluada la expresión, se aplica el operador. Si se quiere que el operador actúe igual se debe usar un objeto temporal, y asignarle el valor actual del objeto. Seguidamente se aplica el operador al objeto actual y finalmente se retorna el objeto temporal.

```
class Tiempo
 {
...
Tiempo operator++(); // Forma prefija
Tiempo operator++(int); // Forma sufija
...
};
Tiempo Tiempo::operator++()
 {
 minuto++;
 while(minuto >= 60)
    {
    minuto -= 60;
    hora++;
    }
 return *this;
 }
Tiempo Tiempo::operator++(int)
 {
 Tiempo temp(*this); // Constructor copia
 minuto++;
 while(minuto >= 60)
    {
    minuto -= 60;
```

```
      hora++;
      }
   return temp;
   }
  void main(void)
   {
   …
   T1.Mostrar();
   (T1++).Mostrar();
   T1.Mostrar();
   (++T1).Mostrar();
   T1.Mostrar();
   …
   }
```

El resultado de ejecutar esta parte del programa será

`17:9` (Valor inicial)

`17:9` (Operador sufijo, el valor no cambia hasta después de mostrar el valor)

`17:10` (Resultado de aplicar el operador)

`17:11` (Operador prefijo, el valor cambia antes de mostrar el valor)

`17:11` (Resultado de aplicar el operador)

Además del operador ++ y -- pueden sobrecargarse prácticamente todos los operadores unitarios:

+, -, ++, --, *, & y **!**.

Un caso particular de los operadores unarios es el de las conversiones de tipo. Para ilustrar su utilidad y cuando aparecen se continuará con el ejemplo de la clase Tiempo. Supongase que se quiere realizar una operación como la siguiente:

```
…
Tiempo T1(12,23);
unsigned int minutos = 432;
T1 += minutos;
…
```

El valor deseado, que sería sumar los minutos indicados al tiempo almacenado en T1, no se calculará correctamente. Lo que ocurre cuando se ejecuta la útima sentencia es lo siguiente. En C++ se realizan conversiones implícitas entre los tipos básicos antes de operar con ellos. Por ejemplo, para sumar un **int** y un **float** se promociona (convierte) el entero a

float y después se realiza la operación entre dos números del mismo tipo, siendo entonces el resultado de la misma un **float**.

Esto es lo que ocurrirá con la expresión del ejemplo. El valor de minutos se intentará convertir a un objeto Tiempo, puesto que el operador += definido espera un objeto de este tipo como operando. Para ello se utilizará el constructor diseñado. Como sólo hay un parámetro, y ambos tienen definidos valores por defecto, entonces el parámetro *h* toma el valor de *minutos*, y el valor de *m* toma el de por defecto (o sea 0). El tipo de *h* es de tipo entero, por lo que se convierte el valor **unsigned int** a **int** y se ejecuta el constructor.

El resultado es que se suman 432 horas, y cuando lo que se deseaba era sumar 432 minutos. Esto se soluciona creando un nuevo constructor que tome como parámetro un **unsigned int,** de forma que el compilador diferencia la operación que tiene que realizar gracias al tipo del argumento:

```
Tiempo(unsigned int m) : hora(0), minuto(m)
{
while(minuto >= 60)
      {
      minuto -= 60;
      hora++;
      }
}
```

En general se podrán realizar conversiones de tipo desde cualquier objeto a un objeto de la clase que se está definiendo mediante la sobrecarga del constructor.

Dado que esto puede provocar muchas ambigüedades en C++ se incluye el modificador de constructores **explicit**. De tal forma que un constructor declarado como explicit nunca se ejecutará de forma implícita, en estas o en otro tipo de conversiones decididas por el compilador. Será necesario especificar que queremos realizar esta conversión. En C++11 este modificador se permitirá usar también como modificador de las conversiones de tipo que veremos a continuación.

La palabra clave explicit sirve para evitar que el compilador realice conversiones implícitas mediante un constructor específico.

Sin embargo, es interesante ver como se trabaja en el caso inverso. Es decir, ahora lo que se desea es dotar a C++ de mecanismos para convertir una instancia de la clase a otro tipo. Por ejemplo, se desea ahora asignar a un entero el valor contenido en Tiempo. La idea es que se desea transformar las horas y minutos en el total de minutos que representan:

```
…
Tiempo T1(12,23);
int minutos;
minutos = T1;
…
```

En este caso se obtendría un error de compilación, ya que el compilador no sabe convertir un objeto Tiempo a entero. Para poder realizar esta operación se deberá diseñar un operador de conversión de tipo que ilustre al compilador de que operaciones deber realizar para obtener el entero equivalente a un objeto Tiempo.

Los operadores de conversión de tipo tienen el siguiente formato:

```
operador <tipo>();
```

No necesitan que se especifique el tipo del valor de retorno, ya que este es precisamente <tipo>. Además, al ser operadores unitarios, tampoco requieren argumentos, puesto que se aplican al propio objeto.

```
class Tiempo
{
...
operator int();
...
}
Tiempo::operator int()
{
return hora*60+minuto;
}
```

Por supuesto, el tipo no tiene por qué ser un tipo básico, puede tratarse de una estructura o una clase.

Los operadores [ ] y ( )

El operador de indexación

El operador [ ] se usa para acceder a valores de objetos de una determinada clase como si se tratasen de arrays. Los índices no tienen por qué ser de un tipo entero o enumerado, ahora no existe esa limitación.

Donde más útil resulta este operador es cuando se usa con estructuras dinámicas de datos: listas, árboles, vectores dinámicos, etc…. Pero también puede servir para crear arrays asociativos, donde los índices sean por ejemplo, palabras.

De nuevo se explicará el uso de este operador usando un ejemplo. La siguiente clase permite obtener el valor resultante de interpolar entre los valores de un vector de *n* componentes, en función de un indice que en vez de ser entero es real:

```
#include <iostream.h>
#include <math.h>
class vector
{
public:
      vector(int n=0,double *v=NULL);
      double operator[](double ind);
      ~vector(){delete [] valores;}
private:
int num;
double *valores;
};
vector::vector(int n,double *v)
{
      valores = new double[n];
      num=n;
      for(int i=0;i<n;i++)valores[i]=v[i];
}
double vector::operator [](double ind)
{
double temp;
int sup,inf;
inf=(int)floor(ind); //entero redondeado abajo
sup=(int)ceil(ind);      //entero redondeado arriba
if(inf>num-1)inf=num-1;
if(sup>num-1)sup=num-1;
if(inf<0)inf=0;
if(sup<0)sup=0;
if(inf<sup)
      temp=(sup-ind)*valores[inf]+(ind-inf)*valores[sup];
else  temp=valores[inf];
return temp;
}

void main(void)
{
double val[4]={1.0,4.0,-1.0,2.0};
```

```
vector mivector(4,val);
for(int i=0;i<31;i++)
      cout<<mivector[i*0.1]<<endl;

}
```

El programa mostrará los valores intermedios de 0.1 en 0.1. Se ha utilizado un valor double como índice, pero de igual forma se podría haber utilizado una cadena de caracteres o un objeto.

Cuando se combina el operador de indexación con estructuras dinámicas de datos como las listas, se puede trabajar con ellas como si se tratada de arrays de objetos, esto dará una gran potencia y claridad al código de los programas.

El operador llamada

El operador llamada ( ) funciona exactamente igual que el operador [], aunque admite más parámetros. Este operador permite usar un objeto de la clase para el que está definido como si fuera una función.

Por tanto el número de parámetros y el valor de retorno dependerán totalmente de lo que decida el programador. Para el ejemplo anterior se va a ampliar el operador indexación definido, de forma que si se utiliza el operador llamada se pude escoger si se desea realizar la interpolación o no:

```
class vector
{
…
double operator()(double ind,bool interpolar=true);
…
}
double vector::operator ()(double ind,bool interpolar)
{
if(interpolar)return (*this)[ind];
 else return valores[(int)ind];
}
void main(void)
{
double val[4]={1.0,4.0,-1.0,2.0};
vector mivector(4,val);
for(int i=0;i<31;i++)
      {
      cout<<mivector[i*0.1]<<" ";
```

```
        cout<<mivector(i*0.1,false)<<endl;
        }
}
```

3.6. Miembros static

Atributos static

En ocasiones se hace necesario de disponer de una variable común a todos los objetos de una clase. Por ejemplo, si se quisiera llevar cuenta de cuantos objetos se han creado de una clase determinada, se debería tener un contador que fuera común a todos los objetos. Esta funcionalidad es proporcionada en C++ por medio de los miembros estáticos de una clase.

Para ilustrar esto se va a definir una clase libro sencilla, que permitirá asignar un identificador único a cada libro creado.

```
#include <iostream.h>
#include <string.h>

class libro
{
public:
 libro (const char *tit, const char *aut);
 ~libro();
 void mostrar();
 static int contador;     //(1)
private:
 char *titulo;
 char *autor;
 int ID;

};

int libro::contador=0;    //(2)

libro::libro(const char *tit,const char *aut)
{
 titulo=new char[strlen(tit)+1];
 autor=new char[strlen(aut)+1];
 strcpy(titulo,tit);
 strcpy(autor,aut);
 ID=++contador;           //(3)
```

```
}

libro::~libro()
{
 delete [] titulo;
 delete [] autor;
}

void libro::mostrar()
{
 cout<<"Libro "<<ID<<": \t"<<titulo<<endl;
 cout<<"\t\t"<<autor<<endl<<endl;
}

void main(void)
{
 libro l1("Cucho","Olaizola");
 libro l2("Tuareg","Vazquez Figueroa");
 libro l3("El Quijote","Cervantes");
 l1.mostrar();
 l2.mostrar();
 l3.mostrar();
}
```

El resultado de ejecutar este código es el siguiente:

Libro 1: Cucho
Olaizola
Libro 2: Tuareg
Vazquez Figueroa
Libro 3: El Quijote
Cervantes

Un atributo **static** no es un atributo específico de un objeto (como es el caso de los atributos autor, titulo o ID) sino que más bien es un atributo de la clase; esto es, un atributo del que sólo hay una copia que comparten todos los objetos de la clase. Por este motivo, un atributo **static** existe y puede ser utilizado aunque no exista ningun objeto de la clase.

Se observa como para declarar un atributo static basta con anteponer **static** a la declaración dentro de la clase. De hecho el nivel de acceso (privado, público o protegido) se mantiene válido para los atributo y métodos de tipo estático. En el ejemplo, la declaración puede verse en (1).

Sin embargo es necesario inicializar (y por tanto crear) la variable estática en algún momento y además sólo una vez. Es decir, tiene que ser inicializado a nivel global y hay que asegurarse de que esta inicialización no se realiza más veces. Es por ello que es necesario escribir la línea (2), en la que se dice que el atributo de la clase libro llamado contador de tipo entero vale cuando es creado 0.

La inicialización de un atributo estático se coloca generalmente en el fichero fuente *.cpp* que contiene las definiciones de los métodos de la clase, puesto que el compilador lee este fichero una sola vez, mientras que el fichero de cabecera (.h) de definición de la clase puede ser incluido multitud de veces, lo que provocaría multitud de inicializaciones con el consiguiente error de inicialización.

Por lo demás, un atributo estático no se diferencia del resto de atributos y es posible acceder al mismo de igual forma que ocurre con los atributos normales (3).

Sin embargo, al ser un atributo que pertenece a la clase y no a un objeto en particular, es una variable que existe desde el principio del programa… ¿No sería posible acceder a su valor sin necesidad de crear un objeto?. La respuesta es que sí.

Por ejemplo, el siguiente *main* sería perfectamente válido:

```
void imprimirLibros()
{
libro l1("Cucho","Olaizola");
libro l2("Tuareg","Vazquez Figueroa");
libro l3("El Quijote","Cervantes");
l1.mostrar();
l2.mostrar();
l3.mostrar();
}

void main(void)
{
cout<<"Numero inicial de libros:";
cout<<libro::contador<<endl;
imprimirLibros();
cout<<"Libros creados:";
cout<<libro::contador<<endl;
}
```

Obsérvese que una traducción literal de lo escrito es totalmente coherente con lo explicado. La expresión *libro::contador* significa el atributo contador perteneciente a la clase libro. Nótese que en el cuerpo del *main* no existe ninguna variable de tipo libro con la que

acceder al atributo, pero como esta es global y existe para todos los objetos, es posible obtener y darle un valor desde cualquier sitio.

Evidentemente, ha sido posible acceder al valor del atributo contador por ser este un atributo público. Sin embargo parece lógico que dicho atributo fuera un atributo privado. Si reescribiéramos la clase libro con el atributo contador como miembro privado estático, entonces el compilador no permitiría la consulta del mismo fuera de un objeto de la clase.

Siempre nos queda la opción de crear un método de interfaz que a través de un objeto de la clase nos remitiera el valor del contador. Sin embargo C++ nos da la posibilidad de consultar este tipo de atributos y de modificar su valor aun siendo privados por medio de los métodos estáticos como se verá a continuación.

Metodos static

Un método declarado como **static** carece del puntero **this** por lo que no puede ser invocado para un objeto de su clase, sino que se invoca en general allí donde se necesite utilizar para la operación para la que ha sido escrito. Desde este punto de vista es imposible que un método static pueda acceder a un miembro no static de su clase; por la misma razon, si puede acceder a un miembro **static.** Luego si un objeto llama a un método static de su clase, hay que tener en cuenta que no podrá acceder a ninguno de sus atributos particulares, a pesar de que su modo de uso no difiere de cualquier otro método de la clase.

Si reescribimos la clase libro con las modificaciones indicadas al final del anterior epígrafe, sería necesario un método static para poder consultar el valor del atributo contador que ahora es privado. Al igual que ocurría con los atributos, un método se declara estático en la declaración de la clase, pero no es necesario (de hecho no hay que hacerlo) en el momento en el que se define el método.

El siguiente ejemplo ilustra estos aspectos así como el modo de llamar a un método static sin necesidad de crear un objeto de la clase. Antes de transcribir el mismo hay que destacar que la utilidad de los miembros static de una clase, no es fácil descubrirla en la primera impresión, pero a medida que se va aprendiendo a programar en C++ se comprueba la gran utilidad y la facilidad con que se resuelven muchos problemas gracias a esta herramienta.

```
#include <iostream.h>
#include <string.h>

class libro
{
public:
      libro (const char *tit, const char *aut);
      ~libro();
```

```
        void mostrar();
        static int getContador(){return contador;};
private:
        char *titulo;
        char *autor;
        int ID;
        static int contador;
};

int libro::contador=0;



libro::libro(const char *tit,const char *aut)
{
titulo=new char[strlen(tit)+1];
autor=new char[strlen(aut)+1];
strcpy(titulo,tit);
strcpy(autor,aut);
ID=++contador;
}

libro::~libro()
{
delete [] titulo;
delete [] autor;
}

void libro::mostrar()
{

        cout<<"Libro "<<ID<<": \t"<<titulo<<endl;
        cout<<"\t\t"<<autor<<endl<<endl;
}
void imprimirLibros()
{
libro l1("Cucho","Olaizola");
libro l2("Tuareg","Vazquez Figueroa");
libro l3("El Quijote","Cervantes");
l1.mostrar();
l2.mostrar();
l3.mostrar();
}
void main(void)
{
cout<<"Numero inicial de libros:";
cout<<libro::getContador()<<endl;
```

```
   imprimirLibros();
   cout<<"Libros creados:";
   cout<<libro::getContador()<<endl;
   }
```

3.7. Ejercicios

El siguiente código extraído de la red realiza una clase para operar cómodamente con números complejos. Algunas de las soluciones adoptadas tienen finalidad puramente docente para mostrar casi todos los aspectos de este capítulo:

```
/******************** fichero Complejo.h*************************/

 // fichero Complejo.h
 // declaración de la clase Complejo
 #ifndef __COMPLEJO_H__
 #define __COMPLEJO_H__
 #include <iostream.h>
 class Complejo
 {
 private:
    double real;
    double imag;
 public:
    // Constructores
    Complejo(void);
    explicit Complejo(double, double im=0.0);
    Complejo(const Complejo&);
    // Set Cosas
    void SetData(void);
    void SetReal(double);
    void SetImag(double);
    // Get Cosas
    double GetReal(void){return real;}
    double GetImag(void){return imag;}
    // Sobrecarga de operadores aritméticos
    Complejo operator+ (const Complejo&);
    Complejo operator- (const Complejo&);
    Complejo operator* (const Complejo&);
    Complejo operator/ (const Complejo&);
    // Sobrecarga del operador de asignación
    Complejo& operator= (const Complejo&);
    // Sobrecarga de operadores de comparación
    friend int operator== (const Complejo&, const Complejo&);
    friend int operator!= (const Complejo&, const Complejo&);
    // Sobrecarga del operador de inserción en el flujo de salida
    friend ostream& operator<< (ostream&, const Complejo&);
```

```
    };
    #endif


   /******************* fichero
Complejo.cpp*************************/

    #include "Complejo.h"
    // constructor por defecto
    Complejo::Complejo(void)
    {
    real = 0.0;
    imag = 0.0;
    }
    // constructor general
    Complejo::Complejo(double re, double im)
    {
    real = re;
    imag = im;
    }
    // constructor de copia
    Complejo::Complejo(const Complejo& c)
    {
    real = c.real;
    imag = c.imag;
    }
    // función miembro SetData()
    void Complejo::SetData(void)
    {
    cout << "Introduzca el valor real del Complejo: ";
    cin >> real;
    cout << "Introduzca el valor imaginario del Complejo: ";
    cin >> imag;
    }
    void Complejo::SetReal(double re)
    {
    real = re;
    }
    void Complejo::SetImag(double im)
    {
    imag = im;
    }
    // operador miembro + sobrecargado
    Complejo Complejo::operator+ (const Complejo &a)
    {
    Complejo suma;
    suma.real = real + a.real;
```

```
suma.imag = imag + a.imag;
return suma;
}
// operador miembro - sobrecargado
Complejo Complejo::operator- (const Complejo &a)
{
Complejo resta;
resta.real = real - a.real;
resta.imag = imag - a.imag;
return resta;
}

// operador miembro * sobrecargado
Complejo Complejo::operator* (const Complejo &a)
{
Complejo producto;
producto.real = real*a.real - imag*a.imag;
producto.imag = real*a.imag + a.real*imag;
return producto;
}
// operador miembro / sobrecargado
Complejo Complejo::operator/ (const Complejo &a)
{
Complejo cociente;
double d = a.real*a.real + a.imag*a.imag;
cociente.real = (real*a.real + imag*a.imag)/d;
cociente.imag = (-real*a.imag + imag*a.real)/d;
return cociente;
}
// operador miembro de asignación sobrecargado
Complejo& Complejo::operator= (const Complejo &a)
{
real = a.real;
imag = a.imag;
return (*this);
}
// operador friend de test de igualdad sobrecargado
int operator== (const Complejo& a, const Complejo& b)
{
if (a.real==b.real && a.imag==b.imag)return 1;
  else return 0;
}
// operador friend de test de desigualdad sobrecargado
int operator!= (const Complejo& a, const Complejo& b)
{
if (a.real!=b.real || a.imag!=b.imag)return 1;
   else return 0;
```

```
    }
    // operador friend << sobrecargado
    ostream& operator << (ostream& co, const Complejo &a)
    {
    co << a.real;
    long fl = co.setf(ios::showpos);
    co << a.imag << "i";
    co.flags(fl);
    return co;
    }


   /**************fichero main.cpp*************************/
    #include "Complejo.h"
    void main(void)
    {
    // se crean dos Complejos con el constructor general
    Complejo c1(1.0, 1.0);
    Complejo c2(2.0, 2.0);
   // se crea un Complejo con el constructor por defecto
    Complejo c3;
   // se da valor a la parte real e imaginaria de c3
    c3.SetReal(5.0);
    c3.SetImag(2.0);
   // se crea un Complejo con el valor por defecto (0.0) del 2°
argumento
    Complejo c4(4.0);
   // se crea un Complejo a partir del resultado de una expresión
   // se utiliza el constructor de copia
    Complejo suma = c1 + c2;
   // se crean tres Complejos con el constructor por defecto
    Complejo resta, producto, cociente;
   // se asignan valores con los operadores sobrecargados
    resta = c1 - c2;
    producto = c1 * c2;
    cociente = c1 / c2;
   // se imprimen los números Complejos con el operador << sobrecargado
    cout << c1 << ", " << c2 << ", " << c3 << ", " << c4 << endl;
    cout << "Primer Complejo: " << c1 << endl;
    cout << "Segundo Complejo: " << c2 << endl;
    cout << "Suma: " << suma << endl;
    cout << "Resta: " << resta << endl;
    cout << "Producto: " << producto << endl;
    cout << "Cociente: " << cociente << endl;

   // se comparan Complejos con los operadores == y != sobrecargados
```

```
   if (c1==c2)
      cout << "Los Complejos son iguales." << endl;
   else
      cout << "Los Complejos no son iguales." << endl;
   if (c1!=c2)
      cout << "Los Complejos son diferentes." << endl;
   else
      cout << "Los Complejos no son diferentes." << endl;
   cout << "Ya he terminado." << endl;
   }
```

4. La Herencia

La mente humana clasifica los *conceptos* de acuerdo a dos dimensiones: *pertenencia* y *variedad*. Se puede decir que el Ford Fiesta es un tipo de coche *(variedad* o, en inglés, una relación del tipo *is* a) y que una rueda es parte de un coche *(pertenencia* o una relación del tipo *has* a). Antes de la llegada de la *herencia*, en C ya se había resuelto el problema de la *pertenencia* mediante las *estructuras*, que podían ser todo lo complejas que se quisiera. Con la *herencia*, como se va a ver en este capítulo, se consigue clasificar los tipos de datos *(abstracciones*) por *variedad*, acercando así un paso más la programación al modo de razonar humano.

Como casi siempre, C++ ha venido ha intentar solucionar, lo que los programadores por su cuenta ya se habían procurado mediante complejas estructuras sobre C. Así, mediante una programación compleja, en la que entra a formar parte importante el pre-procesador, se conseguían los conceptos de herencia tanto simple como múltiple, así como otros aspectos de C++. Todos estos mecanismos, a veces ciertamente con apariencia de enrevesados, han

sido desarrollados respondiendo a necesidades de la programación de alto nivel, por lo que en muchos casos intentaremos hacer hincapié en la aplicación de lo que se explica.

Como ya se ha mencionado, el mecanismo de la herencia es fuertemente característico de la programación orientada a objetos. Por ello, antes de comenzar a ahondar en el concepto, es importante destacar que no deja de ser una herramienta que nos permitirá resolver con comodidad y elegancia ciertos problemas, pero que en sí no es un fin desde el punto de vista de la programación. Uno de los mayores defectos que se dan a la hora de realizar un programa mediante POO, es precisamente el abuso del mecanismo de herencia, y la poca planificación a la hora de definir nuestra estructura. Es bastante común encontrar desarrollos estancados, poco eficientes o poco usados, en C++ debido a la complejidad y poca planificación de la estructura de los objetos. Por este motivo, aún ahora, tras bastantes años con desarrollos sobre C++, siguen saliendo y actualizándose distintas librerías cuya interface y desarrollo siguen siendo en C, frente a otras análogas y de mucho menor éxito realizadas sobre C++.

Por ello, hay que destacar lo siguiente: la herencia es un mecanismo de gran potencia, pero que necesita de una fase de análisis en profundidad previa a su implementación. Si este análisis no se realiza adecuadamente, no sólo perderemos eficiencia, sino que el código terminará corrompido y será aún más ininteligible que sobre C.

4.1. Definición de herencia

La herencia, entendida como una característica de la programación orientada a objetos y más concretamente del C++, permite definir una clase modificando una o más clases ya existentes. Estas modificaciones consisten habitualmente en añadir nuevos miembros (variables o funciones), a la clase que se está definiendo, aunque también se puede redefinir variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de clase base, y la nueva clase que se obtiene se denomina clase derivada. Ésta a su vez puede ser clase base en un nuevo proceso de derivación, iniciando de esta manera una jerarquía de clases. De ordinario las clases base suelen ser más generales que las clases derivadas. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.

Veamos un ejemplo de una herencia simple realizada sobre código C:

```
#define PERSONA     char nombre[40]; \
                    int edad;

typedef struct _persona
      {
      PERSONA
      }persona;

typedef struct _alumno
      {
      PERSONA
      int martricula;
      int curso;
      }alumno;

typedef struct _profesor
      {
      PERSONA
      int departamento;
      int codigo;
      int despacho;
      }profesor;
```

Se observa en el ejemplo cómo podríamos considerar a persona clase base para las otras dos profesor y alumno. Mediante esta estructuración, en determinadas funciones de nuestro programa, podremos entender alumno o profesor como tales, o considerarlos sólo como personas (lo que significaría, acceder a los primeros campos de la estructura, los cuales son comunes a ambos tipos).

En algunos casos una clase no tiene otra utilidad que la de ser clase base para otras clases que se deriven de ella. A este tipo de clases base, de las que no se declara ningún objeto, se les denomina clases base abstractas (ABC, *Abstract Base Class*) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas. Por ejemplo, se puede definir la clase vehiculo para después derivar de ella coche, bicicleta, patinete, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo vehículos.

Las características comunes de estas clases (como una variable que indique si está arado o en marcha, otra que indique su velocidad, la función de arrancar y la de frenar, etc.), pertenecerán a la clase base y las que sean particulares de alguna de ellas pertenecerán sólo a la clase derivada (por ejemplo el número de platos y piñones, que sólo tiene sentido para una

bicicleta, o la función embragar que sólo se aplicará a los vehículos de motor con varias marchas).

Este mecanismo de herencia presenta múltiples ventajas evidentes a primera vista, como la posibilidad de reutilizar código sin tener que escribirlo de nuevo. Esto es posible porque todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

El nivel de acceso protected.

Uno de los problemas que aparece con la herencia es el del control del acceso a los datos. ¿Puede una función de una clase derivada acceder a los datos privados de su clase base? En principio una clase no puede acceder a los datos privados de otra, pero podría ser muy conveniente que una clase derivada accediera a todos los datos de su clase base. Para hacer posible esto, existe el tipo de dato protected. Este tipo de datos es privado para todas aquellas clases que no son derivadas, pero público para una clase derivada de la clase en la que se ha definido la variable como protected.

Por otra parte, el proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base public o private para la clase derivada. En el caso de que la clase base sea public para la clase derivada, ésta hereda los miembros public y protected de la clase base como miembros public y protected, respectivamente. Por el contrario, si la clase base es private para la clase derivada, ésta hereda todos los datos de la clase base como private. La siguiente tabla puede resumir lo explicado en los dos últimos párrafos.

Tipo de dato de la clase base	**Clase derivada de una clase base public**	**Clase derivada de una clase base private**	**Clase sin relación de herencia.**
Private	*No accesible*	*No accesible*	*No accesible*
Protected	*Protected*	*Private*	*No accesible*
Public	*Public*	*Private*	*Accesible (. , ->)*

Tabla 1: Herencia pública y privada.

Como ejemplo, se puede pensar en dos tipos de cuentas bancarias que comparten algunas características y que también tienen algunas diferencias. Ambas cuentas tienen un saldo, un interés y el nombre del titular de la cuenta. La cuenta joven es un tipo de cuenta que requiere la edad del propietario, mientras que la cuenta empresarial necesita el nombre de la empresa. El problema podría resolverse estableciendo una clase base llamada C_Cuenta y creando dos tipos de cuenta derivados de dicha clase base.

Para indicar que una clase deriva de otra es necesario indicarlo en la declaración de la clase derivada, especificando el modo -`public` o `private`- en que deriva de su clase base:

```
class <Clase_Derivada> : [public|private] <Clase_Base>
```

En caso de no especificarse el modo con que se realiza la derivación, entonces se sobreentenderá que la herencia se realiza privadamente.

De esta forma el código necesario para crear esas tres clases mencionadas quedaría de la siguiente forma:

Nótese que algunas de las líneas han tenido que partirse –en concreto los constructores de las clases- debido a su longitud. Esto es válido y conveniente en C++, intentando siempre mantener al máximo la claridad de las definiciones.

```
#include <iostream.h>
class C_Cuenta
 {
 // Variables miembro
 private:
  char *Nombre; // Nombre de la persona
  double Saldo; // Saldo Actual de la cuenta
  double Interes; // Interés aplicado
 public:
 // Constructor
  C_Cuenta(char *nombre, double saldo=0.0, double interes=0.0)
       {
       Nombre = new char[strlen(nombre)+1];
       strcpy(Nombre, nombre);
       SetSaldo(saldo);
       SetInteres(interes);
```

```
        }
 // Destructor
 ~Cuenta(){delete [] Nombre;}
 // Métodos
 char *GetNombre(){ return Nombre; }
  double GetSaldo(){ return Saldo; }
  double GetInteres(){ return Interes; }
  void SetSaldo(double saldo){ Saldo = saldo; }
  void SetInteres(double interes){ Interes = interes; }
  void Ingreso(double cantidad){SetSaldo(GetSaldo()+cantidad);}
  friend ostream& operator<<(ostream& os, C_Cuenta& cuenta){
       os << "Nombre=" << cuenta.GetNombre() << endl;
       os << "Saldo=" << cuenta.GetSaldo() << endl;
       return os;
       }
 };
//CuentaJoven deriva públicamente de la clase Cuenta
       class C_CuentaJoven : public C_Cuenta
 {
 private:
  int Edad;
 public:
  C_CuentaJoven(char *nombre,int edad, double saldo=0.0,
  double interes=0.0):C_Cuenta(nombre, saldo, interes) {
       Edad = laEdad; //especifico de Cuenta Joven
       }
 };
class C_CuentaEmpresarial : public C_Cuenta
 {
 private:
  char *NomEmpresa;
public:
  C_CuentaEmpresarial(char *nombre, char *empresa,
  double saldo=0.0, double interes=0.0)
  :C_Cuenta(nombre, saldo, interes){
       //específico de Cuenta empresarial
       NomEmpresa = new char[strlen(empresa)+1];
       strcpy(NomEmpresa, empresa);
       }
  ~C_CuentaEmpresarial(){delete [] NomEmpresa;}
 };

void main()
 {
 C_CuentaJoven c1("Luis", 18, 10000.0, 1.0);
 C_CuentaEmpresarial c2("Sara", "ELAI Ltd." ,100000.0);
```

```
  cout << c1;
  cout << c2;
  }
```

Métodos y atributos ocultos de la clase base

Si un miembro heredado se redefine en la clase derivada, el nombre redefinido oculta el nombre heredado que ya queda invisible para los objetos de la clase derivada. De esta forma, si en el ejemplo se definiera un nuevo atributo denominado interés en alguna de las dos clases derivadas, entonces se crearía una nueva variable que al tener el mismo nombre impediría al programador ver la heredada de la clase base.

Un ejemplo completo de este aspecto, y de cómo se puede acceder a un atributo o a un método de una clase base oculto por un atributo o método de la clase derivada es el siguiente:

```
class numero
 {
 public:
   numero(int val){valor=val;}
   void imprimir(){cout<<valor;}
 protected:
   int valor;
 };
class decimal:public numero
 {
 public:
   decimal(int val, int dec):numero(val){valor=dec;}
   void imprimir()
       {
       numero::imprimir();
       cout<<','<<valor;
       }
   void mostrar()
       {
       cout<<numero::valor<<','<<valor;
       }
 private:
       int valor;
 };


void main(void)
 {
 decimal num(1,2);
```

```
num.imprimir();

cout<<endl;
num.mostrar();
cout<<endl;
num.numero::imprimir();
}
```

El resultado de ejecutar este programa es el siguiente:

```
1,2
1,2
1
```

Aunque el ejemplo carece de utilidad, es ciertamente ilustrativo de cómo se maneja el operador **scope(::)** cuando tenemos clases derivadas. Se anima al lector a analizar a quien hacen referencia los identificadores valor, e imprimir en cada caso. Nótese que para poder escribir la función mostrar de la clase decimal, ha sido necesario definir valor como protected en la clase base, y que la herencia es pública. De esta forma, el valor no es accesible desde el exterior, pero sí para la clase derivada.

Antes de continuar con la descripción de aspectos específicos de la herencia en C++, es necesario indicar que no todo lo que pertenece a la clase base es heredado. Los elementos que no se heredan son:

- Constructores.
- Destructores.
- Funciones *friend*.
- Funciones y datos estáticos de la clase.
- Operador de asignación (=) sobrecargado.

Los contructores y destructores tienen un tratamiento específico como se verá a continuación. Aunque no se heredan si que se usan en el momento de crearse una instancia de un objeto derivado.

4.2. Construcción y destrucción de clases derivadas: inicializador base.

Un objeto de la clase derivada contiene todos los miembros de la clase base y todos esos miembros deben ser inicializados. Por esta razón el constructor de la clase derivada debe

llamar al constructor de la clase base. Al definir el constructor de la clase derivada se debe especificar un inicializador base. Es decir, en el momento de construirse el objeto, es necesario antes indicar como debe construirse la parte heredada, por lo que es necesario indicar de entre todos los constructores cual de ellos y en con que valores se utilizará.

Como ya se ha dicho las clases derivadas no heredan los constructores de sus clases base. El inicializador base es la forma de llamar a los constructores de las clases base y poder así inicializar las variables miembro heredadas.

Este inicializador base se especifica poniendo, a continuación de los argumentos de un constructor de la clase derivada, el carácter dos puntos (:) y el nombre del constructor de la clase o las clases base, seguido de una lista de argumentos entre paréntesis. Estos argumentos pueden ser función de los argumentos utilizados en el constructor de la clase derivada.

Por ejemplo, el constructor de la clase decimal tenía el siguiente aspecto:

```
decimal(int val, int dec):numero(val){valor=dec;}
```

Con lo que se indica que la parte del objeto correspondiente a número, debe ser inicializado con el primer valor pasado como argumento en el constructor de la clase decimal.

Las listas de argumentos asociadas a las clases base pueden estar formadas por constantes, variables globales o por parámetros que se pasan a la función constructor de la clase derivada.

El inicializador base puede ser omitido en el caso de que la clase base tenga un constructor por defecto. En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

Otro ejemplo un poco más complicado es el de la clase CuentaJoven del apartado anterior. En este caso se utilizan además valores por defecto en el constructor de la clase derivada:

```
C_CuentaJoven(char *nombre, int edad, double saldo=0.0,
 double interes=0.0):C_Cuenta(nombre, saldo, interes)
```

Tiene sentido que se llame a los constructores en el mismo orden en que tienen lugar la derivación de clases. Puesto que la clase base no tiene conocimiento de la clase derivada, cualquier inicialización que ésta necesite realizar es independiente, y posiblemente sea un prerrequisito para cualquier inicialización realizada por la clase derivada, por lo que se debe ejecutar primero.

De forma análoga se operará con el destructor: eliminando primero lo más particular y después lo general. Una función destructor de una clase derivada se ejecuta antes del

destructor de la base. Como la destrucción de un objeto de una clase base implica la destrucción del objeto de la clase derivada, el destructor del objeto derivado se debe ejecutar antes de que se destruya el objeto base.

Por eso una regla importante en la contrucción y destrucción de objetos derivados es la siguiente:

> Los constructores se llaman en el orden de derivación de las clases.
>
> Los destructores se llaman en orden inverso.

El siguiente ejemplo intenta ilustrar este modo de proceder.

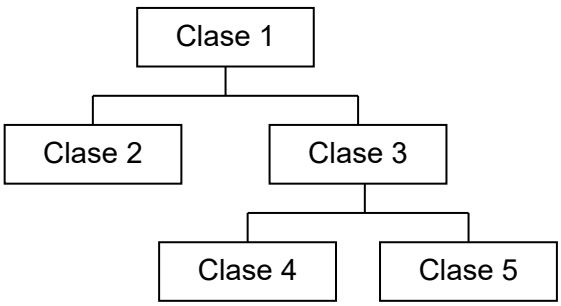
```
#include <iostream.h>

class Base
 {
 public:
   Base() {cout << "\nBase creada\n";}
   ~Base() {cout << "Base destruida\n\n";}
 }

class D_clase1 : public Base
 {
 public:
   D_clase1() {cout << "D_clase1 creada\n";}
   ~D_clase1() {cout << "D_clase1 destruida\n";}
 };

class D_clase2 : public D_clase1
 {
 public:
   D_clase2() {cout << "D_clase2 creada\n";}
   ~D_clase2() {cout << "D_clase2 destruida\n";}
 };

void main()
 {
 D_clase1 d1;
 D_clase2 d2;

 cout << "\n";
 }
```

Este programa generará la siguiente salida:

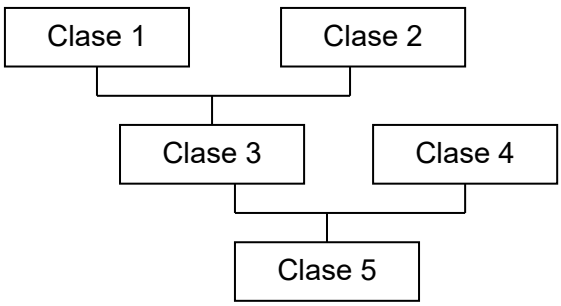
```
Base creada
D_clase1 creada

Base creada
D_clase1 creada
D_clase2 creada

D_clase2 destruida
D_clase1 destruida
Base destruida

D_clase1 destruida
Base destruida
```

4.3. Herencia múltiple

Una clase puede heredar variables y funciones miembro de una o más clases base al mismo tiempo. En el caso de que herede los miembros de una única clase se habla de herencia simple y en el caso de que herede miembros de varias clases base se trata de un caso de herencia múltiple. Esto se ilustra en la siguiente figura:



Herencia Simple: Todas las clases derivadas tienen una única clase base.



Herencia Múltiple: Las clases derivadas tienen varias clases base.

Como ejemplo se puede presentar el caso de que se tenga una clase para el manejo de los datos de la empresa. Se podría definir la clase C_CuentaEmpresarial como la herencia múltiple de dos clases base: la ya bien conocida clase C_Cuenta y nueva clase llamada C_Empresa, que se muestra a continuación:

```
class C_Empresa
 {
 private:
  char *NomEmpresa;
 public:
 C_Empresa(const char*laEmpresa)
       {
       NomEmpresa = new char[strlen(laEmpresa)+1];
       strcpy(NomEmpresa, laEmpresa);
       }
 ~C_Empresa(){ delete [] NomEmpresa; }
 // Otros métodos ...
 };

class C_CuentaEmpresarial : public C_Cuenta, public C_Empresa
 {
 public:
   C_CuentaEmpresarial(char *nombre,char *empresa,
       double saldo=0.0,double interes=0.0)
       :C_Cuenta(nombre, saldo, interes), C_Empresa(empresa)
       {
       // Constructor específico
       }
   // Otros métodos
 };
```

La sintaxis general para heredar una clase de varias clases base es:

```
class <c_derivada> : [public|private] <c_base_1>, [public|private] <base_2>, ...
{
...
}
```

Las clases base se construyen en el orden en el que aparecen en la declaración de la clase derivada de izquierda a derecha. En general, cuando se utiliza una lista de clases base, los constructores se deben llamar en orden de izquierda a derecha. Los destructores se deben llamar en orden de derecha a izquierda.

Los inicializadotes base siguen esta misma sintaxis. Cuando es necesario utilizarlos, se colocan tras la clase base separados por comas. Tal es el caso del ejemplo de la cuenta empresarial expuesto anteriormente.

Hay que destacar que en posteriores lenguajes desarrollados siguiendo la filosofía de POO, la herencia múltiple se ha desechado por ser una posible fuente de errores. La razón principal reside en cómo finalmente van a quedar ordenadas las estructuras de los objetos en memoria. Sin embargo, aunque no existe esta herencia múltiple, podemos generar una serie de clases abstractas intermedias que obtengan el mismo resultado, puesto que el efecto final es el mismo… no es más que una agregación de datos y métodos.

La herencia múltiple puede además producir algunos problemas. En ocasiones puede suceder que en las dos clases base exista una función con el mismo nombre. Esto crea una ambigüedad cuando se invoca a una de esas funciones.

Veamos un ejemplo:

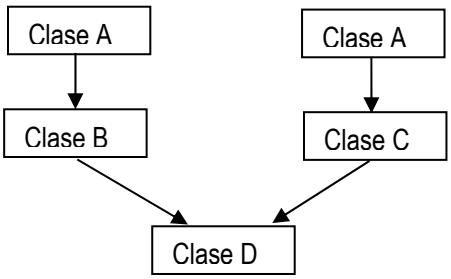
```
#include <iostream.h>
class ClaseA {
public:
 ClaseA() : valorA(10) {}
 int LeerValor(){ return valorA; }
protected:
 int valorA;
};
class ClaseB {
public:
 ClaseB() : valorB(20) {}
 int LeerValor(){ return valorB; }
protected:
 int valorB;
};
class ClaseC : public ClaseA, public ClaseB {};

void  main() {
ClaseC CC;
cout << CC.LeerValor() << endl;// error de compilación
cout << CC.ClaseA::LeerValor() << endl;
cin.get();
}
```

Una solución para resolver la ambigüedad es la que hemos adoptado en el ejemplo. Pero existe otra, también podríamos haber redefinido la función "LeerValor" en la clase derivada de modo que se superpusiese a las funciones de las clases base.

```
#include <iostream.h>
class ClaseA
{
public:
 ClaseA() : valorA(10) {}
 int LeerValor(){ return valorA; }
protected:
 int valorA;
};
class ClaseB
{
public:
 ClaseB() : valorB(20) {}
 int LeerValor(){ return valorB; }
protected:
 int valorB;
};
class ClaseC : public ClaseA, public ClaseB
{
public:
 int LeerValor(){return ClaseA::LeerValor();}
};
void main()
{
ClaseC CC;
cout << CC.LeerValor() << endl;
cin.get();
}
```

4.4. Clases base virtuales

Supongamos que tenemos una estructura de clases como ésta:



La ClaseD heredará dos veces los datos y funciones de la ClaseA, con la consiguiente ambigüedad a la hora de acceder a datos o funciones heredadas de ClaseA. Para solucionar esto se usan las clases virtuales. Cuando derivemos una clase partiendo de una o varias clases base, podemos hacer que las clases base sean virtuales. Esto no afectará a la clase derivada.

Forma de declaración de una clase base virtual:

class *Madre_1* : `virtual` public *Abuela*

{

...

}

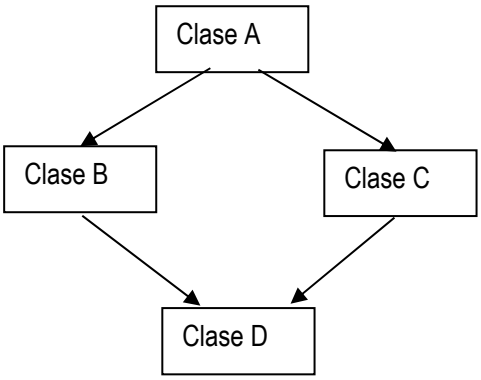
Por ejemplo:

```
class ClaseB : virtual public ClaseA {};
```

Desde el punto de vista de la ClaseB, no hay ninguna diferencia entre ésta declaración y la que hemos usado hasta ahora. La diferencia estará cuando declaramos la ClaseD. Veamos el ejemplo completo:

```
class ClaseB : virtual public ClaseA {};
class ClaseC : virtual public ClaseA {};
class ClaseD : public ClaseB, public ClaseC {};
```

Ahora, la ClaseD sólo heredará una vez la ClaseA. La estructura quedará así:



4.5. Conversiones entre objetos de clases base y clases derivadas

Es posible realizar conversiones o asignaciones de un objeto de una clase derivada a un objeto de la clase base. Es decir se puede ir de lo más particular a lo más general, aunque en esa operación se pierda información, pues haya variables que no tengan a qué asignarse (el número de variables miembro de una clase derivada es mayor o igual que el de la clase de la que deriva).

Por el contrario las conversiones o asignaciones en el otro sentido, es decir de lo más general a lo más particular, no son posibles, porque puede suceder que no se disponga de valores para todas las variables miembro de la clase derivada.

Así pues,en un principio, la siguiente asignación sería **correcta**:

```
Objeto_clase_base = Objeto_clase_derivada
```

mientras que esta otra sería **incorrecta**:

```
Objeto_clase_derivada=Objeto_clase_base
```

Sin embargo, en C++, se permite realizar esta asignación si somos capaces de averiguar que el objeto que aparece como la clase base, en verdad proviene de un objeto de la clase derivada que se quiere asignar. Para ello disponemos del operador *dynamic_cast* el cual comprueba si es posible realizar la asignación.

De igual modo, en C un puntero de un tipo no puede apuntar a un objeto de un tipo diferente. Sin embargo, existe una excepción importante a esta regla que está relacionada solamente con las clases derivadas. En C++ un puntero a una clase base puede apuntar a un objeto de una clase derivada de esta base.

Por tanto se puede guardar la dirección almacenada en un puntero a una clase derivada en un puntero a la clase base. Esto quiere decir que se puede hacer referencia a un objeto de la clase derivada con su dirección contenida en un puntero a la clase base.

Al igual que sucede con los nombres de los objetos, en principio cuando se hace referencia a un objeto por medio de un puntero, el tipo de dicho puntero determina la función miembro que se aplica, en el caso de que esa función se encuentre definida tanto en la clase base como en la derivada.

En definitiva, un puntero a la clase base puede almacenar la dirección de un objeto perteneciente a una clase derivada. Sin embargo, se aplicarán los métodos de la clase a la que pertenezca el puntero, no los de la clase a la que pertenece el objeto. Por eso es conveniente hacer una conversión dinámica forzada.

Para poder utilizar la conversión forzada durante la ejecución, es necesario indicar al compilador que se desea introducir información sobre las clases de los objetos en la ejecución. Para ello, en la práctica se indica como habilitar este modo

En el siguiente ejemplo se pueden ver las distintas posibilidades de asignación que se pueden realizar entre clases base y clases derivadas por medio de las clases CalseA y ClaseB.

```
#include <iostream.h>
class ClaseA{
protected:
  int a;
public:
  ClaseA(int n):a(n){}
  void mostrar(){cout<<a<<endl;}
};
class ClaseB:public ClaseA{
  int b;
public:
  ClaseB(int n, int m):ClaseA(n),b(m){}
  void mostrar(){cout<<a<<',<<b<<endl;};
} ;
void main()
{
ClaseA ca(2);
ClaseB cb(8,4);
ca.mostrar();
cb.mostrar();
//cb=ca; esto generaría un error de compilación (1)
ca=cb;
ca.mostrar();
```

```
cb.mostrar();

ClaseA *c1=new ClaseA(1);
ClaseB *c2=new ClaseB(2,3);

ClaseA *c3=c1,*c4=c2;
c3->mostrar();
c4->mostrar();

ClaseB *c5;
c5 = static_cast<ClaseB *>(c3);
c5->mostrar(); //error en ejecución (2)
c5 = static_cast<ClaseB *>(c4);
c5->mostrar();
delete c1;
delete c2;
}
```

En (1) se produce un error de compilación porque no se permite la asignación desde un objeto base a un objeto derivado por ser el segundo más extenso que el primero, por lo que parte de los atributos no se sabría que valor deben tomar.

Se observa sin embargo que la asignación inversa es válida.

En (2) se muestra una conversión forzosa de tipo. Se obliga al compilador a aceptar que la dirección almacenada en c3 es de un objeto de clase B. En este caso no es así, por lo que si por medio ce c5 ejecutamos un método de la clase B, el resultado puede ser desastroso, puesto que dicho método accederá a una zona de la memoria que no pertenece al objeto y que incluso podría no pertenecer al programa con el consiguiente error del sistema operativo. Hay que destacar que este error no es de compilación sino de ejecución. En la mayoria de los casos, si la zona de memoria accedida es nuestra, lo que se mostrará será el resultado de interpretar la memoria como si ahí estuviera el atributo que estamos consultando.

Por último, en la ultima asignación, el programador se hace responsable de que la conversión forzada es válida. De hecho, en este caso, al ser el objeto apuntado realmente un objeto del tipo ClaseB, la llamada al método mostrar() se realiza correctamente.

Luego el resultado de ejecutar este programa es el siguiente:

```
2
8,4
8
8,4
1
2
1,¿¿¿¿basura informatica????
```

```
2,3
```

Existe en C++ la posibilidad de realizar una conversión de tipos que compruebe si el objeto apuntado por un puntero es o no asignable a un puntero de una clase derivada. Esta conversión –dinámica- consulta el tipo de datos del objeto presente en la memoria durante la ejecución. Si este tipo de datos es convertible al tipo que tiene que recibirlo, entonces se hace la conversión. En caso de no ser así el resultado de esta conversión es cero.

Para poder aplicar dynamic_cast, hay que tener dos precauciones. En primer lugar, esta operación solo puede aplicarse sobre tipos polimórficos. Como se verá en el capítulo siguiente, esto ocurre cuando se tiene que alguno de los métodos de la clase es virtua.

En segundo lugar, hay que indicar al compilador que debe incluir información sobre los tipos de clase en el código. Esta opción de compilación deberá indicarse en las opciones generales del compilador que se esté utilizando.

Dicho esto, el ejemplo quedaría más robusto y elegante si se escribe lo siguiente. En la definición de la clase ClaseA incluiremos un destructor virtual añadiendo la siguiente línea:

```
virtual ~ClaseA(){}
```

Y el final de la función main anterior lo reescribiremos de la forma siguiente:

```
c5= dynamic_cast<ClaseB *>(c3);
if(c5!=NULL)c5->mostrar();
c5= dynamic_cast<ClaseB *>(c4);
if(c5!=NULL)c5->mostrar();
```

En este caso, sólo se realizará la segunda impresión, puesto que el sistema detecta que el objeto apuntado por c3 es de tipo ClaseA, y que por tanto no es una asignación válida la pretendida. Por el contrario, aunque c4 es un puntero de tipo ClaseA, la dirección que contiene es de un objeto de tipo ClaseB, por lo que el operador detecta durante la ejecución que la asignación es válida.

4.6. El constructor de copia y el operador de asignación

Unos de los métodos que se aprendió a sobrecargar y la razón de dicha sobrecarga explicada en el capítulo anterior eran el contructor de copia y la operación de asignación representada por el operador igual. Se comentó que precisamente estas operaciones no se heredaban, y que por tanto deben ser redefinidas por el programador.

Antes de ver como se debe realizar esta operación en caso de tener una clase derivada, es importante aclarar cuando se utiliza un operador o un constructor en el código escrito de nuestro programa.

El siguiente programa utiliza la clase CCuenta ya definida anteriormente:

```
void main()
{
	CCuenta Cuenta1("Tu");
	CCuenta Cuenta2("Yo",1000,10);
	Cuenta1=Cuenta2; //(1)
	CCuenta Cuenta3(Cuenta2); //(2)
	CCuenta Cuenta4=Cuenta2;  //(3)
}
```

En la línea marcada con (1), claramente se utiliza el operador de asignación sobrecargado (si está definido). De igual forma, es claro que en la linea (2) se utiliza el constructor de copia definido. Lo que es aparentemente una excepción es la creación de Cuenta4 en la linea (3). En este caso, aunque aparentemente parece que se utiliza el operador de asignación, se hace uso implícito del constructor de copia.

Una vez aclarado este aspectod ela sintaxis, vamos a ver de que modo se pueden implementar estos dos métodos no heredados en una clase derivada:

```
#include <string.h>
#include <iostream.h>
class Nombre
{
	char *nombre;
public:
	Nombre(char *nom)
	 {
	 nombre=new char[strlen(nom)+1];
	 strcpy(nombre,nom);
	 }
	Nombre(const Nombre &nom)
	 {
	 nombre=new char[strlen(nom.nombre)+1];
	 strcpy(nombre,nom.nombre);
	 }
	Nombre &operator=(const Nombre &nom)
	 {
	 delete [] nombre;
	 nombre=new char[strlen(nom.nombre)+1];
	 strcpy(nombre,nom.nombre);
	 return *this;
	 }
	~Nombre(){delete [] nombre;}
	void mostrar(){cout<<nombre;}
```

```
};
class Alumno:public Nombre
{
	int numero;
public:
	Alumno(char *nom, int num):Nombre(nom),numero(num){}
	Alumno(const Alumno &al):Nombre(al),numero(al.numero){};
	Alumno &operator=(const Alumno &al)
	 {
	 Nombre::operator =(al);
	 numero=al.numero;
	 return *this;
	 }
};

void main()
{
	Alumno uno("Juan Ramírez",43271);
	Alumno dos=uno; //(1)
	Alumno tres("Pepito",41278);
	tres=dos; //(2)

	Nombre extraer(uno); //(3)
	uno.mostrar();
	extraer=dos; //(4)
	dos.mostrar();
}
```

Los métodos que se ejecutan en las instrucciones marcadas son los siguientes: en (1) se ejecuta el constructor de copia de Alumno, pero antes de ejecutarse el código contenido entre las llaves se inicializa por medio del constructor de copia de Nombre, la parte correspondiente a esta clase base. Puesto que *uno* es de clase Alumno, y esta es derivada de nomre, la conversión es váñida e inmediata, tal y como se menciono en el anterior apartado.

En (2) se utiliza el operador de asignación de la clase Alumno. Fíjese que en el código de este operador, se hace un uso explícito del operador de asignación de Nombre.

En (3) se utiliza el constructor de copia de la clase Nombre, puesto que uno se interpreta como un objeto de su clase base. Por último en (4) se hace uso del operador de asignación de la clase base, aprovechando la misma propiedad anterior. Por tanto, el código sacará por pantalla dos veces consecutivas el mensaje *Juan Ramírez*.

Para no tener que reescribir código que hace lo mismo, es habitual que un constructor de copia utilice el propio operador de asignación sobrecargado. De esta forma, se podría reescribir el constructor de copia como:

```
        Alumno(const Alumno &al):Nombre(" ")
        {
        *this=al;
        }
```

En este caso es poco eficiente al no tener un constructor por defecto definido para la clase Nombre, por lo que se hará una reserva de memoria que inmediatamente será de nuevo eliminada por la operación de asignación. Sin embargo es interesante que no exista redundancia en el código de forma que sólo sea necesario modificar o actualizar un sitio para realizar una operación equivalente.

