Curso de C++ por Antonio Lebrón Bocanegra

Este manual está extraido del paquete de software “Tutor C/C++ 1.0”, desarrollado por Antonio Lebrón Bocanegra como proyecto fin de carrera en la Facultad de Informática de Sevilla, y tutelado por Manuel Mejías Risoto. El paquete original era un programa para MsDos, que actuaba como lector paginado del texto del curso. Dicho paquete original no sólo incluía este texto sobre C++, sino otro similar sobre C, así como ejercicios de C y ejercicios de C++.

Tanto esta versión convertida a PDF como el curso original están disponibles en

www.nachocabanes.com/c/

LECCIÓN 1

INTRODUCCION AL CURSO DE C++

El objetivo de este curso es enseñar el lenguaje C++, o dicho de otro modo, enseñar a programar en lenguaje C++.

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- Idea general y origen del lenguaje C++.
- Programación orientada a objetos (OOP).
- Características principales de la OOP:
  * Encapsulación.
  * Herencia.
  * Polimorfismo
- Forma de implementar las clases en C++.

IDEA GENERAL DEL LENGUAJE C++

Aunque C es uno de los mejores lenguajes de programación de propósito general, a medida que un sistema software se va haciendo más grande se va acentuando más algunas deficiencias del C, como es la casi ilimitada libertad que tiene el programador sobre las rutinas que se implementan. El C++ soluciona este problema facilitando la creación de unidades funcionales de caja negra que tienen acceso estrictamente controlado; a estas unidades se les llama objetos, por este motivo se dice que el **C++ es un lenguaje orientado a objetos.**

ORIGEN

El C++ se llamó originalmente **C con clases** y fue desarrollado por Bjarne Stroustrup de los laboratorios Bell en Murray Hill, New Jersey, en 1983.

El C++ se puede considerar como una **ampliación del C estándar**, por lo que la mayor parte de lo que conocemos del C es aplicable al C++.

Es necesario dejar claro en este momento que este tutor enseña el lenguaje C++, no la programación orientada a objetos (OOP), aunque la mayoría de los programas de C++ implementados en este tutor son programas orientados a objetos. Como el lenguaje C se estudió en el tutor de C, **en este tutor estudiaremos todas las características específicas del C++**, es decir, aquellas características que posee el C++ y no las posee el C. Profundizaremos bastante en todas ellas de manera que no dejemos nada sin comentar. Como ya se dijo en el tutor de C, leyendo simplemente este tutor y libros de C++ no se aprende este lenguaje, sino que **es imprescindible e ineludible programar en C++**, y cuanto más se programe, mejor se captarán los conceptos y más rápido se aprenderá.

PROGRAMACION ORIENTADA A OBJETOS

Aunque acabamos de decir que el objetivo de este tutor es enseñar el lenguaje C++ y no la programación orientada a objetos, lo cual podría ocupar otro tutor completo ya que es otra forma de programar y de pensar en términos programáticos, sí es pertinente mostrar cuáles son las principales características de esta nueva metodología de la programación. Estos rasgos principales los podemos resumir en tres: **encapsulación**, **herencia y polimorfismo.**

ENCAPSULACION

La programación orientada a objetos (OOP) está basada en la manipulación de objetos. Un objeto es un concepto que alberga datos y funciones que operan con esos datos. Una vez que se ha implementado un objeto, lo único que necesita saber un programador es la interface con la cual comunicarse con él.

Para entender mejor la idea de objeto, supongamos que un reloj digital es un objeto. Los datos serían: la hora, la fecha, etc. La interface estaría formada por las funciones que manipulan esos datos: mostrar la hora, mostrar la fecha, cambiar la hora, cambiar la fecha, etc. Observad que el usuario no necesita saber cómo están implementadas tales funciones para manipular los datos; incluso se podría cambiar la implementación de estas funciones y la interface seguiría siendo la misma.

Un ejemplo de objeto en programación podría ser una pila. La interface estaría formada por dos funciones: apilar y desapilar. Al usuario del objeto no le importa cuáles son los datos del objeto ni cómo están implementadas estas dos funciones. El programador del objeto podría diseñar la pila como un array estático y cambiar posteriormente la implementación para que la pila esté diseñada como una lista dinámica enlazada. Al usuario de la pila no le afectaría en nada el cambio de diseño en la pila, puesto que la interface seguiría siendo la misma, esto es, formada por las funciones: apilar y desapilar.

Los objetos también reciben el nombre de tipo abstracto de datos (TAD, DAT en inglés).

Y la encapsulación recibe también el nombre de abstracción de datos.

HERENCIA

Una de las características fundamentales en los lenguajes orientados a objetos es la herencia. Gracias a esto **un determinado objeto puede "heredar" propiedades de otros objetos**. Estos vínculos nos permiten, en primer lugar, evitar informaciones duplicadas. Pero su importancia va mucho más allá, ya que el concepto de herencia implica una clasificación y una interdependencia entre los objetos. Veamoslo con un ejemplo.
Supongamos que definimos el objeto animal. Este objeto abstracto tiene una serie de características: es un ser vivo, necesita alimentarse, se reproduce, etc.

Acto seguido definimos el objeto mamífero indicándole que es un animal. Este objeto posee todas las características de los animales y además le añade nuevas características: es vertebrado, tiene mamas, etc.

A continuación definimos el objeto persona indicándole que es un mamífero por lo que herederá todas las características de los mamíferos y de los animales. Además les añadimos otras que son propias a las personas: habla, es racional, etc.

Por último, definimos el objeto alumno indicándole que es una persona. El objeto alumno tendrá todas las características de persona y tendrá también las suyas propias: matrícula, asignaturas que tiene, etc.

Finalmente, definimos Antonio como una instanciación del objeto alumno, es decir, concretamos el objeto abstracto alumno sobre un elemento real, Antonio; no necesitamos definir nada. El lenguaje ya "sabe" que Antonio es un ser vivo, es vertebrado, etc.

POLIMORFISMO

Vamos a explicar este concepto a través de dos ejemplos.

Imaginemos que tenemos dos pilas, una almacena números enteros y la otra almacena números en coma flotante. Las dos funciones básicas que podemos realizar con una pila son meter y sacar. Como tenemos dos pilas, podríamos implementar cuatro funciones diferentes con cuatro nombres diferentes: apilaint, desapilaint, apilafloat y desapilafloat. Aunque esto funcione, estamos complicando el programa a nivel conceptual. Utilizando la idea de polimorfismo podemos simplificar conceptualmente el programa implementando cuatro funciones con dos nombres diferentes: apila y desapila. Cuando invoquemos, por ejemplo, a la función apila, el compilador deberá determinar si llamamos a la función apila de la pila de números enteros o a la función apila de la pila de números en coma flotante. Las funciones apila y desapila se dice que están sobrecargadas.

Los operadores también se pueden sobrecargar en C++. De hecho ya están sobrecargados en casi todos los lenguajes de programación. Por ejemplo, el operador + lo utilizamos en C y C++ para sumar operandos que pueden ser del tipo carácter, entero, coma flotante, etc.
Veamos otro ejemplo interasente de polimorfismo que hace uso de la herencia.

Supongamos que tenemos definido el objeto punto con la función dibujar, la cual pinta un punto. A continuación definimos el objeto línea que hereda las características del objeto punto. En el objeto lína, utilizando la idea de polimorfismo, volvemos a definir la función dibujar, pero en este caso esta función traza una línea; la función dibujar del objeto línea puede utilizar, si lo desea, la función dibujar, heredada del objeto punto, para dibujar cada uno de los puntos de los que está constituida la línea.

