Escuela Técnica Superior de Ingeniería Naval y Oceánica

Universidad Politécnica de Cartagena

Principios para la programación en C++ de software navales basados en NURBS

C++ Programming Principles for NURBS-based naval software

Autor: Juan Antonio Soler Vasco

Director: Dr. Sergio Amat Plata

Junio 2013

Ficha Proyecto Fin de Carrera

Autor	D. Juan Antonio Soler Vasco Ingeniero Técnico Naval, Esp. Estructuras Marinas
E-mail del Autor	juanantoniosolervasco@gmail.com
Director	Dr. Sergio Amat Plata Doctor en Matemáticas Catedrático de la Universidad Politécnica de Cartagena
E-mail del Director	Sergio.amat@upct.es
Título del PFC	Principios para la programación en C++ de software navales basados en NURBS
PFC name	C++ Programming Principles for NURBS-based naval software

Resumen

Actualmente existe un amplio catálogo de programas informáticos navales comercializados. Dichos programas, cada uno orientado a distintas etapas de diseño y construcción del buque, es difícil que se adecúe perfectamente a las necesidades del proyectista, teniendo que recurrir comúnmente a varios programas distintos según la fase de proyecto.

El presente documento, permite adquirir los conocimientos básicos para el desarrollo de un programa informático naval, presentado la enorme ventaja de obtener el máximo rendimiento de un programa comercial, orientando éste a nuestras necesidades mediante herramientas de desarrollo de software muchas veces gratuitas.

Titulación	Ingeniería Naval y Oceánica
Intensificación	-
Departamento	Dpto. de Matemática Aplicada y Estadística
Fecha de presentación	Julio 2013

Agradecimientos:

Agradecer a mi director de proyecto, Dr. Sergio Amat Plata, por su labor incuestionable a la hora de mostrar su ayuda para el desarrollo de éste documento. Su rigurosidad y seriedad han sido imprescindibles para finalizar mi carrera.

Agradecer a todo aquellos profesores que han demostrado su interés por mostrar sus conocimientos de forma tan profesional, inculcándome el valor del trabajo bien hecho.

Dedicatoria:

A todos esos grandes profesores que desde pequeño han pasado por mi vida (D. Antonio Zamora Barrancos, D. José Juan López Espín, etc). Sin ellos yo no habría escrito estas líneas.

A todos mis amigos y mi familia. A mi novia no le pienso dedicar un texto como éste, sería, a mi entender, de las peores formas de demostrarle que la quiero. La invitaré, eso sí, a una buena y merecida cena.

Contenido

1.Introducción a las curvas y superficies NURBS

Introducción

El propósito de este capítulo la definición de curvas y superficies B-Splines Racionales No Uniformes (NURBS). En él se presentan las propiedades generales de las curvas y superficies NURBS, así como de las derivadas de estas.

Definición y propiedades de las curvas NURBS

Una curva NURBS de grado p-ésimo se define mediante:

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i P_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i} \quad a \leq u \leq b \quad (Ec.1)$$

Donde los $\{P_i\}$ son los puntos de control (formando un polígono de control), los $\{w_i\}$ son los pesos, y los $\{N_{i,p}(u)\}$son las funciones base B-spline de grado p-ésimo definidas en el vector nudo no periódico (y no uniforme):

$$U = \{\underbrace{a, \dots, a}_{p+1}, u_{p+1}, \dots, u_{m-p-1}, \underbrace{b, \dots, b}_{p+1}\}$$

A no ser que comenzásemos de otro modo, se asume que a=0, b=1, y $w_i > 0$ para todo i.
Con

$$R_{i,p}(u) = \frac{N_{i,p}(u) w_i}{\sum_{j=0}^{n} N_{i,p}(u) w_j} \quad (Ec.2)$$

nos permite reescribir la ec. 1 de la forma

$$C(u) = \sum_{i=0}^{n} R_{i,p}(u) P_i \quad (Ec.3)$$

Las $\{R_{i,p}(u)\}$ son las funciones base racionales, que son funciones racionales por tramos en $u \in [0,1]$. Las $\{R_{i,p}(u)\}$ tienen las siguientes propiedades derivadas de la ec. 2 y las correspondientes propiedades de $N_{i,p}(u)$:

P.1 No negatividad: $R_{i,p} \geq 0$ para todos los i, p y $u \in [0,1]$;

P.2 Partición de la unidad:

$$\sum_{i=0}^{n} R_{i,p}(u) = 1$$

para todo uϵ [0,1];

P.3 $R_{o,p}$ (0) = $R_{n,p}$ (1) =1;

P.4 Para p>0, todos los $R_{i,p}$ (u) alcanzan exactamente un máximo en el intervalo uϵ [0,1];

P.5 Soporte local: $R_{i,p}(u)$ = 0 para u no contenido en [u_i, u_{i+p+1}). Además, en cualquier espacio entre nudos dado, como mucho p+1 de los $R_{i,p}(u)$ son distintos de cero (en general, $R_{i-p,p}(u)$,…,$R_{i,p}(u)$ son no nulos en [u_i,u_{i+1}));

P.6 Todas las derivadas de $R_{i,p}(u)$ existen en el interior de un espacio entre puntos, donde hay una función racional con denominador no nulo. En un punto, $R_{i,p}(u)$ es p-k veces continuamente diferenciable, donde k es la multiplicidad del punto;

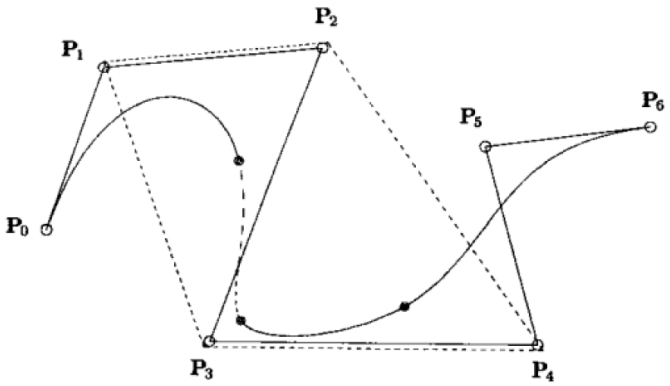
P.7 Si w_i = 1 para todo i, entonces $R_{i,p}$ (u) = $N_{i,p}$ (u) para todo i. Por ejemplo, los $N_{i,p}$ (u) son casos especiales de $R_{i,p}$ (u). De hecho, para cualquier a≠0, si w_i = a para todo i, entonces $R_{i,p}(u)=N_{i,p}(u)$ para todo i.

Las propiedades P.1 a P.7 arrojan importantes características geométricas de las curvas NURBS:

P.8 C(0) = P_o y C(1) = P_n; esto se obtiene de P.3;

P.9 Invarianza afín: Se aplica una transformación afín a la curva mediante la aplicación a los puntos de control; las curvas NURBS son además invariantes bajo la perspectiva de las proyecciones, un dato a tener en cuenta en los gráficos para ordenador.

P.10 Fuerte propiedad de envoltura convexa: Si uϵ[u_i, u_{i+1}), entonces C (u) permanece en la envoltura convexa de los puntos de control Pi-p,…,Pi (ver Fig. 1 donde C (u) para uϵ [1/4, 1/2) (segmento discontinuo) es contenido en la envoltura convexa de {P_1, P_2, P_3, P_4}, el área discontinua); esto se obtiene de P1, P2 y P5.



(a)

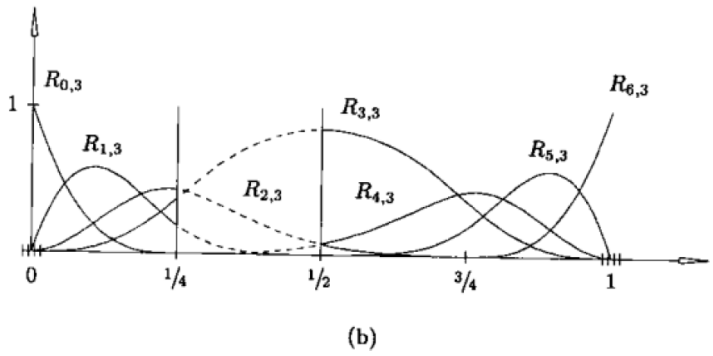


Figura 1: U = {0,0,0,0,1/4,1/2,3/4,1,1,1,1} y $\{w_0,\ldots,w_6\}$ = {1,1,1,3,1,1,1}.
a) Una curva NURBS cúbica; b) funciones base asociadas.

P.11 C (u) es infinitamente diferenciable en el interior de la luz entre puntos y es p-k veces diferenciable en un punto de multiplicidad k;

P.12 Propiedad de disminución de las variaciones: ningún plano tiene más intersecciones con la curva que con el polígono de control (reemplazar la palabra "plano" por "línea" en el caso de curvas bidimensionales);

P.13 Una curva NURBS sin puntos interiores es una curva racional de Bézier, donde $N_{i,p}$ (u) se reduce a $B_{i,n}$ (u). Esto, junto con P.7, implica que las curvas NURBS contienen B-splines no racionales y curvas de Bézier racionales e irracionales como casos especiales;

P.14 Aproximación local: si el punto de control Pi se mueve, o el peso w_i se cambia, esto afecta sólo a ese trozo de la curva en el intervalo $u\epsilon[u_i, u_{i+p+1})$; esto se deriva de P.5

La propiedad P.14 es muy importante para el diseño interactivo de formas. Usando curvas NURBS, podemos usar tanto movimientos de puntos de control como modificación de pesos para obtener el control local de la forma. Las figuras 2 a 6 muestran los efectos de modificar un único peso. Cualitativamente el efecto es: Asumimos que $u\epsilon[u_i,u_{i+p+1})$; entonces si w_i aumenta (disminuye), el punto C(u) se acerca (se aleja) a P_i, y por tanto, la curva se acerca a (se aleja de) Pi. Además, el movimiento de C(u) para un u fijado es a lo largo de una línea recta (fig. 6).

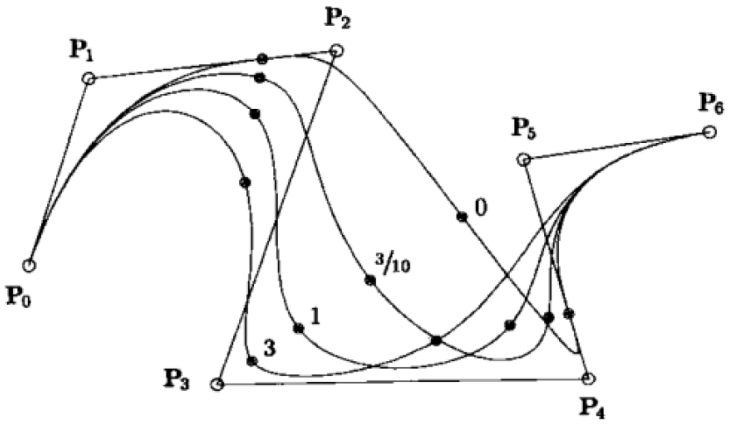


Figura 2: Curvas B-splines cúbicas racionales, con variaciones de w_3.

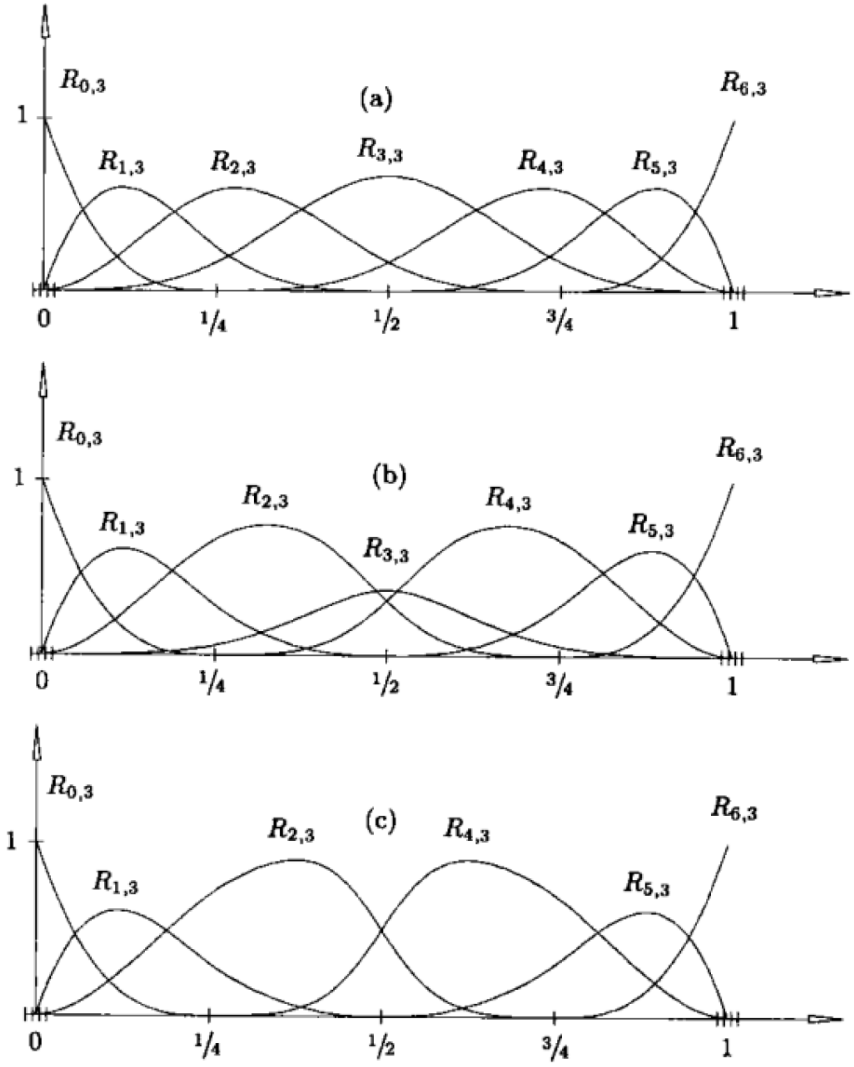


Figura 3: Las funciones bases cúbicas para las curvas de la fig. 2
a)$w_3 = 1$; b) $w_3 = 3/10$; c) $w_3 = 0$.

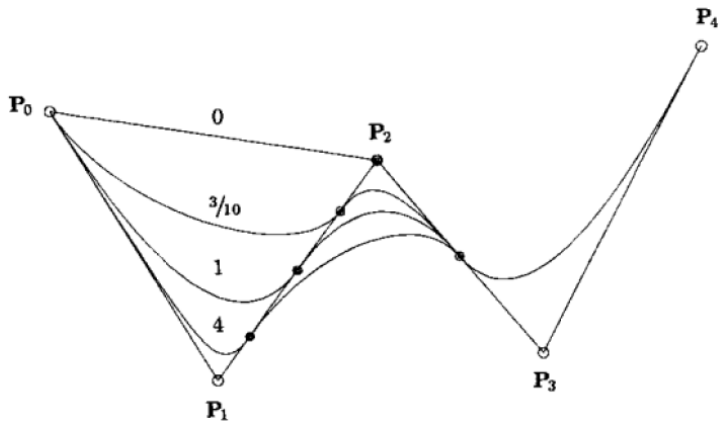


Figura 4: Curvas racionales cuadráticas, con variaciones de w_1.

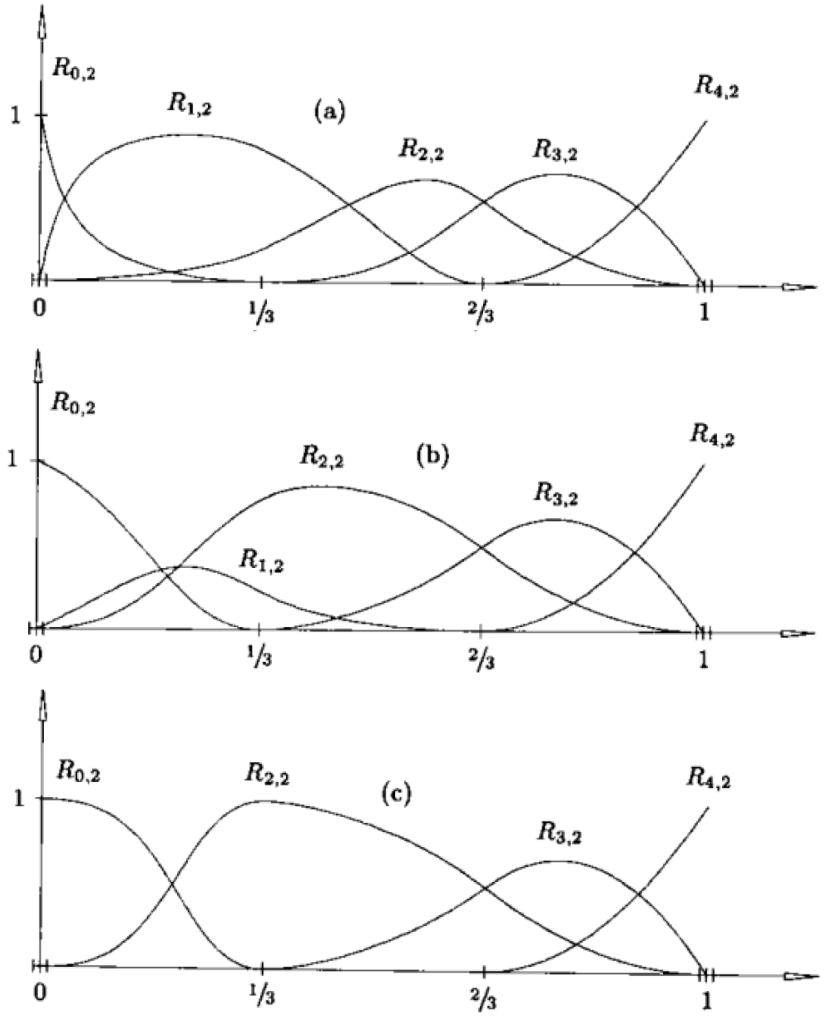


Figura 5: Las funciones bases cuadráticas para las curvas de la fig. 4 a)$w_1 = 4$; b) $w_1 = 3/10$; c) $w_1 = 0$.

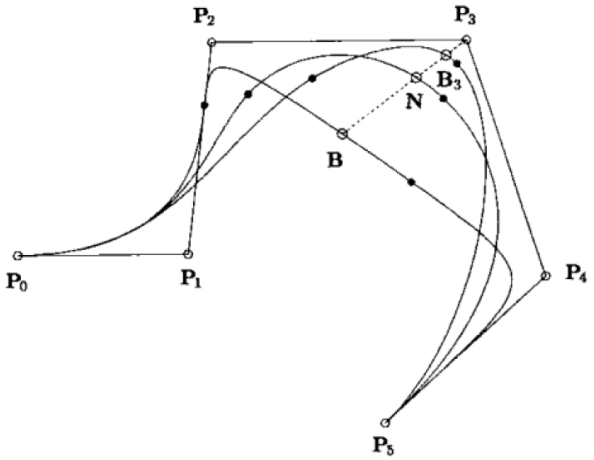


Figura 6: Modificación del peso w_3.

En la fig.6, u es fijo y w_3 cambia. Sea:

$$B = C\ (u;\ w_3 = 0) \qquad \textit{(Ec. 4)}$$

$$N = C\ (u;\ w_3 = 1)$$

Entonces, la línea recta definida por B y N pasa a través de P_3, y por un arbitrario $0 < w3 < \infty$, $B_3 = C(u; w3)$ permanece en este segmento de línea entre B y P3.

Como en el caso de las curvas racionales de Bézier, las coordenadas homogéneas ofrecen un método eficiente para representar curvas NURBS. Para un conjunto dado de puntos de control, $\{P_i\}$, y unos pesos, $\{w_i\}$, los puntos de control sometidos a los pesos serán:

$$P_i^w = (w_i x_i, w_i y_i, w_i, z_i w_i)$$

Entonces se define la B-spline irracional (polinómica por segmentos) en un espacio de cuatro dimensiones como

$$C^w(u) = \sum_{i=0}^{n} N_{i,p}(u) P_i^w \qquad (Ec. 5)$$

Aplicando el mapa de perspectiva, H, a C^w (u), se obtiene la correspondiente curva B-spline racional (racional por segmentos en un espacio tridimensional).

$$C(u) = H\{C^w(u)\} = H\left\{\sum_{i=0}^{n} N_{i,p}(u) P_i^w\right\}$$

Nos referimos tanto a C^w (u) como a C(u) como curvas NURBS, a pesar de que hablando estrictamente, C^w (u) no es una curva racional.

Derivadas de una curva NURBS

Las derivadas de funciones racionales son complicadas, incluyendo denominadores con altos exponentes. En esta sección desarrollaremos fórmulas que expresen las derivadas de C(u) en términos de las derivadas de C^w (u).

Sea:

$$C(u) = \frac{w(u)C(u)}{w(u)} = \frac{A(u)}{w(u)} \qquad (Ec. 6)$$

donde A(u) es la función vectorial cuyas coordenadas son las tres primeras coordenadas de C^w (u). A(u) es el numerador de la ec. 1.16). Entonces

$$C'(u) = \frac{w(u)A'(u) - w'(u)A(u)}{w(u)^2} = \frac{w(u)A'(u) - w'(u)w(u)C(u)}{w(u)^2}$$

$$= \frac{A'(u) - w'(u)C(u)}{w(u)} \qquad (Ec. 7)$$

Como A(u) y w(u) representan las coordenadas de C^w (u), obtenemos las primeras derivadas. Obtenemos derivadas de mayor orden usando la regla de Leibnitz

$$A^{(k)}(u) = \left(w(u)C(u)\right)^{(k)} = \sum_{i=0}^{k} \binom{k}{i} w^{(i)}(u)C^{(k-i)}(u)$$
$$= w(u)C^{(k)}(u) + \sum_{i=1}^{k} \binom{k}{i} w^{(i)}(u)C^{(k-i)}(u)$$

De donde se obtiene

$$C^{(k)}(u) = \frac{A^{(k)}(u) - \sum_{i=1}^{k} \binom{k}{i} w^{(i)}(u)C^{(k-i)}(u)}{w(u)} \quad (Ec.8)$$

La ec. 8 da la derivada k-ésima de C(u) en términos de la derivada k-ésima de A(u), y la primera a través de las derivadas (k-1)-ésimas de C(u) y w(u).

Si derivamos las expresiones para la primera derivada de una curva NURBS en sus puntos finales (u = 0, u =1), obtenemos:

$$A'(0) = \frac{p}{u_{p+1}}(w_1P_1 - w_0P_0)$$
$$w'(0) = \frac{p}{u_{p+1}}(w_1 - w_0)$$

y de la ec. 7

$$C'(0) = \frac{\frac{p}{u_{p+1}}(w_1P_1 - w_0P_0) - \frac{p}{u_{p+1}}(w_1 - w_0)P_0}{w_0}$$

de donde se colige

$$C'(0) = \frac{p}{u_{p+1}}\frac{w_1}{w_0}(P_1 - P_0) \quad (Ec.9)$$

Análogamente:

$$C'(1) = \frac{p}{1 - u_{m-p-1}}\frac{w_{n-1}}{w_n}(P_n - P_{n-1}) \quad (Ec.10)$$

La fig. 7 muestra la primera, segunda y tercera derivadas de una curva NURBS cúbica. Los vectores de la derivada han sido escalados por 0.4, 0.08 y 0.03 respectivamente.

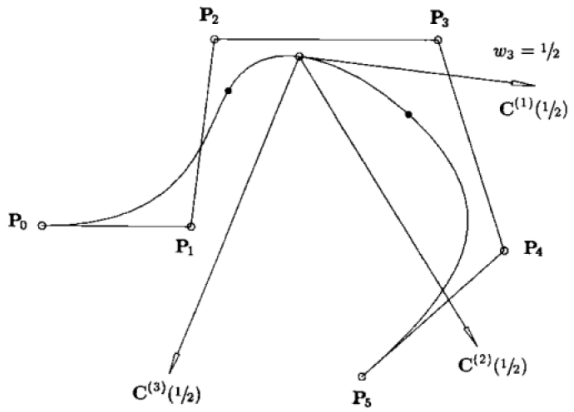


Figura 7: Primera, segunda y tercera derivadas de una curva NURBS cúbica calculadas en u = 1/2, con w_3=1/2 y w_i = 1, i ≠3

Definición y propiedades de las superficies NURBS

Una superficie de grado p en la dirección u, y grado q en la dirección v es una función racional por tramos vectorial en dos variables de la forma

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,p}(v)w_{i,j}P_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,p}(v)w_{i,j}} \quad 0 \le u,v \le 1 \qquad (Ec.\,11)$$

Los $\{P_{i,j}\}$ forman una red de control bidireccional, los $\{w_{i,j}\}$ son los pesos, y los $\{N_{i,p}(u)\}$ y $\{N_{j,q}(v)\}$ son las funciones base B-splines irracionales definidas en los vectores de nodos

$$U = \{\underbrace{0,\dots,0}_{p+1}, u_{p+1},\dots,u_{m-p-1}, \underbrace{1,\dots,1}_{p+1}\}$$

$$V = \{\underbrace{0,\dots,0}_{q+1}, v_{p+1},\dots,v_{m-p-1}, \underbrace{1,\dots,1}_{q+1}\}$$

Donde r = n+p+1 y s = m+q+1.

Si las funciones base racionales por tramos son

$$R_{i,j}(u,v) = \frac{N_{i,p}(u)N_{j,q}(v)w_{i,j}}{\sum_{k=0}^{n}\sum_{l=0}^{m} N_{k,p}(u)N_{l,p}(v)w_{k,l}} \qquad (Ec.\,12)$$

La superficie (ec. 11) puede ser escrita como

$$S(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v)P_{i,j} \qquad (Ec.\,13)$$

Las figuras 8 y 9 muestran ejemplos de superficies NURBS.

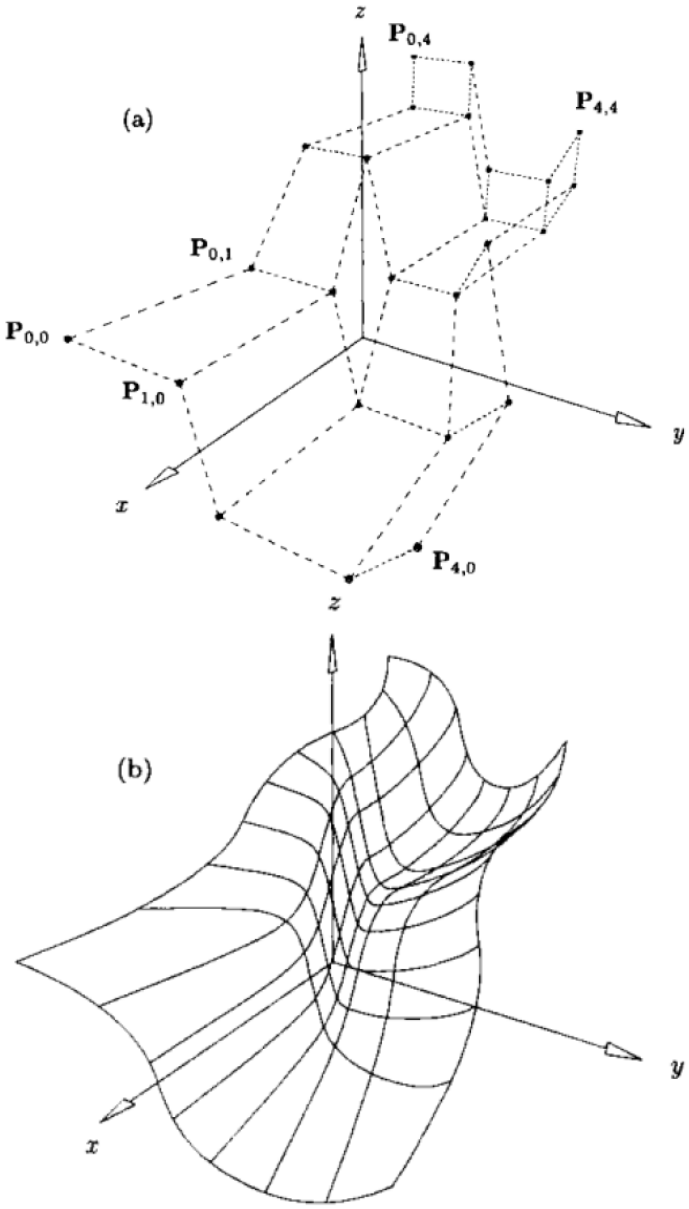


Figura 8: Control de la malla y superficie NURBS bicuadrática con $w_{1,1} = w_{1,2} = w_{2,1} = w_{2,2} = 10$ y el resto de pesos 1, U = {0, 0, 0, 1/3, 2/3, 1, 1, 1}.
a)Malla de control. b) Superficie NURBS bicuadrática.

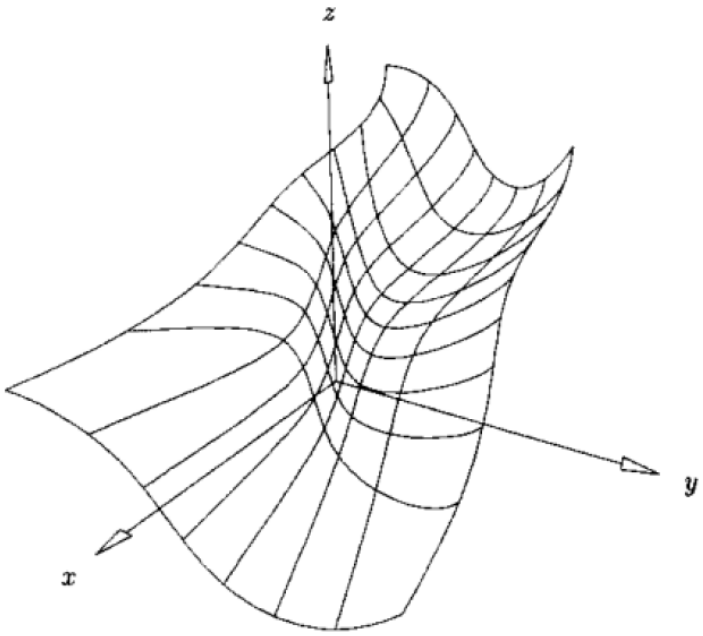


Figura 9: Superficie NURBS bicúbica definida por la malla de control de la fig.8 con U = V = {0, 0, 0, 0, ½, 1, 1, 1, 1} y con los mismos pesos que en la fig. 8.

Las propiedades importantes de las funciones $R_{i,j}(u,v)$ son:

P.15 No negatividad: $R_{i,j}(u,v) \geq 0$ para todo i, j, u y v;

P.16 Partición de la unidad:

$$\sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v) \text{para todo } (u,v) \in [0,1]x[0,1];$$

P.17 Soporte local: $R_{i,j}(u,v)=0$ si (u,v) está fuera del rectángulo dado por $[u_i,u_{i+p+1})x[v_j,v_{j+q+1})$;

P.18 En cualquier rectángulo dado de la forma $[u_{io},u_{io+1})x[v_{jo},v_{jo+1})$, como mucho (p+1)(q+1) funciones base son distintas de cero, en particular los $R_{i,j}(u,v)$ para $i_o\text{-}p \leq 1 \leq i_o$ y $j_{o-q} \leq j \leq j_o$ son distintos de cero;

P.19 Extrema: si p > 0 y q > 0, entonces $R_{i,j}(u,v)$ contiene exactamente un valor máximo;

P.20 $R_{o,o}(0,0) = R_{n,o}(1,0) = R_{o,m}(0,1) = R_{n,m}(1,1) = 1$;

P.21 Diferenciabilidad: en el interior de los rectángulos formados por las líneas de puntos u y v, todas las derivadas parciales de $R_{i,j}(u,v)$ existen. En un punto u (punto v) es p-k(q-k) veces derivable en la dirección u (v), donde k es la multiplicidad del punto;

P.22 Si todos los $w_{i,j} = a$ para $0 \leq i \leq n$, $0 \leq j \leq m$ y $a \neq 0$, entonces
$R_{i,j}(u,v) = N_{i,p}(u)N_{j,q}(v)$ para todo i,j.

Las propiedades 15 a 22 tienen las siguientes propiedades geométricas importantes para las superficies NURBS:

P.23 Interpolación de los puntos en esquina: $S(0,0)=P_{o,o}$; $S(1,0)=P_{n,o}$; $S(0,1)=P_{o,m}$; y $S(1,1)=P_{n,m}$;

P.24 Invarianza afín: Se aplica una transformación afín a la superficie por medio de los puntos de control.

P.25 Fuerte propiedad de envoltura convexa: Se asume que $w_{i,j}\geq 0$ para todo i,j. Si $(u,v)\epsilon[u_{io},u_{io+1})x[v_{jo},v_{jo+1})$, entonces S(u,v) está en la envoltura convexa de los puntos de control $P_{i,j}$; $i_{o-p} \leq i \leq i_o$ y $j_{o-q} \leq j \leq j_o$;

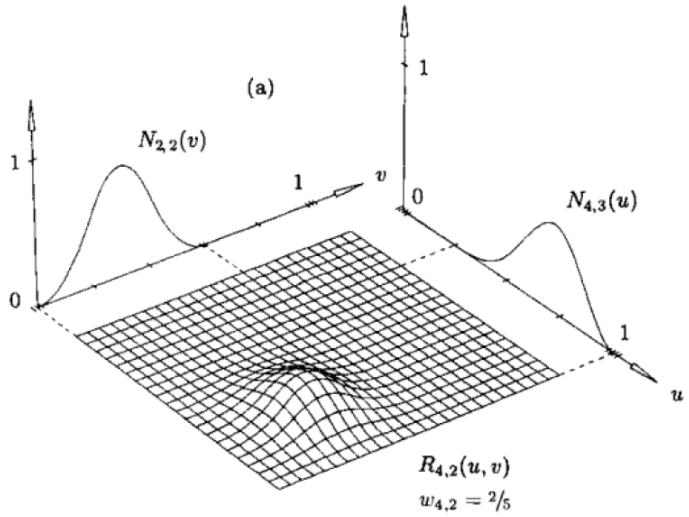
P.26 Modificaciones locales: si $P_{i,j}$ se mueve, o $w_{i,j}$ se cambia, esto afecta a la forma de la superficie sólo en el rectángulo $[u_i,u_{i+p+1})x[v_j,v_{j+q+1})$;

P.27 Las B-splines irracionales, las curvas de Bézier irracionales y las superficies de Bézier racionales son casos especiales de NURBS;

P.28 Diferenciabilidad: S(u,v) es p-k(q-k) veces derivable con respecto a u (v) en el punto u (punto v) de multiplicidad k.

Se remarca que no se conoce ninguna variación que disminuya las propiedades de las superficies NURBS.

Se puede usar tanto el movimiento controlado de los puntos como la modificación de pesos para variar la forma de la superficie. Las figuras 10 y 11 muestran los efectos en las funciones base $R_{i,j}(u,v)$ y la forma de la superficie cuando un único peso, $w_{i,j}$, se modifica.



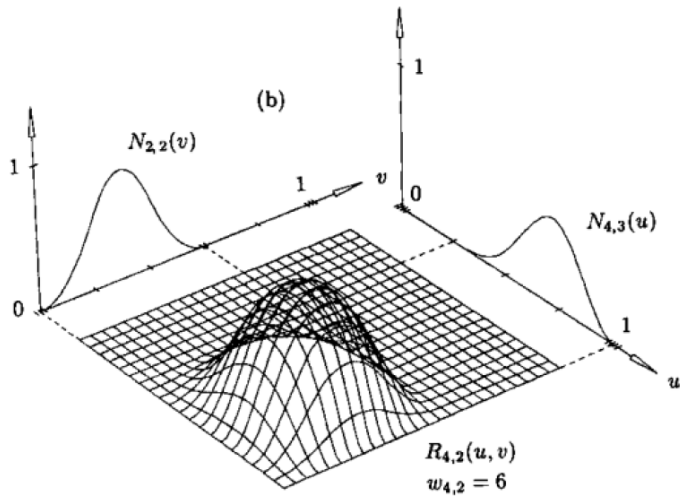
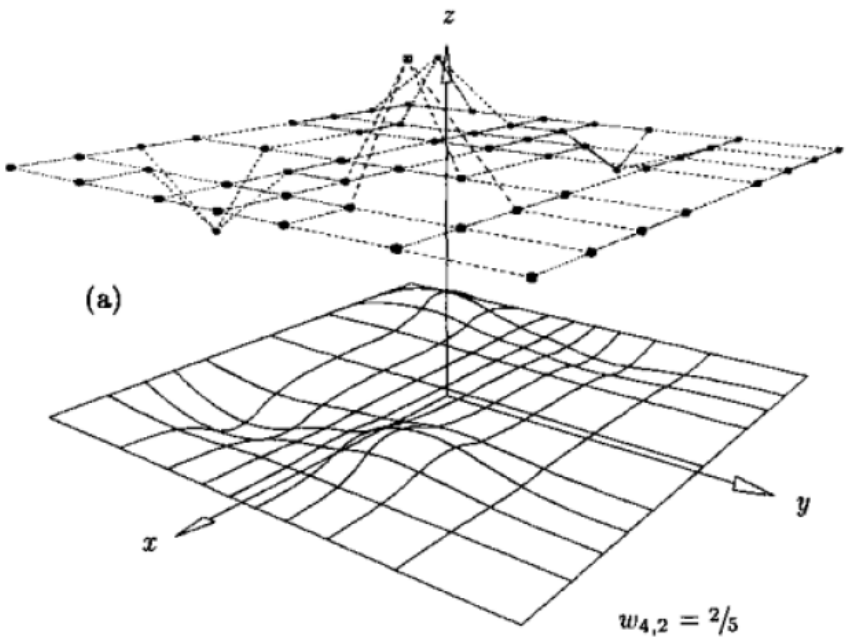


Figura 10: La función base $R_{4,2}(u,v)$, con U={0,0,0,0,1/4,1/2,3/4,1,1,1,1} y V={0,0,0,1/5,2/5,3/5,3/5,4/5,1,1,1}. $w_{i,j}$ = 1 para todo (i,j) ≠(4,2). a)w4,2 = 2/5; b) w4,2 = 6.



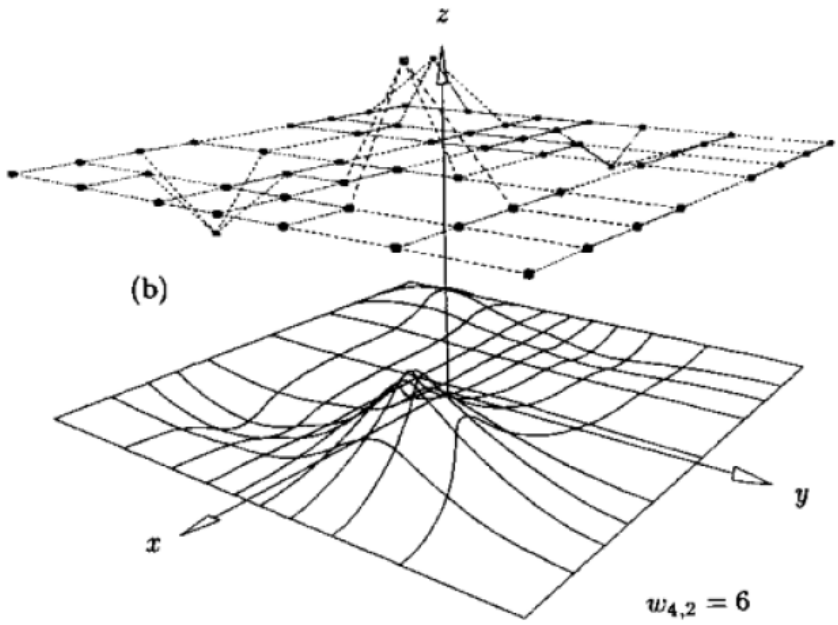


Figura 11: Superficies cúbica x cuadrática correspondientes a la figura 10, con los puntos de control de la malla ocultos para una mejor visualización. a) $w_{4,2} = 2/5$; b) $w_{4,2} = 6$.

Cualitativamente, el efecto en la superficie es: se asume $(u,v)\epsilon[u_i,u_{i+p+1})x[v_j,v_{j+q+1})$; entonces, si $w_{i,j}$ aumenta (disminuye), el punto S(u,v) se acerca a (se aleja de) $P_{i,j}$; y además la superficie es atraída a (alejada de) $P_{i,j}$. Como este es el caso para las curvas, el movimiento de S(u,v) es a lo largo de una línea recta. En la fig.12 (u,v) son fijados y $w_{2,2}$ cambia.

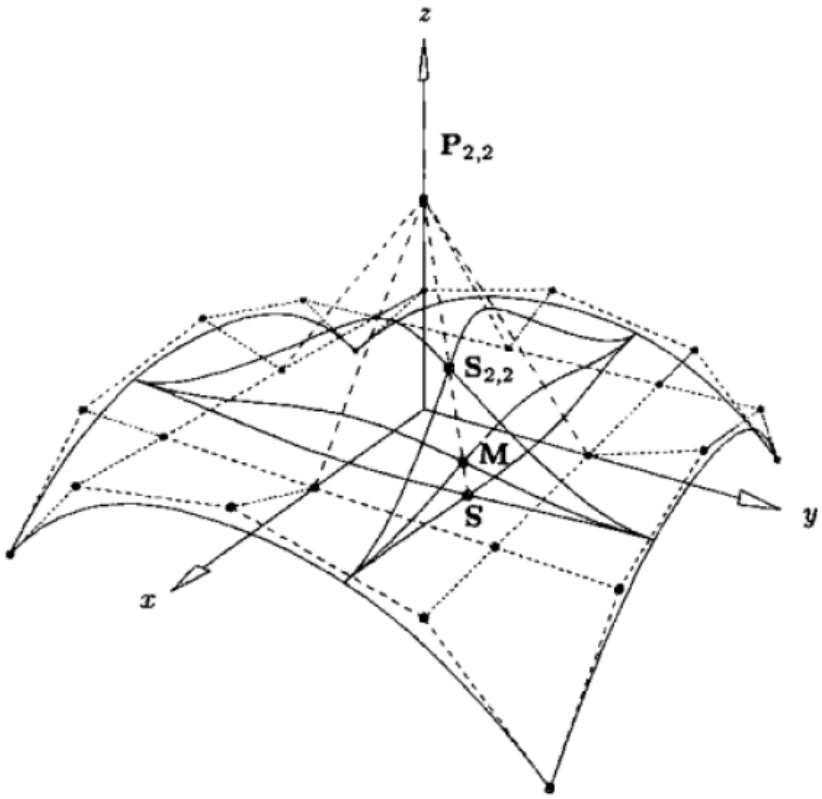


Figura 12: Modificación del peso $w_{2,2}$

Sea:

S = S(u,v; $w_{2,2}$ = 0)
M = S(u,v; $w_{2,2}$ =1) *(Ec. 14)*

Entonces, la línea recta definida por S y M pasa a través de $P_{2,2}$ y por un arbitrario $w_{2,2}$; 0< $w_{2,2}$<∞; $S_{2,2}$ = S(u,v; $w_{2,2}$) permanece en este segmento de línea entre S y $P_{2,2}$.

Es conveniente representar una superficie NURBS usando coordenadas homogéneas, esto es:

$$S^w(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\, N_{j,q}(v) P_{i,j}^w \quad (Ec.15)$$

Donde $P^w_{i,j}$= ($w_{i,j}x_{i,j}$; $w_{i,j}y_{i,j}$; $w_{i,j}z_{i,j}$; $w_{i,j}$). Entonces, S(u,v)=H{ S^w (u,v)}. Nos referimos a la intercambiabilidad entre S^w (u,v) o S(u,v) como superficie NURBS. Estrictamente hablando, S^w(u,v) es un producto tensor, superficie polinómica por tramos en un espacio de cuatro dimensiones. S(u,v) es una superficie racional por tramos en un espacio de tres dimensiones; no es una superficie producto tensor, ya que $R_{i,j}$(u,v) no son productos de funciones base en una variable.

Para obtener las curvas isoparamétricas en una superficie NURBS primeramente fijamos u = u_o

$$C_{u_0}^w(v) = S^w(u_0,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u_0)\, N_{j,q}(v) P_{i,j}^w = \sum_{j=0}^{m} N_{i,q}(v)\left(\sum_{i=0}^{n} N_{i,p}(u_0)\, P_{i,j}^w\right)$$
$$= \sum_{j=0}^{m} N_{j,q}(v) Q_j^w(u_0) \quad (Ec.16)$$

Donde:

$$Q_j^w(u_0) = \sum_{i=0}^{n} N_{i,p}(u_0) P_{i,j}^w$$

Análogamente:

$$C_{u_0}^w(u) = \sum_{i=0}^{n} N_{i,p}(u) Q_i^w(v_0) \quad (Ec.17)$$

Donde:

$$Q_i^w(v_0) = \sum_{j=0}^{m} N_{j,q}(v_0) P_{i,j}^w$$

C^w_{uo}(v) (C^w_{vo}(u)) es una curva de grado q (p) en el vector nudo V (U). El punto S^w(u_o,v_o) permanece en la intersección de C^w_{uo}(v) y C^w_{vo}(u). Proyectando queda

$$C_{u_0}(v) = H\{C_{u_0}^w(v)\} = H\{S^w(u_0,v)\} = S(u_0,v)$$
$$C_{v_0}(u) = H\{C_{v_0}^w(u)\} = H\{S^w(u,v_0)\} = S(u,v_0) \quad (Ec.18)$$

Derivadas de una superficie NURBS

Las derivadas de S(u,v) las calculamos en términos de S^w (u,v). Sea

$$S(u,v) = \frac{w(u,v)S(u,v)}{w(u,v)} = \frac{A(u,v)}{w(u,v)}$$

Donde A(u,v) es el numerador de S(u,v) (Ec. 11). Entonces

$$S_\alpha(u,v) = \frac{A_\alpha(u,v) - w_\alpha(u,v)S(u,v)}{w(u,v)} \quad (Ec.19)$$

Donde α puede ser u ó v. En general

$$A^{(k,l)} = [(wS)^k]^l = \left(\sum_{i=0}^{k}\binom{k}{i}w^{(i,0)}S^{(k-i,0)}\right)^l = \sum_{i=0}^{k}\binom{k}{i}\sum_{j=0}^{l}\binom{l}{j}w^{(i,j)}S^{(k-i,l-j)}$$
$$= w^{(0,0)}S^{(k,l)} + \sum_{i=1}^{k}\binom{k}{i}w^{(i,0)}S^{(k-i,l)} + \sum_{j=1}^{l}\binom{l}{j}w^{(i,j)}S^{(k,l-j)}$$
$$+ \sum_{i=l}^{k}\binom{k}{i}\sum_{j=l}^{l}\binom{l}{j}w^{(i,j)}S^{(k-i,l-j)}$$

Y de donde se deduce que

$$S^{(k,l)} = \frac{1}{w}\left(A^{(k,l)} - \sum_{i=1}^{k}\binom{k}{i}w^{(i,0)}S^{(k-i,l)} - \sum_{j=1}^{l}\binom{l}{j}w^{(0,j)}S^{(k,l-j)} - \sum_{i=1}^{k}\binom{k}{i}\sum_{j=1}^{l}\binom{l}{j}w^{(i,j)}S^{(k-i.l-j)}\right) \quad (Ec.20)$$

De la ec. 20 se obtiene

$$S_{uv} = \frac{A_{uv} - w_{uv}S - w_uS_v - w_vS_u}{w} \quad (Ec.21)$$

$$S_{uu} = \frac{A_{uu} - 2w_uS_u - w_{uu}S}{w} \quad (Ec.22)$$

$$S_{vv} = \frac{A_{vv} - 2w_vS_v - w_{vv}S}{w} \quad (Ec.23)$$

De las ecuaciones 19 y 20 se obtiene:

$$S_u(0,0) = \frac{p}{u_{p+1}}\frac{w_{1,0}}{w_{0,0}}(P_{1,0} - P_{0,0}) \quad (Ec.24)$$

$$S_v(0,0) = \frac{q}{u_{q+1}} \frac{w_{0,1}}{w_{0,0}} \left(P_{0,1} - P_{0,0}\right) \quad (Ec.25)$$

$$S_{uv}(0,0) = \frac{pq}{w_{0,0} u_{p+1} v_{q+1}} \left(w_{1,1} P_{1,1} - \frac{w_{1,0} w_{0,1}}{w_{0,0}} \left(P_{1,0} + P_{0,1}\right) + \left(\frac{2 w_{1,0} w_{0,1}}{w_{0,0}} - w_{1,1} \right) P_{0,0} \right) \quad (Ec.26)$$

La figura 13 muestra las derivadas parciales de primer y segundo orden de una superficie NURBS. Las derivadas parciales de primer orden están escaladas por 1/2, y las de segundo orden están escaladas por 1/3.

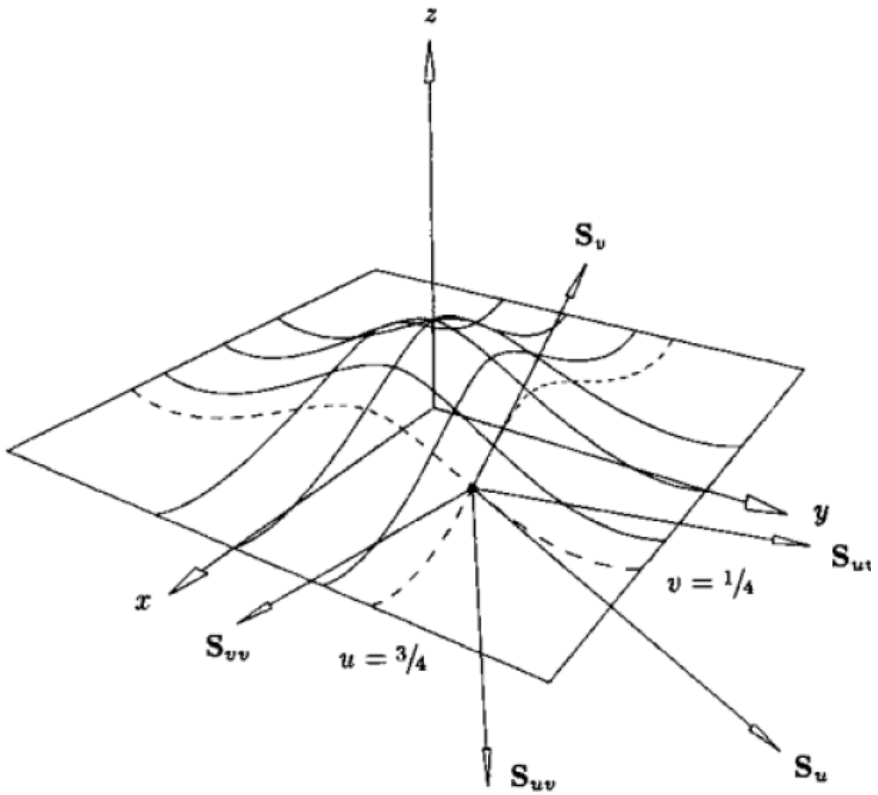


Figura 13: Las derivadas parciales de primer y segundo orden de una superficie NURBS bicúbica en u = ¾ y v = ¼.