4.7. Ejemplo

El siguiente ejemplo es el código utilizado en las prácticas para la realización de una pequeña jerarquía de clases que permite la realización de una pequeña biblioteca:

```
#include <iostream.h>
typedef std::string cadena;

class CFicha
{
  protected:
      cadena referencia;
    cadena titulo;
  public:
    // Constructor
    CFicha(cadena ref= "", cadena tit= "");
    // Destructor
    virtual ~CFicha() {};
    // Otras funciones
    void AsignarReferencia(cadena ref);
    cadena ObtenerReferencia() ;
    void AsignarTitulo(cadena tit);
    cadena ObtenerTitulo() ;
    void Imprimir();
    void PedirFicha();
    friend void TomarLinea(cadena&);

};
class CFichaLibro : public CFicha
{
  private:
    cadena autor;
    cadena editorial;
```

```
    public:
      // Constructor
        CFichaLibro(cadena ref= "", cadena tit= "", cadena aut=
"", cadena ed= "");

      // Otras funciones
      void AsignarAutor(cadena aut);
      cadena ObtenerAutor();
      void AsignarEditorial(cadena edit);
      cadena ObtenerEditorial();
      void PedirLibro();
      void Imprimir();
  };
  class CFichaRevista : public CFicha
  {
    private:
      int NroDeRevista;
      int Anyo;
    public:
      // Constructor
      CFichaRevista(cadena ref= "", cadena tit= "", int an= 0, int
nro= 0);
      // Otras funciones
      void AsignarNroDeRevista(int nro);
      int ObtenerNroDeRevista() ;
      void AsignarAnyo(int any);
      int ObtenerAnyo() ;
      void PedirRevista();
      void Imprimir();
  };
  class CFichaVolumen : public CFichaLibro
  {
    private:
      int NroDeVolumen;
    public:
      // Constructor
      CFichaVolumen(cadena ref = "", cadena tit= "", cadena aut=
"", cadena edit= "", int Nro= 0);
      // Otras funciones
      void AsignarNroDeVolumen(int);
      int ObtenerNroDeVolumen();
      void PedirVolumen();
      void Imprimir();
  };
  class CBiblioteca
  {
    private:
```

```
    std::vector<CFicha *> ficha;
  public:
    // Constructor
    CBiblioteca(int = 100);
    // Destructor
    ~CBiblioteca();
     // Operador de indexación
    CFicha *operator[](int);
    // Otras funciones
    void AnyadirFicha(CFicha *);
    int longitud();
    void VisualizarFichas();
    bool eliminar(cadena);
    int buscar(cadena, int);
    void VisualizarFicha(int i);
};
ostream& operator<<(std::ostream&, CFicha *);

/************************************************************
Fin de las declaraciones… comienzan las definiciones
************************************************************/

CFicha::CFicha(cadena ref, cadena tit)
{
  referencia = ref;
  titulo = tit;
}

void CFicha::AsignarReferencia(cadena ref)
{
  referencia = ref;
}

cadena CFicha::ObtenerReferencia()
{
  return referencia;
}

void CFicha::AsignarTitulo(cadena tit)
{
  titulo = tit;
}

cadena CFicha::ObtenerTitulo()
{
  return titulo;
}
```

```
void CFicha::PedirFicha()
{
  cout << "Referencia.............: ";
  TomarLinea(referencia);
  cin.ignore(100, '\n');
  cout << "Título.................: ";
  TomarLinea(titulo);
}

void TomarLinea(cadena& var)
{
      char valor[100];
      cin.getline(valor,99);
      var+=valor;
}
void CFicha::Imprimir()
{
      cout << "Referencia:   " << referencia.data() <<endl;
      cout << "Titulo:       " << titulo.data() <<endl;
}
CFichaLibro::CFichaLibro(cadena ref, cadena tit, cadena aut,
  cadena edit): CFicha(ref, tit)
{
  autor = aut;
  editorial = edit;
}

void CFichaLibro::AsignarAutor(cadena aut)
{
  autor = aut;
}

cadena CFichaLibro::ObtenerAutor()
{
  return autor;
}

void CFichaLibro::AsignarEditorial(cadena edit)
{
  editorial = edit;
}

cadena CFichaLibro::ObtenerEditorial()
{
  return editorial;
}
```

```
   void CFichaLibro::PedirLibro()
   {
       PedirFicha();
       cout << "Autor.................: ";
       TomarLinea(autor);
       cout << "Editorial.............: ";
       TomarLinea(editorial);
   }
   void CFichaLibro::Imprimir()
   {
         cout << "Autor:        " << autor.data() <<endl;
         cout << "Editorial:    " << editorial.data() <<endl;
   }
   CFichaRevista::CFichaRevista(cadena ref, cadena tit, int
nroderev, int anyo) : CFicha(ref, tit), NroDeRevista(nroderev),
Anyo(anyo){}

   void CFichaRevista::AsignarNroDeRevista(int nroderev)
   {
     NroDeRevista = nroderev;
   }

   int CFichaRevista::ObtenerNroDeRevista()
   {
     return NroDeRevista;
   }

   void CFichaRevista::AsignarAnyo(int anyo)
   {
     Anyo = anyo;
   }

   int CFichaRevista::ObtenerAnyo()
   {
     return Anyo;
   }
   void CFichaRevista::PedirRevista()
   {
       PedirFicha();
       cout << "Nro. de la revista.....: ";
       cin >> NroDeRevista;
       cout << "Año de publicación.....: ";
       cin >> Anyo;
   }
   void CFichaRevista::Imprimir()
   {
         cout << "Nro Revista:  " << NroDeRevista <<endl;
```

```
        cout << "Año      :    " << Anyo <<endl;
  }
  CFichaVolumen::CFichaVolumen(cadena ref, cadena tit, cadena aut,
cadena edit, int nrodevol) : CFichaLibro(ref, tit, aut,
edit),NroDeVolumen(nrodevol){}

  void CFichaVolumen::AsignarNroDeVolumen(int nrodevol)
  {
    NroDeVolumen = nrodevol;
  }

  int CFichaVolumen::ObtenerNroDeVolumen()
  {
    return NroDeVolumen;
  }
  void CFichaVolumen::PedirVolumen()
  {
    PedirLibro();
    cout << "Nro. del volumen.......: ";
    cin>>NroDeVolumen;
  }
  void CFichaVolumen::Imprimir()
  {
        cout << "Tomo:         " << NroDeVolumen <<endl;
  }
  CBiblioteca::CBiblioteca(int n)
  {
    if (n < 0) n = 1;
    ficha.reserve(n);
  }

  CBiblioteca::~CBiblioteca()
  {
    for (int i = 0; i < ficha.size(); i++)delete ficha[i];
  }

  // Indexación
  CFicha *CBiblioteca::operator[](int i)
  {
    if (i >= 0 && i < ficha.size()) return ficha[i];
    else
      {
      cout << "error: índice fuera de límites\n";
      return 0;
      }
  }
```

```
// Asignar un objeto al array
void CBiblioteca::AnyadirFicha(CFicha *pficha)
{
 //añade una ficha al final del vector
 ficha.push_back(pficha);
}

int CBiblioteca::longitud()
{
  //nos dice cuantos elementos se han introducido en el vector
  return ficha.size();
}

// Visualizar todos los objetos
void CBiblioteca::VisualizarFichas()
{
  if (ficha.size()==0)
       {
       cout << "Biblioteca vacía\n";
      return;
       }
  for (int i = 0; i < ficha.size(); i++)VisualizarFicha(i);
}
void CBiblioteca::VisualizarFicha(int i)
{
if (ficha.size()>i)
  {
  cout<<"-----------------------------------"<<endl;
  ficha[i]->Imprimir();
  if (CFichaLibro *p = dynamic_cast<CFichaLibro *>(ficha[i]))
        p->Imprimir();

  if (CFichaVolumen *p=dynamic_cast<CFichaVolumen *>(ficha[i]))
        p->Imprimir();

  if (CFichaRevista *p=dynamic_cast<CFichaRevista *>(ficha[i]))
        p->Imprimir();

  cout<<"-----------------------------------"<<endl;
  }

}
```

```
bool CBiblioteca::eliminar(cadena refe)
{
  // Buscar la ficha y eliminar el objeto
  for (unsigned int i = 0; i < ficha.size(); i++)
    if (refe == ficha[i]->ObtenerReferencia())
    {
      delete ficha[i];
      ficha.erase(ficha.begin()+i);
      return true;
    }
  return false;
}

int CBiblioteca::buscar(cadena str, int pos)
{
  // Buscar un objeto y devolver su posición
  cadena titu, refe;
  if (str.empty()) return -1;
  if (pos < 0) pos = 0;
  for (unsigned int i = pos; i < ficha.size(); i++)
   {
    // Buscar por el título
    titu = ficha[i]->ObtenerTitulo();
    if (titu.empty()) continue;
    // ¿str está contenida en titu?
    if (titu.find(str) != cadena::npos)
      return i;
    // Buscar por la referencia
    refe = ficha[i]->ObtenerReferencia();
    if (refe.empty()) continue;
    // ¿str es la referencia?
    if (str == refe)
      return i;
   }
  return -1;
}
CFicha *leerDatos(int op);
int leerDato();
int menu();



/*********************************************************
*************   MAIN   ********************************
*********************************************************/
void main()
{
```

```
// Crear un objeto con cero elementos
CBiblioteca bibli;
CFicha *unaFicha = 0;

int opcion = 0, pos = -1;
cadena cadenabuscar;
cadena refe;
bool eliminado = false;

do
{
 opcion = menu();
 switch (opcion)
  {
   case 1: // añadir
     cout << "Tipo de ficha < 1-(rev), 2-(lib), 3-(vol) >: ";
     do
                 opcion = (int)leerDato();
      while (opcion < 1 || opcion > 3);
      unaFicha = leerDatos(opcion);
      bibli.AnyadirFicha(unaFicha);
    break;
    case 2: // buscar
      cout << "Título total o parcial, o referencia: ";
      TomarLinea(cadenabuscar);
      pos = bibli.buscar(cadenabuscar, 0);
      if (pos == -1)
                 {
                 if (bibli.longitud() != 0)
                        cout << "búsqueda fallida\n";
                 else
                        cout << "no hay fichas\n";
                 }
      else
        bibli.VisualizarFicha(pos);
    break;
    case 3: // buscar siguiente
      pos = bibli.buscar(cadenabuscar, pos + 1);
      if (pos == -1)
        if (bibli.longitud() != 0)
          cout << "búsqueda fallida\n";
        else
          cout << "no hay fichas\n";
      else
        bibli.VisualizarFicha(pos);
      break;
    case 4: // eliminar ficha
```

```
          cout << "Referencia: ";
               TomarLinea(refe);
          eliminado = bibli.eliminar(refe);
          if (eliminado)
               cout << "registro eliminado\n";
          else
               {
               if (bibli.longitud() != 0)
                cout << "referencia no encontrada\n";
               else
                cout << "no hay fichas\n";
               }
          break;
        case 5: // listado de la biblioteca
          bibli.VisualizarFichas();
          break;
        }
    }
    while(opcion != 6);
}

/*********************************************************
FUNCION: int menu()
ARGUMENTOS: ninguno
RETORNO: int. Devolverá la opción escogida (1-6)
DESCRIPCION: Imprime por pantalla el menú principal y
       solicita al usuario que seleccione. No retornará un
       valor hasta que la selección sea válida.
*********************************************************/
int menu()
{
  cout << "\n\n";
  cout << "1.  Añadir ficha\n";
  cout << "2.  Buscar ficha\n";
  cout << "3.  Buscar siguiente\n";
  cout << "4.  Eliminar ficha\n";
  cout << "5.  Listado de la biblioteca\n";
  cout << "6.  Salir\n";
  cout << endl;
  cout << "   Opción: ";
  int op;
  do
    op = (int)leerDato();
  while (op < 1 || op > 6);
  return op;
}
```

```
/*********************************************************
FUNCION: CFicha *leerDatos(int op)
ARGUMENTOS:
      int op: Indica el tipo de Ficha que se tiene que crear.
          1= Revista, 2= Libro, 3=Volumen
RETORNO: CFicha *. Devolverá un puntero a la ficha creada
      dinámicamente en función de la seleccion y los valores
      introducidos por el usuario.
DESCRIPCION: Funcion principal que es llamada cada vez que
      quiera crear una ficha de cualquire tipo.
*********************************************************/
CFicha *leerDatos(int op)
{
CFicha *obj=NULL;
switch(op)
      {
      case 1:
            {
            CFichaRevista *aux = new CFichaRevista();
            aux->PedirRevista();
            obj=aux;
            }
      break;
      case 2:
            {
            CFichaLibro *aux = new CFichaLibro();
            aux->PedirLibro();
            obj=aux;
            }
      break;
      case 3:
            {
            CFichaVolumen *aux = new CFichaVolumen();
            aux->PedirVolumen();
            obj=aux;
            }
      }
  return obj;
}
/*********************************************************
FUNCION: int leerDatos()
ARGUMENTOS: ninguno
RETORNO: int. Retorna el número leido
DESCRIPCION: Funcion especial para hacer más cómoda y
      segura la lectura de números del teclado.
*********************************************************/
int leerDato()
```

```
{
  int dato = 0;
  cin >> dato;
  while (cin.fail()) // si el dato es incorrecto, limpiar el
  {                  // búfer y volverlo a leer
    cout << '\a';
    cin.clear();
    cin.ignore(100, '\n');
    cin >> dato;
  }
  // Eliminar posibles caracteres sobrantes
  cin.clear();
  cin.ignore(100, '\n');

  return dato;
}
```

5. El Polimorfismo

Con este capítulo se tendrán las herramientas básicas para el desarrollo de un buen programa en C++. Aún quedaría el uso de plantillas, pero estas pueden ser sustituidas en muchos casos por el polimorfismo, y desde luego son más difíciles de utilizar por un programador novel que los mecanismos hasta ahora explicados.

Antes de adentrarse en el concepto de polimorfismo, su utilidad y su casuística, es necesario aclarar algún concepto en lo que se refiere a la superposición y la sobrecarga:

5.1. Superposición y sobrecarga

Tal y como se vió en el capítulo anterior en una clase derivada se puede definir una función que ya existía en la clase base. Esto se conoce como "overriding", o superposición de una función. La definición de la función en la clase derivada oculta la definición previa en la clase base. En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo:

```
<objeto>.<clase_base>::<método>;
```

Cuando se superpone una función, se ocultan **todas** las funciones con el mismo nombre en la clase base. Supongamos que hemos sobrecargado la función de la clase base que después volveremos a definir en la clase derivada:

```
#include <iostream.h>

class ClaseA
{
public:
 void Incrementar() { cout << "Suma 1" << endl; }
 void Incrementar(int n) { cout << "Suma " << n << endl; }
};
class ClaseB : public ClaseA
{
public:
 void Incrementar() { cout << "Suma 2" << endl; }
};

int main()
{
ClaseB objeto;
objeto.Incrementar();
objeto.Incrementar(10); //¿Existe este método?
objeto.ClaseA::Incrementar();
objeto.ClaseA::Incrementar(10);
cin.get();
return 0;
}
```

Ahora bien, no es posible acceder a ninguna de las funciones superpuestas de la clase base, aunque tengan distintos valores de retorno o distinto número o tipo de parámetros. Todas las funciones "incrementar" de la clase base han quedado ocultas, y sólo son accesibles mediante el nombre completo. Por ello la línea marcada dará un error de compilación, puesto que no existe la función incrementar definida en la claseB que reciba como argumento un tipo de datos al que 10 sea convertible.

Tras corregir el error eliminando esta línea, la salida será:

```
Suma 2
Suma 1
Suma 10
```

5.2. Polimorfismo

Ha llegado el momento de introducir uno de los conceptos más importantes de la programación orientada a objetos: *el polimorfismo*.

En lo que concierne a clases, el polimorfismo en C++, llega a su máxima expresión cuando las usamos junto con punteros o con referencias. Como se ha visto, C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base. En eso consiste o se basa el polimorfismo. Hata ahora sólo podemos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán inaccesibles. Esto es debido a que el compilador decide en tiempo de compilación que métodos y atributos están disponibles en función del contenedor.

Para ilustrarlo vamos a ver un ejemplo sobre una estructura de clases basado en la clase "Persona" y dos clases derivadas "Empleado" y "Estudiante":

```
#include <iostream.h>
#include <string.h>

class Persona
 {
 public:
  Persona(char *n) { strcpy(nombre, n); }
  void VerNombre() { cout << nombre << endl; }
 protected:
  char nombre[30];
 };
class Empleado : public Persona
 {
 public:
  Empleado(char *n) : Persona(n) {}
  void VerNombre()
   {
   cout << "Emp: " << nombre << endl;
   }
 };

class Estudiante : public Persona
 {
 public:
  Estudiante(char *n) : Persona(n) {}
  void VerNombre()
   {
```

```
    cout << "Est: " << nombre << endl;
    }
 };
void main() {
Persona *Pepito = new Estudiante("Jose");
Persona *Carlos = new Empleado("Carlos");
Carlos->VerNombre();
Pepito->VerNombre();
delete Pepito;
delete Carlos;
}
```

Por simplificar el código, se ha utilizado memoria estática en vez de dinámica. Evidentemente este es un error funcional grave puesto que el usuario de la clase puede provocar un fallo en la ejecución haciendo un uso correcto de los métodos de interfaz. En cualquier caso, suponiendo que tenemos cuidado, la salida del programa, como es previsible será la siguiente:

Carlos

Jose

Podemos comprobar que se ejecuta la versión de la función "VerNombre" que hemos definido para la clase base, y no la de las clases derivadas. Esto es debido a que la función que se ejecuta se resuelve en tiempo de EJECUCION atendiendo no al tipo de objeto apuntado, sino al tipo del apuntador.

Por eso, si escribimos:

```
Estudiante *Pepito=new Estudiante("Jose");
Empleado *Carlos=new Empleado("Carlos");
```

Entonces si que se ejecutará el método *VerNombre* superpuesto.

Esto mismo sucede con las referencias. Las siguientes líneas de código son totalmente válidas, siendo el efecto análogo al obtenido por medio de punteros:

```
Estudiante Jose("Jose");
Persona &pJose=Jose;
pJose.VerNombre();
```

Es decir, en este caso, atendiendo al recipiente y no a lo apuntado, se utilizará de nuevo el método *VerNombre* de la clase Persona, a pesar de que realmente *pJose* es un alias de un objeto de tipo Estudiante. Ya observamos que un objeto se comporta de distinta forma en función de con que se lo referencie o apunte.

Sin embargo, parece interesante que cada objeto se comporte como debe independientemente del recipiente, es decir, independientemente de con que se referencie o apunte… esto nos lleva al concepto de polimorfismo de la mano de los denominados métodos virtuales.

Métodos virtuales

Un método virtual es un método de una clase base que puede ser redefinido en cada una de las clases derivadas de esta, y que una vez redefinido puede ser accedido por medio de un puntero o una referencia a la clase base, resolviéndose entonces la llamada en función del objeto referido en vez de en función de con qué se hace la referencia.

Que viene a significar que si en una clase base definimos un método como virtual, si este método es superpuesto por una clase derivada, al invocarlo utilizando un puntero o una referencia de la clase base, ¡se ejecutará el método de la clase derivada!.

Cuando una clase tiene algún método virtual –bien directamente, bien por herencia- se dice que dicha clase es polimórfica.

Para declarar un método como virtual se utiliza la siguiente sintaxis:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];
```

Como siempre, la mejor forma de entender los conceptos en programación es ver ejemplos de código con su resultado.

Modifiquemos en el ejemplo anterior la declaración de la clase base "Persona", haciendo que el método VerNombre sea virtual:

```
class Persona
 {
 public:
  Persona(char *n) { strcpy(nombre, n); }
  virtual void VerNombre() {cout << nombre << endl;}
protected:
 char nombre[30];
};
```

Ahora ejecutemos el programa de nuevo. Observamos entonces que la salida es diferente:

Emp: Carlos

Est: Jose

Ahora, al llamar a *Pepito->VerNombre()* se invoca a la función *VerNombre* de la clase *Estudiante*, y al llamar a *Carlos->VerNombre()* se invoca a la función de la clase *Empleado*, a pesar de que tanto *Pepito* como *Carlos* son punteros a la clase *Persona*.

De igual forma ocurrirá si en vez de utilizar punteros, hacemos uso de referencias:

```
void main()
{
Estudiante Pepito("Jose");
Empleado Carlos("Carlos");
Persona &rPepito = Pepito; // Referencia como Persona
Persona &rCarlos = Carlos; // Referencia como Persona
rCarlos.VerNombre(); //cada objeto ejecuta su método
rPepito.VerNombre(); //en vez de utilizar el de Persona
}
```

Por tanto, la idea central del polimorfismo es la de poder llamar a funciones distintas aunque tengan el mismo nombre, según la clase a la que pertenece el objeto al que se aplican. Esto es imposible utilizando nombres de objetos: siempre se aplica la función miembro de la clase correspondiente al nombre del objeto, y esto se decide en tiempo de compilación.

Sin embargo, utilizando punteros puede conseguirse el objetivo buscado. Recuérdese que un puntero a la clase base puede contener direcciones de objetos de cualquiera de las clases derivadas.

Antes de ahondar y ver más ejemplos de métodos virtuales se van a establecer algunas características del mecanismo de virtualidad:

- Una vez que una función es declarada como virtual, lo seguirá siendo en las clases derivadas, es decir, **la propiedad virtual se hereda**.
- Si la función virtual no se define exactamente con **el mismo tipo de valor de retorno y el mismo número y tipo de parámetros** que en la clase base, no se considerará como la misma función, sino como una función superpuesta.
- El nivel de acceso no afecta a la virtualidad de las funciones. Es decir, una función virtual puede declararse como privada en las clases derivadas aun siendo pública en la clase base, pudiendo por tanto ejecutarse ese método privado desde fuera por medio de un puntero a la clase base.

- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado.
- Una llamada a un método normal se resuelve siempre en función del tipo de la referencia o puntero utilizado.
- Una llamada a un método virtual especificando la clase, exige la utilización del operador de resolución de ámbito ::, lo que suprime el mecanismo de ritualidad. Evidentemente este mecanismo solo podrá utilizarse para el método de la misma clase del contenedor o de clases base del mismo.
- Por su modo de funcionamiento interno (es decir, por el modo en que realmente trabaja el ordenador) las funciones virtuales son un poco menos eficientes que las funciones normales.

Vamos a ver algún ejemplo adicional que ayude a entender el polimorfismo y las clases polimórficas.

```
#include <iostream.h>

class Base
{
public:
	virtual void ident(){cout<<"Base"<<endl;}
};
class Derivada1:public Base
{
public:
	void ident(){cout<<"Primera derivada"<<endl;}
};
class Derivada2:public Base
{
public:
	void ident(){cout<<"Segunda derivada"<<endl;}
};
class Derivada3:public Base
{
public:
	void ident(){cout<<"Tercera derivada"<<endl;}
};

void main()
{
	Base base,*pbase;
	Derivada1 primera;
	Derivada2 segunda;
	Derivada3 tercera;
```

```
        pbase=&base;
        pbase->ident();
        pbase=&primera;
        pbase->ident();
        pbase=&segunda;
        pbase->ident();
        pbase=&tercera;
        pbase->ident();
}
```

Evidentemente, el resultado de ejecutar este código es el siguiente:

```
Base
Primera derivada
Segunda derivada
Tercera derivada
```

De nuevo, al ser el método virtual se ejecuta la función del objeto apuntado, independientemente de que se apunte con un puntero de tipo Base.

Si escribiéramos lo siguiente en el cuerpo del main:

```
        pbase=&base;
        pbase->Base::ident();
        pbase=&primera;
        pbase->Base::ident();
        pbase=&segunda;
        pbase->Base::ident();
        pbase=&tercera;
        pbase->Base::ident();
```

Entonces el resultado de la ejecución sería el siguiente:

```
Base
Base
Base
Base
```

Ahora lo que haremos será poner la modificación de virtualidad en la clase Derivada2, y lo quitamos de la clase Base. En este caso el resultado de la ejecución será de nuevo cuatro veces Base. Esto es debido a que desde el punto de vista del puntero que contenedor (de tipo Base) la función ya no es virtual y por tanto se decide en tiempo de compilación como cualquier método normal. Para poder acceder al método definido por las

clases derivadas al menos será necesario que utilicemos un puntero de tipo Derivada2 para aquellas clases que son polimórficas (en este caso sólo lo sería Derivada3).

Por último, si volvemos al programa original, y establecemos que el método ident es privado en la clase Derivada2, se observa que no afecta para nada al funcionamiento de nuestro programa.

El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código original. Esto quiere decir que si desea añadir una nueva clase derivada es suficiente con establecer la clase de la que deriva, definir sus nuevas variables y funciones miembro, y compilar esta parte del código, ensamblándolo después con lo que ya estaba compilado previamente.

El siguiente ejemplo es un poco más largo y complicado respecto de los anteriores, pero es más elocuente ante las posibilidades que nos ofrece el polimorfismo:

```
#include <iostream.h>

class Vehiculo
{
public:
      virtual void muestra(ostream &co){}
      virtual void rellena(){}
      friend ostream& operator <<(ostream &co,Vehiculo &ve)
            {
            ve.muestra(co);
            return co;
            }
};
class Coche:public Vehiculo
{
protected:
      char marca[20];
      char modelo[20];
public:
      void muestra(ostream &co)
      {
      co<<"Coche marca "<<marca<<" modelo "<<modelo<<endl;
      }
      void rellena()
      {
      cout<<"¿Marca?:";
      cin>>marca;
      cout<<"¿Modelo?:";
      cin>>modelo;
```

```
        cin.clear();
        }
};
class Camion:public Coche
{
private:
        int carga;
public:
        void muestra(ostream &co)
        {
        co<<"Camion marca "<<marca<<" modelo "<<modelo<<endl;
        co<<"\tCapacidad de carga: "<<carga<< "Kg."<<endl;
        }
        void rellena()
        {
        Coche::rellena();
        cout<<"¿Carga máxima?:";
        cin>>carga;
        cin.clear();
        }
};

void main()
{
        Vehiculo *flota[4];
        int seleccion=0;
        for(int i=0;i<3;i++)
                {
                cout<<"Seleccione tipo del vehiculo "<<i<<":\n";
                cout<<"(1-Camión, 2-Coche): ";

                while((seleccion<1)||(seleccion>2))
                        cin>>seleccion;
                switch(seleccion)
                        {
                    case 1: flota[i]=new Camion;break;
                        case 2: flota[i]=new Coche;break;
                        }
                seleccion=0;
                flota[i]->rellena();
                }
        cout<<"\n Estos son los vehiculos introducidos:\n";
        for(i=0;i<3;i++)cout<<i<<":"<<*flota[i];
}
```

Un ejemplo de ejecución de este programa es el siguiente:

```
Seleccione tipo del vehiculo 0:
(1-Camión, 2-Coche): 1
¿Marca?:SCANIA
¿Modelo?:SuperTruck
¿Carga máxima?:22000
Seleccione tipo del vehiculo 1:
(1-Camión, 2-Coche): 2
¿Marca?:Seat
¿Modelo?:Ibiza
Seleccione tipo del vehiculo 2:
(1-Camión, 2-Coche): 2
¿Marca?:Renault
¿Modelo?:Twingo

 Estos son los vehiculos introducidos:
0:Camion marca SCANIA modelo SuperTruck
        Capacidad de carga: 22000Kg.
1:Coche marca Seat modelo Ibiza
2:Coche marca Renault modelo Twingo
```

Se observa entonces como es posible almacenar y tratar los objetos como iguales atendiendo a que heredan de una misma clase base, y sin embargo, estos mismos objetos son capaces de realizar acciones distintas o especializadas cuando se ejecutan sus métodos virtuales. Esto es la potencia del polimorfismo, y tal vez gracias al mismo alguno comienze a vislumbrar el porque se parecen tanto y a la vez son distintos los programas que se manejan sobre Windows.

Implementación del mecanismo de virtualidad

A continuación se explica, sin entrar en gran detalle, el funcionamiento de las funciones virtuales. Cada clase que utiliza funciones virtuales tiene un vector de punteros, uno por cada función virtual, llamado v-table. Cada uno de los punteros contenidos en ese vector apunta a la función virtual apropiada para esa clase, que será, habitualmente, la función virtual definida en la propia clase. En el caso de que en esa clase no esté definida la función virtual en cuestión, el puntero de v-table apuntará a la función virtual de su clase base más próxima en la jerarquía, que tenga una definición propia de la función virtual. Esto quiere decir que buscará primero en la propia clase, luego en la clase anterior en el orden jerárquico y se irá subiendo en ese orden hasta dar con una clase que tenga definida la función buscada.

Cada objeto creado de una clase que tenga una función virtual contiene un puntero oculto a la v-table de su clase. Mediante ese puntero accede a su v-table correspondiente y a través de esta tabla accede a la definición adecuada de la función virtual. Es este trabajo extra el que hace que las funciones virtuales sean menos eficientes que las funciones normales.

5.3. Virtualidad en destructores y constructores.

Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor. Un destructor es una función como las demás, por lo tanto, si destruimos un objeto referenciado mediante un puntero a la clase base, y el destructor no es virtual, estaremos llamando al destructor de la clase base. Esto puede ser desastroso, ya que nuestra clase derivada puede tener más tareas que realizar en su destructor que la clase base de la que procede. Si no posiblemente sería necesario definirlo.

Como norma general, el constructor de la clase base se llama antes que el constructor de la clase derivada. Con los destructores, sin embargo, sucede al revés: el destructor de la clase derivada se llama antes que el de la clase base.

Por esa razón, en el caso de que se borre, aplicando delete, un puntero a un objeto de la clase base que apunte a un objeto de una clase derivada, se llamará al destructor de la clase base, en vez de al destructor de la clase derivada, que sería lo adecuado. La solución a este problema consiste en declarar como virtual el destructor de la clase base. Esto hace que automáticamente los destructores de las clases derivadas sean también virtuales, a pesar de tener nombres distintos. De este modo, al aplicar delete a un puntero de la clase base que puede apuntar a un objeto de ese tipo o a cualquier objeto de una clase derivada, se aplica el destructor adecuado en cada caso.

Este problema no se presenta con los constructores y por eso no existe ningún tipo de constructor virtual o similar. Por eso los constructores no pueden ser virtuales. Esto puede ser un problema en ciertas ocasiones. Por ejemplo, el constructor de copia no hará siempre aquello que esperamos que haga. En general no debemos usar el constructor copia cuando usemos punteros a clases base. Para solucionar este inconveniente se suele crear una función virtual "clonar" en la clase base que se superpondrá para cada clase derivada.

Por ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;
class Persona {
public:
```

```
Persona(char *n) { strcpy(nombre, n); }
Persona(const Persona &p);
virtual void VerNombre() {
cout << nombre << endl;
}
virtual Persona* Clonar() { return new Persona(*this); }
protected:
char nombre[30];
};
Persona::Persona(const Persona &p) {
strcpy(nombre, p.nombre);
cout << "Per: constructor copia." << endl;
}
class Empleado : public Persona {
public:
Empleado(char *n) : Persona(n) {}
Empleado(const Empleado &e);
void VerNombre() {
cout << "Emp: " << nombre << endl;
}
virtual Persona* Clonar() { return new Empleado(*this); }
};
Empleado::Empleado(const Empleado &e) : Persona(e) {
cout << "Emp: constructor copia." << endl;
}
class Estudiante : public Persona {
public:
Estudiante(char *n) : Persona(n) {}
Estudiante(const Estudiante &e);
void VerNombre() {
cout << "Est: " << nombre << endl;
}
virtual Persona* Clonar() {
return new Estudiante(*this);
}
};
Estudiante::Estudiante(const Estudiante &e) : Persona(e) {
cout << "Est: constructor copia." << endl;
}
int main() {
Persona *Pepito = new Estudiante("Jose");
Persona *Carlos = new Empleado("Carlos");
Persona *Gente[2];
Carlos->VerNombre();
Pepito->VerNombre();
Gente[0] = Carlos->Clonar();
Gente[0]->VerNombre();
```

```
Gente[1] = Pepito->Clonar();
Gente[1]->VerNombre();
delete Pepito;
delete Carlos;
delete Gente[0];
delete Gente[1];
cin.get();
return 0;
}
```

Hemos definido el constructor copia para que se pueda ver cuando es invocado. La salida es ésta:

```
Emp: Carlos
Est: Jose
Per: constructor copia.
Emp: constructor copia.
Emp: Carlos
Per: constructor copia.
Est: constructor copia.
Est: Jose
```

Este método asegura que siempre se llama al constructor copia adecuado, ya que se hace desde una función virtual.

Si un constructor llama a una función virtual, ésta será siempre la de la clase base. Esto es debido a que el objeto de la clase derivada aún no ha sido creada.

5.4. Funciones virtuales puras y clases abstractas

Habitualmente las funciones virtuales de la clase base de la jerarquía no se utilizan porque en la mayoría de los casos no se declaran objetos de esa clase, y/o porque todas las clases derivadas tienen su propia definición de la función virtual. Sin embargo, incluso en el caso de que la función virtual de la clase base no vaya a ser utilizada, debe declararse.

De todos modos, si la función no va a ser utilizada no es necesario definirla, y es suficiente con declararla como función virtual pura. Una función virtual pura se declara así:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) = 0;
```

La única utilidad de esta declaración es la de posibilitar la definición de funciones virtuales en las clases derivadas. De alguna manera se puede decir que la definición de una función como virtual pura hace necesaria la definición de esa función en las clases derivadas, a la vez que imposibilita su utilización con objetos de la clase base.

Al definir una función como virtual pura hay que tener en cuenta que:

- No hace falta definir el código de esa función en la clase base.
- No se pueden definir objetos de la clase base, ya que no se puede llamar a las funciones virtuales puras.
- Sin embargo, es posible definir punteros a la clase base, pues es a través de ellos como será posible manejar objetos de las clases derivadas.

Se denomina **clase abstracta** a aquella que contiene una o más funciones virtuales puras. El nombre proviene de que no puede existir ningún objeto de esa clase. Si una clase derivada no redefine una función virtual pura, la clase derivada la hereda como función virtual pura y se convierte también en clase abstracta. Por el contrario, aquellas clases derivadas que redefinen todas las funciones virtuales puras de sus clases base reciben el nombre de clases derivadas concretas, nomenclatura únicamente utilizada para diferenciarlas de las antes mencionadas.

Aparentemente puede parecer que carece de sentido definir una clase de la que no va a existir ningún objeto, pero se puede afirmar, sin miedo a equivocarse, que la abstracción es una herramienta imprescindible para un correcto diseño de la Programación Orientada a Objetos.

Habitualmente las clases superiores de muchas jerarquías de clases son clases abstractas y las clases que heredan de ellas definen sus propias funciones virtuales, convirtiéndose así en funciones concretas.

No es posible crear objetos de una clase abstracta, estas clases sólo se usan como clases base para la declaración de clases derivadas.

Las funciones virtuales puras serán aquellas que siempre se definirán en las clases derivadas, de modo que no será necesario definirlas en la clase base.

A menudo se mencionan las clases abstractas como tipos de datos abstractos, en inglés: Abstract Data Type, o resumido ADT.

Como consecuencia de lo dicho se pueden resumir dos reglas a tener en cuenta cuando se utilizan clases abstractas:

- No está permitido crear objetos de una clase abstracta.
- Siempre hay que definir todas las funciones virtuales de una clase abstracta en sus clases derivadas, no hacerlo así implica que la nueva clase derivada será también abstracta.

Para crear un ejemplo de clases abstractas, recurriremos de nuevo a nuestra clase "Persona". Haremos que ésta clase sea abstracta. De hecho, en nuestros programas de ejemplo nunca hemos declarado un objeto "Persona".

Veamos un ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;
class Persona {
public:
Persona(char *n) { strcpy(nombre, n); }
virtual void Mostrar() = 0;
protected:
char nombre[30];
};
class Empleado : public Persona {
public:
Empleado(char *n, int s) : Persona(n), salario(s) {}
void Mostrar() const;
int LeeSalario() const { return salario; }
void ModificaSalario(int s) { salario = s; }
protected:
int salario;
};
void Empleado::Mostrar() const {
cout << "Empleado: " << nombre
<< ", Salario: " << salario
<< endl;
}
class Estudiante : public Persona {
public:
Estudiante(char *n, float no) : Persona(n), nota(no) {}
void Mostrar() const;
float LeeNota() const { return nota; }
void ModificaNota(float no) { nota = no; }
protected:
float nota;
};
void Estudiante::Mostrar() const {
cout << "Estudiante: " << nombre
```

```
<< ", Nota: " << nota << endl;
}
int main() {
Persona *Pepito = new Empleado("Jose", 1000); (1)
Persona *Pablito = new Estudiante("Pablo", 7.56);
char n[30];
Pepito->Mostrar();
Pablito->Mostrar();
cin.get();
return 0;
}
```

La salida será así:

```
Empleado: Jose, Salario: 1000
Estudiante: Pablo, Nota: 7.56
```

En este ejemplo combinamos el uso de funciones virtuales puras con polimorfismo. Fíjate que, aunque hayamos declarado los objetos "Pepito" y "Pablito" de tipo puntero a "Persona" (1), en realidad no creamos objetos de ese tipo, sino de los tipos "Empleado" y "Estudiante".

5.5. Ejemplos

Como ejemplo se puede suponer que la cuenta_joven y la cuenta_empresarial ya utilizadas y descritas anteriormente, con una forma distinta de abonar mensualmente el interés al saldo.

- En la cuenta_joven, no se abonará el interés pactado si el saldo es inferior a un límite.
- En la cuenta_empresarial se tienen tres cantidades límite, a las cuales se aplican factores de corrección en el cálculo del interés. El cálculo de la cantidad abonada debe realizarse de la siguiente forma:

- Si el saldo es menor que 50000, se aplica el interés establecido previamente.
- Si el saldo está entre 50000 y 500.000, se aplica 1.1 veces el interés establecido previamente.

- Si el saldo es mayor a 500.000, se aplica 1.5 veces el interés establecido previamente.

El código correspondiente quedaría de la siguiente forma:

```
class C_Cuenta {
// Variables miembro
private:
double Saldo; // Saldo Actual de la cuenta
double Interes; // Interés calculado hasta el momento, anual,
// en tanto por ciento %
public:
//Constructor
C_Cuenta(double unSaldo=0.0, double unInteres=4.0)
{
SetSaldo(unSaldo);
SetInteres(unInteres);
}
// Acciones Básicas
inline double GetSaldo()
{ return Saldo; }
inline double GetInteres()
{ return Interes; }
inline void SetSaldo(double unSaldo)
{ Saldo = unSaldo; }
inline void SetInteres(double unInteres)
{ Interes = unInteres; }
void Ingreso(double unaCantidad)
{ SetSaldo( GetSaldo() + unaCantidad ); }
virtual void AbonaInteresMensual()
{
SetSaldo( GetSaldo() * ( 1.0 + GetInteres() / 12.0 / 100.0) );
}
// etc...
};

class C_CuentaJoven : public C_Cuenta
{
public:
C_CuentaJoven(double unSaldo=0.0, double unInteres=2.0,
double unLimite = 50.0E3) :C_Cuenta(unSaldo, unInteres)
{
Limite = unLimite;
}
```

```
virtual void AbonaInteresMensual()
{
if (GetSaldo() > Limite)
SetSaldo( GetSaldo() * (1.0 + GetInteres() / 12.0 / 100.0) );
else
SetSaldo( GetSaldo() );
}
private:
double Limite;
};

class C_CuentaEmpresarial : public C_Cuenta {
public:
C_CuentaEmpresarial(double unSaldo=0.0, double unInteres=4.0)
: C_Cuenta(unSaldo, unInteres)
{
CantMin[0] = 50.0e3;
CantMin[1] = 500.0e3;
}
virtual void AbonaInteresMensual()
{
SetSaldo( GetSaldo() * (1.0 + GetInteres() * CalculaFactor() /
12.0 / 100.0 ));
}
double CalculaFactor()
{
if (GetSaldo() < CantMin[0])
return 1.0;
else if (GetSaldo() < CantMin[1])
return 1.1;
else return 1.5;
}
private:
double CantMin[2];
};
```

6. Plantillas

La generalidad es una propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o alguno de sus miembros. Esta propiedad no es imprescindible en un lenguaje de programación orientado a objetos y ni siquiera es una de sus características. Esta característica del C++ apareció mucho más tarde que el resto del lenguaje, al final de la década de los ochenta. Esta generalidad se alcanza con las plantillas (templates).

La utilidad principal de este tipo de clases o funciones es la de agrupar variables cuyo tipo no esté predeterminado. Así el funcionamiento de una pila, una cola, una lista, un conjunto, un diccionario o un array es el mismo independientemente del tipo de datos que almacene (int, long, double, char, u objetos de una clase definida por el usuario). En definitiva estas clases se definen independientemente del tipo de variables que vayan a contener y es el usuario de la clase el que debe indicar ese tipo en el momento de crear un objeto de esa clase.

6.1. Introducción

Hemos indicado que en la programación clásica existía una clara diferenciación entre los datos y su manipulación, es decir, entre los datos y el conjunto de algoritmos para manejarlos. Los datos eran tipos muy simples, y generalmente los algoritmos estaban agrupados en funciones orientadas de forma muy específica a los datos que debían manejar.

Posteriormente la POO introdujo nuevas facilidades: La posibilidad de extender el concepto de dato, permitiendo que existiesen tipos más Complejos a los que se podía asociar la operatoria necesaria. Esta nueva habilidad fue perfilada con un par de mejoras adicionales: La posibilidad de ocultación de determinados detalles internos, irrelevantes para el usuario, y la capacidad de herencia simple o múltiple .

Observe que las mejoras introducidas por la POO se pueden sintetizar en tres palabras: Composición, ocultación y herencia. De otro lado, la posibilidad de incluir juntos los datos y su operatoria no era exactamente novedosa. Esta circunstancia ya existía de forma subyacente en todos los lenguajes. Recuerde que el concepto de entero (int en C) ya incluye implícitamente todo un álgebra y reglas de uso para dicho tipo. Observe también que la POO mantiene un paradigma de programación orientado al dato (o estructuras de datos). De hecho los "Objetos" se definen como instancias concretas de las clases, y estas representan nuevos tipos-de-datos, de modo que POO es sinónimo de Programación Orientada a Tipos-de-datos

Desde luego la POO supuso un formidable avance del arsenal de herramientas de programación. Incluso en algunos casos, un auténtico balón de oxígeno en el desarrollo y mantenimiento de aplicaciones muy grandes, en las que se estaba en el límite de lo factible con las técnicas programación tradicional. Sin embargo, algunos teóricos seguían centraron su atención en los algoritmos. Algo que estaba ahí también desde el principio. Se dieron cuenta que frecuentemente las manipulaciones contienen un denominador común que se repite bajo apariencias diversas. Por ejemplo: La idea de ordenación "Sort" se repite infinidad de veces en la programación, aunque los objetos a ordenar y los criterios de ordenación varíen de un caso a otro. Alrededor de esta idea surgió un nuevo paradigma denominado *programación genérica o funcional*.

La programación genérica está mucho más centrada en los algoritmos que en los datos y su postulado fundamental puede sintetizarse en una palabra: generalización. Significa que, en la medida de lo posible, los algoritmos deben ser parametrizados al máximo y expresados de la forma más independiente posible de detalles concretos, permitiendo así que puedan servir para la mayor variedad posible de tipos y estructuras de datos.

Los expertos consideran que la parametrización de algoritmos supone una aportación a las técnicas de programación, al menos tan importante, como fue en su

momento la introducción del concepto de herencia, y que permite resolver algunos problemas que aquella no puede resolver.

Observe que la POO y la programación genérica representan enfoques en cierta forma ortogonales entre si:

- La **programación orientada al dato** razona del siguiente modo: Representemos un tipo de dato genérico (por ejemplo int) que permita representar objetos con ciertas características comunes (peras y manzanas por ejemplo). Definamos también que operaciones pueden aplicarse a este tipo (por ejemplo aritméticas) y sus reglas de uso, independientemente que el tipo represente peras o manzanas en cada caso.
- Por su parte la **programación funcional** razona lo siguiente: Construyamos un algoritmo genérico (por ejemplo sort), que permita representar algoritmos con ciertas características comunes (ordenación de cadenas alfanuméricas y vectores por ejemplo). Definamos también a que tipos pueden aplicarse a este algoritmo y sus reglas de uso, independientemente que el algoritmo represente la ordenación de cadenas alfanuméricas o vectores.

Con el fin de adoptar los paradigmas de programación entonces en vanguardia, desde sus inicios C++ había adoptado conceptos de lenguajes anteriores. Uno de ellos, la programación estructurada , ya había sido recogida en el diseño de su antecesor directo C. También adoptó los conceptos de la POO entonces emergente . Posteriormente ha incluido otros conceptos con que dar soporte a los nuevos enfoques de la programación funcional; básicamente plantillas y contenedores. Las plantillas, que se introdujeron con la versión del Estándar de Julio de 1998 son un concepto tomado del Ada. Los contenedores no están definidos en el propio lenguaje, sino en la Librería Estándar.

6.2. Concepto de plantilla

Las plantillas ("Templates"), también denominadas tipos parametrizados, son un mecanismo C++ que permite que un tipo pueda ser utilizado como parámetro en la definición de una clase o una función.

Ya se trate de clases o funciones, la posibilidad de utilizar un tipo como parámetro en la definición, posibilita la existencia de entes de nivel de abstracción superior al de función o clase concreta. Podríamos decir que se trata de funciones o clases genéricas; parametrizadas (de ahí su nombre). Las "instancias" concretas de estas clases y funciones conforman familias de funciones o clases relacionadas por un cierto "denominador común", de forma que

proporcionan un medio simple de representar gran cantidad de conceptos generales y un medio sencillo para combinarlos.

Para ilustrarlo intentaremos una analogía: si la clase Helado-de-Fresa representara los helados de fresa, de los que las "instancias" concretas serían distintos tamaños y formatos de helados de este sabor, una plantilla Helado-de-<tipo> sería capaz de generar las clases Helado-de-fresa; Helado-de-vainilla; Helado-de-chocolate, etc. con solo cambiar adecuadamente el argumento <tipo>. En realidad respondería al concepto genérico de "Helado-de". Las instancias concretas de la plantilla forman una familia de productos relacionados (helados de diversos sabores). Forzando al máximo la analogía diríamos "especialidades".

Observe que el mecanismo de plantillas C++ es en realidad un generador de código parametrizado. La conjunción de ambas capacidades: Generar tipos (datos) y código (algoritmos) les confiere una extraordinaria potencia. Si bien el propio inventor del lenguaje reconoce que a costa de "cierta complejidad", debida principalmente a la variedad de contextos en los que las plantillas pueden ser definidas y utilizadas

La idea central a resaltar aquí es que una plantilla genera la definición de una clase o de una función mediante uno o varios parámetros. A esta instancia concreta de la clase o función se la denomina una especialización o especialidad de la plantilla. Un aspecto crucial del sistema es que los parámetros de la plantilla pueden ser a su vez plantillas.

Para manejar estos conceptos utilizaremos la siguiente terminología:

- Plantilla de clase (template class) o su equivalente: clase genérica.
- Plantilla de función (template function) o su equivalente: función genérica.

Definida una plantilla, al proceso por el cual se obtienen clases especializadas (es decir para alguno de los tipos de datos específicos) se denomina instanciación o de la plantilla o especialización de una clase o método genérico. A las funciones y clases generadas para determinados tipos de datos se las denomina entonces como clases o funciones concretas o especializadas.

Como se ha indicado, las plantillas representan una de las últimas implementaciones del lenguaje y constituyen una de las soluciones adoptadas por C++ para dar soporte a la programación genérica. Aunque inicialmente fueron introducidas para dar soporte a las técnicas que se necesitaban para la Librería Estándar (para lo que se mostraron muy adecuadas), son también oportunas para muchas situaciones de programación. Precisamente la exigencia fundamental de diseño de la citada librería era lograr algoritmos con el mayor grado de abstracción posible, de forma que pudieran adaptarse al mayor número de situaciones concretas.

El tiempo parece demostrar que sus autores realizaron un magnífico trabajo que va más allá de la potencia, capacidad y versatilidad de la Librería Estándar C++ y de que otros lenguajes hayan seguido la senda marcada por C++ en este sentido. Tal es el caso de Java, con su JGL ("Java Generic Library"). Lo que comenzó como una herramienta para la generación parametrizada de nuevos tipos de datos (clases), se ha convertido por propio derecho en un nuevo paradigma, la metaprogramación (programas que escriben programas).

De lo dicho hasta ahora puede deducirse, que las funciones y clases obtenidas a partir de versiones genéricas (plantillas), pueden obtenerse también mediante codificación manual (en realidad no se diferencian en nada de estas últimas). Aunque en lo tocante a eficacia y tamaño del código, las primeras puedan competir en igualdad de condiciones con las obtenidas manualmente. Esto se consigue porque el uso de plantillas no implica ningún mecanismo de tiempo de ejecución. Las plantillas dependen exclusivamente de las propiedades de los tipos que utiliza como parámetros y todo se resuelve en tiempo de compilación. Podríamos pensar que su resolución es similar a las macros de C en el que se hacía una sustitución textual de la expresión indicada en el *define* por la expresión puesta a continuación, pero en función de unos parámetros. De igual forma, cuando se especializa una plantilla, se escribe el código con los tipos sustitudos por lo indicado en la plantilla, y después se procede a compilar tanto el código escrito manualmente como el escrito automáticamente por este mecanismo.

Las plantillas representan un método muy eficaz de generar código (definiciones de funciones y clases) a partir de definiciones relativamente pequeñas. Además su utilización permite técnicas de programación avanzadas, en las que implementaciones muy sofisticadas se muestran mediante interfaces que ocultan al usuario la complejidad, mostrándola solo en la medida en que necesite hacer uso de ella. De hecho, cada una de las potentes abstracciones que se utilizan en la Librería Estándar está representada como una plantilla. A excepción de algunas pocas funciones, prácticamente el 100% de la Librería Estándar está relacionada con las plantillas, de ahí que hasta ahora no se halla hecho mucha referencia a esta librería perteneciente al estándar de C++.

La palabra clave *template*

C++ utiliza una palabra clave específica **template** para declarar y definir funciones y clases genéricas. En estos casos actúa como un especificador de tipo y va unido al par de ángulos < > que delimitan los argumentos de la plantilla:

```
template <T> void fun(T& ref); // función genérica
template <T> class C {/*...*/}; // clase genérica
```

En algunas otras (raras) ocasiones la palabra template se utiliza como calificador para indicar que determinada entidad es una plantilla (y en consecuencia puede aceptar argumentos) cuando el compilador no puede deducirlo por sí mismo.

Bien, pues una vez expuestas las ideas principales referentes al concepto de plantilla, vamos a ver en primer lugar como se realzan funciones genéricas, para después explicar el concepto y el modo de funcionamiento de las clases genéricas.

6.3. Plantillas de funciones

Para ilustrar gráficamente su utilidad utilizaremos un ejemplo clásico: queremos construir una función max(a, b) que pueda utilizarse para obtener el mayor de dos valores, suponiendo que estos sean de cualquier tipo capaz de ser ordenado, es decir, cualquier tipo en el que se pueda establecer un criterio de ordenación (establecemos a > b si a está después que b en el orden).

El problema que presenta C++ para esta propuesta es que al ser un lenguaje fuertemente tipado, la declaración c max(a, b) requiere especificar el tipo de argumentos y valor devuelto. En realidad se requiere algo así:

tipoT max(tipoT a, tipoT b);

y la sintaxis del lenguaje no permite que tipoT sea algo variable. Una posible solución es sobrecargar la función max(), definiendo tantas versiones como tipos distintos debamos utilizar.

```
double max(double a,double b){return a>b?a:b;}
int max(int a,int b){return a>b?a:b;}
…
```

Otra alternativa sería utilizar una macro:

```
#define max(a, b) ((a > b) ? a : b)
```

pero esto presenta sus inconvenientes. Empezando porque su utilización permitiría comparar un entero con una estructura o una matriz, algo que está claramente fuera del propósito de la función que pretendemos.

La solución al problema enunciado es utilizar una función genérica (plantilla). La sintaxis de su definición es la siguiente:

```
template <class T> T max(T a, T b)
{
  return (a > b) ? a : b;
}
```

<class T> es la lista de parámetros. Representa el/los parametros de la plantilla. Los parámetros de una plantilla funciona en cierta forma como los argumentos de una macro (el trabajo de esta macro es generar código de funciones). Es importante significar que utilizamos dos conceptos distintos (aunque relacionados): los parámetros de la plantilla (contenidos en la lista template <....>) y los argumentos de la función (argumentos con que se invoca la función en cada caso concreto).

Lo mismo que en las funciones explícitas, las genéricas pueden ser declaradas antes de su utilización:

```
template <class T> T max(T, T);
```

y definidas después:

```
template <class T> T max(T a, T b)
{
  return (a > b) ? a : b;
}
```

La idea fundamental es que el compilador deduce los tipos concretos de los parámetros de la plantilla de la inspección de los argumentos actuales utilizados en la invocación . Por ejemplo, la plantilla anterior puede ser utilizada mediante las siguientes sentencias:

```
int i, j;
UnaClase a, b;
...
int k = max(i, j);                  // (1)
UnaClase c = max(a, b);             // (2)
```

En (1) los **argumentos de la función** son dos objetos tipo int; mientras en (2) son dos objetos tipo UnaClase. El compilador es capaz de construir dos funciones aplicando los parámetros adecuados a la plantilla. En el primer caso, el parámetro es un int; en el segundo un tipo UnaClase. Como veremos más adelante, es de la máxima importancia que el compilador sea capaz de deducir los parámetros de la plantilla a partir de los argumentos actuales (los utilizados en cada invocación de la función), así como las medidas sintácticas adoptadas cuando esto no es posible por producirse ambigüedad.

Una función genérica puede tener más argumentos que la plantilla. Por ejemplo:

```
template <class T> void func(T, inf, char, long, ...);
```

También puede tener menos:

```
template <class T> void func();
```

La forma de operar en este caso para que el compilador deduzca el parámetro correcto T a utilizar en la plantilla, se muestra más adelante cuando se hable de la especificación explícita de los argumentos de una plantilla.

Llegados a este punto es conveniente hacer algunas observaciones importantes:

Las funciones genéricas son entes de nivel de abstracción superior a las funciones concretas (en este contexto preferimos llamarlas funciones explícitas), pero **las funciones genéricas solo tienen existencia en el código fuente** y en la mente del programador. Hemos dicho que el mecanismo de plantillas C++ se resuelve en tiempo de compilación, de modo que en el ejecutable, y durante la ejecución, no existe nada parecido a una función genérica**, solo existen especializaciones** (instancias de la función genérica).

Esta característica de las funciones genéricas es de la mayor importancia. Supone que pueden escribirse algoritmos muy genéricos en los que **los detalles dependen del tipo de objeto con el que se utiliza** (el algoritmo). En nuestro ejemplo, el criterio que define que objeto *a* o *b* es mayor, no está contenido en la función max(), sino en la propia clase a que pertenecen ambos objetas en donde ha debido sobrecargarse el operador > para ese tipo concreto. Esta es justamente la premisa fundamental de la programación genérica.

La instanciación de la plantilla se produce cuando el compilador encuentra que es necesaria una versión concreta (especialidad) de la función genérica. Esto sucede cuando existe una invocación como en el ejemplo, la línea (2) , o se toma la dirección de la función (por ejemplo para iniciar un puntero-a-función). Entonces se genera el código apropiado en concordancia con el tipo de los argumentos actuales.

Ocurre que **si esta instancia aparece más de una** vez en un módulo, o es generada en más de un módulo, el enlazador las refunde automáticamente en una sola definición, de forma que solo exista una copia de cada instancia. Dicho en otras palabras: en la aplicación resultante solo existirá una definición de cada función. Por contra, si no existe ninguna invocación no se genera ningún código.

Aunque la utilización de funciones genéricas conduce a un código elegante y reducido, que no se corresponde con el resultado final en el ejecutable. Si la aplicación utiliza muchas plantillas con muchos tipos diferentes, el resultado es la **generación de gran cantidad de código** con el consiguiente consumo de espacio. Esta crecimiento del código es conocida como "Code bloat", y puede llegar a ser un problema. En especial cuando se utilizan las plantillas de la Librería Estándar, aunque existen ciertas técnicas para evitarlo. Como regla

general, las aplicaciones que hace uso extensivo de plantillas resultan grandes consumidoras de memoria (es el costo de la comodidad).

Puesto que cada instancia de una función genérica es una verdadera función, cada especialización dispone de su propia copia de las variables estáticas locales que hubiese. Se les pueden declarar punteros y en general gozan de todas las propiedades de las funciones normales, incluyendo la capacidad de sobrecarga

Veamos un caso concreto con una función genérica que utiliza tanto una clase Vector como un entero:

```
#include <iostream.h>
class Vector
{
public:
 float x, y;
 bool operator>(Vector v)
   {
   return ((x*x + y*y) > (v.x*v.x + v.y*v.y))? true: false;
   }
};
template<class T> T max(T a, T b){ return (a > b) ? a : b;}

void main()
{
  Vector v1 = {2, 3}, v2 = {1, 5};
  int x = 2, y = 3;
  cout <<"Mayor: "<<max(x, y)<< endl;
  cout <<"Mayor:"<<max(v1, v2).x<<", "<<max(v1,v2).y << endl;
}
```

El resultado de ejecutar el programa es el siguiente:

Mayor: 3

Mayor: 1, 5

Otro ejemplo clásico en la definición de una plantilla de función es el caso de la función permutar:

```
#include <iostream.h>
template <class S> void permutar(S&, S&);
void main(void)
 {
```

```
 int i=2, j=3;
 cout << "i=" << i << " " << "j=" << j << endl;
 permutar(i, j);
 cout << "i=" << i << " " << "j=" << j << endl;
 double x=2.5, y=3.5;
 cout << "x=" << x << " " << "y=" << y << endl;
 permutar(x, y);
 cout << "x=" << x << " " << "y=" << y << endl;
 }
template <class S> void permutar(S& a, S& b)
 {
 S temp;
 temp = a;
 a = b;
 b = temp;
 }
```

Otro ejemplo, es el de la realización de un algoritmo de ordenación por selección directa. En este caso se hará uso de la función genérica permutar. Observese que a causa de ello, el numero de funciones generadas automáticamente pasa a ser cuatro. Si se implementase el código de la función genérica permutar directamente en el interior de la función genérica ordenar, el número de funciones generadas automáticamente hubiera sido la mitad:

```
#include <iostream.h>
template <class S> void permutar(S& a, S& b);
template <class T> void ordenar (T *vector, int num);

void main(void)
 {

 float mivector[10]={2,4,6,8,1,3,5,7,9,0};
 char cadena[10]="efghBACDI";

 ordenar(mivector,8);
 ordenar(cadena,10);

 for(i=0;i<10;i++)cout<<mivector[i];
 cout<<endl<<cadena;
 }

template <class S> void permutar(S &a, S &b)
 {
 S temp;
 temp = a;
```

```
 a = b;
 b = temp;
 }
template <class T> void ordenar (T *vector, int num)
{
int i,j;
for(i=0;i<num-1;i++)
     for(j=i+1;j<num;j++)
        if(vector[i]<vector[j])permutar(vector[i],vector[j]);
}
```

Metodos genéricos

Las funciones genéricas pueden ser miembros (métodos) de clases:

```
class A
  {
  template<class T> void func(T& t) // def de método genérico
      {
       ...
      }
  ...
  }
```

La definición de métodos genéricos puede hacerse también fuera del cuerpo de la clase como con cualquier otro método, mediante el uso del operador de resolución de ámbito:

```
class A
   {
    template<class T> void func(T& t);//decl método genérico
    ...
   }
...
template <class T> void A::func(T& t)  //definición del método
  {
   // código del método...
  }
```

Aunque aún no se han explicado las clases genéricas, es conveniente indicar ya que los miembros genéricos pueden ser a su vez miembros de clases genéricas, en cuyo caso pueden hacer uso de los parámetros de la plantilla de clase genérica. Un ejemplo de un método genérico es el siguiente:

```
#include <iostream.h>
```

```
class A
 {
  public:
  int x;
  template<class T> void fun(T t1, T t2);
  A (int a = 0) { x = a; }
 };

template<class T> void A::fun(T t1, T t2)
      {   cout << "Valor-1: " << t1
       << ", Valor-2: " << t2
       << ", Valor-x: " << x << endl;
      }

void main(void)
 {
 A a(7), b(14);
 a.fun(2, 3);
 b.fun('x', 'y');
 }
```

Salida:

Valor-1: 2, Valor-2: 3, Valor-x: 7

Valor-1: x, Valor-2: y, Valor-x: 14

Parámetros de la plantilla

La definición de la función genérica puede incluir más de un argumento. Es decir, el especificador *template* <...> puede contener una lista con varios tipos. Estos parámetros pueden ser tipos Complejos o fundamentales, por ejemplo un int; incluso especializaciones de clases genéricas y constantes de tiempo de compilación.