FORMA DE IMPLEMENTAR LAS CLASES

Para terminar con esta introducción al lenguaje C++ vamos a hacer una recomendación. Todos los programas ejemplos de esta lección están en un sólo fichero. La mayoría de los ejemplos están constituidos por la definición de un tipo abstracto de datos (llamados clase en C++ así como a la instanciación de una clase se le llama objeto) y la función main para probar la clase definida. Los programas ejemplos se han realizado en un sólo fichero para facilitar su exposición y explicación en el tutor. No obstante, en la práctica es recomendable dividir el diseño de una clase en dos ficheros: un fichero sería de cabecera (con extensión .H) y en él estarían todas las declaraciones de la clase, y el otro fichero sería de código ejecutable (con extensión .CPP) y en él estarían todas las definiciones relativas a la clase declarada. Al principio del fichero que tiene las definiciones hacemos un #include del fichero que contiene las declaraciones.
Cuando queramos utilizar esa clase en un programa de C++ tenemos dos posibilidades:

1) Si el programa no lo hacemos como proyecto, simplemente hacemos un #include del fichero que contiene las definiciones de la clase al principio del fichero del programa.

2) Si el programa lo hacemos como proyecto, añadimos a la lista de ficheros del proyecto, el fichero de las definiciones de la clase, bien con extensión .CPP o bien con extensión .OBJ. Además hay que tener en cuenta que debemos hacer un #include del fichero con las declaraciones de la clase, en cada fichero del proyecto que utilice dicha clase.

LECCIÓN 2

C++ COMO UNA MEJORA DEL C

Esta lección explica las nuevas características de C++ no orientadas a objetos. Se trata de un importante número de pequeñas adiciones que hace el C++ sobre el C.

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- Nuevo estilo de comentario (con ****).
- Declaraciones (en cualquier sitio).
- Los nombres de struct y enum son tipos.
- Operador de resolución de ámbito (::).
- Declaraciones por referencia (con el operador **&**).
- Uniones anónimas (uniones sin nombre).
- Conversión de tipo explícita (notación funcional de los moldes).
- Funciones en línea (**inline**).
- Argumentos por defecto.

- Funciones sobrecargadas (antiguamente se especificaba con **overload**).
- Operadores de almacenamiento libre (**new** y **delete**).
- Entrada/Salida estándar (con los flujos **cout** y **cin**, y con los operadores sobrecargados **<<** y **>>**, que se encuentran declarados en los ficheros de cabecera **stream.h** y **iostream.h**).

COMENTARIOS

C++ introduce un nuevo estilo de comentario con el símbolo **//**.

Este símbolo indica comentario hasta final de línea.

Ejemplos:

```
int x; // a es una variable de tipo int
int y; /* y es una variable de tipo int */
```

DECLARACIONES

En C, las declaraciones locales han de ir inmediatamente después de las llaves de comienzo de bloque. En C++ se puede declarar una variable en cualquier sitio, existiendo desde el punto de declaración hasta el final del bloque en la que se ha declarado.

Ejemplo:

```
{
  int x;
  x = 10;
  int y; // esta línea no es correcta en C, pero sí en C++
  y = 20;
}
```

Una declaración muy frecuente en C++ es la siguiente:

```
for (int i = iinic; i <= ifin; i++)
  {
    sentencias
  }
```

La variable i sólo tiene existencia en el bloque en el que está declarada.

Ejemplo:

```
for (register int i = 0; i < IMAX; i++)
  {
    int t = v[i-1];
    v[i-1] = v[i];
    v[i] = t;
  }
```

En C++, los nombres de enum y de struct son tipos. Esto nos permite hacer los siguiente:

```
enum ecolores { rojo, verde, azul };
struct scolores { ecolores color; char nombre_color[20]; };
scolores color; // color es de tipo scolores
```

En C, las anteriores sentencias habría que hacerlas del siguiente modo:

```
enum ecolores { rojo, verde, azul };
```

```
struct scolores { enum ecolores color; char nombre_color[20]; };
struct scolores color; // color es de tipo struct scolores
```

OPERADOR DE RESOLUCION DE AMBITO (::)

C es un lenguaje estructurado en bloque. C++ hereda los mismos conceptos de bloque y ámbito.

El operador :: se usa del siguiente modo:

::variable

y tiene el significado de permitir el acceso a variable, que debe estar declarada externamente.

```
// Ejemplo del operador ::
// Este programa imprime  3 1 2 1

#include <stdio.h>

int i = 1; // i externa

void main (void)
{
  int i = 2; // i local a función main()
  {
    int i = 3; // i local al bloque en la que está declarada
    printf ("%d %d ", i, ::i); // imprime  3 1
  }
  printf ("%d %d ", i, ::i); // imprime  2 1
}
```

DECLARACIONES POR REFERENCIA Y LLAMADAS POR REFERENCIA

El operador **&** tiene en C++ un significado adicional a los que ya tiene en C: declarar una variable o un parámetro de una función como referencia de otra variable.

La forma general de declara una variable como referencia de otra es:

tipo & identificador = objeto

Ejemplo:

```
#include <stdio.h>
void main (void)
{
  int i;
  int& i1 = i;
  int& i2 = i1;
  int& i3 = i2;
  i = 1;
  printf ("%d ", i1); // imprime  1
  i2 = 2;
  printf ("%d ", i3); // imprime  2
}
```

Otros ejemplos:

```
double a[10];
```

```
  double& ultimo = a[9]; // ultimo es un alias para a[9]
  char& nueva_linea = '\n';
```

El nombre ultimo es una alternativa para el elemento del array a[9].

Estos nombres, una vez que son inicializados, no pueden ser cambiados.

También es posible inicializar una referencia a un literal, lo cual crea una referencia a una localización desconocida donde se almacena el literal.
El uso principal del operador & con el significado que acabamos de describir se da en los argumentos pasados por referencia. Observar estas dos versiones de la misma función:

```
  // estilo C                                  // estilo C++
  void intercambiar (int *px, int *py)         void intercambiar (int&x, int&y)
  {                                            {
    int aux = *px;                               int aux = x;
    *px = *py;                                   x = y;
    *py = aux;                                   y = x;
  }                                            }
```

Es obvio que la segunda versión de la función intercambiar() es mucho más clara que la primera.
También los valores devueltos por las funciones pueden ser por referencia. Como es de suponer, el valor devuelto ha de ser una referencia a una variable no local a la función que devuelve el valor. Ejemplo:

```
  #include <stdio.h>

  int& f (int &x)
  {
    return x;
  }

  void main (void)
  {
    int y;
    f(y) = 20; // en realidad el 20 se asigna a y
    printf ("%d", y); // imprime  20
  }
```

Estas dos funciones son incorrectas:

```
  int& f1 (void)
  {
    int x = 10;
    return x;
  }

  int& f2 (void)
  {
    return 10;
  }
```

El compilador debe informar de error de compilación en ambos return.

UNIONES ANONIMAS

Las uniones anónimas son las uniones que no tienen nombre. El C no dispone de ellas.

Ejemplo:

```
// C++                  // C
union                   union
  {                       {
    int d;                   int d;
    float f;                 float f;
  };                      } u;
f = 1.1;                u.f = 1.1;
printf ("%d", d);       printf ("%d", u.d);
```

CONVERSION DE TIPO EXPLICITA

En C++, el nombre de un tipo puede ser usado como una función para realizar una conversión de tipo. Esto supone una alternativa a los moldes.

Ejemplo:

```
#include <stdio.h>

void main (void)
{
   int i = 10;
   float f1 = i;          // conversión de tipo implícita
   float f2 = (float) i; // conversión de tipo explícita: notación molde
   float f3 = float (i); // conversión de tipo explícita: notación funcional

   printf ("%g %g %g", f1, f2, f3); // imprime  10 10 10
}
```

Otro ejemplo de conversión explí-
cita mediante notación funcional:

```
struct st { int d; float f; };
typedef st *pst;
char *str = "abcdef";
pst p = pst (str);
```

FUNCIONES INLINE

En el curso de C vimos que las macros pueden dar problemas como éste:

Dado
```
  #define CUAD(x) x*x
```
la sentencia
```
  CUAD(a+b)
```
expande a:
```
  a+b*a+b
```
que no es evidentemente la expresión esperada.