Interrogación de superficies

Se entiende por *interrogación de superficies* la determinación y cálculo de propiedades geométricas de superficies ya construidas. Este campo resulta de extraordinaria importancia en la industria naval pues dichas propiedades geométricas constituyen una medida de la "calidad" del diseño realizado.

Por supuesto, se han definido en los últimos años un gran número de métodos de interrogación de superficies, siendo los siguientes los más tradicionalmente utilizados.

- Métodos basados en la iluminación:

Los métodos basados en iluminar las superficies presentan la ventaja de su sencillez de formulación y de implementación por computador, así como de su interpretación visual y geométrica. Entre ellos citamos:

-Líneas de reflexión
-Isofotas.

Las *líneas de reflexión* ayudan a valorar la estética de la superficie ante modelos de iluminación. Esencialmente la idea consiste en reflejar una familia de líneas rectas sobre una superficie desde un punto fijo de visión. Las líneas de reflexión son exactamente las líneas que surgen de este proceso sobre la superficie. La idea reproduce el proceso físico de aplicar papel de aluminio sobre un modelo de arcilla, iluminar éste y visualizar con la ayuda de un espejo. Matemáticamente, este proceso puede enunciarse como:

$$\frac{P-A}{\| P-A \|} + \frac{L-P}{\| L-P \|} = 2N\left(N \cdot \frac{P-A}{\| P-A \|}\right)$$

Donde N(u,v) es el vector normal a la superficie S(u,v), A es el punto de observación, L es la línea a proyectar, P es un punto sobre la superficie y "·" denota producto escalar.

A continuación se muestran dos ejemplos de interrogación de superficies mediante líneas de reflexión. Aparentemente, ambos casos responden a la misma superficie:

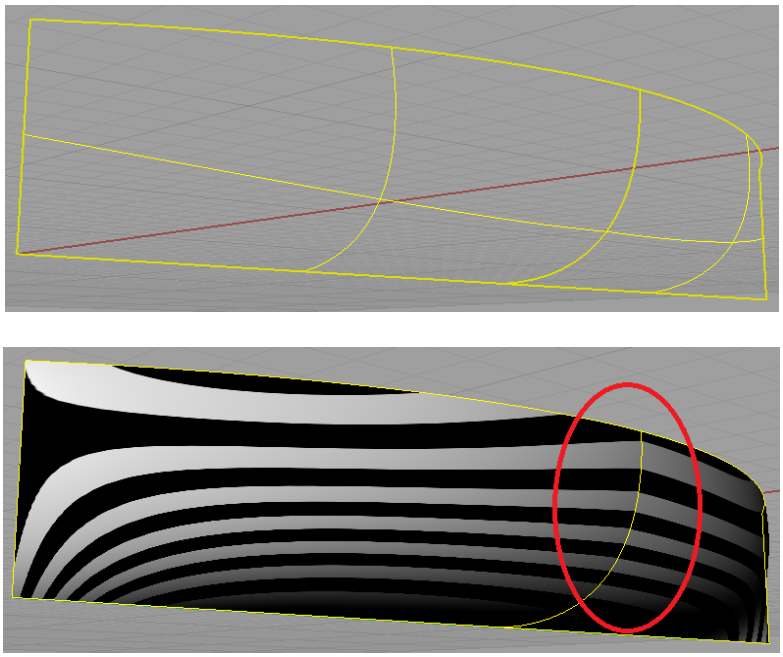


Figura 14. Se muestran dos superficies unidas sin continuidad. Se puede apreciar el cambio brusco de dirección de las líneas de reflexión.

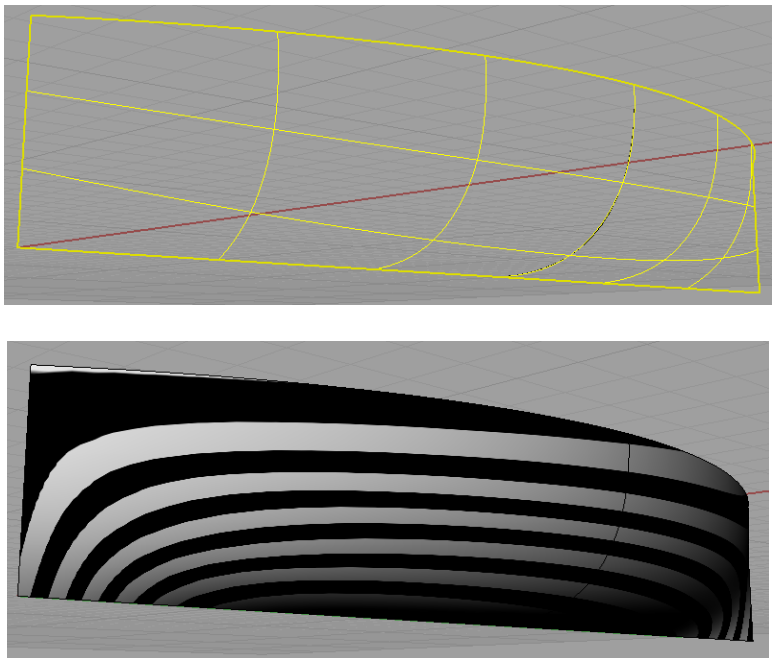
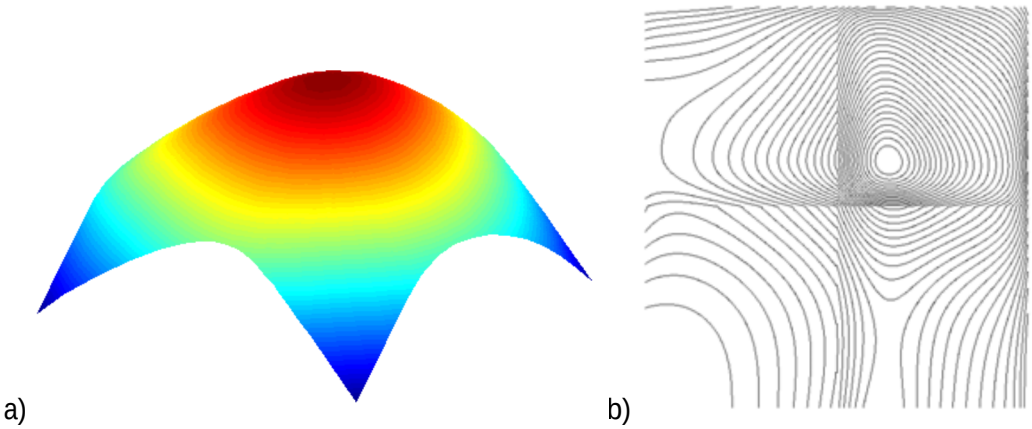


Figura 15. Uniendo ambas superficies, pero con continuidad, las líneas de reflexión reflejan continuidad de la superficie final (*alisada*)

Las curvas *isofotas*,10 definidas por aquellos puntos en los que la normal a la superficie bajo estudio forma un ángulo dado con una cierta dirección de la luz, permiten detectar discontinuidades entre parches. Efectivamente, si **L** es la dirección de la luz, las isofotas vienen dadas por la condición:

N(*u, v*).**L** = *c* = *cte* (Ec. 1)

La Figura 2 revela el papel que juegan las isofotas en interrogación: a primera vista, la superficie NURBS de la izquierda (formada por 4 parches unidos con continuidad C^2) parece perfectamente regular y no es posible extraer información sobre su estructura, tal como el número de parches y su continuidad. Sin embargo, sus isofotas, mostradas a la derecha, indican claramente la existencia de los 4 parches y la no continuidad C^2 de las curvas isofotas entre parches. Es de destacar que si una superficie presenta continuidad C^r sus isofotas son curvas de continuidad C^{r-1}. En general, la condición de la Ec. 1 debe aplicarse para varios valores de *c*, calculando numéricamente sus soluciones. Sin embargo, uno puede comprobar, digamos, 200 direcciones con resultados correctos y que las isofotas revelen problemas en la dirección 201. Por ello, en nuestro trabajo diario debemos considerar un método automático de comprobación de la continuidad.



a) b)

Figura 16: Superficie NURBS formada por 4 parches (a) y sus isofotas (b)

- Métodos basados en derivadas

Las curvaturas media y gaussiana juegan un papel clave en el estudio de la continuidad de las superficies. Su cálculo se basa en las derivadas de la superficie NURBS, discutida en la Sección 2.3. A partir de ellas, se determinan los primeros y los segundos coeficientes principales (*E,F,G*) y (*e, f, g*) respectivamente como:

$E = \mathbf{S}_u.\mathbf{S}_u$; $F = \mathbf{S}_u.\mathbf{S}_v$; $G = \mathbf{S}_v.\mathbf{S}_v$
$e = \mathbf{N}.\mathbf{S}_{uu}$; $f = \mathbf{N}.\mathbf{S}_{uv}$; $g = \mathbf{N}.\mathbf{S}_{vv}$

La familia de planos que contienen a la normal **N** a la superficie **S** en un punto dado (*u,v*) interseca a dicha superficie en una familia de curvas llamadas *curvas normales*, descritas por *{u = u(t), v = v(t)}*, que poseen su propia curvatura, llamada *curvatura normal* κ:

$$\kappa = \frac{-eu^{\cdot 2} + 2fu^{\cdot}v^{\cdot} + gv^{\cdot 2}}{Eu^{\cdot 2} + 2Fu^{\cdot}v^{\cdot} + Gv^{\cdot 2}}$$

para la dirección específica ($u^{\cdot}$, $v^{\cdot}$). En este contexto, las *direcciones principales* *e*1 y *e*2 y las *curvaturas principales* κ1 y κ2 indican, respectivamente, las direcciones y magnitudes del mínimo y máximo de todas las posibles curvaturas normales en un punto. Por tanto, las curvaturas principales deben cumplir

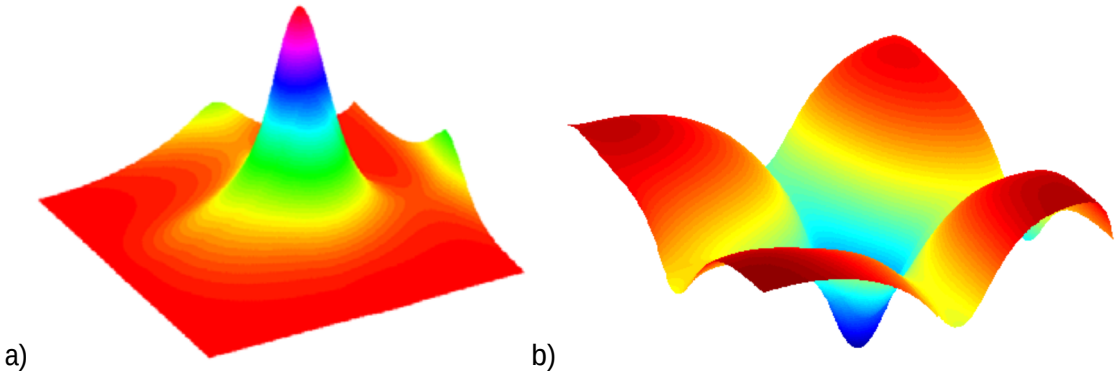
$$\frac{\partial \kappa}{\partial u^{\cdot}} = \frac{\partial \kappa}{\partial v^{\cdot}} = 0; es\ decir\ \kappa 2 - 2H\kappa + K = 0$$

Donde:

$K = \frac{eg - f^2}{EG - F^2}$; y

$$H = \frac{2Ff - (Eg + Ge)}{2(EG + F^2)}$$

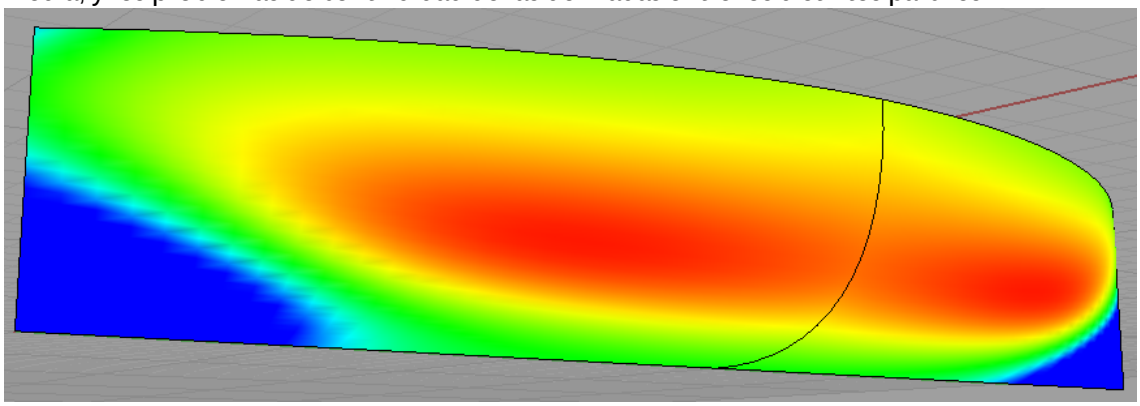
reciben el nombre de *curvatura gaussiana K* y *curvatura media H,* respectivamente.



a) b)

Figura 17: Curvaturas Gaussiana (a) y media (b) de la superficie NURBS de la figura

La visualización de la curvatura gaussiana *K* (por ejemplo, mediante una gama de colores o como una superficie) es una herramienta ampliamente utilizada para analizar la calidad de las superficies. Una variación suave de la curvatura gaussiana indica que el comportamiento de la superficie es suave, mientras que una variación rápida es síntoma de comportamiento abrupto. Si además la superficie se compone de múltiples parches (situación usual en el diseño naval), la curvatura permite no sólo evaluar la suavidad de los parches individuales en su interior, sino también la suavidad de las transiciones entre parches, es decir, la continuidad entre parches adyacentes a lo largo de sus curvas frontera. No obstante, la curvatura media *H* es más fiable que la curvatura gaussiana para analizar la continuidad de la curvatura entre parches.11 Ello se debe a que dos superficies con la misma curvatura gaussiana a lo largo de la curva frontera no siempre cumplen la continuidad de la curvatura sobre dicha curva, propiedad que si se satisface con la curvatura media. La Figura 2 muestra las funciones curvatura gaussiana y media de la superficie NURBS de la Figura 1. Como se aprecia, la curvatura gaussiana es bastante regular excepto en el entorno del punto de unión entre los parches de la superficie, donde la curvatura exhibe valores mayores. Esto indica efectivamente que la continuidad está limitada a clase C^2. Por otro lado, la curvatura media evidencia nuevamente la existencia de 4 parches en la superficie, cada uno con su propia curvatura media, y los problemas de continuidad de las derivadas entre los distintos parches.



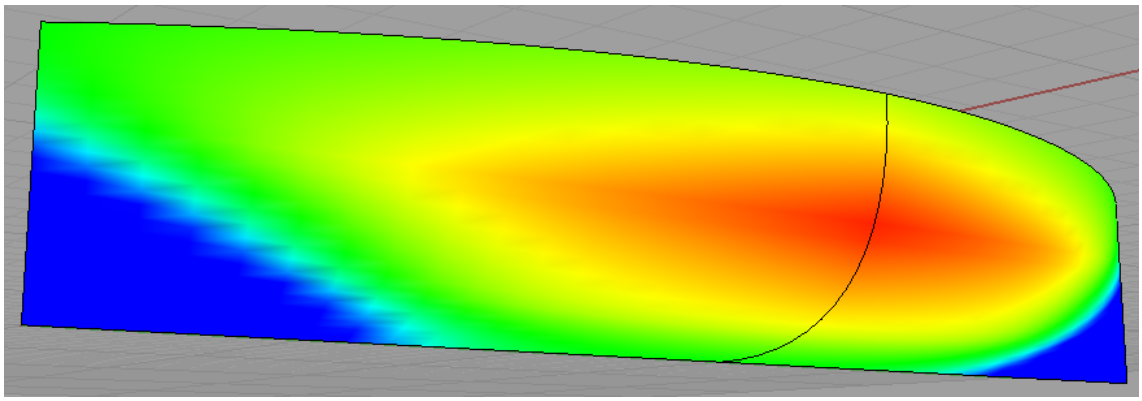


*Figura 18:*En el caso del ejemplo anterior, la curvatura gaussiana muestra la evolución de la curvatura. Aunque se puede intuir cierta discontinuidad en el ejemplo a), ésta no se muestra tan evidente como en el análisis realizado con líneas de reflexión.

A pesar de su importancia en la interrogación de superficies, en muchas ocasiones las curvaturas media y gaussiana no resultan suficientes. Un típico ejemplo se da en los procesos de mecanizado. Por ejemplo, si la fresa que mecaniza una superficie es esférica, su radio debe ser obviamente menor que el más pequeño radio cóncavo de curvatura. Por tanto, la acotación de este valor es de primordial importancia. Para ello basta calcular las cotas superior e inferior de la curvatura en cada punto de la superficie, es decir, las *curvaturas principales*, dadas como las dos soluciones de (10):

$$\kappa_1 = H + \sqrt{H^2 - K}\,;\ \kappa_2 = H - \sqrt{H^2 - K}$$

Relacionadas con las curvaturas principales, las *líneas de curvatura* indican la dirección de la máxima y la mínima curvatura a lo largo de la superficie. Resultan por tanto muy útiles en interrogación de superficies y su cálculo puede ser realizado mediante la integración numérica del siguiente sistema de ecuaciones diferenciales:

$$\begin{cases} \dfrac{du}{ds} = \beta(g - \kappa F) \\ \dfrac{dv}{ds} = -\beta(f + \kappa E) \end{cases}$$

donde

$$\beta = \frac{\pm 1}{\sqrt{E(g + \kappa F)^2 - 2F(g + \kappa F)(f + \kappa E) + G(f + \kappa E)^2}}$$

y κ representa cualquiera de las dos curvaturas principales en (12). Este sistema de dos ecuaciones diferenciales acopladas no lineales no puede ser resuelto, en general, analíticamente y recurrimos a métodos numéricos como un esquema de Runge-Kutta de cuarto orden y paso fijo. Las condiciones iniciales son las coordenadas (*u, v*) del punto inicial y así para cada punto las líneas de curvatura son únicas.

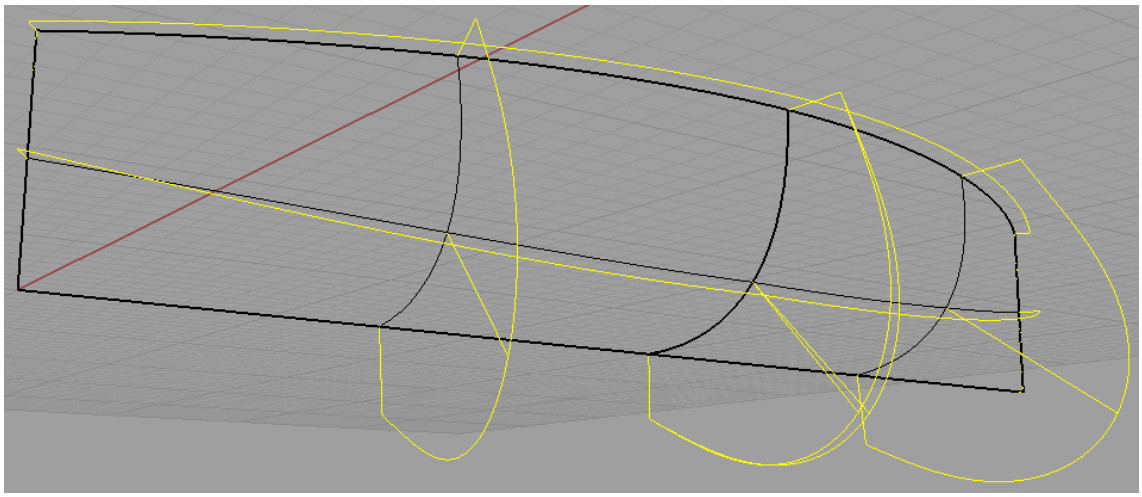


Figura 19: Usando un programa informático, las líneas de curvatura se muestran en las direcciones preseleccionadas (en este caso, u,v). Tanto la escala, como el número de líneas se puede variar. Dichas líneas se representan, en un modelo 3-D, perpendiculares a la superficie, por lo que habrá que variar la vista para detectar discontinuidades.

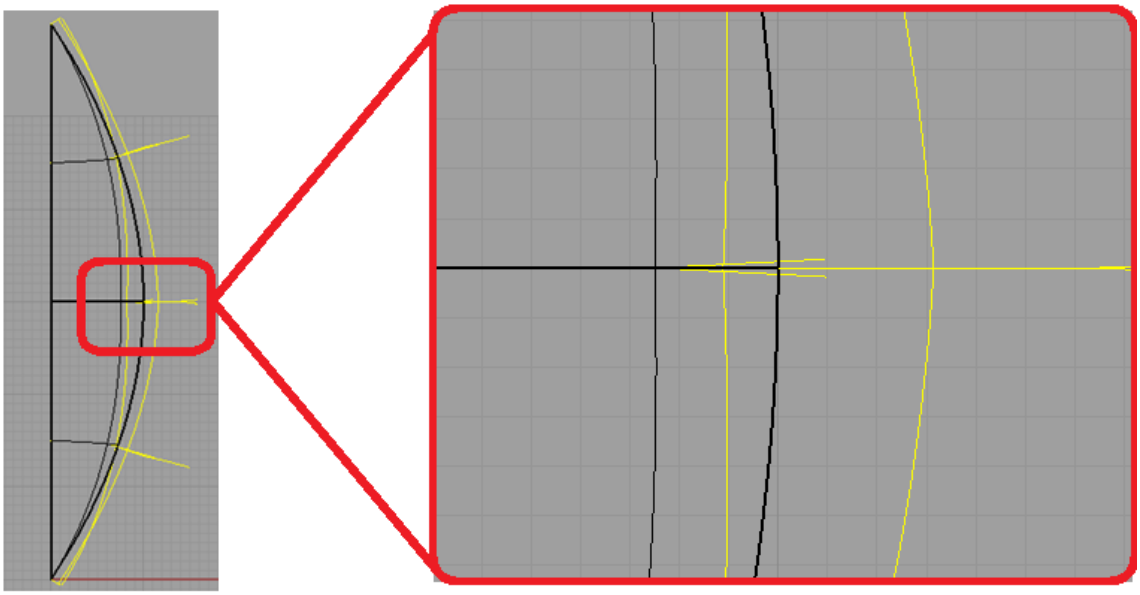


Figura 20: En el caso del ejemplo, observando el modelo en el plano (x,y), se aprecian discontinuidades en las líneas de curvatura de la zona de unión .

2.Introducción al lenguaje C

Introducción

Predecesor y base del lenguaje C++, el lenguaje C fue desarrollado en 1972 por Dennis M. Ritchie, en los laboratorios Bell, para la implementación de Sistemas Operativos (en concreto UNIX). C muestra los elementos esenciales del lenguaje en programas reales. En éste capítulo se busca, por tanto, un punto desde el que comenzar a escribir programas útiles, basados en conceptos básicos del lenguaje: *variables* y *constantes, aritmética, control de flujo, funciones,* y primeras nociones de *entradas* y *salidas*. También se abordarán estructuras de programación, *operadores C* más comunes y útiles, varias declaraciones de flujos, y *librerías* estándar.

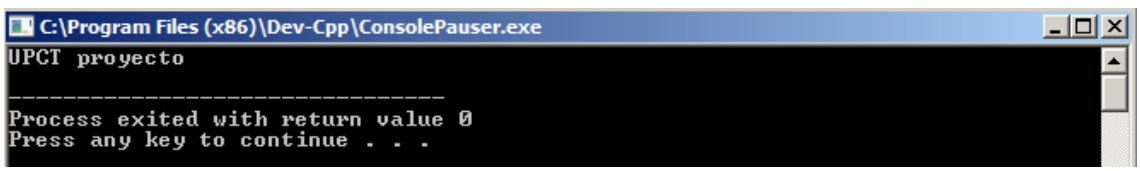
Existen otros aspectos de la programación que no se abordarán, ya que no son objeto de éste proyecto, pero es importante tener en cuenta, como son la capacidad y utilidad general y específica del lenguaje C, o la *elegancia* con la que se programa, entendiendo ésta como la simplicidad matemática utilizada y rendimiento de cálculo (cuanto menos capacidad de proceso se necesite a la hora de ejecutar un programa, tanto más *elegante* se puede considerar).

Lenguaje C: conceptos básicos

La única vía para aprender y comprender un lenguaje de programación, es escribiendo programas. A continuación se mostrará una rápida programación que nos devuelva las palabras “UPCT proyecto”. Para ello se usará un programa con el que *compilar* el proceso, siendo éste el Dev-C++.El proceso en Dev-C++ será: crear el texto del programa, compilarlo satisfactoriamente (es decir, que no contenga errores de sintaxis), cargar el programa, y obtener la respuesta de éste. La estructura del programa será:

```
#include <stdio.h>
main()
{
    printf("UPCT proyecto\n");
}
```

Utilizando el programa Dev C++, compilamos y ejecutamos el programa, apareciendo:



Una programación en C, sea cual sea su tamaño, está compuesta por *funciones* y *variables*. Una función contiene *declaraciones* (*statements*) que especifican las operaciones computacionales a hacer, mientras que las variables adquieren valores durante la computación. Un ejemplo es la función "*main*". Normalmente se tiene libertad para poner nombre a las funciones, pero *main* es especial: el programa comienza ejecutando al principio su *main* (principal). Significa que todo programa debe tener su *main* en algún lugar.

Main normalmente llamará a otras funciones para ayudar a realizar la computación, sobre lo que se escribe o programa, o sobre la ejecución de bibliotecas. Volviendo al ejemplo anterior, la primera línea declara que:

#include <studio.h>

Que no es otra cosa que decir al compilador que incluya información sobre la librería estándar entrada/salida. Dicha declaración aparece siempre al principio de cada programación en C.

Un método de comunicación de datos entre funciones para llamar a la función es proveer de valores, llamados argumentos (*arguments*), a la función a la que llama. Los paréntesis después del nombre de la función indica la lista de argumentos. En el ejemplo anterior, *main* aparece con paréntesis vacíos, por lo que define una función sin argumentos: *main ().*

Las declaraciones de una función está encerrada entre llaves { }. La función *main* contiene, en el caso anterior sólo una declaración:

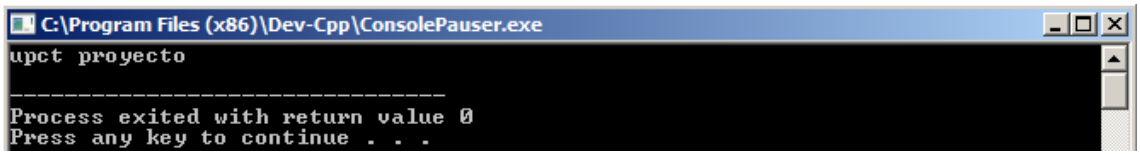
printf("UPCT proyecto\n"}

La función *printf* (en éste caso) es llamada nombrándola, seguida de un paréntesis con la lista de argumentos ("UPCT proyecto\n"). *printf* es una función obtenida de la librería que imprime lo que se encuentre entre comillas dentro de sus paréntesis. La secuencia de caracteres entre comillas dobles (" ", y no ` ´) se denominan *cadena de caracteres* o *cadena constante*. Dichas cadenas, sólo es utilizan en funciones como *printf*.

La secuencia \n indica el paso al siguiente renglón (*newline character*). Se suele utilizar para ordenar las cadenas de caracteres a la hora de ejecutar un programa, si éstas son demasiado largas, o bien, para mostrar en el siguiente renglón la siguiente salida. La función *printf* nunca pasa al siguiente renglón si no se escribe \n cuando se precise. De hecho, podremos declarar varias funciones seguidas *printf* sin indicar \n, obteniendo en la ejecución del programa los caracteres seguidos. Para ver la equivalencia, se propone un ejemplo similar al anterior, viendo que la salida es la misma. En éste caso, las iniciales UPCT se pondrán en minúscula para distinguirlo del anterior caso:

```
#include <stdio.h>
main()
{
printf("upct ");
```

```
printf("proyecto");
printf("\n");
}
```



Otros mecanismos de lenguaje C para las cadenas de caracteres, usando *printf* son \t para tabular, \b para retroceder,\" para añadir comillas y \\ para añadir una contrabarra (\ ).

Variables y expresiones aritméticas

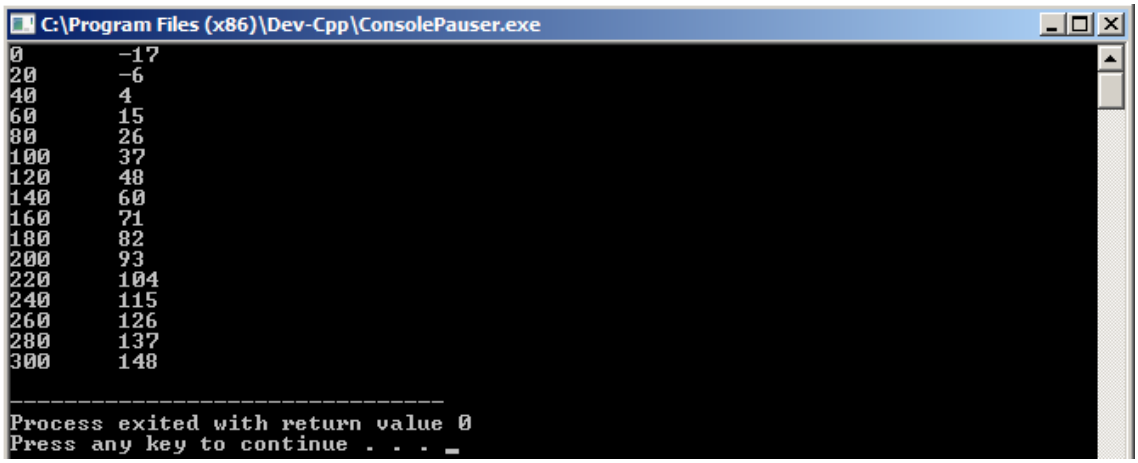
El siguiente programa reproduce la fórmula $^{o}C=(5/9)(F-32)$, obteniendo tras la ejecución una tabla con las equivalencias entre grados centígrados y Farenheit.

El programa consistirá en una función (que llamaremos *main*), y e incluirán declaraciones, comentarios, variables, expresiones aritméticas y bucles:

[*] fahrenheit.cpp

```
1   #include <stdio.h>
2
3   /*Obtencion de equivanecias entre ºFahrenheit-ºCelsius
4   para 0,20,.....,300 ºFahrenheit*/
5   main()
6   {
7       int fahr, celsius;
8       int lower, upper, step;
9
10      lower=0;        /*valor más bajo de la escala de temperaturas*/
11      upper=300;      /*valor mas alto*/
12      step=20;        /*incremento de temperatura*/
13
14      fahr=lower;
15      while (fahr<=upper){
16          celsius=5*(fahr-32)/9;
17          printf("%d\t%d\n",fahr, celsius);
18          fahr=fahr+step;
19      }
20  }
```

Obteniendo tras la compilación y ejecución:



Volviendo a la sintaxis anterior se pueden extraer nuevos comandos propios de la programación en C:

En las líneas 3 y 4:

```
3   /*Obtencion de equivanecias entre ºFahrenheit-ºCelsius
4   para 0,20,.....,300 ºFahrenheit*/
```

Se trata de un comentario. Son notas útiles para la compresión de la programación descritas por el programador. Cualquier cosa que se escriba entre /* y */ son ignoradas por el compilador, sea cual sea su ubicación, por lo se tendrán en cuenta durante la ejecución del programa.

En C, todas las variables deben ser declaradas antes de ser usadas, normalmente justo antes de ser utilizadas en sentencias ejecutables. Una *declaración* anuncia las propiedades de las variables; y consiste en nombrar y listar las variables, como en las líneas 7 y 8:

```
7       int fahr, celsius;
8       int lower, upper, step;
```

El tipo *int* indica que las variables son números enteros. Por otro lado encontramos la variable *float*, que indica un número fraccionario. También existen otros tipos de variables que no son enteros o fraccionarios, como:

char	Carácter simple(1 byte)
short	Entero corto
long	Entero largo
double	Numero fraccionario con el doble de precisión.

El tamaño elegido será en función de la potencia del procesador utilizado y la precisión necesitada. También existen *formaciones (arrays), estructuras* y *uniones* es dichos tipos básicos, *indicadores (pointers)*, y *funciones* que devuelven éste tipo de variables, las cuales se verán más adelante.

Para la computación del programa, se comienza dando valores a las variables (*asignando declaraciones*). En las líneas 10, 11 y 12 se declara:

```
10      lower=0;        /*valor más bajo de la escala de temperaturas*/
11      upper=300;      /*valor mas alto*/
12      step=20;        /*incremento de temperatura*/
```

Que da unos valores iniciales a las variables. Para definirlos correctamente, se acaba la declaración con un punto y coma (;).

Cada línea que se observa en el programa ejecutado, se computa de la misma manera: se usa un bucle que repita el mismo cálculo las veces necesarias. Éste es el propósito del bucle *while* (líneas 15, 16, 17, 18 y 19)

```
15      while (fahr<=upper){
16          celsius=5*(fahr-32)/9;
17          printf("%d\t%d\n",fahr, celsius);
18          fahr=fahr+step;
19      }
```

El bucle *while* opera de la siguiente manera: la condición del paréntesis se comprueba. Si se cumple (*fahr menor o igual a upper*), el cuerpo del bucle (líneas 16, 17 y 18) se ejecuta. Tras cada ejecución se comprueba la condición *fahr<=upper*, volviéndose a repetir si se cumple la condición. En cuanto la condición no se cumpla (*fahr>upper*), el bucle termina, y la ejecución del programa continúa con la declaración que sigue al bucle.

Con dicho bucle, se van generando los distintos valores de grados Celsius en función de los valores de los grados Fahrenheit a través de la fórmula de conversión. Seguidamente, los datos se van mostrando con la función *printf("%d\t%d\n",fahr, celsius)*. El primer argumento de la función *printf* es una cadena de caracteres a mostrar, donde % indica cómo se muestran los argumentos. Por ejemplo, *%d* indica un argumento entero, y la declaración

```
17          printf("%d\t%d\n",fahr, celsius);
```

Indica que se muestren los valores enteros de las variables *fahr* y *celsius*, con una tabulación entre ambas (*\t*), siendo *\n* el comando para pasar al siguiente renglón.

Si atendemos a continuación, los valores obtenidos en la tabla de conversión tras la ejecución del programa, podemos apreciar cierta imprecisión en los valores obtenidos (por ejemplo, 0F = -17.8C y no -17C). Esto es debido al tipo de variables utilizadas en la redacción del programa, siendo más apropiadas las variables tipo *float* (o números fraccionarios). Un segundo programa más preciso sería:

```
1  #include<stdio.h>
2
3  /*tabla conversion fahrenheit-celsius*/
4   main()
5   {
6      float fahr, celsius;
7      float lower, upper, step;
8
9      lower=0;
10     upper = 300;
11     step=20;
12
13     fahr=lower;
14     while (fahr<=upper){
15         celsius=(5.0/9.0)*(fahr-32.0);
16          printf("%3.0f %6.1f\n",fahr,celsius);
17          fahr=fahr+step;
18     }
19  }
```

```
C:\Program F
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9

------------
Process exit
Press any ke
```

Donde se muestra la ejecución en el lado derecho de la imagen. Como se observa, las variables se han declarado como *float*, aumentando la precisión en los datos obtenidos. Para declarar también como *float* el resto de valores utilizados en la ecuación, se le añade un cero decimal. Ésta es una manera de que C interprete los datos obtenidos de sus operaciones como *float*.

En una operación aritmética, si los valores son enteros, la operación se realiza con enterosy obteniendo un entero. Sin embargo, si en una operación aritmética se tiene un valor fraccional (*float*) y otro entero (*int*), automáticamente, se toma el valor entero como fraccional, es decir, transforma *int* en *float*. De modo que bien se podría haber escrito en la ecuación (5.0/9) o (5/9.0) obteniendo valores correctos y no (5/9), ya que en éste último caso, obtendríamos un entero de valor cero.

Otro cambio realizado en éste último ejemplo es la forma de representar los resultados en la función *printf("%3.0f %6.1f\n",fahr, cesius);*. La secuencia *%3.0f* indica los valores obtenidos de *fahr* se muestren al menos 3 caracteres, y sin ningún decimal (los ceros a la izquierda no se muestran). *%6.1f* indica que se muestren al menos 6 caracteres con una cifra decimal, por lo que la ejecución del programa muestra 6 espacios entre el valor *fahr* y *celsius* y una cifra decimal. Para la representación de datos, se tienen también:

%d	Muestra un entero decimal
%6d	Muestra, al menos, los 6 primeros caracteres de un entero decimal
%f	Muestra un numero fraccionario
%6f	Muestra un número fraccionario con al menos las 6 primeras cifras
%.2f	Muestra un número fraccionario con 2 cifras decimales
%6.2f	Muestra un número fraccionario con al menos las 6 primeras cifras y 2 decimales

Se indica número decimal porque *printf* también reconoce valores en base octal (%o), hexadecimal (%x), caracteres (%c), cadena de caracteres (%s) y a sí mismo (%%).

La declaración *for*

For es un bucle, al igual que *while*, pero generalizado, aunque ambos se basan en el mismo principio: la ejecución de órdenes siempre que se cumpla una o varias condiciones. El uso de uno u otro dependerá de las ventajas o del problema a resolver, usando el que mejor se adapte. Se puede ver la forma de trabajar de *for* repitiendo al caso de la conversión entre grados Fahrenheit u grados Celsius:

fahrenheit for.cpp

```
1  #include <stdio.h>
2  /*tabla Fahrenheit-Celsius*/
3  main()
4  {
5      int fahr;
6
7      for(fahr=0; fahr<=300; fahr=fahr+20.0)
8      printf("%3d %6.1f\n",fahr,(5.0/9.0)*(fahr-32))
9  }
```

C:\Program

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

En éste caso, se aprecia una gran simplificación, además del uso de tan solo una variable (*fahr*), y obteniendo los valores de grados Celsius directamente en la función *printf* . La forma de trabajar con *for* tiene tres partes:

for(fahr=0; fahr<=300; fahr=fahr+20.0)

La primera parte (*fahr=0;*), es el valor inicial de la variable en el bucle. Ésta se lee sólo en el primer paso del bucle, es decir, para iniciarlo.

La segunda parte (*fahr<=300*), establece la condición para que se mantenga el bucle en ejecución, es decir, hasta que *fahr* no sea mayor que 300, el bucle seguirá ejecutándose.

La tercera parte introduce el siguiente paso del bucle, incrementando en valor de *fahr* progresivamente en cada ciclo del bucle, y finalizando éste cuando se alcanza la condición impuesta.

El bucle *for* obtiene la misma respuesta pero de forma distinta y, en éste caso, con la sola declaración de una variable *int.*

Constantes simbólicas

El concepto de constantes simbólicas se introduce, básicamente, para aportar comprensión al texto programado. Se define como *nombre simbólico* o *constante simbólica* a una sucesión de caracteres particulares:
#define sucesión de caracteres

La única finalidad es aportar una estructura de programa de manera que cualquier programador pueda dar valores a las constantes simbólicas, comprendiendo mejor cómo funciona el programa tratado. Volviendo al problema anterior:

```
1   #include <stdio.h>
2
3   #define LOWER 0     /*valor mas bajo de la lista*/
4   #define UPPER 300   /*valor mas alto de la lista*/
5   #define STEP  20    /*incremento o paso*/
6
7   /*tabla conversora Fahrenheith-Celsius*/
8
9   main()
10  {int fahr;
11
12  for (fahr=LOWER; fahr<= UPPER; fahr=fahr+STEP)
13  printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
14  }
```

En éste caso damos información al programador sobre que significa, los valores LOWER, UPPER y STEP, siendo libre de definir los valores cualesquiera, y aportando información del significado del bucle *for.*

Caracteres de entrada y salida

Se van a considerar a continuación varios programas para procesar información de entrada o salida.

El modelo de entrada/salida proporcionado por la biblioteca estándar es simple: el texto de entrada/salida se trata como *corriente de caracteres (text stream)*. Se trata de una secuencia de caracteres dividida en líneas.

La librería estándar de C proporciona algunas funciones para leer o escribir caracteres, siendo *getchar* y *putchar* las más simples. En el momento de ser llamado, *getchar* lee el carácter de entrada de una corriente de texto y devuelve su valor, es decir:

c=getchar();

La variable c contiene el siguiente carácter de entrada. Dichos caracteres pueden proceder tanto del teclado como de otro programa.

Por otro lado, putchar (c) muestra el carácter, normalmente en la pantalla.

Copiado de archivos

Dadas *getchar* y *putchar*, se pueden escribir códigos muy útiles conociendo sólo valores de entrada y salida. El ejemplo más simple es un programa que lea los valores de entrada y los adquiera como salida al mismo tiempo:

leer un carácter
while(el carácter no es el indicador del final del archivo)
mostrar el carácter leído
leer un carácter

Traduciendo al lenguaje C se tiene:

```
1   #include <stdio.h>
2   /*convertir la entrada en salida*/
3   main()
4   {
5
6   int c;
7
8   c=getchar();
9   while(c!=EOF){putchar (c);
10  c=getchar();
11  }
12  }
```

Donde el operador != significa “no igual a”

Los caracteres aparecidos en la pantalla son almacenados interiormente como un patrón de bits. El tipo *char* esta específicamente destinado para almacenar caracteres, pero también se pueden utilizar enteros: se usó *int* para definir c por una razón importante.

El problema es distinguir el final de la entrada de dato. La solución es que *getchar* devuelva un valor distinto cuando no haya más entradas, un valor que no pueda ser confundido con un carácter real. Éste valor es llamado EOF, para “final del archivo” (*end of file*). Se puede declarar c para ser lo suficientemente grande para *getchar*. No podremos usar *char* que hagan c tan grande como EOF añadiendo cualquier *char* posible. Por ello se usa *int*.

EOF es un entero definido en <stdio.h>, cuya función es indicar que no hay más información que recuperar de una fuente de datos (en caso de un fichero, que se llegó al final del mismo, y de un flujo de datos, que se finalizó la transferencia de datos)

En definitiva, lo que se consigue en este programa es filtrar los datos de entrada, devolviéndolos como salida si no son estrictamente iguales a EOF.

Una forma más compacta puede ser:

```
#include <stdio.h>

/*Copiar entrada-salida*/
main()
{int c;
while ((c=getchar())!=EOF)
putchar(c);
}
```

Cuando *while* toma un carácter, asignado a c, lo compara con EOF. Si no se cumple la condición se ejecuta el bucle, mostrando el caracter y esperando otro valor de c (*getchar(c)*).

Contador de caracteres

El siguiente programa cuenta caracteres, similar al programa de copiar:

```
#include <stdio.h>

/* contador de caracteres*/

main()
{
long nc;

nc = 0;
while (getchar () != EOF)
++nc;
printf ("%ld\n", nc);

}
```

En ésta sintaxis, se aparecen nuevos comandos:

La declaración ++nc, presenta un nuevo operador, ++, que significa *incrementado por uno*. Es equivalente a nc=nc+1; pero la sintaxis aparece más clara y concisa de ésta manera. Otros operadores de la misma clase sería –nc, que decrementa el valor en uno.

Es importante analizar el uso de la variable tipo *long*, la cual acumula números enteros de hasta 32 bits. Aunque en prácticamente todas las máquinas, *long* e *int* son 32 bits, en otras nos podemos encontrar con que *int* es de 16 bits (valor máximo de 32767), por lo que el contador se satura rápidamente.

Puede ser posible copiar mayores números usando *doublé* (*float* de doble precisión). También se puede plantear el problema mediante la declaración *for*:

```
1  #include <stdio.h>
2
3  /* contador de caracteres de entrada*/
4
5  main ()
6  {double nc;
7  for (nc=0; getchar() !=EOF; ++nc)
8  ;
9  printf("%.0f\n", nc);
10 }
11
```

printf una %f tanto para *float* como para *double,* y %0.f para suprimir la parte decimal, que es cero.

El cuerpo del bucle *for* permanece vacío, ya que todo el trabajo y el incremento de nc se plantea dentro de la condición lógica del *for*. Pero como las condiciones gramaticales de C indica que debe tener un cuerpo, es necesario poner punto y coma, ";" (se denomina *declaración nula,* línea 8) .

Contador de líneas

El siguiente programa cuenta os saltos de línea de entrada a partir de una texto de entrada.

```
1  #include <stdio.h>
2
3  /* contador de lineas de datos intrantes*/
4
5  main()
6  {int c, nl;
7
8  nl=0;
9  while ((c=getchar()) !=EOF)
10 if (c == '\n')
11 ++nl;
12 printf ("%d\n", nl);
13 }
```

El cuerpo del *while* consiste en un *if*, el cual controla el incremento ++nl. La declaración *if* controla la condición entre paréntesis, ejecutando la condición o condiciones que le siguen mientras que se cumpla la condición.

El doble igual (==, línea 10) es la notación en C de "igual a", usado para comparar valores. Éste símbolo se ha de distinguir del igual simple (=), que se usa para asignar valores a variables (línea 8).

El carácter escrito entre comillas simples (' ') representa un valor entero igual al valor numérico del carácter en el código máquina del computador. Éste es llamado como *carácter constante (character constant)*, aunque es sólo otra manera de escribir un entero pequeño.

Por ejemplo, el carácter constante 'A', en el código ASCII su valor es 65, que no es otra cosa que la representación interna (de la propia computadora) de la letra A.

En la línea 10 usamos ésta notación para el carácter constante del salto de línea, '\n', de modo que el contador se incrementa cada vez que detecta el salto de línea, entendiendo éste como su valor en el código ASCII (que es 10, en éste caso).

Contador de palabras

El siguiente programa, cuenta las palabras que aparecen en un texto, definiendo las palabras como una secuencia de caracteres que no contienen un espacio, una tabulación, o un salto de línea. De ésta manera distinguiremos las palabras.

```
1  #include <stdio.h>
2
3  #define IN 1 /* dentro de una palabra*/
4  #define OUT 0 /*fuera de una palbra*/
5
6  /* contador de palabras y caracteres de entrada*/
7  main()
8  {
9      int c, nl, nw, nc, state;
10
11     state = OUT;
12     nl=nw=nc=0;
13     while ((c=getchar()) !=EOF) {
14         ++nc;
15         if(c=='\n')
16         ++nl;
17         if (c==' ' || c=='\n' || c=='\t')
18         state = OUT;
19         else if (state == OUT){
20             state = IN;
21             ++nw;
22         }
23     }
24     printf("%d %d %d\n", nl, nw, nc);
25 }
```

Cada vez que el programa encuentra el primer carácter de una palabra, cuenta una palabra más. La variable *state* registra cuándo el programa se encuentra en una palabra o no; inicialmente es "no en una palabra", que asignamos como el valor OUT (líneas 3, 4 y 11). Es preferible el uso de variables simbólicas (IN y OUT) en vez de 1 y 0 para mejor comprensión del programa, no sólo para éste, sino con orientación a comprender programas más grandes y complejos.

La línea 12 (nl=nw=nc=0), declara las tres variables con valor (inicial) cero.

En la línea 17, los operadores | | significan *"o"* (*or*), de modo que:

```
17          if (c==' ' || c=='\n' || c=='\t')
```

Significa que si c es igual a un espacio **o** es igual a un salto de línea **o** es igual a una tabulación, y sigue con el cuerpo del *if*. Existe también un el operador "y", usado de la misma manera, para condiciones en las que se requiera el cumplimiento de varias declaraciones, y cuya sintaxis es &&. Es importante saber, que las condiciones se evalúan por orden de aparición, de izquierda a derecha, de modo que en cuanto no se cumpla una de las condiciones, se deja de evaluar las demás se ejecuta el cuerpo. Por ejemplo, en el caso anterior, si se detecta que c=='' (la primera condición), ya no se evalúan las demás.

En el ejemplo aparece la declaración *else*, que especifica una acción alternativa para cuando la condición lógica del *if* es falsa. La sintaxis es de la manera:

```
if (expresión);
        declaración 1;
else
        declaración 2;
```

Una y sólo una de las declaraciones asociadas con *if/else* será ejecutada. Si la expresión es verdadera, la declaración 1 se ejecuta; si no, se ejecuta la declaración 2, pudiendo ser dichas declaraciones (1 y 2) una declaración simple o varias declaraciones agrupadas. En el caso anterior, el uso de *else* (línea 19) contiene a declaración con *if*, la cual controla dos declaraciones entre llaves (líneas 20 y 21).

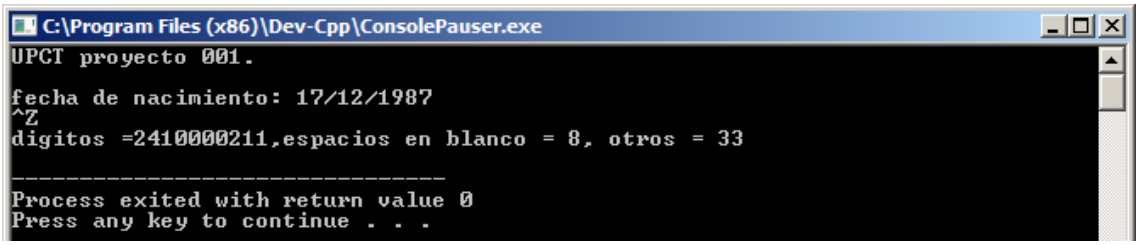
Arrays

A continuación se muestra una programación para contar el número de apariciones de cada dígito, espacio en blanco (espacio, tabulación o nueva línea), y todos los demás caracteres. El objetivo será seguir mostrando aspectos del lenguaje C.

Se tienen doce categorías de entradas (números del 0 al 9, espacios en blanco o demás caracteres), por lo que es conveniente usar un *array* (traducido como "matriz") para contar el número de apariciones de cada digito, en lugar de variables individuales.

```
1   #include <stdio.h>
2
3   /* constador de digitos, espacios en blanco, y otros caracteres*/
4   main()
5   {
6       int c, i, nblanco, notros;
7       int ndigitos[10];
8
9       nblanco = notros = 0;
10      for (i=0; i<10; ++i)
11      ndigitos[i]=0;
12
13      while ((c=getchar()) !=EOF)
14      if(c>='0' && c<='9')
15      ++ndigitos[c-'0'];
16      else if (c== ' ' || c=='\n' || c=='\t')
17      ++nblanco;
18      else
19      ++notros;
20      printf("digitos =");
21      for (i=0;i<10; ++i)
22      printf ("%d", ndigitos[i]);
23      printf (",espacios en blanco = %d, otros = %d\n", nblanco, notros);
24  }
```

Resultando al compilar y ejecutar el programa, y tras escribir un breve texto:



Se observa el resultado del recuento de dígitos (en orden de 0, 1, 2, etc) en una matriz, espacios en blanco, y el resto de caracteres, en éste caso, letras del texto. La expresión ^Z sirve en el programa de ejecución para finalizar el fichero, detectando la función EOF, y ejecutando el final del programa.