En general los parámetros de una plantilla pueden ser:

- Identificadores de tipo, por ejemplo *T*. Estos parámetros van precedidos por la palabra reservada *class* o *typename*:

  template <class A, class B> void func(A, B);

  template <typename A, typename B>void func(A,B);

- Plantillas de clases (adelantando de nuevo):

  template<class A,template<class t> class X> void func(A, X<T>);

- Parámetros de algún otro tipo: primitivo, derivado, definido por el usuario, o de plantilla. Un parámetro de plantilla que no sea un tipo, es una constante dentro de la plantilla, y por lo tanto, no se puede modificar.

  template <class A, int x> void func(A, int);

  template <class T, int p> T fx(T x);

Un ejemplo que ilustra este último caso, es el siguiente en el que se utilizan dos plantillas muy parecidas desde el punto de vista de uso, pero diferentes en cuanto al código compilado es el siguiente:

```
#include <iostream.h>

template <class T, int ind> void imprime(T* v)
{
      for(int i=0;i<ind;i++)cout<<v[i];
}
template <class T> void imprime2(T* v,int ind)
{
      for(int i=0;i<ind;i++)cout<<v[i];
}
void main()
{
      float vector[3]={1,2,3};
      char cadena[]="Hola Mundo";
      imprime<float,3>(vector); //(1)
      imprime<char,8>(cadena);
      imprime2(vector,2);
      imprime2(cadena,5);
}
```

Los argumentos a la plantilla de las funciones genéricas no pueden tener valores por defecto. Lo mismo que en las funciones explícitas, en las funciones genéricas debe existir concordancia entre el número y tipo de los argumentos formales y actuales.

```
template <class A, int x> void func(A, int);
...
```

```
func(T);        // Error!! falta 1 argumento
```

Evidentemente los parámetros de la función, no afectados por la plantilla, si que pueden tener valores por defecto. Por ejemplo, la función imprime2 podría ser reescrita de la forma siguiente:

```
template <class T> void imprime2(T* v,int ind=2)
{
      for(int i=0;i<ind;i++)cout<<v[i];
}
```

y a la hora de ejecutarse podría entonces escribirse:

```
imprime2(vector);
```

Como se observa en el ejemplo, todos los argumentos formales de la plantilla (contenidos en la lista template <....>) deberían estar representados en los argumentos formales de la función. De no ser así, no habría posibilidad de deducir los valores de los tipos de la lista <....> salvo que se indique explícitamente cuando se produzca una invocación específica de la función por medio del uso de <…> –como en la línea (1) del ejemplo de imprime-.

El siguiente código por tanto daría error en la compilación:

```
template <class A, class B> void func(A a) {  // Plantilla
  ...
};
...
func(a);     // Error de compilación !!
```

En este caso el compilador no tiene posibilidad de deducir el tipo del argumento B.

En ocasiones el diseño de la función no permite determinar el tipo de parámetro de la plantilla a partir de los argumentos actuales de la invocación (o sencillamente se quiere obligar al compilador a aplicar los argumentos actuales a una especialización instanciada con unos parámetros concretos). Por ejemplo:

```
template <class T> T* construir ()
{
T *aux=new T;
```

```
if(aux==NULL)
      {
      cout<<"Error de construcción";
      abort();
      }
return aux;
 }
```

La plantilla anterior crea un objeto de cualquier tipo y devuelve un puntero al objeto creado, o aborta el programa en caso de no haber podido crearlo. Se observa que el compilador no puede deducir el tipo de parámetro a utilizar con la plantilla a partir del argumento actual, puesto que en este caso no hay argumento actual. La siguiente línea sería errónea:

```
int* iptr = construir();
```

El compilador arrojaría el mensaje: ERROR, parámetro T desconocido.

Para su utilización debe especificarse explícitamente el tipo de parámetro a utilizar mediante un argumento de plantilla:

```
int* iptr = construir<int>();
```

La gramática del lenguaje exige que cuando el argumento de la plantilla solo es utilizado por la función en el tipo de retorno, debe indicarse explícitamente que tipo de instanciación se pretende mediante la inclusión de parámetros de plantilla <...> entre el nombre de la función y la lista de argumentos: función <...> (...). Esto es coherente con el modo de comportarse gramaticalmente el compilador con las funciones sobrecargadas. Recuérdese que para estas funciones, la diferenciación entre las mismas no puede realizarse por medio del tipo de la función o de retorno.

Esta forma de instanciar una plantilla se denomina instanciación implícita específica de la plantilla o especificación explícita de los parámetros de la plantilla, y en estos casos la lista <...> que sigue al nombre de la función genérica puede incluir los parámetros de la plantilla que sean necesarios. Esta lista no tiene porqué incluir a todos los parámetros actualmente utilizados por la plantilla, ya que el compilador la completa con los tipos que puedan deducirse de la lista de argumentos de la función. Sin embargo, los parámetros faltantes deben ser los últimos de la lista <...> (análogo a lo exigido a los argumentos por defecto en las funciones explícitas). Por ejemplo:

```
template <class A, class B, class C> void func(B b, C c, int i);
```

```
....
func(b, c, i);            // Error!!
func <A, B, C>(b, c, i);  // Ok. B y C redundantes
func<A>(b, c, i);         // Ok.
func<A>(b, c);            // Error!! falta argumento i
```

Un aspecto crucial de las funciones genéricas es que el compilador debe poder deducir sin ambigüedad los argumentos de la plantilla a partir de los argumentos utilizados para la invocación de la función.

Recordemos que en los casos de sobrecarga, la invocación de funciones C++ utiliza un sistema estándar para encontrar la definición que mejor se adapta a los argumentos actuales. También se realizan transformaciones automáticas cuando los argumentos pasados a la función no concuerdan exactamente con los esperados (argumentos formales) Estos mecanismos utilizan unas reglas denominadas **congruencia estándar de argumentos**

En caso de las plantillas de función o funciones genéricas, el compilador deduce los parámetros de la plantilla mediante el análisis de los argumentos actuales de la invocación, pero para esto solo realiza conversiones triviales (menos significativas que las realizadas con las funciones explícitas). Los siguientes ejemplos pueden ayudar a ilustrar estas conversiones tanto para una función genérica como para una normal o explícita:

```
template<class T> bool igual(T a, T b)
{
  return (a == b) ? true : false;
}
bool desigual(double a, double b)
{
  return (a == b) ? false : true;
}
...
int i;
char c;
double d;
...
igual(i, i);       // Ok. invoca igual(int ,int)
igual(c, c);       // Ok. invoca igual(char,char)
igual(i, c);       // Error!! igual(int,char) indefinida
igual(c, i);       // Error!! igual(char,int) indefinida
desigual(i, i)     // Ok. conversión de argumentos efectuada
desigual(c, c)     // Ok. conversión de argumentos efectuada
desigual(i, c)     // Ok. conversión de argumentos efectuada
desigual(d, d)     // Ok. concordancia de argumentos
```

Sobrecarga de funciones genéricas

Hemos señalado que la instanciación de la plantilla se realiza cuando el compilador encuentra una invocación de la función genérica o se obtiene su dirección y que solo puede existir una versión de cada especialización de la función genérica. Estas premisas conducen a que sea posible evitar la generación automática para uno o varios tipos concretos, mediante dos procedimientos:

- Proporcionando una versión codificada de forma "manual" de la función (versión explícita). Es decir escribiendo el código de la función con los parámetros especificados normalmente.
- Forzar una instanciación específica de la plantilla (instanciación explícita), de forma que se genera el código de una especialidad concreta, con independencia de que posteriormente se requiera o no, la utilización del código generado. La instanciación puede realizarse de dos formas:
  - Forzar la instanciación de la plantilla "tal cual" para un tipo particular. Esta instancia explícita tendría el comportamiento genérico definido en la plantilla, por lo que la denominamos instanciación explícita general.
  - Forzar una instanciación para un tipo particular en las mismas condiciones que el anterior (con independencia de la posible utilización del código generado en el programa), pero definiendo un nuevo comportamiento, distinto del general definido en la plantilla. En otras palabras: instanciar una versión sobrecargada de la función para un tipo específico. La denominamos instanciación explícita particular .

Como veremos a continuación, estas posibilidades tienen distinta utilidad y ligeras diferencias de detalle. Aunque son técnicas diferentes, el resultado final es análogo: la existencia de una (o varias) especializaciones concretas de la función, lo que nos obligará a contemplar una generalización de la sobrecarga de funciones que incluya funciones explícitas y genéricas.

Con independencia de la explicación más detallada que sigue, para situarnos en el tema adelantamos un esbozo de lo que significa la literatura anterior referida a un caso muy sencillo:

```
// función genérica (declaración)
```

```
template<class T> T max(T, T);
...
// función genérica (definición)
template<class T> T max(T a, T b) { return (a > b) ? a : b; }
...
// versión explícita
char max(char a, char b) { return (a >= b) ? a : b; }
...
// instanciación explícita general
template long max<long>(long a, long b);
...
// instanciación explícita particular
template<> double max<double>(double a, double b)
      { return (a >= b) ? a : b; };
...
// instanciación implícita específica
int x = max<int>(x, 'c');
```

Para entrar con mayor profundidad en el tema , considere un caso en el que utilizamos una función genérica *igual*( ) para comprobar si dos objetos son iguales, y a la cual iremos aplicando cada uno de los casos anteriores:

```
#include <iostream.h>

class Vector
{
public:
  float x, y;
  Vector(float a,float b):x(a),y(b){}
  bool operator==(const Vector& v)
    {
    return ( x == v.x && y == v.y)? true : false;
    }
};
template<class T> bool igual(T a, T b) función genérica
{
 return (a == b) ? true : false;
}

void main()
{
```

```
  Vector v1(2, 3), v2 (1, 5);
  int x = 2, y = 3;
  double d1 = 2.0, d2 = 2.2;

  if ( igual(v1, v2) ) cout << "vectores iguales" << endl;
  else cout << "vectores distintos" << endl;
  if ( igual(d1, d2) ) cout << "doubles iguales" << endl;
  else cout << "doubles distintos" << endl;
  if ( igual(x, y) ) cout << "enteros iguales" << endl;
  else cout << "enteros distintos" << endl;
}
```

La salida de este programa, como era de esperar es la siguiente:

vectores distintos

doubles distintos

enteros distintos

Hasta aquí nada nuevo: el compilador ha generado y utilizado correctamente las especializaciones de igual( ) para las invocaciones con tipos int, double y Vector.

Versión explícita

Consideremos ahora que es necesario rebajar la exigencia para que dos variables sean consideradas iguales en el caso de que sean doubles. Para ello introducimos una instancia de igual codificada manualmente en el que reflejamos la nueva condición de igualdad:

```
#include <iostream.h>

class Vector
{
public:
  float x, y;
  Vector(float a,float b):x(a),y(b){}
  bool operator==(const Vector& v)
    {
    return ( x == v.x && y == v.y)? true : false;
    }
};
template<class T> bool igual(T a, T b)
```

```
{
  return (a == b) ? true : false;
};
bool igual(double a, double b)     // versión explícita
{
  return (labs(a-b) < 1.0) ? true : false;
};
void main()
{
  Vector v1(2, 3), v2 (1, 5);
  int x = 2, y = 3;
  double d1 = 2.0, d2 = 2.2;

  if ( igual(v1, v2) ) cout << "vectores iguales" << endl;
  else cout << "vectores distintos" << endl;
  if ( igual(d1, d2) ) cout << "doubles iguales" << endl;
  else cout << "doubles distintos" << endl;
  if ( igual(x, y) ) cout << "enteros iguales" << endl;
  else cout << "enteros distintos" << endl;
}
```

Salida:

vectores distintos

doubles iguales

enteros distintos

La versión explícita para tipos double utiliza la función de librería *labs* para conseguir que dos doubles sean considerados iguales si la diferencia es solo en los decimales. La inclusión de esta definición supone que el compilador no necesita generar una versión de igual() cuando los parámetros son tipo *double*. En este caso, el compilador utiliza la versión suministrada "manualmente" por el programador.

Además de permitir introducir modificaciones puntuales en el comportamiento general, las versiones explícitas pueden utilizarse también para eliminar algunas de las limitaciones de las funciones genéricas.

Por ejemplo, si sustituimos la sentencia:

```
if ( igual(d1, d2) ) cout << "doubles iguales" << endl;
```

por:

```
if ( igual(d1, y) ) cout << "doubles iguales" << endl;
```

Se obtiene un error de compilación: *Could not find a match for 'igual<T>(double,int)'*. La razón es que, como hemos visto, el compilador no realiza ningún tipo de conversión sobre el tipo de los argumentos utilizados en las funciones genéricas, y en este caso no existe una definición de *igual()* que acepte un *double* y un *int*. En cambio, la misma sustitución de cuando existe una versión explícita para *igual(double double),* no produce ningún error. La razón es que para las funciones normales el compilador si es capaz de realizar automáticamente determinadas transformaciones de los argumentos actuales para adecuarlos a los esperados por la función.

Instanciación explícita general

El estándar ha previsto un procedimiento para obligar al compilador a generar el código de una especialización concreta a partir de la plantilla-función. Esta instanciación forzada se denomina instanciación explícita, y utiliza el especificador template aislado (sin estar seguido de <...>). Recuerde que la definición de la plantilla igual es:

```
template<class T> bool igual(T a, T b) {...}
```

La sintaxis para generar una versión de igual específica para doubles sería la siguiente:

```
template bool igual<double>(double a, double b);
```

Observe la sintaxis utilizada: la lista de parámetros <...> se ha cambiado de posición respecto a la declaración de la plantilla.

La inclusión de una instanciación explícita como la anterior (la llamaremos general porque sigue el comportamiento general definido por la plantilla), origina la aparición del código correspondiente a la especialización solicitada aunque en el programa no exista una necesidad real (invocación) de dicho código. Esta instancia explícita general desempeña un papel análogo al de una versión que se hubiese codificado manualmente (versión explícita).

Instanciación explícita particular

Observe que la versión instanciada en la expresión anterior es concordante con la plantilla, por lo que no sirve para realizar modificaciones específicas como las realizadas en el ejemplo anterior en el caso de los float. Sin embargo, es posible también especificar una definición particular para la especialidad que se instancia añadiendo el cuerpo adecuado. A esta versión la denominamos instancia explícita particular.

La sintaxis seria la siguiente:

```
template<> bool igual<double>(double a, double b)
   {
```

```
      return (labs(a-b) < 1.0) ? true : false;
     }
```

Los ángulos <> después de template indican al compilador que sigue una especialización particular de una plantilla definida previamente. Como puede figurarse el lector, el resultado es similar al que se obtendría una versión explícita .

Es un error intentar la existencia de más de una definición para la misma función, ya sea esta una instanciación implícita, explícita o una versión codificada manualmente.

Por ejemplo:

```
bool igual(double& a, double& b)
  {
  return (labs(a-b) < 1.0) ? true : false;
  };

template bool igual<double>(double& a, double& b);
```

En las condiciones anteriores el compilador puede generar un error, una advertencia, o sencillamente ignorar el segundo requerimiento, ya que previamente existe una versión explícita de la función con idéntica firma. En cualquier caso es una regla que el compilador dará preferencia a una función normal (versión explícita) sobre cualquier forma de instanciación, explícita o implícita, al utilizar una función.

6.4. Clases genéricas

Hemos indicado al comienzo del capítulo que las clases-plantilla, clases genéricas o generadores de clases, son un artificio C++ que permite definir una clase mediante uno o varios parámetros. Este mecanismo es capaz de generar la definición de clases (instancias o especializaciones de la plantilla) distintas, pero compartiendo un diseño común. Podemos imaginar que una clase genérica es un constructor de clases, que como tal acepta determinados argumentos (no confundir con el constructor de-una-clase, que genera objetos).

Para ilustrarlo veremos la clase *mVector*. Los objetos *mVector* son matrices cuyos elementos son objetos de la clase *Vector*; que a su vez representan vectores de un espacio de dos dimensiones.

El diseño básico de la clase es como sigue:

```
class mVector
 {
   int dimension;
  public:
   Vector* mVptr;
   mVector(int n = 1)
{
       dimension = n;
       mVptr = new Vector[dimension];
       }
  ~mVector() { delete [] mVptr; }
  Vector& operator[](int i) { return mVptr[i]; }
  void mostrar(int);
};
void mVector::mostrar (int i)
  {
  if((i >= 0) && (i <= dimension)) mVptr[i].mostrar();
  }
```

El sistema de plantillas permite definir una clase genérica que instancie versiones de *mVector* para matrices de cualquier tipo especificado por un parámetro. La ventaja de este diseño parametrizado, es que cualquiera que sea el tipo de objetos utilizados por las especializaciones de la plantilla, las operaciones básicas son siempre las mismas (inserción, borrado, selección de un elemento, etc).

Definición de una clase genérica

La definición de una clase genérica tiene el siguiente aspecto:

```
template<lista-de-parametros> class nombreClase
{
 ...
};
```

Una clase genérica puede tener una declaración adelantada (forward) para ser declarada después:

```
template<lista-de-parametros> class nombreClase;
...
template<lista-de-parametros> class nombreClase
{
   ...
};
```

pero recuerde que debe ser definida antes de su utilización y evidentemente, solo puede definirse una vez .

Observe que la definición de una plantilla comienza siempre con template<...>, y que los parámetros de la lista <...> no son valores, sino tipos de datos .

La definición de la clase genérica correspondiente al caso anterior es la siguiente:

```
template<class T> class mVector
 {
   int dimension;
  public:
   T* mVptr;
   mVector(int n = 1)
       {
       dimension = n;
       mVptr = new T[dimension];
       }
  ~mVector() { delete [] mVptr;  }
   T& operator[](int i) { return mVptr[i]; }
   void mostrar (int);
};

template<class T> void mVector<T>::mostrar (int i)
 {
  if((i >= 0) && (i <= dimension)) mVptr[i].mostrar();
 }
```

Observe que aparte del cambio de la declaración, se han sustituido las ocurrencias de Vector (un tipo concreto) por el parámetro T. Observe también la definición de mostrar() que se realiza off-line con la sintaxis de una función genérica.

Recordemos que en estas expresiones, el especificador class puede ser sustituido por typename , de forma que la primera línea puede ser sustituida por:

```
template<typename T> class mVector
```

```
   {
     ...
   };
```

Veamos el ejemplo completo:

```
#include <iostream.h>
class Vector
    {
    public:
     int x, y;
     Vector& operator= (const Vector& v)
      {
      x = v.x; y = v.y;
      return *this;
      }
     void mostrar(){cout << "X = " << x << "; Y = " << y << endl;}
    };

template<class T> class mVector
   {
    int dimension;
  public:
    T* mVptr;
    mVector& operator=(const mVector& mv)
         {
         delete [] mVptr;
         dimension = mv.dimension;
         mVptr = new T[dimension];
         for(int i = 0; i<dimension; i++)
                mVptr[i]= mv.mVptr[i];
         return *this;
         }
    mVector(int n = 1)
         {
         dimension = n;
         mVptr = new T[dimension];
         }
    ~mVector() {delete [] mVptr;}
    mVector(const mVector& mv)
         {
         dimension = mv.dimension;
         mVptr = new T[dimension];
         for(int i = 0; i<dimension; i++)
                mVptr[i]= mv.mVptr[i];

         }
```

```
     T& operator[](int i) { return mVptr[i]; }
     void mostrar (int);
     void mostrar ();
   };

template <class T> void mVector<T>::mostrar (int i)
   {
  if((i >= 0) && (i < dimension)) mVptr[i].mostrar();
  }
template <class T> void mVector<T>::mostrar ()
   {
  cout << "Matriz de: " << dimension << " elementos." << endl;
  for (int i = 0; i<dimension; i++)
         {
         cout << i << "- ";
         mVptr[i].mostrar();
         }
  }

void main()
   {
   mVector<Vector> mV1(5);
   mV1[0].x = 0; mV1[0].y = 1;
   mV1[1].x = 2; mV1[1].y = 3;
   mV1[2].x = 4; mV1[2].y = 5;
   mV1[3].x = 6; mV1[3].y = 7;
   mV1[4].x = 8; mV1[4].y = 9;
   mV1.mostrar();
   mVector<Vector> mV2 = mV1;
   mV2.mostrar();
   mV1[0].x = 9;   mV1[0].y = 0;
   mV2.mostrar(0);
   mV1.mostrar(0);
   mVector<Vector> mV3(0);
   mV3.mostrar();
   mV3 = mV1;
   mV3.mostrar();
 }
```

```
Salida:
Matriz de: 5 elementos.
    0 - X = 0; Y = 1
    1 - X = 2; Y = 3
    2- X = 4; Y = 5
    3- X = 6; Y = 7
```

```
4- X = 8; Y = 9
Matriz de: 5 elementos.
0- X = 0; Y = 1
1- X = 2; Y = 3
2- X = 4; Y = 5
3- X = 6; Y = 7
4- X = 8; Y = 9
X = 0; Y = 1
X = 9; Y = 0
Matriz de: 0 elementos.
Matriz de: 5 elementos.
0- X = 9; Y = 0
1- X = 2; Y = 3
2- X = 4; Y = 5
3- X = 6; Y = 7
4- X = 8; Y = 9
```

Otro ejemplo que describe una clase genérica para la realización de una pila de datos sin que se utilicen listas enlazadas y reserva dinámica de memoria adicional durant funcionmiento de un objeto de una clase instanciada es el siguiente:

```
// fichero Pila.h
template <class T>
// declaración de la clase
class Pila
{
public:
Pila(int nelem=10); // constructor
void Poner(T);
void Imprimir();
private:
int nelementos;
T* cadena;
int limite;
};
// definición del constructor
template <class T> Pila<T>::Pila(int nelem)
{
nelementos = nelem;
cadena = new T(nelementos);
limite = 0;
};
// definición de las funciones miembro
template <class T> void Pila<T>::Poner(T elem)
```

```
{
if (limite < nelementos)
cadena[limite++] = elem;
};
template <class T> void Pila<T>::Imprimir()
{
int i;
for (i=0; i<limite; i++)
cout << cadena[i] << endl;
};
```

El programa principal puede ser el que sigue:

```
#include <iostream.h>
#include "Pila.h"
void main()
{
Pila <int> p1(6);
p1.Poner(2);
p1.Poner(4);
…
```

Miembros de clases genéricas

Los miembros de las clases genéricas se definen y declaran exactamente igual que los de clases concretas. Pero debemos señalar que las funciones-miembro son a su vez plantillas parametrizadas (funciones genéricas) con los mismos parámetros que la clase genérica a que pertenecen.

Consecuencia de lo anterior es que si las funciones-miembro se definen fuera de la plantilla, sus prototipos deberían presentar el siguiente aspecto:

```
template<class T> class mVector
{
  int dimension;
  public:
  T* mVptr;
  template<class T> mVector<T>& operator=(const mVector<T>&);
  template<class T> mVector<T>(int);
  template<class T> ~mVector<T>();
  template<class T> mVector<T>(const mVector<T>& mv);
  T& operator[](int i) { return mVptr[i]; }
  template <class T> void mostrar (int);
  template <class T> void mostrar ();
```

```
};
```

Sin embargo, no es exactamente así por diversas razones: La primera es que, por ejemplo, se estaría definiendo la plantilla mostrar sin utilizar el parámetro T en su lista de argumentos (lo que en un principio no está permitido). Otra es que no está permitido declarar los destructores como funciones genéricas. Además los especificadores <T> referidos a mVector dentro de la propia definición son redundantes.

Estas consideraciones hacen que los prototipos puedan ser dejados como sigue (los datos faltantes pueden ser deducidos por el contexto):

```
template<class T> class mVector
{
  int dimension;
  public:
  T* mVptr;
  mVector& operator= (const mVector&);
  mVector(int);                  // constructor por defecto
  ~mVector();                    // destructor
  mVector(const mVector& mv);    // constructor-copia
  T& operator[](int i) { return mVptr[i]; }
  void mostrar (int);            // función auxiliar
  void mostrar ();               // función auxiliar
};
```

Sin embargo, las definiciones de métodos realizadas off-line (fuera del cuerpo de una plantilla) deben ser declaradas explícitamente como funciones genéricas.

Por ejemplo:

```
template <class T> void mVector <T>::mostrar (int i)
{
  ...
}
```

Miembros estáticos

Las clases genéricas pueden tener miembros estáticos (propiedades y métodos). Posteriormente cada especialización dispondrá de su propio conjunto de estos miembros. Estos miembros estáticos deben ser definidos fuera del cuerpo de la plantilla, exactamente igual que si fuesen miembros estáticos de clases concretas:

```
template<class T> class mVector
  {
```

```
    ...
    static T* mVptr;
    static void mostrarNumero (int);
    ...
    };

  template<class T> T* mVector<T>::nVptr;
  template<class T> void mVector<T>::mostrarNumero(int x){ ... };
```

Métodos genéricos

Hemos señalado que, por su propia naturaleza, los métodos de clases genéricas son a su vez (implícitamente) funciones genéricas con los mismos parámetros que la clase, pero pueden ser además funciones genéricas explícitas (que dependan de parámetros distintos de la plantilla a que pertenecen):

```
  template<class X> class A
  {

    // método genérico de clase genérica
    template<class T> void func(T& t);
    ...
  }
```

Según es usual, la definición del miembro genérico puede efectuarse de dos formas, inline u off-line:

```
  #include <iostream.h>

template<class X> class A
  {
  public:
   int x;
   X* xptr;
   template<class T> void fun(T t1, T t2)
         {
         // método genérico
         cout << "Valor-1: " << t1
         << ", Valor-2: " << t2
         << ", Miembro-x: " << x
         << ", Objeto: " << *xptr << endl;
         }
    A (X* b, int i = 0)
```

```
          {
          x = i;
         xptr = b;
         }
};

int main(void)
   {
  char c = 'c'; char* cptr = &c;
  int x = 13;   int* iptr = &x;
  A<int> a(iptr, 2);
  A<char> b(cptr, 3);
  a.fun(2, 3);
  a.fun('x', 'y');
  b.fun(2, 3);
  b.fun('x', 'y');
  return 0;
  }
```

```
Salida:

Valor-1: 2, Valor-2: 3, Miembro-x: 2, Objeto: 13
Valor-1: x, Valor-2: y, Miembro-x: 2, Objeto: 13
Valor-1: 2, Valor-2: 3, Miembro-x: 3, Objeto: c
Valor-1: x, Valor-2: y, Miembro-x: 3, Objeto: c
```

Instanciación de clases genéricas

Las clases genéricas son entes de nivel superior a las clases concretas. Representan para las clases normales (en este contexto preferimos llamarlas clases explícitas) lo mismo que las funciones genéricas a las funciones concretas. Como aquellas, solo tienen existencia en el código. Como el mecanismo de plantillas C++ se resuelve en tiempo de compilación, ni en el fichero ejecutable ni durante la ejecución existe nada parecido a una clase genérica, solo existen especializaciones. En realidad la clase genérica que se ve en el código, actúa como una especie de "macro" que una vez ejecutado su trabajo en la fase de compilación, desaparece de escena.

Observe que, como ha indicado algún autor, el mecanismo de plantillas representa una especie de polimorfismo en tiempo de compilación, similar al que proporciona la herencia de métodos virtuales en tiempo de ejecución.

A diferencia de lo que ocurre con las funciones genéricas, en la instanciación de clases genéricas el compilador no realiza ninguna suposición sobre la naturaleza de los argumentos a utilizar, de modo que se exige que sean declarados siempre de forma explícita.

Por ejemplo:

```
mVector<char> mv1;
mVector mv2 = mv1;         // Error !!
mVector<char> mv2 = mv1;   // Ok.
```

Sin embargo, como veremos a continuación, las clases genéricas pueden tener argumentos por defecto, por lo que en estos casos la declaración puede no ser explícita sino implícita (referida a los valores por defecto de los argumentos). La consecuencia es que en estos casos el compilador tampoco realiza ninguna suposición sobre los argumentos a utilizar.

Las clases genéricas pueden ser utilizadas en los mecanismos de herencia. En ese caso, la clase derivada estará parametrizada por los mimmos argumentos que la plantilla de la clse base.

Por ejemplo:

```
template <class T> class Base { ... };
template <class T> class Deriv : public Base<T> {...};
```

Los *typedef* son muy adecuados para acortar la notación de objetos de clases genéricas cuando se trata de declaraciones muy largas o no interesan los detalles.

Por ejemplo :

```
typedef basic_string <char> string;
…
string st1;
```

Las clases genéricas pueden tener argumentos por defecto, en cuyo caso, el tipo T puede omitirse, pero no los ángulos <>. Por ejemplo:

```
template<class T = int> class mVector {/* ... */};
...
mVector<char> mv1;    // Ok. argumento char explícito
mVector<> mv2;        // Ok. argumento int implícito
```

Cada instancia de una clase genérica es realmente una clase, y sigue las reglas generales de las clases. Dispondrá por tanto de su propia versión de todos los miembros estáticos si los hubiere. Estas clases son denominadas implícitas, para distinguirlas de las definidas "manualmente", que se denominan explícitas.

La primera vez que el compilador encuentra una sentencia del tipo mVector<Vector> crea la función-clase para dicho tipo; es el punto de instanciación. Con

objeto de que solo exista una definición de la clase, si existen más ocurrencias de este mismo tipo, las funciones-clase redundantes son eliminadas por el enlazador. Por la razón inversa, si el compilador no encuentra ninguna razón para instanciar una clase (generar la función-clase), esta generación no se producirá y no existirá en el código ninguna instancia de la plantilla.

Al igual que ocurre con las funciones genéricas, en las clases genéricas también puede evitarse la generación de versiones implícitas para tipos concretos proporcionando una especialización explícita.

Por ejemplo:

```
class mVector<T> { ... };       // definición genérica
...
class mVector<char> { ... };    // definición específica
```

más tarde, las declaraciones del tipo

```
mVector<char> mv1, mv2;
```

generará objetos utilizando la definición específica. En este caso mv1 y mv2 serán matrices alfanuméricas (cadenas de caracteres).

Observe que la definición explícita comporta dos requisitos:

•Aunque es una versión específica (para un tipo concreto), se utiliza la sintaxis de plantilla: class mVector<...> {...};

•Se ha sustituido el tipo genérico <T> por un tipo concreto <char>.

Resulta evidente que una definición específica, como la incluída, solo tiene sentido si se necesitan algunas modificaciones en el diseño cuando la clase se refiera a tipos char.

Argumentos de la plantilla

La declaración de clases genéricas puede incluir una lista con varios parámetros. Estos pueden ser casi de cualquier tipo: Complejos, fundamentales, por ejemplo un int, o incluso otra clase genérica (plantilla). Además en todos los casos pueden presentar valores por defecto.:

```
template<class T, int dimension = 128> class mVector { ... };
```

Al igual que en las funciones genéricas, en la instanciación de clases genéricas, los valores de los parámetros que no sean tipos Complejos deben ser constantes o expresiones constantes.

```
const int K = 128;
```

```
int i = 256;
mVector<int, 2*K> mV1;              // OK
mVector<Vector, i> mV2;             // Error: i no es constante
```

Este tipo de parámetros constantes son adecuados para establecer tamaños y límites. Sin embargo, por su propia naturaleza de constantes, cualquier intento posterior de alterar su valor genera un error.

Los argumentos también pueden ser otras plantillas, pero solo de clases genéricas. Introducir como argumento una plantilla de funcione genérica no está permitido:

```
template <class T, template<class X> class C> class MatrizC
{    ...
};
template <class T, template<class X> void Func(X a)> class MatrizF
 { // Error!!
   ...
};
```

Hay que tener en cuenta que no existe algo parecido a un mecanismo de "sobrecarga" de las clases genéricas paralelo al de las funciones genéricas. Por ejemplo las siguientes declaraciones producen un error de compilación.

```
template<class T> class mVector { ... };
template<class T, int dimension> class mVector { ... };
```

El compilador daría el siguiente error: *Number of template parameters does not match in redeclaration of 'Matriz<T>'.*

Punteros y referencias a clases implícitas

Como hemos señalado, las clases implícitas gozan de todas las prerrogativas de las explícitas, incluyendo por supuesto la capacidad de definir punteros y referencias. En este sentido no se diferencian en nada de aquellas; la única precaución es tener presente la cuestión de los tipos a la hora de efectuar asignaciones, y no perder de vista que la plantilla es una abstracción que representa múltiples clases (tipos), cada una representada por argumentos concretos.

Consideremos la clase genérica Matriz cuya declaración es:

```
template <class T, int dim =1> class Matriz { /* ... */};
```

La definición de punteros y referencias sería como sigue:

```
Matriz<int,5> m1;                  // Ok. Tres objetos
Matriz<char,5> m2;                 // de tipos
Matriz<char> m3;                   // distintos.
...
Matriz<int,5>* ptrMi5 = &m2        // Error!! tipos distintos
Matriz<char,5>* ptrMch5 = &m2;     // Ok.
Matriz<char,1>* ptrMch1 = &m2;     // Error!! tipos distintos
Matriz<char,1>* ptrMch1 = &m3;     // Ok.
ptrMch5->show();                   // Ok. invocación de método
Matriz<char> m4 = *ptrMch1;        // Ok asignación mediante puntero

void (Matriz<char,5>::* fptr1)(); // Ok. declara puntero a método
fptr1 = &Matriz<char,5>::show;     // Ok. asignación
(m3.*fptr1)();                     // Error!! tipo de m3 incorrecto
(m2.*fptr1)();                     // Ok. invocación de método

Matriz<char,5>& refMch5 = m2;      // Ok. referencia
Matriz<char>& refMch1 = m4;        // Ok.
refMch5.show();                    // Ok. invocación de método
```

Merecen especial atención las sentencias donde se utilizan punteros a miembros de clases implícitas. El tipo de clase está definido en los parámetros. Observe que *fptr1* es puntero a método de clase Matriz<char, 5>, y no puede referenciar un método de m3, que es un objeto de tipo Matriz<char,1>.

```
p1.Imprimir();
Pila <char> p2(6);
p2.Poner('a');
p2.Poner('b');
p2.Imprimir();
}
```

6.5. Clases genéricas en la Librería Estándar

Al principio del capitulo se ha señalado que las plantillas fueron introducidas en C++ para dar soporte a determinadas técnicas utilizadas en la Librería Estándar; de hecho, la STL está constituida casi exclusivamente por plantillas. Hablar y utilizar en profundidad las plantillas de la librería estándar puede permitirnos el reducir drásticamente el esfuerzo

necesario para la programación, son embargo comprenderlas e introducirse en el modo en que se han programado requiere ya de una soltura en el manejo del lenguaje que queda fuera de los objetivos de este curso. Uno de los aspectos más utilizados de las STL son los contenedores, entre los que se podría incluir aquellos dedicados a manejar cadenas de caracteres.

Una idea básica dentro de las STL es el *contenedor*, que es precisamente lo que parece: un sitio en el que almacenar cosas. Ya se ha visto que una de las operaciones más habituales es esta, así como una de las justificaciones para el uso de plantillas de clases (recuérdese el ejemplo de Lista<Esferas> o de la clase genérica mVector. Habitualmente necesitamos este tipo de sitios de almacenamiento cuando de antemano desconocemos cuantos objetos o datos vamos a tener. Los contenedores de la STL hacen esto; son capaces de mantener colecciones de objetos y además se pueden redimensionar. El modo como almacenan estas colecciones de objetos, y como consecuencia las operaciones que se pueden realizar sobre las colecciones, hacen que aparezcan distintos tipos de contenedores. Los contenedores más secillos son:

- **vector**. Un vector es la versión STL de una matriz dinámica de una dimensión. Las instancias de esta clase genérica conforman una secuencia (una clase de contenedor). La clase dispone de acceso aleatorio a sus elementos y de un mecanismo muy eficiente de inserción y borrado de elementos al final. Aunque también pueden insertarse y borrarse elementos en medio, por el modo en que se organizan los elementos en memoria, esta operación es muy poco eficiente. Está definida en el fichero <vector> y responde a la siguiente declaración:

```
template <class T, class Allocator = allocator<T> > class vector;
```

  Una especialización concreta de esta plantilla, cuando el tipo T es un booleano *vector<bool>*, es considerada por la STL como un caso especial, con un código optimizado que incluye métodos no disponibles para el resto de las instancias (los métodos *flip* y *swap*) pensado para trabajar con representaciones bit a bit.

- **deque.** Es muy parecido a vector, una secuencia lineal de elementos, pero admite con eficiencia la inserción de objetos tanto al principio como al final, mientras que sigue comportándose mal para inserciones intermedias. Admite el acceso aleatorio casi con la misma eficiencia que **vector**, pero es especialmente más rápido cuando requiere solicitar más espacio para almacenar más datos (redimensionamiento).

- **list**. Las instancias de esta plantilla conforman también una secuencia que dispone de mecanismos muy eficientes para insertar y eliminar elementos en cualquier punto. Está internamente implementada como una lista doblemente enlazada. Como consecuencia

de su estructura, no está diseñada para el acceso aleatorio siendo este bastante más ineficiente que en las dos clases anteriores. Está definida en el fichero <list> y responde a la siguiente declaración:

```
template <class T, class Allocator = allocator<T> > class list;
```

Los tres tipos de conenedores admiten el método **push_back()** que agregará un nuevo elemento al final de la secuencia (además deque y list, tienen el método **push_front()** el cual permite la inserción al inicio).

El acceso a los elementos del vector o de deque, puede ser aleatorio, y tienen por ello implementada la sobrecarga del operador [ ], pero este no está definido para la clase list. Para contar con un medio de recorrer los contenedores de forma uniforme, y también por razones de consistencia de los datos, se hace uso de un tipo de datos definido para todos los contenedores y denominado ***iterator***.

Un iterador es una clase que abstrae el proceso de moverse a lo largo de una secuencia, sin necesidad de conocer como está estructurada esta secuancia internamente (no nos importa si los objetos en memoria estan ordenados secuencialmente o su orden está determinado por algún tipo de estructura adicional como es el caso de las listas). Contar con este tipo de "recorredor" de secuencias es importanto dado que nos da un modo uniforme de tratarlas, y por tanto nos permitirá utilizar funciones genéricas indistintamente al tipo de contenedor usado.

Todos los contenedores contienen esta clase de objetos definida y que sobrecarga el operador ++ y el operador ->, además de otras operaciones. Además, los contenedores contienen el método begin() y end() que nos retornarán un iterador ubicado en la posición inicial o final de la secuencia respectivamente.

Dado que solo se trata de dar una visión rápida de los contenedores en las STL, el siguiente ejemplo es bastante ilustrativo de su uso:

```
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};


class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw\n"; }
```

```
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};


typedef std::vector<Shape*> Container; //por comodidad definimos
typedef Container::iterator Iter;      //dos tipos con typedef

int main() {
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin();
      i != shapes.end(); i++)
    (*i)->draw(); //se puede usar i->draw() dado que el iterador tiene
                  //sobrecargado el operador ->
  // ... al final si queremos limpiar la memoria:
  for(Iter j = shapes.begin();
      j != shapes.end(); j++)
    delete *j;
}
```

Es interesante ver que es posible cambiar sin problemas el tipo básico del contenedor a una lista o un deque, y el código no se ve alterado más que en la definición del contenedor.

Algunas otras clases habitualmente utilizadas, serían las siguientes:

- **basic_string**: una plantilla para utilizar entidades como secuencias de caracteres. Está definida en el fichero de cabecera <string>, y responde a la siguiente declaración:

```
template <class charT, class traits = char_traits<charT>,
  class Allocator = allocator<charT> >   class basic_string;
```

Como puede verse, acepta tres argumentos, de los que dos tienen valores por defecto. Existen dos especializaciones que tienen nombres específicos. En concreto si *charT* es *char* la especialización se denomina *string*, y *wstring* si *charT* es *wchar_t*. Esto se hace mediante sendos *typedef*:

```
typedef basic_string <char> string;
```

```
typedef basic_string <wchar_t> wstring;
```

de forma que las dos sentencias siguientes son equivalentes:

```
basic_string<char> s1;
string s1;
```

Las cadenas en C++ son en el fondo contenedores de caracteres y como tales pueden ser tratados. Sin embargo, dado su uso tan común, se ha dotado a basic_string de un conjunto de funciones y operadores que simplifican (¡por fin!) el uso de este tipo de datos. Por ejemplo, la concatenación, creación, reserva, comparación, paso a mayusculas, etc... son operaciones incluidas directamente:

```
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1("This ");
  string s2="That ";
  string s3("The other ");
  // operator+ concatena strings
  s1 = s1 + s2;
  cout << s1 << endl;
  // otra forma de concatenar strings
  s1 += s3;
  cout << s1 << endl;
  // o acceder a la vez que se concatena a un caracter
  s1 += s3 + s3[4] + "oh lala";
  cout << s1 << endl;
  if(s1 != s2) {
    cout << "los strings son distitos:" << " ";
    if(s1 > s2)
      cout << "s1 es > s2" << endl;
    else
      cout << "s2 es > s1" << endl;
  }
}
```

- **map**. Esta plantilla está definida en el fichero <map>, y tiene la siguiente declaración:

```
template <class Key, class T, class Compare = less<Key>
  class Allocator = allocator<pair<const Key, T>> >  class map;
```

Las instancias de esta plantilla conforman un contenedor asociativo (una clase de contenedor) que permite almacenar y acceder objetos de tipo T indexados mediante una única clave Key, disponiendo de mecanismos muy eficientes de inserción y borrado.

- **auto_ptr**. Esta plantilla genera punteros "inteligentes" que destruyen automáticamente el objeto señalado cuando ellos mismos salen de ámbito. Su definición está en el fichero <memory>, y su declaración es:

```
template <class X> class auto_ptr;
```

En realidad esta plantilla se ha incluido en la Librería Estándar para ayudar a resolver el problema de los objetos persistentes, creados con el operador new, cuyo referente (el puntero que los señala) es un objeto automático que puede ser destruido inadvertidamente al salir de ámbito. Lo que puede ocurrir en el los procesos de lanzamiento y captura de excepciones.

6.6. Ejemplo

El siguiente estracto de código muestra una clase genérica que sirve para alamecenar punteros de objetos o elementos creados externamente a ella. Internamente se comporta como un vector dinámico que admite por tanto un número indefinido de elementos. Incluye además una serie de operaciones básicas para facilitar el manejo y mantenimiento del conjunto de direcciones almacenadas:

- **Fichero contenedor.h**

```
template <class T> class contenedor
{
private:
   T **array;
   int nElem;
   int elemMax;
public:
       contenedor(void);
   contenedor(contenedor &td); /*copia*/
   ~contenedor(void);
   inline  T* operator [] (int a);
   void operator += (T *ele);
   void destruirObjetos(void);
   void eliminar (int a);
```

```
    T* quitar(int a);
    T* quitar(T* ele);
    int buscar(T* ele);
    void swap(int a, int b);
    void insertar(T *ele, int a);
 inline int numElem (void);
};
```

♦ **Fichero contenedor.cpp**

```
#include "contenedor.h"
template <class T> contenedor<T>::contenedor (void)
    {
    array = new T* [5];
    nElem = 0;
    elemMax = 5;
    }
template <class T> void contenedor<T>::destruirObjetos (void)
    {
    for (int i=0; i<numElem; i++)
    delete  array[i];
    numElem=0;
    }
template <class T> contenedor<T>::~contenedor (void)
    {
    delete [] array;
    }

template <class T> contenedor<T>::contenedor (contenedor & td)
    {
    int i;
    nElem = td.nElem;
    elemMax = td.elemMax;
    array =  new T* [elemMax];
    for(i=0;i<nElem;i++) array[i]=td[i];
    }
template <class T> T* contenedor<T>::operator [] (int a)
    {
    if ((a>=0)&&(nElem>a))return array[a];
    return NULL;
    }
template <class T> void contenedor<T>::operator += (T* ele)
    {

       if (nElem == elemMax)
             {
             elemMax+=5;
```

```
            T** array2 = new T* [elemMax];
            memcpy (array2,array,nElem*sizeof(T*));
            delete [] array;
            array=array2;
            }
   array[nElem] = ele;
   nElem ++;
   }
template <class T>  T* contenedor<T>::quitar(int a)
   {
   T* aux;
   if((a>=0)&&(a<numElem))
      {
      aux=array[a];
      array[a]=NULL;
      for(int i=a+1;i<numElem;i++)
                  array[i-1]=array[i];
      return aux;
      }
   return NULL;
   }
template <class T>  T* contenedor<T>::quitar(T* ele)
   {
   return (quitar(buscar(ele)));
   }
template <class T>  int contenedor<T>::buscar(T* ele)
   {
   for(int i=0;i<nElem;i++)if(array[i]==ele)return i;
   return -1;
   }
template <class T>  void contenedor<T>::swap(int a,int b)
   {
   T* aux;
   if((a>=0)&&(a<nElem)&&(b>=0)&&(b<nElem))aux=a;a=b;b=aux;
   }
template <class T>  void contenedor<T>::insertar(T *ele, int a)
   {
   if(a<0)return;
   (*this)+=ele;
   swap(nElem-1,a);
   }
template <class T> int contenedor<T>::numElem (void)
   {
   return nElem;
   }
```

7. Manejo de excepciones y tratamiento de errores

Uno de los aspectos que con más dificultad se abordan en los distintos lenguajes de programación es el correcto tratamiento de los errores. Es decir, el funcionamiento normal de un algoritmo o programa contiene la dificultad inherente del problema que se quiere resolver, y como consecuencia generará un flujograma más o menos complicado. Sin embargo, todo ello se incrementa en complejidad de forma notable en el momento en el que se pretende verificar y comprobar los posibles errores o situaciones "no nominales" durante el proceso de ejecución.

Este problema no es solo específico de la ingeniería del software, sino que a menudo es el mayor quebradero de cabeza en la mayoría de los proyectos de diseño y en particular aquellos en los que la seguridad es importante. De hecho, se puede decir sin problema, que es más Complejo el sistema de redundancia y seguridad que se incluye en el software de control de un avión que el propio sistema de control.

Al conjunto de situaciones "imprevistas" se la denomina "**excepción**", y al mecanismo dispuesto para reaccionar ante esas situaciones, se lo denomina como

consecuencia "**manejador de excepciones**[6]". Existen muchos tipos de excepciones, pero para ubicar el tema, durante la ejecución de un programa puede ocurrir que un fichero no se pueda grabar (disco lleno), o que se agote la memoria, o que el usuario haya intentado realizar una operación no válida (por ejemplo desde el programa abrir un fichero sobre el que carece de permiso), etc.

7.1. Tratamiento de excepciones en C++

El tratamiento de excepciones C++ sigue un proceso estructurado, por lo que a la hora de diseñarlo se diferencian tres conceptos básicos:

1. Hay una parte de código en el que se considera que se pueden producir situaciones anómalas, y que consecuencia se pueden generar una serie de excepciones. Por eso, se considera que es posible que la acción que esa parte de código quiere realizar no se consiga, y por tanto es una zona de "intento de ejecución" (**try**).
2. Cuando se detecta una circunstancia extraña o anómala (una excepción), se informa al proceso, función, o programa que ha ocurrido algo imprevisto, y se interrumpe el curso normal de ejecución. Para realizarlo se "lanza" una excepción (**throw**). Esa excepción es un objeto que contiene información sobre el error para que el que escuche sea capaz de reaccionar ante esta situación.
3. Para que el programa pueda decidir que hacer en estos casos, tras intentar ejecutar algo susceptible de fallar, indicará una zona del código que se ejecutará para reaccionar ante esas sistuaciones. Para ello "captura" (**catch**) las excepciones lanzadas en la zona de "intento" y se decide que hacer al respecto con la información que estas contienen.

En el fondo el mecanismo es un mecanismo de salto controlado. Al código encargado de reaccionar ante una determinada excepción se lo denomina "manejador" de la excepción.

Durante la ejecución de un programa pueden existir dos tipos de circunstancias excepcionales: **síncronas** y **asíncronas**. Las primeras son las que ocurren dentro del programa. Por ejemplo, que se agote la memoria o cualquier otro tipo de error. Son a estas a las que nos hemos estado refiriendo. Las excepciones asíncronas son las que tienen su origen fuera del programa, a nivel del Sistema Operativo. Por ejemplo que se pulsen las teclas Ctrl+C.

[6] Exception Handler

Generalmente las implementaciones C++ solo consideran las excepciones síncronas, de forma que no se pueden capturar con ellas excepciones tales como la pulsación de una tecla. Dicho con otras palabras: solo pueden manejar las excepciones lanzadas con la sentencia **throw**. Siguen un modelo denominado de excepciones síncronas con terminación, lo que significa que una vez que se ha lanzado una excepción, el control no puede volver al punto que la lanzó. El "handler" **no** puede devolver el control al punto de origen del error mediante una sentencia **return**. En este contexto, un **return** en el bloque **catch** supone salir de la función que contiene dicho bloque.

El sistema estándar C++ de manejo de excepciones no está diseñado para manejar directamente excepciones asíncronas, como las interrupciones de teclado, aunque pueden implementarse medidas para su control. Además las implementaciones más usuales disponen de recursos para menejar las excepciones del Sistema Operativo

Antes de ver un ejemplo completo se verá en primer lugar como se ha previsto en C++ la transcripción de cada uno de estos compoenentes del mecanismo de tratamiento de excepciones.

El bloque "try"

En síntesis podemos decir que el programa se prepara para cierta acción, decimos que "lo intenta". Para ello se especifica un bloque de código cuya ejecución se va a intentar ("try-block") utilizando la palabra clave try:

```
try
{

  // conjunto de instrucciones en las que
  // se puede producir una excepción y que quedan
  // afectadas por la misma
  ...
}
```

Un ejemplo sencillo sería la operación de apertura y grabación de información en un fichero. Es posible que no se pueda abrir el fichero por no tener permisos de escritura en el luegar indicado por el usuario, y por tanto cualquier instrucción de escritura tras fracasar en la apertura debe ser evitada.

La idea del bloque try en este caso podría ejemplarizarse de la siguiente forma:

INTENTA {
    1.- Abrir el fichero
    2.- Guardar la información
    3.- Cerrar el fichero
}
SI HAY UN ERROR CAPTURALO Y EJECUTA
{
  4.- Informa al usuario
}

Luego habitualmente el bloque de intento no sólo incluye la zona susceptible de generar errores, sino que también incluye la parte de código que debe evitar ser ejecutada en caso de que estos errores se produzcan. Así en el ejemplo, si la apertura de fichero da un error, habrá que evitar ejecutar las operaciones indicadas por 2 y por 3. Por ello, si 1 lanza una excepción de "ERROR DE APERTURA", se interrumpe la ejecuón secuencial del programa y se "SALTA" hasta el manejador (4) que informará al usuario de este error.
Se informa por tanto al programa que si se produce un error durante el "intento", entonces se debe transferir el control de ejecución al punto donde exista un manejador de excepciones ("handler") que coincida con el tipo de excepción lanzado. Si no se produce ninguna excepción, el programa sigue su curso normal.

De lo dicho se deduce inmediatamente que se pueden lanzar excepciones de varios tipos y que pueden existir también receptores (manejadores) de varios tipos; incluso manejadores "universales", capaces de hacerse cargo de cualquier tipo de excepción.

Así pues, try es una sentencia que en cierta forma es capaz de especificar el flujo de ejecución del programa. Un bloque-intento debe ser seguido inmediatamente por el bloque manejador de la excepción, el cual se especifica por medio de la palabra clave catch.

El bloque "catch"

Mediante la palabra clave catch especificamos el código encargado de reaccionar ante una determinada excepción o conjunto de excepciones. Como ya se ha comentado, se dice que un manejador "handler" captura una excepción.
En C++ es obligatorio que tras un bloque try se incluya al menos un bloque catch, por lo que la sintaxis anterior quedaría completa de la siguiente forma:

```
try {          // bloque de código que se intenta
  ...
}
catch (...) { // bloque manejador de posibles excepciones
  ...
```

```
}
...            // continua la ejecución normal
```

El "handler" es el sitio donde continua el programa en caso de que ocurra la circunstancia excepcional (generalmente un error) y donde se decide qué hacer. A este respecto, las estrategias pueden ser muy variadas (no es lo mismo el programa de control de un reactor nuclear que un humilde programa de contabilidad). En último extremo, en caso de errores absolutamente irrecuperables, la opción adoptada suele consistir en mostrar un mensaje explicando el error.

Lanzamiento de una excepción " throw "

Las excepciónes no sólo las lanzan las funciones incluidas dentro de las librerías externas, sino que nosotros podemos querer informar que algo no ha ido bien dentro de la lógica de nuestro programa.

Al igual que ocurre con el código de las librerías estandar, esto se realiza por medio de la instrucción throw.

El lenguaje C++ especifica que todas las excepciones deben ser lanzadas desde el interior de un bloque-intento y permite que sean de cualquier tipo. Como se ha apuntado antes, generalmente son un objeto (instancia de una clase) que contiene información. Este objeto es creado y lanzado en el punto de la sentencia throw y capturado donde está la sentencia catch. El tipo de información contenido en el objeto es justamente el que nos gustaría tener para saber que tipo de error se ha producido. En este sentido puede pensarse en las excepciones como en una especie de correos que transportan información desde el punto del error hasta el sitio donde esta información puede ser analizada. Todo esto lo veremos con más detalle a continuación.

Como se ha comentado C++ permite lanzar excepciones de cualquier tipo. Sin embargo lo normal es que sean instancias de una clase tipo X, que contiene la información necesaria para conocer la naturaleza de la circunstancia excepcional (probablemente un error). El tipo X debe corresponder con el tipo de argumento usado en el bloque catch, de tal forma que por medio del tipo de las excepciones será posible ejecutar manejadores distintos aún siendo estas lanzadas desde elmismo bloque try.

> Nota: Se recomienda que las clases diseñadas para instanciar este tipo de objetos (denominadas Excepciones) sean específicas y diseñadas para este fin exclusivo, sin que tengan otro uso que la identificación y manejo de las excepciones.

La expresión throw (X arg) inicializará un objeto temporal arg de tipo X, aunque puede que se generen otras copias por necesidad del compilador. En consecuencia puede ser útil definir un constructor-copia cuando el objeto a lanzar pertenece a una clase que contiene subobjetos, como en el caso de cualquier otra clase cuyos objetos quieran ser pasados por valor (recuérdese el apartado dedicado al contructor de copia y el operador de asignación en el capítulo 3).

En el siguiente ejemplo se pasa el objeto de tipo Out al manejador catch:

```
#include <stdio.h>
bool pass;
class Out{};    // L.3: Para instanciar el objeto a  lanzar

void festival(bool);    // L.4: prototipo
int main()
{
try {                // L.7: bloque intento
      pass = true;
      festival(true);
    }
catch(Out o) {  // L.11: la excepción es capturada aquí
     pass = false; // L.12: si se produce una excepción
   }                    // L.13:
return pass ? (puts("Acierto!"),0) : (puts("Fallo!"),1);
}

void festival(bool firsttime){
   if(firsttime) {Obj o; throw o; } // L.18: lanzar excepción
}
```

7.2. Secuencia de ejecución del manejo de excepciones

Como puede verse, la filosofía C++ respecto al manejo de excepciones no consiste en corregir el error y volver al punto de partida. Por el contrario, cuando se genera una excepción el control sale del bloque **try** que lanzó la excepción (incluso de la función), y pasa al bloque **catch** cuyo manejador corresponde con la excepción lanzada (si es que existe).

A su vez el bloque **catch** puede hacer varias cosas:

- Relanzar la misma excepción

- Saltar a una etiqueta
- Terminar su ejecución normalmente (alcanzar la llave de cierre).

Si el bloque **catch** termina normalmente sin lanzar una nueva excepción, el control se salta todos los bloques catch que hubiese asociados al bloque try correspondiente y sigue la ejecución del programa a continuación.

Puede ocurrir que el bloque **catch** lance a su vez una excepción. Lo que nos conduce a excepciones anidadas. Esto puede ocurrir, por el hecho de que se ejecutan intrucciones y por tanto es posible que alguna de ellas vuelva a generar a su vez una excepción (supongase que es una excepción lanzada por intentar cerrar un fichero que no existe o que no se puede cerrar)

Como se puede ver, un programador avanzado puede utilizar las excepciones C++ como un mecanismo de return o break multinivel, controlado no por una circunstancia excepcional, sino como un acto deliberado del programador para controlar el flujo de ejecución. se lanzan deliberadamente excepciones con la idea de moverse entre bloques a distintos niveles. Dado el nivel de este curso no se recomienda esta práctica hasta que no se esté totalmente familiarizado con la programación estructurada.

Relanzar una excepcion

Si se ha lanzado previamente una excepción y se está en el bloque que la ha capturado, es posible repetir el lanzamiento de la excepción (el mismo objeto recibido en el bloque catch) utilizando simplemente la sentencia throw sin ningún especificador. No perder de vista que el lanzamiento throw, solo puede realizarse desde el interior de un bloque try, al que debe seguir su correspondiente "handler".

El siguiente ejemplo sería incorrecto, dado que se intenta relanzar una excepción desde fuera de un bloque try:

```
try {
  ...
  if (x) throw A();  // lanzar excepción
}
catch (A a) {        // capturar excepción
  ...                // hacer algo respecto al error
  throw;             // Error!! no está en un bloque try
}
```

El modo más habitual de realizar un relanzamiento es debido a que se trata de un bloque try anidado en otro bloque try (en la misma función o en funciones jerarquicamente por encima en la pila de llamadas). El siguiente ejemplo refleja este modo de proceder:

```
void foo();
void main () {
  try {
    ...
    foo();
  }
  catch (A a) {
    ...
  }
  return 0;
}
void foo() {
  ...
  if (x) try {
    throw A();
  }
  catch (A) {
    ...
    throw;
  }
}
```

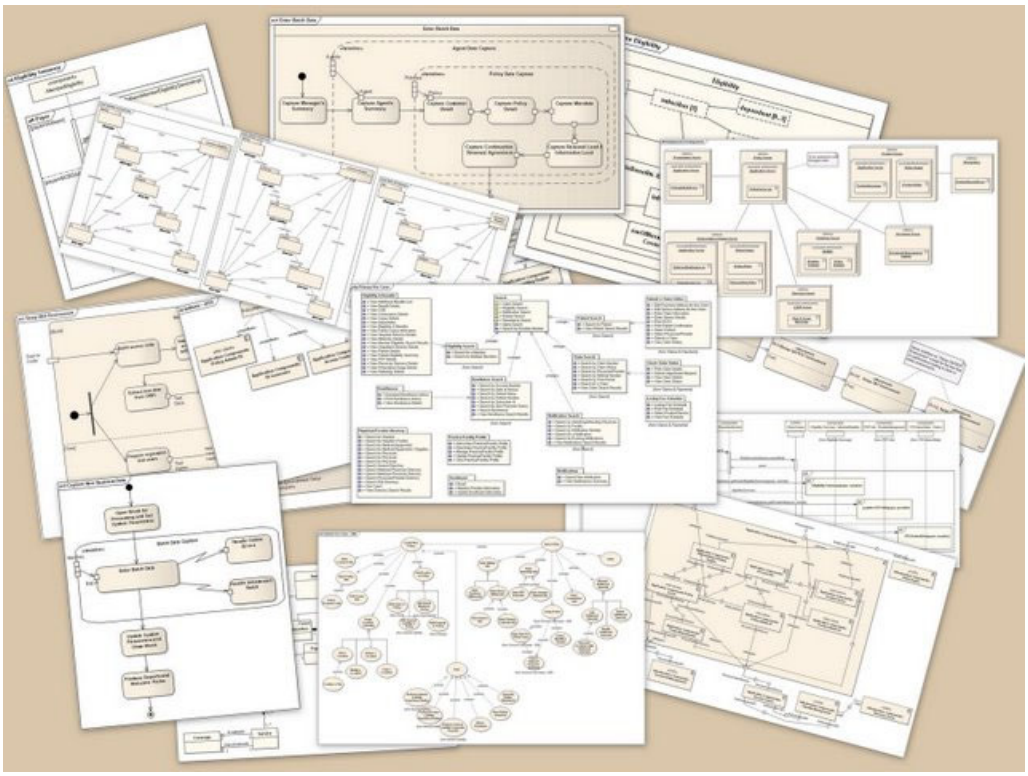
8. Breve Introducción a la representación UML

El término UML proviene de las siglas inglesas de Lenguaje Unificado de Modelado (Unified Modeling Language) y es el lenguaje de modelado de sistemas software más conocido y utilizado en la actualidad, y se encuentra respaldado por el OMG (Object Management Group).