El problema puede ser evitado parentizando la definición de la macro.
Sin embargo, la solución no protege contra tipos impropios.
El C++ ofrece una alternativa elegante
y eficiente usando funciones inline:

```
inline int cuad (int x)
{
  return x*x;
}
```

La palabra clave **inline** le dice al com-

pilador que la función sea compilada
como una macro, es decir, el especi-
ficador inline fuerza al compilador de
C++ a sustituir el cuerpo de código de
cuad() en el lugar en que esta función
es invocada.
Aunque el uso de inline incrementa la velocidad
de ejecución porque se elimina la llamada a la
función, produce un incremento del tamaño del
código, especialmente si la función inline con-
tiene muchas líneas de código y es incovada mu-
chas veces en el programa.

Otro ejemplo de función inline:

```
inline void imprimir (int a, int b)
{
  printf ("\n%d", a);
  printf ("\n%d", b);
}
```

ARGUMENTOS POR DEFECTO

Uno o más argumentos en una función de C++ pueden ser especificado
teniendo valores por defecto.

Ejemplo:

```
#include <stdio.h>

int mult (int x, int y = 1)
{
  return (x * y);
}

void main (void)
{
  printf ("%d %d", mult (5, 6), mult (7)); // imprime  30 7
}
```

Sólo los parámetros finales de una función pueden tener
valores por defecto.

Ejemplo:

```
void f1 (int i, int j = 2);                // legal
void f2 (int i = 3, int j);                // ilegal
void f3 (int i, int j = 4, int k = 5);     // legal
void f4 (int i = 6, int j = 7, int k = 8); // legal
void f5 (int i, int j = 9, int k);         // ilegal
```

FUNCIONES SOBRECARGADAS

El término sobrecarga se refiere al uso del mismo nombre para varios
significados de un operador o una función. El significado seleccionado
depende de los tipos de los argumentos usados por el operador o la
función.

El primer estándar C++ introducía la palabra clave **overload** para in-
dicar que un nombre particular será sobrecargado. En el nuevo estándar

de C++ se considera obsoleto el uso de esta instrucción.

En este momento restringiremos nuestra discusión a la sobrecarga de funciones y dejaremos la sobrecarga de operadores para lecciones posteriores.

```
// Ejemplo de sobrecarga de funciones

overload media_array; // opcional en el nuevo estándar de C++
double media_array (double a[], int tam);
double media_array (int a[], int tam);

double media_array (int a[], int tam)
{
  int sum = 0;
  for (int i = 0; i < tam; ++i)
    sum += a[i]; // ejecuta aritmética de enteros
  return ((double) sum / tam);
}

double media_array (double a[], int tam)
{
  double sum = 0.0;
  for (int i = 0; i < tam; ++i)
    sum += a[i]; // ejecuta aritmética de double
  return (sum / tam);
}
```

El compilador elige automáticamente la función que coincide con los tipos de los argumentos.

Atención: No se puede sobrecargar dos funciones que tengan iguales tipos de argumentos y distintos tipos de valores devueltos. Es ilegal, por lo tanto, lo siguiente:

```
overload f;
int f (void);
double f (void);
```

OPERADORES DE ALMACENAMIENTO LIBRE (new y delete)

Los operadores unarios **new** y **delete** están disponibles para manipular el almacenamiento libre. Estos operadores reemplazan a las funciones de la biblioteca estándar malloc(), calloc() y free(). En C++ se recomienda usar estos operadores a tales funciones. El almacenamento libre se refiere al sistema por el cual el programador gestiona directamente el tiempo de vida de los objetos. El programador crea el objeto usando new y lo destruye usando delete. Esto es importante para estructuras de datos dinámicas como las listas y los árboles.

El operador new se puede usar de las siguientes formas:

```
new nombre_tipo
new nombre_tipo inicializador
new (nombre_tipo)
```

En cada caso se producen dos efectos. Primero, es asignada la cantidad apropiada de almacenamiento para contener al nuevo tipo. Segundo, la dirección base del objeto es devuelta como el valor de la expresión new. La expresión es de tipo void * y puede ser asignada a cualquier tipo puntero. Si no hay suficiente memoria, este operador devuelve nulo. El uso del operador con inicializador se aplica a las clases por

lo que se explica en posteriores lecciones. Después del tipo puede ir entre corchetes el número de elementos de ese tipo a reservar.

Ejemplos:

```
  int *pi;         char *pc;                float *pf; int n = 2;
  pi = new int;    pc = new char[10];       pf = new float[n];
  *pi = 5;         strcpy (pc, "hola");     pf[0] = 1.1; pf[1] = 2.2;
```

El operador delete destruye un objeto creado por new dejando libre el espacio ocupado por este objeto para poder ser reusado. El operador delete se puede usar de las siguientes formas:

```
  delete expresion
  delete [expresion] expresion
```

La primera forma es la más común. La expresión es normalmente una variable puntero usada en una expresión new previa. La segunda forma es utilizada menos frecuentemente y se usa cuando se asignó un array con new. La expresión entre corchetes especifica el número de elementos del array a liberar. Dicho tamaño del vector proporcionado por el usuario es ignorado excepto para algunos tipos definidos por el usuario (referencia: apartado "Vectores de objetos de clases" en lección 4 del tutor de C++). El operador delete no devuelve ningún valor (o también se puede decir que su tipo devuelto es void). El operador delete sólo puede ser aplicado a punteros devueltos por new o a cero; aplicar delete a cero no tiene ningún efecto.

ENTRADA/SALIDA ESTANDAR EN C++

La E/S estándar en C++ se va a estudiar en detalle en lecciones posteriores. Aquí sólo se va a comentar los conocimientos mínimos para poder utilizarla en los ejemplos.

En el primer estándar de C++, la utilización de las facilidades de E/S necesitaba la inclusión del fichero de cabecera **<stream.h>**. En el nuevo estándar es preferible utilizar el fichero de cabecera **<iostream.h>** que mejora y añade nuevas características a <stream.h>. Lo que se va a comentar en este momento requiere uno de los dos ficheros de cabecera anteriores; si el compilador que usas no contiene el fichero iostream.h, seguro que tiene el stream.h.

A partir de este momento, la salida estándar la realizaremos con el identificador **cout** y el operador sobrecargado **<<**, y la entrada estándar la realizaremos con el identificador **cin** y el operador sobrecargado **>>**. Para poder utilizar en un programa los identificadores cout, cin y sus correspondientes operadores sobrecargados, tenemos que hacer al principio de esto:

```
  #include <iostream.h> // para los compiladores que no dispongan de
                        // este fichero: #include <stream.h>

  // Ejemplo de cómo utilizar la E/S de C++

  #include <iostream.h>

  void main (void)
  {
    // equivalente a printf ("C++ es un C mejorado.\n");
    cout << "C++ es un C mejorado.\n";

    // equivalente a printf ("2 + 2 = %d\n", 2 + 2);
    cout << "2 + 2 = " << 2 + 2 << '\n';

    int n;
```

```
    cin >> n; // equivalente a scanf ("%d", &n);

    float f1, f2;
    cin >> f1 >> f2; // equivalente a scanf ("%f%f", &f1, &f2);
  }
```

Los identificadores cout y cin son los nombres de los flujos de salida y entrada estándar, respectivamente. Los operadores << y >> indican la dirección del flujo de información.

LECCIÓN 3

CLASES

Esta lección describe las facilidades de C++ para definir nuevos tipos en los cuales el acceso a los datos está restringido a un conjunto específico de funciones. A estos nuevos tipos se les denomina tipos abstractos de datos (TAD en castellano y ADT en inglés).

Una clase (class) es una extensión de la idea de estructura (struct) en C. El nombre original dado por Stroustrup a este lenguaje fue "C con clases".

Una estructura en C está compuesta por un conjunto de datos. Una clase o una estructura en C++ está compuesta de un conjunto de datos junto con un conjunto de funciones y operadores para manipular esos datos.