La declaración de la línea 7:

```
7       int ndigitos[10];
```

declara que la variable *ndigitos* es una matriz (*array*) de 10 enteros. Los valores del Array siempre comienzan en cero, cuyos elementos se pueden llamar mediante *ndigitos[0], ndigitos[1]…ndigitos[9]*. Esto se realiza en el bucle *for* en las líneas 10 y 11 (donde se realiza el recuento) y en las 21 y 22 para mostrarlas en la pantalla.

Éste programa se basa en el las propiedades de la representación en forma de carácter de los números (dígitos). En la línea 14 y 15:

```
14      if(c>='0' && c<='9')
15      ++ndigitos[c-'0'];
```

Se determina que el carácter leído "c" es un digito, cuyo valor numérico del dígito es c-'0'. Esto trabaja sólo si '0', '1',..., '9' aparecen, incrementando sus contadores.

La decisión de si un carácter es un dígito, espacio en blanco u otro, se toma en las líneas 14 a 19, con la estructura de tipo:

if (condición$_1$)
    declaración$_1$
else if (condicion$_2$)
    declaración$_2$
...
...
else
    declaración$_n$

Ocurre frecuentemente en programas en los que se tomas diferentes vías en función de resultados o valores de entrada determinados. Las condiciones son evaluadas en orden de aparición, de arriba hacia abajo, hasta que una condición es satisfecha, ejecutando su correspondiente declaración (o declaraciones), finalizando toda la estructura del *if*. Si no se cumple ninguna de las condiciones, se ejecuta la declaración del último *else*. En caso de no existir el último *else*, el programa no realizará ninguna acción. La estructura del tipo *else if* (condición)/declaración, se podrá realizar tanta veces como se necesite, haciendo el cuerpo de la estructura *if/else* todo lo largo que se requiera.

Como alternativa al uso de *if/else*, C++ presenta la declaración *switch*, que se verá más adelante, con el objetivo simplificar estructuras en casos en los que se presentan varios *if/else* anidados.

Funciones

Una función proporciona un camino para encapsular un cálculo computacional, el cual pueda ser usado sin importar su aplicación. La propiedad que destaca de las funciones es que no importa cómo trabajan, sino que conociendo qué es lo que hacen, es suficiente. C proporciona ejemplos con una serie de funciones fáciles, convenientes y eficientes, que son llamadas alguna vez.

Anteriormente, se han usado funciones como *printf, getchar* y *putchar*, proporcionadas por la librería estándar. Lo verdaderamente interesante es realizar funciones básicas que resuelvan operaciones frecuentes al usuario.

A continuación se muestra un programa con el que obtener la potencia de números enteros, tipo X^Y, aunque la librería estándar de C ya proporciona una función para ello (*pow(x,y)*).

```
1   #include<stdio.h>
2
3   int potencia(int m, int n);
4   /*funcion para obtener la potencia de numeros enteros*/
5   main()
6   {
7   int i;
8
9   for (i=0;i<10;i++)
10  printf("%d %d %d\n", i, potencia(2,i), potencia (-3,i));
11  return 0;
12  }
13  /*potencia: elevar base a la n-esima potencia, con n>=0*/
14  int potencia(int base, int n)
15  {int i, p;
16
17  p=1;
18  for (i=1;i<=n; ++i)
19  p=p*base;
20  return p;
21  }
```

Vemos que *main()* usa la función potencia en la línea 10. Aunque se encuentran en el mismo programa, se trata de dos partes la cual la segunda es independiente de la primera (ya que la segunda parte es la definición de la propia función)

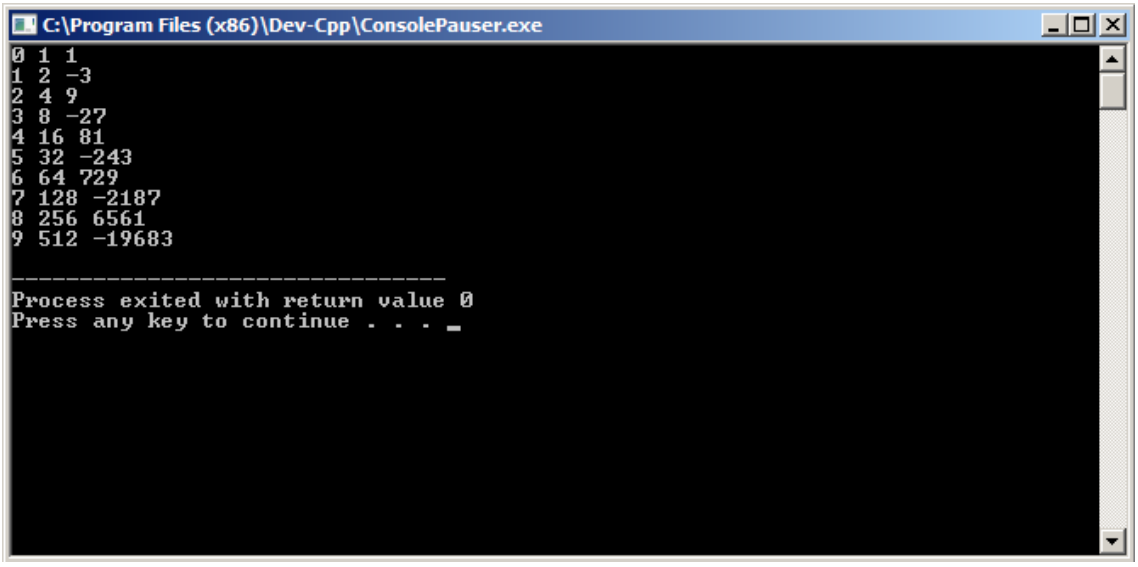
La definición de una función tiene la forma:
retorno-tipo de función-nombre (parámetro declarado, si algo)
{
declaraciones-sentencias
}

La definición de las funciones pueden aparecer en cualquier orden, y en uno o varios archivos, sin embargo, una función no puede ser separada en varios archivos. Si el programa al que se recurre proviene de varios archivos, se podría tener problemas a la hora de compilar, pro sólo por problemas de potencia. Por el momento, no se asumirá ésta posibilidad, y que ambas funciones provienen del mismo archivo.

La función *potencia* es llamada dos veces por *main()* en la línea 10:

```
10  printf("%d %d %d\n", i, potencia(2,i), potencia (-3,i));
```

Cada llamada produce dos argumentos a la función potencia, devolviendo en cada momento el entero con su formato en la pantalla:



La primera línea de la función *potencia* (línea 14):

```
14    int potencia(int base, int n)
```

Declara el tipo y nombre de los parámetros usados, y el tipo de resultado que la función *potencia* devuelve. Es importante señalar que los nombres usados por las variables son locales, es decir, no son visibles fuera de la función *potencia*. Esto quiere decir que, fuera de la función, podremos declarar variables con el mismo nombre sin conflictos. En el caso del ejemplo, se observa el uso de la variable *i* tanto en la función *potencia* como en el programa *main()*, siendo independientes una de otra.

El valor que *potencia* genera es devuelto a *main()* mediante la sentencia *return.* Cualquier expresión puede seguir a *return:*

return expresión;

Es necesario remarcar que, hasta ahora, no se había usado *return* en los programas anteriores *main*. Sin embargo, la declaración de *return* debe devolver, como último valor, el cero. De ésta manera el programa finaliza, evitando condiciones de terminación inusuales o erróneas.

Volviendo a la declaración de la línea 14, justo antes de *main* se indica que la función *potencia* es una variable *int* calculada a partir de dos argumentos *int*, la base y el exponente. Ésta declaración, llamada *función prototipo*, debe concordar con la definición y usos de la función *potencia*.

Por otro lado, los nombres de loa parámetros no tienen por qué coincidir. De hecho, en la *función prototipo potencia* los nombres son totalmente opcionales. La tendencia es a denominar las variables de forma que la función se fácilmente interpretable.

Arrays de caracteres (matriz de caracteres)

El tipo de matriz más común en C es el *array* de caracteres. Para ilustrar el uso de matrices de caracteres o funciones para manipularlas, se va a analizar una simple programación que, a partir de trozos de texto, seleccione el más largo. El programa deberá:

```
while (nueva línea de caracteres)
        if (es más larga que la anterior más larga)
        (guardar la nueva más larga)
        (guardar su longitud)
Print (la línea más larga)
```

Plantear la programación de ésta manera, ayudará a encontrar el camino más eficiente para realizar la operación, evitar errores de sintaxis, o simplemente, no perderle el sentido a lo que se pretende hacer.

Se comenzará escribiendo la función *getline* que analizará las líneas de entrada. La intención es terminar en un programa que se pueda usar en otros contextos. Como mínimo, *getline* tendrá que devolver la señal de final de archivo (*End Of File*), pero para hacerlo más plausible, un mejor diseño podría devolver la longitud de la línea, o cero si el final del archivo es encontrado. Cero es una valor aceptable para *End-Of-File* porque nunca será válido como longitud de línea. En todas las líneas de texto aparece al menos un caracter.

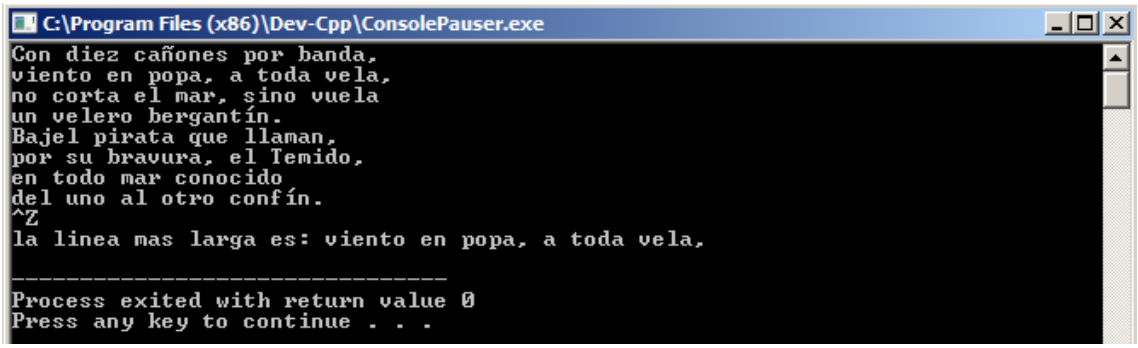
Cuando se encuentre una línea más larga que la anterior, se guardará la nueva más larga. Por lo que se prevé necesitar una función que copie dicha línea.

Finalmente, necesitamos un programa principal (*main*) que controle *getline* y *copy*. El resultado es:

```
#include <stdio.h>
#define LINEAMAX 1000 /*máxima longitud de línea admisible*/

int getline (char linea[], int lineamax);
void copy (char a[], char desde[]);

/*mostrar línea de entrada más larga*/
main()
{
        int longitud; /*long de la línea actual*/
        int maximo;   /*maxima longitud registrada*/
        char linea[LINEAMAX];   /*línea de entrada*/
        char maslarga [LINEAMAX];   /*maxima linea guardada*/

        maximo=0;
        while ((longitud=getline(linea, LINEAMAX))>0)
        if(longitud>maximo){
                maximo=longitud;
                copy(maslarga, linea);
        }
```

```
        if (maximo>0)
        printf("la linea mas larga es: %s", maslarga);
        return 0;
}

/*getline: leer una linea y devolver su longitud*/
int getline (char s[], int lim)
{
        int c, i;

        for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i]=c;
        if (c=='\n') {
                s[i]=c;
                ++i;
        }
        s[i]='\0';
        return i;
}

/*copy: copiar "desde" a "de"*/
void copy(char de[], char desde[])
{
        int i;
        i=0;
        while ((de[i]=desde[i]) !='\0')
        ++i;
}
```

Compilando y ejecutando, se comprueba que funciona el programa:



Las funciones *getline* y *copy* son declaradas al principio del programa, asumiendo que son que están contenidas en un archivo.

Main y *getline* comunican a través de una pareja de argumentos un valor de retorno. En *getline*, los argumentos son declarados en la línea:

int getline (char s[], int lim);

Que especifica que el primer argumento, s, es una matriz (*array*), y el segundo, *lim*, un entero. El propósito de sustituir el tamaño de la matriz en la declaración es dejar a un lado el almacenamiento. La longitud de la matriz s no es necesario en *getline* sino en el principal, *main*. *Getline* usa *return* para mandar un valor de vuelta a *main*.

Algunas funciones devuelven un valor para usarlo, pero otras, como *copy*, son usadas por su efecto, sin devolver valores. El tipo de retorno del *copy* es *void* (nulo), que especifica que ningún valor es devuelto.

Getline usa el carácter '\0' (carácter nulo o *null character*, cuyo valor es cero) al final de ensamblar la matriz, para señalar el final de la línea. Por ejemplo, una sucesión de caracteres como "HOLA\n" aparece en el programa C, como una matriz de caracteres que contienen "\0" al final para señalar el final:

H O L A \n \0

%s en el formato de *printf* indica que el correspondiente argumento será una cadena de caracteres.

Valores externos y aplicación

Las variables en *main*, como *line, maslarga, etc* en el programa anterior, son locales o privadas para *main*. Ninguna otra función tiene acceso directo a ellas. Otras, aparecen en dos funciones de forma independiente, como "i" en las funciones *getline* y *copy*, no teniendo ninguna relación entre ambas. Cada variable se crea en el momento en que se llama a la función, y deja de existir cuando dicha función acaba. Dichas variables pueden ser definidas como *variables automáticas*, a las variables locales.

El por qué las variables automáticas van y vienen cuando la función es llamada, es para que no retengan valores de anteriores llamadas, ya que dichos valores puede tener información que lleven a una operación errónea en la función.

Como alternativa a las variables automáticas, se pueden definir variables *externas* a todas las funciones, es decir, variables a las que tienen acceso todas las funciones a través de su nombre. Puesto que dichas variables son accesibles por todas las funciones, pueden ser usadas en listas de argumentos de varios programas a la vez. Además, éstas pueden conservar los valores en el tiempo, incluso después de que las funciones finalicen.

Una variable externa puede ser *definida* fuera de cualquier función. La variable puede ser *declarada* en cada función que la necesite, indicando el tipo de variable. La declaración puede ser una explícita declaración *externa (extern)* o puedes ser implícita por el contexto. Aplicando este concepto al programa anterior, se va a reescribir el programa con *línea, maslarga,* y

máximo como variables externas, lo que requerirá cambios en las llamadas, declaraciones, y cuerpo de las tres funciones: *main, getline* y *copy*.

```
#include <stdio.h>
#define MAXLINE 1000  /*maxima longitud permitida*/

int maximo;
char linea[MAXLINE];
char maslarga[MAXLINE];

int getline(void);
void copy(void);
/*mostrar linea mas larga*/
main()
{int longitud;
extern int maximo;
extern char maslargo[];

maximo=0;
while ((longitud=getline())>0)
if(longitud>maximo){
        maximo =longitud;
        copy();
}
if (maximo>0)  /*es una linea*/
printf("%s",maslarga);
return 0;
}
/*función getline*/
int getline(void)
{
int c,i;
extern char linea[];

for (i=0; i<MAXLINE-1 && (c=getchar)) !=EOF && c !='\n'; ++i)
linea[i]=c;
if (c=='\n'){
        linea [i]=c;
        ++i;
}
linea [i]='\0';
return i;
}
/*copiar*/
void copy(void){
```

```
        int i;
        extern char linea[0], maslarga[];
        i=0;
        while ((longest[i]=linea[i]) !='\0')
        ++i;
}
```

Las variables externas en *main, getline* y *copy* son definidas en las primeras líneas del ejemplo. Sintácticamente, las variables externas son definidas de la misma forma que variables locales, pero fuera de las funciones. Antes de que una función use una variable externa, el nombre de la variable debe ser conocido por la función, por lo que se ha de declarar la variable en el programa, añadiendo *extern* antes de la declaración.

En ciertas circunstancias, la declaración *extern* puede ser omitida. Si la definición de la variable externa ocurre en un archivo fuente, antes de su uso en una función particular, no es necesaria la declaración *extern*. La declaración *extern* en *main, getline* y *copy* es redundante. De hecho, una práctica habitual es situar todas las variables externas en el principio del programa, y luego omitir todas las declaraciones externas.

Si un programa está hecho a partir de varios archivos fuente, y las variables están definidas en el archivo1 y son usadas por archivo2 y archivo3, será necesaria la declaración *extern* en los archivos 2 y 3. Es también práctica habitual recopilar todas las variables externas en un archivo denominado *encabezado (header)*, el cual será incluido a través de *#include* al principio de la función.

3.Introducción a C++

Introducción

C++ es un lenguaje de programación desarrollado por tomando como base el lenguaje C. Básicamente, el lenguaje C es un lenguaje libre estandarizado sencillo (ISO/IEC 9899:1990), muy útil a la hora de programar software de sistemas o algunas aplicaciones. En el caso de C++(ISO/IEC 14882:2011), su programación está basada en C (o mejor dicho, extensión de C), solo que cuenta con mecanismos que permitan la manipulación de objetos (OOP, programación orientada a objetos). Otra de sus principales mejoras es que permite la programación multiparadigma: permite crear programas usando más de un tipo de programación (uso del mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la manera más eficiente).

Para apreciar las diferencias entre C y C++, se planteará un programa que nos indique "UPCT proyecto. Primer programa en C++":

```
1  /*primer programam en C++*/
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6      cout<<"UPCT proyecto: ";         //Muestra "UPCT proyecto: " en pantalla
7      cout<<"primer programa en C++"; //Muestra "mi primer programa en C++"
8      return 0;
9  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
UPCT proyecto: primer programa en C++
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

A primera vista, la estructura es, en general, muy similar a los programas vistos en C, sólo que con variaciones en el nombre de varias funciones, operadores, etc.

En el caso del lenguaje C++, al igual que C, se hace referencia a la librería básica estándar de entrada/salida, siendo para C++ *iostream* (línea 2). Es importante recordar que, al igual que en C, las líneas que comienzan con una almohadilla (#) son directivas para el preprocesador, es decir, instrucciones previas que se dan al procesador antes de compilar y ejecutar el programa.

La instrucción *using namespace std*, en la línea 3, es necesaria para acceder a los elementos de la librería estándar (en éste caso, *std*). Con ésta expresión, estamos indicando que vamos a usar éstas entidades. Ésta línea es frecuente en programaciones en C++.

La línea 4, *int main(),* muestra en nombre e inicio del programa principal. Ésta sintaxis es igual a C: se declara el nombre de la función y a continuación sus instrucciones entre llaves, {}.

En la línea 6 y 7, aparece la declaración *cout*<<. Ésta está declarada en el archivo estándar *iostream*, en el espacio de nombres *std* (de ahí la instrucción *using namespace std*). Su efecto es muy similar a *printf* en el lenguaje C, pero su sintaxis distinta.

Por último, es necesario establecer la instrucción *return 0* (línea 8) para finalizar la función *main()*, y/o finalizar la consola C++.

En C++, los comentarios se pueden introducir bien a través de /*"comentario"*/ o //"comentario". La diferencia entre ambas es que la primera puede ocupar varias líneas (se suele utilizar en los encabezados de las funciones), mientras que la segunda sólo admite el comentario en la misma línea (su uso se orienta a las aclaraciones acerca de los pasos u operaciones que realiza la función).

Por último, al igual que en C, es importante tener en cuenta la estructura del programa, así como la claridad, simplificación y optimización de las operaciones a la hora de programar. Una estructura ordenada, nos dará facilidades a la hora de corregir errores, detectar fallos o aclararnos con lo que se pretende hacer.

Variables y tipos de datos.

Introducción

Para entender el significado de variable, en el caso de una programación sencilla, se puede hacer un pequeño juego mental. Comenzamos memorizando el número 5 por un lado, y después el 2 por otro, tenemos dos valores distintos retenidos en nuestra cabeza. Ahora, al primer valor sumamos 1, por lo que ahora retenemos el 6 y el 2. Si después los restamos, obtenemos el 4. Éste proceso mental, es similar al que hará una computadora con 2 variables. Dicho proceso, expresado en C++, sería:

```
1   a=5;
2   b=2;
3   a=a+1;
4   resultado=a-b;
```

Por lo tanto, se puede entender una variable como una porción de memoria que destina la computadora a retener un valor determinado.

Identificadores

Cada variable necesita un identificador para distinguirla de otra. Viendo el ejemplo anterior, se cuentan hasta 3 variables: *a, b* y *resultado*. Una variable puede tener (casi) cualquier nombre que se quiera, respetando las reglas del propio lenguaje C++.

Un identificador válido es una secuencia de 1 o más letras, dígitos o caracteres de subrayado (_), no siendo aceptados ni espacios ni signos de puntuación. También tienen la característica de que siempre deben empezar por una letra. También es posible comenzar con _, pero puede dar problemas si coincide con comandos específicos del compilador o identificadores externos. Por último, nunca comenzarán usando un dígito.

Otra regla a considerar cuando se propone un identificador, es que no puede coincidir con comandos específicos estándar del lenguaje C++. Dichos comandos estándar reservados son:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum, explicit,
export, extern, false, float, for, friend, goto, if, inline, int, long,
mutable, namespace, new, operator, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct,
switch, template, this, throw, true, try, typedef, typeid, typename, union,
unsigned, using, virtual, void, volatile, wchar_t, while.
```

Adicionalmente, existen representaciones alternativas para algunos operadores que no podrán ser usados como identificadores, ya que pueden estar reservadas bajo determinadas circunstancias.

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor,
xor_eq.
```

También es importante señalar, que el lenguaje C++ distingue entre mayúsculas y minúsculas. Por lo tanto no podremos hacer referencia a la variable "RESULTADO" a través de la llamada "resultado" o "Resultado". Esto es debido a que el código ASCII (Apéndice II), utilizado por C++, asigna distintos códigos a mayúsculas que a minúsculas.

Tipos de datos fundamentales

Cuando se programa, se almacenan variables en la memoria de la computadora, pero la computadora no reconoce el tipo de datos que estamos guardando. Por otro lado, distintos tipos de variables demandan distintas cantidades de memoria. No requerirá lo mismo almacenar un número entero que un número decimal.

La memoria de las computadoras está organizada en bytes. Un byte, es el mínimo de memoria que se podrá usar en C++. En dicho byte, tenemos la capacidad necesaria para almacenar un entero pequeño (entre 0 y 255) o un simple carácter. Por otro lado, la computadora podrá combinar los bytes para poder almacenar números más largos o números no enteros.

De acuerdo con lo anterior, C++ propone una serie de tipos de datos fundamentales correspondientes a las unidades básicas de almacenamiento:

-Tipo Booleana (*bool*)

-Tipo carácter (*char*)

-Tipo entero (*int*)

-Tipo punto-flotante (*double*)

Como complemento, el usuario podrá definir:

-Enumeración de datos para representar ciertos valores específicos.
Por otro lado, se define:
-Tipo *void*, que indica la ausencia de información.
A partir de los tipos anteriores, se podrán construir:
-Tipo indicador o puntero (*int**)
-Tipo matriz (*Array o char []*)
-Tipo referencia (*double&*)
-Estructuras de datos y clases.

Los de tipo Booleano, carácter y entero se llaman colectivamente de tipo *integral*. El tipo integral y punto flotante se denominan en conjunto de tipo *aritmético*. Enumeraciones y clases se denominan como de tipo *definido por usuario*. Por otro lado, el resto de variables de denominan de tipo *construidas*.

Tipo booleano

Un booleano, *bool*, puede tener uno valor de dos posibles: verdadero (*true*) o falso (*false*). Un booleano se usa para expresar resultados de operaciones lógicas. Por ejemplo:

```
1   void f(int a, int b)
2   {
3       bool b1=a==b    //"="es asignación y "=="es igualdad
4       ...
5   }
```

Si *a* y *b* tienen el mismo valor, *b1* será *verdadero*, si no son iguales, entonces *b1* será *falso*.
Un uso común de *bool* es como tipo de resultado de una función que testea una condición (o predicado). Por ejemplo:

```
1   bool greater(int a, int b) //testear si a>b
2   {
3       return a>b;
4   }
```

Por definición, verdadero tiene asignado el valor 1, y falso el 0. Por tanto, es posible convertir enteros en booleanos de forma implícita: valores distintos de cero son verdaderos, e iguales a cero, falsos:

```
1   bool b=7;     //bool(7) es verdadero, por lo que b es verdadero
2   int i=true;   //int(true) es 1, por lo que i=1
```

Por otro lado, en expresiones aritméticas o lógicas, *bool* se convierte en *int*, es decir, *bool* actúa con los valores 0 y 1(o valores asignados como verdaderos) en éste tipo de operaciones:

```
1   void g()
2   {
3       bool a=true
4       bool b=true
5
6       bool x=a+b;  //si a+b es 2, entonces x es verdadera
7       bool y=a/b;  //si a/b es 1, entonces y es verdadera
8   }
```

Tipo carácter

Una variable tipo *char* puede contener un carácter del código ASCII adaptado, por ejemplo:

char a=A;

Universalmente, un *char* tiene 8 bits, el cual puede tener hasta 256 valores diferentes. Dicho valor será una variante de la norma ISO-646, haciendo referencia a los caracteres que aparecen en el teclado del ordenador. Éste está parcialmente estandarizado debido a las variaciones de idioma y distintos caracteres del mismo (por ejemplo, el código adaptado al español contiene la "ñ y Ñ").

En C++, se asume que el código implementado contiene los dígitos decimales, los 26 caracteres del alfabeto Inglés (en minúsculas y mayúsculas), y algunos signos de puntuación usuales. Aunque C++ acepta cualquier carácter, es importante usar caracteres universalmente aceptados generalmente, sobre todo en casos de exportar un programa a otro país. También es necesario que el código reconozca los caracteres { } [ ] | \; ya que en algunos países no es así.

Cada caracter se corresponde a un valor numérico entero. Por ejemplo, a la "b" le corresponde el valor 98 en el código ASCII. En el ejemplo siguiente podemos obtener el número entero de cada carácter:

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       char c;
6       std::cin>>c;
7       std::cout<<"el valor de "<<c<<" es "<<int(c);
8   }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
A
el valor de A es 65
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

La notación *int (c)* devuelve el valor del enteros asociado al carácter asignado a la variable *c*. La posibilidad de convertir un carácter a un valor entero, lleva a la pregunta: ¿qué signo tiene el valor entero asociado? Los 256 valores que se pueden representar en 8 bits se pueden interpretar como de 0 a 255 (*unsigned char)* o de -127 a 127 (*signed char*). Afortunadamente, los caracteres más comunes se encuentran en el rango de 0 a 127.

Caracteres literales

Un carácter local, a veces llamado carácter constante, es un carácter encerrado entre comillas simples ('a', '0'). El tipo de un carácter literal es *char*. De hecho, los caracteres literales con en realidad constantes simbólicas con el valor numérico entero de su correspondiente casilla en el código máquina. El uso de caracteres literales en vez de notación decimal, hace los programas más "portables".

Tipo entero

Al igual que *char,* los de tipo entero pueden aparecer en tres formas: simple *int, signed int*, y *unsigned int*. Éstos también pueden venir en tres tamaños: *short int, int* simple, y *long int*. Como simplificación, *signed, unsigned, long o short* siempre están referidos a valores enteros, si no se indica el tipo de variable.

Enteros literales

Los enteros literales pueden aparecer en cuatro formas: decimal, octal, hexadecimal, o como caracteres literales. Los más comunes son los de tipo decimal (0, 22, 3578, etc). Para el uso de la base hexadecimal (base 16), tendremos que citar el número precedido de "0x". En el caso de referirnos a un número octal (base 8), se escribirá un 0 precedido del número. De ésta manera:

Decimal	0	2	63	83
Octal	00	02	077	0123
Hexadecimal	0x0	0x2	0x3f	0x53

*Las letras *a, b, c, d, e,* y *f* son usadas para representar los dígitos 10, 11, 12, 13, 14 y 15 en la base hexadecimal.

Por último, se pueden añadir sufijos a los enteros literales, de manera que:

Sufijo	Ejemplo	Significado
U	*3U*	Entero sin signo (*unsigned int*)
L	*5L*	Entero largo (*long int*)

Tipo punto flotante

Representan números punto-flotante. Al igual que los enteros se pueden encontrar en tres tamaños: *float* (precisión simple), *double* (doble precisión), y *long double* (precisión extendida), en función de la precisión que se requiera:

Nombre	Descripción	Tamaño	Rango
float	Numero punto flotante	4 bytes	+-3e(+-38) ≈ (7digitos)
double	Numero punto flotante de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)
long double	Número punto flotante largo de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)

Tipo flotante literal

Por defecto, un tipo flotante literal es un *double*. El compilador, normalmente, advierte si un tipo flotante es demasiado grande para ser representado. Algunas representaciones de estos números pueden ser:

1.12 .23 0.23 1. 1.0 1.2e10 1.23e-15

Es importante no dejar espacios en la definición del número, ya que devolverá un error de sintaxis. Si se precisa de un punto flotante literal de tipo *float*, se añade el sufijo *f* o *F*:

3.14159265f 2.0f 2.997925F

Resumen de variables fundamentales

Nombre	Descripción	Tamaño	Rango
char	Caracter o entero pequeño	1 byte	con signo:-127 a 127 Sin signo: 0 a 255
short int	Entero corto	2 bytes	Con signo: -32768 a 32767 Sin signo: 0 a 65535
int	entero	4 bytes	Con signo: -2147483648 a 2147483647 Sin signo: 0 a 4294967295
long int	entero largo	4 bytes	Con signo: -2147483648 a 2147483647 Sin signo: 0 a 4294967295
bool	Valor booleano: verdadero o falso	1 byte	verdadero (1) o falso (0)
float	Numero de punto flotante	4 bytes	+-3e(+-38) ≈ (7digitos)
double	Numero punto flotante de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)
long double	Número punto flotante largo de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)

Declaración de variables

Introducción

Para el uso de variables en C++, se deberá primero declararlas especificando el tipo de variable. La sintaxis necesaria comienza por indicar el tipo de variable a utilizar (*int, bool, float*, etc) seguido de un identificador válido:

```
1   int a;
2   int A;
3   float fuerza;
4   int b, c, d;
```

Las declaraciones tipo enteros (*char, short, long* e *int*), pueden declararse con o sin signo en el rango de números a utilizar. Las de tipo con signo, pueden representar tanto valores positivos como negativos, mientras que los sin signo sólo representan los valores positivos y cero. Ésta propiedad se declarará antes de declarar el tipo de variable añadiendo *signed* (con signo) o *unsigned* (sin signo) de la forma:

```
1   unsigned short int Eslora;
2   signed int Balance;
3   int Arfada;
```

Por defecto, en la gran mayoría de los compiladores asumen por igual las declaraciones de las líneas 2 y 3 de éste último ejemplo, es decir, *int Arfada* será una variable con signo positivo o negativo, con las mismas características que *signed int Balance*.

Por otro lado, la simple declaración *signed* o *unsigned*, sin especificar el tipo de variable, será equivalente a *signed int* y *unsigned int* respectivamente.

Por último, a continuación se representará una corta programación con operaciones aritméticas sencillas, haciendo referencia al juego mental de la introducción del apartado:

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int a, b;        //declaración de variables
6      int result;
7      a=5;
8      b=2;
9      a=a+1;           //Proceso
10     result=a-b;
11     cout <<result;   //mostrar resultado
12     return 0;        //terminar programa
13 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
4
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Alcance de las variables

En éste apartado se va a analizar el alcance de una variable en función de la zona en la que se declara. Como regla general, una variable que se utilizará a lo largo de todo el programa se declarará en el encabezado del programa, inmediatamente después de las instrucciones al compilador (*variables globales*). Por otro lado, también se pueden precisar variables al ejecutar una función, es decir, se reclama su uso al ejecutar una función, pero carece de significado fuera de ella (*variables locales*):

```
1  #include<iostream>
2  using namespace std;
3
4  int entero;
5  char caracter;                    Variables globales
6  char string [20];
7  unsigned int Tiempo;
8  int main()
9  {
10     unsigned short Edad;          Variables locales
11     float Estatura, peso;
12     cout<<'introduce tu edad; '
13     con<<Edad;                    Instrucciones
14     ...
15 }
```

Como podemos observar en el ejemplo, el alcance de las variables locales está limitado por las llaves, { }, que denotan el principio y fin de la función. En el ejemplo, en el interior de la función podremos llamar tanto a las variables globales y locales. En el caso de una programación en la que intervengan varias funciones, las variables locales sólo tendrán significado dentro de la función declarada:

```
#include<iostream>
using namespace std;

unsigned int Tiempo;
int main()
{
    float Estatura, peso;
    cout<<'introduce tu estatura hace una año; '
...
}
int main2()
{
    float Estatura;
    cout<<'introduce tu estatura actual: '
}
```

En éste caso, se observa la variable *float Estatura;* con el mismo nombre tanto en la función *int main()* e *int main2()*. Sin embargo, al tratarse de variables locales, son totalmente independientes la una de la otra, y no entrarán en conflicto en ningún momento.

Inicialización de variables

Cuando se declara una variable, en principio, su valor es indeterminado hasta que se le asigna uno. Para asignar un valor inicial a las variables, se podrá seguir dos caminos:

El primero, conocido como la forma C, en referencia al lenguaje C, se asigna un valor inicial mediante la secuencia *tipo identificador = valor inicial*. Recordar en éste momento, que un símbolo igual "=" asigna un valor, mientras que doble signo igual "==" compara resultados.

```
int a = 0;
```

Otro camino para inicializar variables, conocido como *inicialización constructora*, es realizado encerrando el valor inicial entre paréntesis de la forma *tipo identificador (valor inicial)*:

```
int a (0);
```

Ambos caminos tienen el mismo significado en C++:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int a=5;
6       int b(2);
7       int result;
8       a=a+3;
9       result = a+b;
10      cout<<result;
11      return 0;
12  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Cadenas de caracteres

Introducción

Se define como cadena de caracteres a aquellas variables de valores no numéricos que son más largos que un solo caracter. EL lenguaje C++ proporciona una librería estándar en la que se define la variable *string*. Ésta no se considera una variable fundamental, pero si es muy usada.

Al no estar incluida en *iostream*, se tendrá que declarar en el encabezado la llamada *#include<string>* para tener acceso al espacio de nombres *std*:

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4   int main()
5   {
6       string micadena="cadena de caracteres";
7       cout <<micadena;
8       return 0;
9   }
10
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
cadena de caracteres
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Por otro lado, la declaración de la variable *string* y la asignación de valores iniciales se realiza de la misma forma que para el resto de tipos de variables, con la cadena de caracteres entre comillas dobles (“ “):

```
1    string micadena="cadena de caracteres";
2    string micadena("cadena de caracteres");
```

Y su comportamiento en una función es totalmente similar. A continuación se muestra un ejemplo en el que se engloba los conceptos de asignación de valores a una variable *string*:

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4   int main()
5   {
6       string micadena("Mi nombre es:");
7       cout <<micadena<<endl;
8       micadena="Juan Antonio Soler";
9       cout <<micadena<<endl;
10      return 0;
11  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Mi nombre es:
Juan Antonio Soler

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Constantes

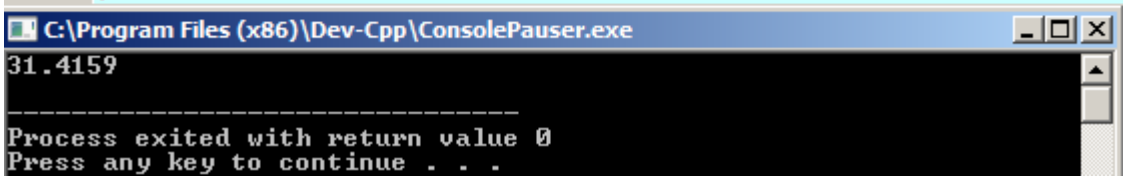
Introducción

En C++, se define una constante como una expresión con un valor fijo. En la definición de los tipos de variables, se vieron el tipo de valores aptos para cada tipo de variable (*int float double, bool, etc*). Éste apartado se centra en la declaración de constantes generales cuyo uso se repita a lo largo de las funciones (por ejemplo, valores de constantes físicas como la gravedad, carga eléctrica de un electrón, o número de Avogadro, o matemáticas como el número e, PI, etc). Dicha directiva se realizará en el encabezado, ya que no es una declaración de C++, sino una instrucción al compilador, de la forma *#define identificador valor*:

```
1    #define PI 3.14159
2    #define NUEVALINEA'\n'
```

El ejemplo define dos nuevas constantes: PI y NUEVALINEA. Una vez definidos, se pueden usar en el resto del código como constantes regulares:

```
#include<iostream>
using namespace std;
#define PI 3.14159
#define NUEVALINEA'\n'

int main()
{
    double r=5.0;    //radio
    double circulo;
    circulo=2*PI*r;
    cout<<circulo;
    cout<<NUEVALINEA;
    return 0;
}
```



Como se ve en el ejemplo, usamos PI como una constante matemática, a la que podemos hacer referencia en todo momento, y NUEVALINEA para los saltos de línea, siendo ésta última útil para no hacer referencia al comando en todo momento: se podrá definir el significado de cada comando para luego hacer referencia al mismo a través del nombre clave. Al igual que en C, en C++ se tienen los siguientes comandos:

\n	Salto de línea	\f	Salto de página
\r	Retorno	\a	Alerta (beep)
\t	Tabulación	\’	Comilla simple
\v	Tabulación vertical	\”	Comilla doble
\b	Backspace	\\	Contrabarra

Como se dijo arriba, *#define* es una directiva al compilador, por lo que no requerirá un punto y coma (;) al final.

Constantes declaradas

Las constantes declaradas son variables a las que se le asigna un valor fijo, a través del sufijo *const*:

```
const int Kilometros=100;
const char tabulador='\t';
```

En el ejemplo, el entero *Kilómetros* y *tabulador* serán constantes, los cuales se podrán dar el trato de variables regulares, pero no se les podrá cambiar el valor.

Operadores

Introducción

Una vez conocidas tanto las variables como las constantes, se puede comenzar a operar con ellas. Para éste propósito, C++ tienen integrados operadores en forma de símbolos (en caso de otros lenguajes, los operadores son caracteres alfabéticos o palabras en inglés). Ésta virtud de C++, hace que sea más internacional, elimina la necesidad de conocer expresiones anglosajonas, y facilita la comprensión de una sintaxis.

Asignación

La asignación proporciona un valor inicial a una variable:

```
1    Int a=1;
```

Ésta declaración asigna el valor 1 al entero a. La parte izquierda se denomina *Lvalue* (*left value,* valor izquierdo) y a la parte derecha *rvalue* (*right value,* valor derecho). *Lvalue* debe ser una variable, mientras que *rvalue* podrá ser una constante, una variable, o el resultado de una operación o combinación de los anteriores. La característica más importante es la regla de "derecha a izquierda": la asignación de variables cumple siempre ésta regla:

```
1    char a=b;
```

A la variable *a* se le asigna el valor *b*, y no al revés. También es importante considerar la posibilidad que, si fuera *b* otra variable declarada anteriormente a *a*, *a* toma el valor de *b* que tiene en el momento de ser declarada. El cambio, más debajo de *b,* no afecta al valor de *a*. El compilador "lee" de arriba hacia abajo, de derecha a izquierda:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int a, b;         //a=?   b=?
6       a=10;             //a=10  b=?
7       b=4;              //a=10  b=4
8       a=b;              //a=4   b=4
9       b=7;              //a=4   b=7
10      cout<<"a="<< a<<'\n';
11      cout<<"b="<< b;
12      return 0;
13  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
a=4
b=7
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

El programa muestra los valores finales de *a* y *b*. En la línea 8 se le asigna a *a* el valor de *b*, por lo que adopta el valor de *b* en ese momento. A continuación, en la línea 9 se le asigna el valor 7 a *b*, mientras que el valor *a* conserva el valor anterior de *b*.

Una propiedad interesante de C++ sobre otros lenguajes es que la operación de asignación puede ser usada como *rvalue* (o parte de *rvalue*) para otra operación de asignación. Por ejemplo:

```
a=2+(b=5);
```

Es equivalente a:

```
b=5;
a=2+b;
```

También es común la siguiente asignación, sobre todo en la inicialización de algún proceso: asigna el valor 0 a las variables *int* (por defecto) *a, b* y *c*.

```
1   a=b=c=0;
```

Operadores aritméticos

Los operadores aritméticos soportados por el lenguaje C++ son:

+	suma	/	división
-	resta	%	modulo
*	multiplicación		

Mientras que los símbolos de suma, resta, multiplicación y división se corresponden a los operadores matemáticos, en el caso de "modulo" se corresponde con el símbolo de porcentaje (%). Dicha operación asigna a una variable el resto de una división:

```
1    a=11%3;
```

En el ejemplo, a la variable *a* se le asigna el valor 2, que es el resto de la división de 11 entre 3.

Asignaciones compuestas

Cuando se busca modificar el valor de una variable para realizar una operación, convirtiéndose ésta en la nueva asignación para esa variable, se recurre a las asignaciones compuestas:

Expresión	Ejemplo	Equivalente a:
+=	a+=1	a=a+1
-=	a-=5	a=a-5
*=	a*=a	a=a*a
/=	a/=b	a=a/b

Éstas expresiones son muy útiles a la hora de realizar estructuras con *while,for* o *if*. También son apreciadas para aportar elegancia a la sintaxis, sintetizando las expresiones al máximo.

Además de las expresiones vistas, existen otras, menos usadas, pero muy potentes en caso de recurrir a ellas:

%=	Asigna el resto de la operación	&=	"Y" y asignación
<<=	Cambio de la izquierda y asignación	|=	"O" inclusivo y asignación
>>=	Cambio de la derecha y asignación	^=	"O" exclusivo y asignación

Incremento y decremento

Los siguientes operadores, el operador de incremento (++) y decremento (--) incrementan o reducen el valor de la variable referida en 1. Son equivalentes a +=1 y -=1 respectivamente. El ejemplo que sigue representa la misma declaración de tres formas distintas:

```
1    c++;
2    c+=1;
3    c=c+1;
```

Una característica del operador ++ y -- es que pueden usarse como prefijos (++a) o sufijos (a++), es decir, antes o después del identificador. Cuando usamos el operador de forma simple (++a; por ejemplo) el efecto es el mismo tanto si se usa como prefijo como sufijo. Sin embargo, en otras expresiones, puede tener un significado distinto, el cual se compara en los siguientes ejemplos:

Ejemplo 1	Ejemplo 2
`B=3;` `A=++B;`	`B=3;` `A=B++;`
El valor de *A* es el mismo que el de *B* incrementado en 1, por tanto: *A=4* y *B=4*	A la variable *A* se le asigna el valor de *B* antes de ser incrementado, por tanto, al final se tiene: *A=3* y *B=4*

Operadores de relación e igualdad

Para evaluar una comparación entre dos expresiones se pueden usar operadores de igualdad y relación. El resultado de la operación de relación es una *Booleano*, es decir, *falso* si no se cumple la relación y *verdadero* si se cumple.

A continuación se listan los operadores de relación e igualdad que se usan C++:

==	Igual a[1]	<	Menor que
!=	No igual a	>=	Mayor o igual que
>	Mayor que	<=	Menor o igual que

Dichas expresiones se pueden usar no solo con valores, sino también con expresiones válidas, incluyendo variables. Por ejemplo, dadas unas variables con valores asignados, obtenemos de las comparaciones *Falso* o *Verdadero (booleano)*:

```
int a=2;
int b=3;
int c=6;
(a==5)        //Devuelve un Falso
(a*b>=c)      //Devuelve un Verdadero
(b+4>a*c)     //Devuelve un Falso
((b=2)==a)    //Devuelve un Verdadero
```

Operadores lógicos

El operador ! es un operador C++ para evaluar la operación booleana *no*. Básicamente indica que si una comparación situada a la derecha de ! no se cumple, devuelve el valor booleano *verdadero*, mientras que si la expresión es correcta, devuelve el valor booleano *Falso*. Por ejemplo:

```
!(5==5)       //Evalua la expresión como Falsa, y que
              //5==5 es verdadero
!(6<=4)       //Evalúa como verdadera la expresion, porque
              //6<=4 es falso
!true         //Devuelve un Falso
!false        //Devuelve un Verdadero
```

[1] No confundir la asignación de valores a las variables (=) con la comparación entre valores (==)

Los operadores lógicos && y || son usados cuando, evaluando dos expresiones, se quiere obtener un solo resultado. El operador && corresponde con la expresión lógica booleana "y". Ésta operación es verdadera si ambos operandos son verdaderos, y falso en cualquier otro caso. A continuación se muestran los posibles resultados de dos variables y la salida obtenida de su comparación con el operador &&:

a	b	a&&b
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

El operador || corresponde con la expresión lógica booleana "o". Ésta operación resulta verdadera cuando alguno de los operandos es verdadero, y es falso sólo cuando todos son falsos. Siguiendo con el ejemplo anterior, se tendrá en éste caso:

a	b	a||b
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Operador condicional

El operador condicional evalúa una expresión devolviendo un valor si dicha expresión es verdadera y otro valor diferente si es evaluada como falsa. El formato es:

```
condición ? resultado1 : resultado2
```

Si *condición* es verdadero la expresión devuelve *resultado1*. Si por el contrario es falsa, devolverá *resultado2*.

Se plantea a continuación el siguiente ejemplo, con el que se comprende mejor su funcionamiento:

```
1   //operador condicional
2   #include<iostream>
3   using namespace std;
4   int main()
5   {
6       int a,b,c;
7       a=2;
8       b=7;
9       c=(a>b)?a:b;
10      cout<<c;
11      return 0;
12  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
7
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

En la línea 9, se define el valor de *c* de la siguiente manera: si *a* es mayor que *b* (*verdadero*), *c* toma el valor de *a*. Por el contrario, si *a* no es mayor que *b* (*falso*), *c* adopta el valor de *b*. Como *a=2<b=7*, la comparación es falsa, por lo que *c* adopta el valor de *b* (7), como vemos en la ejecución del programa.

El operador coma

El operador coma (,) se usa para separar dos o más expresiones que son evaluadas cuando sólo se espera el cumplimiento de una de las expresiones. Cuando el conjunto de expresiones va a ser evaluada para un valor, sólo se considera la que está situada más a la derecha. Por ejemplo:

```
1   a=(b=3,b+2);
```

Podríamos asignar el valor 3 a *b* y después asignar el valor *b+2* a la variable *a*. Sin embargo, la variable *a* contendrá el valor *5* mientras que la variable *b* contenga el *3*.

Operadores bit a bit (*bitwise*)

Los operadores bit a bit modifican variables considerando los patrones bit que representan las acciones asignadas, es decir, símbolos que representan acciones. Los operadores son:

Operadores	Equivalencia	Descripción
&	AND	*Bitwise "y"*
|	OR	*bitwise* "o inclusivo"
^	XOR	*bitwise* "o exclusivo"
~	NOT	Complemento unario (inversión bit)
<<	SHL	Cambio izquierdo
>>	SHR	Cambio derecho

Operador de conversión de tipo explícita

Las conversiones de tipo explícitas permiten convertir un dato dado en un tipo de variable a otro tipo de variable. Hay varios caminos para hacer ésta operación en C++. La técnica más simple, similar a la usada en lenguaje C, es pre-escribir el tipo de variable al que se quiere convertir entre paréntesis antes del identificador de la variable. Por ejemplo:

```
1   int i;
2   float f=3.14;
3   i=(int)f;
```

Asigna a *i* el valor de *f* convertido de *float* (3.14) a un tipo *int* (3), desechando el resto de información del número. Otro camino para realizar la misma operación, propia de C++, es: pre-escribir el tipo de variable a convertir y la variable convertida, a continuación, entre paréntesis:

```
1   i=int(f);
```

Operador tamaño

Éste operador (*sizeof*) toma un parámetro, el cual puede ser de cualquier tipo o variable, y devuelve su tamaño en bytes:

```
1   a=sizeof (char);
```

El ejemplo asignará el valor 1 a la variable a, ya que *char* es de longitud 1-byte por definición. El valor devuelto por *sizeof* es una constante, y es siempre determinada antes de la ejecución del programa.

Otros operadores

Aparte de los vistos, existen una serie unos cuantos operadores más, referidos a aspectos específicos de la programación orientada a objetos. Dichos operadores se tratarán específicamente en la sección de programación orientada a objetos.

Orden de los operadores

Cuando se escriben expresiones complejas con varios operadores, se pueden tener dudas acerca de qué operando es evaluado antes o después. Por ejemplo:

```
1   a=5+7%2;     //no estamos seguros del resultado
2   a=5+(7%2);   //con resultado a=6
3   a=(5+7)%2;   //con resultado a=0
```

La respuesta a la línea 1 es la misma que la de la línea 2, a=6. En C++, hay establecido un orden de prioridad para cada operador, y no solo los operadores aritméticos (que tienen la misma preferencia que la matemática), sino todos. De mayor a menor prioridad, el orden es el siguiente:

Nivel	Operador	Descripción	Agrupamiento
1	`::`	Alcance	Izquierda a derecha
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	Sufijo	Izquierda a derecha
3	`++ -- ~ ! sizeof new delete`	Unario (prefijo)	Derecha a izquierda
	`* &`	Indirección y referencia (punteros)	
	`+ -`	operadores de signo unarios	
4	`(type)`	Conversión de tipo	Derecha a izquierda
5	`.* ->*`	punteros-a-miembro	Izquierda a derecha
6	`* / %`	multiplicativos	Izquierda a derecha
7	`+ -`	aditivos	Izquierda a derecha
8	`>> <<`	Cambio	Izquierda a derecha
9	`< > <= >=`	Relacional	Izquierda a derecha
10	`== !=`	Igualdad	Izquierda a derecha
11	`&`	Bitwise “y”	Izquierda a derecha
12	`^`	Bitwise “xor”	Izquierda a derecha
13	`|`	Bitwise “o”	Izquierda a derecha
14	`&&`	“y” lógico	Izquierda a derecha
15	`||`	“o” lógico	Izquierda a derecha
16	`?:`	Condicional	Derecha a izquierda
17	`= *= /= += -= <<= >>= &= ^= |=`	Asignación	Derecha a izquierda
18	`,`	Coma	Izquierda a derecha

La columna *Agrupamiento* define el orden precedente de cada operador en caso de encontrarse varios operadores del mismo nivel en una expresión.

Por otro lado, y al igual que las matemáticas, el orden de ejecución puede alterarse según nuestros intereses, o simplemente para asegurar el orden que se quiera, mediante el uso de paréntesis. En caso de no estar seguro del nivel de precedencia, es siempre aconsejable el uso de paréntesis.