El objetivo de UML es “proporcionar a desarrolladores de software, arquitectos de sistemas e ingenieros de software de herramientas para el análisis, diseño e implementación de sistemas basados en software, así como modelar procesos de negocio y similares.

Es importante remarcar que UML es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo. Se puede aplicar en el desarrollo de software gran variedad de formas para dar soporte a una metodología de desarrollo de software, pero UML en sí no especifica qué metodología o proceso usar.

Por eso es importante seleccionar correctamente el modelo adecuado y el nivel de representación correcto. UML es muy extenso y en la práctica, la mayoría de los sistemas requieren para su modelado sólo del uso de un 10 o 20% de las posibilidades dadas por UML.



Collage obtenido de Wikipedia en el que se reflejan algunos de los múltiples diagramas que esfecifica UML.

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas .

Finalmente no hay que confundir UML con un lenguaje de programación. De hecho, a pesar de constituir un estándar prácticamente aceptado por el conjunto de programadores, no deja de recibir ciertas críticas debido a la ambigüedad en la interpretación que se puede realizar en los distintos diagramas. Además, al constituir una serie de herramientas de modelado, en general, una misma realidad se representa o dibuja desde distitnas perspectivas que se complementan para finalmente dotar al conjunto de un significado. Luego por eso es muy importante que los diagramas se centren en aquel aspecto que quieren mostrar y no intentar reflejar toda la realidad en un solo gráfico.

8.1. Elementos de contrucción de los diagramas UML

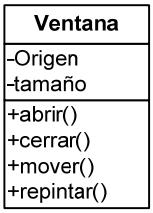
Los diagramas de UML contienen cinco tipos de elementos diferenciados:

1. **Elementos estructurales**: Puede entenderse como la representación de los nombres dentro de un modelo de algo. Por ejemplo, en una urbanización, casa, árbol, carretera…. Son sustantivos del modelo. Constituyen la parte más estática del modelado representando tanto elementos conceptuales como físicos.
2. **Elementos de comportamiento**: de forma análoga, podemos establecer que vienen a ser como los verbos del lenguaje de modelado. Representan comportamientos en el tiempo y en el espacio. Existirán elementos de comportamiento que nos servirán para especificar como evolucionan las cosas además de cómo se relacionan.
3. **Elementos de agrupación**. Son las partes organizativas del modelo. Establece las divisiones en que se puede fraccionar un modelo.
4. **Elementos de anotación**. Como su propio nombre indica son las partes explicativas del modelo UML. Iluminan, explican, detallan aspectos de cualquier elemento del modelo.
5. **Relaciones**. Nos permiten reflejar los modos en que los elementos del modelo se relacionan entre sí.

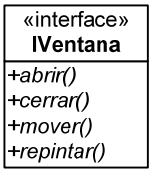
Aunque la explicación detallada de cada uno de estos conjuntos de elementos excede la finalidad de este capítulo, se va a proceder a continuación a enumerar y describir brevemente los distintos componentes así como su representación para cada una de las categorías:

Elementos estructurales

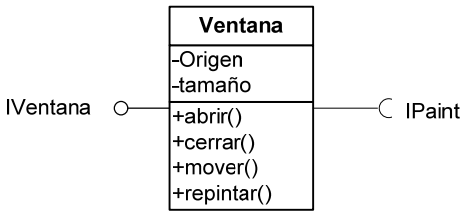
- **CLASE**: Refleja un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Lógicamente constituirá una representación unívoca del elemento homónimo de C++. Se refleja como un rectángulo dividido en tres secciones: nombre, atributos y métodos.



- **INTERFAZ**: Una interfaz corresponde a un conjunto de operaciones que especifican un servicio dado por una clase o componente. Por tanto describe el coportameinto externamente visible de estos elementos. Se diferencia claramente de una clase en en el sentido de que carece de atributos. En C++ no existe de forma directa el concepto de interfaz, aunque estas se pueden realizar por medio de clases virtuales puras sin atributos, y con todos los métodos abstractos. En JAVA este elemento existe como entidad propia. Su representación gráfica se realiza por medio de un caja que lleva el título de <<interfaz>> antes del nombre, y que está dividida en dos zonas (la de identificación, y la de métodos).



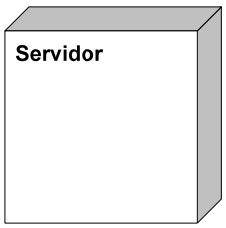
Una interfaz suministrada por una clase al resto de componentes (conjunto de operaciones agrupadas que dan un servicio, por lo que una clase puede tener varias interfaces), se especifica por medio de un circulo unido por una línea a la clase. De igual forma, si una clase requiere de acceder a una interfaz de otro componente, esto se refleja por medio de un semicírculo unido por un segmento a la clase. Por ejemplo, la clase ventana implementa la intefaz IVentana, y requiere del uso de la interfaz IPaint (que es un servicio que le permitirá dibujarse en algún lugar):



- **Caso de uso**: Es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular. Aunque es un concepto importante, de momento para los objetivos de este curso simplemente emplazamos aquí su representación:

- **Componente**: Es una parte modular del sistema de diseño, agrupando por tanto sus elementos lógicos (clases, interfaces), y mostrando fundamentalmente un conjunto de funcionalidades utilizables por el exterior. Por ejemplo, un componente podría ser el reporductor de Video de un sistema de ventanas. Internamente contendrá una complejidad elevada, pero finalizado el componente, lo que se necesita para su uso es las interfaces publicas del componente (abrir un fichero, reproducirlo, subir el volumen… etc). Un aspecto importante es que un componente en principio puede ser intercambiado por otro siempre que se mantenga la interfaz.

- **Nodo**: Elemento físico que existe en tiempo de ejecución y que representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, de capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo.



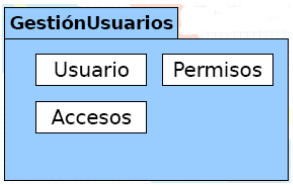
Elementos de comportamiento

Interacción: Comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular para conseguir un propósito específico. Involucra otros muchos elementos, incluyendo mensajes, secuencias de acción (comportamiento invocado por un objeto) y enlaces (conexiones entre objetos)

Estado: Se utiliza para reflejar en una secuencia los estados por los que pasa un objeto o una interacción en respuesta a eventoz o sucesos en general.

Elementos de agrupación

Paquete: Es un mecanismo de propósito general para organizar elementos (estructurales, de comportamiento, e incluso otros elementos de agrupación) en grupos. Al contrario de los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Un paquete forma un espacio de nombres, y se pideen anidar, aunque deben avistarse pasquetes muy anidados:



Elementos de anotación

Nota: el tipo principal de anotación. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones cobre cualquier elemento de un modelo

Elementos de relación

Una relación es una conexión entre elementos. Para diferenciar las distintas relaciones se utilizan diferentes tipos de líneas. Hay cuatro tipos de relaciones: dependencia, asociación, generalización y realización

Dependencia: Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente).

Asociación: Relación estructural que describe un conjunto de enlaces, los cuals son relaciones entre objetos. La agregación es un tipo especila de asociación y representa una relación estructural entre un todo y sus partes.

Generalización: Es una relación de especialización/generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre.

Realización: Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan.

8.2. Modelado estructural

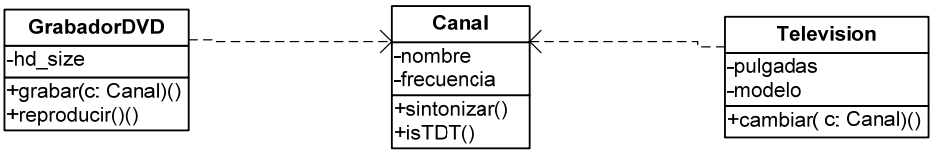
El modelado es la parte de UML que se ocupa de identificar todas las partes importantes de un sistema así como sus interacciones. De igual forma se entiende como modelado estructural, al modelo de los aspectos estáticos de un sistema, para lo cual se utilizan fundamentalmente como sustantivos o elementos estructrales básicos las clases y las interfaces así como sus relaciones entre ellas. Dado que en estos apuntes sólo se pretende dar unas nociones que nos permitan representar la estructura de clases y algo de la evolución de nuestros programas, se procede ahroa a ver los elementos más comunes en este tipo de diagramas denominados DCD (Diagramas de Clases de Diseño).

Un aspecto muy importante a la hora de realizar el modelo estructural de un sistema es el análisis de responsabilidades de las clases y componentes. Entendemos como responsabilidad de una clase al conjunto de fines para el cual esta clase es creada así como sus obligaciones. Es muy buena práctica iniciar la programación especificando las responsabilidades de cada clase. En este proceso es importante abstraer lo necesario y suficiente, y como consecuencia hay que evitar dar excesivas responsabilidades a una sola clase pero tampoco obtener clases con muy pocas responsabilidades que ni siquiera tengan entidad.

La forma en que representamos las interacciones entre clases queda reflejado por sus relaciones. En los diagramas de modelado sin embargo no hay porque reflejar TODAS las relaciones existentes, porque en ese caso el diagrama puede convertirse en algo ilegible. Las relaciones más habituales (y es aquí en donde nos centraremos en este curso, dejando otros diagramas y relaciones para más adelante) son:

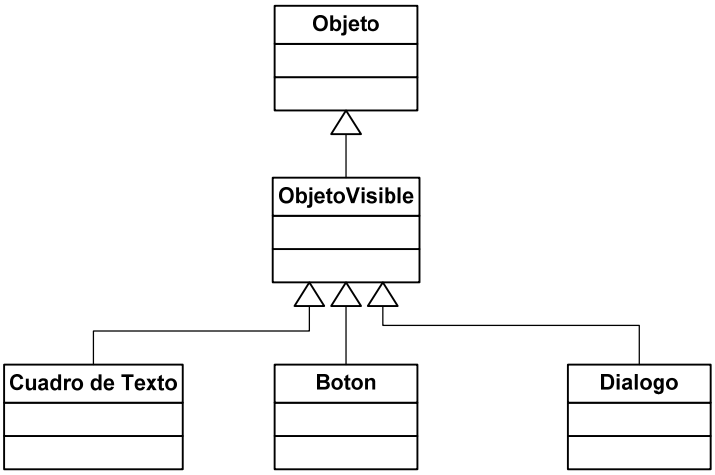
Relación de dependencia:

Un cambio en la especificación del elemento independiente puede afectar al otro elemento implicado en la relación. En el ejemplo tanto la Televisión como el Grabador USAN un determinado Canal. De forma que si cambia la interfaz o el comportamiento de los objetos de tipo Canal, habría que modificar el Grabador y la Televisión, cosa que no ocurre en sentido contrario. Luego existe una dependencia entre clases. Es importante destacar que el Canal no necesita conocer al Grabador ni a la Televisión. En C++ esta relación habitualmente se traduce en una sentencia #include que permite usar la clase de la cual se depende.



Relación de Generalización:

Es la relación que se establece entre un concepto o elemento general (superclase) y un caso más específico de ese elemento (subclase o hijo). El caso más claro es el de una clase (hija) que hereda de otra clase (padre). Como se comentó anteriormente, los objetos hijos se podrán utilizar en cualquier lugar donde aparezca un objeto de tipo padre.

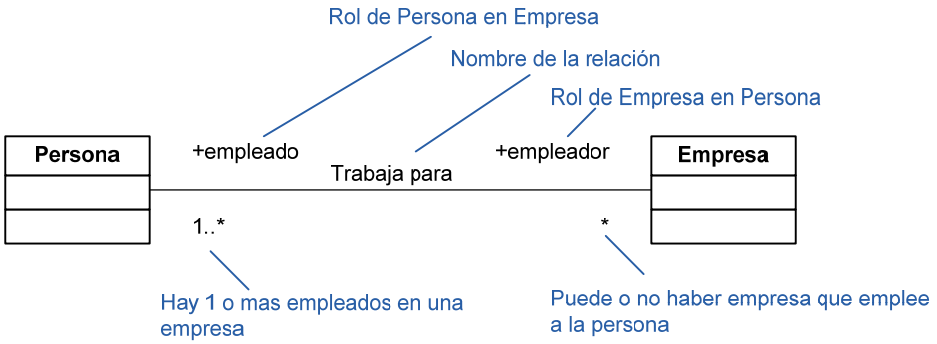


Relación de Asociación:

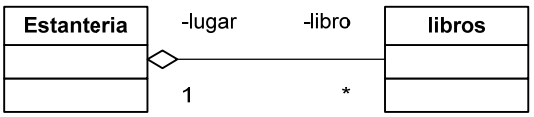
Mediante esta relación representamos como los objetos de un elemento están conectados con los objetos de otros. Pueden establecerse incluso relaciones recursivas, es decir que un objeto esté asociado a un objeto del mismo tipo.

En las relaciones de asociación se suelen incluir adornos para facilitar su comprensión, y para diferenciar los distintos modos que la asociación puede realizarse:

- Nombre: nombre de una asociación que describe la naturaleza de la relación (“contiene”)
- Rol: Aspecto que cada extremo presenta a la clase que se encuentra en el otro lado de la relación (una mesa contiene patas, por lo que las patas son “parte” de la mesa y la mesa es el “contenedor” para las patas, dentro de una relación de “contener”)
- Multiplicidad: Indica cuantos objetos se pueden conectar a través de una instancia de la asociación.



- Agregación: Representa una relación estructural entre iguales (sirve para relacionar todo /parte, pero en el que tanto el todo como la parte tienen una "vida" propia. En C++ refleja agregaciones de objetos por referencia.



- Composición: Como una agregación simple pero en este caso el todo da la "vida" a la parte. En C++, refleja el hecho de que un objeto o conjunto de objetos son atributos del objeto que los contiene.

8.3. Diagramas

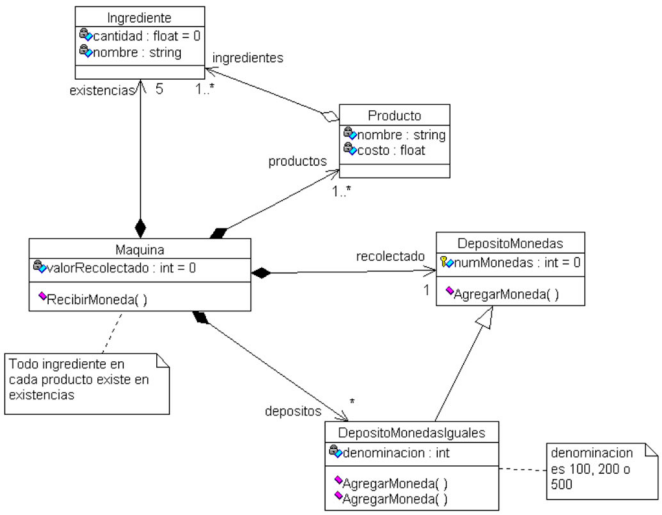
Diagramas de Clases:

Dado que solo se trata de una breve introducción, incluimos en este capítulo uno de los diagramas más recurrentemente utilizados en DOO, el diagrama de clases de diseño. En este se muestra un conjunto de clases, con interfaces, colaboraciones y sus relaciones.

Se usa fundamentalmente para modelar el vocabulario del sistema (abstracciones que son parte del sistema y las que no lo son) así como modelar colaboraciones simples, y el esquema lógico de una base de datos. Por tanto se utilizan para visualizar los aspectos estáticos de los bloques de construcción del sistema.

Para trazar un diagrama de clases en el que se reflejen las colaboraciones simples, en primer lugar habrá que identificar las funciones o comportamientos del sistema que se está modelando, y que quedarán reflejados por el diagrama.

Para cada elemento entonces, habrá que identificar las clases, iterfaces y relaciones con otros elementos. Habrá que ir dontando a estos elementos de contenido, intentando que haya un reparto de responsabilidades entre clases y terminando por convertir las responsabilidaders en atributos.



Diagramas de Sequecia:

Otro de los diagramas más recurrentes cuando se desea reflejar el comportamiento dinámico de las interacciones entre distintos elementos es el diagrama de secuencia.

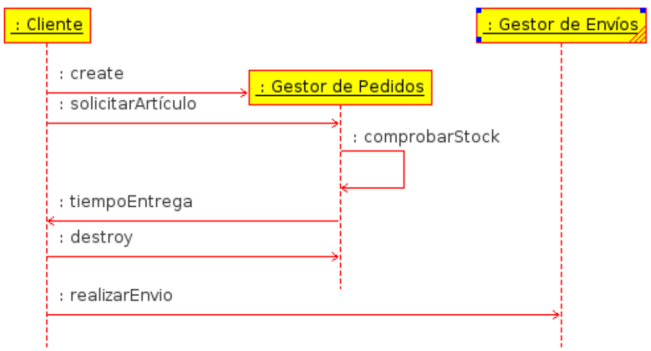
Se modelan los aspectos dinámicos de un sistema mediante interacciones. Una interacción es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos dentro de un contexto para lograr un propósito y se utiliza para modelar el flujo de control dentro de una operación, una clase, un componente, un caso de uso o el propio sistema. En este contexto, se considera un mensaje a la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad. Los tipos basicos de mensaje son:

- Llamada: Invoca una operación sobre un objeto
- Retorno: Devuelve un valor al invocador
- Envío: Envía una señal a un objeto
- Creación
- Destrucción

Las interacciones aparecen en la colaboración de objetos existentes en el ambito de un sistema o un subsistema, o entre los objetos de un mismo subsistema en la implementación de una operación, o en el contexto de una clase (cómo los atributos y diferentes operaciones interaccionan entre sí para dar lugar a una nueva operación).

Los objetos que participan en una interacción son o bien elementos concretos (objetos) o bien elementos prototípicos (clases, nodos, actores y casos de uso).

Un diagrama de secuencia sería el reflejado a continuación:



A diferencia de lo que ocurre en un diagrama de colaboración, destaca la ordenación temporal de los mensajes. Con el se logra una representación visual clara del flujo de control a lo largo del tiempo.

Algunas características destacabñes son:

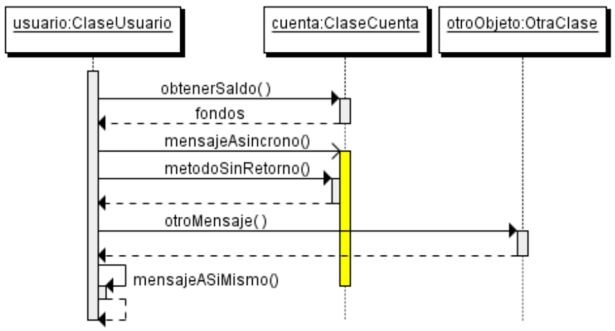
La línea de vida de las instancias o clases: Representa la existencia de un objeto a lo largo de un periodo de tiempo. Si se crean o destruyen objetos durante la interacción, sus líneas de vida aparecen y desaparecen cuando reciben los mensajes estereotipados >>create>> y <<destroy>> respectivamente. La línea de vida son las líneas verticales del diagrama.

El foco de control: el cual representa el periodo de tiempo durante el cual un objeto ejecuta una acción. Se representan con rectángulos vacíos sobre la línea de vida.

A su vez los mensajes se pueden clasificar adicionalmente como:

- Síncronos se corresponden con llamadas a métodos del objeto que recibe el mensaje. El objeto que envía el mensaje queda bloqueado hasta que termina la llamada. Se representan con flechas con la cabeza llena.
- Asíncronos, estos mensajes terminan inmediatamente, y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas con la cabeza abierta.

También se representa la respuesta a un mensaje con una flechadiscontinua



9. Introducción a los Sistemas Operativos

A lo largo de la historia de las computadoras, el sistema operativo no ha dejado de evolucionar como las necesidades de los usuarios y las capacidades de los sistemas informáticos que han cambiado. Como Weizer (1981) ha señalado, los sistemas operativos han evolucionado desde la década de 1940 a través de una serie de generaciones distintas, que corresponden aproximadamente a las de las décadas.

A finales de los 40's el uso de computadora estaba restringido a aquellas empresas o instituciones que podían pagar su alto precio, y no existían los sistemas operativos. En su lugar, el programador debía tener un conocimiento y contacto profundo con el hardware, y en el infortunado caso de que su programa fallara, debía examinar los valores de los registros y páneles de luces indicadoras del estado de la computadora para determinar la causa del fallo y poder corregir su programa, además de enfrentarse nuevamente a los procedimientos de reservar tiempo del sistema y poner a punto los compiladores, enlazadores, etc; para volver a correr su programa, es decir, enfrentaba el problema del procesamiento serial (serial processing).

La importancia de los sistemas operativos nace históricamente desde los 50's, cuando se hizo evidente que el operar una computadora por medio de tableros enchufables en la primera generación y luego por medio del trabajo en lote en la segunda generación se podía mejorar notoriamente, pues el operador realizaba siempre una secuencia de pasos repetitivos, lo cual es una de las características contempladas en la definición de lo que es un programa. Es decir, se comenzó a ver que las tareas mismas del operador podían plasmarse en un programa, el cual a través del tiempo y por su enorme complejidad se le llamó "Sistema Operativo". Así, tenemos entre los primeros sistemas operativos al Fortran Monitor System (FMS) e IBSYS.

Posteriormente, en la tercera generación de computadoras nace uno de los primeros sistemas operativos con la filosofía de administrar una familia de computadoras: el OS/360 de IBM. Fue este un proyecto tan novedoso y ambicioso que enfrentó por primera vez una serie de problemas conflictivos debido a que anteriormente las computadoras eran creadas para dos propósitos en general: el comercial y el científico. Así, al tratar de crear un solo sistema operativo para computadoras que podían dedicarse a un propósito, al otro o ambos, puso en evidencia la problemática del trabajo en equipos de análisis, diseño e implantación de sistemas grandes.

El resultado fue un sistema del cual uno de sus mismos diseñadores patentizó su opinión en la portada de un libro: una horda de bestias prehistóricas atascadas en un foso de brea. Surge también en la tercera generación de computadoras el concepto de la multiproceso, porque debido al alto costo de las computadoras era necesario idear un esquema de trabajo que mantuviese a launidad central de procesamiento más tiempo ocupada, así como el encolado (spooling) de trabajos para su lectura hacia los lugares libres de memoria o la escritura de resultados. Sin embargo, se puede afirmar que los sistemas durante la tercera generación siguieron siendo básicamente sistemas de lote.

En la cuarta generación la electrónica avanza hacia la integración a gran escala, pudiendo crear circuitos con miles de transistores en un centímetro cuadrado de silicio y ya es posible hablar de las computadoras personales y las estaciones de trabajo. Surgen los conceptos de interfaces amigables intentando así atraer al público en general y se promueve el uso de las computadoras como herramientas cotidianas. Se hacen populares el MS-DOS y UNIX en estas máquinas. También es común encontrar clones de computadoras personales y una multitud de empresas pequeñas ensamblándolas por todo el mundo.

Para finales de los 80's, comienza el auge de las redes de computadores y la necesidad de sistemas operativos en red y sistemas operativos distribuidos. La red mundial Internet se va haciendo accesible a toda clase de instituciones y se comienzan a dar muchas soluciones (y problemas) al querer hacer convivir recursos residentes en computadoras con sistemas operativos diferentes. Para los 90's el paradigma de la programación orientada a objetos cobra auge, así como el manejo de objetos desde los sistemas operativos. Las aplicaciones intentan crearse para ser ejecutadas en una plataforma específica y poder ver sus resultados en la pantalla o monitor de otra diferente (por ejemplo, ejecutar una simulación en una máquinacon UNIX y ver los resultados en otra con DOS). Los niveles de interacción se van haciendo cada vez más profundos.

Actualmente podemos resumir que de todo este proceso histórico se extrae que un Sistema Operativo es un conjunto de programas que controla los dispositivos que forman el ordenador (memoria y periféricos), administra los recursos y gestiona la ejecución del resto del software. De alguna forma actúa como enlace entre el usuario y los programas y el hardware del ordenador. Aunque ahora los veremos en más detalle, los objetivos básicos de un sistema operativo podrían resumirse en dos:

- Realizar una eficiente gestión de los recursos del ordenador
- Ocultar los detalles específicos de funcionamiento de los dispositivos, consiguiendo de esta forma un cierto nivel de abstracción respecto del hardware.

9.1. Funciones de un sistema operativo

El objetivo fundamental de los sistemas operativos es gestionar y administrar eficientemente los recursos de un computador, permitiendo la ejecución de programas sin que se produzcan conflictos en el acceso de los recursos utilizados, y sin que ningún programa monopolice un medio determinado.

El sistema operativo efectúa, entre otras, las siguientes funciones:

Desde el punto de vista del usuario común:

- Comandos para entrar y abandonar el sistema.
- Órdenes para modificar la clave de entrada.
- Comandos para definir las características de un terminal.
- Establecer las rutas de búsqueda.
- Ejecución y control de programas.
- Para establecer prioridades en los procesos.
- Para la manipulación de ficheros y subdirectorios.
- Para la información de estado.
- Órdenes de administración

Desde el punto de vista del programador de aplicaciones:

- Creación de procesos y borrado.
- Comunicación y sincronización de procesos.
- Actividades de temporización.
- Gestión y uso de recursos.
- Asignación y liberación de memoria.
- Establecimiento de prioridades.

Desde el punto de vista de la seguridad del sistema:

• Protección de E/S: Para conseguirla se diferencian dos modos de operación: modo usuario y modo supervisor. El cambio de un modo a otro se controla por parte del S.O., siendo sólo posible el cambio a modo supervisor desde un usuario por medio de llamadas a funciones del S.O. De este modo ciertas instrucciones sólo se ejecutarán en modo supervisor y el S.O. Podrá controlar como se realiza la E/S.

• Protección de la memoria: para que la protección de memoria sea eficiente, se necesita generalmente recursos hardware por los que se controla el acceso a la memoria. La implementación de este control, varía dependiendo de la gestión que se haga.

De forma resumida podemos establecer las funciones del sistema operativo en base a los gestores básicos de los que habitualmente consta:

1. Gestión de la CPU o **gestión de procesos**: Responsable de iniciar los programas, finalizarlos, interrumpirlos, reanudarlos, darles tiempo de CPU etc. También debe permitir la comunicación de la CPU con el exterior.

2. **Gestión de memoria**: Controla la cantidad de memoria que necesita cada programa. Permite la coexistencia de varios procesos en memoria central.

3. **Gestión de E/S**: Los programas acceden a los periféricos de forma sencilla.

4. **Gestión de dispositivos de almacenamiento**: organiza la información en archivos y carpetas y permite el acceso rápido y eficiente a dicha información.

5. **Intérprete de comandos**: Las órdenes del usuario son interpretadas y llevadas a cabo.

Gestión de procesos

Estrategias para la gestión de procesos

Existen dos formas básicas de trabajar con un computador: por lotes y de forma interactiva. En esta última, la CPU está constantemente atendiendo al usuario, y en cualquier caso, aunque la CPU tenga varios procesos en memoria, se tiene la impresión de que el usuario está trabajando directamente con el computador. Los primeros sistemas operativos funcionaban como **monotarea** o serie; es decir, hasta que no finaliza la ejecución de un programa no empieza a ejecutarse otro.

Un sistema operativo **multitarea** aprovecha los tiempos inactivos de la CPU, que habitualmente son provocados por los periféricos; así mismo, se deben aprovechar los espacios de la memoria principal no ocupados por los procesos. La multitarea consiste, en esencia, en cargar en la memoria principal varios procesos e ir asignando la CPU, de forma alternativa, a los distintos procesos que estén en ejecución, de forma que se aproveche al máximo la CPU y que varios procesos vayan avanzando en su ejecución, sin necesidad de que finalice completamente uno para iniciar la ejecución de otro. Este modo de funcionamiento también se denomina ejecución concurrente de procesos

Según el sistema operativo en particular se definen diferentes estados para un proceso. Los estados básicos son activo, bloqueado y preparado. Se dice que un proceso está en estado activo, o de ejecución, cuando la CPU está ejecutando sus instrucciones. Se dice que un proceso entra en estado de bloqueo cuando la CPU no puede continuar trabajando con él, a causa de tener que esperar a la realización de una operación de entrada/salida, o a algún otro evento de naturaleza similar. Se dice que un proceso está en estado preparado, o ejecutable, cuando la CPU puede iniciar o continuar su ejecución.

En los sistemas multitarea, cuando un proceso entra en estado de bloqueado, un módulo del sistema operativo denominado distribuidor ("*dispacher*") pasa el turno de ejecución a uno de los procesos de la memoria principal que esté en estado preparado. Para ello, se produce una interrupción que provoca una conmutación de contexto entre procesos. El **cambio de contexto** implica almacenar en una zona de la memoria principal toda la información referente al proceso interrumpido. Esta información, denominada contexto de un proceso, está referida a los contenidos de los registros de la CPU, indicadores de los biestables, punteros a archivos de discos, contenido de la pila, etc.; así como, información de las tablas referentes al estado de los procesos en memoria. El cambio de contexto supone un consumo de tiempo de CPU por parte del distribuidor. Cuando el distribuidor vuelva a dar el turno al proceso interrumpido, por haber finalizado la operación de E/S; es decir, haber salido del estado de bloqueado y entrado en el de preparado, debe producirse la recuperación de contexto en el sentido contrario al anterior, ya que, tiene que restituirse desde la zona auxiliar de memoria los contenidos de los registros salvados, quedando la CPU tal y como estaba en el instante que interrumpió este proceso, y preparada, por tanto, para proseguir su ejecución. Estas operaciones ocurren sucesivamente para cada uno de los procesos existentes en memoria principal.

El tipo de multitarea descrito anteriormente, en el que un proceso deja de ejecutarse por la CPU cuando se produce un evento relativo a una E/S o similar, tiene el inconveniente de que un proceso que contiene muchas operaciones de procesamiento y pocas E/S, puede monopolizar la CPU, hasta que finalice su ejecución. Los sistemas multitarea no necesariamente deben esperar a que un proceso pase al estado de bloqueado para que el

distribuidor lo interrumpa y dé el turno a otro proceso que esté preparado. El sistema operativo debe implementar los correspondientes mecanismos para permitir la ejecución intermitente de todos sus procesos.

Un sistema multitarea debe disponer de las técnicas apropiadas de protección de memoria y de control de concurrencias para permitir el acceso compartido a dispositivos de E/S y archivos. Un sistema **multiusuario** prevé el uso compartido de distintos usuarios, debiendo resolverse cuestiones como la identificación, autentificación, privilegios, y control de todos los usuarios.

El concepto de **tiempo compartido** ("*time sharing*") es una forma de gestionar la multiprogramación para obtener sistemas multiusuario, que requieren tiempos de respuesta adecuados, dando la ilusión a cada usuario que está trabajando en exclusiva con la máquina. En realidad, no se trabaja en exclusiva con el computador, ya que, una CPU sólo puede ejecutar un proceso en un instante dado. Con rigor se dice que la CPU está ejecutando procesos concurrentemente. Es decir, en un determinado intervalo de tiempo determinado la CPU está ejecutando alternativamente varios procesos ubicados en la memoria central.

Las técnicas de tiempo compartido requieren elegir, adecuadamente, un algoritmo de planificación de multitarea. Si existen varios procesos preparados en memoria, el problema que debe resolver el distribuidor es elegir a cuál de ellos darle el turno en la CPU, es decir, qué proceso debe cambiar a estado activo. El módulo del sistema operativo que se encarga de solventar este problema se denomina **planificador** ("*scheduler*"). Este proceso selecciona el proceso que va a ser ejecutado por la CPU en base a un algoritmo preestablecido.

Uno de los algoritmos de planificación de mayor interés, por su antigüedad, sencillez y amplio uso, es el de **petición circular** ("*round robin*"). Este algoritmo, también llamado cooperativo, ha sido usado por Windows y Macintosh. Con el algoritmo de petición circular, a cada uno de los procesos en memoria, se le asigna un intervalo de tiempo fijo, o periodo, llamado quantum. El objetivo de este algoritmo es cambiar de contexto de un proceso a otro, de forma rotatoria, conforme se van consumiendo su quantum.

En los sistemas operativos de tiempo compartido se cambia de contexto, no sólo cuando se le acaba un quantum a un proceso, sino también, en el instante en que quede bloqueado; es decir, el distribuidor provoca una conmutación de contexto del proceso P_i al P_j y da el turno a P_j siempre que se cumpla una de los dos condiciones siguientes:

a) El proceso P_i agote su quantum.

b) El proceso P_i se bloquee.

siendo P_j es el siguiente proceso preparado que le corresponde el turno.

La asignación de quantum suele hacerse con ayuda de un generador de pulsos sincronizado con la frecuencia de la red de suministro de energía eléctrica, 50 Hz en Europa, 60 Hz en Estados Unidos. Transcurrido el período de tiempo correspondiente, 20 ms, o 16.7 ms, se produce una interrupción que lanza a ejecución el módulo gestor del reloj de tiempo real, que entre otros objetivos sirve de referencia para generar el tiempo asociado al quantum.

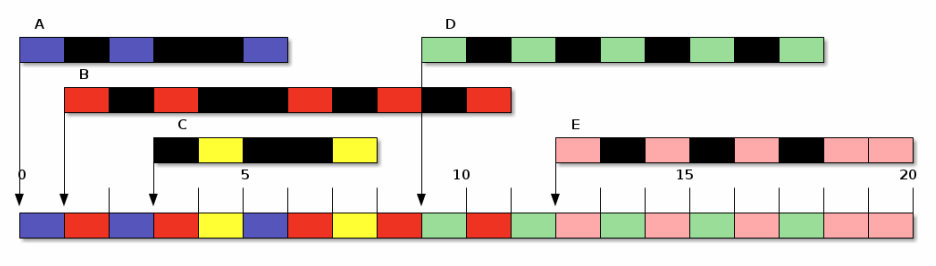


Grafico que muestra el comportamiento de un sistema de planificación "round - robin" para dar atención a procesos (A-E) que se van ejecutando de forma dinámica. En negro se pinta el estado de espera.

La elección del tiempo de quantum es problemática, ya que, si es demasiado pequeño hay que hacer muchas conmutaciones de contexto y la eficiencia de la CPU es baja, pero si es grande los tiempos de respuesta para cada usuario pueden llegar a ser muy bajos, si hubiese ejecutándose concurrentemente 20 procesos, al pulsar una simple tecla podría obtenerse respuestas tan lentas como 5 segundos.

Otro algoritmo de planificación de procesos es el de **asignación de prioridades**, usado por los sistemas operativos OS/2, UNIX y Windows-NT. Consiste en definir prioridades para los procesos, pudiendo el planificador modificar dicha prioridad dinámicamente. El distribuidor da el turno al proceso preparado que tenga asignada la mayor prioridad. Existen varios criterios de asignación de prioridades; así pues, para que el de mayor prioridad, y por tanto activo, no monopolice el uso de la CPU, a cada interrupción del reloj de tiempo real se le baja la prioridad del proceso activo.

Pueden existir **prioridades estáticas**, dadas en función de la relevancia de los usuarios, o **dinámicas**. La finalidad es obtener un aprovechamiento equilibrado de todos los recursos del computador y una mayor productividad; es decir, mayor número de programas ejecutados en un período determinado de tiempo. Se trata de dar mayor prioridad a los procesos que tienen más tiempo de E/S. Una forma de llevar a la práctica este criterio es dar al proceso P, la prioridad $1/f_i$ siendo f_i la fracción de tiempo del último quantum asignado a P_i que realmente ha consumido la CPU.

Otros sistemas operativos de gran interés son los sistemas operativos de **tiempo real**. El concepto de tiempo real hace referencia a que el computador debe garantizar la ejecución de un proceso, o obtención de un resultado dentro de un límite de tiempo preestablecido. Este tiempo puede ser pequeño o grande, dependiendo de la aplicación. Los sistemas de tiempo real se usan ampliamente en control industrial, equipos de conmutación telefónicas, control de vuelo, aplicaciones militares, simuladores, etc. Los sistemas operativos de tiempo real deben ser capaces de responder a los eventos, o interrupciones, que se puedan producir de forma asíncrona, a veces miles por segundo, en unos plazos de tiempo previamente especificados. Frecuentemente son sistemas multitarea en los que los algoritmos de planificación son del tipo de derecho preferencial, con los que siempre se ejecuta la tarea de mayor prioridad y no se interrumpe hasta que finalice, a no ser que se genere otra de prioridad mayor. El estándar POSIX tiene definidas unas extensiones de tiempo real, estableciéndose diferentes niveles de requisitos.

Estados de un proceso

Un proceso, desde el punto de vista de su ejecución, puede encontrarse en una de diversas situaciones o estados, los cuales se representan esquemáticamente en la figura 9.1. La mayoría de ellos ya han sido analizados anteriormente, por lo que se resumirán muy brevemente. Además, cada sistema operativo implementará un determinado conjunto de estados y unas condiciones de paso de un estado a otro, por tanto el funcionamiento aquí descrito pretende ser una referencia para la comprensión de la problemática relativa a la gestión de procesos. Un proceso **nonato** es aquel que no ha iniciado su ejecución. Un ejemplo es un programa retenido en una cola esperando a que el planificador de trabajos y el distribuidor le den paso a ejecución.

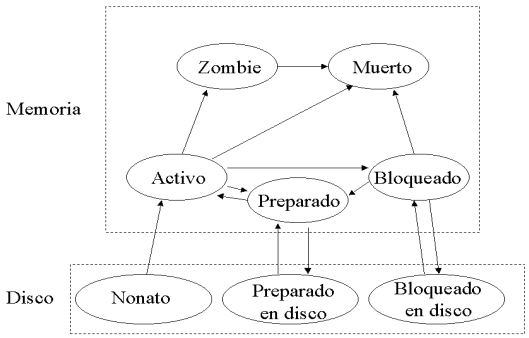


Figura 9. 1. Estados de un proceso

Un proceso está en estado **preparado**, **listo** o **ejecutable**, cuando se encuentra en memoria principal, sin operaciones de E/S pendientes, y apto para entrar en ejecución, en el instante que el distribuidor le asigne la CPU. El estado de ejecución o activo corresponde al proceso que en ese momento está siendo atendido por la CPU. Un proceso activo sale del estado de ejecución, en un sistema de tiempo compartido, cuando pasa a estado de bloqueado o el distribuidor le interrumpe para atender a otro de mayor prioridad, o que le corresponde el turno de ejecución.

Del estado de **bloqueado** se puede pasar al estado preparado cuando acaba la operación de E/S pendiente, o se le asigna el recurso esperado. También, es necesario tener en cuenta que algunos sistemas operativos realizan intercambio entre la memoria principal y el disco, lo cual les permite la ejecución concurrente de más procesos de los que admite la memoria principal, ya que un proceso puede ser trasvasado a disco, definiéndose por tanto el estado de bloqueado intercambiado, o bloqueado en disco. Un ejemplo de proceso bloqueado es aquel que solicita al operador del computador montar una determinada cinta y, si no hay memoria suficiente para los otros trabajos en progreso, lo trasvasa a disco. Si la carga de procesos es grande, incluso los procesos preparados pueden trasvasarse a disco, pasando así al estado de preparado intercambiado o preparado en disco.

Cuando desde un terminal interactivo se activa la comunicación con el computador, o se le da la vez a un proceso de la cola serie, el proceso correspondiente entra en estado de preparado, desplazándose por los estados del diagrama de la figura 9.1. Cuando finaliza la ejecución de un proceso o se detecta un error grave en él, el proceso pasa a estado de muerto, liberando el sistema operativo la zona de memoria que ocupaba.

Un punto importante en la gestión de procesos es la relación que se establece cuando un proceso crea otro, diciéndose que un proceso (P) puede generar a otro proceso, llamado proceso hijo (H). Puede ocurrir, por ejemplo, que el objetivo del proceso hijo sea generar cierta información para el padre y que, por problemas de sincronización entre ambos procesos, finalice la ejecución de P sin que el padre haya captado la información correspondiente. En este caso se dice que el padre esta en estado zombie; o sea, permanece en memoria, y no debe estar en estado de preparado, ni pasar a listo, ni a bloqueado, ya que finalizó su ejecución, del estado zombie debe pasar al estado de muerto, cuando finaliza la ejecución de todos los procesos hijos.

Gestión de la memoria

En los sistemas operativos de monoprogramación la memoria principal se puede organizar de diversas maneras. El sistema operativo puede ocupar las primeras posiciones de memoria, o las últimas, o incluso parte del sistema operativo puede estar en la zona RAM de direcciones bajas, y otra parte de él, como son los gestores de periféricos, o el cargador inicial, pudieran encontrarse en ROM en las direcciones altas. Esta última situación se corresponde con la de los antoguos PCs compatibles, en los que la ROM contiene el sistema básico de entradas y salidas o BIOS ("*Basic Input Output System*", Sistema Básico de Entradas y Salidas).

En cualquier caso, cuando el usuario da una orden, si el proceso que la implementa no está en memoria, el intérprete de comandos del sistema operativo se encarga de cargarlo en memoria desde disco, y el distribuidor da paso a su ejecución. Cuando finaliza el proceso, el sistema operativo visualiza el indicador de petición de orden y espera a que se le dé una nueva orden, en cuyo caso libera la zona de memoria ocupada, sobreescribiendo el nuevo programa sobre el anterior proceso.

En un sistema de multiproceso, cuando un programa se carga en memoria para ser ejecutado, o continuar su ejecución, el sistema operativo le asigna una dirección base, de acuerdo con los espacios libres de memoria, y transforma direcciones virtuales en direcciones físicas. Estas transformaciones suelen ser efectuadas por circuitos especializados que constituyen la MMU ("*Memory Management Unit*", Unidad de Gestión de Memoria), que en la actualidad se suele integrar en el mismo chip de la CPU.

La asignación de memoria para distintos procesos ejecutándose concurrentemente suele hacerse, dependiendo del sistema operativo, de alguna de las formas que se indican en los epígrafes siguientes:

- Particiones estáticas.
- Particiones dinámicas.
- Paginación.
- Segmentación.
- Memoria virtual.

Particiones estáticas

La técnica de particiones estáticas consiste en dividir la memoria en cierto número de bloques o zonas, cada una de las cuales contendrá un proceso. Las direcciones de base son

las direcciones de comienzo de cada partición. El tamaño de la partición es determinado por el operador o por el sistema operativo. Tamaños usuales son 8K, 16K, 32 K, o 64K.

El sistema operativo mantiene una tabla en la que cada fila corresponde a una partición, conteniendo la posición base de la partición; su tamaño, no todas las particiones tienen por qué ser iguales; y el estado de la partición, ocupada o no ocupada. El planificador de trabajos, una vez que una partición está libre, hace que se introduzca el programa de máxima prioridad que haya en la cola de espera y que quepa en dicha partición. Si el espacio de una partición es m palabras y el programa ocupa n posiciones, se verificará siempre que: $n \leq m$.

Particiones dinámicas

La utilización de particiones dinámicas se basa en que los programas son introducidos por el sistema operativo inicialmente en posiciones consecutivas de memoria, no existiendo, por tanto, particiones predefinidas. El sistema operativo puede gestionar el espacio de memoria usando una tabla de procesos, en la que cada línea contiene el número de proceso o identificativo del mismo, el espacio que ocupa y la dirección base. Existe una tabla complementaria a la anterior con los fragmentos o huecos libres. El planificador de trabajos periódicamente consulta la tabla, introduciendo en memoria los programas que quepan en los fragmentos.

Al iniciarse una sesión de trabajo se carga el primer programa, dejando un fragmento libre donde se pueden incluir otros programas. Al finalizarse los procesos, el número de fragmentos crecerá y el espacio disminuirá, llegando un momento en que el porcentaje de memoria aprovechado es muy reducido. El problema puede resolverse haciendo una compactación. Esta consiste en agrupar todos los fragmentos cuando acaba la ejecución de un proceso, quedando así sólo uno grande. La compactación se efectúa reubicando los procesos en ejecución.

Paginación

Con este procedimiento la memoria principal se estructura en marcos de páginas de longitud fija, también denominados bloques, que suelen ser de 512 bytes, 1Kbytes, 2Kbytes, 4Kbytes u 8 Kbytes. De esta forma, por ejemplo, la memoria de un computador de 256 Kbytes podría quedar dividida en 64 marcos de página de 4Kbytes cada uno. Cada marco de página se identifica con un número correlativo, en el caso anterior de 0 a 63. Asimismo, los procesos de

los usuarios se dividen en zonas consecutivas, denominadas páginas lógicas o virtuales, o simplemente páginas. Para un sistema dado, la capacidad de página y marco de página son coincidentes, de forma que cada página se memoriza en un marco.

El fundamento de la paginación reside en que no es necesario que un proceso se almacene en posiciones consecutivas de memoria. Las páginas se almacenan en marcos de página libres, independientemente de que estén o no contiguos. Cada marco de página contiene instrucciones consecutivas. Una instrucción de un proceso se localiza dando el número de marco y la dirección relativa dentro de él. El sistema operativo, en lugar de mantener una tabla de procesos, como en el caso de particiones dinámicas, mantiene tres tipos de tablas:

1. Tabla del mapa de memoria. Contiene tantas filas como marcos de página. Se indica el identificador del proceso que está en cada bloque y, en su caso, si está libre.

2. Tabla de procesos. Cada fila contiene información referente a cada proceso en ejecución. Se indica tamaño y la posición de memoria donde se encuentra la tabla mapa de páginas de ese proceso.

3. Tabla del mapa de páginas. Hay una por proceso, y contiene en número de marco donde se encuentra cada una de sus páginas. La longitud de cada tabla es variable, dependiendo del número de páginas; es decir, depende del tamaño de cada proceso.

Las tablas de paginación o tablas de páginas son una parte integral del Sistema de Memoria Virtual en sistemas operativos, cuando se utiliza paginación. Son usadas para realizar las traducciones de direcciones de memoria virtual (o lógica) a memoria real (o física) y en general el sistema operativo mantiene una por cada proceso corriendo en el sistema.

En cada entrada de la tabla de paginación (en inglés PTE, Page Table Entry) existe un bit de presencia, que está activado cuando la página se encuentra en memoria principal. Otro bit que puede encontrarse es el de modificado, que advierte que la página ha sido modificada desde que fue traída del disco, y por lo tanto deberá guardarse si es elegida para abandonar la memoria principal; y el bit de accedido, usado en el algoritmo de reemplazo de páginas llamado Menos Usado Recientemente (LRU, least recently used). También podrían haber otros bits indicando los permisos que tiene el proceso sobre la página (leer, escribir, ejecutar).

Dado que las tablas de paginación pueden ocupar un espacio considerable de la memoria principal, estas también podrían estar sujetas a paginación, lo que da lugar a una organización paginada de múltiples niveles (o tabla de páginas multinivel). En los sistemas con un tamaño de direcciones muy grande (64 bits), podría usarse una tabla de páginas invertida, la cual utiliza menos espacio, aunque puede aumentar el tiempo de búsqueda de la página.

Las tablas son mantenidas por el sistema operativo y utilizadas por la Unidad de Gestión de Memoria (MMU) para realizar las traducciones. Para evitar un acceso a las tablas de paginación, hay un dispositivo llamado Buffer de Traducción Adelantada (TLB, Translation Lookaside Buffer), acelerando el proceso de traducción

Segmentación

Se denomina segmento a un grupo lógico de información, tal como un programa, una subrutina, una pila, una tabla de símbolos, un array o una zona de datos. Esta unidad no es de un tamaño preestablecido, pues depende del programa de que se trate. Un programa ejecutable, listo para ser ejecutado por la CPU, está formado por una colección de segmentos. Así pues, un programa se considera dividido en segmentos, unos corresponderían al código ejecutable, y otros a los datos utilizados.

La gestión de los segmentos la realiza el sistema operativo, como con las particiones dinámicas, sólo que cada partición no corresponde a un proceso sino a un segmento. El sistema operativo mantiene una tabla con los segmentos de un proceso, de forma similar a la de páginas. Como el tamaño de los segmentos es variable, antes de realizar un acceso a memoria se comprueba que el desplazamiento no supere al tamaño del segmento, para proteger las zonas de memoria ocupadas por otro segmento.

Microprocesadores como el 80286 utilizan el direccionamiento con segmentación. También es usual combinar la segmentación con la paginación, tal es el caso de los computadores IBM-370, GE-645, microprocesadores de Intel como el 80386 y posteriores, etc.

La segmentación permite que ciertos procesos puedan compartir código, rutinas, o datos, sin necesidad de estar duplicados en memoria principal. Así pues, si seis usuarios están utilizando interactivamente un procesador de textos determinado, no es necesario que estuviesen cargadas en memoria seis copias idénticas del código del programa procesador de textos. El código del programa estaría una sola vez, y el sistema operativo se limitaría a anotar en las tablas-mapas de segmentos de cada uno de los procesos la dirección donde se encuentra el código. Por tanto, en un momento dado, en memoria existirían: segmentos asignados a trabajos concretos, segmentos compartidos, y bloques libres de memoria.

Memoria virtual

La memoria virtual permite a los usuarios hacer programas de una capacidad muy superior a la que físicamente tiene la memoria principal del computador. En realidad, la memoria virtual hace posible que la capacidad máxima de los programas esté limitada por el espacio que se le

reserve en disco, y no por el tamaño de la memoria principal. En definitiva, los sistemas con memoria virtual presentan al usuario una memoria principal aparentemente mayor que la memoria física real.

Por otra parte, la memoria virtual permite aumentar el número de procesos en la memoria principal en ejecución concurrente, ya que con ella sólo es necesario que esté en memoria principal un trozo mínimo de cada proceso, y no el proceso completo. Para la gestión de la memoria virtual suele utilizarse alguna de las anteriores técnicas de gestión de la memoria:

- gestión de memoria por páginas,
- gestión de memoria segmentada,
- gestión de memoria segmentada-paginada.

La memoria virtual se basa en que las instrucciones de un programa que se ejecutan sucesivamente, en un corto intervalo de tiempo, están en direcciones muy próximas, lo cual se conoce como principio de localidad temporal; y que los programas suelen estar redactados con gran linealidad; es decir, no suelen abundar los saltos entre posiciones de memoria distantes, que constituye el principio de localidad espacial.

En un sistema de memoria virtual se mantiene en disco un archivo con la imagen del proceso completo, que está troceado en páginas o segmentos, dependiendo del método. En cambio, en la memoria principal únicamente debe estar la página, o segmento, que en ese momento deba estar en ejecución, intercambiándose páginas entre disco y memoria principal cuando sea necesario, lo cual se conoce por "*swapping*".

La gestión de memoria virtual segmentada es más compleja que la del tipo de paginación, ya que, los segmentos son de tamaño variable y son más difíciles de gestionar. Las páginas, por el contrario, son de capacidad constante y preestablecida. Por ello, lo habitual es utilizar gestión de memoria por paginación o por segmentos paginados.

La memoria virtual con paginación combina las técnicas de paginación e intercambio. Por lo general, se utiliza un método de "intercambio perezoso" ("*lazzy swapper*"), únicamente se lleva a memoria una página cuando es necesaria para algún proceso, de esta forma, el número de procesos en ejecución concurrente puede aumentar.

Uno de los temas de mayor interés para el diseño del sistema de gestión de memoria virtual es la búsqueda de algoritmos eficientes para decidir qué página debe sustituirse en caso de fallo de página, ya que cada fallo de página supone una pérdida de tiempo de la CPU, que puede hacer disminuir considerablemente la productividad del computador incluso aunque

toda la gestión se realice con hardware específico. El algoritmo descrito en el párrafo anterior, que utilizan campo de números de referencias, es del tipo LRU. Los algoritmos más conocidos son los siguientes:

- **FIFO** ("First-in-first-out", Primero en Entrar Primero en Salir). Se basa en sustituir la página que lleve más tiempo en memoria
- **LRU** ("Least Recently Used", Último Recientemente Usado). En este caso se sustituye la página menos recientemente usada. Se basa en el principio de localidad temporal, ya que, supone que la página utilizada hace mayor tiempo es menos probable que se use próximamente.
- **NRU** ("No Recently Used", No Recientemente Usado). Una de las múltiples formas de implementar este tipo de algoritmo consiste en utilizar un bit de referencia que se pone a 1 cada vez que se usa la página. Un puntero recorre circularmente la tabla de marcos de página. Inicialmente, el puntero está detenido en un marco de página, cuando se tiene que desalojar una página analiza el bit de referencia del marco al que apunta, si es 1, lo pone a 0 y avanza apuntando a los siguientes procesos, cambiando los bits de referencia de 1 a 0, hasta que encuentre una página con el bit de referencia a 0, en cuyo caso la elimina de la memoria y se detiene en la posición siguiente de la tabla, hasta que se necesite desalojar una nueva página.

Gestión de entradas y salidas

Uno de los objetivos fundamentales del software de entradas y salidas (E/S) del sistema operativo, es que los programadores de aplicaciones puedan realizar las operaciones de E/S con independencia del dispositivo, transparente a las características particulares del hardware que se utiliza. Esto permite hacer programas que utilicen dispositivos y archivos de forma abstracta, sin necesidad de particularizarlos para los dispositivos donde estén almacenados dichos archivos; como son disquetes, discos duros, cintas DAT, etc. También, la independencia del dispositivo implica que se pueda asociar a cada uno de ellos un nombre simbólico, siendo las reglas de denominación uniformes para todos. En UNIX, por ejemplo, los dispositivos se referencian como si fuesen archivos.

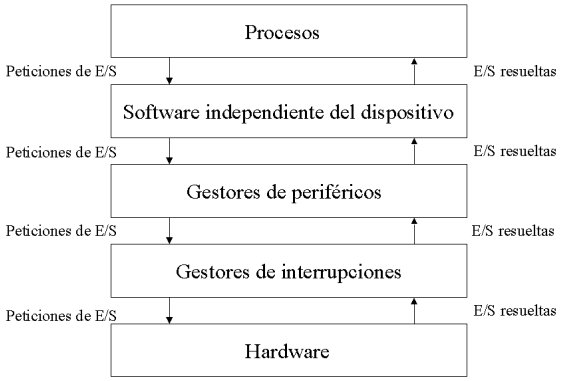


Figura 9. 2 . Gestión de E/S

Se comprende mejor como actúa el software de E/S utilizando un modelo conceptual por capas. El nivel inferior es el hardware, que es el que realmente ejecuta la operación de E/S, y el nivel superior los procesos de los usuarios. Resumidamente, las funciones de cada nivel son las que se indican en la figura 9.2.

El nivel más cercano al usuario es el **nivel de procesos**. Cuando se dispone de un programa con E/S en la fase de creación se une con procedimientos de biblioteca que gestionan dichas entradas y salidas. El objetivo básico de estos procedimientos es situar los parámetros en la zona de memoria adecuada y hacer las correspondientes llamadas al sistema operativo. El sistema operativo puede gestionar los dispositivos de E/S considerándolos de una de las tres formas siguientes:

1. **Dispositivos asignados**. Consiste en asignar ciertos dispositivos a un proceso durante la duración del trabajo; por ejemplo, un teclado, una impresora, una lectora de tarjetas.

2. **Dispositivos compartidos**. Así se asignan los dispositivos que pueden compartirse cocurrentemente por varios procesos, como es el caso de discos o cintas magnéticas. Si dos o más procesos requieren a la vez la misma unidad de cinta magnética, el sistema operativo debe evitar los conflictos que pudiesen plantearse.

3. **Dispositivos virtuales**. Se consigue que dispositivos en principio asignables que puedan compartirse, optimizándose el rendimiento del sistema. Esta técnica se describe con más detalle a continuación.

El sistema que implementa dispositivos de E/S virtuales se suele denominar "*spooler*" ("*Simultaneous Peripheral Operations On Line*", Operaciones Simultáneas Sobre Periféricos En Línea). La idea del gestor de dispositivos virtuales se aplica a periféricos lentos, y consiste en interponer entre el proceso y el periférico lento un dispositivo de memoria auxiliar rápido. Usualmente, el periférico lento es una impresora, un registrador gráfico, etc., y el rápido podría ser un disco. Se observa que los módulos "*spool*" hacen que los procesos trabajen con los periféricos de E/S como archivos en disco, aprovechándose de las mejores prestaciones de este tipo de periférico. Puede decirse que el disco se comporta como si se tratara de un dispositivo de E/S tipo digitalizador o impresora, es decir, el disco soporta dispositivos de E/S virtuales.

En el **nivel de software independiente del dispositivo** se incluyen funciones tales como la denominación de dispositivos, que consiste en la traslación del nombre lógico al nombre físico; sistemas de protección de datos, solicitando contraseña de accesos y tipo de acceso: leer, leer y escribir, etc.; agrupación de la información en el tamaño requerido para formar bloques físicos; gestionar la memoria intermedia entre periféricos y CPU; y la recuperación de cierto tipo de errores.