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- El tipo compuesto **struct**.
- Visibilidad de los miembros de un objeto (**private** y **public**).
- Tipos compuestos **struct** y **class**.
- Declaración de funciones miembros.
- Operador de resolución de ámbito (::).
- Miembro **static**.
- Clases anidadas.
- Estructuras y uniones (**struct** y **union**).
- Punteros a miembros (**::***, **.*** y **->***).
- Precedencia de operadores.

EL TIPO COMPUESTO struct

En C++, a los elementos de una estructura se les llama **miembros**. Además, extiende el concepto de estructura.

Para explicar estas mejoras vamos a hacer dos versiones de un mismo programa: la primera versión utilizando el concepto de struct del C y la segunda utilizando el concepto de struct del C++.

DOS VERSIONES DE UN MISMO PROGRAMA

// VERSION 1. UTILIZA CONCEPTO DE STRUCT DEL C

```
#include <iostream.h> // cout, << sobrecargado; también vale <stream.h>

const int longit_max = 1000;
enum boolean { false, true };

struct pila
  {
    char s[longit_max]; // no permitido en el ANSI C, sí en el ANSI C++
    int cima;
  };

void inicializar (pila *pil)
{
  pil->cima = 0;
}

void meter (char c, pila *pil)
{
  pil->s[++pil->cima] = c;
}

char sacar (pila *pil)
{
  return (pil->s[pil->cima--]);
}

char elem_cima (pila *pil)
{
  return (pil->s[pil->cima]);
}

boolean vacia (pila *pil)
{
  return boolean (pil->cima == 0);
}

boolean llena (pila *pil)
{
  return boolean (pil->cima == longit_max - 1);
}

void main (void)
{
  pila p;
  char *cad;

  inicializar (&p);
  cad = "Ejemplo de struct del C.";

  for (register int i = 0; cad[i]; i++)
    if (! llena (&p))
```

```
      meter (cad[i], &p);

  cout << "Cadena original: " << cad << "\n";
  cout << "Cadena invertida: ";
  while (! vacia (&p))
    cout << sacar (&p);
  cout << "\n";
}
```

// VERSION 2. UTILIZA CONCEPTO DE STRUCT DEL C++

```
#include <iostream.h> // cout, << sobrecargado; también vale <stream.h>

const int longit_max = 1000;
enum boolean { false, true };

struct pila
  {
    private:

      char s[longit_max]; // const no permitida en inicializac. en ANSI C
      int cima;

    public:

      void inicializar (void)
      {
        cima = 0;
      }

      void meter (char c)
      {
        s[++cima] = c;
      }

      char sacar (void)
      {
        return (s[cima--]);
      }

      char elem_cima (void)
      {
        return (s[cima]);
      }

      boolean vacia (void)
      {
        return boolean (cima == 0);
      }

      boolean llena (void)
      {
        return boolean (cima == longit_max - 1);
      }
  };


void main (void)
{
  pila p;
  char *cad;

  p.inicializar ();
  cad = "Ejemplo de struct del C++.";
```

```
  for (register int i = 0; cad[i]; i++)
    if (! p.llena ())
      p.meter (cad[i]);

  cout << "Cadena original: " << cad << "\n";
  cout << "Cadena invertida: ";
  while (! p.vacia ())
    cout << p.sacar ();
  cout << "\n";
}
```

SALIDA DE LOS PROGRAMAS:

La salida del primer programa es:

```
Cadena original: Ejemplo de struct del C.
Cadena invertida: .C led tcurts ed olpmejE
```

y la del segundo:

```
Cadena original: Ejemplo de struct del C++.
Cadena invertida: .++C led tcurts ed olpmejE
```

Los dos son iguales excepto en el mensaje que invierten.

ANALISIS DE LOS EJEMPLOS:

En nuestro primer programa no aparece ningún concepto nuevo y el usuario debe entenderlo todo. En el segundo programa sí aparecen unos cuantos conceptos nuevos del C++.

La primera novedad que se aprecia es que hay definida funciones dentro de la estructura. En C, las estructuras sólo pueden tener datos miembros, en C++, las estructuras también pueden contener **funciones miembros**.

Algunos libros se refieren a las funciones miembros como métodos. Las variables de tipo estructura reciben el nombre de objetos.

Como se aprecia en la función main(), a las funciones de una variable estructura se accede igual que a los datos de esa estructura, es decir, con el operador punto (.).

La segunda novedad que vemos en la estructura del ejemplo son las palabras claves **private** y **public**, las cuales hacen que los miembros de la estructura sean privados o públicos, respectivamente. A los miembros privados sólo pueden acceder las funciones miembros. A los miembros públicos puede acceder cualquier función del programa.

La última novedad que aparece en el programa es la definición implícita de funciones **inline:** todas aquellas funciones que se definen (no que se declaran) dentro de una estructura son inline, aunque no vayan precedidas por la palabra clave inline.

Estas tres novedades de las estructuras de C++ con respecto al C se van a discutir más ampliamente en los apartados siguientes. Intenta entender el segundo programa antes de pasar a las siguientes ventanas e intenta apreciar la diferencia que existe con respecto a la primera versión del mismo programa.

VISIBILIDAD DE LOS MIEMBROS DE UN OBJETO

Desde el punto de vista de la visibilidad o el acceso a los miembros de un objeto, los miembros pueden ser **públicos**, **privados** o **protegidos**.
Los miembros protegidos se estudian en lecciones posteriores.
Los miembros privados son aquéllos que están bajo el ámbito de la palabra clave private seguida por dos puntos (**private:**). Del mismo modo, los miembros públicos son aquéllos que están bajo el ámbito de la palabra clave public seguida por dos puntos (**public:**).
Los miembros privados sólo pueden ser accedidos por las funciones miembros.
Los miembros públicos constituyen la interface con los elementos de la clase.
Si todos los miembros de una clase, datos y funciones, son privados, no hay ninguna forma de que el programa se pueda comunicar con los elementos de ese objeto.

EJEMPLO SOBRE EL ACCESO A LOS MIEMBROS DE UNA CLASE

```
void main (void)
{
  struct
    {
      private:
        int x;
        void f1 (void) { x = 10; }
        void f2 (void) { y = 10; }

      public:
        int y;
        void f3 (void) { x = 20; }
        void f4 (void) { y = 20; }
        void f5 (void) { f1 (); f2 (); f3 (); }
    } s;

  s.x = 1; // ilegal: error de compilación. Miembro x no accesible.
  s.f1 (); // ilegal: error de compilación. Miembro f1() no accesible.
  s.f2 (); // ilegal: error de compilación. Miembro f2() no accesible.
  s.y = 2; // legal
  s.f3 (); // legal
  s.f4 (); // legal
  s.f5 (); // legal
}
```

TIPOS COMPUESTOS struct Y class

Los tipos struct y class son similares. Sólo hay una diferencia entre ellos: **los miembros de una estructura son por defecto públicos, y los miembros de una clase son por defecto privados.** Esto supone que las palabras claves private y public son opcionales.
En este tutor, por convención, los objetos que sólo contengan datos se declararán como struct, y aquéllos que contengan datos y funciones se declararán como class.

EJEMPLOS DE TIPOS EQUIVALENTES

struct s **struct s**

```
  {                                  {
    int i;                <===>        public:
    void f (void);                       int i;
  };                                     void f (void);
                                       };

class c                            class c
  {                                  {
    int i;                <===>        private:
    void f (void);                       int i;
  };                                     void f (void);
                                       };

struct s                           class c
  {                                  {
    int i;                <===>        public:
    void f (void);                       int i;
  };                                     void f (void);
                                       };

class c                            struct s
  {                                  {
    public:                              int i;
      int i;              <===>          void f (void);
      void f (void);                 };
  };

struct s                           class c
  {                                  {
    int i;                <===>        void f (void);

    private:                           public:
      void f (void);                     int i;
  }                                  };
```

DECLARACION DE FUNCIONES MIEMBROS

Antes de que se pueda utilizar una clase,
todos sus miembros deben estar definidos.