Entradas y salidas básicas (Input/Output)

Introducción

Hasta ahora, los programas vistos como ejemplo requerían poca o ninguna interacción con el usuario. Usando la librería básica de entrada/salida, se va a poder interactuar mediante mensajes de pantalla, y dando la opción de introducir datos con el teclado.

C++ usa una abstracción conveniente llamada *streams (corriente)* para permitir entradas o salidas con el usuario a través de pantalla y teclado. Una corriente es un objeto donde un programa puede insertar o extraer caracteres del mismo. Realmente no es necesario saber más acerca de las especificaciones de los medios físicos asociados al *stream*: solo es necesario saber que aceptará o proporcionará caracteres secuencialmente.

La librería estándar de C++ incluye el archivo de cabecera *iostream*, donde la entrada y salida de objetos *streams* estándar es declarada.

Salidas estándar (Output)

Por defecto, la salida estándar de un programa es en la pantalla, y el objeto *stream* definido en C++ es *cout*.

Cout es usado en conjunto con el operador inserción (<<, dos símbolos "menor que"):

```
1   cout<<"secuencia de salida";//muestra en pantalla los caracteres
2   cout<<120;                  //muestra en pantalla 120
3   cout<<x;                    //muestra el contenido de la variable
4                               //x en pantalla
```

El operador << inserta los datos que le siguen en el *stream* de salida *cout*. En la línea 1 del ejemplo anterior los datos introducidos son una secuencia de caracteres, en la línea 2 una constante, y en la línea 3 una variable, finalizando en todas con un punto y coma (;). Notar que, en el caso de la línea 1, una secuencia de caracteres está declarada entre comillas dobles (" ") porque es una secuencia constante de caracteres. En el caso de no declararlas entre comillas, se interpretará como identificador de una variable. Por ejemplo:

```
1   cout<<"UPCT";    //Muestra en pantalla UPCT
2   cout<<UPCT;      //Muestra el contenido de la variable UPCT
```

El operador << puede incluirse varias veces para mejorar estructuras en las salidas. Además, es posible combinarlos con variables:

```
1   cout<<"hola, me llamo: "<<nombre<<"muchas gracias";
```

En el ejemplo se muestra dos cadenas de caracteres en las que se intercala el valor de la variable *nombre*. En caso de asignarle a la variable *nombre* el nombre "Juan", la salida obtenida será "*hola, me llamo Juan muchas gracias*".

Es importante también, para mejorar la apariencia, combinar las cadenas de caracteres con los comandos de salto de línea, tabulador, etc. Por ejemplo:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       cout<<"Buenos dias:";
6       cout<<"mi nombre es Juan";
7       return 0;
8   }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Buenos dias:mi nombre es Juan
```

En la ventana del programa ejecutado, aparece junto "*buenos días: mi nombre es Juan"*, por lo que la apariencia del código es independiente de la apariencia del programa ejecutado. Para saltar de línea será necesario introducir el código "\n":

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       cout<<"Buenos dias:\n";
6       cout<<"mi nombre es Juan";
7       return 0;
8   }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Buenos dias:
mi nombre es Juan
```

Además de \n, también podremos añadir el manipulador *endl* al final de cada línea para saltar de línea, obteniendo el mismo resultado:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       cout<<"Buenos dias:"<<endl<<"mi nombre es Juan";
6       return 0;
7   }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Buenos dias:
mi nombre es Juan
```

EL manipulador *endl* produce exactamente el mismo efecto que \n, pero tiene un comportamiento adicional cuando se utiliza con *streams* "memorizadas": la memoria se vacía con *endl*. De todos modos, *cout* será utilizado sin memoria intermedia en la mayoría de los casos, por lo que se podrá utilizar tanto \n como *endl* sin problemas.

Entrada estándar (Input)

El medio para aportar datos de entrada es generalmente el teclado. En C++, la entrada estándar se realiza a través del *stream cin* seguido del operador de extracción (>>, doble mayor que). El operador puede estar seguido de la variable a la que vamos a dar un valor a partir de los datos recibidos por el *stream*. Por ejemplo:

```
1    int edad;
2    cin>>edad;
```

La primera declaración establece la variable tipo *int, edad.* Tras esto, el programa espera hasta que se le asigne un valor entero a la variable.

Cin sólo puede procesar los datos introducidos una vez se haya pulsado la tecla RETURN. Siempre hay que tener presente el tipo de variable que hay que introducir. Si se solicita un entero, se obtendrá un entero, o si se solicita una cadena de caracteres, se tendrá una cadena de caracteres. A continuación se muestra un ejemplo de su uso:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int i;
6       cout<<"Por favor, introduzca un numero entero: ";
7       cin>>i;
8       cout<<"El valor introducido es "<<i;
9       cout<<" y su cuadradro es: "<<i*i<<".\n";
10      return 0;
11  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Por favor, introduzca un numero entero: 25
El valor introducido es 25 y su cuadradro es: 625.
--------------------------------
```

Al igual que en la forma de declarar variables o *cout*, *cin* admite formas simplificadas de la misma forma que se hacía con *cout*:

```
1    cin>>a>>b>>c
```

En el ejemplo, el primer valor escrito en el teclado será asignado a *a,* el segundo a *b*, y el tercero a *c*.

Cin y Strings:

Se puede usar *cin* para tomar *strings* con el operador de extracción (>>) como se hacían con los tipos de variables fundamentales. Sin embargo, la extracción de *cin* se detiene en el momento que lee un caracter "espacio en blanco". En ese caso, tendríamos que proporcionar una palabra para cada extracción, algo incómodo cuando se trata de introducir textos.

Para conseguir introducir frases, se usa la función *getline*, mucho más recomendable que el uso de *cin*. En el ejemplo que sigue, se muestra su uso y sintaxis:

```
1   #include<iostream>
2   #include<string>
3   using namespace std;
4   int main()
5   {
6       string mistr;
7       cout<<"¿Cual es tu nombre?" ;
8       getline (cin, mistr);
9       cout<<"Hola "<<mistr<<".\n";
10      cout<<"¿Que musica te gusta? ";
11      getline (cin,mistr);
12      cout<<"A mi tambien me gusta "<<mistr;
13      return 0;
14  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
¿Cual es tu nombre?Juan Antonio
Hola Juan Antonio.
¿Que musica te gusta? El rock
A mi tambien me gusta El rock
--------------------------------
```

Es interesante ver en el ejemplo que, tanto en la línea 8 y 11, la función *getline* llama a la variable *mistr*, asignándole primero el valor *Juan Antonio* y luego *El rock*, ya que el valor *Juan Antonio* no se necesita a posteriori.

Stringstream

El archivo de cabecera estándar <*sstream*> define la clase llamada *strinstream*, la cual parte de un objeto basado en una cadena (*string-based*) para ser tratado como un *stream*. Éste camino nos permite la extracción e inserción de o para cadenas, lo cual es especialmente práctico para convertir cadenas en valores numéricos y viceversa. Por ejemplo, si queremos extraer un entero de una cadena de puede escribir:

```
1   string mistr ("1234");
2   int mistr;
3   stringtream(mistr)>>mistr;
```

El ejemplo declara el objeto *string* con el valor "1234", y un objeto tipo *int*. Después, se usa la construcción *stringtream* para reconstruir el objeto de cadena de caracteres a entero. Tras la línea 3, el valor de la variable *mistr* será un entero de valor 1234.

Como ejemplo, se plantea:

```
1  #include<iostream>
2  #include<string>
3  #include<sstream>
4  using namespace std;
5  int main(){
6      string mistr;
7      float Precio=0;
8      int Cantidad=0;
9      cout<<"Introduce el precio: ";
10     getline (cin,mistr);
11     stringstream(mistr)>>Precio;
12     cout<<"introduce la cantidad: ";
13     getline (cin, mistr);
14     stringstream(mistr)>>Cantidad;
15     cout<<"precio total: "<<Precio*Cantidad<<endl;
16     return 0;
17 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Introduce el precio: 24.95
introduce la cantidad: 3
precio total: 74.85

--------------------------------
```

Como se ve en el ejemplo, se adquieren valores numéricos de la entrada estándar de forma indirecta. Usando éste método, en vez de la extracción directa (*cin*), se consigue mayor control acerca del valor de entrada, ya que separamos el valor introducido por el usuario, para luego interpretarlo. Éste método es generalmente más aceptado que la introducción directa de datos.

Estructuras de control

Introducción

Un programa es normalmente ilimitado a la hora de escribir secuencias e instrucciones. Durante ese proceso se puede bifurcar, repetir códigos o tomar decisiones. Para éste propósito, C++ proporciona estructuras de control que pueden servir para especificar qué debe hacer un programa, cuando está bajo ciertas circunstancias.

Como introducción de estructuras de control se va a introducir un nuevo concepto: el *bloque* o *declaración compuesta*. Un bloque es un grupo de declaraciones las cuales están separadas por punto y coma (;), como otra declaración cualquiera en C++, pero agrupadas juntas en un bloque cerrado entre llaves ({ }):

{declaración1; declaración2; declaración3;}

La mayoría de las estructuras de control que se verán a continuación requieren una declaración genérica como parte de su sintaxis. La declaración podrá ser simple (una instrucción básica terminada en punto y coma) o compuesta (varias instrucciones agrupadas en bloque).

Estructura condicional: *if* y *else*

La palabra clave *if* es usada para ejecutar una declaración siempre que se cumpla una condición. La forma es:

if (condición) declaración;

Donde la condición es la expresión que ha de evaluarse. Si la condición es verdadera, la declaración se ejecuta. Si por el contrario es falsa, la declaración es ignorada y el programa continúa justo tras la estructura condicional. Por ejemplo, el siguiente código muestra en pantalla "x es igual a 100" si y sólo si se cumple que x==100:

```
1    if (x==100)
2        cout<<"x es igual a 100";
```

En éste caso, como se tiene sólo una declaración tras la estructura *if*, no es necesario acotarlo con llaves. Para el siguiente ejemplo si serán necesarias:

```
1  if (x==100){
2          cout<<"x es igual a: ";
3          cout<<x;
4  }
```

También se podrá especificar que queremos que pase en caso de que la condición no se cumpla usando la palabra clave *else*. Dicha clave se ha de usar en conjunto con *if*:

if (condición) {declaraciones;} else {declaraciones;}

Por ejemplo:

```
1    if (x==100)
2        cout<<"x es igual a 100";
3        else
4        cout<<"x no es igual a 100";
```

Muestra "x es igual a 100" si y sólo si x==100, y si no se cumple (en cualquier otro caso en el que x no sea igual a 100) "x no es igual a 100".

La estructura if se puede encadenar con la intención de verificar un rango de valores. El siguiente ejemplo muestra el código para verificar si *x* es positiva, negativa, o cualquier otro valor (cero):

```
1   if (x>0)
2       cout<<"x es positiva";
3   else if (x<0)
4       cout<<"x es negativa";
5   else
6       cout<<"x es 0";
```

Y, al igual que una estructura *if* simple, en caso de introducir más de una declaración en cada nivel, se deberán acotar mediante llaves.

Estructuras de iteración (bucles)

Los bucles tienen la función de repetir una operación varias veces, siempre que se cumpla una condición.

El bucle *while*

El bucle *while* (*mientras*) repite una acción *mientras* que una expresión sea verdadera. Su formato es:

while (expresión) declaración;

Por ejemplo, el siguiente código muestra una cuenta atrás desde el número que se quiera:

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       int n;
6       cout<<"Introduce el numero de comienzo de cuenta atras: ";
7       cin>>n;
8
9       while (n>0){
10          cout<<n<<", ";
11          --n;
12      }
13      cout<<"FUEGO \n";
14      return 0;
15  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Introduce el numero de comienzo de cuenta atras: 10
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUEGO
```

Cuando el programa recibe del usuario el número de comienzo de cuenta atrás, inicializa el bucle *while*. Mientras que la variable *n* sea mayor que cero, el programa relee las declaraciones acotadas por las llaves. Los valores de n se van mostrando en pantalla (línea 10) pero, debido a la declaración de la línea 11, la variable *n* va disminuyendo su valor de uno en uno por cada repetición del bucle, de modo que cuando llega a cero, la condición no se cumple, por lo que el programa sale del bucle y continúa en la línea 12 hacia abajo.

Cuando se diseña un bucle, se tiene siempre que considerar que debe terminar en algún punto, por lo que debemos establecer un mecanismo que vuelva la condición falsa. En el caso anterior se consigue restando valores a *n* de uno en uno, hasta que *n* llega a cero.

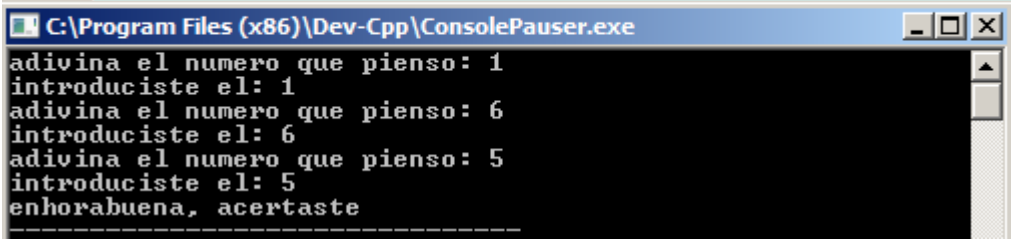
El bucle do-while

Su funcionamiento es exactamente el mismo que el bucle *while*, excepto que la condición en *do-while* es evaluada después de la ejecución de las declaraciones Su formato es de la forma:

do declaración while (condition);

De ésta manera, garantizamos que al menos una vez se realiza la declaración aunque la condición nunca se cumpla. Para ver mejor cómo funciona, se describe ilustra con el siguiente ejemplo:

```
1   #include<iostream>
2   using namespace std;
3   int main(){
4   unsigned long n;
5   do{
6       cout<<"adivina el numero que pienso: ";
7       cin>>n;
8       cout<<"introduciste el: "<<n<<"\n";
9   }while (n!=5);
10  cout<<"enhorabuena, acertaste";
11  return 0;
12  }
```



La estructura *do-while* repite las declaraciones entre llaves (líneas 6 a 8) mientras que se cumpla la condición de *n* distinto de 5. Lo más importante, y lo que le diferencia de la estructura *while* es que la primera iteración se hace antes de comprobar que n sea distinto de 5.

El bucle *for*

Su función se podría definir igual que la de la estructura *while*: repite unas declaraciones dadas mientras que se cumpla una condición. Su formato es de la forma:

for (inicialización; condición; incremento){ declaración(es);}

La diferencia reside en que el bucle *for* es que su propia estructura contiene las condiciones de inicialización e incremento. El bucle *for* está específicamente diseñado para realizar acciones repetitivas con un contador propio. Su funcionamiento es de la forma:

1. Se ejecuta la *inicialización*. Generalmente es un valor inicial asociado a una variable. Éste paso es ejecutado sólo una vez.
2. Se chequea la *condición*. Si es verdadera (se cumple), entonces el bucle continua ejecutándose. Si la condición es falsa (no se cumple), el bucle finaliza en ese mismo momento, sin llegar a ejecutarlo.
3. Se ejecutan las *declaraciones*. Es habitual encontrarse varias declaraciones acotadas por llaves.
4. Finalmente, se ejecuta el *incremento*, volviendo al paso 2.

A continuación se muestra el ejemplo de cuenta atrás mediante un bucle *for*:

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4      for (int n=10; n>0; n--){
5          cout<<n<<", ";
6      }
7      cout<<"FUEGO \n";
8      return 0;
9  }
```

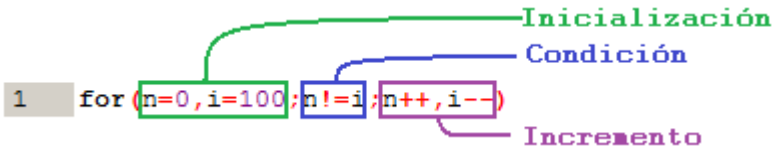
```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUEGO

--------------------------------
```

Los campos *inicialización* e *incremento* son opcionales. Se pueden mantener vacíos, pero en todo caso se ha de poner punto y coma para distinguirlos de la *condición*. Por ejemplo se puede escribir: *for(;n<10;)*, si no se quiere especificar inicialización e incremento, o *for(;n<10;n++)*. Se recurre a éste caso cuando, por ejemplo, a *n* se le ha asignado un valor antes de llegar al bucle *for*.

Opcionalmente, usando el operador coma (,) se puede especificar más de una expresión a la vez en cada uno de los campos del bucle *for*. El operador coma (,) es un separador, que sirve al compilador para distinguir una de otra. Por ejemplo, si se quiere inicializar más de una variable en el bucle:

```
1   for(n=0,i=100;n!=i;n++,i--)
2   {
3       //declaraciones...
4   }
```

El bucle será ejecutado 50 veces, hasta que las variables *n* e *i* coincidan:



n comienza en 0 e *i* en 100. En ese momento, (0 es distinto de 100), la condición se cumple, por lo que se llevan a cabo las declaraciones y se inicializan los incrementos, *n*++ hace que n valga ahora 1 e *i*—hace que valga 99. Éste proceso se repite hasta que *n=50* e *i=50*, momento en el que la condición es falsa, se deja de ejecutar las declaraciones, y el bucle *for* finaliza.

Declaraciones de salto

La declaración *break*

Usando *break* se puede abandonar un bucle cuando se cumpla una condición establecida. Se puede usar como final de un bucle infinito, o forzar su final antes de su final natural. Por ejemplo, se va a usar *break* para interrumpir un bucle *for*, quizás, porque se ha detectado un fallo durante el lanzamiento:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int n;
6       for(n=10; n>0;n--)
7       {
8           cout<<n<<", ";
9           if(n==3){
10              cout<<"cuenta atras abortada";
11              break;
12          }
13      }
14      return 0;
15  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10, 9, 8, 7, 6, 5, 4, 3, cuenta atras abortada
--------------------------------
```

La declaración *continue*

La declaración *continue* hace que, desde ese momento, se continúe ejecutando el bucle omitiendo el resto del boque: se continúa ejecutando el bucle mientras que la condición de *continue* se cumpla, omitiendo las declaraciones que siguen. Por ejemplo, en la siguiente sintaxis se va a omitir el número 5:

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int n;
6      for(n=10; n>0;n--)
7      {
8          if(n==5) continue;
9              cout<<n<<", ";
10     }
11     cout<<"FUEGO\n";
12     return 0;
13 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10, 9, 8, 7, 6, 4, 3, 2, 1, FUEGO

--------------------------------
```

En el ejemplo, en el mismo momento en el que *n* es igual a 5, la declaración *continue* indica que se ignore el resto del bloque, volviendo a la siguiente iteración del bucle *for*. Es por esto que, en la ejecución, no aparece en número 5 en la cuenta atrás.

La declaración *goto*

La declaración *Go to* (*"ir a"*) llama a hacer un salto a otro punto del programa. Ésta declaración ha de ser usada con cautela ya que el salto se realiza sin ningún tipo de restricciones. El punto de destino, es decir, hasta donde se produce el salto, se identifica con una etiqueta, es decir, un identificador válido seguido de punto y coma.

En general, ésta declaración no tiene un uso concreto dentro de la programación orientada al objeto, al menos a nivel de programación básica. Como ejemplo ilustrativo, se va proceder a usar *goto* en la sintaxis del ejemplo de cuenta atrás:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int n=10;
6       bucle:
7           cout<<n<<", ";
8           n--;
9           if(n>0) goto bucle;
10          cout<<"FUEGO\n";
11          return 0;
12  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUEGO

--------------------------------
```

En el ejemplo, la función *goto*, es usada para formar un bucle. En el momento en que *n* es igual a 0, no se ejecuta la declaración *goto* y se pasa a mostrar en la pantalla "*FUEGO*".

La función *exit*

Exit es una función definida en la librería *cstdlib*. El propósito de ésta es la de terminar el programa actual con un código específico de salida. La estructura es de tipo:

void exit (int codigodesalida);

El *codigodesalida* es usado por algunos sistemas operativos y puede ser usado por programas. Por convención, una salida de 0 significa que el programa finalizó normalmente y otro valor indica que ha ocurrido algún error o resultado no esperado.

La estructura selectiva *switch*

La sintaxis de la declaración *switch* es un poco peculiar. Su objetivo es chequear varios valores constantes posibles en una expresión. Algo similar se realiza con la estructura *if/else* vista anteriormente. La estructura de *switch* es de la forma:

```
switch (expresión)
{
case constante1:
    grupo de declaraciones1;
    break;
case constante2:
    grupo de declaraciones2;
    break;
.
.
.
defalut:
    grupo de declaraciones por defecto;
}
```

Switch trabaja de la siguiente manera: primero evalúa una *expresión*, y chequea si es equivalente a alguna de las constantes (*constante1, constante2, etc*), en orden de aparición (si una no se cumple, se chequea la siguiente de forma sucesiva). En el caso de coincidir con alguna de las constantes, se ejecuta el grupo de declaraciones adjuntas a dicha constante. El grupo de declaraciones finaliza con la declaración *break*, con la que la ejecución del programa salte hasta el final de la estructura selectiva *switch*, ya que cuando se cumple una constante, no es necesario seguir evaluando el resto.

Finalmente, si el valor de la *expresión* no ha coincidido con ninguna de las constantes especificadas, el programa ejecutará las declaraciones incluidas en *default* (por defecto, para el resto de los casos). Ésta se puede omitir, de manera que la ejecución pase por la estructura *switch* sin ningún efecto.

Como se indicó arriba, la estructura *switch* e *if/else* son muy similares. A continuación se va a comparar las sintaxis de ambas estructuras:

switch	if/else

switch (x){
  case 1:
  cout<<"x=1";
  break;
  case 2:
  cout<<"x=2";
  break;
  default:
  cout<<"valor de x desconocido";
}
	if(x==1){
	  cout<<"x=1";
	  }
	else if (x==2){
	  cout<<"x=2";
	  }
	else{
	  cout<<"valor de x desconocido";
	  }

La declaración *switch* es un poco peculiar en C++ porque usa etiquetas en vez de boques. Esto fuerza al uso de la declaración *break* después del grupo de declaraciones. Por otra parte, las restantes declaraciones, incluyendo otras etiquetas, podrán ser ejecutadas hasta el final del bloque selectivo *switch* o hasta la aparición de *break*.

Por ejemplo, si no se incluye la declaración *break* después del primer grupo de declaraciones, el programa no saltará automáticamente hasta el final del bloque selectivo *switch* y continuará ejecutando y/o chequeando el resto de declaraciones y casos. Esto hace innecesario el uso de llaves ({ }) acotando las declaraciones, lo que puede ser práctico para ejecutar el mismo bloque de instrucciones para diferentes valores posibles. Por ejemplo:

```
switch (x){
    case 1:
    case 2:
    case 3:
    cout<<"x es 1, 2  o 3";
    break;
    default:
        cout<<"x no es 1, 2 o 3";
}
```

Es importante darse cuenta que, el bloque selectivo *switch,* sólo puede comparar una expresión con constantes. No es posible poner variables como casos (por ejemplo *case n:* donde *n* es una variable) o rangos (case (1, 2, ...) porque no son constantes válidas en C++.

Si se necesita chequear rangos o valores que no son constantes, se debe utilizar la concatenación *if/else*.

Funciones

Introducción

El uso de funciones puede orientar los programas realizados en un camino más modular, es decir, realizar una programación a partir de funciones, con el objetivo de estructurar mejor el programa, haciéndolo más versátil.

Una función es un grupo de declaraciones que son ejecutadas cuando son llamadas en la ejecución de un programa. El formato habitual es:

```
type nombre(parametro1, parámetro2,..){declaraciones}
```

Donde:

- *Type:* es el tipo de dato que va a devolver la función (*int, void, float, etc*)
-*Nombre:* es el identificador por el que será posible llamar a la función.
-*Parámetros* (tantos como sean necesarios): Cada parámetro consiste en el tipo de dato específico seguido de su identificador (por ejemplo, *int x*), siendo éstas variables locales de la función.
-Las *Declaraciones* son el cuerpo de la función. Se trata de un bloque de declaraciones acotadas por llaves({ })

A continuación se muestra un ejemplo, con el que se muestran sus características:

```
1   #include<iostream>
2   using namespace std;
3
4   int adicion (int a, int b)
5   {
6       int r;
7       r=a+b;
8       return (r);
9   }
10  int main()
11  {
12      int z;
13      z=adicion (5,2);
14      cout<<"el resultado es: "<<z;
15      return 0;
16  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
el resultado es: 7
```

Antes de empezar a analizar, es preciso recordar que un programa C++ siempre empieza su ejecución por la función *main (principal)*.

En la línea 10 y 12 se observa que la función *main* comienza declarando una variable local: un entero *int z*. Tras esto, asigna al entero *z* un valor obtenido de resolver la función *adición* para los valores de x=5 e y=2. De forma que:

```
int adicion (int a,int b)
  z=adicion (  5   ,   2  );
```

Los parámetros y argumentos tienen una correspondencia clara: cada valor se asigna en el mismo orden que son declarados en la función *adicion*.

En el momento en que *main* llama a la función *adicion*, la ejecución pasa ésta última función asignando los valores establecidos. Una vez inicializada, se declara la variable local *int r* (líneas 4 y 6). A continuación, a la variable r se le asigna el resultado de la suma de a+b (línea 7). Por último, en la línea 8 se declara *return(r);* finalizando la función *adicion*. A partir de aquí, el control de la ejecución vuelve a *main*, siguiendo la ejecución en el mismo punto donde se pasó a *adicion*. El detalle reside en que ahora, como la función *adicion* devolvió el valor de r, se le asigna dicho valora la variable z de la función *main*:

```
int adicion (int a,int b)
  7
  z=adicion (  5   ,   2  );
```

Por lo que a z se le asigna el valor 7. Por último, la función muestra el resultado en pantalla (línea 14) y el programa principal finaliza (*return 0;*).

A continuación se analizará otro ejemplo, para una mayor comprensión:

```
1  #include <iostream>
2  using namespace std;
3  int diferencia (int a, int b)
4  {
5  int r;
6  r=a-b;
7  return (r);
8  }
9  int main ()
10 {
11 int x=5, y=3, z;
12 z = diferencia (7,2);
13 cout << "El primer resultado es " << z << '\n';
14 cout << "El segundo resultado es " << diferencia (7,2) << '\n';
15 cout << "El tercer resultado es " << diferencia (x,y) << '\n';
16 z= 4 + diferencia (x,y);
17 cout << "El cuarto resultado es " << z << '\n';
18 return 0;
19 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
El primer resultado es 5
El segundo resultado es 5
El tercer resultado es 2
El cuarto resultado es 6
```

En la sintaxis anterior se encuentran 4 formas distintas de llamar a una función: líneas 12, 14, 15 y 16. Por lo tanto se llegan a las siguientes conclusiones:

-Una función independiente de la principal (*main*) puede ser llamada tantas veces como se requiera.
-Como se muestra en la línea 15, los valores en el momento de llamar a la función *diferencia* pueden ser valores de variables locales.
-Como se comprueba en las líneas 14 y 16, la llamada de la función es tratada de forma equivalente a cualquier otra variable.

Funciones sin tipo: el uso de void

Partiendo de la sintaxis de la declaración de una función:

```
1  type nombre(parametro1, parámetro2,..){declaraciones}
```

Se vio que la definición comienza especificando el tipo (*type*) que es la función (o dicho de otro modo, el tipo de constante que devolverá cuando sea llamada). Pero, ¿Qué pasa si esperamos que no devuelva ningún valor?

Imaginad que se espera hacer una función que muestre un mensaje en la pantalla. En principio no se necesita que se devuelva ningún dato. En éste caso podremos usar el tipo *void* para la función. Éste es un indicador específico que indica la ausencia de tipo.

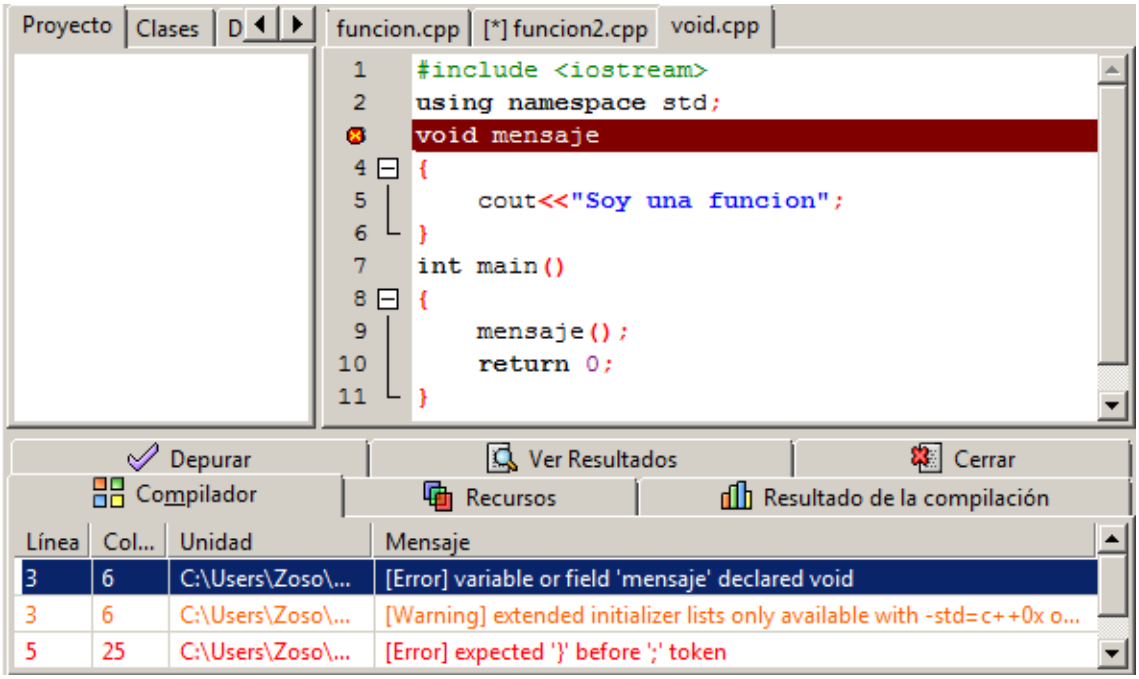
```
1   #include <iostream>
2   using namespace std;
3   void mensaje()
4   {
5       cout<<"Soy una funcion";
6   }
7   int main()
8   {
9       mensaje();
10      return 0;
11  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Soy una funcion
--------------------------------
```

Void puede también ser usado en la lista de parámetros de la función para la especificación explícita de que quiere que la función no toma valores cuando es llamada (aunque es opcional). Por ejemplo, la función *mensaje* se podría haber declarado como:

```
3   void mensaje(void)
4   {
5       cout<<"Soy una funcion";
6   }
```

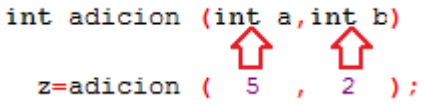
En C++, una lista de parámetros puede ser dejada en blanco cuando se quiere una función sin parámetros. Por otro lado, es importante no olvidar que, aunque no se requieran parámetros en la función, la lista de parámetros debe ser declarada en blanco, es decir, siempre habrá que poner los paréntesis en la definición de una función:



En el ejemplo, el programa indica un fallo de sintaxis al no reconocer el significado de *mensaje*.

Argumentos con origen en valores y por referencia

Hasta éste momento, los valores de los argumentos de las funciones han sido declarados *por valor*, es decir, cuando se llama a una función con unos parámetros dados, ésta se inicializa con los valores de las variables, pero no con las variables en sí. Como se vio en la función *adicion*, cuando es llamada se produce el proceso de forma que:



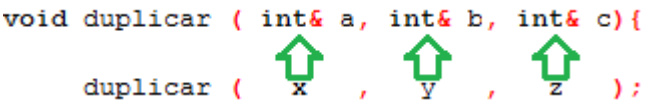
La función no es llamada con las variables “x” e “y”, sino con sus valores 5 y 2.

Sin embargo, existen casos en los que se necesitará manipular desde dentro de una función el valor de una variable externa. Para éste propósito, se puede usar argumentos pasados por referencia, como en la función *duplicar* del ejemplo siguiente:

```
1   #include <iostream>
2   using namespace std;
3
4   void duplicar (int& a,int& b, int& c){
5       a*=2;
6       b*=2;
7       c*=2;
8   }
9   int main()
10  {
11      int x=1, y=3, z=7;
12      duplicar (x,y,z);
13      cout<<"x="<<x<<" ,y="<<y<<" z="<<z;
14      return 0;
15  }
```

La primera cosa que llama la atención, es la forma en la que se declaran los argumentos en la función *duplicar* (línea 4): el tipo de variable de cada parámetro está seguido del símbolo *&* (*int& a*). Con esto se especifica que sus correspondientes argumentos van a ser pasados *por referencia* en vez de por valor.

Cuando una variable es pasada *por referencia* no se está pasando una copia de su valor, sino que se está asociando los valores de la llamada (*x, y, z*) con los valores de la función (*a, b*, *c*):



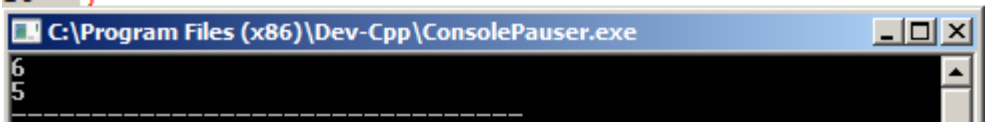
Pasar los datos *por referencia* es también una vía efectiva para permitir que una función devuelva más de un valor. Por ejemplo, a continuación se muestra una función que devuelve el número anterior y posterior al número enviado:

```
1   #include<iostream>
2   using namespace std;
3   void prevsigu (int x, int& prev, int& sigu)
4   {
5       prev=x-1;
6       sigu=x+1;
7   }
8   int main()
9   {
10      int x=100, y, z;
11      prevsigu (x, y, z);
12      cout<<"Num. previo: "<<y<<" , num. siguiente: "<<z;
13      return 0;
14  }
```

Valores por defecto en parámetros

Cuando declaramos una función se puede especificar un valor por defecto para cualquiera de los parámetros. Dicho valor podrá ser usado como argumento si el parámetro está en blanco cuando es llamada la función. Para definir el valor por defecto, se usa el operador de asignación y un valor para el argumento durante la declaración de la función. Si el valor de la función no es especificado cuando es llamada, se tomará el valor por defecto, o de lo contrario, se ignorará. Por ejemplo:

```
1   #include<iostream>
2   using namespace std;
3
4   int dividir (int a, int b=2)
5   {
6       int r;
7       r=a/b;
8       return r;
9   }
10  int main()
11  {
12      cout<<dividir(12);
13      cout<<endl;
14      cout<<dividir (20,4);
15      return 0;
16  }
```

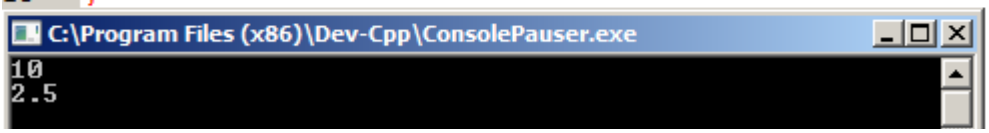


Se puede ver en la línea 12 que se llama a la función especificando sólo un argumento (12). En la función *dividir,* 12 es asignado a *a,* mientras que *b* tiene el valor asociado por defecto de 2, por lo que se muestra el resultado de dividir 12 entre 2 (6). En la línea 14, sin embargo, asignamos argumentos a los dos parámetros, tanto *a(20)* como *b(4)*, por lo que obtenemos el resultado de la división de ambos (5).

Funciones sobrecargadas

En C++ dos funciones diferentes pueden tener el mismo nombre si el tipo de variable con la que es definida es distinta. Por ejemplo:

```
1  #include<iostream>
2  using namespace std;
3  int calcular (int a, int b)
4  {
5      return (a*b);
6  }
7  float calcular (float a, float b)
8  {
9      return(a/b);
10 }
11 int main()
12 {
13     int x=5, y=2;
14     float n=5.0, m=2.0;
15     cout <<calcular (x,y)<<"\n";
16     cout<<calcular (n,m)<<"\n";
17     return 0;
18 }
```



Se han definido dos funciones con el mismo nombre, *calcular* (líneas 3 y 7), pero la primera acepta sólo parámetros enteros (*int*), mientras que la segunda del tipo *float*. EL compilador reconoce a cual llamar en cada caso ya que examina los tipos de variables de cada una cuando es llamada. Se comprueba que esto pasa cuando analizamos los resultados. La primera función *calcular* devuelve el producto de los argumentos (línea 5), mientras que la función *calcular* definida con *float* devuelve la división de los argumentos (línea 9).

Es importante también aclarar que, dos funciones del mismo tipo de respuesta (por ejemplo *int función*), con el mismo nombre, basta que uno de los argumentos sea distinto para que el compilador las distinga.

Funciones en línea (*inline*)

El especificador *inline* indica al compilador que se requiere una sustitución *en línea* al mecanismo usual de llamada de una función. Esto no produce cambios en la función llamada, sino que indica al compilador que introduzca el código de la función llamada en el punto donde se especifica. Ésta operación se realiza cuando la función es llamada de forma recursiva, ahorrando tiempo y potencia. El formato de la declaración es:

```
inline tipo nombre (argumentos) {cuerpo...}
```

La llamada a la función, al igual que todas las propiedades de las funciones se conservan de igual modo.

Algunos compiladores también son capaces de optimizar el código para generar funciones *inline* cuando es conveniente. Éste especificador indica la compilador que *inline* es preferido para esta función.

Recursividad

La recursividad es una propiedad en la que las funciones son llamadas por ellas mismas. Esta propiedad es útil para muchas tareas, como por ejemplo clasificar datos o calcular el factorial de un número. Para éste último caso, la fórmula matemática de cálculo de factorial sería:

n!=n*(n-1)*(n-2)*….*1

Una función recursiva que calcule ésta operación podría ser:

```
1   #include<iostream>
2   using namespace std;
3   long factorial (long a)
4   {
5       if (a>1)
6       return (a*factorial (a-1));
7       else return (1);
8   }
9   int main(){
10      long numero;
11      cout<<"indique un numero: ";
12      cin>>numero;
13      cout<<numero<<"!= "<<factorial (numero);
14      return(0);
15  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
indique un numero: 5
5!= 120
```

En la línea 6 se observa cómo se llama de forma recursiva a la función *factorial* dentro de si misma hasta que el valor de *a* no sea igual que 1 (si no se indicara esto último, entraría en un bucle infinito, multiplicado por 0 y los sucesivos valores negativos)

Ésta función tiene una limitación, y es que se ha usado una variable tipo *long* por simplicidad. Los resultados dados no pueden ser mayores que 10! o 15! dependiendo del sistema operativo (ver tabla resumen de variables en el apartado de tipos de variables).

Declaración de funciones

Hasta ahora, se han definido todas las funciones antes de su primera llamada en la sintaxis del código. Dichas llamadas se han realizado desde una función principal (*main*) que se ha dejado al final del código. Si se cambia éste orden, poniendo el código de la función principal antes de

las funciones a las que llama, es muy posible que aparezcan errores en la compilación. La razón es que para poder llamar a una función debe haber sido declarada en algún punto anterior en el código, como se ha hecho hasta ahora.

Pero existe un camino alternativo para evitar escribir el código de todas las funciones antes de ser usadas por la función principal u otra función. Se puede lograr declarando un *prototipo* de la función, en vez de su totalidad. Dicha declaración será más corta que la definición original, pero con la suficiente información para que el compilador determine su tipo de variable de retorno y argumentos. Es de la forma:

```
tipo nombre (tipo de argumento1, tipo de argumento2,...);
```

Como se observa, es idéntico a la definición de una función, sólo que no se incluye el cuerpo de la función (la parte entre llaves, { }). A cambio, es necesario poner un puto y coma al final de la función prototipo.

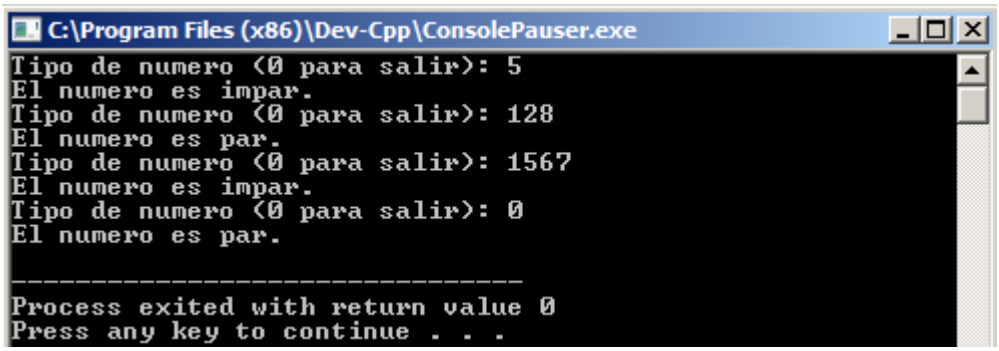
En la enumeración de parámetros no es necesario incluir los identificadores, solo el tipo de variable que se trata. La inclusión de los nombres de cada parámetro, como en la definición de la función, es opcional en la función prototipo. Por ejemplo, se podrán declarar una función prototipo de las siguientes maneras:

```
int prototipo (int identificador1, int identificador2);
int prototipo (int, int);
```

Aun así, la inclusión de los identificadores de las variables es aconsejable, para que la sintaxis sea más legible.

```
1  #include <iostream>
2  using namespace std;
3
4  void impar(int a);
5  void par (int a);
6
7  int main()
8  {
9      int i;
10     do {
11         cout <<"Tipo de numero (0 para salir): ";
12         cin>>i;
13         par (i);
14     } while (i!=0);
15     return 0;
16 }
17
18 void impar (int a){
19 if ((a%2)!=0) cout<<"El numero es impar. \n";
20 }
21
22 void par (int a){
23     if ((a%2)==0) cout<<"El numero es par. \n";
24     else impar(a);
25 }
```

El programa descrito, muestra si un número dado es par o impar. Como se observa, la líneas 4 y 5 son funciones prototipo, las cuales están definidas en las líneas 18 y 22. El resultado de ejecutar el código es satisfactorio:



Es importante resaltar el orden en el que se han escrito las funciones, analizando la importancia de las funciones prototipo en ésta sintaxis. Para el programa principal, *main()* (línea 7), el orden no es importante, ya que bien se podría haber puesto al final sin problemas. Sin embargo, puede ser lógico que la parte principal de un programa esté al principio de la sintaxis y no al final. No pasa así con las funciones *par* e *impar*, ya que *par* da paso a la función *impar* cuando se comprueba que un número no es par. De la forma que está escrito, la función *par* necesita recurrir a la función prototipo *impar* para ejecutarlo, ya que está declarada antes que la función *par*. De no declarar las funciones prototipo, la ejecución llevaría a errores en la compilación.

Otra ventaja que hacen usuales las funciones prototipo es que, a la hora de programar, se pueden añadir muchas funciones sin importar su orden, ya que bastaría con declarar sus correspondientes funciones prototipo para recurrir a ellas en cualquier momento.

Arrays

Introducción

Un *Array* es una serie de elementos del mismo tipo, localizados de forma contigua en la memoria, los que se puede hacer referencia individualmente añadiendo un índice a un único identificador.

Esto significa que, por ejemplo, se puede almacenar 5 valores de tipo *int* en un *array* sin tener que declarar 5 variables distintas, cada una con su propio identificador. Por tanto, usando un *array* se podrán almacenar 5 valores distintos del mismo tipo, *int* por ejemplo, con un único identificador.

Por ejemplo, un *array* que contenga 5 valores enteros llamado *barco* podría ser representado como:

Donde cada espacio en blanco representa un elemento del *array*. Estos elementos están numerados de 0 a 4, siendo siempre el primer elemento el 0, independientemente de lo largo que sea.

Como una variable regular, un *array* deberá ser declarado antes de ser usado. La típica declaración de una *array* en C++ es:

```
tipo nombre [elementos];
```

Donde el tipo debe ser válido (tipo *int, float, etc…*), el nombre un identificador válido, y el campo de elementos (siempre entre llaves de tipo [ ]), especificando cuantos elementos va a contener el *array*.

Por lo tanto, declarar un array llamado *barco* como el mostrado en el diagrama será de la forma:

```
int barco [5];
```

NOTA: EL campo de elementos con llaves de tipo [ ] que representa el número de elementos de un *array* contendrá, debe ser un valor constante, debido a que los *arrays* son bloques de memoria "no dinámica", cuya memoria queda determinada antes de la ejecución del programa. Más adelante, se verá cómo se puede cambial la longitud de un *array* de forma "dinámica", es decir, de la forma que nos interese.

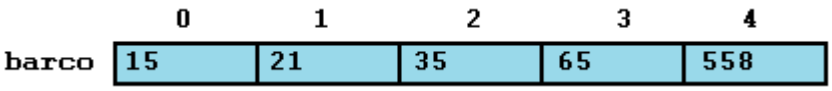
Inicialización de *arrays*

Cuando se declara un *array* local (con alcance sólo dentro de una función), si no se especifica otra cosa, sus elementos se inicializarán con ningún valor por defecto, ya que su contenido está indeterminado hasta que se guarden valores en ellos. Por otro lado, los elementos de *arrays* estáticos o globales, son inicializados automáticamente con sus valores por defecto, cuyo valor es cero en los tipos fundamentales de variables.

En ambos casos, local o global, cuando se declara un *array*, se tiene la posibilidad de asignar valores iniciales a cada uno de los elementos acotando éstos entre llaves de tipo { }. Por ejemplo:

```
int barco [5]={ 15, 21, 35, 65, 558};
```

Ésta declaración crearía un *array* de forma:



Naturalmente, el número de valores declarados entre llaves { } no deberá ser más largo que los valores con los que se define el *array* (entre llaves [ ]). Sin embargo, C++ admite la posibilidad de dejar en blanco el campo con llaves [ ], para luego incluir los valores en el campo de llaves { }, asumiendo que la longitud del *array* según el número de elementos incluidos en el campo de llaves tipo { }:

```
int barco []={ 15, 21, 35, 65, 558};
```

En éste último caso, el *array barco* tendría 5 enteros de longitud.

Acceso a los valores de un *Array*

En algún punto de un programa, donde un *array* es visible, se puede acceder a los valores de cualquiera de sus elementos individualmente si son variables normales, o de lo contrario habría que leer y transformar dicha variable. El formato es tan simple como:

```
nombre [indice]
```

Siguiendo con el ejemplo del *array barco*, cuyos 5 elementos eran de tipo enteros, el nombre con el que hará referencia a los elementos será:

	barco[0]	barco[1]	barco[2]	barco[3]	barco[4]
barco	15	21	35	65	558

Por ejemplo, guardar el valor 75 en el tercer elemento de *barco*, se escribiría la siguiente declaración:

```
barco[2]=75;
```

Teniendo en cuenta que el primer elemento de un *array* es el [0]. Por otro lado, para asignar el valor de un elemento del *array barco* a una variable, por ejemplo *a*, será:

```
a=barco[2];
```

Teniendo en cuenta que, para cualquier situación, la expresión *barco[2]* se comportará como variable entera (*int*).

En C++ es sintácticamente correcto exceder el rango de índices declarado en un *array*. Esto puede crear problemas, ya que aunque no cree problemas en la compilación, sí que puede crearlos durante la ejecución. Las razones se verán más adelante, en la sección sobre el uso de *punteros*.

En éste punto, es importante distinguir entre los dos propósitos que tienen las llaves de tipo [ ] en un *array*. La primera es la de especificar el tamaño de los *arrays* declarados, tal y como se ha visto anteriormente. La segunda es especificar el índice del elemento al que estamos haciendo referencia. Es importante no confundir ambas facetas.