El siguiente **nivel es el de gestión de periféricos**, cada tipo de periférico tiene unas características propias y va conectado a través de un controlador físico. Cada periférico es utilizado por un programa gestor, que se encarga de programar el controlador y traducir peticiones abstractas del nivel de software independiente del dispositivo a código directamente interpretable por el hardware, y enviarlo a una lista de peticiones. Por ejemplo, en el caso de una unidad de disco, al presentarse el gestor correspondiente a una operación de leer un sector, el gestor debe comprobar si el disco está arrancado, posicionar el brazo en el cilindro correspondiente, inicializar el controlador de DMA, transmitir el marco de página de información, etc. En realidad, los gestores de periféricos son los únicos programas que deben tener en cuenta las peculiaridades concretas de los dispositivos.

Como ejemplo particular, en el caso de un disco para conseguir mayores velocidades es habitual leer físicamente cilindros completos, en lugar de sector a sector. Una vez localizado el cilindro se puede leer el cilindro completo a la velocidad de rotación del disco. La información del cilindro se almacena en la caché de disco de forma que si el programa del usuario ordena la lectura sucesiva de sectores de un mismo cilindro, cosa muy probable a causa de la localidad espacial de los datos, no se tendrán que consumir tiempos adicionales de latencia para acceder a los distintos sectores. Esta técnica es implementada por el nivel de gestor del periférico o incluso puede ser incluida en el nivel hardware, el controlador de disco puede incluir la memoria caché. No se incluye en el nivel de software de E/S independiente, ya que éste considera al disco como un conjunto sucesivo de bloques.

El siguiente es el **nivel de gestión de interrupciones**. Este es el nivel de software de E/S que está en contacto directo con el hardware. Cuando un proceso tiene una E/S se inicia la operación, pasa al estado de bloqueado, y espera hasta que acabe la operación de E/S. En la mayoría de los dispositivos de E/S las operaciones de lectura y escritura se inician y finalizan por medio de interrupciones. Cuando se produce la interrupción final un proceso debe cambiar el estado del proceso que ha finalizado la operación de E/S; por tanto, dicho proceso pasará de bloqueado a preparado. Los gestores de interrupciones son pequeños procedimientos que gestionan y lanzan la ejecución de los programas correspondientes al nivel inmediato superior.

Gestión de archivos.

Un archivo puede estructurarse en distintos soportes físicos, dependiendo del uso o capacidad que vaya a tener. Se distingue entre un nivel físico, más o menos Complejo, y nivel lógico, que proporciona una utilización adecuada para el usuario. El sistema operativo hace posible utilizar esta última, haciendo de interfaz entre los dos. En efecto, desde el punto de vista hardware, para almacenar datos o programas sólo existen direcciones físicas. En un

disco toda información se graba o lee en bloques ("*clusters*"), indicándose el número de unidad, la superficie, la pista, y el sector. El sistema operativo posibilita que el usuario no tenga que utilizar direcciones físicas, pudiéndose actuar sobre un archivo y almacenar o recuperar información sin más que indicar su nombre y utilizar ciertas instrucciones del lenguaje de control del sistema operativo.

Sobre la estructura del hardware citada anteriormente el sistema operativo construye dos abstracciones: la de archivo y la de directorio, denominándose sistema de archivos al conjunto de módulos del sistema operativo que se encarga de gestión de archivos y directorios.

Gestión de archivos

El concepto de archivo permite aislar al usuario de los problemas físicos de almacenamiento. En efecto, cuando el usuario desea referirse a un conjunto de información del mismo tipo como una unidad de almacenamiento, no tiene más que crear un archivo dándole el nombre correspondiente. Los archivos se conciben como estructuras con las siguientes peculiaridades:

- Deben ser capaces de contener grandes cantidades de información
- Su información debe permanecer y sobrevivir a los procesos que generan o utilizan
- Distintos procesos deben poder acceder a la información del archivo concurrentemente

Dependiendo del sistema operativo se pueden hacer unas u otras operaciones con los archivos. Cada archivo usualmente contiene nombre, atributos, y datos. Los atributos pueden incluir cuestiones tales como fecha y hora de creación, fecha y hora de la última actualización, bits de protección, contraseña de acceso, número de bytes por registro, capacidad máxima del archivo y capacidad actualmente ocupada.

Los datos se almacenan en el dispositivo de memoria masiva en forma de bloques. El dispositivo más usado para almacenamiento de archivos es el disco. Como la unidad física de almacenamiento es el bloque, éstos pueden grabarse de diversas formas:

- **Lista de enlaces**: Cada disco dispone de una tabla con tantos elementos como bloques físicos, la posición de cada elemento se corresponde biunívocamente con cada bloque, y contiene el puntero del lugar donde se encuentra el siguiente bloque del archivo. Cuando se abre un archivo el sistema de archivos se carga en la memoria principal la lista de enlaces, pudiéndose obtener rápidamente las direcciones, usualmente 3 bytes, de los bloques consecutivos del archivo. Presenta el

inconveniente de que si el disco es muy grande, la lista de enlaces ocupa una capacidad excesiva en memoria principal. Este sistema de grabación es propio de MS-DOS. La lista de enlaces se denomina FAT ("*File Allocation Table*", Tabla de Localización de Ficheros), y cada dirección de bloque se da con 2 bytes para los discos duros. Como cada elemento de la FAT corresponde a un bloque, se puede grabar en él información alternativa al puntero; por ejemplo, indicar si el bloque está deteriorado (H'FFF7), si está libre (H'0000) y por tanto disponible para su uso, o si es el último del archivo (H'FFFF).

- **I-nodos**: Corresponde a la forma de gestionar los archivos por el sistema operativo UNIX. Cada archivo tiene asociado un nodo de índices, o i-nodo, que es una pequeña tabla de tamaño fijo conteniendo los atributos del archivo y las direcciones de un número determinado de los primeros bloques del archivo. Los tres últimos elementos de la tabla indican indirectamente las siguientes direcciones de los bloques del archivo.

Esto da lugar a una serie de estándares de gestión de archivos en los que cabe destacar FAT, NTFS y EXT:

Sistema FAT32

Con diferencia es el sistema de gestión de archivos más utilizado en los dispositivos de almacenamiento móviles, dado que practicamente todos los sistemas operativos lo soportan. Es el sistema de archivos creado por Bill Gates en las versiones iniciales de DOS, que posteriormente ha evolucionado para ir admitiendo más capacidad y mejorar sus caracterísitcas pero manteniendo la compatibilidad en gran medida.

Tabla de asignación de archivos, comúnmente conocido como FAT (del inglés file allocation table), es un sistema de archivos desarrollado para MS-DOS, así como el sistema de archivos principal de las ediciones no empresariales de Microsoft Windows hasta Windows Me.

FAT es relativamente sencillo. A causa de ello, es un formato popular para disquetes admitido prácticamente por todos los sistemas operativos existentes para computadora personal. Se utiliza como mecanismo de intercambio de datos entre sistemas operativos distintos que coexisten en la misma computadora, lo que se conoce como entorno multiarranque. También se utiliza en tarjetas de memoria y dispositivos similares.

Las implementaciones más extendidas de FAT tienen algunas desventajas. Cuando se borran y se escriben nuevos archivos tiende a dejar fragmentos dispersos de éstos por todo el soporte. Con el tiempo, esto hace que el proceso de lectura o escritura sea cada vez más lento. La denominada desfragmentación es la solución a esto, pero es un proceso largo que debe repetirse regularmente para mantener el sistema de archivos en perfectas condiciones. FAT tampoco fue diseñado para ser redundante ante fallos. Inicialmente solamente soportaba

nombres cortos de archivo: ocho caracteres para el nombre más tres para la extensión. También carece de permisos de seguridad: cualquier usuario puede acceder a cualquier archivo.

FAT32 fue la respuesta para superar el límite de tamaño de FAT16 al mismo tiempo que se mantenía la compatibilidad con MS-DOS en modo real. Microsoft decidió implementar una nueva generación de FAT utilizando direcciones de cluster de 32 bits (aunque sólo 28 de esos bits se utilizaban realmente).

En teoría, esto debería permitir aproximadamente 268.435.538 clusters, arrojando tamaños de almacenamiento cercanos a los ocho terabytes. Sin embargo, debido a limitaciones en la utilidad ScanDisk de Microsoft, no se permite que FAT32 crezca más allá de 4.177.920 clusters por partición (es decir, unos 124 gigabytes). Posteriormente, Windows 2000 y XP situaron el límite de FAT32 en los 32 GiB. Microsoft afirma que es una decisión de diseño, sin embargo, es capaz de leer particiones mayores creadas por otros medios.

FAT32 apareció por primera vez en Windows 95 OSR2. Era necesario reformatear para usar las ventajas de FAT32. Curiosamente, DriveSpace 3 (incluido con Windows 95 y 98) no lo soportaba. Windows 98 incorporó una herramienta para convertir de FAT16 a FAT32 sin pérdida de los datos. Este soporte no estuvo disponible en la línea empresarial hasta Windows 2000.

El tamaño máximo de un archivo en FAT32 es 4 GiB (232−1 bytes), lo que resulta engorroso para aplicaciones de captura y edición de video, ya que los archivos generados por éstas superan fácilmente ese límite.

Sistema de ficheros NTFS

NTFS (del inglés New Technology File System) es un sistema de archivos de Windows NT incluido en las versiones de Windows 2000, Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista, Windows 7 y Windows 8. Está basado en el sistema de archivos HPFS de IBM/Microsoft usado en el sistema operativo OS/2, y también tiene ciertas influencias del formato de archivos HFS diseñado por Apple.

NTFS permite definir el tamaño del clúster a partir de 512 bytes (tamaño mínimo de un sector) de forma independiente al tamaño de la partición.

Es un sistema adecuado para las particiones de gran tamaño requeridas en estaciones de trabajo de alto rendimiento y servidores. Puede manejar volúmenes de, teóricamente, hasta 2^{64}–1 clústeres. En la práctica, el máximo volumen NTFS soportado es de 2^{32}–1 clústeres (aproximadamente 16 TiB usando clústeres de 4 KiB).

Su principal inconveniente es que necesita para sí mismo una buena cantidad de espacio en disco duro, por lo que no es recomendable su uso en discos con menos de 400 MiB libres.

Los nombres de archivo son almacenados en Unicode (UTF-16), y la estructura de ficheros en árboles-B, una estructura de datos compleja que acelera el acceso a los ficheros y reduce la fragmentación, que era lo más criticado del sistema FAT.

Se emplea un registro transaccional (journal) para garantizar la integridad del sistema de ficheros (pero no la de cada archivo). Los sistemas que emplean NTFS han demostrado tener una estabilidad mejorada, que resultaba un requisito ineludible considerando la naturaleza inestable de las versiones más antiguas de Windows NT.

Sin embargo, a pesar de lo descrito anteriormente, este sistema de archivos posee un funcionamiento prácticamente secreto, ya que Microsoft no ha liberado su código, como hizo con FAT.

Gracias a la ingeniería inversa, aplicada sobre el sistema de archivos, se desarrollaron controladores como el NTFS-3G que actualmente proveen a sistemas operativos GNU/Linux, Solaris, MacOS X o BSD, entre otros, de soporte completo de lectura y escritura en particiones NTFS.

Sistemas EXT (Linux)

Actualmente se utiliza la cuarta versión de este sistema de ficheros utilizado por Linux. ext4 (fourth extended filesystem o «cuarto sistema de archivos extendido») es un sistema de archivos transaccional (en inglés journaling), anunciado el 10 de octubre de 2006 por Andrew Morton, como una mejora compatible de ext3. El 25 de diciembre de 2008 se publicó el kernel Linux 2.6.28, que elimina ya la etiqueta de "experimental" de código de ext4. Utiliza un árbol binario balanceado (árbol AVL) e incorpora el asignador de bloques de disco Orlov.

Sistemas HFS (Mac OS)

HFS Plus o HFS+ es un sistema de archivos desarrollado por Apple Inc. para reemplazar al HFS (Sistema jerárquico de archivos). También es el formato usado por el iPod al ser formateado desde un Mac.

Los volúmenes de HFS+ están divididos en sectores (bloques lógicos en HFS), de 512 Bytes. Estos sectores están agrupados juntos en un bloque de asignación que contiene uno o más sectores; el número de bloques de asignación depende del tamaño total del volumen. HFS+ usa un valor de dirección para los bloques de asignación mayor que HFS, 32 bit frente a 16 bit de HFS; lo que significa que puede acceder a 232 bloques de asignación. Típicamente un

volumen HFS+ esta embebido en un Envoltorio HFS (HFS Wrapper), aunque esto es menos relevante. El envoltorio fue diseñado para dos propósitos; permitir a los ordenadores Macintosh HFS+ sin soporte para HFS+, arrancar los volúmenes HFS+ y ayudar a los usuarios a realizar la transición a HFS+. HFS+ arrancaba con un volumen de ficheros de solo lectura llamado Where_have_all_my_files_gone?, que explicaba a los usuarios con versiones del Mac OS sin HFS+, que el volumen requiere un sistema con soporte para HFS+. El volumen origina HFS contiene una firma y un desplazamiento en los volúmenes HFS + embebidos en su cabecera del volumen. Todos los bloques de asignación en el volumen HFS que contienen el volumen embebido son mapeados fuera del archivo de asignación HFS como bloques dañados

Gestión de directorios

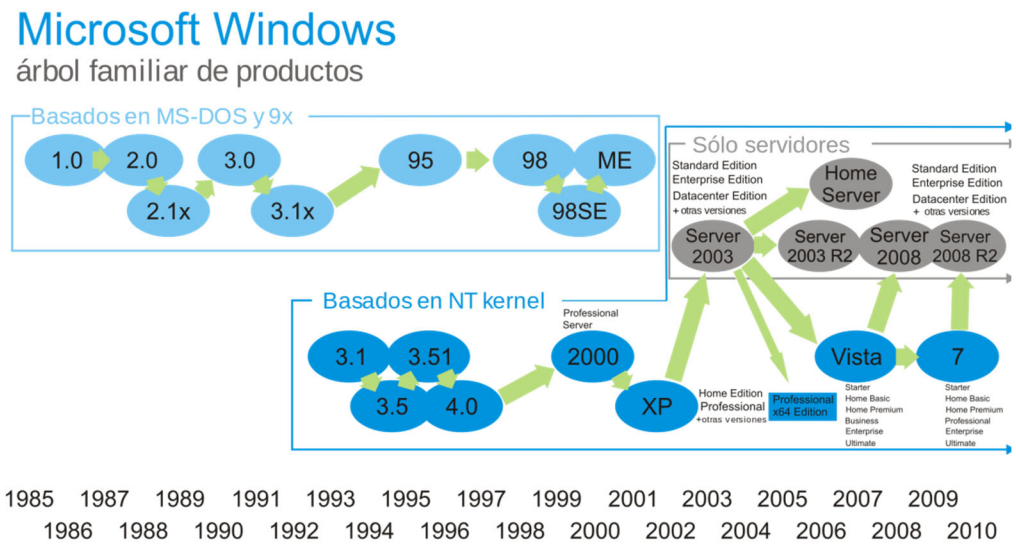
La segunda abstracción que utiliza el sistema operativo para gestionar volúmenes de datos es la de directorio. Los directorios son conjuntos de archivos agrupados, siguiendo algún criterio: directorio del usuario que lo crea, directorio de cartas, directorio de aplicaciones, etc. La estructura global del sistema de archivos suele organizarse en forma de árbol en el que los nodos interiores son directorios, o archivos, y los nodos exteriores son archivos. De un directorio pueden depender archivos u otros directorios, también denominados subdirectorios.

Un directorio se gestiona con una tabla de índices, que contiene un elemento por cada archivo o directorio dependiente de él. Cada elemento está formado por el nombre del archivo dado por el usuario, y por información adicional utilizada por el sistema operativo. La información adicional sobre el archivo puede estar constituida por los atributos y el bloque donde comienza el archivo, caso de MS-DOS. También, la información adicional puede ser un puntero a otra estructura con información sobre el archivo. Este es el caso de UNIX, en el que el puntero sencillamente es la dirección del i-nodo del archivo, que contiene tanto los atributos del archivo como la tabla de posiciones.

Cuando se abre un archivo, el sistema operativo utiliza el camino (“*path*”) para buscar en el directorio el elemento correspondiente al archivo, identificándolo por su nombre. La información se extrae a partir del elemento de la tabla de direcciones del disco y se ubica en la memoria principal. Con ayuda de esta tabla rápidamente pueden realizarse todas las referencias al archivo.

9.2. Sistemas Operativos de Microsoft

La siguiente imagen obtenida de WikiPedia, muestra gráficamente el conjunto de sistemas operativos desarrollados por Microsoft a lo largo de la historia, y las distintas familias a las que cada uno de ellos pertenece:



Sistema Operativo MS-DOS

El nombre MS-DOS corresponde a las siglas de MicroSoft Disk Operating System, es decir, Sistema Operativo de Disco de MicroSoft. Como se muestra en la figura 2.3, la primera versión apareció en 1981, como sistema operativo utilizado para los ordenadores personales de IBM, que en un principio incorporó el micro 8086 de Intel. La versión instalada por IBM se denominaba PC-DOS, mientras que la distribuida por MicroSoft para el resto de fabricantes se denominaba MS-DOS.

A partir de 1981 aparecieron sucesivas versiones y actualizaciones del sistema operativo hasta la versión 6.0 en 1993. En casi todos los casos las nuevas versiones trataban de aprovechar al máximo las prestaciones de los sucesivos microprocesadores de la familia del 8086 de Intel. La versión 1.25 permitía la utilización de nuevos disquetes de doble densidad de 320 KB. La versión 2.0 apareció para el PC-XT,

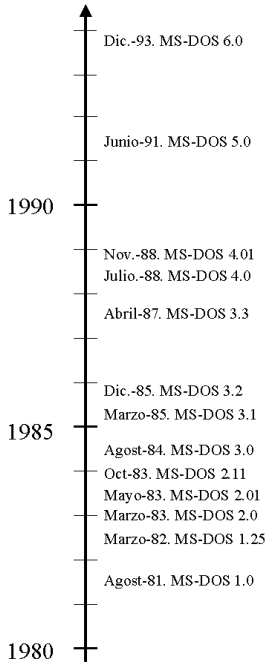
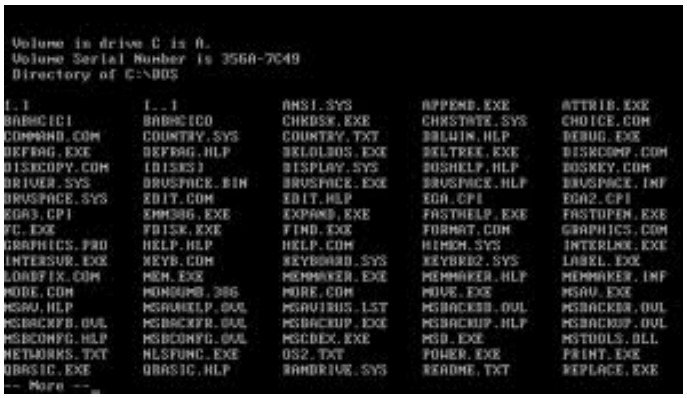


Figura 2. 3 . Evolución histórica de MS-DOS

utilizaba un sistema de archivos jerárquico. Las sucesivas actualizaciones 2.01 y 2.11 eliminaban algunos errores que aparecieron. La versión 3.0 fue una de las más difundidas utilizándose disquetes de 1.2 MB y discos duros de mayor capacidad. Las sucesivas actualizaciones hasta la 3.3, permitieron la utilización de disquetes de mayor capacidad hasta los actuales de 3,5" de 1.5 MB, e incorporaban algunos protocolos para la conexión en red de los ordenadores. En la versión 4.0 se implementó la gestión de memoria EMS, pero aparecieron numerosos errores que limitaron su difusión. En la versión 5.0 se mejoró significativamente la gestión de la memoria, y la versión 6.0 permitía la utilización de mecanismo para la compresión de información en el disco duro, como el "*DoubleSpace*"



Funciones y componentes del DOS

Las principales funciones de MS-DOS son organizar los datos en los discos y la gestión de los diferentes periféricos del PC, prueba de ello es que la mayoría de las órdenes de DOS están relacionadas con el almacenamiento y gestión de los datos en el disco. Para lograr este propósito, el DOS utiliza una estructura arborescente basada en tres elementos fundamentales: unidades, directorios y ficheros. Se dice que la estructura es arborescente porque adopta la forma de un árbol invertido, con las raíces en la parte superior y las hojas en la parte inferior. Las raíces del árbol corresponderían a la unidad. El tronco del árbol sería el directorio principal, que se denomina directorio raíz; y a partir de ese tronco nacen varias ramas, llamadas directorios, de mayor o menor tamaño, de las cuales a su vez cuelgan los ficheros.

El sistema operativo MS-DOS se compone de tres módulo fundamentales, que son responsables de determinadas zonas funcionales de este sistema. Las tres partes más importantes son el BIOS del DOS, no confundir con el ROM-BIOS; el núcleo ("*kernel*") del DOS; y el procesador de comandos.

El BIOS del DOS

El BIOS ("*Basic Input Output System*", Sistema Básico de Entradas y Salidas) del DOS se encuentra en un archivo, que en el PC-DOS de IBM se llama IBMIO.SYS; y que en MS-DOS se llama IO.SYS. Este archivo se encuentra en todos los PC como primer archivo en el directorio raíz del disco de arranque, y está equipado con los atributos de archivo HIDDEN y SYSTEM, para que no aparezca en la pantalla con la llamada del comando DIR.

Cuando DOS quiere comunicarse con alguno de estos dispositivos utiliza los controladores de dispositivos contenidos en ese módulo, que a su vez emplean las rutinas del ROM-BIOS. Sólo el BIOS del DOS y los controladores de dispositivos entran en contacto con el hardware, esto hace que estos sean los elementos más dependientes del *hardware*. En las primeras versiones esta parte del DOS debía ser adaptada por los diferentes fabricantes de hardware, debido a que las diferencias entre los PC de diferentes fabricantes eran significativas.

El núcleo del DOS

El núcleo del DOS, también conocido por su denominación inglesa *Kernel*, se encuentra en el archivo IBMDOS.SYS en el PC-DOS, o en el archivo MSDOS.SYS para MS-DOS. Se encuentra inmediatamente después del archivo IMBIO.SYS, o IO.SYS, en el directorio raíz de la unidad de arranque. Al igual que el anterior archivo no es visible por el usuario, ya que lleva los atributos de archivo SYSTEM y HIDDEN; además, al igual que su predecesor, no se puede borrar porque además se marcó con el atributo READ-ONLY.

En este archivo se encuentra las innumerables funciones del DOS-API, que se pueden alcanzar a través de la interrupción 21h. Todas las rutinas están diseñadas de forma independiente del hardware, y se utilizan para el acceso de otros dispositivos del BIOS del DOS. Por esta razón, este módulo no se ha de adaptar al hardware de los diferentes ordenadores.

El procesador de comandos

Al contrario que en el caso de los dos módulos presentados hasta ahora, el procesador de comandos del DOS, denominado SHELL, se encuentra en un archivo visible con el nombre COMMAND.COM. Este es el programa que se llama automáticamente durante el arranque del ordenador. Muestra el "*prompt"* (símbolo de la línea de comandos) del DOS en la pantalla (A> o C>). Su función principal es procesar las sucesivas entradas del usuario.

El procesador de comandos es el único componente visible del DOS, de modo que en algunos casos es erróneamente considerado como el sistema operativo en sí. En realidad, sólo un programa normal, que funciona bajo el control del DOS. El procesador de comandos no es sin embargo un bloque monolítico, sino que se encuentra dividido en tres módulos, una parte **residente**, una parte **no residente**, y la rutina de **inicialización**.

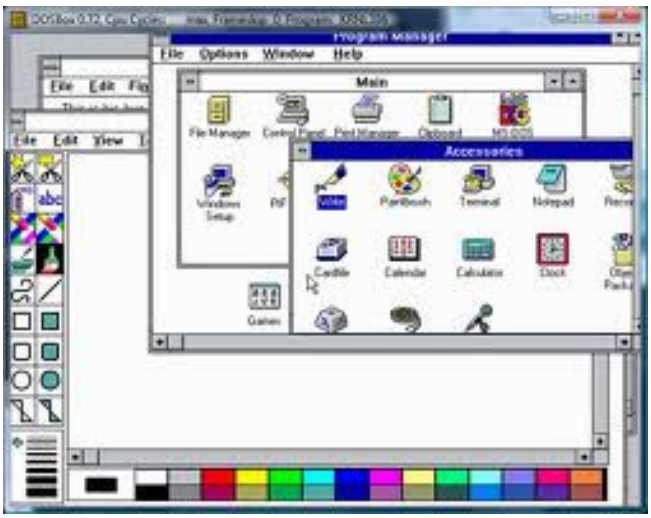
La parte residente, es decir, la parte que se queda constantemente en la memoria del ordenador, contiene diferentes controladores de interrupciones del DOS; por tanto, esta parte se ha de quedar siempre en memoria, ya que de lo contrario, el sistema se quedaría bloqueado, como muy tarde durante la llamada de la siguiente interrupción del hardware, cuyos controladores de interrupciones se encuentran en esta parte.

La parte no residente contiene el código del programa para la salida del indicador del sistema, para leer las entradas de usuario del teclado y para su ejecución. El nombre de este módulo se debe a que puede ser sobrescrito por los programas de usuario, ya que, se encuentra en la parte superior de la memoria, por encima de estos programas. Pero esto no importa, ya que tras la ejecución de un programa se vuelve a pasar a la parte residente. Este comprueba si la parte no-residente fue sobrescrita, y la vuelve a cargar de la unidad de arranque, si fuera necesario.

La parte de inicialización se carga durante el arranque del computador, y acoge las tareas de inicio el sistema DOS. Cuando finaliza su trabajo, ya no es necesaria, y se puede sobreescribir por otro programa.

El primer entorno gráfico Windows

Las primeras versiones de Windows de Microsoft deben ser consideradas como un entorno gráfico; ya que, inicialmente requería del sistema operativo para su funcionamiento. La utilización del entorno de Windows presentaba grandes diferencias respecto al DOS, de las cuales la más inmediata es que se trata de un entorno exclusivamente gráfico, con una filosofía de símbolos e iconos para facilitar la labor del usuario y la necesidad de un ratón o dispositivo análogo. A pesar de que el entorno Windows funciona bajo DOS, existen programas que funcionan bajo DOS, y programas que sólo funcionan bajo el entorno Windows.



Cuando aparecieron las primeras versiones de Windows hacia el año 1986, tanto desde el punto de vista del usuario, como del programador aparecieron unas prestaciones muy limitadas. En esencia, se trataba de una capa que se establecía encima del DOS y cuyas únicas ventajas eran su sistema gráfico; la independencia de los dispositivos, es decir, del hardware; y una multitarea cooperativa entre las escasas aplicaciones que existían. En lo referente a la presentación, se puede decir que aquella primera versión era muy simple.

En pocos años Microsoft lanzó una nueva saga de revisiones bajo la versión 2.0, con un lavado de cara y una mayor eficacia en el código, pero es en la versión 3.0 donde el impacto de Windows rompe todas las previsiones. Por primera vez, Windows aprovechaba por completo de la arquitectura 80386 para construir un sistema de memoria virtual de forma que ésta podía extenderse hasta cuatro veces, siempre y cuando existiese suficiente espacio en el disco duro. Las ventas iniciales de Windows 3.0 fueron espectaculares.

Generalidades sobre Windows

El nombre de Windows es el propio de un sistema que está basado en ventanas. Cada ventana tiene unos atributos particulares, entro los que destaca su posición y tamaño. Mediante el ratón se puede alterar la mayoría de ellas ajustándolas a los propósitos necesarios. También es posible interaccionar con el teclado, pero se parte de la idea de que un usuario de Windows dispone de algún tipo de dispositivo apuntador, tal como un ratón, si quiere alcanzar un mínimo de prestaciones.

Una ventana completa está constituida por un trozo de pantalla, habitualmente con un título, en el que aparece en la parte superior izquierda un menú general con forma de botón, que se despliega al pulsarlo. En la derecha existen otros botones que permiten actuar sobre el tamaño de la ventana. Estos últimos tienen la posibilidad de maximizar, la ventana ocupará toda la pantalla; o minimizar la ventana, pasando a ocupar la mínima extensión, convirtiéndose en un icono. Adicionalmente, la ventana puede ser redimensionada pinchando el borde de la misma con el ratón, de forma que al arrastrarlo se modifica su tamaño. Para moverla, se pincha sobre la barra del título y se arrastra la ventana a la posición requerida. Según lo expuesto, el tamaño de las ventanas es muy variable y la información que contienen puede no ser presentable toda a un tiempo. Aquí entra en juego la figura de las barras de desplazamiento, encargadas de mover el contenido de la ventana sobre un espacio de trabajo que se considera más amplio.

En sus primeras versiones Windows admiíae tres modos de ejecución: real, sólo versión 3.0; estándar; y mejorada. En los dos últimos se rompe la barrera de los 640k que tenía antiguamente el DOS, aprovechando toda la memoria que consiga almacenar nuestro ordenador. La diferencia entre el modo estándar y el mejorado se traduce en un diferente aprovechamiento de la memoria. El modo mejorado requiere disponer de al menos un microprocesador 80386 y 2Mbytes de memoria RAM. Las ventajas de este modo se apoyan más que nada en la existencia de un modo virtual de memoria que permite cuadruplicar la cantidad de memoria visible para las aplicaciones, gracias a la utilización del espacio existente en el disco duro. Más adelante, cuando se hable sobre el Panel de Control se verá el lugar preciso donde configurar esta posibilidad.

Con la versión 3.0 de Windows aparecieron una serie de aplicaciones que cambiaron significativamente la actuación sobre un computador, estos elementos que en la actualidad han sido mejorados permitieron una interfaz más amigable con el ordenador. Los elementos que se describen a continuación permiten cierto control sobre la configuración del sistema operativo, y en su momento representaron un gran avance para la gestión del los recursos del computador. Los principales componentes eran:

- **Administrador de programas.** Esta aplicación, aunque se trata de un programa más, tiene la peculiaridad de ser el primero en cargarse en memoria y permanecer siempre residente. Su función principal es permitir la elección y posterior ejecución de otras aplicaciones, de una forma organizada. El administrador de programas permite iniciar fácilmente las aplicaciones y organizar los archivos y las aplicaciones en grupos lógicos.

- **Administrador de archivos**. Como en cualquier sistema operativo existen métodos para gestionar los archivos que contiene el sistema, así como sus unidades de almacenamiento de información. El Administrador de archivos es una herramienta

que puede emplearse para la organización y utilización de archivos y directorios. Las operaciones básicas que se pueden realizar sobre los disquetes y discos duros son hacer un duplicado del disco, formatearlo, y darle un nombre de volumen.

- **Administrador de impresión**. La razón es que su función principal es meramente física. Se encarga de proporcionar una cola de impresión para diferentes trabajos enviados por una o varias aplicaciones. El Administrador de impresión proporciona un modo de ver y controlar la impresión de los documentos.

- El **portapapeles** es un lugar donde se guarda temporalmente una determinada información. Dicha información puede ser un texto, una imagen o incluso formatos particulares de determinadas aplicaciones. La utilidad de este sistema es muy amplia, siendo su principal función la de intercambiar datos entre aplicaciones o bien copiar determinada información dentro de una misma aplicación. En general, en la mayoría de las aplicaciones existe un menú que tiene el nombre de "Edición" dentro del cual aparecen al menos tres opciones: cortar, pegar y copiar.

- **El panel de control e**s un programa que incluye una serie de componentes que sirven para ajustar la configuración del sistema. El panel de control es una aplicación Windows que permite modificar de forma visual las características del sistema durante la utilización de Windows. Cada una de las opciones que pueden modificarse aparece representada con el icono correspondiente en la ventada del Panel de control. Las opciones existentes son las siguientes: los colores de la pantalla, otras opciones del Escritorio que determinan el aspecto de la pantalla, las fuentes que reconocerán las aplicaciones de Windows, las impresoras utilizadas, los parámetros del teclado y del dispositivo apuntador (ratón), las opciones internacionales, puertos y redes, la fecha y hora del sistema, los sonidos que utilizará el sistema, los parámetros MIDI que utilizará el sintetizador conectado a la computadora, y finalmente las opciones de multitarea para la ejecución de Windows en el modo extendido del 80386.

Protocolos propios de Windows

Protocolo DDE ("*Dynamic Data Exchange*", Intercambio Dinámico de Datos)

El protocolo DDE está orientado al intercambio dinámico de datos que permite que las aplicaciones de Windows puedan intercambiar información y actuar en base a ella de una forma totalmente cooperativa. Bajo DOS, cada programa se comporta como una isla en medio de un océano, obligando al programador a hacer ingeniosos esfuerzos cuando dos aplicaciones necesitan compartir datos. Aunque su diseño esté orientado a la ejecución de un único programa, es posible obtener multitarea dejando programas residentes en memoria. El problema de la comunicación no parece tal cuando una aplicación genera un archivo de salida que otra puede leer, aunque no se trate de un verdadero intercambio de información, dado

que las aplicaciones difícilmente pueden mantener el control sobre en qué momento se debe enviar o recibir la información. Para que dos aplicaciones puedan acceder a áreas de memoria comunes normalmente se utilizan interrupciones que son instaladas a modo de interfaces. Las aplicaciones que intervienen tienen que estar de acuerdo en la forma de acceder a las interrupciones y en el formato de los datos sin que el DOS provea de ningún mecanismo práctico para llevar a cabo la tarea.

En un entorno multitarea como en Windows resulta imprescindible disponer de cierto mecanismos que permitan una comunicación fluida entre aplicaciones. Existen tres métodos que permiten compartir la información. El primer método es el del portapapeles, que dispone de un espacio de datos donde podemos almacenar información. Cuando una aplicación introduce información en el portapapeles todas las demás son informadas de ello, pudiendo acceder a la misma. Este método es utilizado para transferir un bloque de datos, pero no se trata de un protocolo en sí, pues normalmente se considera necesaria la intervención del usuario para decidir en qué momento se realiza la acción. El segundo método requiere la presencia de las librerías de enlace dinámico. El tercer método, mucho más eficaz, es el protocolo DDE. Para la utilización de este protocolo, es imprescindible que haya sido previsto en el diseño de las aplicaciones. Algunas de ellas, como la mayoría de las que son distribuidas con Windows, no disponen de esta capacidad, como puede ser el Write, PainBrush, Portapapeles, etc. Tal vez, la opción más avanzada e interesante del DDE es el enlace permanente de datos, que consiste en mantener una conversación entre dos aplicaciones de forma que determinado conjunto de datos es actualizado en tiempo real.

Protocolo OLE (“*Object Linked and Embedded*” Objeto Unido y Embebido)

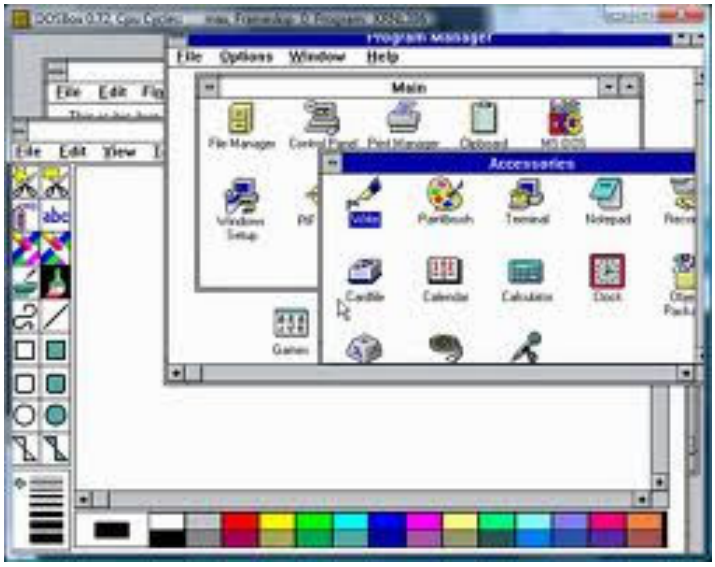
Ante la gran importancia del intercambio de información entre aplicaciones, se desarrolló un nuevo protocolo denominado OLE, que hizo su aparición en la versión 3.10. El OLE, en líneas generales, establece un método estándar para que distintas aplicaciones puedan enlazar y compartir sus documentos. Se tienen dos conceptos nuevos: el de objeto como conjunto de datos; y el de documento, o contenedor, donde se pueden almacenar los objetos Un objeto, por ejemplo un dibujo, tiene un único propietario que es la aplicación que lo creó, pero puede tener múltiples destinos, es decir, ser utilizado por distintas aplicaciones.

El protocolo OLE está basado en mensajes que establece una conversación entre cliente y servidor, informando acerca de los objetos: aplicación origen, formatos, cambios en los mismos, solicitud de actualizaciones, etc. OLE implica ciertos cambios en la mayoría de los programas, con el fin de que lo puedan gestionar y dejen de considerar como exclusiva la información que manejas. A nivel de usuario OLE utiliza comandos a los que ya se está

acostumbrado, tales como copiar, pegar, cortar, etc., que aparecen normalmente en el menú de edición.

Sistema Operativo: Windows 95.

Windows 95 supuso un paso muy importante en la filosofía de trabajo de los PCs. Se integra dentro de la familia de productos Windows de Microsoft, convirtiéndose en el heredero de Windows 3.1, y compartiendo características de Windows para Trabajo en Grupo, y Windows NT. Windows 95 es un entorno de trabajo de 32 bits con una interfaz orientada a objetos, funciones para trabajo en red ayuda interactiva, gestión de módem y fax. Soportó las últimas tecnologías utilizadas hasta la fecha, como OLE 2.0, "Plug and Play" (pinchar y ejecutar), llamadas a procesos remotos, telefonía móvil, etc.



En primer lugar, hay que señalar que Windows 95 es por sí mismo un sistema operativo, a diferencia de Windows 3.1 que necesita trabajar sobre el MS-DOS. Esto quiere decir que el usuario cuando arranca el ordenador no tiene primero que cargar el DOS y, luego, ejecutar la entonces famosa orden WIN para cargar Windows. Ahora cuando se enciende el ordenador se arranca directamente en Windows 95. Por otra parte, el usuario ya no tiene que instalar primero MS-DOS y luego Windows, sino sólo instalar Windows 95 directamente.

En segundo lugar, es necesario destacar que Windows 95 es un sistema operativo de 32 bits, optimizado para los ordenadores que, en su mayoría, usan arquitectura de 32 bits. Las ventajas de trabajar con arquitectura de 32 bits y en modo protegido son bastante grandes: los programadores pueden usar un direccionamiento de memoria lineal y olvidarse de las limitaciones que impone las direcciones segmentadas. Además, se pueden ejecutar controladores de dispositivo en modo protegido, lo que permite virtualizarlos; es decir, que un mismo dispositivo pueda ser compartido por varios programas a la vez, lo cual es un

requisito imprescindible para proporcionar verdadera multitarea. Los controladores virtuales soportan todos los dispositivos hardware del ordenador, incluyendo disco, teclado, monitor, puertos paralelo y serie, etc. También hay controladores virtuales para el ratón, SmartDrive, sistema de ficheros VFAT, sistema de fichero CD-ROM, compresión de disco DriveSpace (y DoubleSpace), tarjetas y protocolos de red, dispositivos SCSI, etc.

Windows 95 es un sistema operativo multitarea, ya que permite ejecutar múltiples aplicaciones simultáneamente mejor que lo hacía Windows 3.1. Ahora bien, la eficacia de la multitarea depende en gran medida del tipo de aplicaciones que se están ejecutando. Si se trata de aplicaciones para Windows 3.1, que son aplicaciones de 16 bits, la multitarea es parecida a la que se ofrece en Windows 3.1. Pero con las aplicaciones Windows de 32 bits se permite alcanzar multitarea completa.

Plug and Play (Conectar y Ejecutar)

Una de las características más atractivas de Windows 95 es que se trata del primer sistema operativo Plug and Play. El estándar o conjunto de normas Plug and Play pretende crear una estructura que permita gestionar de forma inteligente la instalación y configuración de nuevos dispositivos, sin requerir la intervención del usuario. Con un sistema Plug and Play el usuario puede añadir y quitar dispositivos o conectarse y desconectarse a una red o estación maestra sin necesidad de reinicializar el sistema y definir varios parámetros. El sistema Plug and Play detecta automáticamente la existencia de un nuevo dispositivo, determina la configuración óptima y permite que las aplicaciones se autoajusten automáticamente teniendo en cuenta los cambios ocurridos. Por ejemplo, una aplicación que muestra difuminadas, o inactivas, las opciones para CD-ROM y que reconoce inmediatamente la presencia de una unidad CD-ROM, activando la posibilidad de seleccionar las opciones para CD-ROM. De esta forma, los usuarios ya no necesitan conmutar puentes, activar interruptores, seleccionar IRQs libres; ni siquiera tiene que cambiar los ficheros de configuración del sistema operativo par añadir los nuevos controladores.

Para lograr este objetivo de una instalación instantánea y automática, un ordenador ha de poseer tres componentes: un sistema operativo Plug and Play, una BIOS Plug and Play y, por supuesto, dispositivos compatibles Plug and Play. Estos tres componentes son imprescindibles para evitar totalmente la intervención del usuario durante el proceso de instalación, pero la existencia de dos o uno de ellos también simplifica la configuración hardware.

En primer lugar, los dispositivos Plug and Play tienen que ser capaces de identificarse a sí mismos, es decir, informar al ordenador de qué tipo de dispositivo se trata y de los requisitos que necesita. Por ejemplo, una tarjeta de sonido Plug and Play debe indicar al ordenador que se trata de una tarjeta de sonido y que necesita seleccionar una IRQ, un canal DMA y una dirección de memoria base. Además, debe permitir que el ordenador configure de forma automática dichos requisitos, lo cual implica que los dispositivos Plug and Play tienen que poder modificarse a sí mismos. Por ejemplo, en el caso de una impresora que exige la presencia de un puerto paralelo bidireccional que permita enviar datos desde la impresora al ordenador, y no sólo desde el ordenador a al impresora.

En segundo lugar, tiene que haber una BIOS Plug and Play que esté preparada para pedir y aceptar las características y los requisitos de los nuevos dispositivos. Esta BIOS detectará la presencia de un nuevo dispositivo y, en vez de generar varios pitidos seguidos de un escueto mensaje, pedirá los valores por defecto para los requisitos del dispositivo y seleccionará automáticamente los adecuados.

Por último, es necesario un sistema operativo Plug and Play, tal como Windows 95, que gestione todos los componentes cargando los controladores de dispositivo adecuados y reconfigurando el sistema automáticamente sin intervención del usuario.

El estándar Plug and Play no sólo pretende alcanzar una configuración automática de los dispositivos hardware, sino ser capaz de reconocer cambios dinámicos de configuración. Esta característica es fundamental en la informática móvil, puesto que los usuarios de portátiles necesitan poder conectarse a redes locales sin tener que apagar el ordenador o reconfigurarlo. Un sistema operativo Plug and Play, reconoce inmediatamente los nuevos dispositivos instalados y los recursos que necesitan, cargando de forma automática los controladores de dispositivos adecuados. Las aplicaciones son informadas de los cambios dinámicos para que puedan aprovechar las nuevas características, o bien impidan el acceso a dispositivos no existentes. Supuestamente ya no es preciso apagar y reinicializar el ordenador cuando realicen cambios en la configuración hardware, y ni siquiera han de intervenir en el proceso de configuración.

Interfaz orientada a objetos

Una de las principales aportaciones de Windows 95 era su nueva interfaz que presentaba una organización más racional y eficiente de los recursos, aunque supone un cambio radical frente a la filosofía tradicional de Windows. La interfaz de Windows 3.1 está orientada a iconos y su funcionamiento básico es seleccionar un icono y aplicarle una propiedad eligiendo una opción de un menú general que es siempre igual. Por el contrario, la interfaz de Windows 95 está orientada a objetos y ya no existe un menú general para todos los iconos, sino que cada objeto posee su propio menú dependiendo del tipo de objeto del que se trate. En Windows 95 desaparecen el Administrador de Programas y el Administrador de Archivos y en su lugar se implementa un escritorio gráfico. En este escritorio el usuario puede situar objetos de distintos tipos, ya sean programas carpetas de programas, impresoras, unidades de disco, etc.

La interfaz de Windows 95 adopta la tecnología orientada a objetos; es decir, el usuario siempre realiza las mismas operaciones sobre los objetos, pero éstas operaciones se interpretan de distinta forma según el tipo de objeto. Por ejemplo, al hacer doble clic sobre un icono se abre dicho icono. La acción de abrir el icono es distinta según el tipo de objeto: si es un icono de programa, se ejecuta el programa; si es un icono de documento se ejecuta el programa en el que se creó el documento y se carga automáticamente para su revisión o modificación; si es el icono de una unidad de disco, se muestra una ventana con el contenido de la unidad; si es el icono de una impresora aparece la lista de tareas a imprimir, etc.

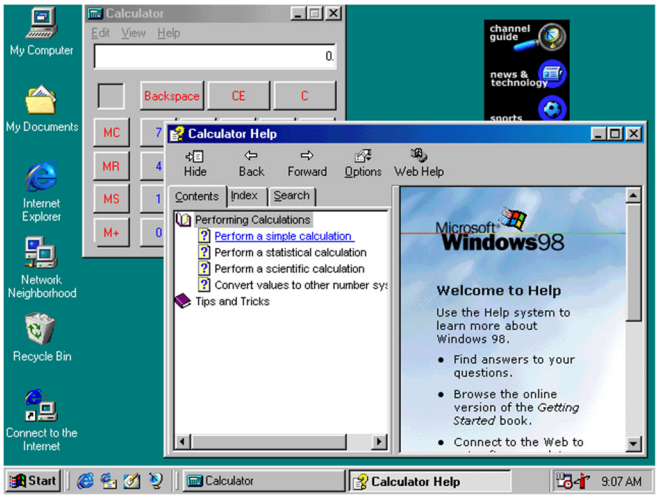
Otro ejemplo donde se demuestra esta metodología orienta a objetos es en el denominado Menú de Contexto, un menú emergente que aparece al pulsar el botón secundario del ratón. El botón secundario es el botón derecho del ratón en los usuarios diestros. El Menú de Contexto posee distintas opciones según el tipo de objeto, pero siempre incluye las operaciones más utilizadas con el objeto, como: eliminar, cortar, copiar y pegar, abrir, etc.

Sin duda alguna, la característica más importante del Menú de Contexto es una opción denominada Propiedades que permite configurar adecuadamente el objeto. Finalmente, otro aspecto importante referente a los objetos es la posibilidad de arrastrar un icono sobre otro, acción que se interpreta según el tipo de objeto que se está arrastrando y el objeto de destino.

En la parte inferior de la pantalla existe una Barra de Tareas que cumple un papel fundamental en la nueva interfaz además de sustituir a la Lista de Tareas de Windows 3.1. Cada vez que se ejecuta una aplicación, o se minimiza, aparece un icono en la Barra de Tareas que representa dicha aplicación. Puesto que la Barra de Tareas siempre está activa, pulsando el icono se accede inmediatamente a la aplicación. Esto resuelve uno de los principales problemas que tienen los usuarios de Windows 3.1: saber exactamente el número de ventanas que tiene abiertas con aplicaciones ejecutándose y saber cómo acceder a una ventana que no aparece en pantalla.

Sistema Operativo: Windows 98

Windows 98 surge de la continua mejora de Windows 95. Se considera que existen dos grandes versiones de Windows 95, la versión de Windows 95 original y la versión Windows 95 OSR2, aparecida en 1996 y que sólo se puede conseguir comprando un ordenador nuevo. Entre estas dos versiones hay grandes diferencias; por ejemplo, Windows 95 OSR2 soporta FAT32, mientras que Windows 95 original no lo hace. Además de esta primera diferenciación, hay usuarios que trabajan con Windows 95 y otros que trabajan con Windows 95 más Internet Explorer 4. La interfaz, las posibilidades de configuración del escritorio y otras características importantes han cambiado con Explorer. Finalmente, Microsoft proporciona nuevos componentes como un Service Pack y varias versiones nuevas de componentes como Exchange, acceso telefónico a redes, soporte USB, etc. En el año 1999 apareció la versión Windows'98 SE que incorporó un nuevo Service Pack, que incorporaba controladores para nuevos periféricos y utilidades del sistema operativo.



La nueva interfaz

Windows 98 utiliza la interfaz de Explorer 4, que es similar a la de Windows 95 pero añadiendo propiedades para la gestión de páginas Web. En realidad lo que ha hecho Microsoft es asociar con cada directorio una página Web, de forma que al examinar el contenido de ese directorio, se muestre la página Web. En la terminología de Microsoft esto se llama vista Web y se puede activar o desactivar a voluntad en cada carpeta.

El escritorio de Windows, puede utilizar como fondo una página Web, permite instalar los componentes activos, pequeños trozos de código HTML que muestran información cambiante, como la cotización de bolsa o las últimas noticias de un determinado tema, y soporta protectores de pantalla de canales. Todas estas características son resultado de la vista Web, pues el escritorio no es más que una carpeta del disco duro y, por tanto, posee la capacidad de entender y mostrar código HTML. Las barras de herramientas incluyen ahora nuevas opciones. Ahora puede incluir en la barra de tareas iconos que ejecutan programas automáticamente de forma rápida y cómoda. Además hay varios iconos predefinidos entre los que destaca uno que minimiza todas las ventanas abiertas, dejando el escritorio limpio.

Pero sin duda la novedad más interesante es la capacidad de trabajar con dos monitores. El funcionamiento es muy sencillo: hay que instalar en el ordenador dos tarjetas de vídeo, que pueden ser 2 PCI O 1 PCI y AGP, ya que no se pueden usar tarjetas de vídeo ISA, y conectar los monitores correspondientes. A partir de ese momento Windows funciona con una pantalla virtual que es la suma de los dos monitores. Por ejemplo, si tiene en un monitor 800 x 600 y en otro 640 x 480, se crea una pantalla virtual de 800 x 1080 donde se puede colocar libremente los objetos. Es decir, puede arrastrar un icono desde una pantalla a otra, pues

para Windows se trata de una sola pantalla virtual. También puede hacer que un programa se ejecute en uno de los monitores, usando el otro para otras tareas.

Ventajas de la FAT32

Una de las mejoras más importantes de Windows 98 es la colección de herramientas para gestión de discos duros. En primer lugar, Windows 98 soporta FAT32, un sistema de archivos que presenta importantes ventajas frente a la FAT 16 tradicional. Todos los archivos se almacenan en unidades de información llamadas clusters, cuyo tamaño varía según el tamaño del disco. La FAT ("*File Allocation Table"*, Tabla de gestión de archivos) es una tabla que contiene la secuencia de clusters de un archivo, es decir, cuál es la cadena de clusters que componen los datos de un archivo. Cuando se abre un archivo, el sistema operativo sigue la FAT para saber en qué clusters del disco están almacenados los datos del archivo. Windows 95, MS- DOS, Windows NT, OS/2, Linux y el resto de sistemas operativos han soportado la FAT16, una tabla que utiliza números de 16 bits para identificar cada cluster. El problema es que con números de 16 bits sólo se puede gestionar un máximo de 2^{16} clusters, es decir, 65.536 clusters, que para el tamaño actual de los discos es demasiado poco.

Herramientas de sistema y hardware

Además del convertidor de FAT32, Windows 98 incluye varias herramientas de sistema como el Liberador de espacio en disco duro, un programa de copias de seguridad, un programa de información del sistema.

Windows 98 está preparado para las últimas tecnologías hardware y especificaciones que estarán presentes de forma masiva en los próximos años. Se soportan las nuevas tarjetas de vídeo AGP ("*Accelerated Graphics Port*", Puerto para Aceleración de Gráficos), que usan un chip de vídeo que aumenta drásticamente la velocidad de procesamiento gráfico al evitar el bus PCI, y utilizar una línea directa entre el subsistema gráfico y la memoria del ordenador. También se reconocen los nuevos módulos de cámaras digitales y los lectores DVD. Windows 98 está preparado para los ordenadores con conectores de tipo USB ("*Universal Serial Bus*", Bus Serie Universal), que permite conectar hasta 127 dispositivos serie a un único conector del PC, como escáner, cámaras digitales, ratones, módems, impresoras, etc. Esta es una forma cómoda de eliminar la necesidad de cables y puertos serie adicionales. En este sentido, Windows 98 soporta el estándar IEEE 1394, cuya función es similar al de USB. En el campo de la administración de energía, Windows 98 sigue el estándar ACPI ("*Advanced Configuration and Power Interface*", Configuración Avanzada e Interfaz de Alimentación), que permite al ordenador activar o desactivar automáticamente periféricos como discos duros, impresoras o tarjetas de red; por ejemplo, desactiva el consumo, y con ello el ruido del disco duro, cuando se producen tiempos de inactividad, activándose automáticamente al usar el teclado o el ratón. Señalar también que Windows 98 soporta IrDA 3.0 ("*Infrared Data Association*",

Asociación de Datos por Infrarrojos), la última versión del estándar para comunicación inalámbrica a través de infrarrojos.

Finalmente Windows Scripting Host permite ejecutar archivos VisualBasic Script o Java Script, tanto desde el entorno de Windows como desde la línea de comandos del MS-DOS. De esta forma, se pueden crear en estos lenguajes verdaderos programas Windows que se ejecutan de forma sencilla y cómoda. A grandes rasgos, es una forma de retomar los habituales archivos por lotes BAT, pero ahora con funciones de Windows y con múltiples posibilidades. El motor de Windows Scripting Host soporta los lenguajes VBScript y Jscript, pero compañías independientes pueden crear controles ActiveX para otros lenguajes como Perl, TCL o REXX.

Comunicaciones

El apartado de las comunicaciones es uno de los más importantes en todo sistema operativo actual. La inclusión de Explorer 4 en Windows 98 supone dotarle de un valor añadido importante, pues añade el navegador Explorer, un programa de correo, y grupos de noticias Outlook Express, un editor de páginas Web llamado Front-Page Express, y una herramienta de telefonía y vídeo conferencia denomianda NetMeeting.

Más novedades en comunicaciones son la nueva versión de Acceso telefónico a redes 1.2, que incluye soporte para multienlace; incluyéndose el protocolo PPTP para crear las denominadas VPN (“Virtual Private Networks”, Redes Privadas Virtuales), que son conexiones seguras a una red desde Internet, actuando de la misma forma que si estuviera conectado a la red. Una de las novedades más importantes de Windows 98, conocida como Web TV, se trata de la posibilidad de recoger páginas Web a través de las señales de TV, algo similar al teletexto.

Sistema Operativo: Windows Millenium

El sistema operativo Windows Millenium fue el sucesor de Windows'98 SE para el entorno de los usuarios denominados *domésticos* según la terminología de Microsoft. Apareciendo en septiembre del 2000. De acuerdo con este fabricante el objetivo es proporcionar un entorno estable, seguro y sencillo al usuario, de modo que no precise de avanzados conocimientos de administración de usuarios, o de administración de redes de ordenadores.

Este nuevo sistema operativo incorpora avances en los dispositivos multimedia, acordes con las prestaciones de los computadores en que se instalan; es decir, Pentium II o III, con elevadas capacidades de almacenamiento en disco y en memoria RAM. La finalidad es ofrecer un entorno amigable al usuario que le permita realizar fácilmente tareas de edición de vídeo, reproducción y/o creación de sonido, acceder a los diferentes servicios de Internet, etc. Ejemplo de estas nuevas prestaciones son las aplicaciones que Microsoft, y otros fabricantes, suministran para Windows Millenium; como por ejemplo: Movie Maker, Windows Media Player,..., para las aplicaciones de multimedia; y Microsoft Explorer 5.5, MSN Messenger y NetMeeting para acceso a los diversos servicios que ofrece Internet.

Las herramientas de administración y gestión del sistema operativo, a las que tiene acceso el usuario, son similares a las de Windows 98, aunque en algunos casos cambian de aspecto. Pocas utilidades han sido añadidas a este sistema operativo, entre ellas destaca la de restauración del sistema, que pretende recuperar al ordenador en caso de quedar bloqueado; el desarrollo de varios asistentes para facilitar la gestión y administración de aplicaciones; y el control de instalación de aplicaciones, que tendría la finalidad de evitar sobreescrituras no deseadas en el proceso de instalación de nuevos programas.

Sistema Operativo: Windows NT

Las características más significativas de Windows NT (New Technology) es que está destinado para ofrecer servicios avanzados de administración al usuario, siendo un verdadero sistema operativo multitarea. Esto se debe, entre otras a las siguientes razones:

- Aumento de la velocidad y capacidad de memoria de los microprocesadores.
- Soporte de memoria virtual.
- Aumento de las aplicaciones cliente/servidor en redes locales.

Todo esto permitió que las aplicaciones fueran más complejas e interrelacionadas, de forma que un usuario puede estar utilizando varias aplicaciones simultáneamente. Microsoft desarrolló Windows NT con una apariencia cara la usuario similar a Windows 3.1, pero basado en un concepto radicalmente distinto. Windows NT explota la potencia de sus microprocesadores contemporáneos de 32 bits y suministra una capacidad completa de multitarea en un entorno monousuario. Además, ofrece una gran potencia al poder ejecutar aplicaciones escritas para otros sistemas operativos y cambiar de plataforma hardware sin modificar el sistema operativo ni las aplicaciones.

Características básicas de Windows-NT

Windows NT tiene una estructura modular, lo que le da una gran flexibilidad. Además, se puede ejecutar en una gran variedad de plataformas y soporta aplicaciones escritas para otros sistemas operativos. Al igual que en otros sistemas operativos, en Windows NT se distingue entre el software orientado a las aplicaciones, que se ejecuta en modo usuario; y el software del sistema operativo, que se ejecuta en modo privilegiado como administrador. Este ultimo tiene acceso a los datos del sistema y al hardware, mientras que el resto de usuarios tiene accesos limitados.

En la figura 9.4 se muestra la estructura de Windows NT. Para conseguir que NT tenga la misma visión del hardware, independientemente del sistema en el que se esté ejecutando, el sistema operativo se divide en cuatro capas:

1. **Capa de abstracción del hardware** (HAL, "*Hardware Abstraction Layer*"). El objetivo de esta capa es que el núcleo vea por igual todo el hardware, independientemente de la plataforma que se use. Para ello realiza la conversión entre las órdenes, y las respuestas, del hardware genérico y el de una plataforma

específica; como por ejemplo, 80486 o Pentium de Intel, PowerPC de Motorola, Alpha AXP de Digital, etc.