DOS FORMAS DE DEFINIR UNA FUNCION MIEMBRO

1) Definir la función miembro dentro de la clase. En este caso, aunque la definición de la función no vaya precedida por la palabra inline, la función es inline y será tratada como una macro. Naturalmente, se puede utilizar la palabra clave inline si se desea, aunque no sea necesario con las funciones miembros.

2) Declarar la función dentro de la clase, es decir, escribir su prototipo dentro de la clase, y definirla fuera, es decir, en el ámbito externo. En este caso, el nombre de la función miembro en la definición debe ir precedido con el nombre de la clase seguido por el operador de resolución de ámbito (::). El nombre de la clase es necesario porque puede haber otras clases que utilicen el mismo nombre de función.
Las funciones miembros, al igual, que las demás funciones, también se pueden sobrecargar y sus argumentos pueden tener valores por defecto.

EJEMPLO DE DECLARACION DE FUNCIONES MIEMBROS

```
#include <iostream.h>

class clase
  {
    private:
      int x;

    public:
      inline void inicializar (void); // declaración de función inline
      void asignar (int y) { x = y; } // definición de función inline
      int devolver (void);            // declaración de función no inline
  };

void clase::inicializar (void)
{
  x = 0;
}

int clase::devolver (void)
{
  return x;
}

void main (void)
{
  clase c;
  cout << (c.inicializar (), c.devolver ())
       << ' '
       << (c.asignar (100), c.devolver ())
       << '\n'; // imprime:  0 100
}
```

OPERADOR DE RESOLUCION DE AMBITO (::)

En la lección anterior dijimos que podíamos referirnos explicítamente a un miembro externo de la siguiente forma:

::identificador

Otra forma de utilizar este operador es:

clase::identificador

lo cual es útil para distinguir explicítamente entre los nombres de miembros de clase y otros nombres.

En las dos expresiones anteriores identificador puede ser tanto el nombre de una variable como el nombre de una función.

El operador unario :: tiene la misma prioridad que los operadores monarios **()**, **[]**, **->** y ., esto es, tiene la prioriodad más alta.

EJEMPLO DEL OPERADOR DE RESOLUCION DE AMBITO

```
#include <iostream.h>

int x;

class clase
  {
    private:
      int x;

    public:
      void asigx (void) { x = 2; }
      int devexpr (void) { return (x + clase::x + ::x); }
  };

void main (void)
{
  clase c;

  x = 1;
  c.asigx ();

  cout << c.devexpr () << " " << x + ::x; // imprime  5 2
}
```

MIEMBRO static

Los datos miembros pueden ser declarados con el modificador de clase de almacenamiento static. No pueden ser declarados auto, register o extern. **Un dato miembro que es declarado static es compartido por todas las variables de esa clase y es almacenado en un lugar únicamente.** A causa de esto, se accede a este miembro, desde fuera de la clase, en la forma nombre_de_clase::identificador si tiene visibilidad pública.

EJEMPLO DE MIEMBRO static

```
#include <iostream.h>

struct estructura
  {
    int x;
    static int y;
  };

int estructura::y = 0;

void main (void)
{
  estructura c1, c2, c3;

  estructura::y++;

  c1.y++;
  c2.y++;
  c3.y++;

  c1.x = c2.x = c3.x = 0;
```

```
  c1.x++;
  c2.x++;
  c3.x++;

  cout << estructura::y << c1.y << c2.y << c3.y; // imprime  4444
  cout << c1.x << c2.x << c3.x; // imprime  111
}
```

CLASES ANIDADADAS

Las clases se pueden anidar. **La clase interna no está dentro del ámbito de la clase externa, sino que está en el mismo ámbito que la clase externa.** Puesto que esto puede conducir a confusión, es preferible no utilizar clases anidadas.

EJEMPLO DE CLASES ANIDADAS

```
char c; // ámbito exterior
class x // declaración de clase exterior
  {
    char c;
    class y // declaración de clase interior
      {
        char d;
        void f (char e) { c = e; }
      };
    char g (x* q) { return q->d; } // error de sintaxis
  };
```

COMENTARIO SOBRE EL EJEMPLO ANTERIOR

La función miembro f de class y, cuando usa c, está usando la c de ámbito externo. Es como si class y estuviese declarada en el mismo nivel y el mismo bloque interior o ámbito de fichero que class x. La función miembro g de class x, cuando usa d, está intentando acceder al miembro privado de class x, lo cual produce un error de compilación.

ESTRUCTURAS Y UNIONES

Hemos dicho que una estructa es simplemente una clase con todos sus miembros públicos, esto es

```
struct s { ...
```

es una forma corta de escribir

```
class s { public: ...
```

Las estructuras se usan cuando no se quiere ocultar los datos.

Una unión (union) se define como una estructura (struct)

donde todos sus miembros tienen la misma dirección.
Los especificadores de acceso de C++ (public, private y protected) no pueden ser usados en las uniones.

EJEMPLO DE union

```
#include <iostream.h>

union un
  {
    int x;
    float y;

    void asigx (int xx) { x = xx; }
    void asigy (float yy) { y = yy; }
    int devx (void) { return x; }
    float devy (void) { return y; }
  };

void main (void)
{
  un u;

  u.x = 15;
  cout << u.devx () << '\n'; // imprime  15

  u.y = 5.5;
  cout << u.devy () << '\n'; // imprime  5.5

  u.x = 15;
  cout << u.devy () << '\n'; // imprime valor indefinido

  u.y = 5.5;
  cout << u.devx () << '\n'; // imprime valor indefinido
}

partir
```

PUNTEROS A MIEMBROS

Es posible tomar la dirección de un miembro de una clase.
Con los operadores de resolución de ámbito (::) y de contenido (*****) podemos declarar un puntero a un miembro de una clase.

Ejemplo:
  int cl::*pcl; // pcl es un puntero a un miembro entero de la clase cl
Los operadores . y * forman un nuevo operador en C++, **.***, con el cual referenciamos un miembro de una clase a través de un puntero al miembro de esa clase.

En la siguiente ventana se muestra un ejemplo del uso de este operador.

*EJEMPLO DEL OPERADOR .**

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

```
class cl
  {
    public:
      int suma;
      void cl::sumatorio (int x);
  };

void cl::sumatorio (int x)
{
  suma = 0;
  for (register int i = x; i; i--)
    suma += i;
}

void main (void)
{
  int cl::*pi;            // puntero a un miembro de cl que es de tipo int
  void (cl::*pf) (int x); // puntero a una función miembro de cl que
                          // devuelve void y acepta un int como parámetro
  cl clase;               // clase es un objeto de tipo cl

  pi = &cl::suma;         // obtiene dirección del dato miembro suma
  pf = &cl::sumatorio;    // obtiene dirección de la función miembro
                          // sumatorio

  (clase.*pf) (5);        // calcula el sumatorio de 5
  cout << "El sumatorio de 5 es " << clase.*pi << "\n";
}
```

Los operadores -> y * forman un nuevo operador en C++, **->***, con el cual referenciamos un miembro de una clase referenciando un puntero a un miembro de esa clase a través de un puntero a la clase.

En la siguiente ventana se muestra un ejemplo del uso de este operador.