```
int barco[5];
barco[2]=75;
```

Para integrar e ilustrar todas éstas características, se muestran los siguientes ejemplos válidos:

```
b=barco[a+1];
barco[barco[a]]=barco[2]+5;
```

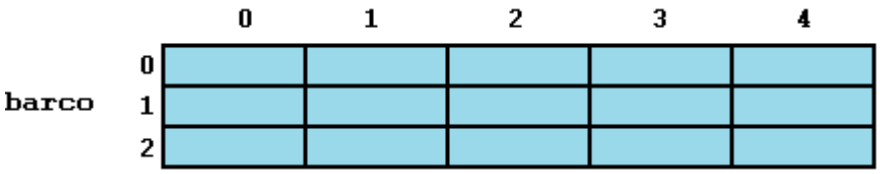
El ejemplo de a continuación ilustra cómo llamar a cada elemento de un array mediante un bucle *for*, para obtener la suma de todos los elementos del array:

```
1   #include <iostream>
2   using namespace std;
3
4   int barco[]={15,21,35,65,558};
5   int n, resultado=0;
6   int main()
7   {
8       for(n=0;n<5;n++){
9           resultado +=barco[n];
10      }
11      cout<<resultado;
12      return 0;
13  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
694
```

Arrays multidimensionales

Se pueden definir los *arrays* multidimensionales como "*arrays* de *arrays*". Por ejemplo, un array bidimensional puede imaginarse como una tabla bidimensional (o matriz) cuyos elementos tienen la misma capacidad de almacenaje:

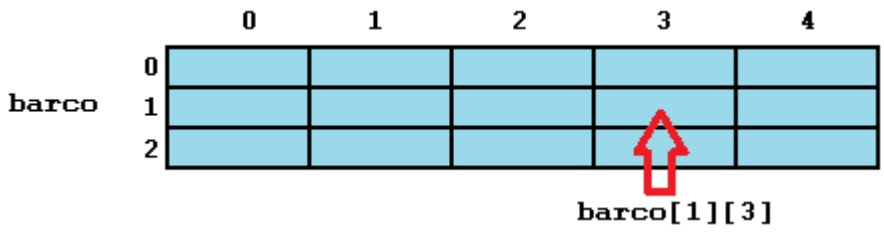


Barco representa un *array* bidimensional de 3 por 5 elementos de tipo *int*. La forma de declarar un *array* como el del diagrama es:

```
int barco [3][5];
```

Por otro lado, si se quiere hacer referencia de un elemento del *array* bidimensional, por ejemplo al elemento de la segunda fila y tercera columna, se declarará:

```
barco[1][3]
```



La capacidad de un *array* es prácticamente infinita: se pueden declarar tantas dimensiones como se quiera (por ejemplo de 6 dimensiones). Pero, cuanto más complicado y grande sea un

array, mayores necesidades de memoria requerirá, aumentando ésta rápidamente por cada dimensión añadida. Por ejemplo, la declaración:

```
char siglo [100][365][24][60][60];
```

Ésta crea un *array* de elementos de tipo caracter de cada segundo de un siglo, es decir, 3153600000 caracteres, con un coste de 3 gigabytes de memoria.

Los *arrays* multidimensionales son una abstracción orientada a la programación, ya que siempre es posible declarar un simple *array* con tantos elementos como se precise. Volviendo al ejemplo *siglo*, su *array* simple, con la misma capacidad, se declara como:

```
char siglo [3153600000];//producto de 100*365*24*60*60
```

La única diferencia es que con los *arrays* multidimensionales, el compilador recuerda la profundidad de la dimensión imaginada por el usuario. En el siguiente ejemplo, se comparan dos partes de un código, ambas con el mismo efecto, pero una usando un *array* multidimensional y la otra un simple *array*:

Array multidimensional

```
#define DESPL 5
#define ESLORA 3
int barco [DESPL][ESLORA];
int n,m;
int main ()
{
for (n=0;n<DESPL;n++)
for (m=0;m<ESLORA;m++)
{
barco[n][m]=(n+1)*(m+1);
}
return 0;
}
```

Array pseudo-multidimensional

```
#define DESPL 5
#define ESLORA 3
int barco [DESPL * ESLORA];
int n,m;
int main ()
{
for (n=0;n<DESPL;n++)
for (m=0;m<ESLORA;m++)
{
barco[n*ESLORA+m]=(n+1)*(m+1);
}
return 0;
}
```

Ninguno de los códigos produce ningún efecto en pantalla al ser ejecutados, pero ambos asignan valores a la memoria de *barco* de la siguiente forma:

		0	1	2	3	4
	0	1	2	3	4	5
barco	1	2	4	6	8	10
	2	3	6	9	12	15

Se ha usado *constantes definidas* (*#define*) para simplificar en lo posible futuras modificaciones del programa. Por ejemplo, en caso de que se decida expandir el *array* a un *DESPL* de 4 en vez de 3, tan solo hay que cambiar la línea inicial sustituyendo 3 por 4, sin tener que modificar nada más.

Arrays como parámetros

En algún momento se va a necesitar pasar un *array* como parámetro en una función. En C++ no es posible pasar un bloque completo de memoria por valor como parámetro a una función, pero si es posible pasar su dirección. En la práctica, éste hecho produce el mismo efecto y, además, el proceso resulta mucho más rápido y eficiente.

Para aceptar *arrays* como parámetros de funciones sólo se tienen que tener en cuenta lo siguiente: habrá que especificar el sus parámetros el tipo de elemento del *array*, con su identificador y llaves vacías del tipo [ ], Por ejemplo:

```
void proceso { int argumento[]}
```

La función anterior acepta un parámetro tipo "*array* de enteros" llamado *argumento*. El ejemplo siguiente ilustra el tratamiento de *arrays* en un programa sencillo:

```
1  #include <iostream>
2  using namespace std;
3  void mostrararray(int arg[], int longitud){
4      for (int n=0; n<longitud; n++)
5      cout <<arg[n]<<" ";
6      cout<<" \n";
7  }
8  int main ()
9  {
10     int primerarray[]={5, 10, 15};
11     int segundoarray[]={ 2, 4, 6, 8, 10};
12     mostrararray (primerarray,3);
13     mostrararray (segundoarray,5);
14     return 0;
15 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
5 10 15
2 4 6 8 10

--------------------------------
```

Como puede verse, el primer parámetro (*int arg[],* línea 3) acepta cualquier *array* cuyos elementos sean de tipo *int*, sea cual sea su longitud. Por ésta razón se ha incluido un segundo parámetro que limita la longitud de cada *array* que se va a pasar a la función, (incluida la primera). Esto permite al bucle *for* que muestre en pantalla el *array* para conocer el rango, para asegurar que el *array* pasado no está fuera de rango.

En una declaración es también posible incluir *arrays* multidimensionales. El formato, por ejemplo, de un *array* tridimensional será:

```
tipo_base[][profundidad][profundidad]
```

Por ejemplo, una función con un *array* multidimensional como argumentos se declararía como:

```
void proceso (int array[][3][4])
```

Es importante darse cuenta de que las primeras llaves [ ] están vacías, y las siguientes no. Esto es así porque el compilador debe determinar en la función cuál es la profundidad de cada dimensión adicional.

Los *arrays*, tanto simples como multidimensionales, pasados como argumentos en funciones deben ser tratados con mucho cuidado, ya que es muy fácil llegar a errores. Es recomendable tener muy claro lo que se va a hacer. En el capítulo de *punteros* se ilustran vías de uso de los *arrays* y sus operaciones, proporcionando una mejor comprensión.

Secuencias de caracteres

Introducción

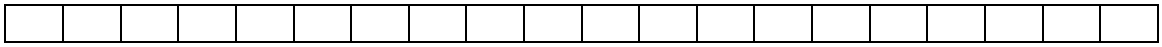
Tal y como se ha visto anteriormente, la librería estándar de C++ implementa un potente tipo de cadena, que es muy apropiada para manipular cadenas de caracteres. No obstante, como las cadenas son de hecho secuencias de caracteres, se pueden representar como un *array* de caracteres.

Por ejemplo, el siguiente *array*:

```
char nombre [20];
```

Es un *array* con capacidad para almacenar 20 elementos de tipo *char*. Éste se puede representar como:

nombre

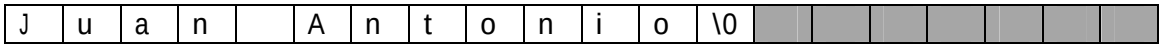


Por lo tanto, en éste *array*, en teoría, se podrá almacenar secuencias de caracteres de hasta 20 caracteres, pero también más cortas. Por ejemplo, *nombre* puede almacenar, en distintos puntos del programa, las palabras “Sergio” o “Juan Antonio”, aunque tengan menos de 20 caracteres.

Sin embargo, y debido a que la secuencia de caracteres es más corta que su longitud total, se usa un caracter adicional para avisar del final de la secuencia: el *caracter nulo (null character)*, cuya constante literal es “*\0*” (contrabarra y cero). Volviendo a la representación anterior, se tendrá:

nombre

nombre



Se observa pues que el valor nulo se incluye como valor almacenado para avisar del final de la secuencia, mientras que las zonas sombreadas representan valores *char* indeterminados.

Inicialización de secuencias de caracteres terminadas en cero

Los *arrays* de caracteres se rigen por las mismas normas que el resto de *arrays*. Por ejemplo, si se quiere inicializar un *array* de caracteres con alguna secuencia de caracteres predeterminada, se puede hacer tal y como se hacía para los casos vistos en el capítulo *Array*:

```
char saludo[]={'H','o','l','a','\0'};
```

En éste caso se podría haber declarado un *array* de de 5 elementos, uno por cada caracter mas el caracter nulo al final. Pero los *arrays* de elementos tipo *char* tienen un método adicional para analizar sus valores: usando cadenas literales.

En las expresiones utilizadas en los apartados anteriores, se han usado constantes que representan cadenas de caracteres enteras. Éstas eran especificadas acotando el texto con comillas dobles, como por ejemplo:

```
"el resultado es: "
```

Las comillas dobles con constantes literales cuyo tipo es, de hecho, el valor nulo de un *array* de caracteres. De modo que las cadenas de caracteres encerradas entre comillas dobles siempre tienen el caracter nulo, \0, automáticamente añadido al final.

Por lo tanto, se puede inicializar el *array* de elementos *char* llamado *saludo* con el caracter nulo de las siguientes dos maneras:

```
char saludo[]="Hola";
char saludo[]={'H','o','l','a','\0'};
```

En ambos casos el *array* de caracteres *saludo* es declarado con un tamaño de 5 elementos de tipo *char*: 4 para los caracteres de la palabra *Hola* más uno del caracter nulo.

Es importante resaltar que se están declarando *arrays* de caracteres en todo momento, y no asignando valores ya que se declara ya con los mismos. De hecho, a la hora de copiar o pasar datos, se tienen las mismas restricciones que para cualquier otro *array*, por lo que no se podrán copiar bloques de datos con una simple operación de asignación.

Uso de secuencias de caracteres terminadas en cero

Las secuencias de caracteres terminadas en cero es la forma normal de introducir cadenas en C++, por lo que se usan en muchos procedimientos. De hecho, las cadenas regulares literales de éste tipo (*char[ ]*) y pueden también ser usadas en la mayoría de los casos.

Por ejemplo, *cin* y *cout* proporcionan secuencias terminadas en cero como contenedores válidos de secuencias de caracteres, por lo que pueden ser usados directamente para extraer cadenas desde *cin* o ser insertadas éstas en un *cout*. Por ejemplo:

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5   char pregunta[] = "Por favor, escriba su nombre: ";
6   char saludo[] = "Hola ";
7   char nombre [80];
8   cout << pregunta;
9   cin >> nombre;
10  cout << saludo << nombre;
11  return 0;
12  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Por favor, escriba su nombre: Juan
Hola Juan
```

Como se puede ver, se han declarado tres *arrays* de elementos tipo *char*. Los 2 primeros son inicializados con cadenas de caracteres literales, mientras que el tercero (línea 7) se deja sin inicializar (pero tendrá un tamaño de hasta 80 caracteres). En cualquier caso, se especifica el tamaño del *array* : en el caso de los dos primeros (líneas 5 y 6), el tamaño está implícitamente definido por la longitud de la cadena de caracteres literales.

Finalmente, las secuencias de caracteres almacenados en *arrays* de tipo *char* pueden ser fácilmente convertidos en objetos de tipo *string* usando el operador de asignación:

```
3   string micadena;
4   char matriz[]="ingenieria naval";
5   micadena=matriz;
```

Punteros (*pointers*)

Introducción

Hasta ahora se ha visto cómo se le asignan celdas de memoria a las variables, y cómo se puede acceder a ellas usando sus identificadores. Éste camino no exige conocer la localización física de la memoria, ya que con el identificador era suficiente.

La memoria de una computadora se puede asemejar a una sucesión de celdas, en las que en cada una tiene la capacidad de almacenaje de 1 byte. Estas celdas están numeradas de forma consecutiva dentro de un mismo bloque. Por tanto, cada celda tiene una dirección concreta que la distingue entre las demás.

Operador de referencia (&)

Tan pronto como se declara una variable, se le asigna la cantidad de memoria necesaria para almacenarla (y la dirección de ésta memoria). Generalmente, no se decide la localización exacta en la memoria del ordenador en la que se quiera guardar, sino que se sigue un proceso automático ejecutado por el sistema operativo de la computadora. Sin embargo, en algunos casos se puede estar interesado en conocer la dirección en la que la variable está almacenada

durante la ejecución del programa, con el objetivo de operar con las posiciones relativas a la misma.

La dirección que localiza a una variable en la memoria se denomina *referencia de la variable*. Ésta referencia puede ser obtenida adjuntando como prefijo del identificador el símbolo *and* (&), conocido como *operador referencia*, y que puede sr literalmente traducido como *la dirección de*. Por ejemplo:

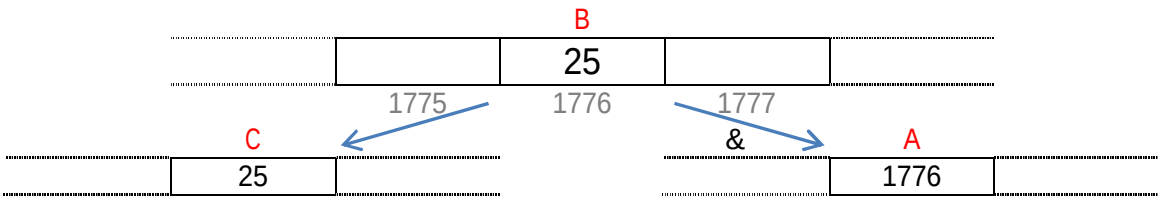
```
A=&B
```

Ésta declaración asignaría a la variable A la dirección de la variable B. Cuando una variable está precedida por el operador &, no se está haciendo referencia a su contenido, sino a su referencia (o dirección en la memoria).

Ahora se supondrá que la referencia de B durante la ejecución es 1776 (éste número es totalmente arbitrario, es sólo como analogía para entender el funcionamiento de la máquina). Considerando el siguiente fragmento de código:

```
B=25;
C=B;
A=&B;
```

Los valores contenidos en cada variable tras la ejecución siguen el siguiente proceso, presentado como un diagrama, y con los identificadores en rojo:



Primero se ha asignado el valor 25 a B (variable la cual hemos atribuido la dirección 1776). La segunda declaración asigna a C el valor de B, con lo que C=25, mientras que en la tercera declaración se ha asignado el valor de la dirección de B, con lo que A para a tener el valor de la dirección de B (en éste caso, se ha supuesto 1776).

La variable que almacena la referencia de otra variable, como el caso de A, es lo que se denomina *puntero*. Los punteros tienen un papel muy potente en el lenguaje C++ en programación avanzada.

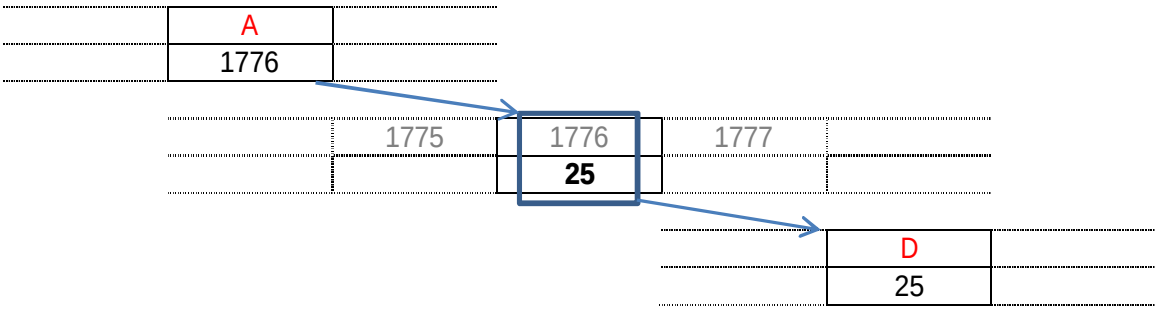
Operador de indirección o desreferencia (*)

Usando un puntero se puede tener acceso directo a un valor almacenado en la variable a la que hace referencia. Para hacer esto, simplemente es necesario preceder el identificador del puntero con un asterisco (*), realizando una operación que se puede traducir como “Valor apuntado por”.

Por tanto, siguiendo con la analogía que se ha realizado anteriormente, si se tiene que:

```
D=*A
```

Lo que se traduce como “D igual al valor apuntado por A”:



Se ve claramente la diferencia: A almacena el valor 1776, mientras que *A se refiere al valor almacenado en la celda número 1776 de la memoria, que en éste caso es 25. Es curioso que en ningún momento se hace referencia a la variable que tiene como valor el 25 (que anteriormente era B). Por tanto, como resumen se tiene que:

-& es el operador referencia y puede sr leído como “la dirección de”.
-* es el operador indirección y se puede leer como “el valor apuntado por”.

Ambos tienen significados complementarios (o contrarios). Una variable referenciada con & puede ser desreferenciada con *.

Por último, y para terminar de comprender los distintos significados se van a analizar las siguientes declaraciones, ésta vez, a través de asignaciones:

```
B==25
&B==1776
A==1776
*A==25
```

La primera expresión está clara considerando que el operador de asignación realizado a B es B=25. El segundo usa el operador de referencia (&), donde devuelve la dirección de la variable B, donde se asume que es 1776. En el tercero se recuerda que anteriormente se asignó a A el valor de &B (1776), de modo que en el cuarto, al aplicarle el operador de desreferencia (*), se obtendrá el valor de B (25).

Declaración del tipo de variable de un puntero

Una vez visto la utilidad de los operadores punteros, se torna necesario especificar en su declaración cuál es el tipo de variable al que se hace referencia a través del puntero. No es lo mismo que se haga referencia a un tipo *char, int* o *float.*

La declaración de punteros cumple el siguiente esquema:

```
tipo * nombre;
```

Donde *tipo* especifica el tipo de dato el cual se va a apuntar. Éste tipo no es el tipo del apuntador, sino el tipo del dato que el apuntador hace referencia. Por ejemplo:

```
int * numero;
char * caracteres;
float * PI;
```

Son tres declaraciones de punteros. Cada una se entiende que hace referencia a un tipo distinto de variable. De hecho, todos son puntadores y todos necesitan el mismo espacio en la memoria. Sin embargo, los datos a los que se apuntan no tienen el mismo tamaño, ya que cada uno es de un tipo distinto (*int, char* y *float*). Aun así, aunque las necesidades de memoria en los apuntadores es la misma, se distinguen los tipos: *int*, char** e *float*,* dependiendo del dato al que se apunta.

Ahora, como ejemplo se plantea el siguiente código:

```
1  #include <iostream>
2  using namespace std;
3  int main ()
4  {
5  int valor1, valor2;
6  int * miapuntador;
7  miapuntador = &valor1;
8  *miapuntador = 10;
9  miapuntador = &valor2;
10 *miapuntador = 20;
11 cout << "valor 1 es " << valor1 << endl;
12 cout << "valor 2 es " << valor2 << endl;
13 return 0;
14 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
valor 1 es 10
valor 2 es 20
```

Se observa que no se le da un valor directamente a las variables *valor1* y *valor2*, sino que le son asignadas a través de un juego de apuntadores en las líneas 7 a 10, recibiendo el valor de los distintos valores que adquiere la variable *miapuntador*. Se demuestra, por tanto, que un apuntador puede tomar distintas variables a lo largo de un programa.

A continuación se muestra un ejemplo un poco más elaborado:

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int primervalor=5,segundovalor=15;
6       int *p1,*p2;
7       p1=&primervalor;     //p1=direccion de primervalor
8       p2=&segundovalor;    //p2=dirección de segundovalor
9       *p1=10;              //valor apuntado por p1=10
10      *p2=*p1;             //v. apuntado por p2=V.apuntado por p1
11      p1=p2;               //p1=p2 (valor del apuntador es copiado)
12      *p2=20;              //valor apuntado por p1=20
13      cout<<"primer valor es "<<primervalor<<endl;
14      cout<<"segundo valor es "<<segundovalor<<endl;
15      return 0;
16  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
primer valor es 10
segundo valor es 20
```

Es importante darse cuenta que las expresiones con apuntadores *p1* y *p2* se encuentran sin y con el operador desreferencia (*). Por otro lado, es destacable la forma en la que son declarados (línea 6), ya que es necesario indicar ambos con el asterisco. Por otro lado, el tipo de la segunda variable declarada, en caso de que se requiera un *int* simple, bien se podría poner:

```
int *p1, p2;
```

Punteros y *arrays*

El concepto de *array* está muy ligado al de un puntero. De hecho, el identificador de un *array* es equivalente a la dirección del primer elemento, como un puntero es equivalente a la dirección del primer elemento al que apunta, por lo que ambos parten del mismo concepto. Por ejemplo, planteando las declaraciones siguientes:

```
int numeros[20];
int *p;
```

La siguiente operación de asignación podría ser válida:

```
p = numeros;
```

Después de todo, *p* y *números* podrían ser equivalentes y podrían tener las mismas propiedades. La única diferencia es que podría cambiar el valor del puntero *p* por otro, mientras *números* siempre apuntará al primer valor de los 20 elementos tipo *int* que hayan sido definidos. Por otro lado, semejante a *p*, que es un puntero ordinario, *números* es un *array*, por lo que puede ser considerado un *puntero constante*. De modo que la siguiente declaración no puede ser válida:

```
numeros = p;
```

Porque *números* es un *array*, operando como *puntero constante*, y no se pueden asignar valores a constantes.

Debido a las características de las variables, todas las expresiones que incluyen punteros en el siguiente ejemplo, son perfectamente válidas:

```
1   #include <iostream>
2   using namespace std;
3   int main(){
4       int numeros[5];
5       int *p;
6       p=numeros; *p=10;
7       p++; *p=20;
8       p=&numeros[2]; *p=30;
9       p=numeros +3; *p=40;
10      p=numeros; *(p+4)=50;
11      for (int n=0; n<5; n++)
12      cout<<numeros[n]<<", ";
13      return 0;
14  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
10, 20, 30, 40, 50,
```

En el capítulo acerca de los *arrays* se vio cómo se usan las llaves de tipo [ ] varias veces con el objetivo de especificar el índice de un elemento del *array* al que se refiere. En el ejemplo anterior, los operadores entre llaves [ ], actúan también como un operador de desreferencia, conocido como *operador offset* (*compensación*). Éstos desreferencian a la variable que les sigue, al igual que haría el operador *, pero también, por otro lado, se puede sumar el valor entre paréntesis a la dirección a la que se desreferencia. Por ejemplo:

```
a[5]=0;      //a[offset de 5]=0
*(a+5)=0;    //valor apuntado por(a+5)=0
```

Ambas expresiones son válidas y equivalentes si *a* es un apuntador o si *a* es un *array*.

Inicialización de un puntero

Cuando se declaran punteros los cuales se van a especificar explícitamente cual es la variable que va a ser apuntada, se pueden seguir varias vías, como por ejemplo:

```
int numero;
int *A=&numero;
```

El comportamiento del código anterior es equivalente a:

```
int numero;
int *A;
A=&numero;
```

Cuando se inicializa un puntero siempre se tiene que asignar el valor de referencia que el puntero apunta (*A*), nunca el valor que se señaló (**numero*). Se debería aclarar que en el

momento de la declaración de un puntero, el asterisco (*) indica sólo que es un puntero, y no que se trata del operador desreferencia (aunque en ambos se use el mismo símbolo). Se trata de dos funciones distintas con el mismo símbolo. No se debe confundir con:

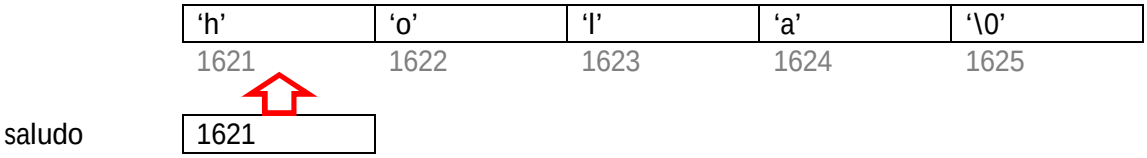
```
int numero;
int *A;
*A=&numero;
```

Éste último ejemplo es incorrecto, y no tiene sentido.

Como en el caso de los *arrays*, el compilador permite el caso especial en el que se quiera inicializar el contenido al que señala el puntero con constantes en el mismo momento en el que el puntero se declara:

```
char * saludo = "hola";
```

En éste caso, un espacio en la memoria es reservado para almacenar *hola*, y el puntero del primer caracter del bloque de memoria es asignado a *saludo*. Si se imagina que *hola* es almacenado en bloques sucesivos de memoria, cuyas direcciones comienzan con la dirección 1621, se puede representar la anterior declaración como:



Es importante indicar que *saludo* contiene el valor 1621, y no *h* ni *hola*, aunque 1621 es, de hecho, la dirección de ambos.

El puntero *saludo* señala a una secuencia de caracteres, la cual puede ser leída como si fuera un *array*. Por ejemplo, se tendrá acceso al quinto elemento del *array* con alguna de las siguientes expresiones:

```
*(saludo+4)
saludo[4]
```

Punteros aritméticos

La realización de operaciones aritméticas con puntero es un poco diferente que lo visto para los tipos de variables ordinarios. Para comenzar, sólo las operaciones de suma y resta tienen sentido dentro de las operaciones aritméticas con punteros, pero por otra parte, tanto como la suma como la resta pueden tener diferentes comportamientos con punteros de acuerdo con el tamaño y tipo de datos que señalan.

Cuando se vieron los tipos fundamentales de variables, se analizó que cada tipo ocupa más o menos espacio en la memoria. Por ejemplo, asumiendo como ejemplo en una computadora y un compilador dado, una tipo *char* ocupa 1 byte, mientras que un *short* y un *long* requieren 2 y 4 bytes respectivamente.

Suponiendo que definen los siguientes tres punteros en un compilador:

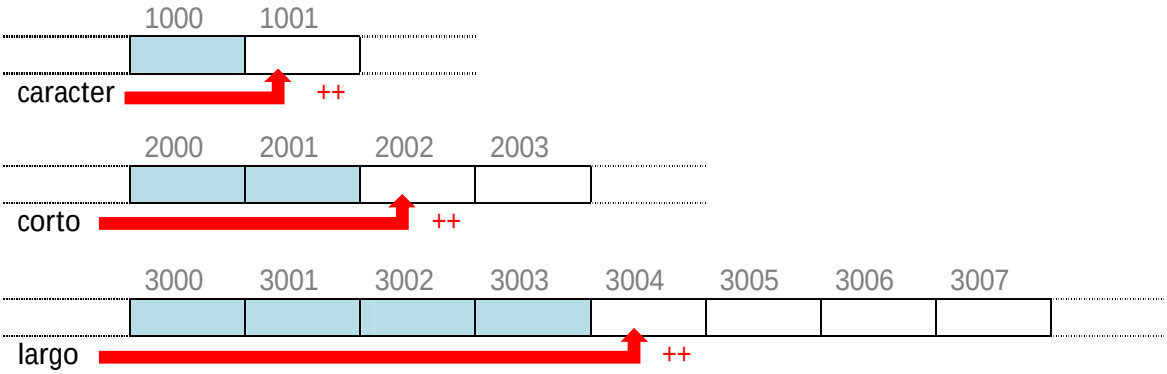
```
char *caracter;
short *corto;
long *largo;
```

Y considerando que, hipotéticamente, ocupan cada uno las posiciones de memoria cuyas direcciones son 1000, 2000 y 3000 para *caracter, corto* y *largo* respectivamente.

Si se declara:

```
caracter++;
corto++;
largo++;
```

Caracter, tal y como se espera, para a contener el valor 1001, pero *corto* no pasa a contener el valor 2001, sino el 2002, y *largo* pasa a contener el 3004, en vez de incrementarse en 1. Esto es debido a que al añadir una unidad al puntero, se señala al siguiente elemento del mismo tipo que el que se ha definido, y, por lo tanto, el tamaño en bytes del tipo señalado se añade al puntero:



Esto es aplicable cuando se añade o substrae cualquier número a un puntero. El efecto sería el mismo con el código:

```
caracter = caracter + 1;
corto = corto + 1;
largo = largo + 1;
```

Tanto el operador incremento (++) como decremento (--) tienen preferencia en orden de aplicación de órdenes que el operador desreferencia (*), pero ambos tienen un especial significado cuando son usados como sufijo (la expresión es evaluada con el valor que tenía antes del incremento). Aun así, la siguiente expresión puede llevar a confusión:

```
*p++
```

Como ++ tiene prioridad sobre *, dicha expresión es equivalente a **(p++)*. Por lo tanto, se consigue incrementar el valor de *p* (con el objetivo de apuntar al siguiente elemento). Por otro lado, para apuntar a un valor antes de ser incrementado, la sintaxis sería:

```
(*p)++
```

Aquí, la expresión habría sido evaluada con el valor apuntado por p incrementado en uno. El valor de *p* (al apuntador propiamente dicho) no habría sido modificado (lo que se modifica es lo que ha señalado el puntero).

Para aclarar lo expuesto, dado el siguiente código:

```
*p++= *q++;
```

Y sabiendo que ++ tiene mayor preferencia que*, tanto *p* como *q* son incrementados, pero como los operadores ++ están situados como sufijos, el valor asignado a **p* es **q* antes de que p y q sean incrementados. Tras esto, son incrementados. Esto podría haber sido declarado como:

```
*p = *q;
++p;
++q;
```

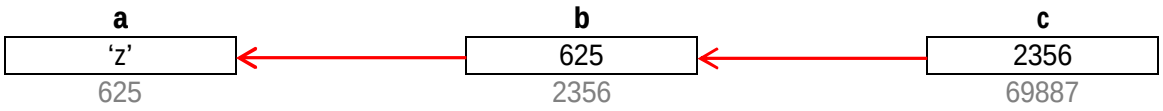
En cualquier caso, si el orden de preferencia no se tiene claro, siempre es recomendable el uso de paréntesis ( ) para evitar resultados inesperados, o simplemente, para dar al código más legibilidad.

Punteros de punteros

C++ permite el uso de punteros que señalan a otros punteros, señalando éstos últimos, a otros datos u otros punteros. Para realizar esto, tan solo es necesario añadir otro asterisco en cada nivel de referencia en la declaración:

```
char a;
char *b;
char **c;
a='z';
b=&a;
c=&b;
```

Si se le asignan las direcciones a las celdas de memoria que contienen cada dato, por ejemplo, el 625 a *a*, el 2356 a *b*, y 69887 a *c,* el proceso seguido se puede ejemplificar como:



El valor que toma cada variable está contenido dentro de la celda, estando las direcciones de cada celda justo debajo de esta. Si se analiza *c*, y las formas en las que puede ser declarado, se comprueba que adquiere distintos significados, de forma que:

-*c* es tipo *char*** con el valor *2356*.

-**c* es tipo *char** con el valor *625*.

-***c* es tipo *char* con valor *'z'*.

Punteros tipo *void*

El puntero tipo *void* es especial. En C++, *void* representa la ausencia de tipo de variable, por lo que loa puntero *void* apuntan a valores sin tipo de variable (por lo tanto, con longitudes y propiedades de desreferencia desconocidas).

Esto permite que los puntero *void* señalen cualquier tipo de variable, desde un entero o *float* hasta una cadena de caracteres. Pero acarrean una gran limitación: los datos apuntados por éstos no pueden ser directamente desreferenciados, por lo que siempre se tendrá que obtener su dirección en el puntero *void* a algún otro tipo de puntero que señale a un tipo concreto de variable antes de desreferenciarlos.

Una de las funciones en los que pueden ser usados éste tipo de punteros, es para pasar parámetros genéricos a una función:

```
1   #include <iostream>
2   using namespace std;
3   void incremento (void*datos, int longitud)
4   {
5       if (longitud==sizeof(char)){
6           char*caracter; caracter=(char*)datos; ++(*caracter);
7       }
8       else if (longitud == sizeof(int))
9       {int* entero; entero=(int*)datos; ++(*entero);
10      }
11  }
12  int main()
13  {
14      char a = 'x';
15      int b = 2525;
16      incremento (&a,sizeof(a));
17      incremento (&b,sizeof(b));
18      cout<<a<<", "<<b<<endl;
19      return 0;
20  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
y, 2526
```

En las líneas 5 y 8 se aplica el operador *sizeof*, integrado en la librería estándar de C++, el cual devuelve el tamaño en bytes del parámetro indicado. Para memoria no dinámica éste valor es constante. Por lo tanto, por ejemplo *sizeof (char)* es 1, ya que las variables tipo *char* son de 1 solo byte.

En puntero nulo

Un puntero nulo es un apuntador regular, de cualquier tipo de variable, el cual adquiere un valor especial indicando que no está indicando ninguna referencia válida o dirección de memoria existente. Éste valor es el resultado de asignar el valor 0 a cualquier tipo de puntero:

```
int *p;
p=0;     //p tiene un valor de puntero 0;
```

Éste no debe confundirse con los punteros tipo *void*. Un puntero nulo es un valor que cualquier puntero puede adquirir para representar que no está señalando nada, mientras que un puntero tipo *void* es un tipo especial de apuntador el cual puede señalar a cualquier valor sin un tipo de variable específica.

Puteros de funciones

C++ permite apuntar a funciones. El típico uso de ésta herramienta es para pasar una función como un argumento de otra función, ya que no se puede pasar sin referencia. Con el objeto de declarar un puntero a una función que ha sido declarada, es como el prototipo de una función, excepto que el nombre de la función se cierra entre paréntesis () más un asterisco insertado antes de su identificador:

```
1   #include <iostream>
2   using namespace std;
3   int suma (int a, int b)
4   { return (a+b); }
5   int resta (int a, int b)
6   { return (a-b); }
7   int operacion (int x, int y, int
8   (*llamarfuncion)(int,int))
9   {
10  int g;
11  g = (*llamarfuncion)(x,y);
12  return (g);
13  }
14  int main ()
15  {
16  int m,n;
17  int (*menos)(int,int) = resta;
18  m = operacion (7, 5, suma);
19  n = operacion (20, m, menos);
20  cout <<n;
21  return 0;
22  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
8
--------------------------------
```

En el ejemplo, *menos* es el puntero de la función que tenga dos parámetros de tipo *int*. Ésta es inmediatamente asignada a la función *resta* (línea 17).

Memoria Dinámica

Introducción

Hasta ahora, en los programas propuestos, la cantidad de memoria requerida en los programas, venía determinada por la cantidad, tipo y tamaño de las variables declaradas en el código, antes de ejecutar el programa. Pero puede pasar, y es muy común, que aparezcan variables cuyo tamaño no esté definido en el código, sino que sea consecuencia de alguna operación durante la ejecución de un programa, como por ejemplo, en el caso en el que se requiera un dato de entrada definido por el usuario. Para responder a éstas necesidades, el lenguaje C++ implementa, a través de los comandos *new* y *delete*, lo que se conoce como *memoria dinámica*.

Operadores *new*

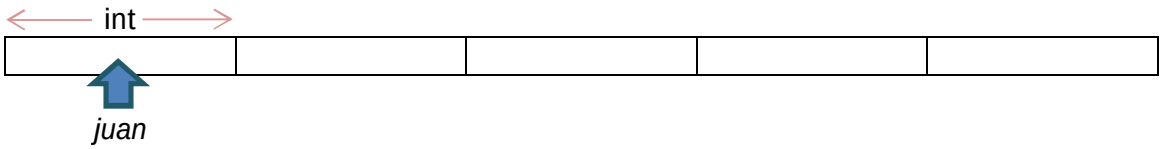
Para solicitar memoria dinámica, se recurre al operador *new*. El operador, irá seguido por el tipo de dato especificado y el número necesario de éstos entre corchetes [ ]. El operador devolverá un puntero con la dirección del comienzo del bloque de memoria guardado a la variable. El operador *new* es de la forma:

```
puntero=new Tipo;
puntero=new tipo[numero_de_elementos];
```

La primera expresión del ejemplo, es usada para localizar la memoria, la cual sólo contendrá el elemento de tipo *Tipo*. La segunda expresión es usada para asignar un bloque de elementos (*array*) de tipo *Tipo*, siendo el *numero_de_elementos* la longitud del *array*. Por ejemplo:

```
int * juan;
juan=new int[5];
```

En éste caso, el sistema asigna dinámicamente espacio para 5 elementos de tipo *int* y devuelve el puntero del bloque de memoria, relacionándolo con *juan*. Por lo tanto, *juan* apunta a un bloque válido de memoria con espacio suficiente para 5 enteros:



Al primer elemento apuntado por *juan* se puede tener acceso mediante la expresión *juan[0]* o la expresión **juan.* Ambos son equivalentes tal y como se analizó en la sección sobre punteros. Para acceder al segundo elemento se declarará como *juan[1],* o *(juan+1)*, y así sucesivamente.

Aunque pueda parecer similar o equivalente a la propia declaración de un *array* simple, hay una gran diferencia: el tamaño de un *array* ordinario debe ser un valor constante, cuyo tamaño límite se ha de decidir en el momento de la declaración (antes de ser ejecutado), mientras que la asignación y localización de memoria dinámica durante la ejecución del programa (*runtime*), usando una variable o constante para definir su tamaño.

Los datos introducidos en la memoria son almacenados por esta, de modo que puede llegar a saturarse si se requiere mucho. Por lo tanto, es importante comprobar que los datos introducidos han sido correctamente almacenados o no:

C++ proporciona dos métodos estándar para chequear que la operación ha sido satisfactoria.

La primera opción es mediante manejo de excepciones. Usando éste método, una excepción de tipo *bad_alloc* es lanzada cuando la localización falla. La definición de manejo por excepciones se explica más adelante detenidamente. En éste caso, sólo es importante saber que el programa lanzará una excepción si se detecta que la dirección de los datos falla, o simplemente, lo finalizará. Éste método se utiliza por defecto por el operador *new*, y es usado mediante la declaración:

```
juan =new int[5]; //Si falla, se lanza una excepción
```

Otro método es el conocido como *nothrow*, el cual, cuando una localización de memoria falla, incluso lanzando una excepción *bad_alloc* u ordenando terminar el programa, el puntero devuelto por *new* es un puntero nulo, por lo que el programa sigue su ejecución.

Éste método es especificado mediante *nothrow*, declarado con el cabecero <*new*>, como argumento de *new*:

```
juan= new (nothrow) int[5];
```

En éste caso, si la localización de éste bloque de memoria falla, el fallo puede ser detectado si se chequea le valor del puntero, de la forma:

```
int *juan;
juan= new(nothrow) int [5];
if (juan==0){
    //error al asignar memoria. tomar medidas.
}
```

Éste método (*nothrow*) requiere mayor trabajo que el método por excepción visto antes, ya que el valor devuelto debe ser chequeado en cada puntero de memoria para detectar el fallo, por lo que no se suele usar en grandes proyectos de programación, recurriendo normalmente al método por excepción (será explicado más adelante).

Operadores *delete*

La cantidad de memoria dinámica durante la ejecución de un programa es limitada en muchos casos, por lo que puede ser necesario liberar memoria dinámica utilizada con anterioridad en caso que no se necesiten los datos almacenados. Éste es el propósito del operador *delete*:

```
delete puntero;
delete [] puntero;
```

La primera expresión del ejemplo podría ser usada para eliminar memoria ocupada por un elemento sencillo, y el segundo por elementos de tipo *array*.

El valor a eliminar puede ser un puntero que señale un bloque de memoria previamente asignado mediante *new* o incluso un puntero nulo (en cuyo caso no se producirá ningún efecto, pero tampoco ningún fallo).

```
1  #include<iostream>
2  #include<new>
3  using namespace std;
4
5  int main()
6  {
7      int i,n;
8      int *p;
9      cout<<"¿Que cantidad de numeros quieres escribir? ";
10     cin>>i;
11     p=new(nothrow) int[i];
12     if (p==0)
13     cout<<"Error: la memoria no ha sido localizada";
14     else
15     {
16         for (n=0;n<i;n++)
17         {
18         cout<<"introduce un numero: ";
19         cin>>p[n];
20         }
21     cout<<"Los numeros introducidos ha sido: ";
22     for(n=0;n<i;n++)
23     cout<<p[n]<<", ";
24     delete[] p;
25     }
26     return 0;
27 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
¿Que cantidad de numeros quieres escribir? 4
introduce un numero: 12
introduce un numero: 24
introduce un numero: 46
introduce un numero: 68
Los numeros introducidos ha sido: 12, 24, 46, 68,
```

En el ejemplo se muestran las características descritas. Por ejemplo, en la línea 11, se asigna la memoria necesaria para almacenar los números introducidos por el usuario (*i*), por lo que ya no se trata de una constante. Debido a ésta opción, el usuario podrá intentar introducir grandes cantidades. En el caso de que se declare un *i* tan alto que la memoria se vea saturada, se tiene un mensaje preparado para éste caso (líneas 12 y 13).

Memoria dinámica en C

Los operadores *new* y *delete* son propios del lenguaje C++, por lo que no existen en lenguaje C. Sin embargo, usando el leguaje C puro y sus bibliotecas, se puede usar la memoria dinámica mediante funciones como *malloc, calloc, realloc* y *free*, las cuales están presentes en C++ usando el archivo de cabecera <*cstdlib*>.

Los bloques de memoria asignados a éstas funciones no son necesariamente compatibles cuando se devuelven a través de *new*, por lo que éstas funciones han de ser manipuladas con sus propias funciones y operadores de C.

Estructuras de datos

Introducción

Hasta ahora se han visto el comportamiento de grupos secuenciales de datos en C++. Estas estructuras son a veces restrictivas, ya que no siempre se tendrá que almacenar una serie de datos o elementos del mismo tipo, sino un conjunto de diferentes elementos con diferentes tipos de datos.

Estructuras de datos

Una estructura de datos, es un grupo de elementos dato agrupados bajo un mismo nombre. Éstos elementos dato, conocido como *miembros*, pueden ser de diferentes tipos o diferentes longitudes. La sintaxis seguida en C++ para declarar una estructura de datos es la siguiente:

```
struct nombre_de_la_estructura {
    tipo_de_miembro_1 nombre_del_miembro_1;
    tipo_de_miembro_2 nombre_del_miembro_2;
    tipo_de_miembro_3 nombre_del_miembro_3;
    .
    .
}nombre_de_objetos;
```

Donde *nombre_de_la_estructura* es el nombre del tipo de estructura, *nombre_de_objetos* puede ser un conjunto de identificadores válidos para objetos que tengan el tipo de ésta estructura. Entre llaves { } se declara el conjunto de miembros dato, donde cada uno es especificado con su tipo e identificador válido.

La primera característica que se debe conocer es que una estructura de datos crea un nuevo tipo: por cada estructura de datos creada, se crea un nuevo tipo de variable cuyo nombre es el identificador de la estructura de datos (*nombre_de_la_estructura*), el cual puede ser utilizado en el resto del programa como cualquier otro tipo de variable. Por ejemplo:

```
struct producto{
    int peso;
    float precio;
};

producto manzana;
producto platano, sandia, fresa;
```

Primero se ha declarado un nuevo tipo de estructura llamada *producto*, con dos miembros: *peso* y *precio,* cada uno con un tipo de variable fundamental. Tras esto, se ha utilizado el nombre de la estructura para declarar cuatro objetos de tipo *producto: manzana, platano, sandia* y *fresa.*

Una vez declarado, *producto* se convierte en un nuevo tipo de variable, al igual que son *int, float* o *char*, y a través de éste, se pueden declarar distintos objetos (variables) como son *manzana, platano, sandia* o *fresa*.

Tal y como se vio anteriormente en la sintaxis de la estructura, los objetos del tipo *producto,* pueden ser declarados la final de la declaración de la estructura de datos, al menos los que son conocidos en ése momento. Por lo que el ejemplo anterior podría haberse declarado, de forma más compacta, como:

```
struct producto{
    int peso;
    float precio;
}manzana, platano, sandia, fresa;
```

Una vez se han declarado los distintos objetos de una determinada estructura, se puede por tanto, proceder a operar con los miembros directamente. Para hacer esto, se usa un punto (.) insertado entre el nombre del objeto y el nombre del miembro. Por ejemplo, se puede operar con cualquiera de los miembros si éstos son objetos estándar de sus respectivos tipos:

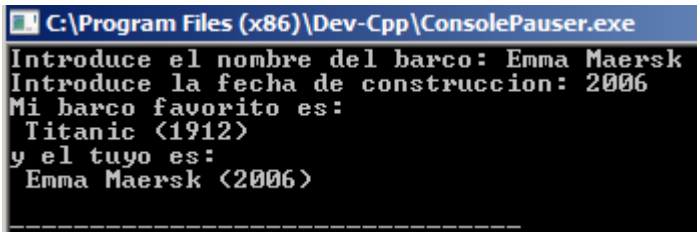
```
manzana.peso
manzana.precio
fresa.precio
fresa.peso
```

Donde cada uno de los objetos hace referencia a un miembro determinado y adopta el tipo de variable con la que el miembro fue declarado: *manzana.peso* y *fresa.peso* serán de tipo *int* mientras que *manzana.precio* y *fresa.precio* serán de tipo *float*.

A continuación se muestra un ejemplo donde se puede apreciar el uso de una estructura de datos como un tipo de variable fundamental:

```
 1  #include <iostream>
 2  #include <string>
 3  #include <sstream>
 4  using namespace std;
 5  struct barcos {
 6  string nombre;
 7  int fecha;
 8  } mio, tuyo;
 9  void printbarcos (barcos nombre);
10  int main ()
11  {
12  string mistr;
13  mio.nombre = "Titanic";
14  mio.fecha = 1912;
15  cout << "Introduce el nombre del barco: ";
16  getline (cin,tuyo.nombre);
17  cout << "Introduce la fecha de construccion: ";
18  getline (cin,mistr);
19  stringstream(mistr) >> tuyo.fecha;
20  cout << "Mi barco favorito es:\n ";
21  printbarcos (mio);
22  cout << "y el tuyo es:\n ";
23  printbarcos (tuyo);
24  return 0;
25  }
26  void printbarcos (barcos barco)
27  {
28  cout << barco.nombre;
29  cout << " (" << barco.fecha << ")\n";
30  }
```

Obteniendo en la ejecución del programa:



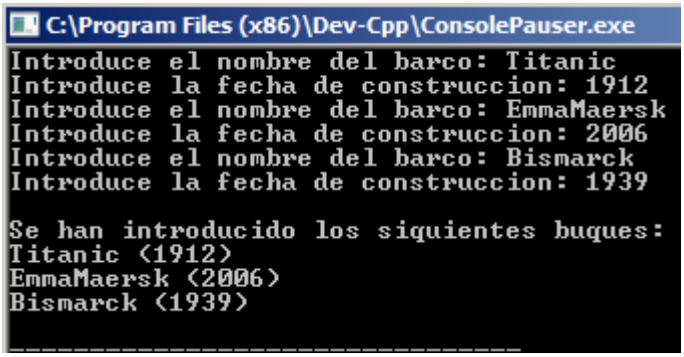
El ejemplo muestra cómo pueden ser usados los miembros junto a los objetos como variables. Por ejemplo, el *tuyo.fecha* es un tipo de variable *int*, y *mio.nombre* es un tipo *string*.

Los objetos *mio* y *tuyo* pueden también ser tratados como variables de tipo *barcos*, como se hace por ejemplo en la función *printbarcos*. Por lo tanto, una de las ventajas más importantes de las estructuras de datos es que, o bien nos podemos referir a sus miembros de forma individual, o a toda la estructura (como un bloque) con un solo identificador.

Las estructuras de datos son también muy usadas para representar bases de datos, especialmente si se considera la posibilidad de construir *arrays* de éstas:

```
 1  #include <iostream>
 2  #include <string>
 3  #include <sstream>
 4  using namespace std;
 5  #define N_barcos 3
 6  struct barcos {
 7  string nombre;
 8  int fecha;
 9  } buques [N_barcos];
10  void printbarco (barcos barco);
11  int main ()
12  {
13  string mistr;
14  int n;
15  for (n=0; n<N_barcos; n++)
16  {
17  cout << "Introduce el nombre del barco: ";
18  getline (cin,buques[n].nombre);
19  cout << "Introduce la fecha de construccion: ";
20  getline (cin,mistr);
21  stringstream(mistr) >> buques[n].fecha;
22  }
23  cout << "\nSe han introducido los siquientes buques:\n";
24  for (n=0; n<N_barcos; n++)
25  printbarco (buques[n]);
26  return 0;
27  }
28  void printbarco (barcos barco)
29  {
30  cout << barco.nombre;
31  cout << " (" << barco.fecha << ")\n";
32  }
```

Obteniendo la respuesta al ejecutar de la forma:



Punteros a estructuras de datos

Al igual que a cualquier otro tipo de variable, las estructuras pueden ser apuntadas por su propio tipo de punteros:

```
struct barcos{
    string nombre;
    int fecha;
};

barcos buque;
barcos *pbuque;
```

Donde *buque* es un objeto de la estructura de tipo *barcos*, y *pbuque* es el puntero de *buque* (su contenido a través de la dirección en la memoria).

A continuación se muestra un ejemplo donde se introducen punteros, y a la vez, un nuevo operador: el operador de *flecha* (->):

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>
4  using namespace std;
5  struct barcos {
6  string nombre;
7  int fecha;
8  };
9  int main ()
10 {
11 string mistr;
12 barcos buque;
13 barcos * pbuque;
14 pbuque = &buque;
15 cout << "Introduce en nombre del buque: ";
16 getline (cin, pbuque->nombre);
17 cout << "introduce la fecha de construccion: ";
18 getline (cin, mistr);
19 (stringstream) mistr >> pbuque->fecha;
20 cout << "\nHas introducido:\n";
21 cout << pbuque->nombre;
22 cout << " (" << pbuque->fecha << ")\n";
23 return 0;
24 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
Introduce en nombre del buque: Principe de asturias R11
introduce la fecha de construccion: 1983

Has introducido:
Principe de asturias R11 (1983)
```

En el ejemplo se muestra que el operador de flecha, equivale en su uso al operador *desreferencia* visto en el capítulo de punteros, sólo que su uso es exclusivo para relacionar datos de miembros de estructuras de datos. Dicho operador sirve para acceder a un miembro de un objeto al que se hace referencia. En la línea 16 se declara que:

```
pbuque->nombre
```

Que también podría haberse escrito como:

```
(*pbuque).nombre
```

Ambas expresiones son válidas y producen el mismo efecto, evaluando el miembro *nombre* de la estructura de datos apuntado por el puntero *pbuque.* Por otro lado, se debe diferenciar de la declaración:

```
*pbuque.nombre
```

Que es equivalente a:

```
*(pbuque.nombre)
```

Ambas acceden al valor apuntado por un hipotético puntero llamado *nombre* de la estructura objeto *pbuque* (que en éste caso ya no se trata de un puntero). El siguiente cuadro muestran las posibles combinaciones de punteros y miembros de una estructura:

Expresión	Que es lo que evalúa	Equivalencia
a.b	Miembro *b* del objeto *a*	
a->b	Miembro *b* del objeto apuntado por *a*	(*a).b
*a.b	Valor apuntado por el miembro *b* del objeto *a*	*(a.b)

Estructuras anidadas

Las estructuras de datos pueden ser anidadas de manera que un elemento de una estructura puede también ser válido para otra estructura:

```
struct barcos{
    string nombre;
    int fecha;
};

struct astilleros{
    string nombre;
    string email;
    buques;
}Navantia;

astilleros *pastilleros=&Navantia;
```

Tras la declaración del ejemplo, se podrán usar cualquiera de las siguientes expresiones:

```
Navantia.nombre;
Navantia.buques.nombre
Navantia.buques.fecha
pastilleros->buques.fecha
```

Donde los dos últimos casos son ejemplos de cómo se puede referir al mismo miembro.

Otros tipos de datos

Tipos de datos *definidos*

C++ permite la definición de tipos de datos definidos en base a los ya existentes. Ésta operación se permite a través de la palabra clave *typedef*, cuyo formato es:

```
typedef tipo_existente nombre_del_nuevo_tipo;
```

Donde *tipo_existente* es un tipo fundamental de C++ o un tipo compuesto, y *nombre_del_nuevo_tipo* es en nombre del nuevo tipo definido. Por ejemplo:

```
typedef char C;
typedef unsigned int Palabra;
typedef char * pChar;
typedef char campo [50];
```

En éste caso se han definido los tipos *c, Palabra, pChar* y *campo* como *char, unsigned int, char** y *char [50]* respectivamente, por lo que se podrán usar en declaraciones a posteriori como cualquier otro tipo válido de variable:

```
C mychar, anotherchar, *ptc1;
Palabra mi_palabra;
pChar ptc2;
campo nombre;
```

Más concretamente, y como se ha visto en los ejemplos, *typedef* no crea un tipo diferente, sino que crea un sinónimo de uno existente. Esto significa que el tipo de *mi_palabra* del ejemplo anterior, puede ser considerado tanto como *Palabra* o como *unsigned int*, siendo ambos el mismo tipo.