2. **Núcleo** ("*kernel*"). Es la parte central del sistema operativo y se encarga de gestionar la planificación y conmutación de contexto, los manipuladores de excepciones y de interrupciones, y la sincronización.

3. **Subsistemas**. En esta capa se incluyen módulos para funciones específicas que hacen uso de los servicios básicos dados por el núcleo. Estos módulos son: el gestor de memoria virtual, el gestor de procesos, el monitor de referencia de seguridad, y módulo de llamadas a los procedimientos locales. Hay un subsistema, el gestor de entrada/salida que evita que el núcleo interaccione directamente con hardware, esto se requiere por razones de eficiencia y, también en parte, por las operaciones de E/S. En dicho gestor se incluye el sistema de archivos, el gestor de caché, los controladores de dispositivos y de red, etc.

4. **Servicios del sistema**. Esta es la última capa y proporciona una interfaz para el software que se ejecuta en modo usuario.

Por encima de estas capas están los **subsistemas protegidos**, que se encargan de la comunicación con el usuario final. Un subsistema protegido proporciona una interfaz de usuario gráfica, la línea de órdenes del usuario. Otra función que tienen asignados es suministrar la interfaz de programación de aplicación (API, "*Application Programming Interface*"). Esto significa que aplicaciones creadas para otros sistemas operativos pueden ejecutarse sobre NT sin ninguna modificación, como es el caso de OS/2, MS-DOS, o Windows.

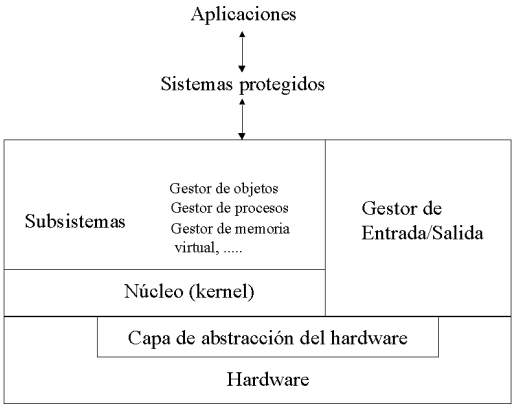


Figura 9. 4 . Estructura interna de Windows NT

Para soportar esta estructura, Windows NT utiliza un **modelo cliente/servidor**. Este tipo de modelo de computación cliente/servidor es normal en los sistemas distribuidos, y se pueden adoptar para uso interno en un único sistema. En este caso, cada servidor se implementa como uno o más procesos, de forma que cada proceso espera que algún cliente realice una petición de alguno de sus servicios. Esta petición se hace enviando un mensaje al servidor que realiza la operación pedida y contesta

mediante otro mensaje. Así pues, el servidor es un programa que se limita a realizar aquellas funciones que le son solicitadas desde el exterior.

Este tipo de arquitectura cliente/servidor simplifica mucho el sistema operativo básico, se pueden construir nuevos API sin ningún conflicto, además de ser una forma natural para el cálculo distribuido. Por otro lado, también se aumenta la fiabilidad, puesto que cada servidor se ejecuta en un proceso separado con su propia partición de memoria y protegido de otros servidores, de forma que si falla no corrompe al resto del sistema.

Otro aspecto importante de Windows NT es su soporte de **hebras (“*threats*”)** dentro de los procesos. Las hebras incorporan algunas de las funcionalidades asociadas tradicionalmente con los procesos. Una hebra es una unidad básica de ejecución, que se puede interrumpir para que el procesador pase a otra hebra. Desde el punto de vista del planificador y del distribuidor, este concepto es equivalente al de proceso en algunos sistemas operativos anteriores. En Windows NT un proceso es un conjunto de una, o más, hebras junto con los recursos del sistema asociados, como son las particiones de memoria, archivos, dispositivos, etc. Esto corresponde al concepto de programa en ejecución.

Otra de las características de Windows NT es la utilización de los conceptos de diseño **orientado a objetos**. Aunque no es un sistema operativo orientado a objetos puro, ya que no está implementado en un lenguaje orientado a objetos. Las estructuras de datos que residen completamente dentro de un ejecutable no se representan como objetos. Además NT no incorpora algunas características que son comunes en los sistemas orientados a objetos, como la herencia y el polimorfismo.

Según la metodología de programación orientada a objetos que W-NT usa están la encapsulación y las clases e instancias de objetos. La encapsulación hace referencia a que los objetos consistan de varios datos, llamados atributos; y uno o más procedimientos que se pueden ejecutar sobre estos datos, llamados servicios. Una clase de objetos es una plantilla en la que se guardan los atributos y servicios de un objeto y se definen ciertas características del mismo.

En NT no todas las entidades son objetos. De hecho, los objetos se emplean en los casos donde los datos están abiertos para acceso en modo usuario o cuando el acceso a los datos es compartido o restringido. Entre las entidades representadas por objetos están los procesos, las hebras, los archivos, los semáforos, los relojes y las ventanas. NT maneja todos los tipos de objetos de una forma similar, mediante el gestor de objetos, cuya responsabilidad es crear y eliminar los objetos según la petición de las aplicaciones y otorgar acceso a servicios de objetos y datos. Un objeto, ya sea un proceso o una hebra, puede referenciar a otro objeto abriendo un gestor del mismo.

En NT los objetos pueden estar etiquetados, o no. Cuando un proceso crea un objeto no etiquetado, el gestor de objetos devuelve un gestor al objeto, y la única forma de referenciarlo es mediante su gestor. Los objetos etiquetados tienen un nombre que puede usarse por otro proceso para obtener el gestor del objeto. Los objetos etiquetados tienen asociada la información de seguridad en la forma de un testigo de acceso. Esta información se puede utilizar para limitar el acceso al objeto. Por ejemplo, un proceso puede crear un objeto semáforo etiquetado de forma que sólo los procesos que los conozcan lo puedan utilizar. El testigo de acceso asociado con el semáforo tendrá una lista de todos los procesos que pueden acceder a él.

Gestión de procesos

El diseño de los procesos Windows NT está marcado por la necesidad de proporcionar soporte para los diferentes entornos de sistemas operativos. Como cada SO tiene su forma particular de nombrar y representar a los procesos, de proteger los recursos de los mismos, de efectuar la comunicación entre procesos y la sincronización, etc. La estructura de los procesos y los servicios proporcionados por el núcleo de NT son relativamente sencillos, y de propósito general. Esto permite que cada subsistema del sistema operativo emule una funcionalidad y estructura de los procesos particular.

Entre las características de los procesos de NT se pueden resaltar:

- Los procesos en NT son implementados como objetos
- Un proceso ejecutable puede tener una o más hebras.
- Tanto los objetos de los procesos como los da las hebras llevan incorporadas las capacidades de sincronización.
- El núcleo de NT no mantiene ninguna relación con los procesos que crea, incluidas las relaciones entre procesos padre y procesos hijos.

La concurrencia entre los procesos se consigue porque hebras de diferentes procesos se pueden ejecutar concurrentemente. Además, si se dispone de varias CPUs, se pueden asignar distintos procesos a hebras del mismo proceso, de forma que se ejecuten concurrentemente. Las hebras del mismo proceso pueden intercambiar información entre ellas a través de la memoria compartida y tener acceso a los recursos compartidos del proceso.

Dentro de la arquitectura de objetos de Windows NT se proporcionan mecanismos de sincronización entre las hebras. Para implementar los servicios de sincronización se tiene

la familia de objetos de sincronización, que son entre otros: procesos, hebras, archivos, sucesos semáforos y relojes. Los tres primeros objetos tienen otros usos, pero también se pueden utilizar para sincronización, el resto de los tipos de objetos se diseñan específicamente para soportar la sincronización.

Gestión de la memoria

Windows NT se diseñó para implementarlo en diferentes tipos de procesadores, y en uno en los que más se pensó fue en el Intel 80486. Por ello, en Windows NT se adopta el tamaño de página de 4 Kbyte del 80486, como base de su esquema de memoria virtual. Para entender el esquema de memoria de Windows NT es conveniente estudiar los servicios de memoria virtual del 80486 y del 80386.

En estos procesadores de Intel se incluye un hardware especial para soportar la segmentación y la paginación. Aunque estas funciones pueden ser desactivadas, pudiéndose elegir entre las opciones siguientes:

- Memoria ni segmentada, ni paginada
- Memoria paginada pero no segmentada
- Memoria segmentada pero no paginada
- Memoria segmentada y paginada.

Cuando se emplea segmentación, cada dirección virtual, o lógica, se compone de un campo de referencia al segmento de 16 bits, dos de ellos destinados a los mecanismos de protección y catorce para especificar el segmento. Se cuenta además, con un campo de desplazamiento en el segmento de 32 bits; de esta forma, el espacio de memoria virtual que puede ver un usuario es de 2^{46}=64 TB; mientras que el espacio de direcciones físicas que se puede direccionar con 32 bits es de 4 Gb.

Existen dos formas de protección asociadas con cada segmento: niveles de privilegio, y de atributos de acceso. Son cuatro los niveles de privilegio, del 0 al 3, cuando se trata de un segmento de datos corresponde a la clasificación del mismo y si es un segmento de programa es su acreditación. Un programa sólo puede acceder a segmentos de datos para los que el nivel de acreditación es menor o igual que el de clasificación. El uso de estos niveles de privilegio depende del sistema operativo. Generalmente los niveles 0 y 1 corresponden al sistema operativo, el nivel 2 a algunos subsistemas y el nivel 3 a las aplicaciones. Los atributos

de acceso de un segmento de datos indican si el permiso de acceso es de lectura/escritura o de sólo lectura, y para un segmento de programa expresan si es de lectura/ejecución o de sólo lectura.

La dirección virtual se tiene que transformar en una dirección física mediante un mecanismo análogo al explicado anteriormente. La segmentación es una característica opcional que se puede desactivar. Cuando no se usa segmentación los programas emplean direcciones lineales, que corresponden a las direcciones en el espacio de memoria del proceso que utilizan 46 bits, la cual hay que convertir en una dirección real de 32 bits. El mecanismo de paginación utilizado para hacer esto consiste en una operación de búsqueda en una tabla de dos niveles. El primer nivel es un directorio de páginas de 1024 entradas, con lo que se divide el espacio de memoria de 4 Gb en grupos de 1024 páginas, de 4Mb de longitud cada una con su propia tabla de páginas. Cada tabla de páginas contiene 1024 entradas y cada entrada corresponde a una página de 4Kb. El gestor de memoria puede usar un directorio de páginas para todos los procesos, un directorio de páginas para cada proceso o una combinación de ellos. El directorio de páginas de la tarea actual está en memoria principal, pero las tablas de páginas pueden estar en memoria virtual.

Sistema Operativo: Windows 2000

Windows 2000 no es realidad un único sistema operativo, sino que se trata de una familia de sistemas cada uno de ellos con unas prestaciones y, por lo tanto, destinado a un público diferente. La familia está compuesta por cuatro sistemas operativos: *Professional, Server, Advanced Sever y Datacenter Server*.

Fundamentalmente, la versión *Professional* está destinada a usuarios con Windows 95, Windows 98, o Windows NT; mientras que el resto son sistemas destinados a plataformas donde ahora se ubica Windows NT Server o, incluso, en instalaciones donde actualmente se encuentra instalada alguna de las variantes de Unix que migrarán a Windows 2000. Lo más habitual es la utilización de Windows 2000 *Professional* y *Server*, dejando el *Advanced Server* y el *Datacenter Server* para grandes corporaciones.

Windows 2000 es un sistema basado en la tecnología NT, es decir, que siguiendo con una numeración coherente, Windows 2000 en realidad debería haber sido Windows NT 5.0. Desde el punto de vista de la gestión de los equipos, las características de Windows 2000 hacen que una red basada en éste sistema sea más fácil de administrar, facilitando la instalación de equipos clientes y mejorando el soporte de perfiles flotantes y acceso remoto. Respecto a Windows 98 ha heredado la facilidad de uso y el soporte de dispositivos *plug and play*; pero intentando mantener la robustez y fiabilidad de Windows NT.

Otro de los grandes avances que se han producido en Windows 2000 se encuentra en la seguridad a todos los niveles. No sólo se ha mejorado la seguridad de cara al acceso al sistema o protección de archivos, sino que también los controladores son más seguros que en Windows 9x/NT. Esto se debe a que Microsoft **certifica los controladores** de los fabricantes que lo soliciten. Aunque, se pueden instalar controladores sin certificar, si se opta por los certificados, se tiene una mayor certeza de que todo va a funcionar correctamente, evitando en gran manera el bloqueo de ordenador debido a los controladores.

El **encriptado** de archivos forma parte de otra de las características de seguridad interesantes de Windows 2000. Así, si alguien consigue saltarse los sistemas de seguridad y llevarse un archivo que no le corresponde, no podrá leerlo ya que estará encriptado. También se ha de tener en cuenta todas las características que aporta Explorer 5.01 con respecto a Internet, ya que es la versión del navegador que se integra en el propio sistema. A todo lo anterior hay que añadir las mejoras que tiene para los usuarios móviles, es decir, aquellos que utilizan principalmente Windows en un portátil.

Mejoras de Windows 2000.

Windows 2000 *Professional* obtiene un buen rendimiento en la ejecución de aplicaciones de 32 bits, ya que su arquitectura está completamente basada en 32 bits. Soporte hasta 4 GB de memoria RAM y hasta dos procesadores con multiproceso simétrico. Hay muchas placas base que se venden con posibilidad de incorporar dos procesadores. Sin embargo, con Windows 9x en una placa con dos procesadores, siempre uno de los procesadores está parado, ya que no soporta multiproceso simétrico

Para conseguir una buena escalabilidad, la versión *Server* soporta hasta 4 procesadores para multiproceso simétrico, llegando hasta ocho en la versión *Advanced Server*, con hasta 4 GB de memoria RAM, y 8 GB en *Advanced Server*. En el caso de *Advanced Server* se incorpora un equilibrado de cargas entre servidores para conseguir un mayor rendimiento en servidores críticos.

Sistema Operativo: Windows XP

El sistema operativo Windows XP (Windows eXPerience) apareció en el mercado en octubre del 2001. Su principal característica es que proviene de una fusión entre las versiones de Windows 9x &Me y las versiones de Windows NT & 2000; es decir, pretende ser un sistema con la robustez y seguridad de la familia de W-NT, y la facilidad de uso y compatibilidad de W'9x. Como resultado Microsoft ha incorporado en W-XP el núcleo del sistema operativo de W-NT y los mecanismos y herramientas de gestión de W'9x.

Esta fusión de las dos familias de sistemas operativos de Microsoft es una vieja aspiración de este fabricante, que pretende fusionarlos en un solo. Esta no es una tarea fácil y de hecho uno de los principales requisitos de diseño es hacia qué tipo de usuario está destinado el sistema operativo. En el caso de Windows XP, el usuario denominado *doméstico* será el destinatario. Este tipo de perfil limita significativamente el sistema operativo, ya que se supone que la persona que va a trabajar y administrar la máquina, no debe tener problemas al instalar aplicaciones y que todo debería instalarse casi automáticamente, lo cual impide plantearse cuestiones tan elementales como implantar una buena gestión de usuarios, el control de procesos, etc.

Algunas de las principales novedades aparecidas con Windows XP han sido la activación de la licencia del sistema operativo, y la certificación del hardware. La activación de licencia es un intento de lucha contra el pirateo del software, consiste en registrar cada una de las instalaciones que realiza el usuario, de forma que combinando los números series del hardware del computador y de la licencia del sistema operativo, entonces Microsoft devuelve una clave que permite la utilización correcta del ordenador, y en caso contrario se limitaría drásticamente su uso. Evidentemente, surgen numerosas cuestiones que resolver, como son la actualización del hardware del equipo, el uso de equipos diferentes, y la aparición de cracks que se invalidan los mecanismos de activación. El segundo punto citado de certificación del hardware es un intento de estandarización de todos los dispositivos que puedan ser

instalados al computador; aunque la finalidad es interesante, este también es un punto con una gran polémica, ya que, los avances en velocidad y prestaciones del nuevo hardware siempre son superiores a los avances en los estándares; y el otro aspecto es la gestión de la certificación, ya que podría dar lugar a un monopolio muy cuestionable.

Otras novedades incorporadas en W-XP son referentes a la interfaz que se suministra al usuario. Se pueden abrir varias sesiones simultáneamente; se puede cambiar el aspecto del escritorio volviendo al de W'9x; y se pueden definir sesiones en equipos remotos, lo que permitiría la reutilización de equipos obsoletos.

Sistema Operativo: Windows Vista

El proceso de desarrollo terminó el 8 de noviembre de 2006 y en los siguientes tres meses fue entregado a los fabricantes de hardware y software, clientes de negocios y canales de distribución. El 30 de enero de 2007 fue lanzado mundialmente y fue puesto a disposición para ser comprado y descargado desde el sitio web de Microsoft. La aparición de Windows Vista viene más de cinco años después de la introducción de Windows XP, es decir, el tiempo más largo entre dos versiones consecutivas de Microsoft Windows

El Windows Vista es el primer sistema operativo de Microsoft concebido para garantizar una compatibilidad total con EFI (Extensible Firmware Interface), la tecnología llamada a reemplazar a las arcaicas BIOS que desde hace más de dos décadas han formado parte indisoluble de los ordenadores personales, por lo tanto no empleará MBR (Master Boot Record), sino GPT (GUID Partition Table).

Introduce las ventanas dibujadas con gráficos vectoriales usando XAML y DirectX. Para ello, se utilizaría una nueva API llamada Windows Presentation Foundation, cuyo nombre en código es Avalon, que requiere una tarjeta gráfica con aceleración 3D compatible con DirectX.

Agrega una interfaz de línea de comando denominada Windows PowerShell, que finalmente se ofreció como una descarga independiente para Windows Vista y Windows XP SP2.

Se anunció una nueva extensión de base de datos al sistema de archivos llamada WinFS. El desarrollo de dicho sistema de ficheros ha sido abandonado por Microsoft, por lo tanto no será incluido en Windows Vista, por el momento, siendo compensado por un sistema de búsqueda basado en la indexación.

Integra directamente en el sistema un lector de noticias RSS (Really Simple Syndication, por sus siglas en inglés).

En Windows Vista la utilidad de restauración del sistema ha sido actualizada e implementada como herramienta de inicio de sesión, facilitando así el rescate del sistema. Además incluye

un sistema unificado de comunicaciones llamado Windows Comunication Foundation, cuyo nombre en código es Indigo.

Un sistema antispyware denominado Windows Defender, que también se liberó para Windows XP con SP2 (aunque requiere la validación del sistema).

Añade al firewall de sistema la capacidad de bloquear conexiones que salen del sistema sin previa autorización.

Se incluye Windows ReadyBoost, la cual es una tecnología de caché de disco incluida por primera vez en el sistema operativo Windows Vista. Su objetivo es hacer más veloces a aquellos computadores que se ejecutan con el mencionado sistema operativo mediante pendrives, tarjetas SD, compactFlash o similares.

Y se incorpora al sistema la herramienta Encriptador de disco BitLocker, para la protección de datos extraviados en las versiones Enterprise y Ultimate.

Mejora notablemente el sistema de control de cuentas de usuario, que es una característica del sistema que limita las operaciones de determinados tipos de usuarios en el equipo. A diferencia de las anteriores versiones de Windows, los nuevos usuarios de Windows Vista (con cuenta estándar) no tienen derechos de administrador por defecto, como la instalación y la modificación de registros del sistema. Se caracteriza por tener una variante de la bandera de Windows en dos colores, azul claro y amarillo en la esquina inferior derecha de cada botón o archivo de instalación. Para realizar tareas administrativas, el monitor se oscurece, se bloquea cualquier orden del ratón o teclado; y aparece una ventana de confirmación, la cual solo autoriza aceptando la orden o tecleando una contraseña. Solo permite la activación o desactivación de éste.

Incluye un Sync Center para sincronización de Windows Vista con Pocket PC sin necesidad de instalar el Active Sync.

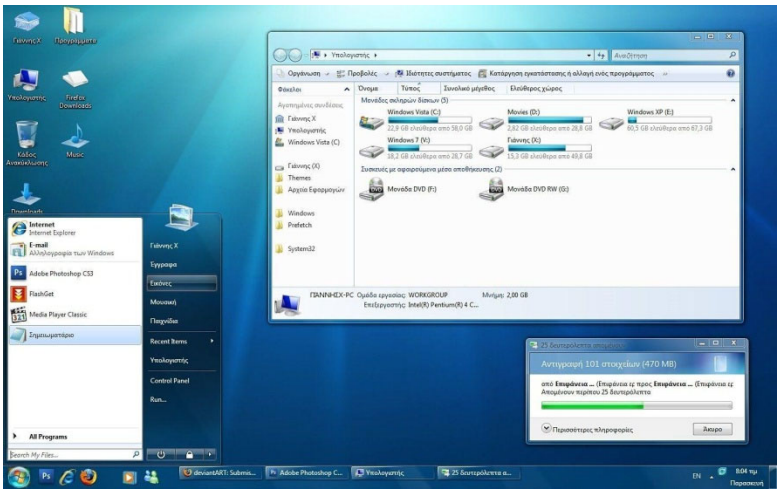
Incorpora un sistema de protección llamado Windows Software Protection Platform (WSPP) que es más potente que el actual Windows Genuine Advantage (WGA). Cuando detecte que la copia es ilegal, lo primero que hará será avisar al usuario y si el usuario no logra obtener una copia auténtica el programa empezará a ir desactivando opciones del sistema, como son el Aero o el Windows Defender hasta únicamente dejar activo lo más básico como es el navegador.

Carga aplicaciones un 15% más rápido que Windows XP gracias a la característica SuperFetch.

Se reduce en un 50% la cantidad de veces que es necesario reiniciar el sistema después de las actualizaciones.

Sistema Operativo: Windows 7

A diferencia del gran salto arquitectónico y de características que sufrió su antecesor Windows Vista con respecto a Windows XP, Windows 7 fue concebido como una actualización incremental y focalizada de Vista y su núcleo NT 6.0, lo que permitió mantener cierto grado de compatibilidad con aplicaciones y hardware en los que éste ya era compatible.4 Sin embargo, entre las metas de desarrollo para Windows 7 se dio importancia a mejorar su interfaz para volverla más accesible al usuario e incluir nuevas características que permitieran hacer tareas de una manera más fácil y rápida, al mismo tiempo que se realizarían esfuerzos para lograr un sistema más ligero, estable y rápido. Diversas presentaciones ofrecidas por la compañía en 2008 se enfocaron en demostrar capacidades multitáctiles, una interfaz rediseñada junto con una nueva barra de tareas y un sistema de redes domésticas simplificado y fácil de usar denominado «Grupo en el hogar», además de importantes mejoras en el rendimiento general del sistema operativo.



Windows 7 incluye varias características nuevas, como mejoras en el reconocimiento de escritura a mano, soporte para discos duros virtuales, rendimiento mejorado en procesadores multinúcleo, mejor rendimiento de arranque, DirectAccess y mejoras en el núcleo. Windows 7 añade soporte para sistemas que utilizan múltiples tarjetas gráficas de proveedores distintos (heterogeneous multi-adapter o multi-GPU), una nueva versión de Windows Media Center y un gadget, y aplicaciones como Paint, Wordpad y la calculadora rediseñadas. Se añadieron varios elementos al Panel de control, como un asistente para calibrar el color de la pantalla, un calibrador de texto ClearType, Solución de problemas, Ubicación y otros sensores, Administrador de credenciales, iconos en el área de notificación, entre otros. El Centro de Seguridad de Windows se llama aquí Centro de actividades, y se integraron en él las categorías de seguridad y el mantenimiento del equipo.

La barra de tareas fue rediseñada, es más ancha, y los botones de las ventanas ya no traen texto, sino únicamente el icono de la aplicación. Estos cambios se hacen para mejorar el desempeño en sistemas de pantalla táctil. Estos iconos se han integrado con la barra «Inicio rápido» usada en versiones anteriores de Windows, y las ventanas abiertas se muestran agrupadas en un único icono de aplicación con un borde, que indica que están abiertas. Los accesos directos sin abrir no tienen un borde. También se colocó un botón para mostrar el escritorio en el extremo derecho de la barra de tareas, que permite ver el escritorio al posar el puntero del ratón por encima.

Se añadieron las «Bibliotecas», que son carpetas virtuales que agregan el contenido de varias carpetas y las muestran en una sola vista. Por ejemplo, las carpetas agregadas en la biblioteca «Vídeos» son: «Mis vídeos» y «Vídeos públicos», aunque se pueden agregar más, manualmente. Sirven para clasificar los diferentes tipos de archivos (documentos, música, vídeos, imágenes).

Una característica llamada «Jump lists» guarda una lista de los archivos abiertos recientemente. Haciendo clic derecho a cualquier aplicación de la barra de tareas aparece una jump list, donde se pueden hacer tareas sencillas según la aplicación. Por ejemplo, abrir documentos recientes de Office, abrir pestañas recientes de Internet Explorer, escoger listas de reproducción en el reproductor, cambiar el estado en Windows Live Messenger,anclar sitos o documentos, etcétera

Existen seis ediciones de Windows 7, construidas una sobre otra de manera incremental, aunque solamente se centrarán en comercializar dos de ellas para el común de los usuarios: las ediciones Home Premium y Professional. A estas dos, se suman las versiones Starter, Home Basic, y Ultimate, además de la versión Enterprise, que está destinada a grupos empresariales que cuenten con licenciamiento Open o Select de Microsoft.

- Starter: Es la versión de Windows 7 con menos funcionalidades. Posee una versión incompleta de la interfaz Aero que no incluye los efectos de transparencia Glass, Flip 3D o las vistas previas de las ventanas en la barra de inicio y además no permite cambiar el fondo de escritorio. Está dirigida a PC de hardware limitado —como netbooks—, siendo licenciada únicamente para integradores y fabricantes OEM. Incluye una serie de restricciones en opciones de personalización y de programas, además de ser la única edición de Windows 7 sin disponibilidad de versión para hardware de 64 bits.
- Home Basic: Versión con más funciones de conectividad y personalización, aunque su interfaz seguirá siendo incompleta como en la edición Starter. Sólo estará disponible para integradores y fabricantes OEM en países en vías de desarrollo y mercados emergentes.

- Home Premium: Además de lo anterior, se incluye Windows Media Center, el tema Aero completo y soporte para múltiples códecs de formatos de archivos multimedia. Disponible en canales de venta minoristas como librerías, tiendas y almacenes de cadena.
- Professional: Equivalente a Vista Business, pero ahora incluirá todas las funciones de la versión Home Premium más «Protección de datos» con «Copia de seguridad avanzada», red administrada con soporte para dominios, impresión en red localizada mediante Location Aware Printing y cifrado de archivos. También disponible en canales de venta al público.
- Ultimate: Añade características de seguridad y protección de datos como BitLocker en discos duros externos e internos, Applocker, Direct Access, BranchCache, soporte a imágenes virtualizadas de discos duros (en formato VHD) y el paquete de opción multilenguaje.
- Enterprise: Esta edición provee todas las características de Ultimate, con características adicionales para asistir con organizaciones IT. Únicamente se vende por volumen bajo contrato empresarial Microsoft software Assurance. También es la única que da derecho a la suscripción del paquete de optimización de escritorio MDOP.
- Ediciones N: Las ediciones N están disponibles para actualizaciones y nuevas compras de Windows 7 Home Premium, Professional y Ultimate. Las características son las mismas que sus versiones equivalentes, pero no incluyen Windows Media Player. El precio también es el mismo, ya que Windows Media Player puede descargarse gratuitamente desde la página de Microsoft

Sistema Operativo: Windows 8

El Windows 8 es la versión actual del sistema operativo de Microsoft Windows, producido por Microsoft para su uso en computadoras personales, incluidas computadoras de escritorio en casa y de negocios, computadoras portátiles, netbooks, tabletas, servidores y centros multimedia. Añade soporte para microprocesadores ARM, además de los microprocesadores tradicionales x86 de Intel y AMD. Su interfaz de usuario ha sido modificada para hacerla más adecuada para su uso con pantallas táctiles, además de los tradicionales ratón y teclado. Microsoft también anunció que Aero Glass no estará presente en la versión final de Windows 8.

Microsoft lanzó a la venta la versión final de Windows 8, el 26 de octubre de 2012, 3 años después del lanzamiento de su predecesor Windows 7. Se lanzó al público general una versión de desarrollo ("Consumer Preview") el 29 de febrero de 2012. Microsoft finalmente anunció una versión casi completa de Windows 8, la Release Preview, que fue lanzada el 31 de mayo de 2012 y es la última versión preliminar de Windows 8 antes de su lanzamiento oficial. El desarrollo de Windows 8 concluyó con el anuncio de la versión RTM el 1 de agosto de 2012

Windows 8 ha recibido duras críticas desde su lanzamiento, lo que ha motivado ventas por debajo de las expectativas para la empresa desarrolladora. Incluso el propio Paul Allen, co fundador de Microsoft, ha dicho que este sistema operativo es «extraño y confuso» en un primer contacto, pero se ha mostrado confiado en que los usuarios aprenderán a «querer» la nueva versión.

The Verge pensó que el énfasis de Windows 8 en la tecnología fue un aspecto importante de la plataforma, y que los dispositivos de Windows 8 (especialmente aquellos que combinan los rasgos de las computadoras portátiles y las tabletas) «convertiría inmediatamente al iPad en algo pasado de moda» debido a las capacidades del modelo híbrido del sistema operativo y el creciente interés en el servicio de la nube. Algunas de las aplicaciones incluidas en Windows 8 fueron consideradas básicas y con carencia de una funcionalidad precisa, pero las aplicaciones de Xbox fueron elogiadas por su promoción de una experiencia de entretenimiento en multi-plataforma. Otras mejoras y características (como el historial de archivos, los espacios de almacenamiento y las actualizaciones para el administrador de tareas) fueron considerados como cambios positivos.36 Peter Bright de Ars Technica sintió que mientras sus cambios de interfaz de usuario quizás los eclipse, la mejoría, el administrador de archivos actualizados, la funcionalidad de un nuevo almacenamiento, las características expandidas de seguridad y la

actualización del Administrador de Tareas de Windows 8 fueron notables mejorías positivas para el sistema operativo. Bright pensó que esa dualidad de Windows 8 hacia las tabletas y los PC tradicionales fueron un aspecto «extremadamente ambicioso» de la plataforma, pero se mantuvo crítico ante la decisión de Microsoft de emular el modelo de Apple como una plataforma de distribución donde implementa una Windows Store.

La interfaz de Windows 8 ha sido objeto de reacciones mixtas. Bright indicó que el sistema de Edge UI del puntero y desplazamiento «no fueron muy obvios» debido a la carencia de instrucciones proporcionadas por el sistema operativo en las funciones accedidas a través del interfaz del usuario, incluso por el manual de vídeo añadido en el lanzamiento del RTM (que solamente instruye a los usuarios a apuntar las esquinas de la pantalla y el toque de sus lados). A pesar de este «obstáculo» autodescrito, Bright aclara que la interfaz de Windows 8 trabajó muy bien en algunos lugares, pero empezó a ser incoherente cuando se cambia entre los ambientes «Metro» y de escritorio, algunas veces a través de medios inconsistentes.37 Tom Warren de The Verge aclaró que la nueva interfaz fue «asombrosa como sorprendente», contribuyendo a una experiencia «increíblemente personal» una vez que es personalizado por el usuario. Al mismo tiempo, Warren vio que la interfaz tiene una empinada curva de aprendizaje, y fue difícil de usar con un teclado y un ratón. Sin embargo, se señaló que, si bien obliga a los usuarios a utilizar la nueva interfaz con una utilidad más táctil, fue un movimiento arriesgado para Microsoft en su conjunto, que era necesario con el fin de impulsar el desarrollo de aplicaciones para el almacén de Windows.

Dos notables desarrolladores de videojuegos criticaron a Microsoft por adoptar una aplicación de jardín vallado similar a otras plataformas de móviles con la introducción de la Windows Store —ya que sentían que estaba en conflicto con la visión tradicional de la PC como una plataforma abierta, debido a la naturaleza cerrada de la tienda y los requisitos de certificación para la compatibilidad y la regulación de los contenidos. Markus "Notch" Persson se negó a aceptar una ayuda de un desarrollador de Microsoft para certificar su popular videojuego Minecraft para la compatibilidad de Windows 8, replicando con una petición para la compañía a «cesar de intentar arruinar la PC como una plataforma abierta». Gabe Newell (cofundador de Valve Corporation que desarrolló el software Steam) describió a Windows 8 como «una catástrofe para cualquiera en el espacio de la PC» debido a la naturaleza cerrada del Windows Store

Sistemas Operativos para pequeños dispositivos: CE, Mobile yPhone

Windows CE (conocido oficialmente como Windows Embedded Compact y anteriormente como Windows Embedded CE,1 también abreviado como WinCE) es un sistema operativo desarrollado por Microsoft para sistemas embebidos. Windows CE no debe confundirse con

Windows Embedded Standard, que es un sistema basado en Windows NT; Windows CE está desarrollado independientemente.

Windows Mobile es un sistema operativo móvil compacto desarrollado por Microsoft, y diseñado para su uso en teléfonos inteligentes (Smartphones) y otros dispositivos móviles.

Se basa en el núcleo del sistema operativo Windows CE y cuenta con un conjunto de aplicaciones básicas utilizando las API de Microsoft Windows. Está diseñado para ser similar a las versiones de escritorio de Windows estéticamente. Además, existe una gran oferta de software de terceros disponible para Windows Mobile, la cual se podía adquirir a través de Windows Marketplace for Mobile.

Originalmente apareció bajo el nombre de Pocket PC, como una ramificación de desarrollo de Windows CE para equipos móviles con capacidades limitadas. En la actualidad, la mayoría de los teléfonos con Windows Mobile vienen con un estilete digital, que se utiliza para introducir comandos pulsando en la pantalla.

Si bien muchos pensamos que Windows Mobile habia sido descontinuado temporalmente en favor del nuevo sistema operativo Windows Phone, la amplia gama de teléfonos industriales

ha hecho a Microsoft optar por una tercera linea de sistemas operativos para móviles que ha llamado Windows Embedded Handheld 6.5, que vendría a ser la nueva linea de sistemas operativos basados en Windows Mobile 6.5

Windows Phone es un sistema operativo móvil desarrollado por Microsoft, como sucesor de la plataforma Windows Mobile.2 A diferencia de su predecesor, está enfocado en el mercado de consumo generalista en lugar del mercado empresarial3 por lo que carece de muchas funcionalidades que proporcionaba la versión anterior. Microsoft ha decidido no hacer compatible Windows Phone con Windows Mobile por lo que las aplicaciones existentes no funcionan en Windows Phone haciendo necesario desarrollar nuevas aplicaciones. Con Windows Phone, Microsoft ofrece una nueva interfaz de usuario que integra varios servicios en el sistema operativo. Microsoft planeaba un estricto control del hardware que implementaría el sistema operativo, para evitar la fragmentación con la evolución del sistema, pero han reducido los requisitos de hardware de tal forma que puede que eso no sea posible.4

El 29 de octubre de 2012 se lanzó al mercado Windows Phone 8 solo para nuevos dispositivos, debido a un cambio completo en el kernel que lo hace incompatible con dispositivos basados en la versión anterior. Esta versión incluye nuevas funciones que de acuerdo a Microsoft lo harán competitivo con sistemas operativos como iOS de Apple o Android de Google.5 Con esta versión comienza la fragmentación de Windows Phone ya que que los dispositivos basados en Windows Phone 7 no pueden actualizarse a Windows Phone 8

9.3. Sistema Operativo: Unix

Unix es un sistema operativo multiusuario y multitarea escrito en el lenguaje C. En su diseño se puso especial cuidado en aislar las rutinas dependientes de hardware, de forma que fuera fácil transportarlo a diferentes plataformas. Así se disponen de versiones para prácticamente todo tipo de computadores, desde ordenadores personales, o estaciones de trabajo, hasta los supercomputadores.

Historia

Unix tiene una larga e interesante historia. El primer desarrollo de Unix se hizo en los laboratorios Bell en el lenguaje ensamblador de la máquina. Pronto se vio que no era práctico tener que reescribir el sistema completamente para cada máquina, por lo que decidieron hacerlo en C, un lenguaje de alto nivel.

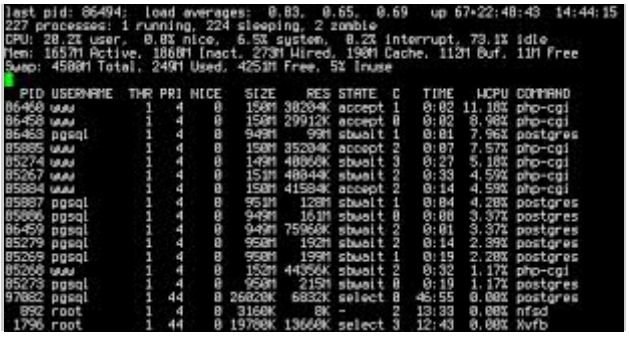
La descripción de Unix se hizo de dominio público en una comunicación técnica de Ritchie y Thompson en 1974. Por ese trabajo ambos recibieron el premio Turing de la ACM. La publicación de Unix hizo que se despertara un gran interés por su estudio. AT&T, dueña entonces de las laboratorios Bell, no tuvo inconveniente en otorgar licencias gratuitas de Unix a instituciones y universidades para que pudieran incorporar sus propias ideas y mejoras en le sistema. Pronto surgieron dos líneas principales de desarrollo de Unix. Por una parte, apareció en 1976 la versión 6, que fue el primer estándar del mundo académico, seguida en 1978 por la versión 7. Se les puede considerar como los antecesores de las versiones comerciales Unix desarrolladas en los años 80 por AT&T.

Por otra parte, la Universidad de California en Berkeley modificó sustancialmente el código original y generó una versión denominada Unix BSD ("*Berkeley Software Distributions"*, Distribuciones de Software de la Universidad de Berkeley). Las numerosas mejoras de Unix BSD hicieron que fabricantes como Sun Microsystems y DEC basaran sus versiones de Unix en la de Berkeley.

Mientras tanto, AT&T continuó desarrollando y mejorando su sistema y en 1982 salió de los laboratorios Bell una nueva versión comercial de Unix denominada Unix System III, que no tuvo gran éxito y fue seguida rápidamente por Unix System V. Al final de la década de los 80 se tenían dos versiones incompatibles de Unix, la Unix 4.3 BSD y la versión 3 de Unix System V, con sus respectivos dialectos, puesto que cada vendedor añadía a sus propias características. Debido a este panorama ha habido distintos intentos de crear un estándar. El primer intento serio se realizó bajo los auspicios del IEEE *Standards Board,* a través del proyecto POSIX ("*Portable Operating System Interface X*", Interfaz X Portable del Sistema Operativo). Este estándar, conocido como 1003.1 es la intersección del Unix System V y del

Unix 4.3BSD. el resultado fue algo parecido al antepasado común de los dos, la versión 7 de Unix.

La realidad es que en la actualidad sigue habiendo tantos sistemas Unix como vendedores, y algunos más desarrollados con una finalidad académica, como Linux, BSD Unix. IBM sigue comercializando sus máquinas con su propia versión denominada AIX, Hewllettt-Packard suministra sus equipos con su versión HP-UX, Sun Microsystems ha pasado de su versión SUN OS, basada en el Unix 4.3BSD, a Solaris, donde se han incorporado nuevas ideas para soportar sus nuevas máquinas RISC con multiprocesadores. Así se podía continuar indicando otras versiones comerciales. En lo que sigue, el sistema que se describe es Unix System V, aunque se comenten algunas características de Posix.



Descripción.

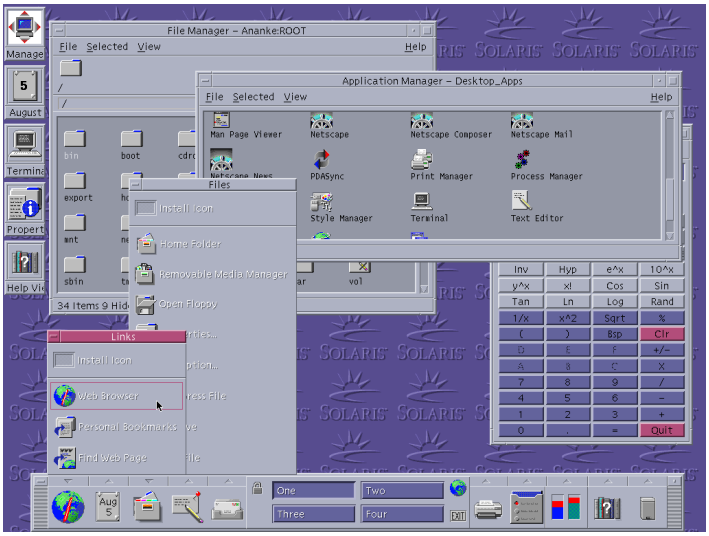
El hardware está rodeado por el núcleo de Unix, que es el auténtico SO, denominado así para enfatizar su aislamiento de las aplicaciones y de los usuarios. Sigue a continuación la capa de librerías, que contiene un procedimiento para cada llamada al sistema. En el estándar Posix se especifica la interfaz de la librería y no el de la llamada al sistema. Por encima de estas capas, todas las versiones de Unix proporcionan una serie de programas de utilidades, como el procesador de órdenes (“*shell*”), los compiladores, los editores, los programas de procesamiento de texto y las utilidades para el manejo de los archivos. Desde el terminal el usuario llama directamente a estos programas.

Se pueden considerar tres interfaces distintas:

- La interfaz de llamadas al sistema.
- La interfaz de librería.
- La interfaz del usuario, que está formada por los programas de utilidades estándar.

El auténtico sistema operativo es el núcleo que interactúa directamente con el hardware y las rutinas primitivas, que corresponden al bloque de control del hardware y en la parte superior está la interfaz de llamadas al sistema, que permite al software de alto nivel tener acceso a funciones específicas del núcleo. El resto del núcleo se puede dividir en dos partes principales, que están asociadas al control de procesos y a la gestión de archivos y de la entrada/salida.

La parte de control de procesos se responsabiliza de la planificación y distribución de los procesos, de la sincronización y comunicación entre procesos, y de la gestión de la memoria. La parte de gestión de archivos intercambia datos entre la memoria y los dispositivos externos, tanto de cadena de caracteres como de bloques, para lo que se disponen de distintos controladores de periféricos. En la transferencia orientada a bloques, se utiliza un método de caché de disco, que interpone un buffer del sistema en la memoria principal entre el espacio de direcciones de usuario y el dispositivo externo.



Control y sincronización de procesos.

En Unix todos los procesos, excepto dos procesos básicos del sistema, se crean mediante órdenes del programa de usuario. Los dos procesos son los de arranque e inicialización. Los nueve estados en los que estar un proceso son:

- Ejecución en modo usuario.
- Ejecución en modo núcleo.

- Preparado para ejecución en memoria principal, tan pronto como lo planifique el núcleo.
- Espera en memoria principal. El proceso está en la memoria principal esperando que ocurra un suceso.
- Preparado para ejecución pero en memoria secundaria. El intercambiador, el proceso 0, debe transferirlo a la memoria principal antes de que el núcleo pueda planificar su ejecución.
- Espera en memoria secundaria. El proceso está aguardando un suceso y tiene que intercambiarse desde la memoria secundaria.
- Adelantado. El proceso se está ejecutando desde el modo núcleo al modo usuario, pero el núcleo lo adelanta y hace un cambio de contexto para planificar otro proceso. Esencialmente es lo mismo que estado de preparado, la distinción se hace para enfatizar la forma de entrar en el estado adelantado.
- Creado o nonato. El proceso se ha creado de nuevo y está en un estado de transición. El proceso existe pero aún no está preparado para ejecución. Este es el estado inicial para todos los procesos, excepto para el proceso 0.
- Zombi. El proceso ejecuta la llamada *exit*() (fin del programa), pero todavía no puede finalizar por lo que queda en estado zombi. Normalmente, un proceso pasa a estado de zombi cuando está a la espera de que finalice alguno de sus hijos.

Comunicación entre procesos

En Unix se tienen distintos algoritmos para la comunicación y sincronización entre procesos. Entre estos, los más importantes son:

- Tuberías (“*pipes*”).
- Señales.
- Mensajes.
- Memoria compartida.
- Semáforos.

Las tuberías permiten transferir datos entre procesos y sincronizar la ejecución de los mismos. Usan un modelo de productor-consumidor, y hay una cola FIFO donde un proceso escribe y el otro lee.

Otra forma de comunicación es mediante el envío de señales. Una señal es un mecanismo software que informa a un proceso de que ha ocurrido algún suceso asíncrono. Una señal es similar a una interrupción hardware. Todas las señales se tratan igual, aquellas que ocurren al mismo tiempo se presentan al proceso simultáneamente, sin ningún orden en particular. Cuando se envía una señal a un proceso, el núcleo actualiza un bit en un campo de señales de la entrada de la tabla del proceso, dependiendo del tipo de señal.

Las tuberías y las señales constituyen una forma limitada de comunicación. Los otros mecanismos estándar de comunicación son:

1. Los mensajes, que permiten a los procesos enviar a cualquier proceso cadenas de datos con un determinado formato.

2. La memoria compartida, que posibilita que los proceso comparten parte de su espacio de direcciones virtuales.

3. Los semáforos, que pueden sincronizar la ejecución de los procesos.

Los mensajes son cadenas de datos con un formato establecido. El proceso que envía un mensaje especifica el tipo del mismo con cada uno de los que envía. El receptor puede usar este dato como un criterio de selección, de forma que puede atender a los mensajes según el orden de llegada en una cola o por el tipo. Cuando un proceso intenta enviar un mensaje a una cola que está llena, el proceso queda suspendido, lo mismo ocurre si intenta leer de una cola vacía.

Pero quizá, la forma más rápida de comunicación entre procesos en Unix es mediante la memoria compartida, que permite que varios procesos accedan al mismo espacio de direcciones virtuales de memoria. Los procesos leen y escriben en la memoria compartida utilizando las mismas instrucciones máquina. Para manipular la memoria compartida se utilizan llamadas al sistema similares a las llamadas de los mensajes.

Los semáforos sincronizan y gestionan el uso de recursos. Se crean en conjuntos, un conjunto consta de uno o más semáforos. Esta generalización de los semáforos da una considerable flexibilidad en el funcionamiento de la sincronización y coordinación de los procesos.

Gestión de la memoria

En las primeras versiones de Unix no se tenían esquemas de memoria virtual, únicamente se utilizaban particiones variables de memoria. En la actualidad, muchas de las implementaciones hacen uso de memorias virtual paginada, utilizándose esquemas de intercambio y paginación.

Aunque el esquema de gestión de la memoria varía de un sistema a otro, en Unix System V se emplean unas estructuras de datos, que con pequeños cambios, son independientes de la máquina. Estas estructuras son: la tabla de páginas, el descriptor de bloques del disco, la tabla de datos de página y la tabla del intercambiador. Se tiene una tabla de páginas para cada proceso, con una entrada para cada una de las páginas de memoria virtual del proceso. Asociada con cada página de un proceso hay una entrada en el descriptor de bloques del disco que describe la copia en el disco de la página virtual. En la tabla de datos del marco de página se describe cada uno de los marcos de la memoria real. El índice de la tabla es el número del marco. Hay varios punteros utilizados para crear listas dentro de la tabla. Los marcos disponibles se enlazan juntos en una lista de marcos libres.

Por último, la tabla de intercambio tiene una entrada por cada página en el dispositivo y existe una por cada dispositivo de intercambio. Esta tabla tiene dos entradas: el controlador de referencia, que es el número de puntos de entradas de la tabla de páginas a una página en el dispositivo de intercambio, y el identificador de la página en la unidad de almacenamiento.

Sistema de archivos

En Unix se distinguen cuatro tipos de archivos:

- **Ordinarios**. Son los archivos que contiene la información del usuario, los programas de aplicación o de utilización del sistema.
- **Directorios**. Son archivos que contiene listas de nombres de archivos, más los punteros asociados a los nodos-i. Están organizados de una forma jerárquica.
- **Especiales**. Estos corresponden a los periféricos: impresoras, discos, etc.
- **Etiquetados**. Son los tubos etiquetados discutidos anteriormente.

Todos los tipos de archivos de Unix se gestionan por el sistema operativo mediante los nodos-i. Estos corresponden a una tabla que contiene los atributos y las direcciones de los bloques del archivo. Los archivos se ubican dinámicamente, según es necesario, sin usar una preubicación, de esta forma, los bloques de un archivo en el disco no son necesariamente contiguos.

Subsistema de entrada/salida

Para el sistema operativo Unix todos los periféricos están asociados a un archivo especial, que se gestiona por el sistema de archivos, pudiéndose leer y escribir como otro archivo más. Se consideran dos tipos de periféricos: de bloque y de carácter. Los periféricos de bloque son periféricos de almacenamiento de acceso arbitrario (por ejemplo los discos). Los periféricos orientados a caracteres incluyen a los otros tipos, por ejemplo las impresoras o los terminales.

La E/S se puede realizar utilizando un buffer, como una zona de almacenamiento intermedio de los datos procedentes o con destino a los periféricos. Hay dos mecanismos de buffer: sistema de caché y colas de caracteres. El caché de buffer es esencialmente una caché de disco. La transferencia de datos entre la caché del buffer y el espacio de E/S del proceso del usuario se realiza mediante DMA, ya que ambos están localizados en la memoria principal. El otro mecanismo de buffer, las colas de caracteres, resulta apropiado para los periféricos orientados a caracteres. El dispositivo de E/S escribe una cola de caracteres que son leídos por el proceso o, inversamente, el proceso los escribe y el periférico los lee. En ambos casos se utiliza un modelo de productor consumidor.

La E/S sin buffer es simplemente una operación de acceso directo a memoria (“DMA”, “Direct Memory Access”), entre el periférico y el espacio de memoria del proceso. Este es el método más rápido para realizar una entrada/salida, sin embargo, un proceso que esté realizando una transferencia de entrada/salida sin buffer está bloqueado en la memoria principal y no puede intercambiarse.

9.4. Sistema Operativo: Linux

Linux fue el proyecto original de un estudiante de informática llamado Linus Torvalds que entonces tenía veintitrés años. Linux empezó siendo un pasatiempo para Linus, que esperaba crear una versión más sólida de UNIX para usuarios de Minix. Tal y como apuntábamos antes, Minix es un programa desarrollado por el profesor de informática Andrew Tannebaum.

El sistema Minix se escribió para demostrar algunos conceptos informáticos que se encuentran en los sistemas operativos. Torvalds incorporó estos conceptos en un sistema autónomo que imitaba a UNIX. El programa se puso a disposición de los estudiantes de informática de todo el mundo y muy pronto contó con muchos seguidores, incluyendo a aquellos que participaban en sus grupos de debate de Usenet. Linus Torvalds decidió entonces proporcionar una plataforma más accesible para los usuarios de Mimix que pudiera ejecutarse en cualquier IBM PC y centró su atención en las recién aparecidas computadoras basadas en 80386 debido a las propiedades de conmutación de tareas que incorporaba la interfaz en modo protegido del 80386.

Propiedad de Linux

Linus Torvalds conserva los derechos de autor del kernel básico de Linux. Red Hat, Inc, posee los derechos de la distribución Red Hat y Paul Volkerding, que se acogen a la GPL ("*General Public License*", Licencia Pública Gerenal) de GNU. De echo, Linux y muchos de los que han contribuido al desarrollo de Linux, han protegido su trabajo con la GPL de GNU.

En ocasiones, esta licencia se denomina GNU Copyleft, que no es más que un juego de palabras con el término inglés Copyright. Esta licencia cubre todos los programas producidos por GNU y por la FSF ("*Free Software Foundation*", Fundación de Software de Libre distribución). Esta licencia permite a los desarrollares crear programas para el público en general. La premisa fundamental de GNU es la de permitir a todos los usuarios acceso libre a los programas con la posibilidad de modificarlos, si así lo desean. La única condición impuesta es que no puede limitarse el código modificado; es decir, que el resto de usuarios tiene derecho también a utilizar el nuevo código.

El GNU Copyleft, o GPL, permite a los creadores de programas conservar sus derechos de autor, pero permitiendo al resto de usuarios la posibilidad de copiarlos, modificarlos y hasta de venderlos. Sin embargo, al hacerlo, no pueden limitar ningún derecho similar a los que compren el programa. Si se vende el programa tal y como está, o una modificación del mismo,

el usuario debe facilitar además el código fuente. Por ello, cualquier versión de Linux incorpora siempre el código fuente completo.

La distribución de Linux corre a cargo de distintas compañías, cada una de ellas con su propio paquete de programas, aunque todas faciliten un núcleo de archivos que conforman una versión de Linux. Las más difundidas son: Debian Linux, Red Hat, Slackware, Mandrake y Ubuntu.

Características de Linux

La mayoría de variantes de UNIX integran un tipo de multitareas llamado multitarea preferente, es decir, que cada programa tiene garantizada la oportunidad de ejecutarse y se ejecuta precisamente el tipo de multitarea que incorpora Linux.

Alguna de las ventajas que ofrece la multitarea preferente, además de reducir los tiempos muertos, es decir, aquellos momentos en los que no puede ejecutar ninguna aplicación porque todavía no se ha completado una tarea anterior, posee una gran flexibilidad, que le permite abrir nuevas ventanas sin tener que cerrar otras con las que está trabajando.

Linux y otros sistemas operativos de multitarea preferentes ejecutan los procesos preferentes, controlando los procesos que esperan para ejecutarse, así como los que se están ejecutando. Después, el sistema programa cada proceso para que todos tengan acceso al microprocesador. El resultado es que las aplicaciones abiertas parecen estar ejecutándose al mismo tiempo, aunque en realidad existe una demora de una aplicación y el momento programado por Linux para volver a ocuparse de ese proceso. La otra gran característica de Linux es que le permite acceder a los códigos de fuente del sistema operativo Linux según sus preferencias. Los fabricantes comerciales nunca le permitirían acceder a estos códigos de fuente para modificar sus productos.

La capacidad de Linux para asignar el tiempo de microprocesador simultáneamente a varias aplicaciones, lógicamente permitió ofrecer acceso a varios usuarios a la vez, ejecutando cada uno de ellos una o varias aplicaciones. La gran ventaja de Linux y de sus características de multitarea y multiusuario es que más de un usuario puede trabajar con la misma versión de la aplicación al mismo tiempo y desde el mismo terminal o desde terminales distintos. Sin embargo, esta capacidad de Linux no debe confundirse con el hecho de que varios usuarios puedan actualizar el mismo archivo simultáneamente, algo que podría llevar a la confusión y al caos total, y por ello resulta indeseable.

***Shell* programables**

El proceso de exploración que realiza el shell se denomina análisis y consiste en la descomposición de las órdenes en componentes que se puedan procesar más fácilmente. Cada componente se interpreta y ejecuta, incluyendo los caracteres especiales que confieren un significado adicional al shell. Estos caracteres especiales se amplían aún más en sus correspondientes procesos de órdenes y se ejecutan.

Aunque muchas versiones UNIX y Linux incluyen más de un tipo de shell, todos ellos funcionan básicamente del mismo modo. Un *shell* realiza la tarea de mediar entre el usuario y el núcleo del sistema operativo Linux. La diferencia esencial entre los tres shell disponibles radica en la sintaxis de la línea de órdenes. Aunque no supone una limitación estrictamente hablando, el uso de las órdenes del shell C o la sintaxis de los *shells Bourne* o *bash* pueden traerle problemas.

Algunos usuarios van incluso más allá, diseñando programas que enlazan procesos y aplicaciones para reducir su trabajo a veces hasta una única sesión de entrada de datos, consiguiendo así que el sistema actualice de una sola vez los numerosos paquetes de programas.

Independencia de dispositivos bajo Linux

UNIX soluciona los problemas que supone añadir otros periféricos, contemplando cada uno de ellos como un archivo aparte. Cuando se necesitan nuevos dispositivos, el administrador del sistema añade al kernel el enlace necesario. Este enlace, también denominado controlador de dispositivo, se ocupa de que el kernel y el dispositivo se fusionen del mismo modo cada vez que se solicita el servicio del dispositivo.

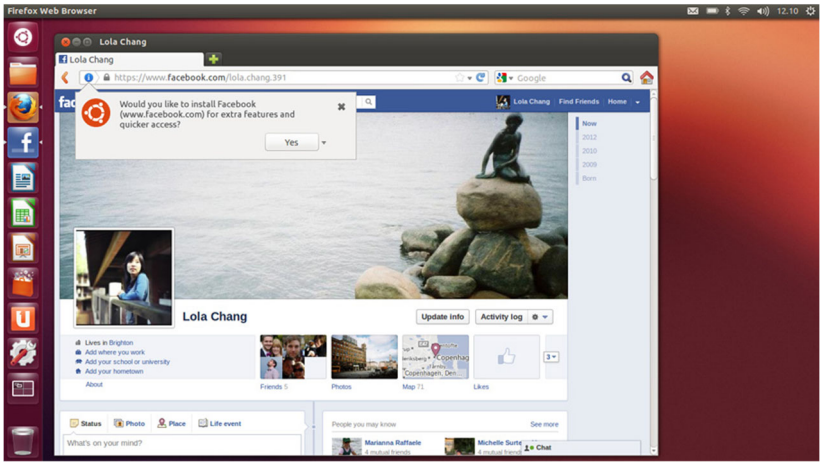
A medida que se van desarrollando y distribuyendo mejores periféricos, el sistema operativo de UNIX permite al usuario un acceso inmediato y sin restricciones a sus servicios, en cuanto los dispositivos se enlazan al kernel. La clave de la independencia de los dispositivos reside precisamente en la adaptabilidad del kernel.

Ventajas y desventajas del uso de Linux

Linux integra además una implementación completa del protocolo de red TCP/IP. Linux permite conectarse a Internet y acceder a toda la información que incluye. Linux también dispone de un sistema completo de correo electrónico con el que se puede enviar y recibir mensajes a través de Internet, y otras redes de ordenadores.