*EJEMPLO DEL OPERADOR ->**

```
#include <iostream.h>

class cl
  {
    public:
      int suma;
      void cl::sumatorio (int x);
  };

void cl::sumatorio (int x)
{
  suma = 0;
  for (register int i = x; i; i--)
    suma += i;
}

void main (void)
{
  int cl::*pi;            // puntero a un miembro de cl que es de tipo int
  void (cl::*pf) (int x); // puntero a una función miembro de cl que
                          // devuelve void y acepta un int como parámetro
  cl clase;               // clase es un objeto de tipo cl
  cl *pcl;                // pcl es un puntero a un objeto del tipo cl

  pcl = &clase;           // asigna a pcl la dirección del objeto clase

  pi = &cl::suma;         // obtiene dirección del dato miembro suma
```

```
  pf = &cl::sumatorio;    // obtiene dirección de la función miembro
                          // sumatorio

  (pcl->*pf) (5);         // calcula el sumatorio de 5 usando -> para llamar
                          // a la función
  cout << "El sumatorio de 5 es " << pcl->*pi << "\n"; // usa -> para acceder
                                                       // a un dato miembro
}
```

PRECEDENCIA DE OPERADORES

Una vez vistos todos los operadores que añade el C++ al lenguaje C (new, (delete, ::, .*, ->*) vamos a mostrar la tabla de precedencia completa de los operadores de C++.

Precedencia de Operadores
==========

En la siguiente tabla de precedencia de operadores, los operadores son divididos en 16 categorías.

La categoría #1 tiene la precedencia más alta; la categoría #2 (operadores unarios) toma la segunda precedencia, y así hasta el operador coma, el cual tiene la precedencia más baja.

Los operadores que están dentro de una misma categoría tienen igual precedencia.

Los operadores unarios (categoría #2), condicional (categoría #14), y de asignación (categoría #15) se asocian de derecha a izquierda; todos los demás operadores se asocian de izquierda a derecha.

# Categoría	Operador	Qué es (o hace)
1. Más alto	()	Llamada a función
	[]	Indexamiento de array
	->	Selector de componente indirecta de C++
	::	Resolución/acceso de ámbito de C++
	.	Selector de componente directa de C++
2. Unario	!	Negación Lógica (NO)
	~	Complemento a 1
	+	Más unario
	-	Menos unario
	++	Preincremento o postincremento
	--	Predecremento o postdecremento
	&	Dirección
	*	Contenido (indirección)
	sizeof	(devuelve tamaño de operando, en bytes)
	new	(asignador de memoria dinámica en C++)
	delete	(desasignador de memoria dinámica en C++)
3. Multiplicativo	*	Multiplica
	/	Divide
	%	Resto (módulo)
4. Acceso a los miembros	.*	Operador de referencia de una dirección en C++.
	->*	Operador de referencia de una dirección en C++.

5. Aditivo	+	Más binario
	-	Menos binario
6. Desplaza-miento	<<	Desplazamiento a la izquierda
	>>	Desplazamiento a la derecha
7. Relacional	<	Menor que
	<=	Menor o igual que
	>	Mayor que
	>=	Mayor o igual que
8. Igualdad	==	Igual a
	!=	Distinto a
9.	&	AND entre bits
10.	^	XOR entre bits
11.	|	OR entre bits
12.	&&	AND lógico
13.	||	OR lógico
14. Condicional	?:	(a ? x : y significa "si a entonces x, si no y")
15. Asignación	=	Asignación simple
	*=	Asignar producto
	/=	Asignar cociente
	%=	Asignar resto (módulo)
	+=	Asignar suma
	-=	Asignar diferncia
	&=	Asignar AND entre bits
	^=	Asignar XOR entre bits
	|=	Asignar OR entre bits
	<<=	Asignar desplazamiento hacia la izquierda
	>>=	Asignar desplazamiento hacia la derecha
16. Coma	,	Evaluar

Todos los operadores de la tabla se pueden sobrecargar (referencia de sobrecarga de operadores en lección 5 del tutor de C++) excepto los siguientes:

.	Selector de componente directa de C++
.*	Referencia en C++
::	Resolución/acceso de ámbito en C++
?:	Condicional

LECCIÓN 4

CONSTRUCTORES Y DESTRUCTORES

Esta lección gira en torno a los constructores y destructores.

Un constructor es una función miembro de un objeto que es llamada automáticamente cuando se crea el objeto. Tiene el mismo nombre de la clase.

Un destructor es una función miembro de un objeto que es llamada automáticamente cuando se destruye el objeto. Tiene el mismo nombre de la clase precedido por .

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- Concepto de constructor y destructor.
- Constructores y destructores de objetos estáticos.
- Almacenamiento libre.
- Vectores de objetos de clase.
- Objetos como miembros.
- Autorreferencia: el puntero this.
- Funciones miembros constantes.
- Funciones miembros volátiles.

CONCEPTO DE CONSTRUCTOR Y DESTRUCTOR

Un **constructor** es una función miembro que tiene el mismo nombre que la clase. Los constructores se utilizan normalmente para inicializar datos miembros y asignar memoria usando new. Un **destructor** es una función miembro que tiene el mismo nombre que la clase precedido por el carácter . Los destructores se utilizan normalmente para liberar, usando delete, la memoria asignada por el constructor.

Los constructores pueden sobrecargarse y pueden tomar argumentos; en los destructores no se permite ninguna de estas dos cosas. Los constructores y los destructores no devuelven nada.
Los constructores son invocados automáticamente cuando se crea el objeto. Los destructores son invocados automáticamente cuando se destruye el objeto.

EJEMPLO DE CONSTRUCTOR

```
#include <iostream.h>

class clase
  {
    int x;

    public:
      clase (int i) { x = i; }
      int devx (void) { return x; }
  };

void main (void)
```

```
{
  clase c1; // error de compilación
  clase c2 (2);
  cout << c2.devx (); // imprime  2
}
```

EJ. DE CONSTR. CON ARGS. POR DEFECTO

```
#include <iostream.h>

class clase
  {
    int x;

    public:
      clase (int i = 0) { x = i; }
      int devx (void) { return x; }
  };

void main (void)
{
  clase c1, c2 (1);
  cout << c1.devx (); // imprime  0
  cout << c2.devx (); // imprime  1
}
```

EJEMPLO DE CONSTRUCTOR SOBRECARGADO

```
#include <iostream.h>

class clase
  {
    int x;

    public:
      clase (void) { x = 0; }
      clase (int i) { x = i; }
      int devx (void) { return x; }
  };

void main (void)
{
  clase c1;
  clase c2 (5);
  clase c3 (); // atención: esto es el prototipo de una función
  cout << c1.devx (); // imprime  0
  cout << c2.devx (); // imprime  5
  cout << c3.devx (); // error de compilación: c3 no es objeto sino func.
}
```

EJEMPLO DE CONSTRUCTOR Y DESTRUCTOR

```
#include <iostream.h> // para utilizar: cout, << sobrecargado
```

```
enum boolean { false, true };

class pila
  {
    private:    // representación oculta para el TAD pila

      char *s;
      int longit_max;
      int cima;

    public:    // interface pública para el TAD pila

      pila (void) { s = new char[100]; longit_max = 100; cima = 0; }
      pila (int tam) { s = new char[tam]; longit_max = tam; cima = 0; }
      pila (int tam, char str[]);
      ~pila (void) { delete s; }
      void inicializar (void) { cima = 0; }
      void meter (char c) { s[++cima] = c; }
      char sacar (void) { return (s[cima--]); }
      char elem_cima (void) { return (s[cima]); }
      boolean vacia (void) { return boolean (cima == 0); }
      boolean llena (void) { return boolean (cima == longit_max - 1); }
  };

pila::pila (int tam, char str[])
{
  s = new char[tam];
  longit_max = tam;
  for (register int i = 0; i < longit_max && str[i] != 0; i++)
    s[i+1] = str[i];
  cima = i;
}

void main (void)
{
  char *cad = "Ejemplo";
  pila p (20, cad);

  cout << "Cadena original: " << cad << "\n"; // imprime  Ejemplo
  cout << "Cadena invertida: ";
  while (! p.vacia ())                        // imprime  olpmejE
    cout << p.sacar ();
  cout << "\n";
}
```

CONSTRUCTORES Y DESTRUCTORES DE OBJETOS ESTATICOS

En algunas implementaciones está indefinido si el constructor para un objeto estático local es llamado o no en la función que es declarado. Esto quiere decir que los argumentos de los constructores para objetos estáticos deben ser expresiones contantes en tales implementaciones. Por ejemplo:

```
  void f (int a)
  {
    static clase c (a); // error en algunos sistemas
  }
```

EJEMPLO DE FUNCIONAMIENTO DE OBJETOS ESTATICOS LOCALES EN TURBO C++

```
#include <iostream.h>

class clase
  {
    public:
      int x;
      clase (int i) { x = i; }
  };

void f (int a)
{
  static clase c (a); // el constructor es invocado en primera llamada a f()
  cout << c.x;
}

void main (void)
{
  f (10); // imprime  10
  f (20); // imprime  10
}
```

Si un programa termina usando la función exit(), se llamará a los destructores para los objetos estáticos, pero si el programa termina usando la función abort(), no se llamará a tales destructores. Notad que esto implica que exit() no termina un programa inmediatamente. LLamar a exit() en un destructor puede causar una recursión infinita.