Puede ser útil el uso de *typedef* como un alias de un tipo de uso frecuente en un programa. Como ejemplo, para evitar anglicismos, se podría plantear en la cabecera de cada programa creado en España, y para uso exclusivo de personas de habla española, un código el cual *traduzca* los tipos fundamentales en su significado en español.

Por otro lado, también es muy útil a la hora de variar el tipo de una variable muy utilizada en un programa. Por ejemplo, si se está programando, y en determinado momento se da cuenta de que una variable muy utilizada, definida como tipo *typedef int a*, se conviene cambiar a tipo *float*, por ejemplo, bastaría con cambiar la definición en la cabecera del tipo como *typedef float a*.

Uniones

Las uniones permiten que se tenga acceso a una misma porción de memoria a través de diferentes tipos de datos, ya que éstos tienen asignado el mismo bloque de memoria. Su declaración y uso es similar a las estructuras de datos, pero su funcionalidad es totalmente diferente:

```
union nombre_de_union{
    Tipo_miembro_1 nombre_miembro_1;
    Tipo_miembro_2 nombre_miembro_2;
    Tipo_miembro_3 nombre_miembro_3;
    .
    .
} nombre_objetos;
```

Todos los elementos de la declaración *unión* ocupan la misma porción de memoria. Su tamaño será el del mayor elemento declarado en la lista de miembros. Por ejemplo:

```
union mistipos {
    char c;
    int i;
    float f;
}mistipos;
```

Define los tres elementos:

```
mistipos.c
mistipos.i
mistipos.f
```

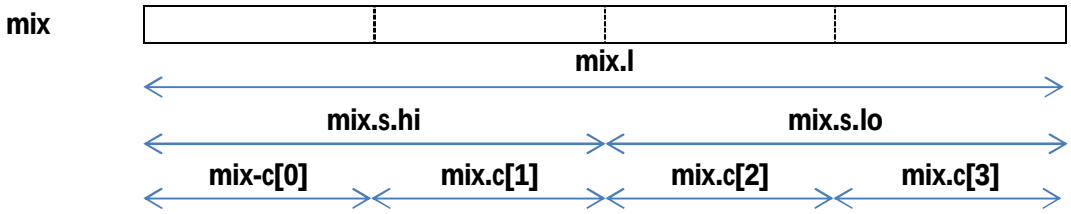
Donde cada uno tiene su propio tipo de variable. Como cada uno accede a la misma localización de memoria, la variación de cualquiera de ellos afecta a todas las variables de la unión, o dicho de otro modo, no se podrán almacenar distintos valores de forma independiente en cada uno de loa miembros.

Un uso común de éste tipo de estructura es la unión de un tipo elemental en un *array* o estructura de datos de elementos menores. Por ejemplo:

```
union mix_t {
    long l;
    struc{
    short hi;
    short lo;
    } s;
    char c[4];
}mix;
```

Se definen tres nombres los cuales permiten el acceso al mismo grupo de 4 bytes: *mix.l, mix-s* y *mix.c*, y los cuales pueden ser usados como se quiera para acceder a esos 4 bytes, tanto si se requiere un tipo *long*, como dos tipo *short*, o un *array* de elementos tipo *char,* respectivamente. Si se tienen tipos mezclados, *arrays* y estructuras de datos en la unión, se tendrán por tanto distintas configuraciones para tener acceso a los datos almacenados. Éstos pueden ser representados como:

El orden exacto de los miembros de la unión en la memoria depende de la plataforma del PC, por lo que se tendrá que tener especial cuidado en el uso y aplicación de ésta estructura, ya que podrá crear problemas de portabilidad.



Uniones anónimas

En C++ se tiene la opción de declarar uniones anónimas. Si se declara una unión sin ningún nombre (unión anónima), se podrá tener acceso a sus miembros directamente a través de los nombres de los miembros. Por ejemplo, si se analizan las diferencias entre las dos estructuras se tiene:

Estructura con unión regular	Estructura con unión anónima

`struct {`
`  char Nombre [50];`
`  char Armador [50];`
`  union {`
`  float Euros;`
`  int Dolares;`
`  }precio;`
`}Buque;`
	`struct {`
	`  char Nombre [50];`
	`  char Armador [50];`
	`  union {`
	`  float Euros;`
	`  int Dolares;`
	`  };`
	`}Buque;`

La única diferencia entre los dos códigos es que en el segundo caso, la unión no tiene nombre (*precio*). La diferencia se aprecia cuando se accede a los miembros *Euros* y *Dólares* de un objeto de ese tipo. Por ejemplo, para un objeto del primer tipo, se tendrá acceso como:

```
Buque.precio.Euros
Buque.precio.Dolares
```

Mientras que para un objeto del segundo tipo, se accederá como:

```
Buque.Euros
Buque.Dolares
```

Por otro lado, y haciendo hincapié en el significado de una unión, sólo se podrá almacenar el *precio* de un *Buque* en *Euros* o *Dolares*, ya que ambos miembros tienen asignados el mismo bloque de memoria.

Enumeraciones

Las enumeraciones crean nuevos tipo de datos para contener datos sin estar limitados por los valores de los tipos fundamentales. Su estructura es de la forma:

```
enum nombre_de_enumeracion {
    valor1,
    valor2,
    valor3,
    .
    .
}nombre_objetos;
```

Por ejemplo, se podría crear un nuevo tipo de variable llamado *color* para almacenar los colores con la siguiente declaración:

```
enum colores {negro, azul, verde, rojo, rosa, amarillo,
    blanco, purpura};
```

Como se ve en el ejemplo, no son declarados con ningún tipo de dato fundamental. Lo declarado se trata de un nuevo tipo de datos que no están basados en ningún otro tipo. Los posibles valores que éste nuevo tipo (*colores*) pueden tomar serán constantes declaradas entre llaves. Por ejemplo, una vez declarado *enum colores*, las declaraciones que siguen son válidas:

```
colores micolor;
micolor= azul;
if(micolor == verde) micolor=rojo;
```

Las enumeraciones son tipos compatibles con valores numéricos, de modo que sus constantes son siempre asignadas internamente como valores numéricos enteros. Si no se especifica ningún valor, el primer valor entero equivalente será el 0, siguiendo los siguientes en progresión de +1. De modo que siguiendo la declaración de *colores* anterior, el *negro* sería equivalente al 0, el *azul* al 1, *verde* al 2, y así sucesivamente.

También se puede especificar el valor entero asignado para enumerar los tipos que se toman. Si el valor constante que sigue no viene dado por una valor entero, automáticamente se toma el valor anterior y se suma 1, siguiendo así la progresión. Por ejemplo:

```
enum meses {enero=1, febrero, marzo, abril, mayo, junio,
julio, agosto, septiembre, octubre, noviembre, diciembre
}año;
```

En éste caso, la variable *año* del tipo *meses* puede contener cualquiera de los 12 posibles valores, desde *enero* a *diciembre*, los cuales son equivalentes a los valores 1 a 12 (y no de 0 a 11, ya que a enero se le asignó el valor 1).

4.Programación orientada a objetos

Clases

Introducción

Una clase es un concepto expandido de estructura de datos: ya no sólo almacena datos, sino que es capaz de almacenar tanto datos como funciones.

Un objeto es una instanciación de una clase. En términos de variables, una clase puede ser un tipo de variable, y un objeto puede ser la variable en sí.

Las clases son generalmente declaradas usando la palabra clave *class*, con el formato siguiente:

```
class nombre_clase {
    especificador_de_acceso1:
        miembro1;
    especificador_de_acceso2:
        miembro1;
    ...
}Nombre_de_los_objetos;
```

Donde el *nombre_clase* es un identificador válido de la clase, *Nombre_de_los_objetos* es una lista opcional de nombres para objetos de ésta clase. El cuerpo de la declaración puede contener miembros y, opcionalmente, especificadores de acceso.

En general, la declaración es muy similar a las estructuras de datos, solo que ahora también se podrá incluir funciones como miembros, además de los especificadores de acceso. Dichos especificadores podrán ser *private (privado), protected (protegido)* o *public (público)*. Éstos tienen las siguientes características:

-Los especificadores *private* de una clase indican que los miembros declarados son accesibles sólo desde otros miembros de la misma clase o desde sus *friends (amigos)*.

-Los especificadores *protected* de una clase indican que los miembros declarados son accesibles desde los miembros de su misma clase y sus amigos, pero también desde miembros de sus clases derivadas.

-Finalmente, los miembros especificados como *public* son accesibles desde cualquier elemento cuando el objeto es visible.

Por defecto, todos los miembros de una clase declarada por la palabra clave *class* tienen acceso privado para todos sus miembros. Por lo tanto, cualquier miembro que sea declarado antes que un especificador de clase automáticamente tiene acceso privado. En el ejemplo:

```
class Rectángulo {
    int x, y;
    public:
    void tomar_valores (int,int);
    int area (void);
}recta;
```

Se declara una clase llamada *Rectángulo* y un objeto de ésta clase llamado *recta*. Ésta clase contiene cuatro miembros: dos miembros dato de tipo entero (los miembros *x* e *y*) con acceso privado (por defecto) y dos miembros adicionales, en éste caso funciones, con acceso público: *tomar_valores* y *área*, cuyas acciones no han sido especificadas, sólo su declaración.

Nótese la diferencia entre en nombre de la clase y el nombre del objeto: en el ejemplo anterior, *Rectángulo* es el nombre de la clase, mientras de *recta* es el nombre del objeto de tipo *Rectángulo*. La misma relación guardan a continuación *int* e *a*:

```
int a;
```

Donde *int* es el nombre del tipo de variable (la clase) y *a* es el nombre de la variable (el objeto).

Después de las declaraciones de *Rectángulo* y *recta*, se podrá realizar el cuerpo de la programación para cualquiera de los miembros públicos del objeto *recta* como si fueran funciones o variables normales, escribiendo el nombre de los objetos seguido de un punto (.) más el nombre del miembro. Todo es muy similar a la estructura de datos. Por ejemplo:

```
recta.tomar_valores (3,4);
miarea=recta.area();
```

Los únicos miembro a los que no se tiene acceso en el cuerpo del programa son *x* e *y*, ya que tienen acceso privado y sólo pueden ser referidos a través de miembros de la misma clase.

Para el caso del ejemplo *Rectángulo*, un programa válido sería:

```
1   #include<iostream>
2   using namespace std;
3   class Rectangulo{
4       int x, y;
5       public:
6           void tomar_valores(int,int);
7           int area() {return (x*y);}
8   };
9
10  void Rectangulo::tomar_valores (int a, int b){
11      x=a;
12      y=b;
13  }
14  int main(){
15      Rectangulo recta;
16      recta.tomar_valores (3,4);
17      cout<<"area: "<<recta.area();
18      return 0;
19  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
area: 12
```

En la línea 10 se introduce el operado *alcance (scope "::")* en la definición de *tomar_valores()*. Éste es usado para definir un miembro de la clase desde fuera de la definición de la propia clase.

Otra característica del programa descrito es que la función *área()* ha sido incluida directamente dentro de la definición de la clase *Rectangulo* para aumentar al máximo la simplicidad de la sintaxis. Sin embargo, *tomar_valores()* es un prototipo declarado dentro de la clase (línea 6), estando definido fuera de ésta (línea 10). Es en la línea 10 donde se declara el operador alcance (::) para especificar que se ha definido una función miembro de la clase *Rectangulo*, y que no se trata de una función global general.

El operador alcance(::) especifica la clase a la que pertenece el miembro declarado a continuación, garantizando las mismas propiedades y características de la función como si hubiera sido declarada directamente dentro de la clase. Por ejemplo, en la función *tomar_valores ()* se han de usar las variables *x* e *y*, las cuales son miembros privados de la clase *Rectangulo*, por lo que sólo se pueden acceder desde otros miembros de su clase.

La única diferencia entre la definición entre la definición completa de una función en la clase o incluir su prototipo y después su definición, es que en el primer caso la función será considerada automática como un miembro en línea por el compilador, mientras que en el segundo caso será un miembro normal de la clase (fuera de línea), En cuanto a comportamiento, no supone ninguna diferencia.

Los miembros *x* e *y* tienen acceso privado (si no se señala otra cosa, todos los miembros de una clase son privados). Al ser declarados privados, se deniega su acceso desde fuera de la clase. Esto toma sentido en el punto en el que se define un miembro de la función para que tomen valores a través del objeto: el miembro de la función *tomar_valores()*. Por lo tanto el resto del programa no necesita acceso directo a éstas. Quizás, en un ejemplo sencillo como el anterior, no se aprecia la verdadera utilidad de proteger las variables, pero en proyectos

mayores puede ser importante que los valores no puedan ser modificados de manera inesperada, al menos desde el punto de vista del objeto.

Una de las grandes ventajas de una clase es que, sea cual sea el tipo, se podrá declarar varios objetos para una misma clase. Por ejemplo, partiendo de la sintaxis de *Recangulo*, se puede declarar un objeto adicional, en este caso *rectab*, aparte de *recta*:

```
1   #include<iostream>
2   using namespace std;
3   class Rectangulo{
4       int x, y;
5       public:
6           void tomar_valores(int,int);
7           int area() {return (x*y);}
8   };
9
10  void Rectangulo::tomar_valores (int a, int b){
11      x=a;
12      y=b;
13  }
14  int main(){
15      Rectangulo recta, rectab;
16      recta.tomar_valores (3,4);
17      rectab.tomar_valores (5,6);
18      cout<<"area del rectangulo: "<<recta.area()<<endl;
19      cout<<"area del rectangulo b: "<<rectab.area()<<endl;
20      return 0;
21  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
area del rectangulo: 12
area del rectangulo b: 30
```

En éste caso en concreto, la clase (tipo de objetos) llamada es *Rectangulo*, en la cual hay dos objetos: *recta* y *rectab*, cada con sus propias variables y sus propias funciones.

Como se comprueba en la ejecución del programa, la llamada *recta.area()* obtiene un resultado distinto a *rectab.area()*. Esto es debido a que cada objeto de la clase *Rectangulo* tienen sus propias variables *x* e *y*, por lo que, cada objeto, tienen sus propias funciones *tomar_valores* y *área()*, usadas cada uno con sus variables.

Éste es un concepto básico de la *programación orientada al objeto*: Los datos y las funciones son ambos miembros de un objeto. No es necesaria la declaración de una gran cantidad de variables globales, que deban ser pasadas como parámetros de una función a otra, sino que pueden ser tomados como objetos, cada uno con sus propios datos declarados como miembros. Por ejemplo, en el caso visto arriba, no ha sido necesario declarar ningún parámetro en ninguna de las llamadas de *recta.area()* o *rectab.area()*. Esas funciones miembro utilizan directamente los datos de los miembros de sus respectivos objetos *recta* y *rectab*.

Constructores y destructores

Los objetos necesitan, generalmente, inicializar variables o asignar memoria dinámica durante su proceso de inicialización y para evitar retornar valores inesperados durante su ejecución. En el ejemplo del apartado anterior, en caso de haber llamado al miembro *area()* antes de llamar a la función *tomar_valores()* probablemente se hubieran tenido resultados inesperados ya que los miembros *x* e *y* no le habrían sido asignado valores.

Para evitar éstas situaciones, las clases incluyen una función llamada *constructor*, la cual es llamada cada vez que se crea un objeto de su clase. Dicha función constructora, de mismo nombre que la clase, no podrá devolver ningún tipo de valor, ni tan siquiera *void*.

Volviendo al ejemplo visto en el apartado anterior, se va a implementar un constructor en *Rectángulo*:

```
1   #include<iostream>
2   using namespace std;
3   class Rectangulo{
4       int base, altura;
5       public:
6           Rectangulo(int,int);
7           int area() {return (base*altura);}
8   };
9
10  Rectangulo::Rectangulo (int a, int b){
11      base=a;
12      altura=b;
13  }
14  int main(){
15      Rectangulo recta (3,4);
16      Rectangulo rectab(5,6);
17      cout<<"area del rectangulo: "<<recta.area()<<endl;
18      cout<<"area del rectangulo b: "<<rectab.area()<<endl;
19      return 0;
20  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
area del rectangulo: 12
area del rectangulo b: 30
```

Tal y como puede observarse, el resultado es el mismo que para los ejemplo anteriores, pero la función miembro *tomar_valores()* ha sido eliminada, sustituyéndose por un constructor que realice la misma opción: inicializa los valores *base* y *altura* con los parámetros asignados.

En la línea 15 y 16 se observa que, en el momento de crear los objetos de la clase (*recta* y *rectab*), se le asignan los valores con los que inicializarse. Se introduce por tanto una característica importante de los constructores, y es que no pueden ser llamados como funciones miembro regulares, sino que podrán ser ejecutadas sólo cuando se crea un nuevo objeto de la clase.

En el ejemplo también se confirma lo dicho anteriormente: ni la declaración prototipo del constructor (línea 6) como su definición (líneas 10 a 12) incluyen valores de retorno, ni siquiera *void*.

La función de clase *desructor* desarrolla la función contraria. Es automáticamente llamada cuando un objeto es destruido, por ejemplo, cuando su existencia se hace innecesaria (por

ejemplo, cuando se define un objeto local para usar con una función, y la función termina), o porque se trata de un objeto asignado dinámicamente que se libera mediante el operador *delete*.

El destructor debe tener el mismo nombre que la clase, pero precedido del signo *tilde* (~), no retornando ningún valor.

El uso del destructor es especialmente útil cuando a un objeto se le asigna memoria dinámica durante su existencia, y se requiere liberar dicha memoria asignada al objeto.

```
1   #include<iostream>
2   using namespace std;
3   class Rectangulo{
4       int *base, *altura;
5       public:
6           Rectangulo(int,int);
7           ~Rectangulo();
8           int area() {return (*base**altura);}
9   };
10  Rectangulo::Rectangulo(int a, int b){
11      base=new int;
12      altura=new int;
13      *base=a;
14      *altura=b;
15  }
16  Rectangulo::~Rectangulo(){
17      delete base;
18      delete altura;
19  }
20  int main(){
21      Rectangulo recta (3,4), rectab(5,6);
22      cout<<"area del rectangulo: "<<recta.area()<<endl;
23      cout<<"area del rectangulo b: "<<rectab.area()<<endl;
24      return 0;
25  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
area del rectangulo: 12
area del rectangulo b: 30
```

Sobrecarga de constructores

Al igual que cualquier función, un constructor puede ser sobrecargado si a más de una función, con el mismo nombre, se le asignan diferentes tipos o números a sus parámetros. Volviendo atrás, para las funciones sobrecargadas el compilador llamaría a los parámetros que coincidan con los argumentos usados en la llamada de la función. En el caso de los constructores, los cuales son llamados en el momento que se crea un objeto, el ejecutado será mismo que coincida con los argumentos pasados en la declaración del objeto:

```
1   #include<iostream>
2   using namespace std;
3   class Rectangulo{
4       int base, altura;
5       public:
6           Rectangulo ();
7           Rectangulo(int,int);
8           int area(void) {return (base*altura);}
9   };
10  Rectangulo::Rectangulo(){
11      base=5;
12      altura=6;
13  }
14  Rectangulo::Rectangulo(int a, int b){
15      base=a;
16      altura=b;
17  }
18  int main(){
19      Rectangulo recta (3,4);
20      Rectangulo rectab;
21      cout<<"area del rectangulo: "<<recta.area()<<endl;
22      cout<<"area del rectangulo b: "<<rectab.area()<<endl;
23      return 0;
24  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
area del rectangulo: 12
area del rectangulo b: 30
```

En éste caso, *rectab* ha sido declarada sin argumentos, de modo que ha sido inicializada con el constructor sin parámetros, por lo que se inicializa con los valores *base=5* y *altura=6* de las líneas 11 y 12.

En cuanto a sintaxis, es importante señalar que la declaración del objeto, el cual va a ser llamado con los valores por defecto del constructor (línea 20, *rectab*), no incluye paréntesis en su declaración:

```
Rectangulo rectab;      //Correcto
Rectangulo rectab();    //Incorrecto
```

Constructor por defecto

Si no se quiere declarar un constructor durante la definición de una clase, el compilador asume que la clase tendrá un constructor por defecto sin argumentos. De modo que, tras la declaración de una clase como la siguiente:

```
class Ejemplo{
    public
    int a, b, c;
    void producto(int n, int m) {a=n; b=m; c=a*b};
};
```

El compilador asume que *Ejemplo* tiene un constructor por defecto, por lo que se podrán declarar objetos de esa clase de forma simple declarándolo sin argumentos:

```
Ejemplo caso1
```

Por otro lado, en el momento en el que se declara un constructor propio de la clase, el compilador deja de tener un constructor por defecto. De modo que habrá que declarar todos los objetos de la clase de acuerdo con el prototipo del constructor definido para la clase:

```
class Ejemplo{
    public
    int a, b, c;
    Ejemplo (int n, int m) {a=n; b=m;};
    void producto() {c=a*b};
};
```

En el ejemplo se ha declarado un constructor que toma dos parámetros de tipo *int*. Por lo tanto una declaración de objeto correcta podría ser:

```
Ejemplo caso1 (2,3);
```

Pero:

```
Ejemplo caso1
```

Deja de ser válida para éste caso, ya que ha sido declarado la clase para un constructor explícito, el cual ha sustituido el constructor que se tiene por defecto.

Por otro lado, el compilador no sólo crea un constructor por defecto, en el caso de no ser especificado, sino también crea tres funciones miembro de forma implícita en el caso de no ser declaradas específicamente. Éstas son las funciones *copy constructor (constructor copia), copy assignment operator (operador copia asignada),* y un destructor por defecto.

El constructor copia y el operador copia asignada copian todos los datos contenidos en otro objeto a los datos de los miembros del objeto actual. Para *Ejemplo*, el constructor copia declarado por el compilador podría ser similar a:

```
Ejemplo::Ejemplo (const Ejemplo& rv){
    a=rv.a; b=rv.b; c=rv.c;
}
```

Por lo tanto, las siguientes declaraciones de objetos serán correctas:

```
Ejemplo caso1 (2.3);
Ejemplo caso2 (caso1)    //copia del constructor (copia de caso1)
```

Punteros a las clases

Será perfectamente válida la creación de punteros que apunten a clases. Se deberá por tanto considerar, una vez declarado un tipo de clase válido, el nombre de clase como el tipo de puntero. Por ejemplo:

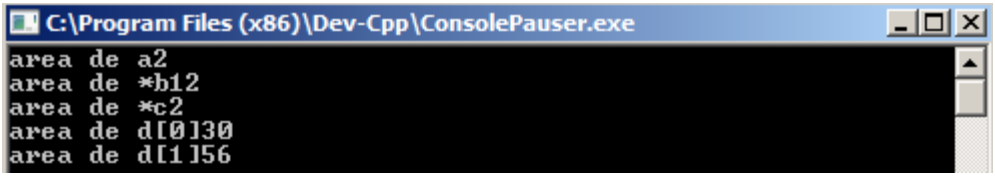
```
Rectangulo * precta
```

Es un puntero de un objeto e la clase *Rectangulo*.

Tal y como pasa en las estructuras de datos, con el objetivo de referirnos directamente a un miembro del objeto apuntado por un apuntador, se puede usar el operado flecha (->) de indirección. A continuación se muestran las distintas combinaciones posibles:

```
#include<iostream>
using namespace std;
class Rectangulo{
    int base, altura;
    public:
        void tomar_valores (int,int);
        int area (void) {return (base*altura);}
};
void Rectangulo::tomar_valores(int a, int b){
    base=a;
    altura=b;
}
int main(){
    Rectangulo a, *b, *c;
    Rectangulo *d= new Rectangulo[2];
    b=new Rectangulo;
    c=&a;
    a.tomar_valores (1,2);
    b->tomar_valores (3,4);
    d->tomar_valores (5,6);
    d[1].tomar_valores (7,8);
    cout<<"area de a"<<a.area()<<endl;
    cout<<"area de *b"<<b->area()<<endl;
    cout<<"area de *c"<<c->area()<<endl;
    cout<<"area de d[0]"<<d[0].area()<<endl;
    cout<<"area de d[1]"<<d[1].area()<<endl;
    delete [] d;
    delete b;
    return 0;
}
```

Cuyos resultados tras la ejecución son:



Como resumen, a continuación se muestran los operadores mostrados en el ejemplo anterior, para una mayor comprensión:

Expresión	Interpretación
*x	apuntado por x
&x	dirección de x
x.y	miembro y del objeto x
x->y	miembro y del objeto apuntado por x

(*x).y	miembro y del objeto apuntado por x (equivalente al anterior)
x[0]	primer objeto apuntado por x
x[1]	segundo objeto apuntado por x
x[n]	(n+1) objeto apuntado por x

Clases definidas por una estructura y unión

Las clases pueden ser definidas no sólo por la palabra clave *class*, sino también con las palabras *struct* y *union*.

Los conceptos de clase y estructura de datos son muy similares para ambas palabras clave (*struct* y *unión*), y pueden ser usadas en C++ para declarar clases (por ejemplo, *structS* puede también poseer funciones miembro en C++, no solo miembros dato). La única diferencia entre ambos es que los miembros de las clases declaradas con la palabra clave *struct* son de acceso público por defecto, mientras que para las clases era privado por defecto. Para todos los demás propósitos, las palabras clave son equivalentes.

El concepto de uniones es diferente con respecto al de las clases declaradas con *struct* y *class*, ya que las uniones sólo almacenan un miembro dato cada vez, aunque al tratarse de un tipo de clase, podrán almacenar funciones como miembros. Por defecto, y al igual que *struct*, el acceso es público.

Operadores sobrecargados

C++ incorpora la opción de usar operadores estándar con las clases como complemento de los tipos fundamentales. Por ejemplo:

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

Se trata, obviamente, de un código válido en C++, ya que todas las variables de la suma son de tipo fundamental. Sin embargo, no es tan obvio que se pueda realizar la siguiente operación, tal y como está escrita:

```
struct {
    string product;
    float precio;
};a,b,c;
a=b+c;
```

De hecho, éste ejemplo puede crear un error al compilarlo, ya que no se ha definido el comportamiento de la clase con respecto a la operación de suma. Sin embargo, gracias a la capacidad de C++ para sobrecargar operadores, se pueden diseñar clases que podrán realizar operaciones usando operadores estándar. A continuación se muestra una lista de todos los operadores que pueden ser sobrecargados:

Operadores sobrecargados																					
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>	<<=	>>=	==	!=	<=	>=	++	--	%

& ^ ! | ~ &= ^= |= && || %= [] () , ->* -> new delete new[] delete[]

Para sobrecargar un operador con el objetivo de usarlo con las clases es necesario declarar *funciones operador*, las cuales actúan como funciones regulares cuyos nombres son la palabra clave *operator* seguido del símbolo del operador. El formato es:

```
tipo operator simbolo (parámetros) {/*...*/}
```

A continuación se muestra un ejemplo en el que se sobrecarga el operador suma (+). Se va a crear una clase en la que se almacenará vectores bidimensionales para luego sumar dos de ellos; *a(3,1)* y *b(1,2)*, por lo que se obtendrá el vector (4,3):

```
1  #include<iostream>
2  using namespace std;
3  class Vector{
4      public:
5          int x,y;
6          Vector (){};
7          Vector (int,int);
8          Vector operator + (Vector);
9  };
10 Vector::Vector (int a, int b){
11     x=a;
12     y=b;
13 }
14 Vector Vector::operator+ (Vector parametro) {
15     Vector temp;
16     temp.x=x+parametro.x;
17     temp.y=y+parametro.y;
18     return (temp);
19 }
20 int main(){
21     Vector a (3,1);
22     Vector b (1,2);
23     Vector c;
24     c=a+b;
25     cout<<"("<<c.x<<","<<c.y<<")";
26     return 0;
27 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
(4,3)
```

A priori, el ejemplo puede ser confuso debido a cantidad de veces que aparece el identificador *Vector*: algunas de estas apariciones se refieren al nombre de clase *Vector*, y en otros casos se trata de funciones con el mismo nombre (constructores que deben tener el mismo nombre de la clase). Por tanto, no se debe confundir:

```
Vector (int, int);       //Funcion de nombre Vector(constructor)
Vector operator+ (b);    //Funcion que devuelve a Vector
```

La función *operator+* de la clase *Vector* se encarga de sobrecargar el operador suma. Ésta función puede ser llamada de forma implícita usando el operador, o explícitamente usando el nombre de la función, siendo ambas expresiones equivalentes:

```
c = a + b;
c = a.operator+ (b);
```

Es importante resaltar que se ha incluido un constructor vacío (sin parámetros), el cual ha sido definido con un bloque vacío:

```
Vector () {  };
```

Esto es necesario debido a que ya ha sido declarado explícitamente otro constructor:

```
Vector (int, int);
```

Cuando se declara un constructor explícitamente, con cualquier número de parámetros, el constructor por defecto sin parámetros que el compilador declara automáticamente deja de existir, por lo que será necesario que el usuario declare dicho constructor vacío para construir objetos sin ningún parámetro. Por otro lado, la declaración:

```
Vector=c;
```

Incluido en *main()*, podría no haber sido válido.

Sin embargo, la implementación de un bloque vacío, es una mala implementación de un constructor, ya que no se utiliza la mínima funcionalidad que generalmente se espera de un constructor, que no es otra cosa que la inicialización de todas las variables miembro de la clase. En éste caso, el constructor deja las variables *x* e *y* sin definir. Por otro lado, una definición más aconsejable podría haber sido similar a:

```
Vector () {x=0; y=0; };
```

Aunque no ha sido incluida para tratar de simplificar el código al máximo.

Al igual que una clase incluye un constructor por defecto y un constructor copia sin haber sido declarados, también incluye una definición por defecto para el operador asignación (=) con la propia clase como parámetro. La finalidad por la que es definida por defecto es la de copiar el contenido de información de los miembros del objeto pasados como argumentos (situados a la derecha del signo), al lado izquierdo del signo:

```
Vector d (2,3);
Vector e;
e=d;                    //copia mediane el operador asignacion
```

La función copia del operador asignación es la única función operador miembro implementada por defecto. Por supuesto, ésta puede ser redefinida por el usuario para cualquier ora finalidad, como por ejemplo, copiar sólo ciertos miembros de la clase o ejecutar distintos procedimientos de inicialización.

La sobrecarga de operadores no fuerza a la operación a actuar con su significado matemático, aunque es recomendable. Por ejemplo, el código puede no ser intuitivo se usa *operator+* para

restar dos clases, o el *operator*== para llenar de ceros una clase, aunque son casos perfectamente posibles.

Aunque el prototipo de una función *operator*+ puede ser obvio, ya que toma el valor que está a la derecha del operador como parámetro para la operación de la función miembro del objeto situado a su izquierda, otros operadores pueden no ser tan obvios. A continuación se muestra una tabla en la que se muestran las diferentes funciones operador, viendo cómo han de ser declaradas (se pondrá el símbolo @ en lugar del operador en cada caso):

Expresión	Operador	Función miembro	Función global
@a	+ - * & ! ~ ++ --	A::operator@()	operator@A
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c,...)	()	A::operator() (B,C...)	-
a->x	->	A::operator->()	-

Donde *a* es un objeto de la clase *A*, *b* es un objeto de la clase *B*, y *c* es un objeto de la clase *C*.

Tal y como se ve en el cuadro, se tienen dos vías distintas para sobrecargar un operador: como función miembro o como una función global. Su uso es indistinto, aunque es importante recordar que las funciones globales no podrán tener acceso a los miembros privados o protegidos de una clase, a menos que la función global tenga es estatus de "amiga" (característica que se verá más adelante)

La palabra clave *this*

La palabra clave *this* representa un puntero al objeto cuya función miembro está siendo ejecutada. Se trata de un puntero al propio objeto.

Uno de sus usos es el de chequear si un parámetro pasado a la función miembro es el propio objeto. Por ejemplo:

```
1   #include<iostream>
2   using namespace std;
3   class Identificador{
4       public:
5           int soyyo (Identificador& param);
6   };
7   int Identificador::soyyo (Identificador& param)
8   {
9       if(&param==this)return true;
10      else return false;
11  }
12  int main(){
13      Identificador a;
14      Identificador* b =&a;
15      if (b->soyyo(a))
16      cout<<"si, &a es b";
17      return 0;
18  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
si, &a es b
```

También es común el uso de una función miembro *operator*= que devuelva objetos por referencia (evitando el uso temporal de objetos). Siguiendo con los ejemplos de vectores vistos anteriormente, se podría escribir la función *operator*= como:

```
Vector vector==operatorP(const Vector param)
{
    x=param.x;
    y=param.y;
    return *this;
}
```

De hecho, la función es muy similar al código que el compilador genera implícitamente para la clase si no se incluye la función miembro *operator*= para copiar objetos a la clase.

Miembros estáticos

Una clase puede contener miembros estáticos (palabra clave *static*), ya sean datos o funciones.

Los miembros dato estáticos de una clase son conocidos como variables de clase, ya que existe un único valor para todos los objetos de la misma clase. Su contenido no difiere de un objeto a otro dentro de la misma clase.

Por ejemplo, se podría usar una variable dentro de una clase como contador del número de objetos de esa clase que han sido asignados, como el siguiente ejemplo:

```
1   #include<iostream>
2   using namespace std;
3   class Identificador{
4       public:
5           static int n;
6           Identificador () {n++;};
7           ~Identificador () {n--;};
8   };
9   int Identificador::n=0;
10  int main(){
11      Identificador a;
12      Identificador b[5];
13      Identificador*c=new Identificador;
14      cout<<Identificador::n<<endl;
15      return 0;
16  }
```

De hecho, los miembros estáticos tienen las mismas propiedades que las variables globales pero con influencia dentro de las clases. Por ésta razón, para evitar ser declaradas varias veces, sólo se podrá incluir el prototipo (su declaración, línea 6) en la declaración de la clase pero no su definición (su inicialización). Con el objetivo de inicializar un miembro dato estático, se deberá incluir su definición formal fuera de la clase, con alcance global, como en el ejemplo anterior (línea 9):

```
9   int Identificador::n=0;
```

Ya que se trata de un único valor variable para todos los objetos de la misma clase, puede ser referido como miembro de cualquier objeto de la clase o bien directamente a través del nombre de la clase (opción solo válida para miembros estáticos):

```
cout<<a.n;
cout<<Identificador::n;
```

Éstas dos llamadas se refieren a la misma variable: el valor estático *n* de la clase *Identificador* compartidos por todos los objetos de ésta clase.

Al igual que se introduce un dato estático en una clase, también es posible introducir una función miembro estática. Ésta representa lo siguiente: se tienen funciones globales que son llamadas como si fueran objetos miembro de una clase dada. Éstas sólo pueden hacer referencia a datos estáticos, y no a datos no-estáticos de la clase, ni tampoco podrán ser usadas con la palabra clave *this*, debido a que hacen referencia a objetos punteros y en esas funciones, de hecho, no hay miembros de los miembros objeto sino que son miembros directos de la clase.

Amigos y herencias

Introducción

Tal y como se analizó en el apartado anterior, la declaración de una función miembro ordinaria está caracterizada por:

-La función tiene acceso a las zonas privadas de la clase,
-la función está dentro del alcance de la clase, y
-la función puede ser llamada por un objeto (a través del puntero *this*).

Bien, en éste punto se introduce el término *amigo*, que no es otra cosa que otorgar los derechos de una función miembro a una función externa a la clase.

Si se requiere declarar una función externa como amigo de una clase, para que ésta tenga acceso a los miembros privados y protegidos de una clase, se tendrá que declarar un prototipo de dicha función externa a la clase, precedida de la palabra clave *friend*:

```
1   #include <iostream>
2   using namespace std;
3   class Rectangulo {
4       int base, altura;
5       public:
6           void tomar_valores (int, int);
7           int area () {return (base * altura);}
8           friend Rectangulo duplicar (Rectangulo);
9   };
10  void Rectangulo::tomar_valores (int a, int b) {
11      base = a;
12      altura = b;
13  }
14  Rectangulo duplicar (Rectangulo rectaparam)
15  {
16      Rectangulo rectares;
17      rectares.base = rectaparam.base*2;
18      rectares.altura = rectaparam.altura*2;
19      return (rectares);
20  }
21  int main () {
22      Rectangulo recta, rectab;
23      recta.tomar_valores (2,3);
24      rectab = duplicar (recta);
25      cout << rectab.area();
26      return 0;
27  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
24
```

La función *duplicar* es una función amiga de *Rectangulo*. De ésta manera, a través de *Rectangulo duplicar* se ha podido acceder a los miembros de la clase *Rectangulo*, en éste caso, *base* y *altura*, los cuales son, por defecto, privados. Si se analiza el código, ni en la declaración de *duplicar()* ni más tarde en *main()* ha sido considerado *duplicar* un miembro de la clase de *Rectangulo*. Simplemente, se ha accedido a sus miembros privados y protegidos sin ser un miembro.

Las funciones amigas pueden servir, por ejemplo, para permitir operaciones entre dos clases diferentes. Generalmente, el uso de funciones amigas está fuera de la metodología de programación orientada a objetos, debido a que es preferible es uso de miembro de la misma clase para realizar operaciones con los mismos. En el caso del ejemplo anterior, se podría haber acortado al integrar *duplicar* en la clase *Rectangulo*.

Clases amigas

Al igual que con las funciones amigas, la declaración de una clase como amiga de otra, permita a la primera el acceso a sus miembros protegidos y privados:

```
1   #include<iostream>
2   using namespace std;
3   class Cuadrado;
4   class Rectangulo{
5       int base, altura;
6       public:
7           int area(){return (base*altura);}
8           void convertir (Cuadrado a);
9   };
10  class Cuadrado{
11      private:
12          int lado;
13      public:
14          void tomar_lado(int a) {lado=a;}
15          friend class Rectangulo;
16  };
17  void Rectangulo::convertir (Cuadrado a){
18      base=a.lado;
19      altura=a.lado;
20  }
21  int main(){
22      Cuadrado sqr;
23      Rectangulo recta;
24      sqr.tomar_lado(4);
25      recta.convertir(sqr);
26      cout<<recta.area();
27      return 0;
28  }
```

En el ejemplo, ha sido declarado *Rectangulo* como amigo de la clase *Cuadrado* de modo que las funciones miembro de *Rectangulo* podrán tener acceso a los miembros protegidos de *Cuadrado*.

AL principio del ejemplo, en la línea 3 aparece la declaración de la clase *Cuadrado* vacía. Esto es necesario porque la declaración de la clase *Rectangulo* se refiere a *Cuadrado* (como parámetro de *void::convertir(),* en la línea 8). La definición de *Cuadrado* se incluye más tarde, por lo que la definición vacía es necesaria ya que dentro de la clase *Rectangulo* no se vería su definición, al estar a continuación.

Es importante remarcar el que una clase sea amiga de otra, no implica que ocurra lo mismo en la otra dirección: no es recíproco. En éste caso, *Rectangulo* es una clase amiga de *Cuadrado*, por lo que tiene acceso a sus miembros, pero *Cuadrado* no es amigo de *Rectangulo,* por lo que *Cuadrado* no tiene acceso a los miembros privados de *Rectangulo*.

Otra propiedad entre clases amigas es que no es transitiva: El amigo de un amigo no se considera amigo a menos que sea explícitamente especificado.

Herencia entre clases

Una propiedad clave en las clases de C++ es la herencia. Ésta propiedad permite crear clases a partir de otras clases, lo que las convierten automáticamente en clases "padres" de otras. Por ejemplo, Se supondrá que se quiere declarar una serie de clases que describan polígonos como *Rectángulo* o *Triangulo*. Éstos tienen propiedades en común, como por ejemplo, que las áreas de ambos pueden ser definidas conociendo su *base* y *altura*.

Por ejemplo, ése caso se podría representar como, a partir de una clase (por ejemplo, *Poligono*), derivan las otras dos, heredando propiedades comunes:

La clase *Poligono* podría contener miembros en común con los dos tipos de polígonos. En éste caso, *base* y *altura*. Por tanto, se pueden considerar *Rectangulo* y *Triangulo* como clases derivadas de *Poligono*, pero con propiedades que las distinguen la una de la otra.

Las clases que derivan de otras, tienen acceso a todos los miembros de la clase base. Esto significa que si una clase base incluye un miembro *A* y después derivamos en otra clase con un miembro llamado *B*, la clase derivada podrá contener los miembros *A* y *B*.

Para derivar una clase a partir de otra, se usarán dos puntos (:) en la declaración de la clase derivada usando el formato siguiente:

```
class nombre_clase_derivada: public nombre_clase_base
{/*...*/}
```

El especificador de acceso público, *public*, puede ser reemplazado por cualquiera de los especificadores de acceso: *protected* o *private*. Éste especificador de acceso describe el nivel de acceso mínimo para los miembros inherentes a la clase base.

```
1   #include<iostream>
2   using namespace std;
3   class Poligono{
4       protected:
5           int base, altura;
6           public:
7               void tomar_valores (int a, int b)
8               {base=a; altura=b;}
9   };
10  class Rectangulo: public Poligono{
11      public:
12          int area(){
13              return (base*altura);
14          }
15  };
16  class Triangulo: public Poligono {
17      public:
18          int area()
19          {return (base*altura/2);
20          }
21  };
22  int main(){
23      Rectangulo rect;
24      Triangulo trgl;
25      rect.tomar_valores(4,5);
26      trgl.tomar_valores (4,5);
27      cout<<rect.area()<<endl;
28      cout<<trgl.area();
29      return 0;
30  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
20
10
--------------------------------
```

Los objetos de las clases *Rectangulo* y *Triangulo* contienen miembros heredados de *Poligono*. Éstos son: *base*, *altura* y *tomar_valores()*.

El especificador de acceso *protected* es similar a *private*. La única diferencia es la propiedad hereditaria. Cuando una clase recibe herencia de otra, los miembros de la clase derivada pueden acceder a los miembros protegidos (*protected*) heredados de la clase base, pero no de los miembros privados (*private*).

Como se pretende hacer accesible los miembros *base* y *altura* a las clases *Rectangulo* y *Triangulo* desde la clase base *Poligono*, se deberá pues declarar los miembros como *protected*.

Como resumen, se puede organizar los diferentes tipos de acceso en base a la forma a la que se accede a ellos como:

Acceso	*public*	*protected*	*private*
miembros de la misma clase	Si	Si	Si
miembros de clases derivadas	Si	Si	No
no miembros	Si	No	No

En el cuadro anterior, cuando se refiere a *no miembros*, se está refiriendo al acceso fuera de cualquier clase, como puede ser una función (*main()*) u otra clase sin ninguna relación.

En el ejemplo visto arriba, los miembros heredados por *Rectangulo* y *Triangulo* tienen los mismos permisos de acceso que los que se tiene para la clase base *Poligono*:

```
Poligono::base                  //acceso protegido
Rectangulo::base                //acceso protegido

Poligono::tomar_valores()       //acceso público
Rectangulo::tomar_valores()     //acceso público
```

Esto es porque se ha usado la palabra clave *public* para definir la relación de herencia de cada clase derivada:

```
class Rectangulo: public Poligono {...};
```

La declaración *public* después de los dos puntos (:) denota el máximo nivel de acceso a todos los miembros heredados de la clase que le sigue (*Poligono* en este caso). Si por otro lado se especifica un nivel de acceso más restrictivo como *protected,* todos los miembros públicos de la clase base son heredados como protegidos en la clase derivada. Por último, si se especifica el nivel más restrictivo (*private*) todos los miembros de la clase base son heredados como privados.

Por ejemplo, si *hijo* es una clase derivada de *madre*, se puede definir:

```
class hijo:protected madre;
```

En éste caso, todos los miembros públicos para *madre*, serán privados para la clase que los hereda, en éste caso *hijo*. Por otro lado, ésta característica no restringe que *hijo* tenga sus propios miembros públicos. Éste nivel de restricción se limita a los miembros heredados de *madre*.

Si no se quisiese especificar explícitamente el nivel de acceso de los miembros heredados, el compilador asumirá como privados las clases declaradas con la palabra clave *class* y como miembros públicos a las clases construidas con la palabra clave *struct*.

Características no heredadas entre clases

En principio, una clase derivada recibe como herencia todos los miembros de una clase base excepto:

-Su constructor y destructor.
-Sus miembros con operador =().
-Sus amigos.

Aunque los constructores y destructores de la clase base no son inherentes a la misma, el constructor definido por defecto (es decir, el constructor sin parámetros) y su destructor son siempre llamados cuando un nuevo objeto de una clase derivada es creado o destruido.

Si la clase base no posee un constructor por defecto o se requiera sobrecargar un constructor en caso que se quiera crear un objeto derivado, se deberá especificar en cada definición del constructor de la clase derivada:

```
nombre_constr_derivado(parametros):nombre_constr_base (parametros)
{...}
```

Por ejemplo:

```
1   #include<iostream>
2   using namespace std;
3   class madre{
4       public:
5           madre(){
6               cout<<"madre:sin parametros\n";}
7           madre (int a){
8               cout<<"madre:parametro entero\n";}
9   };
10  class hijo:public madre{
11      public:
12          hijo(int a)
13              {cout<<"hijo:parametro entero\n\n";}
14  };
15  class hija:public madre{
16      public:
17          hija(int a):madre(a)
18          {cout<<"hija:parametro entero\n\n";}
19  };
20  int main(){
21      hijo Sergio (0);
22      hija Sonia (0);
23      return 0;
24  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
madre:sin parametros
hijo:parametro entero

madre:parametro entero
hija:parametro entero
```

En el ejemplo se puede comprobar la diferencia entre el constructor *madre*, que es llamado cuando un nuevo objeto *hijo* es creado y cuando se trata el objeto *hija*. La diferencia reside en la declaración de *hijo* e *hija*:

```
hijo(int a)             //llamada por defecto (no se especifica nada)
hijo (int a):madre(a)   //Constructor con especificación: llamar a...
```

Herencia múltiple

En C++ se da la posibilidad de que una clase herede miembros de dos o más clases. Ésta operación puede realizarse simplemente separando las diferentes clases base con comas durante la declaración de la clase derivada. Por ejemplo, si se tiene una clase específica para

mostrar mensajes en pantalla (*Output*) y se quiere que las clases *Rectangulo* y *Triangulo* hereden los miembros de *Output*, y a la vez los de *Poligono*, se podría declarar:

```
class Rectangulo:public Poligono,public Output;
class Triangulo:public Poligono,public Output;
```

A continuación se muestra un ejemplo de aplicación:

```
1   #include<iostream>
2   using namespace std;
3   class Poligono{
4       protected:
5           int base, altura;
6           public:
7               void tomar_valores (int a, int b)
8               {base=a; altura=b;}
9   };
10  class Output{
11      public:
12          void output (int i);
13  };
14  void Output::output (int i){cout<<i<<endl;
15  }
16  class Rectangulo: public Poligono, public Output{
17      public:
18          int area(){return (base*altura);}
19  };
20  class Triangulo: public Poligono, public Output {
21      public:
22          int area()
23          {return (base*altura/2);}
24  };
25  int main(){
26      Rectangulo rect;
27      Triangulo trgl;
28      rect.tomar_valores(4,5);
29      trgl.tomar_valores (4,5);
30      rect.output (rect.area());
31      trgl.output (trgl.area());
32      return 0;
33  }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
20
10
--------------------------------
```

Polimorfismos

Punteros a clases base

Una de las propiedades clave de las clases derivadas es que un puntero a una clase derivada es compatible (en cuanto a tipo de variable se refiere) con un puntero a la clase base. Los polimorfismos son el arte de tomar las ventajas de éste simple pero muy versátil y poderoso proceso, que dotan de la programación orientada a objetos de su gran potencial.

Para introducir el concepto, se va a procederá reescribir el programa *Poligono* de las secciones anteriores considerando la compatibilidad entre punteros:

```
1  #include<iostream>
2  using namespace std;
3  class Poligono{
4      protected:
5          int base, altura;
6          public:
7              void tomar_valores (int a , int b)
8              {base=a; altura=b;}
9  };
10 class Rectangulo:public Poligono{
11     public:
12         int area()
13         {return (base*altura);}
14 };
15 class Triangulo:public Poligono{
16     public:
17         int area()
18         {return (base*altura/2);}
19 };
20 int main(){
21     Rectangulo rect;
22     Triangulo trgl;
23     Poligono*Ppoli1=&rect;
24     Poligono*Ppoli2=&trgl;
25     Ppoli1->tomar_valores (4,5);
26     Ppoli2->tomar_valores (4,5);
27     cout<<rect.area()<<endl;
28     cout<<trgl.area();
29     return 0;
30 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
20
10
```

En el ejemplo, en la función *main*, se han creado dos punteros a los objetos *Poligono (Ppoli1* y *Ppoli2)* en las líneas 23 y 24. Cuando se asignan referencias a *rect* y *trgl* a a éstos punteros, y dado que éstos son objetos de clases derivadas de *Poligono*, ambos son operadores de asignación válidos.

La única limitación es que usando **Ppoli1* y *Ppoli2* en vez de *rect* y *trgl* es que **Ppoli1* y *Ppoli2* son de tipos de *Poligono** y por lo tanto sólo se podrán usar esos punteros para referir a miembros que *Rectángulo* y *Triangulo* hayan heredado de *Poligono*. Por ésta razón cuando se llama al miembro *area()* al final del programa se usan directamente los objetos *rect* y *trgl* en vez de los punteros **Ppoli1* y *Ppoli2*.

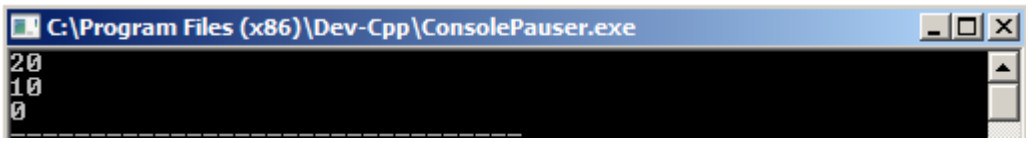
Con el objetivo de usar *area()* con los punteros de la clase *Poligono*, éste miembro debería haber sido declarado como miembro de *Poligono*, y no solo en las clases derivadas. El problema reside en que *Rectangulo* y *Triangulo* implementan diferentes versiones de *area ()*, por lo que no se puede implementar *area()* en la clase base.