Igualmente, incorpora una completa interfaz gráfica de usuario ("*GUI*"), la Xfree86 basada en el popular sistema X-Windows, y que supone una implementación completa del sistema X-Windows, al que se puede acceder libremente con Linux. Xfree86 ofrece los

elementos GUI habituales que incluyen otras plataformas GUI comerciales, como Windows y OS/2.



Muchas especificaciones de los sistemas abiertos requieren el cumplimiento de POSIX ("*Portable Operating System Interface X*"), lo que significa alguna forma de UNIX. Actualmente, Linux cumple con estos estándares, de hecho, Linux se diseñó para permitir la portabilidad del código fuente, por lo que sí tiene un programa de la empresa que se ejecute sobre una versión de UNIX, podrá trasladar fácilmente dicho programa a un sistema que ejecute Linux.

Linux no dispone de un servicio técnico, lo cual supone un problema para su utilización dentro de una estructura empresarial. Lo mismo sucede con las aplicaciones de Linux porque, aunque existan algunos programas comerciales, la mayoría los desarrollan pequeños grupos que después los ponen a disposición del público. No obstante, muchos desarrolladores prestan su ayuda cuando se le solicita.

Otra desventaja de Linux es que su instalación puede resultar difícil y no funciona en todas las plataformas de hardware. A diferencia de los programas comerciales, donde un mismo equipo de desarrolladores pasa meses construyendo y probando un programa en distintas condiciones y con diferentes hardware, los desarrolladores de Linux se encuentran repartidos por todo el mundo. No existe un programa formal que garantice la calidad de Linux, sino que los distintos desarrolladores lanzan sus versiones cuando quiere. Además, el hardware admitido por Linux depende del utilizado por el desarrollador en el momento de escribir esa parte del código. Por tanto, Linux no funciona con todo el hardware disponible actualmente para PC.

9.5. Máquinas virtuales

Una máquina virtual es un conjunto de programas que simulan la ejecución de otros programas incluso de otros sistemas operativos.

Varios sistemas operativos distintos pueden coexistir sobre el mismo ordenador, en sólido aislamiento el uno del otro, por ejemplo para probar un sistema operativo nuevo sin necesidad de instalarlo directamente.

La máquina virtual puede proporcionar una arquitectura de instrucciones que sea algo distinta de la de la verdadera máquina. Es decir, podemos simular hardware.

Ejemplos: VmWare, VirtualBox, Microsoft Virtual Server, etc.

10. Sistemas Distribuidos: Redes

10.1. Fundamentos de redes.

Definición y tipos.

Una red de computadores es una agrupación de dos o más computadores que se comunican entre sí. Según la dimensión de la red se suelen dividir en dos grupos:

- Redes de área local (“LAN”, Local Area Network). Conectan computadores cercanos unos de las otros. En algunos casos, "local" significa dentro de la misma habitación o edificio; en otros casos, se refiere a computadores ubicados a varios kilómetros de distancia.
- Redes de área extendida o redes de gran alcance (“WAN”, Wide Area Network). Constan de computadores que se encuentran en diferentes ciudades o incluso países. Son redes de larga distancia, debido al gran trayecto que debe recorrer la información que intercambian.

Objetivos de las redes.

El gran auge que han tenido las redes de computadores se debe a la enorme ventaja que supone disponer de equipos de trabajo interconectados. Las principales ventajas se enumeran a continuación.

- Compartir recursos. Todos los programas, datos y equipos, están disponibles para cualquier ordenador de la red que lo solicite, sin importar la localización física del recurso y del usuario.
- Mejora de la fiabilidad. Proporcionan una alta fiabilidad, al contar con fuentes alternativas de suministro. La presencia de múltiples CPU, significa que si una de ellas deja de funcionar, las otras son capaces de su trabajo, aunque se tenga un rendimiento global menor.
- Ahorro económico. Los ordenadores pequeños tienen una mejor relación coste/rendimiento, comparada con la ofrecida por las máquinas grandes. Estas son, a grandes rasgos, diez veces más rápidas que el más rápido de los microprocesadores, pero su costo es miles de veces mayor.
- Medio de comunicación. Una red de ordenadores puede proporcionar un poderoso medio de comunicación entre personas que se encuentran muy alejadas entre sí. A la larga el uso de redes, como un medio para enriquecer la comunicación entre seres humanos, puede ser más importante que los mismos objetivos técnicos, como por ejemplo, la mejora de la fiabilidad.

En la figura 10.1, se muestra la clasificación de sistemas multiprocesadores distribuidos de acuerdo con su tamaño físico. En la parte superior se encuentran las máquinas de flujo de datos, que son ordenadores con un alto nivel de paralelismo y muchas unidades funcionales trabajando en el mismo programa. En el siguiente nivel se encuentran las máquinas multiprocesador, que son sistemas que se comunican a través de memoria compartida. Seguidamente, las redes locales, que son ordenadores que se comunican por medio del intercambio de mensajes. Finalmente, a la conexión de dos o más redes se le denomina interconexión de redes.

Distancia entre procesadores	Procesadores ubicados en el mismo	
.1 m	La tarjeta del circuito	Máquina de flujo de datos
1 m	El sistema	Multiprocesador
10 m	El cuarto	Red local
100 m	El edificio	
1 km	Los terrenos de la universidad	
10 km	La ciudad	Red de gran alcance
100 km	El país	
1000 km	El continente	Interconexión de redes de gran alcance
10000 km	El planeta	

Figura 10. 1 Clasificación de procesadores según su interconexión

Aplicaciones de las redes.

Para dar una idea sobre algunos usos importantes de redes de ordenadores, A continuación se describen brevemente varios ejemplos.

- Acceso a programas remotos. Una empresa que ha producido un modelo que simula la economía mundial puede permitir que sus clientes se conecten usando la red y ejecuten el programa para ver cómo pueden afectar a sus negocios las diferentes proyecciones de inflación, de tasas de interés y de fluctuaciones de tipos de cambio.
- Acceso a bases de datos remotos. Hoy en día, ya es muy fácil ver, por ejemplo, a cualquier persona hacer desde su casa reservas de avión, autobús, barco y hoteles, restaurantes, teatros, etc., para cualquier parte del mundo y obteniendo la confirmación de forma instantánea. En esta categoría también caen las operaciones bancarias que se llevan a cabo desde el domicilio particular, así como las noticias del periódico recibidas de forma automática.
- Medios alternativos de comunicación. La utilización del correo electrónico, que permite mandar y recibir mensajes de cualquier parte del mundo, muestra el gran potencial de las redes en su empleo como medio de comunicación. El correo electrónico es capaz de transmitir la voz digitalizada, fotografías e imágenes móviles de vídeo.

Arquitecturas de redes

La mayoría de las redes se organizan en una serie de capas o niveles, con objeto de reducir la complejidad de su diseño. Cada una de ellas se construye sobre su predecesora. El número de capas, el nombre, el contenido, y la función de cada una varían de una red a otra. Sin embargo, en cualquier red, el propósito de cada capa es ofrecer ciertos servicios a las capas superiores, liberándolas del conocimiento detallado sobre cómo se realizan dichos servicios.

La capa n en una máquina conversa con la capa n de otra máquina. Las reglas y convenciones utilizadas en esta conversación se conocen como protocolo de la capa n. A las entidades que forman las capas correspondientes en máquinas diferentes se les denomina proceso pares; es decir de igual a igual. En otras palabras, son los procesos pares los que se comunican mediante el uso del protocolo.

En realidad, no existe una transferencia directa de datos desde la capa n de una máquina a la capa n de otra, sino más bien, cada capa pasa la información de datos y control a la capa inmediatamente inferior; y así sucesivamente, hasta que se alcanza la capa localizada en la parte más baja de la estructura. Debajo de la 1 está el medio físico, a través del cual se realiza la comunicación real.

Entre cada par de capas adyacentes existe una interfaz, la cual define los servicios y operaciones primitivas que la capa inferior ofrece a la superior. Al conjunto de capas y protocolos se le denomina arquitectura de la red. Las especificaciones de ésta deberán contener la información suficiente que le permita al diseñador escribir un programa o construir el hardware correspondiente a cada capa, y que siga en forma correcta el protocolo apropiado. Tanto los detalles de realización, como las especificaciones de las interfaces, no forman parte de la arquitectura, porque se encuentran escondidas en el interior de la máquina y no son visibles desde el exterior. Más aún, no es necesario que las interfaces de todas las máquinas en una red sean iguales, supuesto que cada una de las máquinas utilice correctamente todos los protocolos.

La abstracción del proceso par es importante para el diseño de redes, sin esta técnica de abstracción sería difícil, dividir el diseño de una red completa; es decir, sería un problema intratable si no se divide en varios más pequeños y manejables, el diseño de capas individuales. En la figura 10.2 se muestra un esquema del significado de los términos capa, protocolo e interfaz.

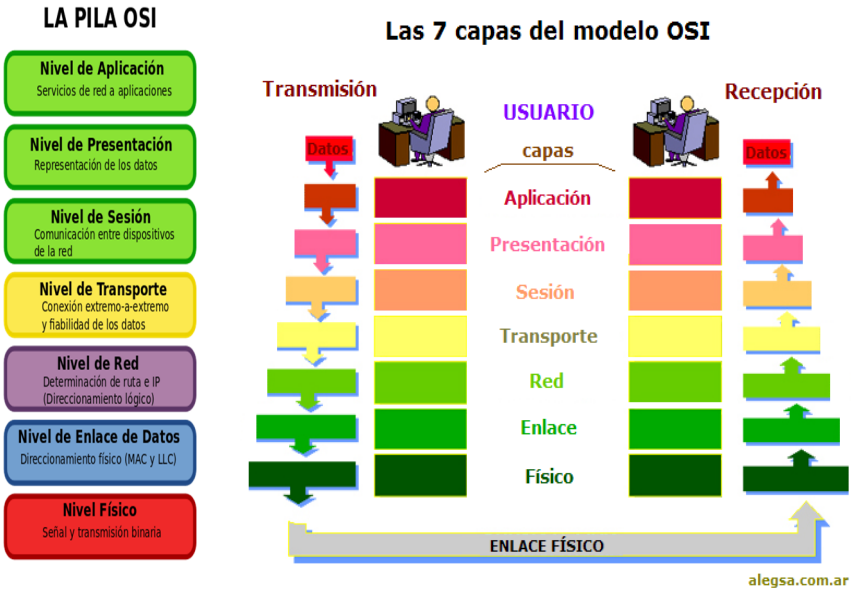


Figura 10. 2 Capas, protocolos e interfaces

10.2. Modelo de referencia OSI

El modelo que se va a tratar a continuación está basado en una propuesta desarrollada por la Organización Internacional de Normas (ISO). Este modelo supuso un primer paso hacia la normalización internacional de varios protocolos, [Stallings 00] y [Tanenbaum 97]. A este modelo se le conoce como Modelo de Referencia Interconexión de sistemas abiertos, más conocido como modelo OSI de ISO, ("OSI", Open System Interconnection). Este modelo de referencia se refiere a la conexión de sistemas heterogéneos, es decir, a sistemas dispuestos a establecer comunicación con otros distintos.

El modelo OSI consta de 7 capas. Los principios aplicados para el establecimiento de siete capas fueron los siguientes:

1. Una capa se creará en situaciones donde se necesita un nivel diferente de abstracción.

2. Cada capa deberá efectuar una función bien definida, que le proporcione entidad propia.

3. La función que realizará cada capa deberá seleccionarse con la intención de definir protocolos normalizados internacionalmente.

4. Los límites de las capas deberán seleccionarse tomando en cuenta la minimización del flujo de información a través de las interfaces.

5. El número de capas deberá ser lo suficientemente grande para que funciones diferentes no tengan que ponerse juntas en la misma capa y, por otra parte, también deberá ser lo suficientemente pequeño para que su arquitectura no llegue a ser difícil de manejar.

Obsérvese que le modelo OSI, por sí mismo, no es una arquitectura de red, dado que no especifica, en forma exacta, los servicios y protocolos que se utilizarán en cada una de las capas. Sólo indica lo que cada capa deberá hacer.

Capas del modelo OSI.

- ***Capa física.***

La capa física se ocupa de la transmisión de bits a lo largo del canal de comunicación. Los problemas de diseño a considerar aquí son los aspectos mecánico, eléctrico, de procedimiento de interfaz y el medio de transmisión física, que se encuentra bajo la capa física.

- ***Capa de enlace.***

La tarea primordial de la capa de enlace consiste en, a partir de un medio de transmisión común y corriente, transformarlo en una línea sin errores de transmisión para la capa de red. Esta tarea requiere que el emisor trocee la entrada de datos en tramas de datos, típicamente construidas por algunos cientos de octetos. Las tramas así formadas son transmitidas de forma secuencial, y requieren de un asentimiento del receptor que confirme su correcta recepción.

- ***Capa de red.***

La capa de red se ocupa del control de la operación de la subred. Un punto de suma importancia en su diseño, es la determinación sobre cómo encaminar los paquetes de origen a destino. Si en un momento dado hay demasiados paquetes presentes en la subred, ellos mismos se obstruirán mutuamente y darán lugar a un cuello de botella. El control de tal congestión dependerá de la capa de red. Además, es responsabilidad de la capa de red resolver problemas de interconexión de redes heterogéneas.

- ***Capa de transporte.***

La función principal de la capa de transporte consiste en aceptar los datos de la capa de sesión, dividirlos, siempre que sea necesario en unidades más pequeñas, pasarlos a la capa de red, y asegurar que todos ellos lleguen correctamente al otro extremo. Además, todo este trabajo se debe hacer de manera eficiente, de tal forma que aísle la capa de sesión de los cambios inevitables a los que está sujeta la tecnología del hardware.

Es una capa de tipo origen-destino, o extremo a extremo. Es decir, un programa en la máquina origen lleva una conversación con un el correspondiente programa par, que se encuentra en la máquina destino, utilizando para ello las correspondientes cabeceras de los mensajes de control. Los protocolos de las capas inferiores se llevan a cabo entre cada máquina y su vecino inmediato, y no entre las máquinas de origen y destino, que podrían estar separadas grandes distancias. Antes de multiplexar varios flujos de mensajes en un canal, la capa de transporte debe ocuparse del establecimiento y liberación de conexiones a través de la red.

- ***Capa de sesión.***

La capa de sesión permite que los usuarios de diferentes máquinas puedan establecer sesiones entre ellos. A través de una sesión se puede llevar a cabo un transporte de datos ordinario, tal y como lo hace la capa de transporte, pero mejorando los servicios que ésta proporciona y que se utilizan en algunas aplicaciones.

Uno de los servicios de la capa de sesión consiste en gestionar el control de diálogo. Las sesiones permiten que el tráfico vaya en ambas direcciones al mismo tiempo, o bien, en una sola dirección en un instante dado. Otros servicios relacionados con esta capa son la administración del testigo y la sincronización.

- ***Capa de presentación.***

A diferencia de las capas inferiores, que únicamente están interesadas en el movimiento fiable de bits de un lugar a otro, la capa de presentación se ocupa de los aspectos de sintaxis y semántica de la información que se transmite.

La capa de presentación está también relacionada con otros aspectos de representación de la información. Por ejemplo, la compresión de datos se puede utilizar aquí para reducir el

número de bits que tienen que transmitirse, y el concepto de criptografía se necesita utilizar a menudo por razones de privacidad y de autentificación.

- ***Capa de aplicación***.

La capa de aplicación contiene una variedad de protocolos que se necesitan frecuentemente. Por ejemplo, hay centenares de tipos de terminales incompatibles en el mundo. Una forma de resolver este problema consiste en definir un terminal virtual de red abstracto.

Otra función de la capa de aplicación es la transferencia de archivos. Distintos sistemas de archivos tienen diferentes convenciones para denominar un archivo, así como diferentes formas para representar las líneas de texto, etc. La transferencia de archivos entre dos sistemas diferentes requiere de la resolución de éstas y de otras incompatibilidades.

Transmisión de datos en el modelo OSI.

En la figura 3.3, se muestra un ejemplo de cómo pueden transmitirse los datos mediante el empleo del modelo OSI. El proceso emisor tiene algunos datos que enviar al proceso receptor. Este entrega los datos a la capa de aplicación, la cual añade entonces la cabecera de aplicación (“AH”, application header), la cual puede ser nula y entrega el elemento resultante a la capa de presentación.

La capa de presentación transforma este elemento de diferentes formas, con la posibilidad de introducir una cabecera en la parte frontal, dando el resultado a la capa de sesión. Es importante observar que la capa de presentación desconoce qué parte de los datos que le dio la capa de aplicación, corresponde a AH, y cuáles son los que corresponden a los verdaderos datos del usuario.

Este proceso se sigue repitiendo hasta que los datos alcanzan la capa física, lugar en donde efectivamente se transmiten datos a la máquina receptora. La idea fundamental, a lo largo de este proceso, es que si bien la transmisión efectiva de datos es vertical, como se muestra en la figura 3.3, cada una de las capas está programada como si fuera una transmisión horizontal.

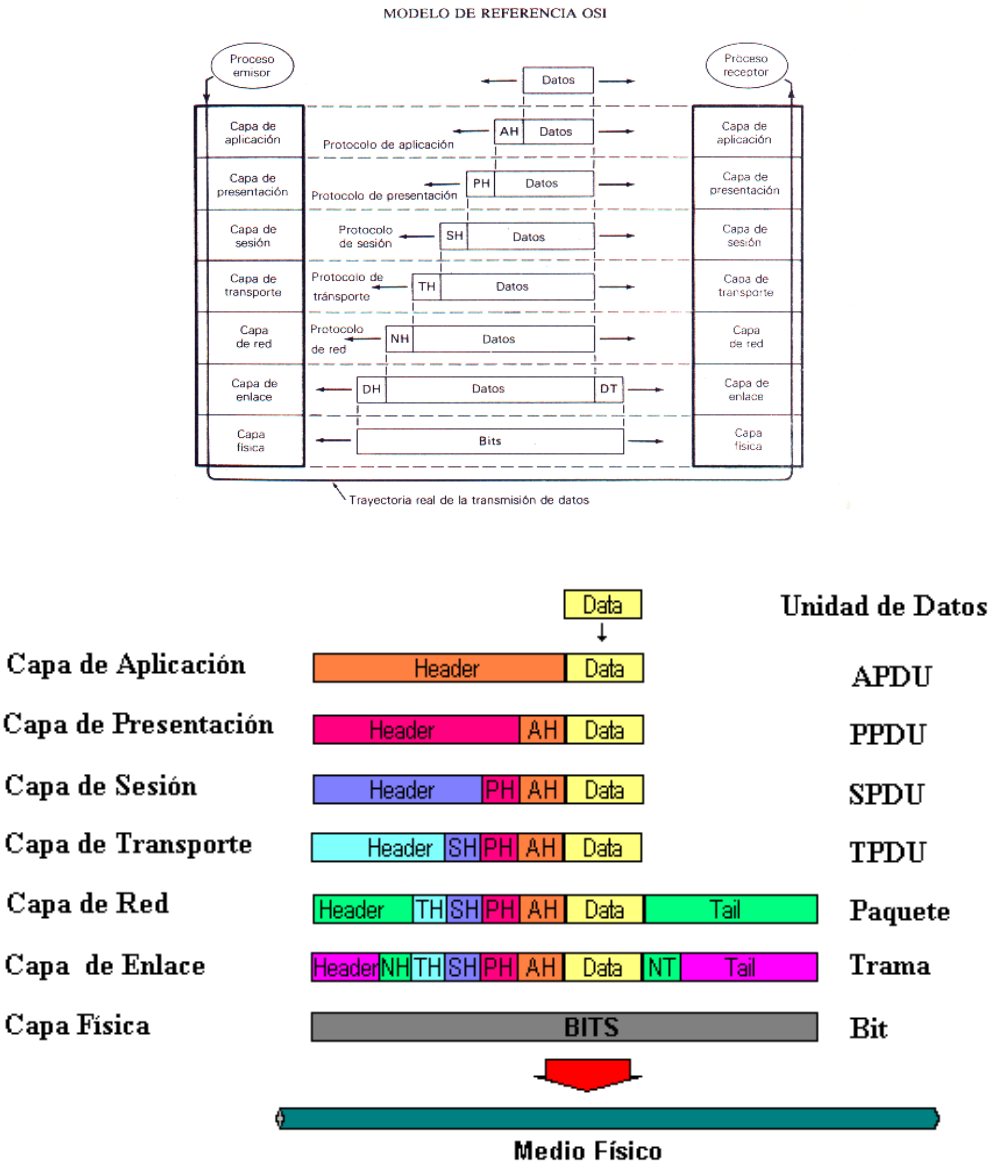


Figura 3. 3 Ejemplo de utilización del modelo OSI de ISO

Terminología y servicios del modelo OSI

Se llaman entidades a los elementos activos que se encuentran en cada una de las capas. Las entidades pueden ser software, como un proceso; o hardware, como el procesamiento realizado un chip inteligente de E/S. Las entidades de la misma capa, pero de diferentes máquinas, se conocen como entidades pares o iguales. A las entidades de la capa 7 se les conoce como entidades de aplicación; a las de la capa 6 como entidades de presentación, etc.

Las entidades de la capa N desarrollan un servicio que utiliza la capa (N+1), en este caso a la capa N se le denomina proveedor del servicio y a la capa (N+1) usuario del servicio. La capa N puede utilizar servicios de la capa (N-1) con objeto de proporcionar su servicio.

Los servicios se encuentran disponibles en el correspondiente Punto de Acceso al Servicio ("SAP", Service Access Point). Los SAP de la capa N son los lugares en donde la capa (N+1) puede acceder a los servicios que se ofrecen. Cada uno de los SAP tiene una dirección que lo identifica de forma particular.

Para que se lleve a cabo el intercambio de información entre dos capas, deberá existir un acuerdo sobre un conjunto de reglas acerca de la interfaz. En una interfaz típica, la entidad de la capa (N+1) pasa una Unidad de Datos de la Interfaz ("IDU", Interface Data Unit) a la entidad de la capa N, a través del correspondiente SAP.

El IDU consiste en una Unidad de Datos del Servicio ("SDU", Service Data Unit) y de alguna información de control. La SDU es la información que se pasa, a través de la red, a la entidad par y posteriormente, a la capa (N+1). La información de control es necesaria porque ayuda a que las capas inferiores realicen su trabajo; por ejemplo, el número de bytes en el SDU, que no forma parte de los datos.

Para hacer la transferencia de una SDU, podrá ser necesario su fragmentación por parte de la entidad de la capa N en varias partes, de tal forma que a cada una de ellas se le asigne una cabecera y se envíe como una distinta Unidad de Datos del Protocolo ("PDU", Protocol Data Unit). Las entidades pares utilizan las cabeceras de la PDU para llevar a cabo su protocolo de igual a igual. Por medio de ellos se identifica cuáles son las PDU que contienen datos y cuáles las que llevan información de control.

Con frecuencia a las PDU de transporte, sesión y aplicación se les conoce como Unidad de Datos del Protocolo de Transporte ("TPDU", Transport Protocol Data Unit), Unidad de Datos del Protocolo de Sesión ("SPDU", Service Protocol Data Unit) y Unidad de Datos del Protocolo de Aplicación ("APDU", Application Protocol Data Unit), respectivamente.

Servicios orientados a conexión y sin conexión.

Las capas pueden ofrecer dos tipos diferentes de servicios a las capas que se encuentran sobre ellas; uno orientado a conexión y otro sin conexión.

- **El servicio orientado a la conexión:**

Se modeló basándose en el sistema telefónico. Para hablar con alguien por teléfono, se debe tomar el teléfono, marcar el número, hablar y colgar. Similarmente, para utilizar una red con servicio orientado a conexión, el usuario del servicio establece primero una conexión, la utiliza y después termina la conexión.

El aspecto fundamental de la conexión es que actúa de forma parecida a un tubo: el que envía, introduce objetos por un extremo, y el receptor los recoge, en el mismo orden, por el otro extremo.

- **El servicio sin conexión:**

Se modela con base en el sistema postal. Cada mensaje, o carta, lleva consigo la dirección completa del destino y cada uno de ellos se encamina, en forma independiente, a través del sistema.

Relación entre servicios y protocolos

Los conceptos de servicio y protocolo tienen un significado diferente, y son complementarios entre sí, la definición de servicio y protocolo según la terminología OSI es la siguiente:

Un servicio es un conjunto de primitivas, u operaciones, que una capa proporciona a la superior. El servicio define las operaciones que la capa efectuará en beneficio de los usuarios, pero no dice nada con respecto a cómo se realizan dichas operaciones. Un servicio se refiere a una interfaz entre dos capas, siendo la capa inferior la que provee el servicio y la capa superior la que utiliza el servicio.

Un protocolo, a diferencia del concepto de servicio, es un conjunto de reglas que gobiernan el formato y el significado de las tramas, paquetes o mensajes que son intercambiados por las entidades corresponsales dentro de una capa. Las entidades utilizan protocolos para realizar

sus definiciones de servicio, teniendo libertad para cambiar de protocolo, pero asegurándose de no modificar el servicio visible a los usuarios.

Normalización de redes

En los primeros tiempos en que apareció el concepto de redes, cada compañía fabricante de ordenadores tenía sus propios protocolos; por ejemplo, IBM tenía más de una docena. Esto daba como resultado que aquellos usuarios que adquirían ordenadores de diferentes compañías, no podían conectarlos y establecer una sola red con ellos. El caos generado por esta incompatibilidad dio lugar a la exigencia de los usuarios para que se estableciera una normalización al respecto.

Esta normalización no solamente iba a facilitar la comunicación entre ordenadores construidos por diferentes compañías, sino también traería como beneficio, el incremento en el mercado para los productos que se plegaran a dicha norma, que conduciría a una producción masiva, una economía de escala por incremento de la producción, así como otro tipo de beneficios cuya tendencia sería disminuir su precio y alentar su posterior aceptación.

Las normas se dividen en dos categorías que pueden definirse como: de facto y de jure. Las normas De Facto (derivado del latín, que significa "del hecho"), son aquellas que se han establecido sin ningún planteamiento formal. Las normas IBM PC y sus sucesoras son normas de facto para ordenadores pequeños de oficina, porque docenas de fabricantes decidieron copiar fielmente las máquinas que IBM sacó al mercado. En contraste, las normas De Jure (derivado del latín, que significa "por ley"), son normas formales, legales, adoptadas por un organismo que se encarga de su normalización. Las autoridades internacionales encargadas de la normalización se dividen, por lo general, en dos clases: la establecida por convenio entre gobiernos nacionales, y la establecida sin un tratado entre organizaciones voluntariamente.

Las normas internacionales son producidas por la ISO (Organización Internacional de Normalización), que es una organización voluntaria, fuera de tratados y fundada en 1946, cuyos miembros son las organizaciones nacionales de normalización correspondientes a los 89 países miembros. Entre sus miembros se incluyen a la ANSI (Estados Unidos), UNE (España), BSI (Gran Bretaña), AFNOR (Francia), DIN (Alemania) y otros 85 organismos.

La ISO consta de aproximadamente 200 comités técnicos (TC), cuyo orden de numeración se basa en el momento de su creación, ocupándose cada uno de ellos de un tema específico. Cada uno de los TC tiene subcomités (SC), los cuales a su vez se dividen en grupos de trabajo (WG).

Otro participante importante en el mundo de las normas es el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos), que es la organización profesional más grande del mundo. Esta

institución, además de publicar numerosas revistas y programar un número muy importante de conferencias anuales, ha establecido un grupo dedicado al desarrollo de normas en el área de la ingeniería eléctrica y computación.

Ejemplos de redes

Actualmente se encuentra funcionando un número muy grande de redes en todo el mundo. Algunas de ellas son redes públicas operadas por proveedores de servicios portadores comunes o PTT, otras están dedicadas a la investigación, también hay redes en cooperativa operadas por los mismos usuarios y redes de tipo comercial o corporativo. Las redes, por lo general, se caracterizan por los puntos citados a continuación:

- La *historia* y la *administración* pueden variar desde una red cuidadosamente elaborada por una sola organización, con un objetivo muy bien definido, hasta una colección específica de máquinas, cuya conexión se fue realizando con el paso del tiempo, sin ningún plan maestro o administración central que la supervisara.

- Los servicios ofrecidos, que pueden variar desde una comunicación arbitraria de proceso a proceso, hasta el desarrollo de un sistema de correo electrónico, transferencia de archivos o acceso remoto.

- Los diseños técnicos se diferencian según el medio de transmisión empleado, los algoritmos de encaminamiento y denominación utilizados, el número y contenido de las capas presentes y los protocolos usados.

- Por último, las comunidades de usuarios, que pueden variar desde una sola corporación, hasta aquella que incluye todos los ordenadores científicos que se encuentran en el mundo industrializado.

Algunos ejemplos de redes son:

- Redes públicas: Las empresas privadas, y los gobiernos de varios países han comenzado a ofrecer servicios de redes a cualquier organización que desee subscribirse a ellas. La subred es propiedad de la compañía operadora de redes y proporciona un servicio de comunicación para los clientes y terminales. Aunque las redes públicas, en diferentes países, son en general, muy diferentes en cuanto a su estructura interna, todas ellas utilizan el modelo OSI y las normas CCITT o los protocolos OSI para todas las capas.

- ARPANET es la Red de la Agencia de Proyectos de Investigación Avanzada, cuyo origen está en la red ARPA. Esta red es propiedad de la Agencia de Proyectos de

Investigación Avanzada de la Defensa, que pertenece al Departamento de Defensa de Estados Unidos. Gran parte del conocimiento actual sobre redes de ordenadores se debe directamente al resultado del proyecto ARPANET.

- CSENET realmente, CSNET no es una red real como ARPANET, sino más bien es una metared que utiliza los servicios de transmisión brindados por otras redes y añade una capa protocolo uniformadora en la parte superior, para hacer que todo ello parezca como una sola red lógica para los usuarios.

- BITNET se inició en el año 1981 por iniciativa de la Universidad de Nueva York y de la Universidad de Yale. La idea por la que se montó esta red fue la creación de una red universitaria, parecida a CSNET, que incluyera todos los Departamentos de la misma. Esta se extendió en alrededor de 150 localidades de Estados Unidos y entre 260 de Europa, a través de su equivalente denominada EARN, denominada Red Europea de Investigación Académica (“REIA”).

- USENET esta red fue desarrollada en las Universidades de Duke y North Carolina, ofrece un servicio de red de noticias para los sistemas UNIX. Las noticias por red se dividen en cientos de grupos de noticias con una gran diversidad de temas, algunos de ellos son de carácter técnico y otros no. Los usuarios de USENET se pueden subscribir a cualquier grupo que les interese.

- SNA es la arquitectura de redes de IBM, y se traduce por Arquitectura de Redes de Sistemas. El modelo OSI se configuró tomando como base la red SNA, incluyendo el concepto de estratificación, el número de capas seleccionadas y sus funciones aproximadas. SNA es una arquitectura de red que permite que los clientes de IBM construyan sus propias redes privadas, tomando en cuenta a los hosts y la subred. Aunque es posible llevar a cabo una correspondencia aproximada de las capas SNA con las capas del modelo OSI, si se observa con detalle, se puede apreciar que los dos modelos no tienen una correspondencia completa, especialmente en las capas 3, 4 y 5.

- INTERNET: Red de redes. Se habla extensamente de ella en el apartado siguiente.

10.3. Protocolo TCP/IP

El protocolo TCP/IP (Transmission Control Protocol/Internet Protocol, Protocolo de Control de Transmisión/Protocolo de Internet) proporciona un sistema independiente de intercambio de datos entre ordenadores y redes locales de distinto origen, [Comer 96], [Orfali 98], y [Hahn 94]. Este protocolo ha evolucionado hasta nuestros tiempos a pesar de que han aparecido otros como el ATM (Asynchronous Transfer Mode, Modo de Transferencia Asíncrono), que han demostrado ser más potentes pero menos difundidos.

Las funciones propias de una red de computadoras pueden ser divididas en las siete capas propuestas por ISO para su modelo de referencia OSI; sin embargo la implantación real de una arquitectura puede diferir de este modelo. Las arquitecturas basadas en TCP/IP proponen cuatro capas en las que las funciones de las capas de sesión y presentación son responsabilidad de la capa de aplicación y las capas de enlace y física son vistas como la capa de Interfaz a la Red. Por tal motivo, para TCP/IP sólo existen las capas Interfaz de Red, la de Intercomunicación en Red, la de Transporte y la de Aplicación. Como puede verse TCP/IP presupone independencia del medio físico de comunicación, sin embargo existen estándares bien definidos a los niveles de enlace de datos y físico que proveen mecanismos de acceso a los diferentes medios y que en el modelo TCP/IP deben considerarse la capa de Interfaz de Red; siendo los más usuales el proyecto IEEE802, Ethernet, Token Ring y FDI.

Las cuatro capas del modelo TCP/IP se describen seguidamente:

- **Capa de Aplicación**. Invoca programas que acceden servicios en la red. Interactuan con uno o más protocolos de transporte para enviar o recibir datos, en forma de mensajes o bien en forma de flujos de bytes.

- **Capa de Transporte**. Provee comunicación extremo a extremo desde un programa de aplicación a otro. Regula el flujo de información. Puede proveer un transporte confiable asegurándose que los datos lleguen sin errores y en la secuencia correcta. Coordina a múltiples aplicaciones que se encuentren interactuando con la red simultáneamente de tal manera que los datos que envíe una aplicación sean recibidos correctamente por la aplicación remota, esto lo hace añadiendo identificadores de cada una de las aplicaciones. Realiza además una verificación por suma, para asegurar que la información no sufrió alteraciones durante su transmisión.

- **Capa Internet**. Controla la comunicación entre un equipo y otro, decide qué rutas deben seguir los paquetes de información para alcanzar su destino. Conforma los paquetes IP que será enviados por la capa inferior. Desencapsula los paquetes recibidos pasando a la capa superior la información dirigida a una aplicación.

- **Capa de Interfaz de Red**. Emite al medio físico los flujos de bits y recibe los que de él provienen. Consiste en los gestores de los dispositivos que se conectan al medio de transmisión.

Para entender el funcionamiento de los protocolos TCP/IP debe tenerse en cuenta la arquitectura que ellos proponen para comunicar redes. Tal arquitectura ve como iguales a todas las redes a las que se conecta, sin tener en cuenta el tamaño de ellas, ya sean locales o de cobertura amplia. Se establece que todas las redes que intercambiarán información deben estar conectadas a un mismo computador o equipo de procesamiento (dotados con dispositivos de comunicación); a tales computadoras se les denomina compuertas, pudiendo recibir otros nombres como enrutadores, routers, o puentes.

Para que en una red dos computadoras puedan comunicarse entre sí ellas deben estar identificadas con precisión Este identificador puede estar definido en niveles bajos (identificador físico) o en niveles altos (identificador lógico) de pendiendo del protocolo utilizado. TCP/IP utiliza un identificador denominado dirección Internet o dirección IP, cuya longitud es de 32 bites. La dirección IP identifica tanto a la red a la que pertenece una computadora como a ella misma dentro de dicha red. En la siguiente tabla se establecen los diferentes tipos de redes que se pueden establecer según la dirección IP.

Clase	Rango	N° de Redes	N° de Host Por Red	Máscara de Red	Broadcast ID
A	1.0.0.0 - 126.255.255.255	126	16.777.214	255.0.0.0	x.255.255.255
B	128.0.0.0 - 191.255.255.255	16.384	65.534	255.255.0.0	x.x.255.255
C	192.0.0.0 - 223.255.255.255	2.097.152	254	255.255.255.0	x.x.x.255
(D)	224.0.0.0 - 239.255.255.255	histórico			
(E)	240.0.0.0 - 255.255.255.255	histórico			

Teniendo en cuenta una dirección IP podría surgir la duda de cómo identificar qué parte de la dirección identifica a la red y qué parte al nodo en dicha red. Lo anterior se resuelve mediante la definición de las "Clases de Direcciones IP". Para clarificar lo anterior puede verse que una red con dirección clase A queda precisamente definida con el primer octeto de la dirección, la clase B con los dos primeros y la C con los tres primeros octetos. Los octetos restantes definen los nodos en la red específica.

Las subredes tienen una gran importancia en la implantación de redes. Estas subredes son redes físicas distintas que comparten una misma dirección IP. Deben identificarse una de otra usando una máscara de subred. La máscara de subred es de cuatro bytes y para obtener el número de subred se realiza un operación AND lógica entre ella y la dirección IP de algún equipo. La máscara de subred deberá ser la misma para todos los equipos de la red IP.

Uno de los inconvenientes de estas direcciones IP es su complicada memorización por parte de los usuarios. Para solucionarlo, se adoptó una tabla denominada nombre de dominio que traducía las direcciones IP a una serie de códigos más comprensibles. Esto permite que una dirección IP como «123.89.100.102» puede traducirse mediante la tabla de dominios en «copernico.latoa.upm.e». La primera parte del dominio indica el nombre de la máquina a quien corresponde la dirección IP, en este caso copernico; a continuación, viene el nombre de la organización a la que pertenece el servidor, que sería el latoa.upm; en tercer lugar, se encuentra un código que puede referirse al tipo de organización a la que pertenece el sistema, o al país donde reside. Así, existen códigos como «.gov» (organismo gubernamental), «.edu» (universidades o centros educativos), «.com» (redes comerciales pertenecientes a empresas) o códigos de países como «.it» para Italia, «.mx» para México, o «.es» para España.

Conversión de direcciones en la red

El Protocolo de Resolución de Direcciones ARP (Address Resolution Protocol) permite a un equipo obtener la dirección física de un equipo destino, ubicado en la misma red física, proporcionando solamente la dirección IP destino.

Las direcciones IP y física de la computadora que consulta es incluida en cada emisión general ARP, el equipo que contesta toma esta información y actualiza su tabla de conversión. ARP es un protocolo de bajo nivel que oculta el direccionamiento de la red en las capas inferiores, permitiendo asignar, a nuestra elección, direcciones IP a los equipos en una red física.

Una conversión dinámica de direcciones Internet a direcciones físicas es la más adecuada, debido a que se obtiene la dirección física por respuesta directa del nodo que posee la dirección IP destino. Una vez que la dirección física se obtiene ésta es guardada en una tabla temporal para subsecuentes transmisiones, de no ser así podría haber una sobrecarga de tráfico en la red debido a la conversión de direcciones por cada vez que se transmitiera un paquete.

La implementación del protocolo ARP requiere varios pasos que se describen a continuación. En primer lugar, la interfaz de red recibe un datagrama IP a enviar a un equipo destino, en este nivel se coteja la tabla temporal de conversión, si existe una la referencia adecuada ésta se incorpora al paquete y se envía. Si no existe la referencia un paquete ARP de emisión general, con la dirección IP destino, es generado y enviado. Todos los equipos en la red física reciben el mensaje general y comparan la dirección IP que contiene con la suya propia, enviando un paquete de respuesta que contiene su dirección IP. El computador origen actualiza su tabla temporal y envía el paquete IP original, y los subsecuentes, directamente a la computadora destino.

El funcionamiento de ARP no es tan simple como parece. Supóngase que en una tabla de conversión exista un mapeo de una máquina que ha fallado y se le ha reemplazado la interfaz de red; en este caso los paquetes que se transmitan hacia ella se perderán pues ha cambiado la dirección física, por tal motivo la tabla debe eliminar entradas periódicamente.

Protocolo de Internet (IP)

El Protocolo Internet proporciona un servicio de distribución de paquetes de información orientado a no conexión de manera no fiable. La orientación a no conexión significa que los paquetes de información, que será emitido a la red, son tratados independientemente, pudiendo viajar por diferentes trayectorias para llegar a su destino. El término no fiable significa más que nada que no se garantiza la recepción del paquete. Las principales características de este protocolo son:

- Protocolo orientado a no conexión.
- Fragmenta paquetes si es necesario.

- Direccionamiento mediante direcciones lógicas IP de 32 bits.
- Si un paquete no es recibido, este permanecerá en la red durante un tiempo finito.
- Realiza el "mejor esfuerzo" para la distribución de paquetes.
- Tamaño máximo del paquete de 65635 bytes.
- Sólo ser realiza verificación por suma al encabezado del paquete, no a los datos éste que contiene

La unidad de información intercambiada por IP es denominada datagrama. Tomando como analogía los marcos intercambiados por una red física los datagramas contienen un encabezado y una área de datos. IP no especifica el contenido del área de datos, ésta será utilizada arbitrariamente por el protocolo de transporte.

La Unidad de Transferencia Máxima determina la longitud máxima, en bytes, que podrá tener un datagrama para ser transmitida por una red física. Obsérvese que este parámetro está determinado por la arquitectura de la red: para una red Ethernet el valor de la MTU es de 1500 bytes. Dependiendo de la tecnología de la red los valores de la MTU pueden ir desde 128 hasta unos cuantos miles de bytes. La arquitectura de interconexión de redes propuesta por TCP/IP indica que éstas deben ser conectadas mediante una compuerta. Sin obligar a que la tecnología de las redes físicas que se conecten sea homogénea. Por tal motivo si para interconectar dos redes se utilizan medios con diferente MTU, los datagramas deberán ser fragmentados para que puedan ser transmitidos. Una vez que los paquetes han alcanzado la red extrema los datagramas deberán ser reensamblados.

Enrutamiento de paquetes

El enrutamiento es el proceso que determinar la trayectoria que un datagrama debe seguir para alcanzar su destino. A los dispositivos que pueden elegir las trayectorias se les denomina enrutadores. En el proceso de enrutamiento intervienen tanto los equipos como las compuertas que conectan redes. El termino compuerta es impuesto por la arquitectura TCP/IP de conexión de redes, sin embargo una compuerta puede realizar diferentes funciones a diferentes niveles, una de esas funciones puede ser la de enrutamiento y por tanto recibir el nombre de enrutador.

Existen dos tipos de enrutamiento; el directo y el indirecto. Debido a que en el enrutamiento directo los datagramas se transmite de un equipo a otro, en la misma red física, el proceso es muy eficiente. La vinculación entre la dirección física y la IP se realiza mediante el ARP.

En el enrutamiento directo la transmisión de datagramas IP entre dos equipos de la misma red física sin la intervención de compuertas. El emisor encapsula el datagrama en la trama de la red, efectuando la vinculación entre la dirección física y la dirección IP, y envía la trama resultante en forma directa al destinatario. En el enrutamiento indirecto las compuertas forman una estructura cooperativa, interconectada. Las compuertas se envían los datagramas hasta que se alcanza a la compuerta que puede distrubuirla en forma directa a la red destino.

El enrutamiento en IP se basa en la gestión de tablas. Las tablas de enrutamiento están presentes en todo equipo que almacene información de cómo alcanzar posibles destinos. En las tablas no se almacena la ruta específica a un equipo, sino aquellas a la red donde se encuentre. Cada puerto de comunicación de la compuerta debe poseer una dirección IP. Para que en los equipos no exista una tabla excesivamente grande, que contenga todas las rutas a las redes que se interconecta un equipo, es de gran utilidad definir una ruta por defecto. A través de esta ruta se deberán alcanzar todas las redes destino. La ruta por defecto apunta a un dispositivo que actúa como compuerta de la red donde se encuentre ubicado el equipo que la posee.

la arquitectura de interconexión de redes de TCP/IP cada par de redes se conectan mediante compuertas. Para que los paquetes alcancen sus redes destino las compuertas deben contar con mecanismos mediante los cuales ntercambien la información de las redes que conecta cada uno.

La arquitectura de enrutamiento por Compuerta Núcleo se basa es definir una compuerta que centraliza las funciones de enrutamiento entre redes, a esta compuerta se le denomina núcleo. Cada compuerta en las redes a conectar tiene como compuerta por defecto a la compuerta núcleo. Varias compuertas núcleo pueden conectarse para formar una gran red; entre las compuertas núcleo se intercambiará información concerniente a las redes que cada una de ellas alcanzan. La arquitectura centralizada de enrutamiento fue la primera que existió. Sus principales problemas radican no tanto en la arquitectura en sí, si no en la forma en que se propagaban las rutas entre las compuertas núcleo.

Conforme las complejidades de las redes aumentaron se debió buscar un mecanismo que propagase la información de rutas entre las compuertas. Este mecanismo debía ser automático esto obligado por el cambio dinámico de las redes. De no ser así las transiciones entre las compuertas podían ser muy lentas y no reflejar el estado de la red en un momento dado. Para resolver este problema se define el vector de distancia. Se asume que cada compuerta comienza su operación con un conjunto de reglas básicas de cómo alcanzar las redes que conecta. Las rutas son almacenadas en tablas que indican la red y los saltos para alcanzar esa red. Periódicamente cada compuerta envía una copia de las tablas que alcanza directamente. Cuando una compuerta recibe el comunicado de la otra actualiza su tabla

incrementando en uno el número de saltos. Este concepto ayudó a definir cuantas compuertas debería viajar un paquete para alcanzar su red destino. Mediante el vector una compuerta puede saber a que otra compuerta enviar el paquete de información, sabiendo que ésta podría no ser la última compuerta por la que el paquete tendría que viajar. Este esquema permite tener varios caminos a una misma red, eligiendo el camino más corto, es decir aquella compuerta que con menos saltos conduzca a la red destino.

Protocolo de Control de Transferencia (TCP)

El Protocolo de Control de Transferencia (TCP) proporciona una comunicación bidireccional completa mediante circuitos virtuales. Desde el punto de vista del usuario la información es transmitida por flujos de datos. La fiabilidad en la transmisión de datos se basa en:

- Asignación de números de secuencia a la información segmentada.
- Validaciones de la información recibida.
- Reconocimiento de paquetes recibidos.

Se utiliza el principio de ventana deslizable para esperar reconocimientos y reenviar información. Este mecanismo proporciona una transferencia fiable de flujos de información. Aunque está íntimamente relacionado con IP TCP es un protocolo independiente de propósito general. Al ser un protocolo de alto nivel su función es que grandes volúmenes de información lleguen a su destino correctamente, pudiendo recobrar la pérdida esporádica de paquetes.

Offsets Octeto		0								1								2								3							
Octeto	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Puerto de origen																Puerto de destino															
4	32	Número de secuencia																															
8	64	Número de acuse de recibo (si ACK es establecido)																															
12	96	Longitud de Cabecera				Reservado				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Tamaño de Ventana														
16	128	Suma de verificación																Puntero urgente (si URG es establecido)															
20	160	Opciones (Si la Longitud de Cabecera > 5, relleno al final con "0" bytes si es necesario)																															
...	...	...																															

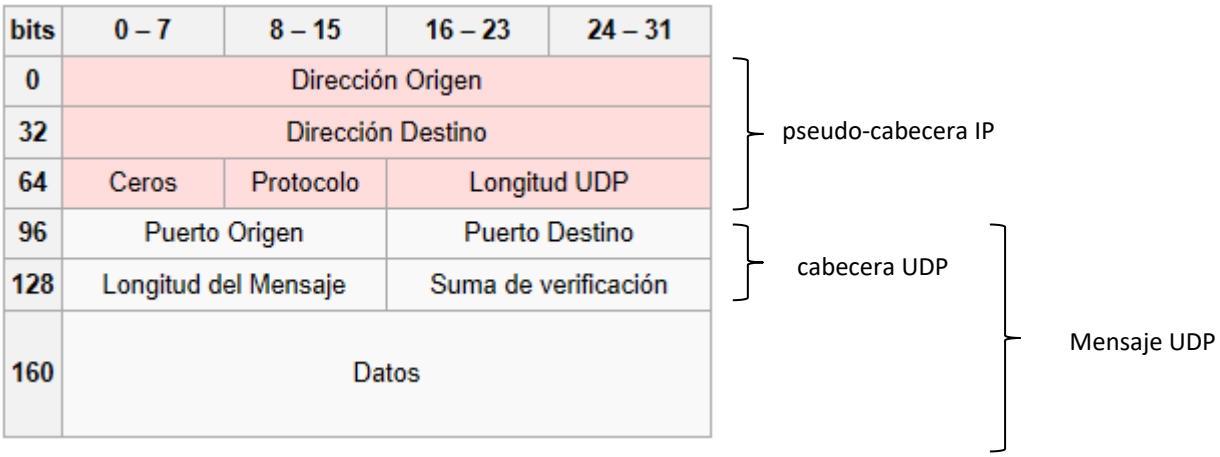
Cada vez que un paquete es enviado se inicializa un contador de tiempo, al alcanzar el tiempo de expiración, sin haber recibido el reconocimiento, el paquete se reenvía. Al llegar el reconocimiento el tiempo de expiración se cancela. A cada paquete que es enviado se le asigna un número de identificador, el equipo que lo recibe deberá enviar un reconocimiento

de dicho paquete, lo que indicará que fue recibido. Si después de un tiempo dado el reconocimiento no ha sido recibido el paquete se volverá a enviar. Obsérvese que puede darse el caso en el que el reconocimiento sea el que se pierda, en este caso se reenviará un paquete repetido.

Protocolo de Datagramas de Usuario

Este protocolo (UDP) proporciona de mecanismos primordiales para que programas de aplicación de se comuniquen con otros en computadoras remotas. Utiliza el concepto de puerto para permitir que múltiples conexiones accedan a un programa de aplicación.

Provee un servicio no confiable orientado a no conexión. El programa de aplicación tiene la total responsabilidad del control de confiabilidad, mensajes duplicados o perdidos, retardos y paquetes fuera de orden. Este protocolo deja al programa de aplicación a ser explotado la responsabilidad de una transmisión fiable. Con él puede darse el caso de que los paquetes se pierdan o bien no sean reconstruidos en forma adecuada. Permite un intercambio de datagramas más directo entre aplicaciones y puede elegirse para aquellas que no demanden una gran cantidad de datagramas para operar óptimamente.



10.4. Orígenes y servicios de Internet.

Origen y evolución

Para evitar que un ataque nuclear pudiera dejar aisladas a las instituciones militares y universidades, en 1969 el ARPA (Advanced Research Projects Agency), una agencia subsidiaria del Departamento de Defensa de Estados Unidos, desarrolló una red denominada ARPAnet basada en el protocolo de intercambio de paquetes.

Un protocolo de intercambio de paquetes es un sistema que divide la información en partes y las envía una por una al ordenador de destino con un código de comprobación. Si el código de comprobación no es correcto, se solicita al ordenador de destino que vuelva a enviar los paquetes corruptos.

La ventaja de este sistema de transmisión es principalmente su fiabilidad de los datos, independientemente de la calidad de la línea utilizada. Los datos llegan incluso si no funcionan, o son destruidos parte de los nodos de la red, factor que influyó decisivamente para su adopción por parte del gobierno norteamericano.

Otra ventaja es que este tipo de sistemas permite distribuir más fácilmente los datos, ya que cada paquete incluye toda la información necesaria para llegar a su destino, por lo que paquetes con distinto objetivo pueden compartir un mismo canal. Además, es posible comprimir cada paquete para aumentar la capacidad de transmisión, o encriptar su contenido para asegurar la confidencialidad de los datos. Estas virtudes han asegurado la supervivencia de los protocolos desde las primeras pruebas realizadas en 1968 por el Laboratorio Nacional de Física del Reino Unido hasta nuestros días.

El primer protocolo utilizado por ARPAnet fue el denominado NCP (Network Control Protocol, Protocolo de Control de Red), que se empleó en la red hasta 1982. En este año se adoptó el protocolo TCP/IP procedente de los sistemas Unix, que cada vez tenían más importancia dentro de ARPAnet.

En 1969 el ARPA instaló el primer servidor de información en un ordenador Honeywell 516 que incorporaba nada menos que 12 KBytes de memoria. Pronto se añadirían otros servidores por parte del Instituto de Investigación de Stanford, de la Universidad de California en Santa Barbara, y de la Universidad de Utah que formaron los primeros nodos de ARPAnet.

Sin embargo, la red Internet como «red de redes» no comenzó a funcionar hasta después de la primera conferencia de comunicaciones por ordenador en octubre de 1972. En esta convención ARPAnet presentaba una red de 40 nodos y se propuso su conexión con otras redes internacionales. Representantes de varios países formaron así el INWG (Inter Network

Working Group) para establecer un protocolo común con el ARPA, empezando a dar forma lo que hoy en día conocemos por Internet.

Con el tiempo ARPAnet fue sustituida por NSFnet, la red de la fundación nacional para la ciencia de EEUU, como organismo coordinador de la red central de Internet. Desde los primeros pasos de ARPAnet, hasta hoy en día, la red ha sufrido pocos cambios, comparado con los avances de la informática. Los cambios más drásticos se han producido en la infraestructura de la red, aumentando la velocidad de transmisión hasta permitir el funcionamiento de aplicaciones multimedia y la transmisión de vídeo o sonido en tiempo real.

También han sufrido cambios el tipo de servicios ofrecidos por Internet, ya que si bien las utilidades en modo texto han sobrevivido hasta nuestros días, la verdadera estrella de la Red es la World Wide Web, un servicio de consulta de documentos hipertextuales que ha logrado gran popularidad tanto entre expertos como entre profanos.

Uno de los cambios más espectaculares que ha sufrido Internet es el del número de usuarios. Desde los primeros tiempos de ARPAnet en los que sólo una decena de ordenadores y un centenar de usuarios tenían acceso a la red, hemos pasado a cifras importantes. Se calcula que existen unos 4 millones de sistemas conectados actualmente a Internet, facilitando acceso a unos 50 millones de usuarios en todo el mundo.

Las cifras son aún más sorprendentes si se considera que el crecimiento actual del censo de usuarios de Internet es aproximadamente de un diez por ciento mensual. Estas cifras indican, por ejemplo, que en el siglo XXI los usuarios de Internet podrían alcanzar en número a los que ven televisión actualmente. Esto hace que Internet se está convirtiendo en una realidad de nuestro tiempo, y puede provocar una pequeña revolución en nuestra forma de vida, del mismo modo que lo han hecho los ordenadores, teléfonos móviles, discos compactos, etc.

Este fenómeno ha atraído los intereses de muchas empresas de todos los sectores, que ven en Internet un vehículo ideal para actividades comerciales, técnicas o de marketing, además de un medio de distribución directa de software, y en general de información de todo tipo.

Los servicios más importantes de la Red

Hasta ahora hemos hablado de Internet como simple medio de comunicación, gracias al cual los ordenadores pueden intercambiar datos mediante un determinado protocolo. Sin embargo, a través de estas líneas de comunicación existen multitud de servicios de todo tipo, algunos de los cuales se describen a continuación.

CORREO ELECTRÓNICO

Uno de los primeros servicios que la red ARPAnet incorporó desde sus inicios fue el correo electrónico. Como el correo normal, su versión electrónica, también denominada e-mail, proporciona un medio para que cualquier usuario de una red pueda enviar mensajes a otras personas.

A pesar de que cada usuario puede estar utilizando un ordenador o una aplicación de correo electrónico diferente, e incluso pertenecer a redes de ordenadores no conectadas directamente a Internet, el protocolo utilizado SMTP (Simple Mail Transfer Protocol, Protocolo de Transmisión de Mensajes Simples), asegura una absoluta compatibilidad en el intercambio de mensajes.

En un principio, este servicio se limitaba a poner en contacto a las personas mediante el intercambio más o menos rápido de mensajes. En la actualidad, es posible mandar todo tipo de datos binarios pudiendose incluir en nuestros mensajes todo tipo de imágenes, sonidos y ficheros binarios, incluidos programa ejecutables.

Existen también otros servicios que pueden ser utilizados a través del correo electrónico: acceso a librerías de software, consulta de información sobre ciertos temas, participación en juegos de estrategia, o discusión con otros usuarios en foros temáticos.

Las direcciones de correo electrónico están compuestas por un nombre de usuario, el símbolo «@» y el nombre completo del dominio del ordenador que estamos utilizando o del proveedor de acceso que hemos contratado. Si el usuario con el que queremos contactar no se encuentra en una red unida directamente a Internet, existen unas tablas de conversión que permitirán a nuestros mensajes llegar a su destino a través de las oportunas «pasarelas», que no son otra cosa que ordenadores que hacen de puente entre dos redes y que permiten cierto intercambio de información entre ellas.

Si se dispone de un acceso a Internet se puede contar con una dirección propia de correo electrónico y un espacio, denominado buzón, en el disco del ordenador que funciona como servidor de correo. En este espacio se irán almacenando los mensajes de correo electrónico que vayan llegando con nuestra dirección para que, cuando nos conectemos, podamos comprobar si tenemos correo nuevo y acceder a él. En muchos casos el tamaño de este buzón será limitado. Al enviar o recibir un mensaje de correo electrónico debe tenerse en cuenta que su viaje por Internet va a ser más lento que con otro tipo de servicios, ya que tiene asignado un ancho de banda menor que el de otros de acceso directo.

Desde su creación el correo electrónico es sin duda el servicio más utilizado dentro de Internet. Millones de personas de todo el mundo envían sus mensajes, incluyendo en algunos casos imágenes digitales y música. Por otro lado, este medio está desplazando en importancia al Fax en muchos ambientes empresariales, ya que ofrece grandes ventajas de costes y

calidad además de una gran flexibilidad, a pesar del inconveniente de su escasa rapidez de transmisión.

TELNET

El programa Telnet permite acceder mediante un terminal virtual a una máquina conectada a Internet, siempre que se conozca su dirección IP, o nombre de dominio, y se disponga de un nombre de usuario y la correspondiente clave de acceso. Según sea el sistema al que se accede, se tendrá que utilizar un tipo de terminal distinto. Los más habituales son el vt100 para máquinas Unix o VAX y el ANSI para PCs.

Una imagen del programa TELNET sobre Windows se puede observar en la figura 3.5. Para la conexión con la otra máquina deberemos introducir el Nombre de host o su dirección IP, el Puerto por el que se realizará la conexión y el Tipo de terminal.

Figura 10. 4 Interfaz para Telnet, o conexión remota

Además de acceder a cuentas privadas en ordenadores remotos, es posible introducirse en servicios gratuitos o en programas de búsqueda como el «Archie». Este tipo de servicios mediante Telnet ha perdido mucha popularidad en los últimos años, ya que otros programas como los navegadores de World Wide Web ofrecen acceso a información con todo lujo de

gráficos y con posibilidad de usar el ratón, mientras que el Telnet sólo puede funcionar en modo texto. Sin embargo, aún son muchos los servicios que se ofrecen mediante Telnet para los nostálgicos de los terminales de texto, y en todo caso su utilidad como herramienta para trabajar desde casa sigue siendo importante.