ALMACENAMIENTO LIBRE

Consideremos el ejemplo:

```
  #include <iostream.h>

  class clase
    {
      char *s;

      public:
        clase (int n) { s = new char[n]; }
        ~clase (void) { delete s; }
    };

  void main (void)
  {
    clase *p = new clase (10);
    clase *q = new clase (20);
    delete p;
    delete p; // posiblemente un error en tiempo de ejecución
  }
```

El constructor clase::clase() será llamado dos veces, así como también el destructor clase::~clase(). El C++ no ofrece ninguna garantía de que el destructor será llamado para un objeto creado con new, a no ser que se utilice delete. El programa anterior no libera nunca q, pero libera dos veces p. Dependiendo del tipo de p y de q, el programador puede o no considerar esto un error. No liberar un objeto no es normalmente un error, sino que se considera malgastar memoria. Liberar dos veces p es

normalmente un error serio. El resultado típico de aplicar delete al mismo puntero dos veces es un bucle infinito en la rutina de gestión de almacenamiento libre, pero el comportamiento en este caso no está especificado por la definición del lenguaje y depende de la implementación.

VECTORES DE OBJETOS DE CLASE

Para declarar un vector de objetos de una clase con un constructor, esa clase debe tener un constructor que pueda ser llamado sin una lista de argumentos. Ni siquiera pueden ser usados los argumentos por defecto.

EJEMPLO DE VECTORES DE OBJETOS

```
#include <iostream.h>

class cl1
  {
    int x;

    public:
      cl1 (int i) { x = i; }
      int devx (void) { return x; }
  };

class cl2
  {
    int x;

    public:
      cl2 (int i = 0) { x = i; }
      int devx (void) { return x; }
  };

class cl3
  {
    int x;

    public:
      cl3 (void) { x = 0; }
      cl3 (int i) { x = i; }
      int devx (void) { return x; }
  };

class cl4
  {
    int x;

    public:
      void asigx (int i) { x = i; }
      int devx (void) { return x; }
  };

cl1 vcl1[10]; // ilegal: error en compilación
cl2 vcl2[10]; // ilegal: error en compilación
cl3 vcl3[10];
cl4 vcl4[10];
```

```
void main (void)
{
  vcl4[6].asigx(1);
  cout << vcl3[5].devx() << " " << vcl4[6].devx(); // imprime  0 1
}
```

Para vectores que no son asignados usando new, el destructor es llamado implícitamente para cada elemento del vector cuando ese vector es destruido. Sin embargo, esto no se puede hacer implícitamente para vectores creados con new, ya que el compilador no puede distinguir el puntero a un simple objeto del puntero al primer elemento de un vector de objetos.

EJEMPLO DE LLAMADAS A DESTRUCTORES EN VECTORES DE OBJETOS

```
#include <iostream.h>

const int n = 3;

class clase
  {
    public:
      static int contc, contd;
      clase (void) { cout << 'c' << ++contc << ' '; }
      ~clase (void) { cout << 'd' << ++contd << ' '; }
  };

int clase::contc, clase::contd;

void f (void)
{
  clase *p = new clase;
  clase *q = new clase[n];
  delete p; // una clase
  delete q; // problema: n clases
}

void g (void)
{
  clase *p = new clase;
  clase *q = new clase[n];
  delete p;
  delete [n] q;
}

void h (void)
{
  clase c1;
  clase vc2[n];
}

void main (void)
{
  clase::contc = clase::contd = 0;
  cout << '\n';
  f (); // imprime  c1 c2 c3 c4 d1 d2

  clase::contc = clase::contd = 0;
  cout << '\n';
  g (); // imprime  c1 c2 c3 c4 d1 d2 d3 d4
```

```
  clase::contc = clase::contd = 0;
  cout << '\n';
  h (); // imprime  c1 c2 c3 c4 d1 d2 d3 d4
}
```

OBJETOS COMO MIEMBROS

Los objetos (ya sean clases, estructuras o uniones), al igual que cualquier variable, pueden ser miembros de otros objetos.

Los constructores de los objetos miembros pueden ser invocados entre el nombre de la definición (no declaración) del constructor del objeto y la llave de apertura de la definición de la función constructora. Si se van a invocar varios constructores de objetos miembros, estas llamadas se separan por comas.

EJEMPLO DE OBJETOS COMO MIEMBROS

```
/*
  Este programa imprime:
  c1_1  c1_2  c1_3  c2_1  x2_10  x1_10  x1_10  x1_10  d2_1  d1_1  d1_2  d1_32
*/

#include <iostream.h>

class cl1
  {
    static int contc1, contd1;
    int x1;

    public:
      cl1 (int i) { x1 = i; cout << "c1_" << ++contc1 << "  "; }
      ~cl1 (void) { cout << "d1_" << ++contd1 << "  "; }
      void impr (void) { cout << "x1_" << x1 << "  "; }
  };

int cl1::contc1 = 0, cl1::contd1 = 0;

class cl2
  {
    static int contc2, contd2;
    int x2;
    cl1 a, b, c;

    public:
      cl2 (int i): a (i), b (i), c (i)
        { x2 = i; cout << "c2_" << ++contc2 << "  "; }
      ~cl2 (void) { cout << "d2_" << ++contd2 << "  "; }
      void impr (void) { cout << "x2_" << x2 << "  "; }
      void impra (void) { a.impr (); }
      void imprb (void) { b.impr (); }
      void imprc (void) { c.impr (); }
  };

int cl2::contc2 = 0, cl2::contd2 = 0;

void main (void)
{
```

```
  cl2 c (10);
  c.impr ();
  c.impra ();
  c.imprb ();
  c.imprc ();
}

/*
  Si no queremos que el constructor cl2 sea inline, haríamos:

  class cl2
    {
      ...
      public
        cl2 (int i);
      ...
    };

  cl2::cl2 (int i): a (i), b (i), c (i)
  {
    x2 = i;
    cout << "c2_" << ++contc2 << "  ";
  }
*/
```

AUTORREFERENCIA: EL PUNTERO this

La palabra clave **this** denota un puntero autorreferencial declarado implícitamente. Puede ser usado en cualquier función miembro.
A todas las funciones miembros de un objeto se les pasa un argumento oculto: this. Este argumento implícito siempre apunta al objeto para el cual se ha invocado la función miembro.

EJEMPLO DEL PUNTERO this

```
#include <iostream.h>

class x
  {
    int a, b;

    public:
      x (int aa, int bb) { a = aa; b = bb; }
      int leera (void) { return a; } // devuelve el valor de x::a
      int leerb (void) { return this->b; } // devuelve el valor de x::b
  };

void main (void)
{
  x x (1, 2);
  cout << x.leera () << x.leerb (); // imprime  12
}
```

FUNCIONES MIEMBROS CONSTANTES

Una función miembro constante no modifica los datos miembros de un objeto.

Una función miembro constante es el único tipo de función miembro que puede ser llamada por un objeto constante.