Miembros virtuales

Se denomina como miembro virtual a un miembro de una clase base que es redefinido en una clase derivada. Para redefinir un miembro como miembro virtual, se utiliza la palabra clave *virtual*:

```
 3  class Poligono{
 4      protected:
 5          int base, altura;
 6          public:
 7              void tomar_valores (int a , int b)
 8              {base=a; altura=b;}
 9              virtual int area()
10              {return(0);}
11  };
12  class Rectangulo:public Poligono{
13      public:
14          int area()
15          {return (base*altura);}
16  };
17  class Triangulo:public Poligono{
18      public:
19          int area()
20          {return (base*altura/2);}
21  };
22  int main(){
23      Rectangulo rect;
24      Triangulo trgl;
25      Poligono poli;
26      Poligono*Ppoli1=&rect;
27      Poligono*Ppoli2=&trgl;
28      Poligono*Ppoli3=&poli;
29      Ppoli1->tomar_valores (4,5);
30      Ppoli2->tomar_valores (4,5);
31      Ppoli3->tomar_valores (4,5);
32      cout<<Ppoli1->area()<<endl;
33      cout<<Ppoli2->area()<<endl;
34      cout<<Ppoli3->area();
35      return 0;
```

NOTA: en la sintaxis del ejemplo faltan las dos primeras líneas *(#include<iostream>* y *using namesapce std*, las cuales, aunque no aparezcan, son necesarias. El resultado en la ejecución del programa es:



Analizando el ejemplo, se tiene que las tres clases existentes (*Poligono, Rectangulo* y *Triangulo*) poseen todas los mismos miembros: *base, altura, tomar_valores()* y *area()*.

La función miembro *area()* ha sido declarada como virtual en la clase base (linea 9) ya que después será redefinida en cada clase derivada (línea 14 y 19). Para verificar la relevancia de declarar la función miembro *area()* como virtual, si se elimina la palabra clave *virtual* de la declaración en la línea 9, y se ejecuta el programa, aparecerá que todos los resultados de ejecutar *area()* son cero. Esto es porque de hecho, todas las *area()* de cada objeto (*Rectangulo::area(), Triangulo::area()* y *Poligono::area()*, respectivamente) llama a la función *area()* a través del puntero *Poligono**.

Por lo tanto, la palabra clave *virtual* tiene el objetivo de llamar a un miembro de la clase derivada como si se tratara de un miembro de la clase base, cuando éste es convenientemente llamado por un puntero (más concretamente, cuando el tipo del puntero es un puntero a la clase base pero que apunta hacia una clase derivada, como en el ejemplo en las líneas 23 a 28).

En general, una clase que define o contiene a una función virtual, se denomina *clase polimórfica*.

Clases base abstractas

Las clases base abstractas son un concepto parecido a la clase *Poligono* visto en el ejemplo de la sección anterior. La única diferencia es que en el ejemplo previo se definió una función válida *area()* con una funcionalidad mínima, por ejemplo para el objeto *poli*. Por otro lado, en un clase base abstracta, se podría dejar la función miembro *area()* sin ninguna implementación. Esto se declara a través del código *=0* (igual a cero) en la declaración de la función:

```
class Poligono{
    protected:
        int base, altura;
        public:
            void tomar_valores (int a, int b){
                base=a, altura=b;
            }
            virtual int area ()=0;
};
```

En el ejemplo, la función miembro *area()* no deja de ser una función, aunque no haya sido declarado ninguna implementación (es decir, la parte entre corchetes { }). Éste tipo de funciones es llamada *función virtual pura*, y todas las clases que contienen, al menos, una *función virtual pura*, como *clases base abstractas*.

La principal diferencia entre una clase base abstracta y una clase polimórfica es que en una clase base abstracta al menos uno de sus miembros implementados se le podrán crear objetos.

Por otro lado, una clase sin objetos es accesible por otra vía: se podrán crear punteros a ésta, proporcionando ventajas sobre las habilidades polimórficas. Por lo tanto una declaración como:

```
Poligono poli;
```

No será válida para una clase base abstracta como la declarada antes, porque implica la inicialización de un objeto. Sim embrago, los punteros siguientes serán perfectamente válidos:

```
Poligono*Ppoli1;
Poligono*Ppoli2;
```

Son perfectamente válidos. Por otro lado, los punteros de clases base abstractas pueden ser usados para apuntar a objetos de clases derivadas. Si se aplican todos éstos conceptos al programa de la clase *Poligono* se tiene:

```
1  #include<iostream>
2  using namespace std;
3  class Poligono{
4      protected:
5          int base, altura;
6          public:
7              void tomar_valores (int a , int b)
8              {base=a; altura=b;}
9              virtual int area (void)=0;
10 };
11 class Rectangulo:public Poligono{
12     public:
13         int area()
14         {return (base*altura);}
15 };
16 class Triangulo:public Poligono{
17     public:
18         int area()
19         {return (base*altura/2);}
20 };
21 int main(){
22     Rectangulo rect;
23     Triangulo trgl;
24     Poligono*Ppoli1=&rect;
25     Poligono*Ppoli2=&trgl;
26     Ppoli1->tomar_valores (4,5);
27     Ppoli2->tomar_valores (4,5);
28     cout<<Ppoli1->area()<<endl;
29     cout<<Ppoli2->area();
30     return 0;
31 }
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
20
10
```

Revisando el programa se observa cómo se refiere a distintos objetos, de clases derivadas distintas, con el mismo puntero *Poligono**. Ésta característica es tremendamente útil. Por ejemplo, se podrá crear una función miembro de la clase base abstracta *Poligono* que pueda

mostrar en pantalla el resultado de la función *area()* aunque *Poligono* no tenga ninguna implementación de ésta función:

```
#include<iostream>
using namespace std;
class Poligono{
    protected:
        int base, altura;
        public:
            void tomar_valores (int a , int b)
            {base=a; altura=b;}
            virtual int area (void)=0;
            void mostrararea (void)
            {cout<<this->area()<<endl;}
};
class Rectangulo:public Poligono{
    public:
        int area()
        {return (base*altura);}
};
class Triangulo:public Poligono{
    public:
        int area()
        {return (base*altura/2);}
};
int main(){
    Rectangulo rect;
    Triangulo trgl;
    Poligono*Ppoli1=&rect;
    Poligono*Ppoli2=&trgl;
    Ppoli1->tomar_valores (4,5);
    Ppoli2->tomar_valores (4,5);
    Ppoli1->mostrararea();
    Ppoli2->mostrararea();
    return 0;
```

Los miembros virtuales y las clases abstractas proporcionan a C++ las características polimórficas que hacen de la programación orientada a objetos un instrumento muy útil en grandes proyectos. Por supuesto, se han mostrado usos muy simples de éstas características, pero éstas toman gran relevancia cuando son aplicadas a matrices (*arrays*) de objetos u objetos almacenados dinámicamente.

Siguiendo con el ejemplo de aplicación mostrado hasta ahora, pero aplicando el concepto de objetos almacenados dinámicamente, se tiene:

```
1  #include<iostream>
2  using namespace std;
3  class Poligono{
4      protected:
5          int base, altura;
6          public:
7              void tomar_valores (int a , int b)
8              {base=a; altura=b;}
9              virtual int area (void)=0;
10             void mostrararea (void)
11             {cout<<this->area()<<endl;}
12 };
13 class Rectangulo:public Poligono{
14     public:
15         int area()
16         {return (base*altura);}
17 };
18 class Triangulo:public Poligono{
19     public:
20         int area()
21         {return (base*altura/2);}
22 };
23 int main(){
24     Poligono*Ppoli1=new Rectangulo;
25     Poligono*Ppoli2=new Triangulo;
26     Ppoli1->tomar_valores (4,5);
27     Ppoli2->tomar_valores (4,5);
28     Ppoli1->mostrararea();
29     Ppoli2->mostrararea();
30     delete Ppoli1;
31     delete Ppoli2;
32     return 0;
33 }
```

En las líneas 24 y 25, los punteros *Ppoli* son declarados como tipo de puntero de *Poligono*, pero los objetos dinámicamente almacenados han sido declarados mediante de clases derivadas:

```
Poligono*Ppoli1=new Rectangulo;
Poligono*Ppoli2=new Triangulo;
```

5.Programación en C++ de un Plug-in de cálculo naval para Rhinoceros

Introducción

El objetivo de éste apartado no es tanto el resultado que se consigue con el plug-in, sino la comprensión y el alcance que puede tener el usuario realizando cualquier tipo de plug-in y adaptar Rhino a sus propias necesidades. Por ejemplo, en el caso de la obtención de las propiedades hidrostáticas de una geometría, puede ser muy práctico para el usuario ya que no sería necesaria la exportación de la geometría a un programa de análisis de formas navales (como *Hidromax*).

Por otro lado, la mejora y desarrollo de un plug-in, tiene la enorme ventaja de que, partiendo de dos programas como base (en éste caso Visual C++ y Rhinoceros 4.0 SR9), el usuario obtiene un programa muy específico y totalmente gratuito, ya que tanto el editor de plug-ins, como la biblioteca *openNURBS* son gratuitos y descargables desde internet desde organizaciones sin ánimo de lucro.

Otra gran ventaja, es el control de los cálculos que realiza un software. Cuando se programa el código de un plug-in (en éste caso), se controla y comprende todos los cálculos realizados por el ordenador, teniendo conocimiento (como ingenieros) de los errores de cálculo, casos en los que se puede aplicar ciertas herramientas, detectar errores de diseño, etc. Es habitual que, al comenzar a utilizar un software, no se conozca cómo éste realiza ciertos cálculos, o qué métodos numéricos utiliza, por lo que a veces se desconoce el error que se comete durante la ejecución de un programa. Éste problema se resuelve si se conoce el código del programa, acotando los errores producidos.

Por último, y no menos importante, el desarrollo de un programa adaptado a cualquier campo (como el naval) podrá ser comercializado, por lo que el beneficio no solo se obtiene del ahorro de gastos a la hora de adquirir softwares específicos, sino también de la posibilidad de venderlos.

Inicialización del desarrollo de un plug-in de Rhinoceros

Las aplicaciones de C++ en el campo de la ingeniería son muchísimas, ya no solo como procesado de datos o diseño, son también como control de procesos o interpretación de datos de entrada. Por tanto, es común la programación y manipulación de curvas y superficies NURBS a través de C++.

Por otro lado, la posibilidad de obtener una biblioteca sobre superficies NURBS es fácil y gratuita a través de desarrolladores independientes, siendo un referente *openNURBS initiative (*http://www.opennurbs.org/). Dicha iniciativa tiene el objetivo de proveer a usuarios de CAD, CAM, CAE y desarrolladores de software de herramientas que permitan la transferencia de geometrías 3-D entre aplicaciones-

En definitiva, openNURBS proporciona un conjunto de herramientas de código abierto para lectura y escritura de los modelos en formato 3DM, cuya plataforma de desarrollo de todas las funciones es Rhino.

Una de las aplicaciones más frecuentes es la creación de Plug-ins para Rhinoceros basados en lenguaje C++. Un Plug-in, desde la perspectiva de un usuario, es un módulo de software que amplía la funcionalidad de Rhino (en éste caso) añadiendo nuevos comandos, características, o capacidades. Desde el punto de vista de Rhino, se distinguen 5 tipos de Plug-ins:

-Utilidad general: Extensión de propósito general, el cual contendrá uno o más comandos.

-Archivo de importación: Extensión de Rhino cuyo propósito es el de importar datos de otros formatos de archivo en Rhino.

-Archivo de exportación: Extensión que permite exportar datos a otros formatos desde Rhino. Un único plug-in de exportación puede soportar varios tipos de archivos de exportación.

-Renderizado personalizado: Una representación personalizada, como tipos de materiales, texturas y luces con objeto de producir imágenes y gráficos de distintas calidades.

-Digitalización 3-D: Permite la interactuación con el Digitalizador 3-D de Rhino.

Requisitos necesarios para la creación de plug-ins de Rhino

Rhino dispone de herramientas que facilitan el ensamblaje de un Plug-in a través de C++. Se trata de Rhino C++ Plug-in Software Development Kit (SDK), totalmente gratuito y descargable de la página oficial de Rhino. Dicho Software facilita y orienta la creación de un Plug-in, con un formato ya predeterminado por el propio Software.

Rhino C++ SDK se ejecuta a través de Microsoft Visual Studio, siendo necesario disponer de versiones muy concretas para evitar incompatibilidades. En el caso de usar Rhinoceros 4 Service Release 9, o Rhinoceros 5 a 32 bit, se precisa de la versión Microsoft Visual Studio C++ 2005, mientras que para el uso de Rhinoceros 5 a 64 bits, se necesita de la versión Microsoft Visual C++ 2010. Ninguna de las otras versiones son compatibles, no siquiera las versiones Express de Microsoft Visual Studio.

Por otro lado, Rhino sólo facilita Rhino C++ SDK para las versiones de Rhinoceros 5 y Rhinoceros 4 SR9, por lo que si se cuenta con una versión anterior de Rhinoceros 4, se deberá actualizar a través de su página oficial con el paquete Service Release 9. Por tanto:

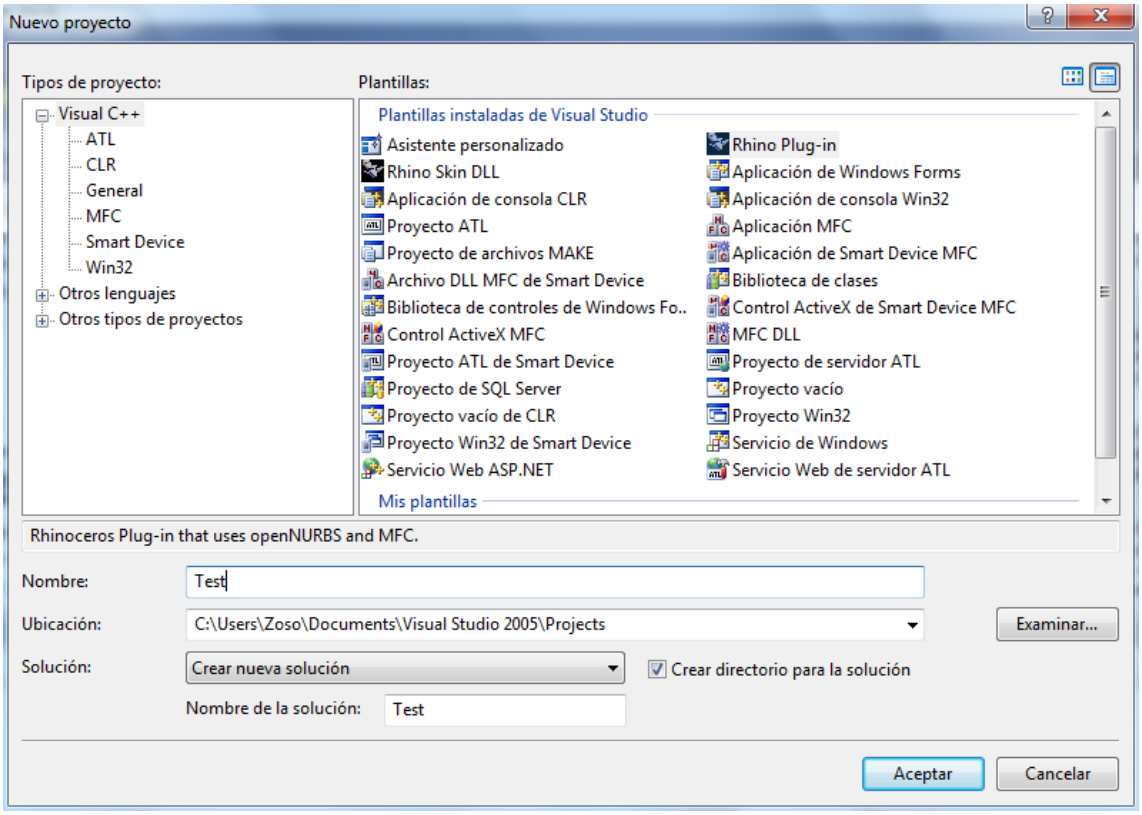
Versiones disponibles de Rhino C++ SDK	Versiones de Rhinoceros compatibles	Versiones de Microsoft Visual C++ compatibles
-Rhino 4 C++ SDK	-Rhinoceros 4 SR9 -Rhinoceros 5 32 bits	-Microsoft Visual C++ 2005
-Rhino 5 C++ SDK	-Rhinoceros 5 64 bits	-Microsoft Visual C++ 2010

Por último, en caso de utilizar Rhinoceros 4.0 RS9 con Microsoft Visual C++ 2005,

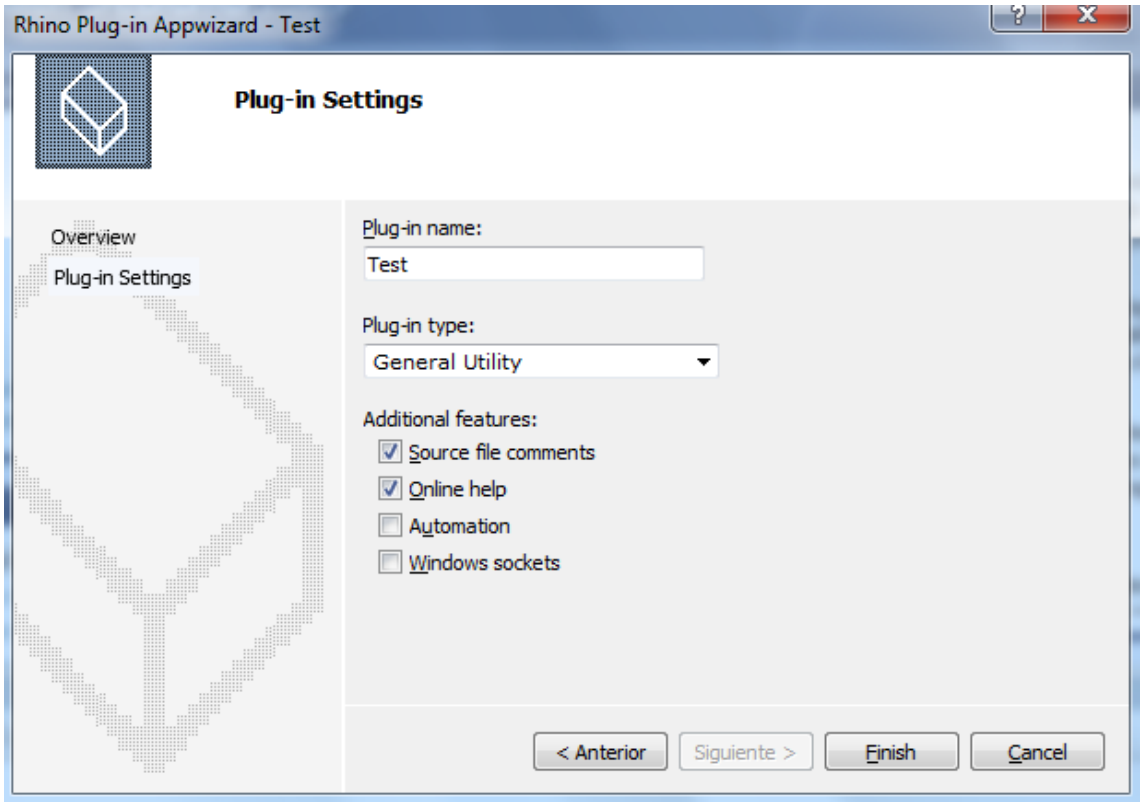
Creación de Plug-ins

Para la generación de un nuevo Pulg-in, existe una aplicación en Rhino C++ SDK el cual genera el código funcional de un Plug-in. Como tutorial, se va a usar dicha aplicación.

Para comenzar, se abre Visual C++, y se crea un nuevo proyecto, a través de *Nuevo->Proyecto* del menú *Archivo* de Visual Studio. En la ventana de nuevo proyecto, se seleccionará el tipo de proyecto que se va a realizar, seleccionando la plantilla *Rhino Plug-in* del menú de plantilla de Visual C++:



Una vez seleccionado, el propio editor de Rhino guiará el proceso de selección de las bases del Plug-in, dependiendo del tipo del Plug-in que se quiera hacer (nombre, tipo de plug-in y características adicionales). Como nombre, se usará *Test*, para probar si funciona el Plug-in.



Como características adicionales, se podrá seleccionar cualquiera de las siguientes:

-*Source File Documents*: Se seleccionará ésta opción si se quiere que el editor cree comentarios verbales sobre los archivos generados.

-*Online Help*: Ésta opción permite la posibilidad de pedir ayuda a través de internet al soporte técnico. Si es seleccionada, se dispondrá de un elemento en el menú de ayuda de Rhinoceros automáticamente.

-*Automation*: Ésta opción permite al programa creado la manipulación de objetos implementados en otros programas. Seleccionando ésta opción se expone al programa a la automatización del Plug-in por parte de otro cliente.

-*Windows sockets*: Indica que el programa creado es compatible con sockets de Windows. Dichos Sockets permiten escribir programas comunicados a través de redes TCP/IP.

Una vez seleccionados los elementos que se requieran (por ejemplo, se elegirán las opciones indicadas por defecto). Al finalizar el asistente, se crean varios archivos, visibles a través de *Ver->Explorador de soluciones*, de los cuales cabe destacar los siguientes:

-*cmdTest.cpp*; que muestra el código fuente de Rhino del comando Test.

-*Resource.h*; Declara las definiciones constantes de tipo *#Define*.

-*TEST.DEF*; Archivo de la definición de módulos de Windows.

-*Test.rc*; Archivo de recursos *script*.

-*TestApp.h*; Aplicación del archivo de cabecera clase que contiene la declaración de la clase *CTestApp.*

-*TestApp.cpp*; Aplicación que implementa el archivo de clase que contiene los miembros de *CTestApp.*

-*TestPlugIn.h*; Archivo clase de cabecera Plug-in que contiene la declaración de la clase *CTestPlugIn.*

-*TestPlugIn.cpp*; Aplicación que implementa el archivo de clase Plug-in, el cual contiene los miembros de *CTestApp.*

Por otro lado, también se comprueban las bibliotecas usadas por el editor de Plug-in y Rhinoceros, siendo éstas las librerías *openNURBS.lib* y *Rhino4.lib*. Dichas librerías son la base del programa, donde han sido declaradas todas las funciones y clases necesarias para definir, visualizar y manipular las superficies NURBS, entre otras herramientas. En el caso de openNURBS, se trata de una biblioteca descargable de licencia gratuita, base de cualquier programa que manipule NURBS.

Compilación y acceso al Plug-in

Siguiendo con el ejemplo, se va a proceder a ejecutar un plug-in sin ninguna aplicación, sólo con el objetivo de analizar el proceso de compilación y acceso en Rhinoceros.

El editor de Plug-in de Rhino crea también el archivo, *Test.vcproj*, el cual especifica todas las dependencias de archivos al compilador, accediendo a las opciones marcadas.

Antes de comenzar con el código del Plug-in, es necesaria la cumplimentación de la documentación que presenta el editor por defecto. Dicha documentación proporcionará información al usuario acerca de necesitar ayuda en el uso del Plug-in. Ésta viene presentada en el archivo *TestPlugIn.cpp*, en la que se modificarán las líneas de código con la información requerida, por ejemplo:

```
// When completed, delete the following #error directive.
#error Developer declarations block is incomplete!
RHINO_PLUG_IN_DEVELOPER_ORGANIZATION( L"UPCT" );
RHINO_PLUG_IN_DEVELOPER_ADDRESS( L"Av.Alfonso XIII Cartagena" );
RHINO_PLUG_IN_DEVELOPER_COUNTRY( L"Spain" );
RHINO_PLUG_IN_DEVELOPER_PHONE( L"123.456.7890" );
RHINO_PLUG_IN_DEVELOPER_FAX( L"123.456.7891" );
RHINO_PLUG_IN_DEVELOPER_EMAIL( L"soporte_ayuda@upct.es" );
RHINO_PLUG_IN_DEVELOPER_WEBSITE( L"http://www.upcty.com" );
RHINO_PLUG_IN_UPDATE_URL( L"http://www.upct.com/soporte_ayuda" );
```

Es importante, tal y como indica el comentario (en verde), que se elimine la directiva *#error*. Éste error se provoca sólo para que el diseñador del Plug-in no olvide cumplimentar la información, lo que quedaría indicado a la hora de compilar el Pulg-in. De modo que debe quedar de la forma (obviamente, el mensaje de error en verde, no es necesario eliminarlo):

```
// When completed, delete the following #error directive.
RHINO_PLUG_IN_DEVELOPER_ORGANIZATION( L"UPCT" );
RHINO_PLUG_IN_DEVELOPER_ADDRESS( L"Av.Alfonso XIII Cartagena" );
RHINO_PLUG_IN_DEVELOPER_COUNTRY( L"Spain" );
RHINO_PLUG_IN_DEVELOPER_PHONE( L"123.456.7890" );
RHINO_PLUG_IN_DEVELOPER_FAX( L"123.456.7891" );
RHINO_PLUG_IN_DEVELOPER_EMAIL( L"soporte_ayuda@upct.es" );
RHINO_PLUG_IN_DEVELOPER_WEBSITE( L"http://www.upcty.com" );
RHINO_PLUG_IN_UPDATE_URL( L"http://www.upct.com/soporte_ayuda" );
```

Una vez realizada ésta operación, el Plug-in puede ser finalizado y ejecutado (ya que no se ha especificado nada, el Plug-in no realizará ninguna acción). Una vez guardado, se creará en la carpeta *Debug* del proyecto, un archivo llamado *Test_d.rhp*.

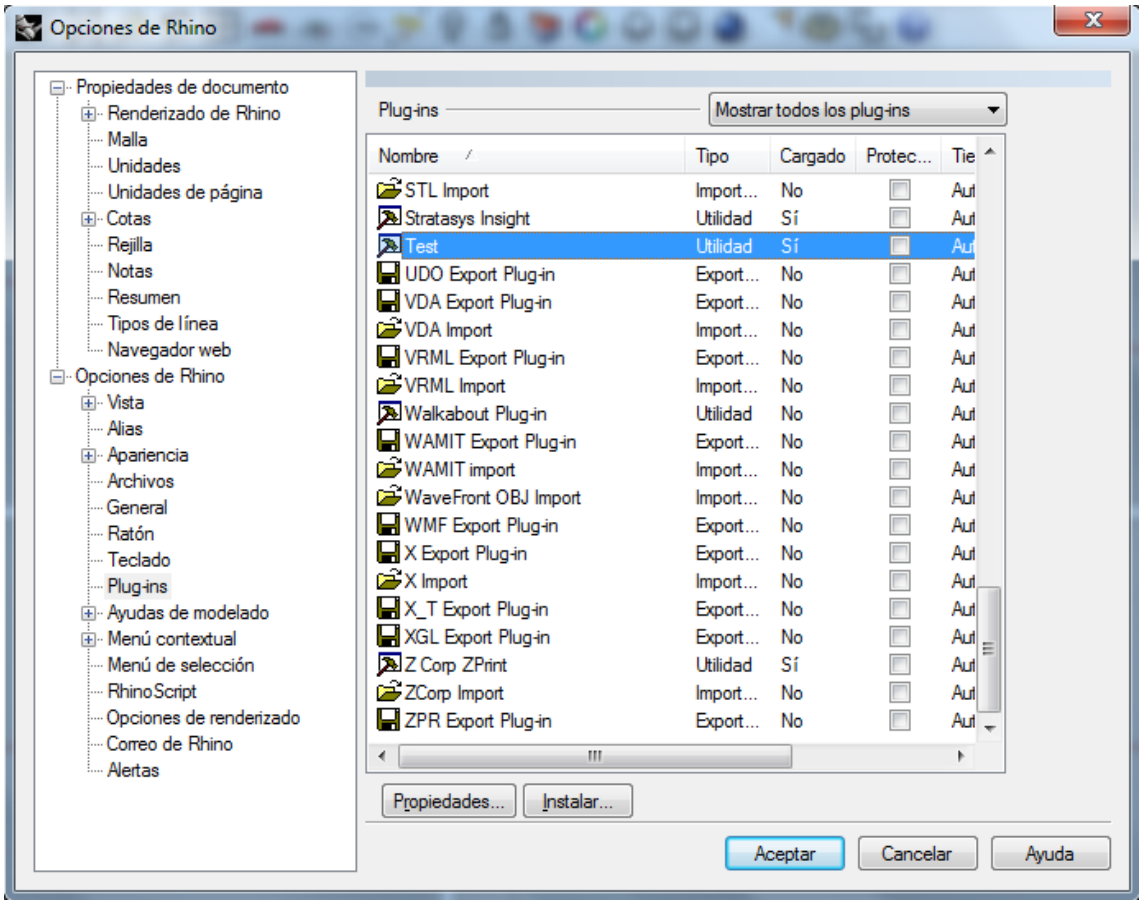
Para inicializar el plug-in en Rhino, se debe depurar en Visual C++. Las opciones de depuración (*Debug*) de Visual C++, proporcionadas por Rhino C++ SDK son las siguientes:

-*Debug*: Es la opción de depuración estándar. Ésta configuración vinculará con la depuración de las librerías MFC, Rhino, y openNURBS. Los pulg-ins construidos con ésta configuración sólo podrá ser usados en la versión depurada de Rhino (Rhino4_d.exe) instalada con Rhino SDK.

-*Pseudo-Debug*: Se trata de una configuración de lanzamiento, que también incluye información depurada (*Debug*). Ésta configuración se enlazará (que no vinculará) con las bibliotecas MFC, Rhino y openNURBS, por lo que tendrá capacidad de depuración limitada. Los Plug-ins construidos bajo ésta configuración, sólo se pueden utilizar con la versión de Rhino instalada (Rhino 4.exe).

-*Release*: Se trata de una configuración de lanzamiento regular. Ésta configuración enlaza con la bibliotecas MFC, Rhino, y openNURBS. Los Plug-ins construidos con ésta configuración sólo pueden ser usados con la versión de Rhino instalada (Rhino 4.exe).

A continuación, al depurar el plug in que se ha creado anteriormente, a través de *Release*, se podrá llamar a dicho plug-in a través del programa Rhinoceros. Si se ejecute Rhinoceros 4, y se accede a *Herramientas/Opciones/Plug-ins/Instalar….* Al abrir la ventana *Instalar…* se busca y selecciona el archivo *Test.rhp*, aparecerá en la lista de Plug-ins instalados, finalizando el proceso de implementación. Si además se cliquea sobre *Test* en la lista de Plug-ins, aparecerá n los datos de contacto descritos anteriormente:



Realización de un plug-in con los que obtener datos hidrostáticos de una carena

Introducción

Una vez se ha visto cómo inicializar el editor de plug-ins en Visual C++, se va a proceder a la realización de un código con el que obtener las propiedades hidrostáticas de una geometría dada en Rhinoceros.

La inicialización de los archivos necesarios se realizara tal y como se ha visto en el apartado anterior, solo que ésta vez se decidirá el nombre con el que se denominará el *plug-in*. Será con éste nombre, con el que también se ejecutará en Rhinoceros, una vez que esté realizado.

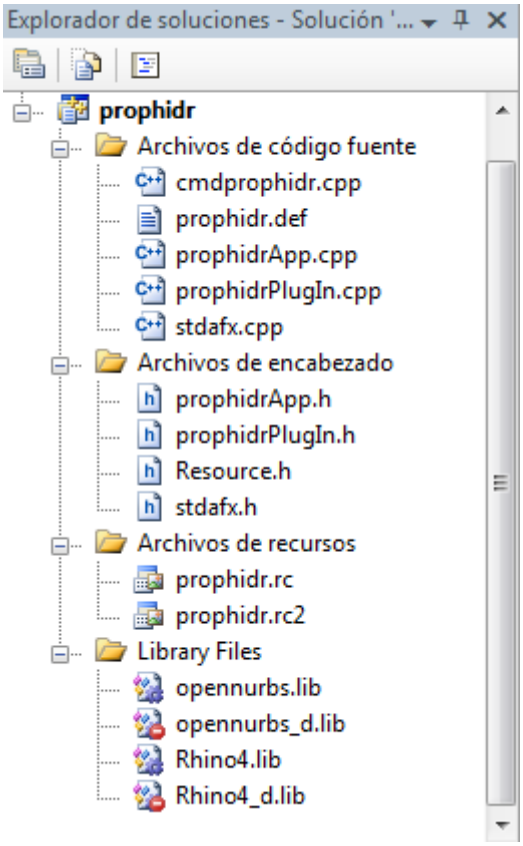
Como recursos necesarios para el desarrollo del código, sería necesario un análisis de las bibliotecas con las que se realiza el código, ya que en todo momento, para interactuar con el entorno Rhino y con las superficies NURBS, se moverá entre las bibliotecas *openNURBS* y *Rhino.*

Por otro lado, y para agilizar el proceso de adaptación y comprensión del potencial de las bibliotecas, Rhino 4.0 SDK proporciona un archivo de ayuda al que recurrir para la comprensión, aplicación e implantación de todas las clases, funciones, estructuras, tipos,

variables y macros que los componen. Dicho archivo se adjunta en el CD del proyecto con el nombre *Rhino 4.0 SDK Help*.

Descripción del código

En primer lugar, es necesario saber en qué archivo se tendrá que escribir el código que computará las órdenes establecidas en éste. Tal y como se ha visto en el apartado anterior, éste archivo es el denominado *cmdprophidr.cpp* (que muestra el código fuente de Rhino del comando *prophidr*). Éste archivo se ejecutará una vez se ordene en Rhino con la palabra clave *prophidr*.



EL propio Wizard de Rhino 4.0 SDK ha generado el encabezado necesario para la correcta ejecución, siendo descrito con comentarios. En primer lugar, se crea una clase con el identificador *Ccommandprophidr*, dando el propio programa indicaciones de cómo realizar la cabecera del archivo.

```
class CCommandprophidr : public CRhinoCommand
{
public:
    CCommandprophidr() {}
    ~CCommandprophidr() {}
    UUID CommandUUID()
    {
    static const GUID prophidrCommand_UUID =
    { 0x85EEC19C,0x2ADF,0x4988,{0x9E,0x62,0xC,0xC2,
    0xF6,0x6,0xB,0x2E}};
    return prophidrCommand_UUID;
    }
    const wchar_t* EnglishCommandName() { return L"prophidr"; }
    const wchar_t* LocalCommandName() { return L"prophidr"; }
    CRhinoCommand::result RunCommand( const CRhinoCommandContext& );
};
static class CCommandprophidr theprophidrCommand;
```

La clase *Ccommandprophidr* también viene definida con ciertas propiedades ya definidas por Rhino SDK, las cuales no conviene alterar, ya que son propiedades de la clase. Cabe destacar el comando *UUID commandUUID*, el cual crea un código asignado con el que se identificará el plug-in, y con el que se distinguirá de los demás. Por tanto, si el identificador (*id*) coincide con el de otro plug-in, puede que el uno de los dos no trabaje. Por otro lado, los comandos que le siguen servirán para adaptar el plug-in a otros idiomas y llamamiento del comando:

```
const wchar_t* EnglishCommandName() { return L"prophidr"; }
// Devueleve el comando en idioma inglés
const wchar_t* LocalCommandName() { return L"prophidr"; }
// Devuelve el nombre del comando una vez localizado
CRhinoCommand::result RunCommand( const CRhinoCommandContext& );
// Rhino llama a  RunCommand para reproducir el comando
```

Por último, se crea el objeto de la clase *prophidr*:

```
// El único objeto de la clase CCommandprophidr.
// No debe crearse nigúna otra instancia de la clase CCommandprophidr
static class CCommandprophidr theprophidrCommand;
```

Una vez se ha creado la clase y el objeto, ya se puede comenzar a trabajar sobre las acciones que hará el plug-in. En éste caso, la intención es obtener ciertas propiedades hidrodinámicas, calculadas a partir de datos obtenidos de la geometría de la carena. Aunque las posibilidades son muy grandes, se van a fijar los siguientes objetivos de cálculo:

-Propiedades obtenidas de la superficie de flotación: Área, TCI, Centro de flotación.

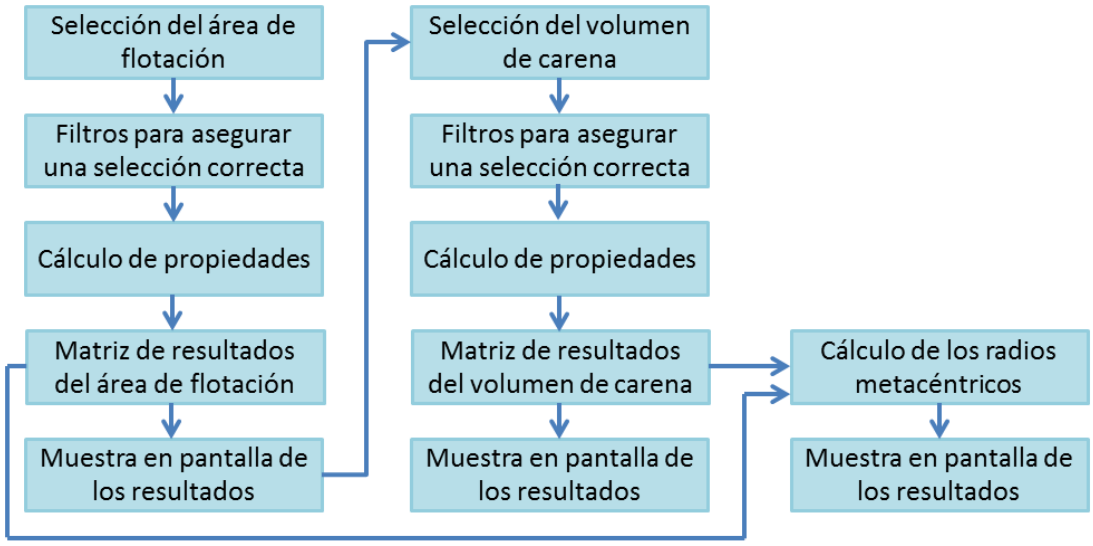
-Propiedades obtenidas del volumen de carena: volumen, desplazamiento, posición el centro de carena, y área mojada.

-Propiedades obtenidas de ambas: Radios metacéntricos transversal y longitudinal, tomando las inercias longitudinal y transversal de la superficie de flotación y el volumen de carena.

Es importante, a continuación, organizar cómo se pretende determinar las características descritas. En Rhino, es habitual que para la realización de cualquier acción, se seleccione la

geometría y después se proceda a la acción deseada. En principio, para los objetivos marcados, se podría establecer, por ejemplo, una selección de la geometría a analizar, realizar los cálculos necesarios, y obtener en la pantalla los resultados.

Por tanto, el código se puede plantear de la siguiente manera:



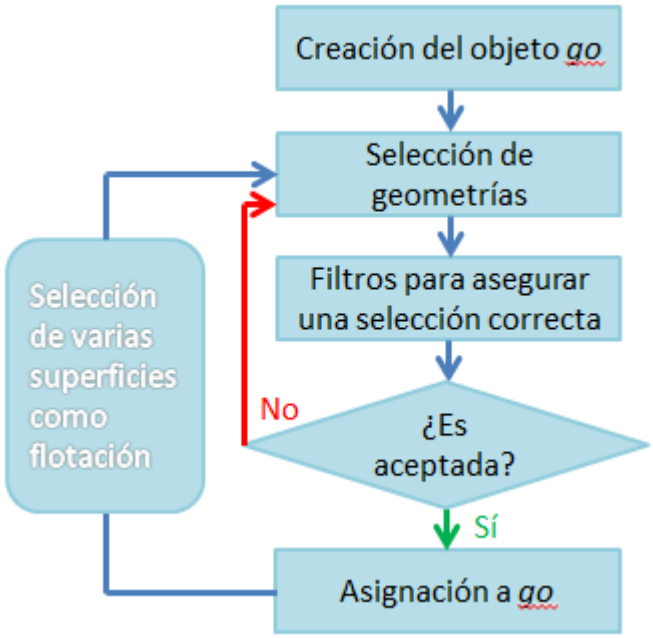
Como se ha visto anteriormente, se trabajan con dos bibliotecas: la biblioteca de Rhino, que aporta acciones propias de Rhino, y la biblioteca openNURBS, que muestra las acciones y creación de superficies y curvas NURBS. Ambas bibliotecas son muy extensas, por lo que para poder realizar un plug-in, hay que estar familiarizado con ambas. Como ayuda, también se dispone de Rhino SDK Help, donde se describen ambas bibliotecas minuciosamente.

Selección del área de flotación

Éste paso es simple: en primer lugar se procede a la creación de un objeto de la clase encargada de la selección de la geometría, para luego, en base a dicho objeto, ir aplicando los distintos miembros de la clase. La clase que se encarga de la selección de geometrías del entorno de Rhino, es de la biblioteca propia de Rhino, y se denomina *CRhinoGetObject*:

```
CRhinoGetObject go;
go.SetCommandPrompt( L"Seleccione el área de flotación" );
```

Como se ve en el código, se crea un objeto (*go*) de la clase *CRhinoGetObject*. A la vez, se le asignan los objetos y se pide que se muestre el texto indicado en rojo a través del miembro *SetCommandPrompt*. Éstas son as órdenes necesarias para la selección de uno (o varios) elementos, aunque éstos todavía no ha sido asignados al objeto *go*, ya que es necesario la aplicación de filtros de selección previos a la asignación:



Filtro de selección

Antes de indicar qué hacer con las superficies seleccionadas, es importante establecer un filtro con el que verificar que las geometrías seleccionadas en el paso anterior son correctas para el cálculo que se va a realizar. Para ello, a partir del objeto *go*, el cual mantiene a las geometrías seleccionadas, se le van aplicando distintos miembros para verificar que son válidas para los cálculos que se van a realizar.

El primer paso a realizar es la aplicación del miembro *SetGeometryFilter*, con el que establecerá que las superficies seleccionadas en *CRhinoGetObject* sean, para éste caso, sólo superficies o polisuperficies. Esto se consigue con el código siguiente:

```
go.SetGeometryFilter(
      CRhinoGetObject::surface_object |
      CRhinoGetObject::polysrf_object
      );
```

Donde *surface_object* y *polysrf_object* son los atributos aplicados a través del operador alcance (::), que indican que se ha seleccionado bien una superficie o una polisuperficie.

Para impedir que se seleccionen sub-objetos (por ejemplo, curvas pertenecientes a un grupo), se indica con el código siguiente como *false* (al ser tipo *void*, las selecciones posibles serán *true* o *false)*:

```
go.EnableSubObjectSelect( false );
```

Por otro lado, también se puede elegir si las geometrías seleccionadas pertenecen a un grupo sean seleccionadas todas a la vez o no, es decir, si se ha construido una polisuperficie como superficie de flotación, y se han agrupado en un grupo, la siguiente línea permite que se seleccione el grupo entero:

```
go.EnableGroupSelect( true );
```

También se añade la posibilidad de cancelar el proceso de selección, escapando de la selección y, bien continuar con el siguiente paso, o bien cancelar el proceso completo, a través del miembro *GetObjects()*. Éste miembro tiene varios atributos (se pueden comprobar en el archivo Rhino SDK Help), pero los declarados aquí serán en 1, para cancelar la operación de selección, y el 0, para continuar el proceso de selección:

```
go.GetObjects( 1, 0 );
```

Por último, se debe exigir que el proceso de selección haya sido un éxito, de manera que a través de un condicional, se interrumpa el proceso. Dicho análisis necesita del miembro *CommandResult()*, que evalúa los resultados que se van obteniendo de *go.GetObjects(1,0)*:

```
if( go.CommandResult() != success )
  return go.CommandResult();
```

Por lo que se exige que *go.CommandResult()* sea un éxito (que cumpla con lo establecido), a partir de la declaración *go.GetObjects(1,0)*.

A través de todo éste filtro, se van seleccionando las superficies que son aceptadas para los cálculos que vendrán a continuación. El paso siguiente será almacenar la geometría seleccionada. Para ello, se va a crear un array con tantos elementos como superficies hayan sido seleccionadas. Como en principio no se tiene por qué saber qué superficies van a formar parte del array, el tamaño vendrá establecido por un bucle.

En éste caso, se va a seleccionar un array de memoria dinámica, ya que una vez finalizada la ejecución del plug-in, no será necesaria la existencia de dicho array. La siguiente línea crea un array de elementos constantes (con la dirección de la geometría seleccionada), con el identificador *geom* (en función de los elementos *go* que se van contabilizando)

```
ON_SimpleArray<const ON_Geometry*> geom( go.ObjectCount() );
```

Para ir ensamblando el Array, se diseña un bucle *for* de la siguiente forma:

```
int i;
for( i = 0; i < go.ObjectCount(); i++ )
{
  const ON_Geometry* geo = go.Object(i).Geometry();
  if( 0 == geo )
    return failure;
  geom.Append( geo );
}
```

El funcionamiento es el siguiente: *ON_Geometry* es un constructor, el cual crea elementos denominados *geo* y les va proporcionando un identificador, para cada objeto *go* que va recibiendo. Se establece también que si no se ha seleccionado ningún objeto, se devuelva un fallo que haga finalizar la ejecución. Por último, *geom.Append(geo)* permite añadir elementos en caso de que no se hayan contabilizado elementos *geo*.

Una vez finalizado éste paso, ya se tienen las geometrías seleccionadas y almacenadas con las que se podrá comenzar su análisis.

Cálculo de propiedades

Para la evaluación de propiedades de superficies, ya sean abiertas o cerradas, la biblioteca *openNURBS* dispone de la clase *ON_MassProperties*, la cual realiza distintas operaciones sobre las superficies o volúmenes con los que se trabaja.

La forma de operar de *ON_MassProperties*, requiere de una caja donde se encierran los elementos, creada a través de *ON_BoundingBox*. Las dimensiones, por tanto, de dicha caja serán función de las direcciones de las superficies seleccionadas:

```
// Caja delimitadora de los objetos
ON_BoundingBox bbox;

for( i = 0; i < geom.Count(); i++ )
  geom[i]->GetBoundingBox( bbox, bbox.IsValid() );
```

Por tanto, se crea una caja con el identificador *bbox*, a la cual se va dando forma mediante un bucle *for* que repase todos los elementos del Array *geom*. Es *GetBoundingBox* el encargado de dar las coordenadas máximas de *bbox* en función de *geom*.

Una vez definida la caja contenedora, se obtiene las coordenadas del centro de dicha caja. Esto es sólo necesario para realizar los cálculos de momentos respecto a un punto cercano y no desde el origen, evitando números grandes que puedan aumentar la memoria duración del proceso:

```
ON_3dPoint base_point = bbox.Center();
```

El siguiente paso, será crear un Array donde almacenar todos los datos que se van a ir obteniendo del análisis del Array *geom* creado anteriormente:

```
ON_SimpleArray<ON_MassProperties> MassProp;
```

Para darle el mismo tamaño que el array *geom*, se define la siguiente línea:

```
MassProp.Reserve( geom.Count() );
```

.Reserve es un método reservado que aumenta la capacidad del array *MassProp* a la contabilizada en *geom* a través de *geom.Count()*.

El siguiente paso se trata de calcular las propiedades de la superficie. Para ello, se crea un bucle *for* en el que evaluar una por una las superficies almacenadas en *geom*. Los datos obtenidos se recuperan mediante un puntero a la clase *ON_MassProperties*mp*:

```
for( i = 0; i < geom.Count(); i++ )
{
  ON_MassProperties* mp = &MassProp.AppendNew();

  if( const ON_Surface* srf = ON_Surface::Cast(geom[i]) )
    srf->AreaMassProperties( *mp, true, true, true,
    false, 0.000001,0.000001 );

  else if( const ON_Brep* brep = ON_Brep::Cast(geom[i]) )
    brep->AreaMassProperties( *mp, true, true, true,
    false, 0.000001,0.000001 );
}
```

El siguiente paso, tal y como se muestra en el código anterior es obtener si se trata de una superficie o de una malla, realizando el cálculo de la misma forma: se llama al método *AreaMassProperties*, el cual se encargará de realizar las operaciones de obtención de distintas características:

AreaMass-Properties(mp*,	,bool	,bool	,bool	,bool	,0.000001	, 0.00001)
	Calcula el área	Calcula el momento de primer orden	Calcula el momento de segundo orden	Calcula el producto de los momentos	Tolerancia relativa	Tolerancia absoluta

Si se define algunas de las referencias a los cálculos como *false,* no realizará las operaciones asignadas, con el ahorro de coste computacional.

Conforme se van realizando las operaciones solicitadas, los datos obtenidos se van almacenando en un array distinto para cada superficie que hay en el array *geom*. Por tanto, habrá que crear un objeto que recupere y sume los arrays obtenidos en el paso anterior para así obtener un array con el total de las propiedades:

```
ON_MassProperties results;
results.Sum( MassProp.Count(), MassProp.Array() );
```

El array *results* suma mediante el método *.Sum* a los arrays *MassProp.Array()*.

Muestra de los resultados obtenidos

Una vez se tiene la matriz *results*, ya se puede proceder a la recuperación de los datos que interesen y mostrarlos en pantalla. Para una mejor visualización de los resultados, también se añadirá automáticamente un punto que represente el centro del área de flotación:

```
ON_3dPoint pt( results.m_x0, results.m_y0, results.m_z0 );
context.m_doc.AddPointObject( pt );
context.m_doc.Redraw();
```

Mientras que para mostrar los resultados se utilizará el método *Print* de la clase *RhinoApp()*:

```
RhinoApp().Print( L"Area=%g",results.Area() );
RhinoApp().Print(L" m^2\n");
RhinoApp().Print( L"TCI (dens. 1.025 ton/m^3)=%g",
    results.Area()*1.025/100 );
RhinoApp().Print(L" ton\n");
RhinoApp().Print( L"Centro de Flotación;\n");
RhinoApp().Print( L"CFL=%g",results.m_y0 );
RhinoApp().Print(L" m\n");
RhinoApp().Print( L"Calado en la flotación=%g",results.m_z0 );
RhinoApp().Print(L" m\n");
RhinoApp().Print( L"Inercia transversal=%g",
    results.CentroidCoordMomentsOfInertia().x );
RhinoApp().Print(L" m^4\n");
RhinoApp().Print( L"Inercia longitudinal=%g",
    results.CentroidCoordMomentsOfInertia().y );
RhinoApp().Print(L" m^4\n");
```

Cálculo del volumen de carena

Siguiendo con el código, el proceso habrá que repetirlo tomando una nueva geometría (el volumen de carena) y variar los filtros, aunque el proceso es muy parecido al visto para el área de flotación. Ésta vez, el filtro aceptará superficies, polisuperficies y mallas, y el objeto creado será como identificador *go2*, para diferenciar el proceso del proceso llevado a cabo en el cálculo del área de flotación:

```
CRhinoGetObject go2;
go2.SetCommandPrompt( L"Seleccione el volumen de carena" );
go2.SetGeometryFilter(
      CRhinoGetObject::surface_object |
      CRhinoGetObject::polysrf_object |
      CRhinoGetObject::mesh_object
      );
go2.SetGeometryAttributeFilter(
      CRhinoGetObject::closed_surface |
      CRhinoGetObject::closed_polysrf |
      CRhinoGetObject::closed_mesh
      );
go2.EnableSubObjectSelect( false );
go2.EnableGroupSelect( true );
go2.GetObjects( 1, 0 );
if( go2.CommandResult() != success )
  return go2.CommandResult();
```

En el método *SetGeometryAttributeFilter* se establece que se seleccione una superficie cerrada, por lo que se tendrá que tener el volumen de carena correctamente identificado, cerrado y agrupado.