FTP

Este es otro de los servicios más populares dentro de Internet y sigue siendo uno de los más útiles a la hora de transmitir o recibir ficheros de cualquier tipo. El FTP (File Transfer Protocol, Protocolo de Transferencia de Ficheros). Es un programa que funciona con el protocolo TCP/IP y que permite acceder a un servidor de este sistema de intercambio para recibir o transmitir archivos de todo tipo.

Además de conocer la dirección de la máquina de la que se quiere extraer ficheros, también es necesario asegurarse de que en ese ordenador esté funcionando un programa «demonio» o servidor de FTP. En el caso de que exista, tendremos que conocer un nombre de usuario y una clave de acceso válidos, a menos que el servidor permita efectuar FTP anónimos.

Existen multitud de sistemas a lo largo de la red dedicados a ofrecer servicio de FTP anónimo, poniendo a disposición ficheros de todo tipo. Muchas empresas como Creative Labs, IBM o Microsoft utilizan este método para ofrecer a los usuarios las últimas versiones de los drivers o utilidades disponibles para sus productos.

También existen servidores dedicados exclusivamente a recoger todo tipo de software con licencia shareware y freeware para cualquier sistema operativo. Además, dentro de Internet encontramos servidores dedicados a recoger documentos como las famosas «FAQ» (Frequently Asked Questions, Preguntas y respuestas habituales), que tienen respuesta a las preguntas más frecuentes realizadas por los usuarios de Internet dentro del servicio de News, o los documentos «FYI» (For Your Information, Información para usted), con información de todo tipo sobre la Red.

Aunque ya han pasado muchos años desde la primera implementación del FTP, este servicio sigue siendo el método de transporte de datos por excelencia en Internet.

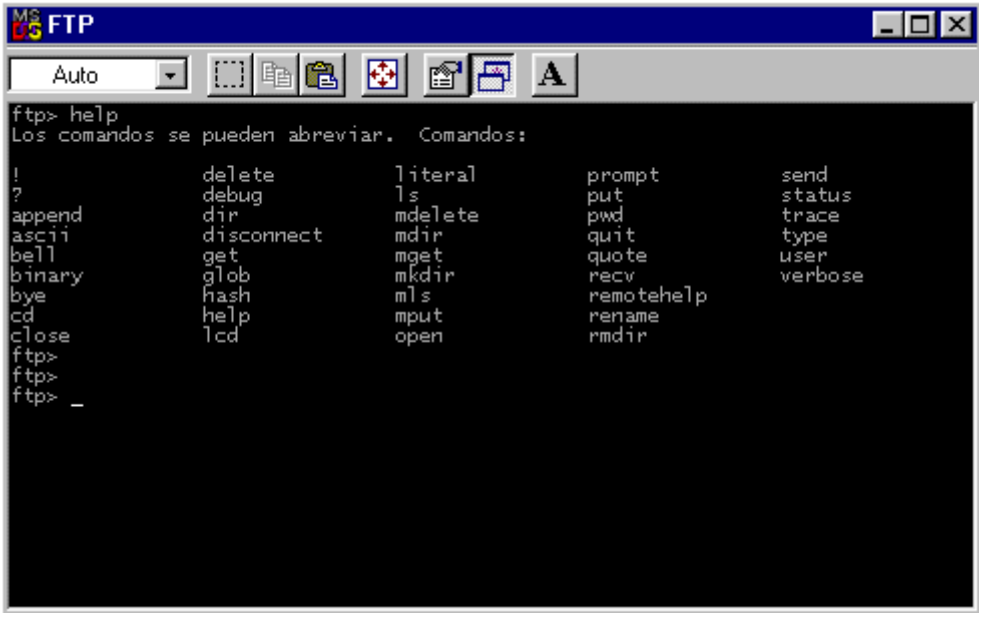


Figura 10. 5 Comandos de FTP

WWW: ("*World Wide Web*", Extensa Telaraña Mundial)

La WWW es una red de servidores dentro de Internet que ofrece páginas hipertextuales en un formato denominado HTML (HyperText Markup Language, Lenguaje de Marcas de Hipertexto). Es un lenguaje de definición de páginas con extensiones hipertextuales portable a cualquier tipo de plataforma gráfica. Estas páginas contienen, además de texto en varios formatos, imágenes, sonidos o vídeo, y permiten lanzar la lectura de páginas de otros servidores activando ciertas palabras resaltadas dentro del mismo documento.

En otras palabras, al consultar un documento de WWW se encontrarán algunas palabras o direcciones resaltadas o en otro color. Al activar estas palabras mediante el ratón pasaremos a otra página que puede pertenecer al mismo servidor o a cualquier otra máquina dentro de Internet, o puede activar cualquier otro servicio de Internet como la consulta de un grupo de News o la transmisión o recepción de ficheros mediante FTP.

Tanto las páginas Web como otros servicios de Internet tienen asignada su propia URL (Uniform Resource Locator, Código Uniforme de Recursos). Se trata de un código que

contiene la identificación del servicio, la dirección del servidor y si es necesario el directorio donde se encuentran los ficheros necesarios dentro del sistema remoto.

Así, una URL de la forma «http://www.mec.es» indica que queremos consultar una página de World Wide Web en formato http ("HyperText Transfer Protocol", Protocolo de Transferencia de Hipertexto). En el servidor de dirección de dominio completa «www.mec.es». Si quisiéramos realizar un FTP, la URL sería por ejemplo «ftp://ftp.microsoft.com/pub», donde «ftp.microsoft.com» es la dirección del servidor y «/pub» es el directorio donde se encuentran los ficheros que nos interesan.

Los documentos de hipertexto no son propios de Internet, sino que se encuentran en numerosas aplicaciones informáticas. Un ejemplo de su utilización son las ayudas del propio Sistema Operativo Windows; que incorporan, en el propio texto, enlaces a otras páginas de ayuda para aclarar ciertos conceptos.

El lenguaje HTML

Navegar por Internet mediante un programa cliente de World Wide Web se convierte en una tarea sumamente sencilla y agradable, aprovechando todas las posibilidades de la Red. Esta sensación ha sido aumentada considerablemente gracias a empresas como Netscape, que han desarrollado programas cliente realmente amistosos con el usuario y de muy sencillo manejo, incorporando multitud de utilidades para facilitar la exploración de la Red.

Todas estas virtudes son posibles gracias al lenguaje HTML propuesto por vez primera, en marzo del año 1989, en los laboratorios de investigación de física de partículas de Suiza, el CERN. Esta institución encargó a uno de sus departamentos el desarrollo de un sistema de lectura de documentos que facilitara la divulgación científica, [Alvarez 97], [Musciano 98]

No fue sino hasta el año 1990, cuando el proyecto de esta prestigiosa institución, recibió el nombre definitivo de World Wide Web. A finales de ese mismo año empezaron a funcionar las primeras versiones del sistema en plataformas NeXT. A partir de la presentación de este prototipo en determinadas convenciones y especialmente en un seminario que ofreció el propio CERN, en junio de 1991, muchas instituciones se interesaron por el proyecto.

En 1992 ya se encontraba disponible el primer cliente de World Wide Web y su difusión se realizaba mediante FTP anónimo, distribuyéndose rápidamente y ganando cierta popularidad en el entorno de Internet.

En 1994, año de la primera conferencia internacional de World Wide Web, ya se encontraban registrados oficialmente unos 1.500 servidores de este sistema. En la actualidad se calcula

que los servidores de este tipo pueden llegar a ser varias decenas de miles, aunque el cálculo probablemente se quede corto dada la dificultad del recuento por el rápido crecimiento del número de ellos.

Desde que el CERN sacara a la luz su primera especificación, el lenguaje HTML fue adquiriendo cada vez más posibilidades, incorporando capacidades multimedia y más recientemente la posibilidad de crear páginas en tres dimensiones. Casi sin darse cuenta los responsables del CERN habían creado uno de los fenómenos de mayor repercusión en Internet a lo largo de su historia.

Actualmente el lenguaje HTML ha llegado ya a su versión 4, estando la versión 5 en implantación. En la versión 4 las especificaciones abarcaban ya muchas más posibilidades que las propuestas originales. Esta versión fue diseñada para superar ciertas limitaciones de desarrollos anteriores, admitiendo verdaderas tablas y solucionando ciertos problemas con la alineación y justificación de texto reseñadas en anteriores versiones. En esta nueva definición también es posible introducir objetos multimedia con distintos formatos. Además, permite incluir applets, que son pequeños módulos ejecutables, como los de Java, o los COM (Component Object Model, Modelo de Objetos de Componentes) desarrollados por Microsoft, y que permiten añadir enlaces o controles OLE.

Gracias a estas facilidades proporcionadas por el lenguaje muchas otras empresas han empezado a desarrollar multitud de applets con distintas funciones. Un applet es un pequeño programa ejecutable que envía el servidor al programa cliente para que funcione incorporado a la página.

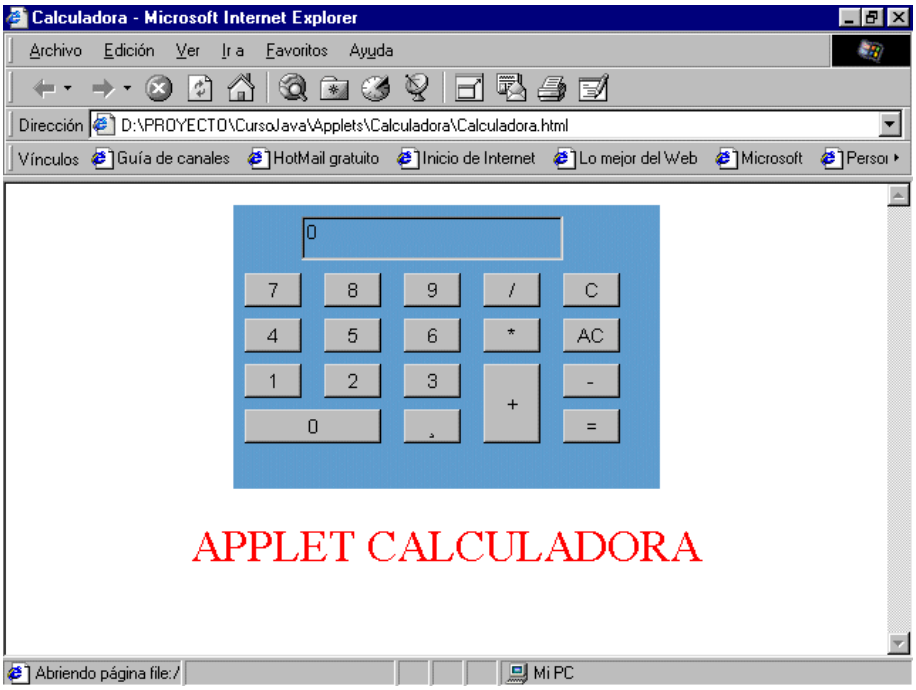


Figura 10. 6 Ejemplo de un applet que simula una calculadora

De esta manera, aunque el lector de páginas no incorpore ciertas funciones, éstas pueden añadirse desde el servidor para que pueda leer, por ejemplo, diversos formatos multimedia sin incorporar en el mismo programa el lector para dichos documentos.

Aplicaciones externas

Además de los applets, muchas empresas y desarrolladores independientes han desarrollado aplicaciones externas que proporcionan a los navegadores más posibilidades. Mediante ciertas aplicaciones se puede escuchar sonido en vivo o ver televisión a través de una página de Web.

Entre las aplicaciones accesorias que han sido desarrolladas últimamente encontramos aquellas que permiten hablar y recibir sonido a través de Internet. Estos programas convierten nuestro ordenador en un verdadero teléfono digital de cobertura mundial con funciones interesantes y un ahorro considerable en llamadas internacionales.

La ventaja de los applets frente a este tipo de añadidos es que se cargan mientras se está consultando la página y luego son borrados del disco local. Además, si el propio servidor es el

que proporciona la aplicación para reproducir los ficheros que contiene, la compatibilidad de medios está asegurada.

Para sacar el máximo partido a la navegación por la World Wide Web es aconsejable conseguir un navegador de 32 bits que pueda funcionar en cualquiera de los sistemas operativos de la actualidad que admitan multitarea. De esta manera podemos esperar la carga de una página de Web, mientras nos ocupamos de otras cosas, e incluso utilizar a la vez otro servicio Internet para no desperdiciar el ancho de banda de la comunicación.

Algunos de los navegadores, como es el caso de la última versión del Netscape Navigator, incorporan herramientas para el uso de otros servicios de Internet como correo electrónico o News, facilitando la interacción con la Red.

Además de las funciones de los propios navegadores, hay que tener en cuenta las múltiples utilidades que ofrecen los servidores de WWW. Las más útiles y populares son las herramientas de búsqueda por la Red, que mediante determinados criterios y consultando una base de datos de direcciones de páginas más o menos extensa nos ofrecen una lista de direcciones que cumplen con los criterios que hemos especificado.

Navegadores.

Desde que diera sus primeros pasos en el laboratorio suizo de física nuclear del CERN allá por el año 1989, la World Wide Web ha experimentado un espectacular aumento, tanto en oferta de servidores, como en número de usuarios, convirtiéndose sin lugar a dudas en el servicio más popular dentro de Internet, sobre todo por la facilidad de acceso a los recursos de la red.

La World Wide Web, por su propia estructura, tiene la virtud de convertir la exploración de Internet en una tarea sencilla y agradable, ya que ofrece la posibilidad de acceder a gran cantidad de información de una manera intuitiva mediante un sistema de consulta basado en un entorno gráfico. Sin embargo, este conjunto de posibilidades sólo son accesibles a través del programa adecuado, se precisa de un navegador, o programa cliente que sea capaz de interpretar el lenguaje de descripción de página que utiliza la WWW.

Casi todas estas aplicaciones son de libre distribución para la mayoría de las plataformas, utilizando entornos gráficos tan extendidos como X-Window y Windows. La utilización del

entorno gráfico permite sacar el máximo partido de las posibilidades hipertextuales de la WWW, ofreciendo funciones que facilitan el acceso a los recursos de Internet.

Los programas clientes de WWW, o navegadores, empezaron a desarrollarse en los laboratorios de programación del NCSA ("National Centre for Supercomputing Applications", Centro Nacional para Aplicaciones de Supercomputación). Desde los primeros años, se ha mantenido en la vanguardia de los diseñadores de este tipo de aplicaciones. En estos laboratorios se comenzó a dar forma la primera interfaz para la consulta de páginas de World Wide Web, que en el año 1993 recibió el nombre de Mosaic. A partir del lanzamiento del primer Mosaic, otras compañías como Netscape empezaron a desarrollar sus propios clientes Web, ofreciendo soluciones particulares y propuestas para mejorar el sistema de consulta de páginas Web y optimizar así las prestaciones del servicio.

De hecho, existe una gran diferencia entre explorar la red con un buen programa cliente de WWW que abarque todas las posibilidades del lenguaje de descripción HTML y que añada sus propias utilidades, a realizar esta misma operación con un programa de menor sofisticación técnica que puede recortar sensiblemente las posibilidades del sistema.

11. Comunicación por Sockets

Como se ha visto en el capítulo anterior, para que varios ordenadores se comuniquen entre sí, es claro que deben estar de acuerdo en cómo transmitirse información, de forma que cualquiera de ellos pueda entender lo que están transmitiendo los otros, de la misma forma que nosotros nos ponemos de acuerdo para hablar en inglés cuando uno es italiano, el otro francés, el otro español y el otro alemán.

Al "idioma" que utilizan los ordenadores para comunicarse cuando están en red se le denomina **protocolo**. Como se ha visto, hay muchos protocolos de comunicación, entre los cuales el más extendido es el **TCP/IP** porque entre otras cosas es el que se utiliza en **Internet**.

De hecho, tanto en Linux/Unix como en Windows contamos con serie de librerías que nos permiten enviar y recibir datos de otros programas, en C/C++ o en otros lenguajes de programación, que estén corriendo en otros ordenadores de la misma red.

En este capítulo se presenta una introducción a los sistemas distribuidos, los servicios proporcionados en POSIX para el manejo de Sockets, que son los conectores necesarios (el recurso software) para la comunicación por la red, y su uso en nuestra aplicación. No pretende ser una guía exhaustiva de dichos servicios sino una descripción práctica del uso más sencillo de los mismos, y

como integrarlos en nuestra aplicación para conseguir nuestros objetivos. De hecho, en el curso del capítulo se desarrolla una clase C++ que encapsula los servicios de Sockets, permitiendo al usuario un uso muy sencillo de los mismos que puede valer para numerosas aplicaciones, aunque obviamente no para todo.

11.1. LOS SOCKETS

Una forma de conseguir que dos programas se transmitan datos, basada en el protocolo TCP/IP, es la programación de sockets. Un socket no es más que un "canal de comunicación" entre dos programas que corren sobre ordenadores distintos o incluso en el mismo ordenador.

Desde el punto de vista de programación, un socket no es más que un "fichero" que se abre de una manera especial. Una vez abierto se pueden escribir y leer datos de él con las habituales funciones de read() y write() del lenguaje C. Hablaremos de todo esto con detalle más adelante.

Existen básicamente dos tipos de "canales de comunicación" o sockets, los orientados a conexión y los no orientados a conexión.

En el primer caso ambos programas deben conectarse entre ellos con un socket y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. Esta es la parte TCP del protocoloTCP/IP, y garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato y no importa que los programas se queden "bloqueados" esperando o transmitiendo datos. Si uno de los programas está atareado en otra cosa y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos.

En el segundo caso, no es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no. Es el llamado protocolo UDP, y garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos. Se utiliza cuando es muy importante que el programa no se quede bloqueado y no importa que se pierdan datos. Imaginemos, por ejemplo, un programa que está controlando la temperatura de un horno y envía dicha temperatura a un ordenador en una sala de control para que éste presente unos gráficos de temperatura. Obviamente es más importante el control del horno que el perfecto refresco de los gráficos. El programa no se puede quedar bloqueado sin atender al horno simplemente porque el ordenador que muestra los gráficos esté ocupado en otra cosa.

11.2. ARQUITECTURA CLIENTE SERVIDOR

A la hora de comunicar dos programas, existen varias posibilidades para establecer la conexión inicialmente. Una de ellas es la utilizada aquí. Uno de los programas debe estar arrancado y en espera de que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Al programa que actúa de esta forma se le conoce como servidor. Su nombre se debe a que normalmente es el que tiene la información que sea disponible y la "sirve" al que se la pida. Por ejemplo, el servidor de páginas web tiene las páginas web y se las envía al navegador que se lo solcite.

El otro programa es el que da el primer paso. En el momento de arrancarlo o cuando lo necesite, intenta conectarse al servidor. Este programa se denomina cliente. Su nombre se debe a que es el que solicita información al servidor. El navegador de Internet pide la página web al servidor de Internet.

En este ejemplo, el servidor de páginas web se llama servidor porque está (o debería de estar) siempre encendido y pendiente de que alguien se conecte a él y le pida una página. El navegador de Internet es el cliente, puesto que se arranca cuando nosotros lo arrancamos y solicita conexión con el servidor cuando nosotros escribimos en la barra del navegador , por ejemplo, http://www.elai.upm.es.com

En los juegos de red, por ejemplo el Age of Empires, debe haber un servidor que es el que tiene el escenario del juego y la situación de todos los jugadores en él. Cuando un nuevo jugador arranca el juego en su ordenador, se conecta al servidor y le pide el escenario del juego para presentarlo en la pantalla. Los movimientos que realiza el jugador se transmiten al servidor y este actualiza escenarios a todos los jugadores.

Resumiendo, servidor es el programa que permanece pasivo a la espera de que alguien solicite conexión con él, normalmente, para pedirle algún dato. Cliente es el programa que solicita la conexión para, normalmente, pedir datos al servidor.

EL SERVIDOR

A partir de este punto comenzamos con lo que es la programación en C/C++ de los sockets. Una vez incluida e inicializada la librería de sockets (paso que hay que hacer en Windows, no así en Linux) en C los pasos que debe seguir un programa servidor son los siguientes:

- **Apertura de un socket**, mediante la función **socket()**. Esta función devuelve un descriptor de fichero normal, como puede devolverlo open(). La función socket() no hace absolutamente nada, salvo devolvernos y preparar un descriptor de fichero que el sistema posteriormente asociará a una conexión en red.

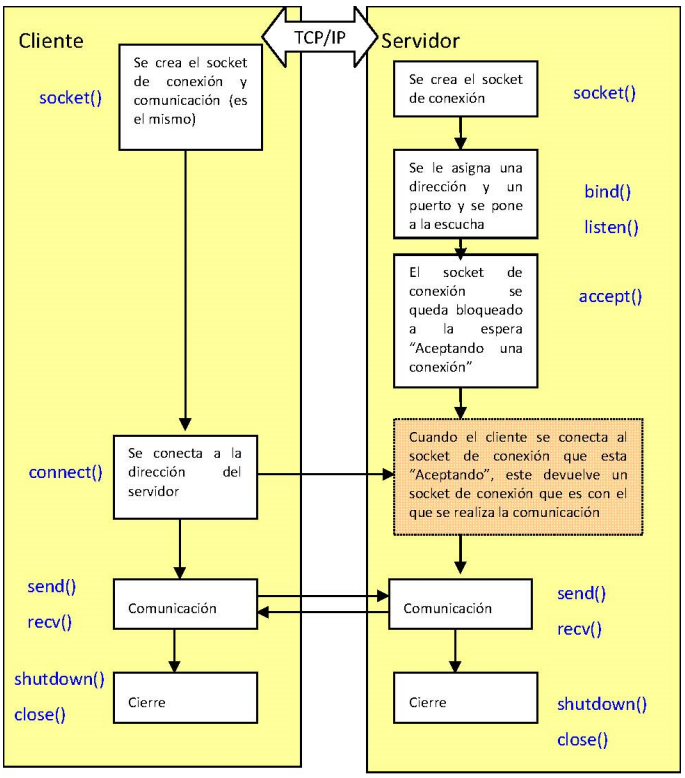
- **Avisar al sistema operativo** de que hemos abierto un socket y queremos que asocie nuestro programa a dicho socket. Se consigue mediante la función **bind()**. El sistema todavía no atenderá a las conexiones de clientes, simplemente anota que cuando empiece a hacerlo, tendrá que avisarnos a nosotros. Es en esta llamada cuando se debe indicar el número de puerto al que se quiere atender.
- Avisar al sistema de que **comience a atender dicha conexión** de red. Se consigue mediante la función **listen()**. A partir de este momento el sistema operativo anotará la conexión de cualquier cliente para pasárnosla cuando se lo pidamos. Si llegan clientes más rápido de lo que somos capaces de atenderlos, el sistema operativo hace una "cola" con ellos y nos los irá pasando según vayamos pidiéndolo.
- Pedir y **aceptar las conexiones** de clientes al sistema operativo. Para ello hacemos una llamada a la función **accept()**. Esta función le indica al sistema operativo que nos dé al siguiente cliente de la cola. Si no hay clientes se quedará bloqueada hasta que algún cliente se conecte.
- **Escribir y recibir datos** del cliente, por medio de las funciones **write()** y **read()**, que son exactamente las mismas que usamos para escribir o leer de un fichero. Obviamente, tanto cliente como servidor deben saber qué datos esperan recibir, qué datos deben enviar y en qué formato.
- **Cierre de la comunicación** y del socket, por medio de la función **close(),** que es la misma que sirve para cerrar un fichero.

EL CLIENTE

Los pasos que debe seguir un programa cliente son los siguientes:

- **Apertura de un socket**, como el servidor, por medio de la función **socket()**
- **Solicitar conexión** con el servidor por medio de la función **connect()**. Dicha función quedará bloqueada hasta que el servidor acepte nuestra conexión o bien si no hay servidor en el sitio indicado, saldrá dando un error. En esta llamada se debe facilitar la dirección IP del servidor y el número de puerto que se desea.
- **Escribir y recibir datos** del servidor por medio de las funciones **write()** y **read()**.
- **Cerrar la comunicación** por medio de **close()**.

A continuación se presenta el código de un programa cliente y de un programa servidor, para describir breve y generalmente los servicios de sockets implicados. Este código es prácticamente el más básico posible, sin comprobación de errores. El funcionamiento será como sigue: Primero se arranca el programa servidor, que inicializa el socket servidor y se queda a la espera de una conexión. A continuación se debe lanzar el programa cliente que se conectará al servidor. Una vez que ambos estén conectados, el servidor enviara al cliente unos datos (una frase) que el cliente mostrará por pantalla, y finalmente terminarán ambos programas. El funcionamiento en líneas generales queda representado en la siguiente figura:



Programa cliente

El código del programa cliente, más simple que el del servidor, es el siguiente:

```
//includes necesarios para los sockets
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define INVALID_SOCKET -1

int main()
{
//declaracion de variables
	int socket_conn;//handle del socket utilizado
	struct sockaddr_in server_address;
	char address[]="127.0.0.1";
	int port=12000;
// Configuracion de la direccion IP de connexion al servidor
```

```
        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = inet_addr(address);
        server_address.sin_port = htons(port);
//creacion del socket
        socket_conn=socket(AF_INET, SOCK_STREAM,0);
//conexion
        int len= sizeof(server_address);
        connect(socket_conn,(struct sockaddr *) &server_address,len);
//comunicacion
        char cad[100];
        int length=100; //lee un máximo de 100 bytes
        int r=recv(socket_conn,cad,length,0);
        std::cout<<"Rec: "<<r<<" contenido: "<<cad<<std::endl;
//cierre del socket
        shutdown(socket_conn, SHUT_RDWR);
        close(socket_conn);
        socket_conn=INVALID_SOCKET;
        return 1;
}
```

A continuación se describe brevemente el programa:

Las primeras líneas son algunos #includes necesarios para el manejo de servicios de sockets. En el caso de querer utilizar los sockets en Windows, el fichero de cabecera y la librería con la que hay que enlazar se podrían establecer con las líneas:

```
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")
```

En las primeras líneas del main() se declaran las variables necesarias para el socket.

```
int socket_conn;//the socket used for the send-receive
struct sockaddr_in server_address;
```

La primera línea declara el descriptor o *handle* del socket (de tipo entero) que se utiliza tanto para la conexión como para la comunicación. Este número no es más que un identificador que nos dará el sistema operativo para poder indicar un socket determinado.

La segunda declaración declara una estructura de datos que sirve para almacenar la dirección IP y el número de puerto del servidor y la familia de protocolos que se utilizaran en la comunicación. Como ya se ha explicado, un ordenador puede comunicarse simultáneamente con muchos programas y muchos ordenadores. De alguna forma el puerto es como una extensión a la dirección IP. Haciendo una analogía, la dirección podría entenderse como la dirección de la escuela, mientras que el pueeto es un despacho o extensión determinado. La asignación de esta estructura a partir de la IP definida como una cadena de texto y el puerto definido como un entero se hace como sigue:

```
char address[]="127.0.0.1";
int port=12000;
server_address.sin_family = AF_INET;
```

```
server_address.sin_addr.s_addr = inet_addr(address);
server_address.sin_port = htons(port);
```

Nótese que la IP que utilizaremos será la "127.0.0.1". Esta IP es una IP especial que significa la máquina actual (dirección local). Realmente ejecutaremos las dos aplicaciones (cliente y servidor) en la misma máquina, utilizando la dirección local de la misma. No obstante esto se puede cambiar. Para ejecutar el servidor en una máquina que tiene la IP "192.168.1.13" por ejemplo, basta poner dicha dirección en ambos programas, ejecutar el servidor en esa máquina, y el cliente en cualquier otra (que sea capaz de enrutar mensajes hacia esa IP).

También es posible utilizar funciones del sistema operativo para intentar obtener una dirección IP en base al conocimiento de los nombres de los ordenadores.

Hay dos ficheros en Unix/Linux y en Windows que nos facilitan esta tarea, aunque hay que tener permisos de root para modificarlos. Estos ficheros serían el equivalente a una agenda de teléfonos, en uno tenemos apuntados el nombre de la empresa con su número de teléfono y en el otro fichero el nombre de la persona y su extensión (UPM-EUITI, tlfn 91 336 6799; Miguel Hernando, extensión 6878; ...)

- **/etc/hosts** en UNIX/LINUX o **\system32\drivers\etc\hosts**.en Windows: Esta es la agenda en la que tenemos las empresas y sus números de teléfono. En este fichero hay una lista de nombres de ordenadores conectados en red y la dirección IP de cada uno. Habitualmente en el lado del cliente se suele colocar el nombre del servidor y su dirección IP. Luego, desde el programa, se puede realizar una llamada a la función **gethostbyname()**, la cual si le pasamos el nombre del ordenador como una cadena de caracteres, devuelve una estructura de datos entre los que está la dirección IP.
- **/etc/services** o **system32\drivers\etc\services**: Este fichero es el equivalente a la agenda donde tenemos apuntados los distintos departamentos/personas de la empresa y sus números de extensión telefónica. En este fichero hay una lista de servicios disponibles, indicando nombre de servicio, número de servicio y tipo (ftp/udp). Tanto el servidor como el cliente deben tener en este fichero el servicio que están atendiendo/solicitando con el mismo número y tipo de servicio. El nombre puede ser distinto, igual que cada uno en su agenda pone el nombre que quiere, pero no es lo habitual. Desde programa, tanto cliente como servidor, deben hacer una llamada a la función **getservbyname()**, a la que pasándole el nombre del servicio, devuelve una estructura de datos entre los que está el número de puerto y el tipo.

Note que para iniciar un tipo de comunicación mediante un socket se requiere designar un puerto de comunicaciones. Esto significa que algunos puertos deben reservarse para éstos usos “bien conocidos”.

- 0-1023: Estos puertos pueden ser sólo enlazados por el sistema.
- 1024-5000: Puertos designados a aplicaciones conocidas.
- 5001-64K-1: Puertos disponibles para usos particulares.

A continuación se muestran algunos de los servicios registrados en ***service*** de una máquina Windows:

```
# Copyright (c) 1993-2004 Microsoft Corp.
#
# This file contains port numbers for well-known services defined by IANA
#
# Format:
#
# <service name>  <port number>/<protocol>  [aliases...]   [#<comment>]
#

echo                7/tcp
echo                7/udp
discard             9/tcp    sink null
discard             9/udp    sink null
systat             11/tcp    users                 #Active users
systat             11/udp    users                 #Active users
daytime            13/tcp
daytime            13/udp
qotd               17/tcp    quote                 #Quote of the day
qotd               17/udp    quote                 #Quote of the day
chargen            19/tcp    ttytst source         #Character generator
chargen            19/udp    ttytst source         #Character generator
ftp-data           20/tcp                          #FTP, data
ftp                21/tcp                          #FTP. control
ssh                22/tcp                          #SSH Remote Login Protocol
telnet             23/tcp
smtp               25/tcp    mail                  #Simple Mail Transfer Protocol
time               37/tcp    timserver
time               37/udp    timserver
rlp                39/udp    resource              #Resource Location Protocol
nameserver         42/tcp    name                  #Host Name Server
nameserver         42/udp    name                  #Host Name Server
nicname            43/tcp    whois
domain             53/tcp                          #Domain Name Server
domain             53/udp                          #Domain Name Server
bootps             67/udp    dhcps                 #Bootstrap Protocol Server
bootpc             68/udp    dhcpc                 #Bootstrap Protocol Client
tftp               69/udp                          #Trivial File Transfer
gopher             70/tcp
finger             79/tcp
http               80/tcp    www www-http          #World Wide Web
hosts2-ns          81/tcp                          #HOSTS2 Name Server
hosts2-ns          81/udp                          #HOSTS2 Name Server
kerberos           88/tcp    krb5 kerberos-sec     #Kerberos
kerberos           88/udp    krb5 kerberos-sec     #Kerberos
hostname          101/tcp    hostnames             #NIC Host Name Server
```

Se puede observar como el protocolo **http** del navegador aparece reflejado en el conocido puerto 80, o como el protocolo **smtp** se realiza en principio sobre el puerto 25.

A continuación se crea el socket, especificando la familia de protocolos (en este caso protocolo de Internet AF_INET) y el tipo de comunicación que se quiere emplear (orientada a la conexión=SOCK_STREAM, no orientada a la conexión=SOCK_DGRAM).

El primer parámetro es **AF_INET** o AF_UNIX para indicar si los clientes pueden estar en otros ordenadores distintos del servidor o van a correr en el mismo ordenador. **AF_INET** vale para los dos casos. **AF_UNIX** sólo para el caso de que el cliente corra en el mismo ordenador que el servidor, pero lo implementa de forma más eficiente. Si pusieramos **AF_UNIX**, el resto de las funciones varían ligeramente.
El segundo parámetro indica si el socket es orientado a conexión (**SOCK_STREAM**) TCP/IP o no lo es (**SOCK_DGRAM**) UDP/IP. De esta forma podremos hacer sockets de red o de Unix tanto orientados a conexión como no orientados a conexión.

El tercer parámetro indica el **protocolo** a emplear. Habitualmente se pone 0, lo cual significa que el sistema lo escogerá por defecto.

En nuestro caso vamos a utilizar comunicación orientada a la conexión y por tanto fiable.

```
//creacion del socket
socket_conn=socket(AF_INET, SOCK_STREAM,0);
```

Esta función generalmente no produce errores, aunque en algún caso podría hacerlo. Como regla general conviene comprobar su valor, que será igual a -1 (INVALID_SOCKET) si la función ha fallado.

A continuación se intenta la conexión con el socket especificado en la dirección del servidor. Como se observa en el ejemplo, esta dirección viene dada por una estructura que ya contiene en el formato propio de la red la dirección, el puerto y el protocolo de red. Los campos que obligatoriamente hay que rellenar son los reflejados en el ejemplo:

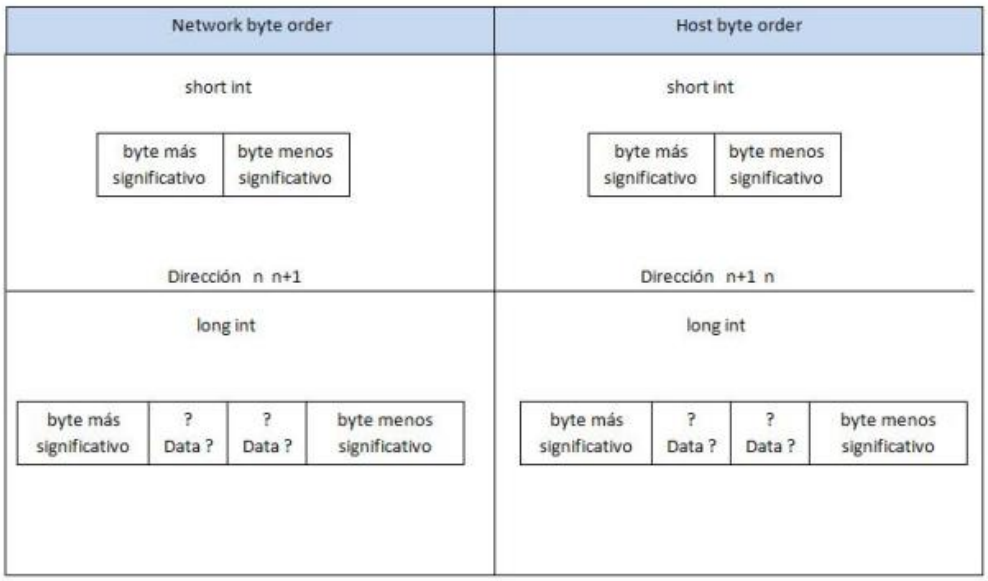
```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr(address);
server_address.sin_port = htons(port);
```

- **sin_family** es el tipo de conexión (por red o interna), igual que el primer parámetro de **socket()**.
- **sin_port** es el número de puerto correspondiente al servicio que podriamos haber obtenido con **getservbyname()**. El valor estaría en en el campo **s_port** de Puerto de la estructura devuelta por getserbyname(). En el ejemplo sin embargo se especifica el número de puerto directamente, pero convirtiéndolo al formato Finalmente **sin_addr.s_addr** es la dirección del cliente al que queremos atender

Para entender el uso de htons y **inet_addr**, es necesario indicar como es importante que en la red se envien los datos con un formato claro. Las máquinas en función de la arquitectura y el procesador utilizan un modo diferente de representar los números. **inet_addr,** además de conseguir que la dirección quede representada con un orden de bytes correcto, lo que logra es traducir una dirección IP dada como una secuencia de caracteres en un long int de 32 bytes.

Network byte order y Host byte order son dos formas en las que el sistema puede almacenar los datos en memoria. Está relacionado con el orden en que se almacenan los bytes en la memoria RAM.

Si al almacenar un short int (2 bytes) o un long int (4 bytes) en RAM, y en la posición más alta se almacena el byte menos significativo, entonces está en *network byte order*, caso contrario es host byte order.



Cuando enviamos los datos por la red deben ir en un orden especificado, sino enviaríamos todos los datos al revés. Lo mismo sucede cuando recibimos datos de la red, debemos ordenarlos al orden que utiliza nuestro sistema. Debemos cumplir las siguientes reglas:

- Todos los bytes que se transmiten hacia la red, sean números IP o datos, deben estar en network byte order.
- Todos los datos que se reciben de la red, deben convertirse a host byte order.

Para realizar estas conversiones utilizamos las funciones que se describen a continuación.

- **htons()** - host to network short - convierte un short int de host byte order a network byte order (es el caso del número de servicio o puerto).
- **htonl()** - host to network long - convierte un long int de host byte order a network byte order.
- **ntohs()** - network to host short - convierte un short int de network byte order a host byte order.
- **ntohl()** - network to host long - convierte un long int de network byte order a host byte order.

Puede ser que el sistema donde se esté programando almacene los datos en *network byte order* y no haga falta realizar ninguna conversión, pero si tratamos de compilar el mismo código fuente en otra plataforma host byte order no funcionará. Como conclusión, para que nuestro código fuente sea portable se debe utilizar siempre las funciones de conversión.

```
   //conexion
   int len= sizeof(server_address);
   connect(socket_conn,(struct sockaddr *) &server_address,len);
```

Esta función connect() fallará si no esta el servidor preparado por algún motivo (lo que sucede muy a menudo). Por lo tanto es más que conveniente comprobar el valor de retorno de connect() para actuar en consecuencia. Se podría hacer algo como:

```
   if(connect(socket_conn,(struct sockaddr *)
&server_address,len)!=0)
   {
   std::cout<<"Client could not connect"<<std::endl;
   return -1;
   }
```

Si la conexión se realiza correctamente, el socket ya esta preparado para enviar y recibir información.

En este caso hemos decidido que va a ser el servidor el que envía datos al cliente. Esto es un convenio entre el cliente y el servidor, que adopta el programador cuando diseña e implementa el sistema.

Como el cliente va a recibir información, utilizamos la función de recepción. En esta función, se le suministra un buffer en el que guarda la información y el número de bytes máximo que se espera recibir. La función recv() se bloquea hasta que el servidor envíe alguna información.

Dicha información puede ser menor que el tamaño máximo suministrado. El valor de retorno de la función recv() es el numero de bytes recibidos.

```
//comunicacion
char cad[100];
int length=100; //read a maximum of 100 bytes
int r=recv(socket_conn,cad,length,0);
std::cout<<"Rec: "<<r<<" contenido: "<<cad<<std::endl;
```

Por ultimo se cierra la comunicación y se cierra el socket.

```
shutdown(socket_conn, SHUT_RDWR);
close(socket_conn);
socket_conn=INVALID_SOCKET;
```

Servidor

El código del programa servidor es algo más Complejo, ya que debe realizar más tareas. La principal característica es que se utilizan dos sockets diferentes, uno para la conexión y otro para la comunicación. El servidor comienza enlazando el socket de conexión a una dirección IP y un puerto (siendo la IP la de la máquina en la que corre el servidor), escuchando en ese puerto y quedando a la espera “Accept” de una conexión, en estado de bloqueo. Cuando el cliente se conecta, el “Accept” se desbloquea y devuelve un nuevo socket, que es por el que realmente se envían y reciben datos.

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <iostream>
#define INVALID_SOCKET -1

int main()
{
	int socket_conn=INVALID_SOCKET;//used for communication
	int socket_server=INVALID_SOCKET;//used for connection
	struct sockaddr_in server_address;
	struct sockaddr_in client_address;
// Configuracion de la direccion del servidor
	char address[]="127.0.0.1";
	int port=12000;
	server_address.sin_family = AF_INET;
	server_address.sin_addr.s_addr = inet_addr(address);
	server_address.sin_port = htons(port);
//creacion del socket servidor y escucha
	socket_server = socket(AF_INET, SOCK_STREAM, 0);
	int len = sizeof(server_address);
	int on=1; //configuracion del socket para reusar direcciones
	setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(on));
//escucha
```

```
        bind(socket_server,(struct sockaddr *) &server_address,len);
// Damos como maximo 5 clientes conectados simultáneamente.
        listen(socket_server,5);
//aceptacion de cliente (bloquea hasta la conexion)
        unsigned int leng = sizeof(client_address);
        socket_conn = accept(socket_server,
            (struct sockaddr *)&client_address, &leng);
//notese que el envio se hace por el socket de communicacion
        char cad[]="Hola Mundo";
        int length=sizeof(cad);
        send(socket_conn, cad, length,0);
//cierre de los dos sockets, el servidor y el de comunicacion
        shutdown(socket_conn, SHUT_RDWR);
        close(socket_conn);
        socket_conn=INVALID_SOCKET;
        shutdown(socket_server, SHUT_RDWR);
        close(socket_server);
        socket_server=INVALID_SOCKET;
        return 1;
}
```

Hasta la creación del socket del servidor, el programa es similar al cliente, quitando la excepción de que se declaran los dos sockets, el de conexión y el de comunicación. La primera diferencia son las líneas:

```
//configuracion del socket para reusar direcciones
int on=1;
setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(on));
```

Estas líneas se utilizan para que el servidor sea capaz de re-usar la dirección y el puerto que han quedado abiertos sin ser cerrados correctamente en una ejecución anterior. Cuando esto sucede, el sistema operativo deja la dirección del socket reservada y por tanto un intento de utilizarla para un servidor acaba en fallo. Con estas líneas podemos configurar y habilitar que se re-usen las direcciones previas.

La segunda diferencia es que en vez de intentar la conexión con connect(), el servidor debe establecer primero en que dirección va a estar escuchando su socket de conexión, lo que se establece con las líneas:

```
int len = sizeof(server_address);
bind(socket_server,(struct sockaddr *) &server_address,len);
// Damos como maximo una cola de 5 conexiones.
listen(socket_server,5);
```

La función **bind()** enlaza el socket de conexión con la IP y el puerto establecidos anteriormente.

La llamada a **bind()** lleva tres parámetros:

- **Handle del socket** que hemos abierto

- Puntero a la **estructura Direccion.** A diferencia de lo que ocurría con el cliente, en el caso del servidor se especifica la dirección del cliente al que queremos atender. En este campo introducimos INADDR_ANY, si queremos atender a un cliente cualquiera. La estructura admitida por este parámetro es general y valida para cualquier tipo de socket y es del tipo **struct sockaddr**. Cada socket concreto lleva su propia estructura. Los **AF_INET** como este caso llevan **struct sockaddr_in**, los **AF_UNIX** llevan la estructura **struct sockaddr_un**. Por eso, a la hora de pasar el parámetro, debemos hacer un "cast" al tipo **struct sockaddr**.

- **Longitud** de la estructura Direccion.

Esta función también es susceptible de fallo. El fallo más común es cuando se intenta enlazar el socket con una dirección y puerto que ya están ocupados por otro socket. En este caso la función devolverá -1, indicando el error. A veces es posible que si no se cierra correctamente un socket (por ejemplo, si el programa finaliza bruscamente), el SO piense que dicho puerto esta ocupado, y al volver a ejecutar el programa, el bind() falle, no teniendo sentido continuar la ejecución. La gestión básica de este error podría ser:

```
   if(0!=bind(socket_server,(struct sockaddr *)
&server_address,len))
   {
         std::cout<<"Fallo en el Bind()"<<std::endl;
         return -1;
   }
```

La función **listen()** permite definir cuantas peticiones de conexión al servidor serán encoladas por el sistema. Nótese que esto no significa que realmente se atiendan las peticiones de conexión. Es el usuario a través de la función accept() el que acepta una conexión. El numero de conexiones dependerá de cuantas veces ejecute el programa dicho accept().

```
   //aceptacion de cliente (bloquea hasta la conexion)
   unsigned int leng = sizeof(client_address);
   socket_conn = accept(socket_server,
   (struct sockaddr *)&client_address, &leng);
```

Lo más importante del **accept()** es que en su modo normal bloquea el programa hasta que realmente se realiza la conexión por parte del cliente. A esta función se le suministra el socket de conexión, y devuelve el socket que realmente se utilizará para la comunicación. Si algo falla en la conexión, la función devolverá -1, lo que corresponde a nuestra definición de socket invalido INVALID_SOCKET, lo que podemos comprobar:

```
if(socket_conn==INVALID_SOCKET)
{
      std::cout<<"Error en el accept"<<std::endl;
      return -1;
}
```

Una vez que se ha realizado la conexión, la comunicación se hace por el nuevo socket, utilizando las mismas funciones de envío y recepción que se podrían usar en el cliente. Notese que un modo de atender a varios clientes, es ir haciendo accepts consecutivos y creando sockets de comunicación hasta alcanzar un número determinado, momento en el que ya trabajamos con las comunicaciones a los distintos clientes.

En el ejemplo actual, realmente sólo atendemos a un cliente cada vez y por convenio hemos establecido que será el servidor el que envía un primer mensaje al cliente. El código siguiente envía el mensaje "Hola Mundo" por el socket:

```
char cad[]="Hola Mundo";
int length=sizeof(cad);
//notese que el envio se hace por el socket de communicacion
send(socket_conn, cad, length,0);
```

La función send() también puede fallar, si el socket no esta correctamente conectado (se ha desconectado el cliente por ejemplo). La función devuelve el número de bytes enviados correctamente o -1 en caso de error. Típicamente, si la conexión es buena, la función devolverá como retorno un valor igual a "length", aunque también es posible que no consiga enviar todos los datos que se le ha solicitado. Una solución completa debe contemplar esta posibilidad y reenviar los datos que no han sido enviados. No obstante y por simplicidad, realizamos ahora una gestión sencilla de este posible error:

```
if(lenght!=send(socket_conn, cad, length,0))
{
      std::cout<<"Fallo en el send()"<<std::endl;
      return -1;
}
```

El cierre de los sockets se realiza de la misma manera que en el cliente, exceptuando que se deben cerrar correctamente los dos sockets, el de conexión y el de comunicación. La salida por pantalla al ejecutar las aplicaciones (primero arrancar el servidor y luego el cliente) debería ser (en el lado del cliente):

```
Rec: 11 contenido: Hola Mundo
```

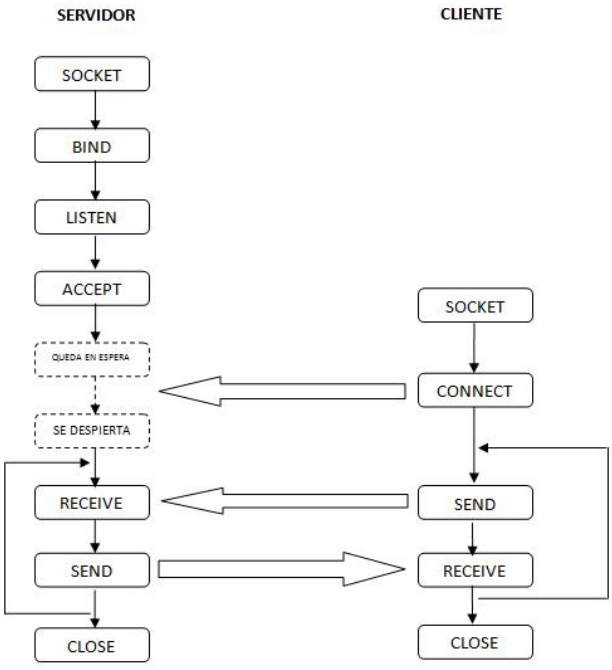
Nótese que los bytes recibidos son 11 porque incluyen el carácter nulo '\0' de final de la cadena.

11.3. Modelos de conexión Cliente-Servidor

Existen diversas formas de atender las conexiones de clientes en un servidor. Sin embargo dado que la programación concurrente no la trataremos en este curso, simplemente se exponen a continuación para entender la importancia de estas estructuras. La primera de ellas responde al modo visto en el ejemplo anterios, la segunda requiere de el uso de threads (hilos de ejecución de un proceso), y la tercera es la estructura habitual con comunicación no orientada a la conexión.

Modelo Simple Cliente-Servidor: Conexión Stream

En el siguiente gráfico se muestran los procesos que se ejecutan cuando se realiza una conexión Simple Cliente-Servidor:



En el servidor:

- Se crea un socket de comunicación mediante la función socket().
- El socket está formado por una dirección ip + un puerto de conexión, para ello hay que asociar esa información al socket mediante la función bind().
- La parte servidor de la aplicación debe establecer una cola de conexiones entrantes para atender a los clientes según el orden de llegada, esto se hace mediante la función listen().
- Ahora el servidor deberá atender las conexiones entrantes por el puerto que establecimos mediante la función accept().
- Tras establecer una comunicación entre un cliente y un servidor se enviarán datos mediante la función send() y se recibirán mediante la función recv().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación.

En el cliente:

- Se crea un socket de comunicación mediante la función socket().
- El servidor está aceptando conexiones entrantes, así que nos conectamos al servidor mediante la función connect().
- Recibimos y enviamos datos, pues ya estamos en contacto con el equipo remoto, mediante las funciones recv() y send().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación.

Modelo Concurrente Cliente-Servidor: Conexión Stream

Para entender elmodelo concurrente, es preciso al menos tener un primer concepto de lo que es un hilo de ejecución o *thread*:

Concepto de thread

Un thread es la unidad básica de ejecución en la mayoría de los S.O. . Cualquier programa que se ejecute consta de, al menos, un thread.

Un thread se puede considerar como la agrupación de un trozo de programa junto con el conjunto de registros del procesador que utiliza y una pila de máquina. El conjunto de los registros y de la pila de cada thread se denomina **contexto**. Como sabemos, en un Sistema Operativo multitarea, la CPU se reparte entre cada programa a ejecutar. Para ser más precisos, el S.O. reparte la CPU entre todos los threads a ejecutar en cada momento (pues un programa puede contener varios threads), simplemente adueñandose de esta y saltando al

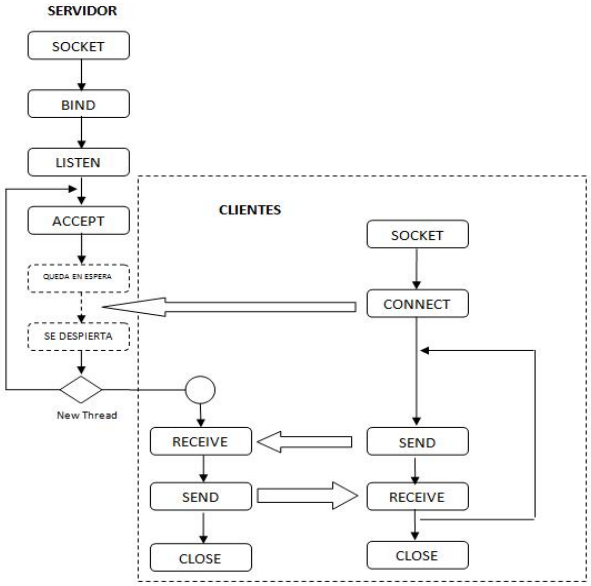
siguiente. Sin embargo, esta conmutación no se puede hacer de cualquier manera. Cada vez que el S.O. se adueña de la CPU para cedersela a otro *thread*, los registros y la pila (o sea, el **contexto** del hilo) contienen unos valores determinados. Por eso, el S.O. guarda todos esos datos en cada cambio, de modo que al volver a conmutar al thread inicial, pueda restaurar el contexto inicial. La mayoría de los S.O. multitarea son de tipo **preemptivo**, lo que significa que la CPU puede ser arrebatada en cualquier momento. Esto significa que un *thread* no puede saber cuando se le va a arrebatar la CPU, por lo que no puede guardar los registros ni la pila de forma 'voluntaria'.

La diferencia más importante entre hilo y proceso, consiste en que dos procesos distintos en un sistema multitarea, no comparten recursos ni siquiera el mapa de memoria, de tal forma que la comunicación entre los mismos se realiza por medio de mecanismos de comunicación dados por el sistema operativo (memorias compartidas, pipes, mensajes…). Son como dos instancias diferentes del mismo tipo. Sin embargo los threads, salvo la pila y el contexto de la CPU, comparten recursos, memoria y mapa de direcciónes y por tanto las zonas de datos son comunes para todos los threads de un mismo **proceso**. Una de las ventajas que tienen por tanto es que la conmutación entre hilos de un mismo programa (proceso) es muy rápida, frente al cambio de contexto entre dos procesos.

Por otro lado, debemos recordar que cada thread se ejecuta de forma absolutamente independiente. De hecho, cada uno trabaja como si tuviese un microprocesador para el solo. Esto significa que si tenemos una zona de datos compartida entre varios threads de modo que puedan intercambiar información entre ellos, es necesario usar algún sistema de sincronización para evitar que uno de ellos acceda a un grupo de datos que pueden estar a medio actualizar por otro thread. Estos problemas clásicos en los sistemas concurrentes quedan fuera del enfoque del curso, pero baste aquí por lo menos mencionarlo.

Modelo concurrente cliente-servidor

En el siguiente gráfico se muestran los procesos que se ejecutan cuando se realiza una conexión Concurrente Cliente-Servidor:



En el servidor:

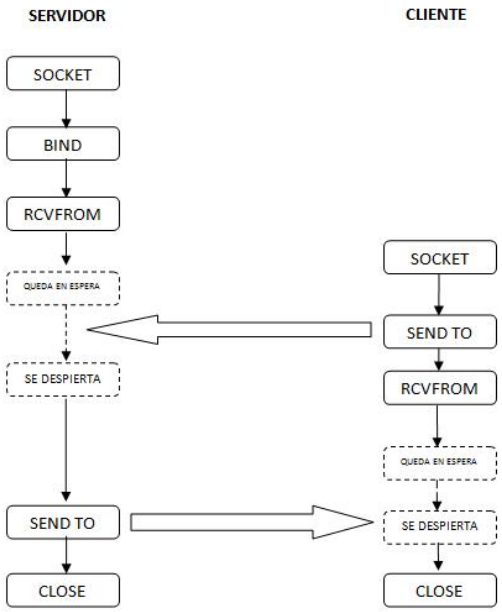
- Se crea un socket de comunicación mediante la función socket().
- El socket está formado por una dirección ip + un puerto de conexión, para ello hay que asociar esa información al socket mediante la función bind().
- La parte servidor de la aplicación debe establecer una cola de conexiones entrantes para atender a los clientes según el orden de llegada, esto se hace mediante la función listen().
- Ahora el servidor deberá atender las conexiones entrantes por el puerto que establecimos mediante la función accept(). La única diferencia con el caso anterior es que una vez aceptada una conexión se creará un nuevo hilo de ejecución (thread) encargado de dicha comunicación y en el código principal se volverán a aceptar conexiones entrantes. Esto quiere decir que por cada nueva conexión existirá un nuevo thread o hilo de ejecución.
- Tras establecer una comunicación entre un cliente y un servidor se enviarán datos mediante la función send() y se recibirán mediante la función recv().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación y finalizar su thread correspondiente.

En el cliente:

- Se crea un socket de comunicación mediante la función socket().
- El servidor está aceptando conexiones entrantes, así que nos conectamos al servidor mediante la función connect().
- Recibimos y enviamos datos, pues ya estamos en contacto con el equipo remoto, mediante las funciones recv() y send().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación.

Modelo Cliente-Servidor: Conexión Datagram

En el siguiente gráfico se muestran los procesos que se ejecutan cuando se realiza una conexión Cliente-Servidor mediante Datagrams:



En el servidor:

- Se crea un socket de comunicación mediante la función socket().
- El socket está formado por una dirección ip + un puerto de conexión, para ello hay que asociar esa información al socket mediante la función bind().
- Tras establecer una puerto de comunicación queda a las espera de datos entrantes mediante la función recvfrom() o envía datos a algún cliente mediante sendto().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación y finalizar su thread correspondiente.

En el cliente:

- Se crea un socket de comunicación mediante la función socket().
- El servidor ya ha establecido un puerto de comunicación por lo que podemos enviarle datos mediante sendto() o esperar por algún dato que el servidor desee enviarnos mediante recvfrom().
- Al finalizar la comunicación deberemos cerrar el socket de comunicación.

Anexo I. Soluciones a los ejercicios.

Capítulo 2

EJERCICIO 2.1

EJERCICIO 2.2

```
#include <math.h>

float norma(float *v, int n=3, bool bnormaliza=false)
{
	float modulo=0;
	int i;
	for(i=0;i<n;i++)modulo+=v[i]*v[i];
	modulo=(float)sqrt((float)modulo);
	if((bnormaliza)&&(modulo>0.0001F)){
		for(i=0;i<n;i++)v[i]/=modulo;
		}
	return modulo;
}
```

EJERCICIO 2.3

```
   a=3  b=5 c=6
```

EJERCICIO 2.4

```
1 5 7 11 13 17 19 23 25 29
```

EJERCICIO 2.5

```
OK      nota1=bien;
MAL     nota1=1;
OK      nota1=(notas)1;
MAL     nota1=suspenso+1;
MAL     nota1=suspenso+aprobado;
OK      var=5+bien;
OK      var=bien+5;
OK      var=notable+bien;
OK      var=nota1+bien;
MAL     nota1++;
OK      for(var=suspenso;var<sobresaliente;var++)
              printf("%d",var);
MAL     for(nota2=suspenso;nota2<sobresaliente;nota2++)
              printf("%d",(int)nota2);
```

EJERCICIO 2.6

```
Valores: 18 14 17
```