Como se puede observar, el resto de comandos son iguales que para el caso del área. Ésta vez, en el array de almacenamiento de la geometría, es necesario identificar la geometría con otro nombre (*geom2*) para diferenciarlo de *geom*. La función del bucle *for,* es la misma que para el caso del área:

```
ON_SimpleArray<const ON_Geometry*> geom2( go2.ObjectCount() );
for( i = 0; i < go2.ObjectCount(); i++ )
{
  const ON_Geometry* geo = go2.Object(i).Geometry();
  if( 0 == geo )
    return failure;
  geom2.Append( geo );
}
```

Por último, se procede al cálculo de las propiedades de la geometría. En éste caso, se vuelve a crear una caja contenedora de la geometría para tomar referencia denominada *bbox2*:

```
ON_BoundingBox bbox2;
for( i = 0; i < geom2.Count(); i++ )
  geom2[i]->GetBoundingBox( bbox2, bbox2.IsValid() );

ON_3dPoint base_point2 = bbox2.Center();

ON_SimpleArray<ON_MassProperties> MassProp2;
MassProp2.Reserve( geom2.Count() );
```

Para proceder al cálculo y obtención de las propiedades del volumen, se utiliza un método similar al usado para el área, pero específico para volúmenes:

```
for( i = 0; i < geom2.Count(); i++ )
{
  ON_MassProperties* mp = &MassProp2.AppendNew();

  if( const ON_Surface* srf = ON_Surface::Cast(geom2[i]) )
    srf->VolumeMassProperties( *mp, true, true, false,
    false, base_point );

  else if( const ON_Brep* brep = ON_Brep::Cast(geom2[i]) )
    brep->VolumeMassProperties( *mp, true, true, false,
    false, base_point );

  else if( const ON_Mesh* mesh = ON_Mesh::Cast(geom2[i]) )
    mesh->VolumeMassProperties( *mp, true, true, false,
    false, base_point );
}
```

El funcionamiento y el orden de los parámetros a calcular es el mismo que para el área:

VolumeMass-Properties(mp*,	,bool	,bool	,bool	,bool	Base_point
	Calcula el área	Calcula el momento de primer orden	Calcula el momento de segundo orden	Calcula el producto de los momentos	Punto base sobre el que calcular los datos.

A la hora de representar los datos obtenidos, se procede de la misma manera: se almacenan los datos obtenidos en un Array de resultados (*results 2*) del que más tarde se extraerán los datos exigidos:

```
 ON_MassProperties results2;
results2.Sum( MassProp.Count(), MassProp2.Array() );
```

Por último, se procede a representar el centro de carena, añadiendo un punto en las coordenadas calculadas, y se muestran y calculan las distintas características de la geometría:

```
ON_3dPoint pt2( results2.m_x0, results2.m_y0, results2.m_z0 );
context.m_doc.AddPointObject( pt2 );
context.m_doc.Redraw();


RhinoApp().Print( L"Volumen de carena = %g", results2.Volume() );
RhinoApp().Print(L" m^3\n");
RhinoApp().Print( L"Desplazamiento (dens.=1.025 ton/m^3) = %g",
    results2.Volume()*1.025 );
RhinoApp().Print(L" ton\n");
RhinoApp().Print( L"Posición longitudinal del centro de carena, "
    "LCB = %g", results2.m_x0 );
RhinoApp().Print(L" m\n");
RhinoApp().Print( L"Altura del centro de carena; KB = %g",
    results2.m_z0 );
RhinoApp().Print(L" m\n");
RhinoApp().Print( L"Radio metacéntrico transversal; BMt = %g",
    results.CentroidCoordMomentsOfInertia().x/ results2.Volume() );
RhinoApp().Print(L" m\n");
RhinoApp().Print( L"Radio Metacéntrico longitudinal; BMl = %g",
    results.CentroidCoordMomentsOfInertia().y/ results2.Volume() );
RhinoApp().Print(L" m\n");
```

Cálculo del área mojada

Para el cálculo del área mojada se necesita crear otra matriz de resultados (*results3*), con datos obtenidos de aplicar a la geometría 2 (*geom2*), contenida en la caja *bbox2*, el método *AreaMassProperties*, por lo que como se ha visto anteriormente, sólo se deberá aplicar el último paso del cálculo de propiedades y representarlo en pantalla. La única salvedad es que en la definición de las propiedades a calcular sólo se marcara como *true* la que se refiere al cálculo del área:

```
ON_SimpleArray<ON_MassProperties> MassProp3;
MassProp3.Reserve( geom2.Count() );

for( i = 0; i < geom2.Count(); i++ )
{
  ON_MassProperties* mp = &MassProp3.AppendNew();

  if( const ON_Surface* srf = ON_Surface::Cast(geom2[i]) )
    srf->AreaMassProperties( *mp, true, false, false, false,
    1.000000,1.000000 );

  else if( const ON_Brep* brep = ON_Brep::Cast(geom2[i]) )
    brep->AreaMassProperties( *mp, true, false, false, false,
    1.000000,1.000000 );
}
```

Al igual que en los dos casos anteriores, se ensambla una nueva matriz de resultados, donde no se debe olvidar que el área calculada contiene también al área de flotación, por lo que habrá que restarla (*restult3s.Area()-restults.Area()*):

```
ON_MassProperties results3;
results3.Sum( MassProp3.Count(), MassProp3.Array() );

RhinoApp().Print( L"Area mojada = %g",
    results3.Area()-results.Area());
RhinoApp().Print(L" m^2\n");

return success;
```

Por último, para finalizar el programa, se devuelve el código *success*, terminando el proceso.

Código completo *prophidr*:

A continuación se muestra el código completo *prophidr*, procedente del archivo *cmdprophidr*. En dicho archivo, se podrá ver el código con los comentarios necesarios para su comprensión:

```
#include "StdAfx.h"
#include "prophidrPlugIn.h"


class CCommandprophidr : public CRhinoCommand
{
public:

      CCommandprophidr() {}
      ~CCommandprophidr() {}

      UUID CommandUUID()
      {
            // {85EEC19C-2ADF-4988-9E62-0CC2F6060B2E}
    static const GUID prophidrCommand_UUID =
    { 0x85EEC19C, 0x2ADF, 0x4988, { 0x9E, 0x62, 0xC, 0xC2, 0xF6, 0x6,
0xB, 0x2E } };
    return prophidrCommand_UUID;
      }
```

```
      const wchar_t* EnglishCommandName() { return L"prophidr"; }
      const wchar_t* LocalCommandName() { return L"prophidr"; }
      CRhinoCommand::result RunCommand( const CRhinoCommandContext& );
};

static class CCommandprophidr theprophidrCommand;


//CÓDIGO PARA EL CÁLCULO DE LAS PROPIEDADES HIDROSTÁTICAS

CRhinoCommand::result CCommandprophidr::RunCommand( const
CRhinoCommandContext& context )
{
      //Cálculo de propiedades del área mojada
CRhinoGetObject go;
go.SetCommandPrompt( L"Seleccione el área de flotación" );
go.SetGeometryFilter(CRhinoGetObject::surface_object |
CRhinoGetObject::polysrf_object
);
go.EnableSubObjectSelect( false );
go.EnableGroupSelect( true );
go.GetObjects( 1, 0 );
if( go.CommandResult() != success )
    return go.CommandResult();

ON_SimpleArray<const ON_Geometry*> geom( go.ObjectCount() );
int i;
for( i = 0; i < go.ObjectCount(); i++ )
  {
    const ON_Geometry* geo = go.Object(i).Geometry();
    if( 0 == geo )
      return failure;
    geom.Append( geo );
  }

ON_BoundingBox bbox;
for( i = 0; i < geom.Count(); i++ )
    geom[i]->GetBoundingBox( bbox, bbox.IsValid() );

ON_3dPoint base_point = bbox.Center();
ON_SimpleArray<ON_MassProperties> MassProp;
MassProp.Reserve( geom.Count() );

for( i = 0; i < geom.Count(); i++ )
{
    ON_MassProperties* mp = &MassProp.AppendNew();
    if( const ON_Surface* srf = ON_Surface::Cast(geom[i]) )
    srf->AreaMassProperties( *mp, true, true, true, true,
1.000000,1.000000 );

    else if( const ON_Brep* brep = ON_Brep::Cast(geom[i]) )
    brep->AreaMassProperties( *mp, true, true, true, true,
    1.000000,1.000000 );
}

ON_MassProperties results;
results.Sum( MassProp.Count(), MassProp.Array() );
```

```
ON_3dPoint pt( results.m_x0, results.m_y0, results.m_z0 );
context.m_doc.AddPointObject( pt );
  context.m_doc.Redraw();

  RhinoApp().Print( L"Area=%g",results.Area() );
  RhinoApp().Print(L" m^2\n");
  RhinoApp().Print( L"TCI (dens. 1.025 ton/m^3)=%g",
        results.Area()*1.025/100 );
  RhinoApp().Print(L" ton\n");
  RhinoApp().Print( L"Centro de Flotación;\n");
  RhinoApp().Print( L"CFL=%g",results.m_y0 );
  RhinoApp().Print(L" m\n");
  RhinoApp().Print( L"Calado en la flotación=%g",results.m_z0 );
  RhinoApp().Print(L" m\n");
  RhinoApp().Print( L"Inercia transversal=%g",
        results.CentroidCoordMomentsOfInertia().x );
  RhinoApp().Print(L" m^4\n");
  RhinoApp().Print( L"Inercia longitudinal=%g",
        results.CentroidCoordMomentsOfInertia().y );
  RhinoApp().Print(L" m^4\n");

//Calculo de propiedades del volumen de carena

  CRhinoGetObject go2;
  go2.SetCommandPrompt( L"Seleccione el volumen de carena" );
  go2.SetGeometryFilter(
        CRhinoGetObject::surface_object |
        CRhinoGetObject::polysrf_object |
        CRhinoGetObject::mesh_object
        );
  go2.SetGeometryAttributeFilter(
        CRhinoGetObject::closed_surface |
        CRhinoGetObject::closed_polysrf |
        CRhinoGetObject::closed_mesh
        );
  go2.EnableSubObjectSelect( false );
  go2.EnableGroupSelect( true );
  go2.GetObjects( 1, 0 );
  if( go2.CommandResult() != success )
    return go2.CommandResult();

  ON_SimpleArray<const ON_Geometry*> geom2( go2.ObjectCount() );
  for( i = 0; i < go2.ObjectCount(); i++ )
  {
    const ON_Geometry* geo = go2.Object(i).Geometry();
    if( 0 == geo )
      return failure;
    geom2.Append( geo );
  }

  ON_BoundingBox bbox2;
  for( i = 0; i < geom2.Count(); i++ )
    geom2[i]->GetBoundingBox( bbox2, bbox2.IsValid() );

  ON_3dPoint base_point2 = bbox2.Center();

  ON_SimpleArray<ON_MassProperties> MassProp2;
  MassProp2.Reserve( geom2.Count() );

  for( i = 0; i < geom2.Count(); i++ )
  {
```

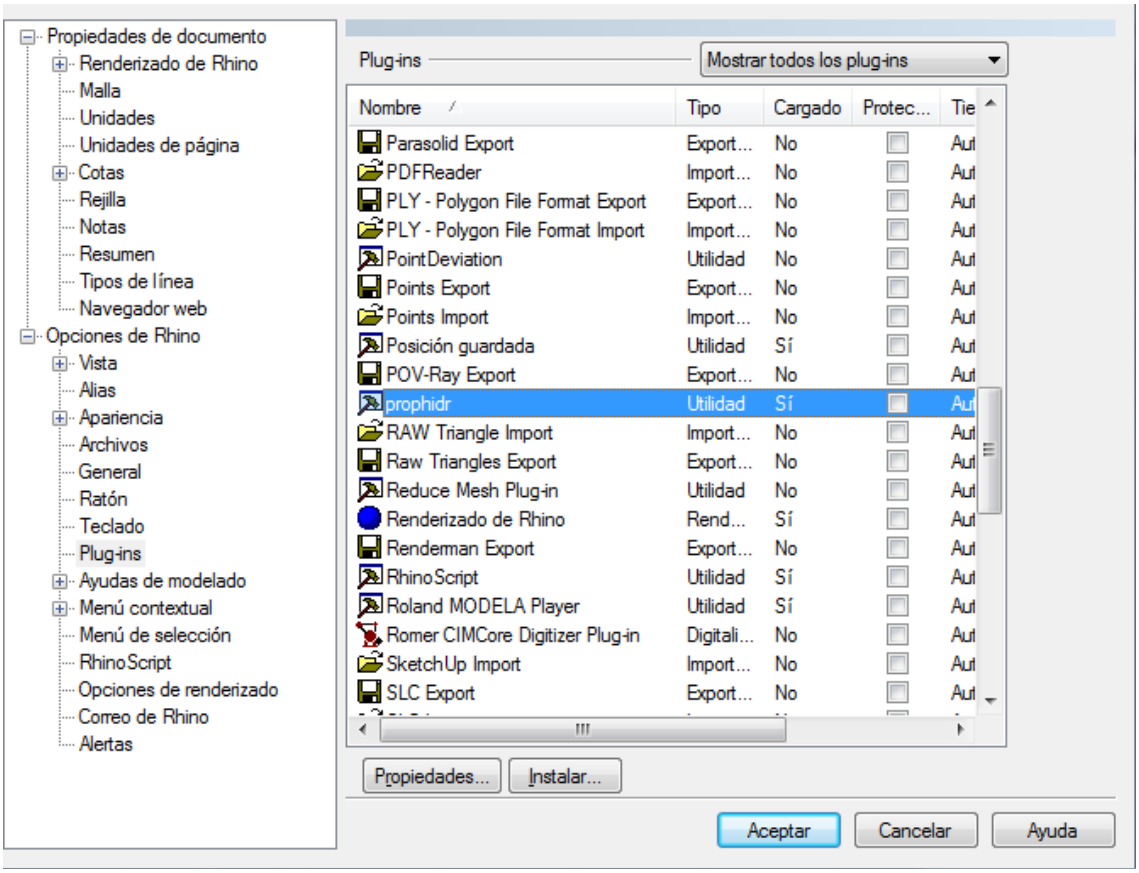
```
    ON_MassProperties* mp = &MassProp2.AppendNew();

    if( const ON_Surface* srf = ON_Surface::Cast(geom2[i]) )
      srf->VolumeMassProperties( *mp, true, true, false, false,
      base_point );

    else if( const ON_Brep* brep = ON_Brep::Cast(geom2[i]) )
      brep->VolumeMassProperties( *mp, true, true, false, false,
      base_point );

    else if( const ON_Mesh* mesh = ON_Mesh::Cast(geom2[i]) )
      mesh->VolumeMassProperties( *mp, true, true, false, false,
      base_point );
  }
ON_MassProperties results2;
results2.Sum( MassProp.Count(), MassProp2.Array() );

ON_3dPoint pt2( results2.m_x0, results2.m_y0, results2.m_z0 );
context.m_doc.AddPointObject( pt2 );
context.m_doc.Redraw();


  RhinoApp().Print( L"Volumen de carena = %g", results2.Volume() );
  RhinoApp().Print(L" m^3\n");
  RhinoApp().Print( L"Desplazamiento (dens.=1.025 ton/m^3) = %g",
results2.Volume()*1.025 );
  RhinoApp().Print(L" ton\n");
  RhinoApp().Print( L"Posición longitudinal del centro de carena, LCB
= %g", results2.m_x0 );
  RhinoApp().Print(L" m\n");
  RhinoApp().Print( L"Altura del centro de carena; KB = %g",
results2.m_z0 );
  RhinoApp().Print(L" m\n");
  RhinoApp().Print( L"Radio metacéntrico transversal; BMt = %g",
results.CentroidCoordMomentsOfInertia().x/ results2.Volume() );
  RhinoApp().Print(L" m\n");
  RhinoApp().Print( L"Radio Metacéntrico longitudinal; BMl = %g",
results.CentroidCoordMomentsOfInertia().y/ results2.Volume() );
  RhinoApp().Print(L" m\n");

      //Cálculo del área mojada

  ON_SimpleArray<ON_MassProperties> MassProp3;
  MassProp3.Reserve( geom2.Count() );

  for( i = 0; i < geom2.Count(); i++ )
  {
    ON_MassProperties* mp = &MassProp3.AppendNew();

      if( const ON_Surface* srf = ON_Surface::Cast(geom2[i]) )
      srf->AreaMassProperties( *mp, true, false, false, false,
      1.000000,1.000000 );

      else if( const ON_Brep* brep = ON_Brep::Cast(geom2[i]) )
      brep->AreaMassProperties( *mp, true, false, false, false,
      1.000000,1.000000 );
  }

  ON_MassProperties results3;
  results3.Sum( MassProp3.Count(), MassProp3.Array() );
```

```
  RhinoApp().Print( L"Area mojada = %g",
        results3.Area()-results.Area());
  RhinoApp().Print(L" m^2\n");

  return success;
}

// FINAL del codigo prophidr
```

Depuración y ejecución del Plug-in

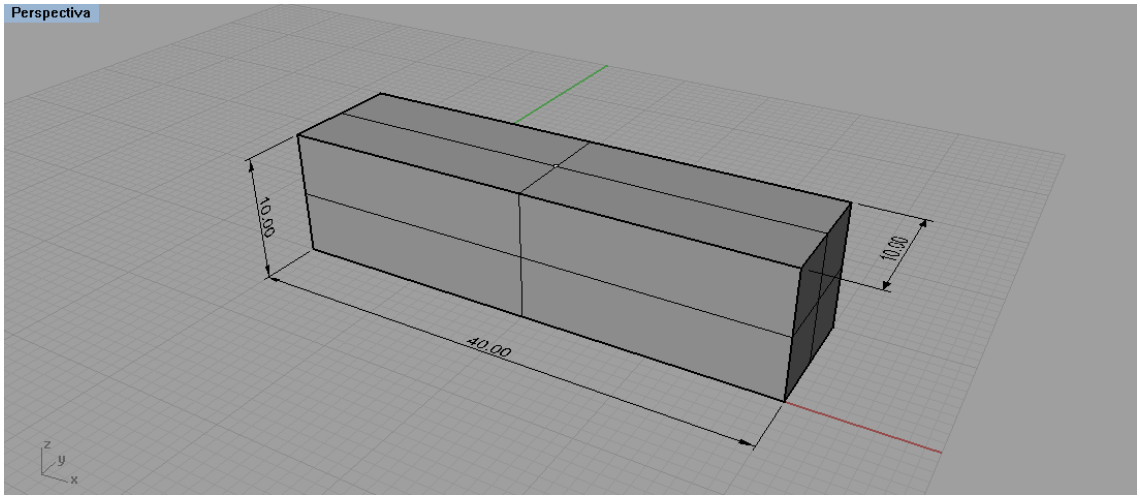
Una vez implementado el código en Visual C++, será necesario depurar el programa. La depuración detectará los posibles errores que se hayan cometido a lo largo de la redacción del código. Normalmente éstos errores están correctamente acotados por el propio programa, incluso ofreciendo soluciones esperadas (por ejemplo, al no escribir un punto y coma en un lugar esperado, el propio programa lo indica).

Si la depuración tiene éxito, a continuación aparecerá cómo se ejecuta y se abre Rhino. A partir de aquí, se siguen los pasos vistos en la introducción de éste apartado: se accede a las propiedades de Rhino para instalar el plug-in (Archivo/Propiedades…/Pulg-ins/Intalar…). Se selecciona el plug-in en su directorio y se instala, apareciendo en la lista de plug-ins cargados:



Ejemplo 1

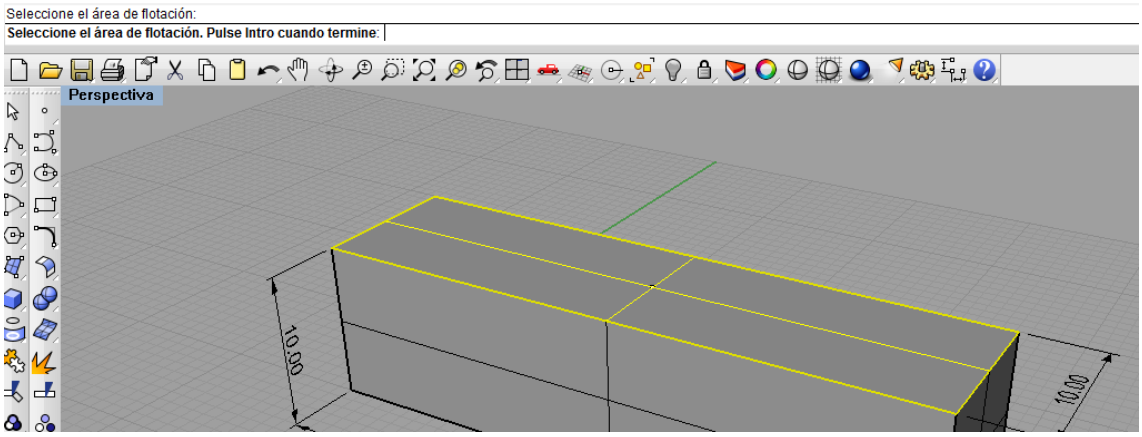
A partir de aquí, ya sólo falta verificar que funciona correctamente. Para ello, se va a analizar una pontona, la cual es sencillo saber cuáles son sus características hidrodinámicas. Por ejemplo, si se le dan unas dimensiones de L=40m, B=10m y T=10m:



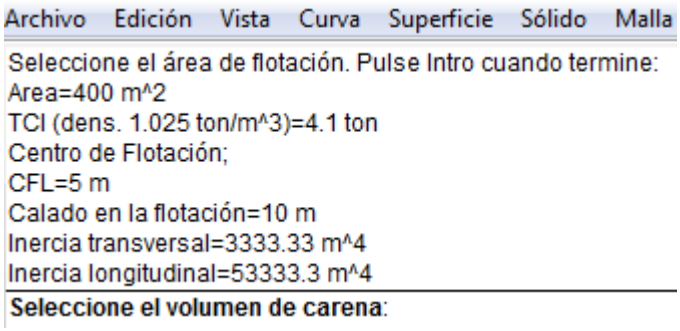
Dada sus formas, no es difícil obtener sus características hidrostáticas de forma directa:

Propiedades hidrodinámicas de una pontona					
Densidad A.m.	1,025	ton/m^3	Inercia longitudinal	53333,333	m^4
L	40	m	Inercia Transversal	3333,333	m^4
B	10	m	BMl	13,333	m
T	10	m	BMt	0,833	m
Volumen de carena	4000	m^3	Sup.mojada	1400	m^2
Desplazamiento	4100	ton	LCB	20	m
Área superficie flotación	400	m^2	KB	5	m
TCI	4,1	ton	Pos. long CFL	20	m

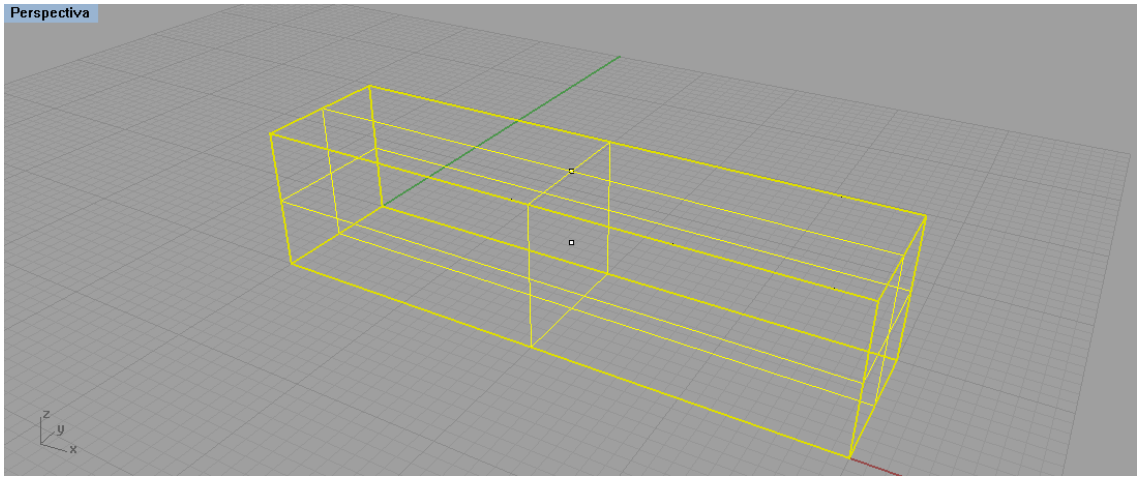
Si aplicamos a la geometría en Rhinoceros la orden de ejecutar el plug-in diseñado (basta con escribir *prophidr* para ejecutarlo), es importante recordar que se tendrá que tener, por un lado la superficie de flotación, y por otro lado el volumen de carena. Si seguimos el cuadro de diálogo, primero pedirá que se seleccione la superficie de flotación:



Como tal y como se diseñó, también se admiten polisuperficies, es necesario pulsar intro una vez que se hayan seleccionado la totalidad del área de flotación. Al pulsar intro, aparecerán los resultados de los cálculos referidos a la superficie de flotación:



Si se comprueban los datos obtenidos con los calculados analíticamente, se comprueba que son correctos. Acto seguido, tal y como se observa en la imagen anterior, se pide la selección del volumen de carena:



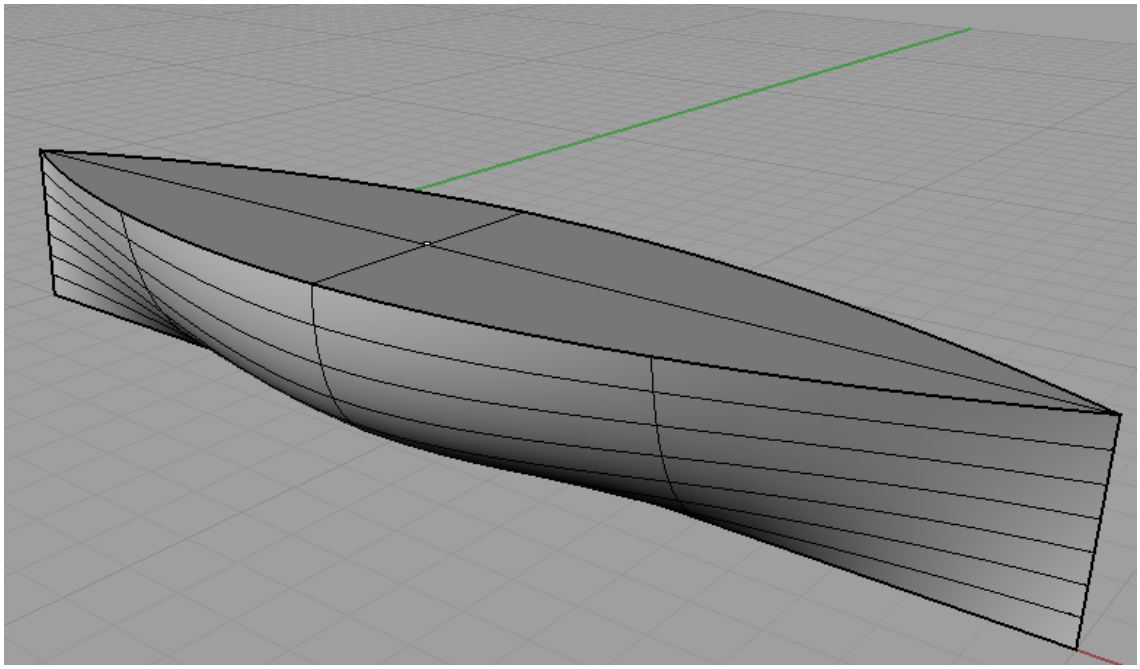
Pulsando nuevamente intro, aparecerán los resultados del resto de cálculos, coincidiendo perfectamente con los calculados analíticamente:

```
Archivo  Edición  Vista  Curva  Superficie  Sólido  Malla
Seleccione el volumen de carena. Pulse Intro cuando termine:
Volumen de carena = 4000 m^3
Desplazamiento (dens.=1.025 ton/m^3) = 4100 ton
Posición longitudinal del centro de carena, LCB = 20 m
Altura del centro de carena; KB = 5 m
Radio metacéntrico transversal; BMt = 0.833333 m
Radio Metacéntrico longitudinal; BMl = 13.3333 m
Area mojada = 1400 m^2
Comando:
```

Una vez realizado la segunda tanda de operaciones, el proceso finaliza satisfactoriamente.

Por supuesto, dicho plug-in es aplicable a formas de carenas más complejas como las siguientes:

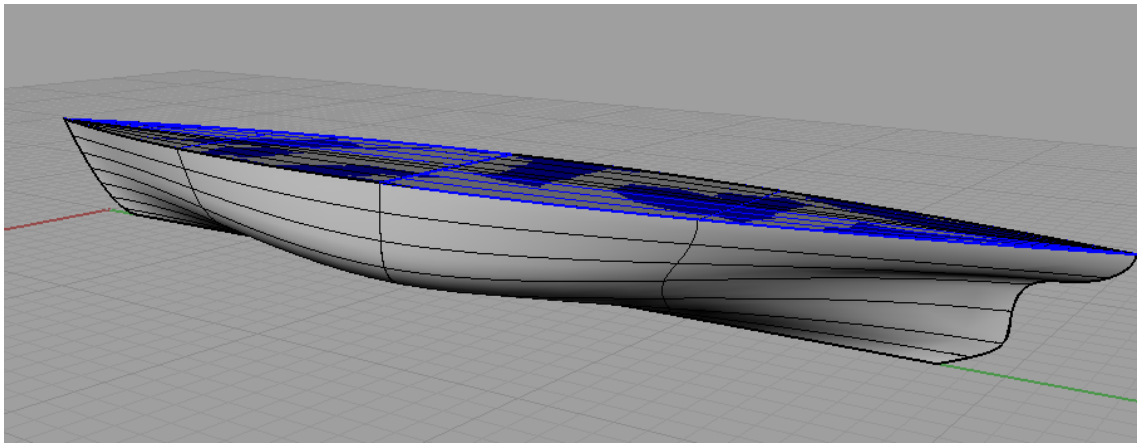
Ejemplo 2:



Resultados del análisis con el plug-in *prophidr*:

```
Seleccione el área de flotación. Pulse Intro cuando termine:
Area=66.4159 m^2
TCI (dens. 1.025 ton/m^3)=0.680763 ton
Centro de Flotación;
CFL=-1.52097e-16 m
Calado en la flotación=3 m
Inercia transversal=94.3919 m^4
Inercia longitudinal=1326.05 m^4
Seleccione el volumen de carena:
Seleccione el volumen de carena. Pulse Intro cuando termine:
Volumen de carena = 144.679 m^3
Desplazamiento (dens.=1.025 ton/m^3) = 148.296 ton
Posición longitudinal del centro de carena, LCB = 10 m
Altura del centro de carena; KB = 1.77515 m
Radio metacéntrico transversal; BMt = 0.652424 m
Radio Metacéntrico longitudinal; BMl = 9.16549 m
Area mojada = 151.076 m^2
Comando:
```

Ejemplo 3:



Resultados del análisis con el plug-in *prophidr*:

```
Area=244.83 m^2
TCI (dens. 1.025 ton/m^3)=2.50951 ton
Centro de Flotación;
CFL=20.0243 m
Calado en la flotación=4 m
Inercia transversal=25878.9 m^4
Inercia longitudinal=892.854 m^4
Seleccione el volumen de carena:
Seleccione el volumen de carena. Pulse Intro cuando termine:
Volumen de carena = 745.326 m^3
Desplazamiento (dens.=1.025 ton/m^3) = 763.959 ton
Posición longitudinal del centro de carena, LCB = -2.6224e-16 m
Altura del centro de carena; KB = 2.28699 m
Radio metacéntrico transversal; BMt = 34.7215 m
Radio Metacéntrico longitudinal; BMl = 1.19794 m
Area mojada = 474.396 m^2
Comando:
```

6.Apéndice I: Conceptos de C++

Variables y tipos de datos fundamentales

Nombre	Descripción	Tamaño	Rango
char	Caracter o entero pequeño	1 byte	con signo:-127 a 127 Sin signo: 0 a 255
short int	Entero corto	2 bytes	Con signo: -32768 a 32767 Sin signo: 0 a 65535
int	entero	4 bytes	Con signo: -2147483648 a 2147483647 Sin signo: 0 a 4294967295
long int	entero largo	4 bytes	Con signo: -2147483648 a 2147483647 Sin signo: 0 a 4294967295
bool	Valor booleano: verdadero o falso	1 byte	verdadero (1) o falso (0)
float	Numero de punto flotante	4 bytes	+-3e(+-38) ≈ (7digitos)
double	Numero punto flotante de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)
long double	Número punto flotante largo de doble precisión	8 bytes	+-7e(+-308) ≈ (15digitos)

Declaración y alcance de las variables

```
1   #include<iostream>
2   using namespace std;
3
4   int entero;
5   char caracter;                    Variables globales
6   char string [20];
7   unsigned int Tiempo;
8   int main()
9   {
10      unsigned short Edad;          Variables locales
11      float Estatura, peso;
12      cout<<'introduce tu edad; '
13      con<<Edad;                    Instrucciones
14      ...
15  }
```

Declaración de cadenas de caracteres

```
1   string micadena="cadena de caracteres";
2   string micadena("cadena de caracteres");
```

Declaración de constantes

Declaración en la cabecera:

```
1   #define PI 3.14159
2   #define NUEVALINEA'\n'
```

Constantes declaradas:

```
1   const int Kilometros=100;
2   const char tabulador='\t';
```

Comandos especiales:

\n	Salto de línea	\f	Salto de página
\r	Retorno	\a	Alerta (beep)
\t	Tabulación	\'	Comilla simple
\v	Tabulación vertical	\"	Comilla doble
\b	Backspace	\\	Contrabarra

Operadores:

Asignación y comparación:

=	Asignación	Asigna un valor a una variable: int b=1;
==	Comparación	Compara dos valores: if int b==a; {...}

Operadores aritméticos:

+	suma	/	división
-	resta	%	modulo
*	multiplicación		

Asignaciones compuestas:

Expresión	Ejemplo	Equivalente a:
+=	a+=1	a=a+1
-=	a-=5	a=a-5
*=	a*=a	a=a*a
/=	a/=b	a=a/b

%=	Asigna el resto de la operación	&=	"Y" y asignación
<<=	Cambio de la izquierda y asignación	|=	"O" inclusivo y asignación
>>=	Cambio de la derecha y asignación	^=	"O" exclusivo y asignación

Incremento (++) y decremento (--):

Ejemplo 1	Ejemplo 2
`1 B=3;` `2 A=++B;`	`1 B=3;` `2 A=B++;`
El valor de *A* es el mismo que el de *B* incrementado en 1, por tanto: *A=4* y *B=4*	A la variable *A* se le asigna el valor de *B* antes de ser incrementado, por tanto, al final se tiene: *A=3* y *B=4*

Los ejemplos son igualmente aplicables para el caso de decremento (--).

Operadores de relación e igualdad:

==	Igual a[2]	<	Menor que
!=	No igual a	>=	Mayor o igual que
>	Mayor que	<=	Menor o igual que

Operadores lógicos:

-Operador !: Operador para evaluar la operación booleana *no*. Indica que si una comparación situada a la derecha de ! no se cumple, devuelve el valor booleano *verdadero*, mientras que si la expresión es correcta, devuelve el valor booleano *Falso*.

```
1   !(5==5)      //Evalua la expresión como Falsa, y que
2                //5==5 es verdadero
3   !(6<=4)      //Evalúa como verdadera la expresion, porque
4                //6<=4 es falso
5   !true        //Devuelve un Falso
6   !false       //Devuelve un Verdadero
```

-Operador &&: corresponde con la expresión lógica booleana "y". Ésta operación es verdadera si ambos operandos son verdaderos, y falso en cualquier otro caso:

a	b	a&&b
Verdadero	Falso	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

-El operador ||: corresponde con la expresión lógica booleana "o". Ésta operación resulta verdadera cuando alguno de los operandos es verdadero, y es falso sólo cuando todos son falsos:

a	b	a||b
Verdadero	Falso	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Operador condicional:

```
condición ? resultado1 : resultado2
```

Si *condición* es verdadero la expresión devuelve *resultado1*. Si por el contrario es falsa, devolverá *resultado2*.

Operador coma:

El operador coma (,) se usa para separar dos o más expresiones que son evaluadas cuando sólo se espera el cumplimiento de una de las expresiones:

[2] No confundir la asignación de valores a las variables (=) con la comparación entre valores (==)

```
1    a=(b=3,b+2);
```

Operadores bit a bit (*bitwise*):

Operadores	Equivalencia	Descripción
&	AND	*Bitwise "y"*
|	OR	bitwise "o inclusivo"
^	XOR	bitwise "o exclusivo"
~	NOT	Complemento unario (inversión bit)
<<	SHL	Cambio izquierdo
>>	SHR	Cambio derecho

Operadores de conversión de tipo explícita:

Permiten convertir un dato dado en un tipo de variable a otro tipo de variable:

```
1    int i;
2    float f=3.14;
3    i=(int)f;
```

Operador tamaño *sizeof*:

Toma un parámetro, el cual puede ser de cualquier tipo o variable, y devuelve su tamaño en bytes:

```
1    a=sizeof (char);
```

Orden de los operadores:

Nivel	Operador	Descripción	Agrupamiento
1	`::`	Alcance	Izquierda a derecha
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	Sufijo	Izquierda a derecha
3	`++ -- ~ ! sizeof new delete`	Unario (prefijo)	Derecha a izquierda
	`* &`	Indirección y referencia (punteros)	
	`+ -`	operadores de signo unarios	
4	`(type)`	Conversión de tipo	Derecha a izquierda
5	`.*  ->*`	punteros-a-miembro	Izquierda a derecha
6	`* /  %`	multiplicativos	Izquierda a derecha
7	`+ -`	aditivos	Izquierda a derecha
8	`>>  <<`	Cambio	Izquierda a derecha
9	`< > <= >=`	Relacional	Izquierda a derecha
10	`== !=`	Igualdad	Izquierda a derecha
11	`&`	Bitwise "y"	Izquierda a derecha
12	`^`	Bitwise "xor"	Izquierda a derecha

13	|	Bitwise “o”	Izquierda a derecha
14	&&	“y” lógico	Izquierda a derecha
15	||	“o” lógico	Izquierda a derecha
16	?:	Condicional	Derecha a izquierda
17	= *= /= += -= <<= >>= &= ^= |=	Asignación	Derecha a izquierda
18	,	Coma	Izquierda a derecha

Entradas y salidas básicas:

Para permitir datos de entrada, será necesario incluir en la cabecera *#include<string>*:

Salida cout<<”texto”<<a<<endl;

Entrada it edad;

cin>>edad;

Entrada getline (cin, variable);

Conversión

```
1   string mistr ("1234");
2   int mistr;
3   stringtream(mistr)>>mistr;
```

Estructuras de control

Estructura condicional *if/else*:

if (condición) declaración;

if (condición) {declaraciones;} else {declaraciones;}

Estructuras de iteración:

while (mientras) *while (expresión) declaración;*

do-while *do declaración while (condition);*

El bucle for:

for (inicialización; condición; incremento) {declaración(es);}

Declaraciones de salto

break	Interrumpe un bucle cuando se da una condición establecida
continue	Continúa ejecutando un bucle ignorando el resto del bloque
goto	Realiza un salto a otro punto del programa
exit	(librería cstdlib)Termina el programa actual con un código específico de salida

switch	Chequea varios valores constantes posibles en una relación

Funciones

Definición

Grupo de declaraciones ejecutadas cuando son llamadas durante la ejecución de un programa:

```
1   type nombre(parametro1, parámetro2,..){declaraciones}
```

Argumentos en funciones:

Argumentos con origen en valores

```
int adicion (int a,int b)
  z=adicion ( 5 , 2 );
```

Argumentos por referencia

```
void duplicar ( int& a, int& b, int& c){
    duplicar ( x , y , z );
```

Valores por defecto

```
int dividir (int a, int b=2)
```

b tiene el valor por defecto =2

Funciones en línea (*inline*):

```
inline tipo nombre (argumentos) {cuerpo...}
```

Funciones *prototipo*:

```
tipo nombre (tipo de argumento1, tipo de argumento2,...);
```

-Por ejemplo:

```
int prototipo (int identificador1, int identificador2);
int prototipo (int, int);
```

Arrays

Definición:

```
tipo nombre [elementos];
```

-Por ejemplo:

```
int barco [5];
```

-Inicialización:

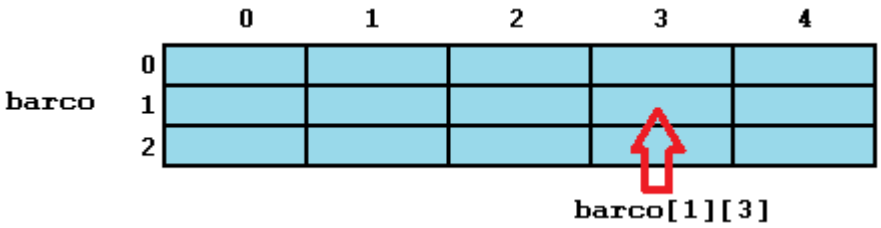
```
int barco [5]={ 15, 21, 35, 65, 558};
```

-Acceso a valores:

	barco[0]	barco[1]	barco[2]	barco[3]	barco[4]
barco	15	21	35	65	558

Arrays multidimensionales:

```
barco[1][3]
```



Arrays como parámetro de una función:

```
void proceso { int argumento[]}
```

Secuencias de caracteres

Definición

```
char nombre [20];
char saludo[]="Hola";
char saludo[]={'H','o','l','a','\0'};
```

Punteros

Operador de referencia (&) y desreferencia (*)

Referencia	&	`A=&B`	A=la dirección de B
Desreferencia	*	`D=*A`	D=valor apuntado por A

Declaración de punteros:

```
int * numero;
char * caracteres;
float * PI;
```

Inicialización de punteros

```
int numero;
int *A=&numero;
```

Equivalente a:

```
int numero;
int *A;
A=&numero;
```

Memoria dinámica

Operador *new*

```
puntero=new Tipo;
puntero=new tipo[numero_de_elementos];
```

Operador *delete*:

```
delete puntero;
delete [] puntero;
```

Estructuras de datos

Definición:

```
struct nombre_de_la_estructura {
    tipo_de_miembro_1 nombre_del_miembro_1;
    tipo_de_miembro_2 nombre_del_miembro_2;
    tipo_de_miembro_3 nombre_del_miembro_3;
    .
    .
}nombre_de_objetos;
```

Operador Flecha:

Operador flecha -> Equivalente a operador indirección (*), pero su uso está limitado a miembros de estructuras de datos o clases

-Combinaciones entre miembros y punteros de una estructura:

Expresión	Que es lo que evalúa	Equivalencia
a.b	Miembro *b* del objeto *a*	
a->b	Miembro *b* del objeto apuntado por *a*	(*a).b
*a.b	Valor apuntado por el miembro *b* del objeto *a*	*(a.b)

Otros tipos de datos

Tipos de datos *definidos* (*typedef*)

```
typedef tipo_existente nombre_del_nuevo_tipo;
```

Uniones

```
union nombre_de_union{
    Tipo_miembro_1 nombre_miembro_1;
    Tipo_miembro_2 nombre_miembro_2;
    Tipo_miembro_3 nombre_miembro_3;
    .
    .
} nombre_objetos;
```

Uniones anónimas

Estructura con unión regular	Estructura con unión anónima

`struct {`
`  char Nombre [50];`
`  char Armador [50];`
`  union {`
`  float Euros;`
`  int Dolares;`
`  }precio;`
`}Buque;`
	`struct {`
	`  char Nombre [50];`
	`  char Armador [50];`
	`  union {`
	`  float Euros;`
	`  int Dolares;`
	`  };`
	`}Buque;`

Acceso a miembros:
`Buque.precio.Euros`
`Buque.precio.Dolares`
	Acceso a miembros:
	`Buque.Euros`
	`Buque.Dolares`

Enumeraciones

```
enum nombre_de_enumeracion {
    valor1,
    valor2,
    valor3,
    .
    .
}nombre_objetos;
```

Programación orientada a objetos

Clases

Definición:

```
class nombre_clase {
    especificador_de_acceso1:
        miembro1;
    especificador_de_acceso2:
        miembro1;
    ...
}Nombre_de_los_objetos;
```

Operador alcance (*scope "::"*)

Es usado para definir un miembro de la clase desde fuera de la definición de la propia clase:

```
void Rectangulo::tomar_valores (int a, int b){
```

Tipos de acceso a miembros:

Acceso	*public*	*protected*	*private*
miembros de la misma clase	Si	Si	Si
miembros de clases derivadas	Si	Si	No
no miembros	Si	No	No

Constructores y destructores

Constructor, o *función constructora,* es llamado cada vez que se crea un objeto de su clase. Dicha función constructora tendrá el mismo nombre que la clase.

Nombre_clase nombre_objeto(valores) Constructor

~Nombre_clase nombre_objeto(valores) Destructor

Expresiones, operadores y punteros habituales:

Expresión	Interpretación
*x	apuntado por x
&x	dirección de x
x.y	miembro y del objeto x
x->y	miembro y del objeto apuntado por x
(*x).y	miembro y del objeto apuntado por x (equivalente al anterior)
x[0]	primer objeto apuntado por x
x[1]	segundo objeto apuntado por x
x[n]	(n+1) objeto apuntado por x

Operadores sobrecargados

-Definición:

```
tipo operator simbolo (parámetros) {/*...*/}
```

-Operadores que permiten su sobrecarga:

Operadores sobrecargados
+ - * / = < > += -= *= /= << >> <<= >>= == != <= >= ++ -- % & ^ ! | ~ &= ^= |= && || %= [] () , ->* -> new delete new[] delete[]

-Definición de los operadores sobrecargados: situación

Expresión	Operador	Función miembro	Función global
@a	+ - * & ! ~ ++ --	A::operator@()	operator@A
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c,...)	()	A::operator() (B,C...)	-
a->x	->	A::operator->()	-

Donde *a* es un objeto de la clase *A*, *b* es un objeto de la clase *B*, y *c* es un objeto de la clase *C*.

This

La palabra clave *this* representa un puntero al objeto cuya función miembro está siendo ejecutada. Se trata de un puntero al propio objeto.

```
int Identificador::soyyo (Identificador& param)
{
    if(&param==this)return true;
    else return false;
```

Miembros estáticos o variables de clase

static tipo Identificador;

Clases derivadas

```
class nombre_clase_derivada: public nombre_clase_base
{/*...*/}
```

Herencia múltiple

```
class Rectangulo:public Poligono,public Output;
class Triangulo:public Poligono,public Output;
```

7.Apéndice II: Código ASCII

Caracteres de control (0-31, 127)

Binario	Decimal	Hex	Abreviatura	Nombre/Significado
0000 0000	0	00	NUL	Carácter Nulo
0000 0001	1	01	SOH	Inicio de Encabezado
0000 0010	2	02	STX	Inicio de Texto
0000 0011	3	03	ETX	Fin de Texto
0000 0100	4	04	EOT	Fin de Transmisión
0000 0101	5	05	ENQ	Consulta
0000 0110	6	06	ACK	Acuse de recibo
0000 0111	7	07	BEL	Timbre
0000 1000	8	08	BS	Retroceso
0000 1001	9	09	HT	Tabulación horizontal
0000 1010	10	0A	LF	Salto de línea
0000 1011	11	0B	VT	Tabulación Vertical
0000 1100	12	0C	FF	De avance
0000 1101	13	0D	CR	Retorno de carro
0000 1110	14	0E	SO	Mayúsculas fuera
0000 1111	15	0F	SI	En mayúsculas
0001 0000	16	10	DLE	Enlace de datos / Escape
0001 0001	17	11	DC1	Dispositivo de control 1 — oft. XON
0001 0010	18	12	DC2	Dispositivo de control 2
0001 0011	19	13	DC3	Dispositivo de control 3 — oft. XOFF
0001 0100	20	14	DC4	Dispositivo de control 4
0001 0101	21	15	NAK	Confirmación negativa

Binario	Decimal	Hex	Abreviatura	Nombre/Significado
0001 0110	22	16	SYN	Síncrono en espera
0001 0111	23	17	ETB	Fin de Transmisión del Bloque
0001 1000	24	18	CAN	Cancelar
0001 1001	25	19	EM	Finalización del Medio
0001 1010	26	1A	SUB	Substituto
0001 1011	27	1B	ESC	Escape
0001 1100	28	1C	FS	Separador de fichero
0001 1101	29	1D	GS	Separador de grupo
0001 1110	30	1E	RS	Separador de registro
0001 1111	31	1F	US	Separador de unidad
0111 1111	127	7F	DEL	Eliminar

Caracteres imprimibles (32-126)

Binario	Dec	Hex	Representación
0010 0000	32	20	espacio ( )
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+

Binario	Dec	Hex	Representación
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F

Binario	Dec	Hex	Representación
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_
0110 0000	96	60	`
0110 0001	97	61	a

Binario	Dec	Hex	Representación
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|

Binario	Dec	Hex	Representación
0111 1101	125	7D	}
0111 1110	126	7E	~

8.Bibliografía

- **The NURBS book 2ª edición**.
  Les Piegl y Wayne Tiller.
  Editorial Springer. 1996.

- **Métodos computacionales para interrogación de superficies NURBS. Aplicaciones en la industria del automóvil.**
  A. Iglesias, A. Gálvez, J. Puig-Pey, F. Gutierrez.
  Métodos Numéticos en ingeniería y ciencias palicadas. CIMNE, Barcelona 2002

- **B-Spline and NURBS Curves**.
  Marko Sulejic.
  FB Computerwissenschaften A-5020 Salzburg, Austria. 2011

- **NURBS Cuves: A guide for the uninitiated.**
  Philip J. Schneider.
  MacTech, the journal of Apple technology.

- **Introduction to Curves and Surfaces**.
  Peter Chambers.
  SIGGRAPH. 1996

- **The C programming language**.
  Brian W. Kernighan, Dennis M. Ritchie.
  Prentice Hall Software Series. 1988

- **The C++ Programming Language**.
  Bjarne Stroustrup.
  AT&T Labs Murray Hill, New Jersey 3ª Edición. 1997

- **C++ Language tutorial**.
  Juan Soulié.
  Disponible en http://www.cplusplus.com/doc/tutorial/. 2007

- **Manual Básico De Programación En C++**.
  Apoyo a la investigación C.P.D. Servicios Informáticos U.C.M.

9.Enlaces de interés

- **http://www.opennurbs.org.**
  OpenNURBStm initiative. Acceso a la librería openNURBS.

- **http://wiki.mcneel.com/developer/cplusplusplugins.**
  Rhino C++ Plug-in SDK. Acceso y descarga del editor de plug-ins Rhino 4 C++ SDK o Rhino 5 C++ SDK.

- **http://www.rhino3d.com/es/resources/.**
  Acceso a plug-ins comercializados y recursos para desarrolladores.

- **http://www.bloodshed.net/devcpp.html**
  Acceso a la descarga del programa DEV C++