Una función miembro constante puede ser llamada también por objetos no constantes.

No se puede calificar una definición de función con const y entonces intentar modificar los datos miembros (esto generará un error).

Para crear una función miembro constante hay que colocar el calificador const después de la lista de argumentos y antes de abrir la llave de definición de la función.

Las funciones miembros constantes sólo pueden llamar a otras funciones miembros constantes.

Los constructores y los destructores no necesitan ser declarados funciones miembros constantes para que sean invocados por objetos constantes.

EJEMPLO DE USO CORRECTO Y INCORRECTO DE FUNCIONES MIEMBROS CONSTANTES

```
// Ejemplo de funciones miembros constantes y objetos constantes

class clase
  {
    private:
      int x;

    public:
      clase (void) { x = 0; }

      ~clase () { }
      int dev (void) const { return x; }   // CORRECTO: función constante no
                                           // modifica ningún miembro
      void incr (void) { x++; }            // CORRECTO: función miembro
                                           // ordinaria (no constante)

/*
      clase (int i) const { x = i; }       // ERROR: función constante
                                           // modificando miembro
      void incr2 (void) const { x++; }     // ERROR: función constante
                                           // modificando miembro
      void incr3 (void) const { incr (); } // ERROR: función constante
                                           // llama a función no constante
*/
  };

void main (void)
{
  const clase cl1; // cl1 es un objeto constante

  cl1.dev ();      // CORRECTO: función constante llamada por objeto
                   // constante
/*
  cl1.incr ();     // ERROR: función no constante llamada por objeto
                   // constante
*/

  clase cl2;       // cl2 es un objeto ordinario (no constante)
  cl2.dev ();      // CORRECTO: función constante llamada por objeto no
                   // constante
```

```
  cl2.incr ();      // CORRECTO: función no constante llamada por objeto no
                    // constante
}
```

FUNCIONES MIEMBROS VOLATILES

Las funciones miembros volátiles son análogas a las funciones miembros constantes, y son creadas de la misma forma, colocando el calificador volatile después de la lista de argumentos y antes del cuerpo de la función.

Sólo funciones miembros volátiles pueden ser llamadas por objetos volátiles, y las funciones miembros volátiles pueden sólo llamar a otras funciones miembros volátiles.

Al contrario que las funciones miembros constantes, las funciones miembros volátiles pueden modificar los datos miembros de un objeto.

Los constructores y los destructores no necesitan ser declarados funciones miembros volátiles para ser invocados por objetos volátiles.

Una función miembro puede ser constante y volátil.

EJEMPLO DE USO CORRECTO Y INCORRECTO DE FUNCIONES MIEMBROS VOLATILES

```
// Ejemplo de funciones miembros volátiles y objetos volátiles

class clase
  {
    private:
      int x;

    public:
      clase (void) { x = 0; }                 // constructor no volátil
      clase (int i) volatile { x = i; }       // constructor volátil

      ~clase () { }                           // destructor no volátil
      int dev (void) volatile { return x; }   // función miembro volátil
      void incr (void) { x++; }               // función miembro no volátil
/*
      void incr2 (void) volatile { incr (); } // ERROR: función volátil
                                              // llama a función no volátil
*/
  };

void main (void)
{
  volatile clase cl1; // cl1 es un objeto volátil

  cl1.dev ();       // CORRECTO: función volátil llamada por objeto
                    // volátil
/*
  cl1.incr ();      // ERROR: función no volátil llamada por objeto
                    // volátil
*/
  clase cl2;        // cl2 es un objeto ordinario (no volátil)
  cl2.dev ();       // CORRECTO: función volátil llamada por objeto no
                    // volátil
  cl2.incr ();      // CORRECTO: función no volátil llamada por objeto no
```

```
                    // volátil
}
```

ALGUNAS DECLARACIONES DE CLASES CURIOSAS

```
#include <iostream.h>

class clase1
  {
    public:
      int i;
      float f;
      int devi (void) { return i; }
      float devf (void) { return f; }
  };

void f (void)
{
  clase1 c = { 10, 20.35 };
  cout << c.devi () << ' ' << c.devf () << '\n'; // imprime  10 20.35
}

class clase2
  {
   int a, b;

    public:
      clase2 (int i, int j): a (i), b (i + j) { }
      int deva (void) { return a; }
      int devb (void) { return b; }
  };

void g (void)
{
  clase2 c (1, 2);
  cout << c.deva () << ' ' << c.devb () << '\n'; // imprime  1 3
}

class clase3
  {
    int a, b;

    public:
      clase3 (int i, int j): a (i) { b = j; }
      int deva (void) { return a; }
      int devb (void) { return b; }
  };

void h (void)
{
  clase3 c (2, 3);
  cout << c.deva () << ' ' << c.devb () << '\n'; // imprime  2 3
}

class clase4
  {
    public:
      int d;
      float f;
      int devd (void) const volatile { return d; }
      float devf (void) volatile const { return f; }
  };
```

```
void i (void)
{
  clase4 c = { 2, 3.4 };
  cout << c.devd () << ' ' << c.devf () << '\n'; // imprime  2 3.4
}

void main (void)
{
  f ();
  g ();
  h ();
  i ();
}
```

LECCIÓN 5

CONVERSIONES Y SOBRECARGA DE OPERADORES

En esta lección se estudia todo lo referente a las conversiones de tipos (tanto explícitas como implícitas) y la sobrecarga de operadores (los operadores sobrecargados son una forma corta de invocar a las funciones de operadores).

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- **Conversiones tradicionales:** notación molde y conversiones implícitas.
- **Conversiones de TAD's:** notación funcional y conversiones implícitas.
- **Algoritmo de selección de función sobrecargada:** tres reglas.
- **Especificador friend:** hace funciones y clases amigas de una clase.
- **Sobrecarga de operadores:** palabra clave operator.
- **Objetos grandes:** pasarlos por referencia en los argumentos.
- **Asignación e inicialización:** dos conceptos diferentes.
- **Ejemplo de sobrecarga de algunos operadores:** (), [], ,, new y delete.

CONVERSIONES TRADICIONALES

Las conversiones tradicionales del C++ son las conversiones de tipo que realiza el C, y que por lo tanto, también las realiza el C++.

Las reglas generales de conversión son las siguientes:

Conversión automática en una expresión aritmética	**x op y**

```
|=============================================================================|
|                                                                             |
|  Primero:                                                                   |
|    Todo char y short son convertidos a int.                                 |
|    Todo unsigned char y unsigned short son convertidos a unsigned.          |
|                                                                             |
|  Segundo:                                                                   |
|    Si después del primer paso, los dos operandos de la expresión son de     |
|    tipo diferente, entonces de acuerdo a la jerarquía de tipos              |
|      int < unsigned < long < unsigned long < float < double                 |
|    el operando de tipo más bajo es convertido al tipo del operando de       |
|    tipo más alto y el valor de la expresión es de ese tipo.                 |
|                                                                             |
-------------------------------------------------------------------------------
```

EJEMPLOS DE CONVERSIONES

```
char c; short s; unsigned u; int i; long l; float f; double d;
     Expresión     Tipo             Expresión     Tipo
     -----------   -------          -----------   --------
     c - s / i     int              u * 3 - i     unsigned
     u * 3.0 - i   double           f * 3 - i     float
     c + 1         int              3 * s * l     long
     c + 1.0       double           d + s         double
```

CONVERSIONES DE TAD

El C++ introduce la notación funcional para hacer conversiones explícitas de tipo.

Una notación funcional de la forma

```
  nombre_tipo (expresion)
```

es equivalente a la notación molde

```
  (nombre_tipo) (expresion)
```

Así, estas dos expresiones son equivalentes:

```
  x = float (i);
  x = (float) i;
```

En C++ es preferible utilizar la notación funcional.

Un constructor con un argumento es de hecho una conversión de tipo del tipo del argumento al tipo de la clase del constructor.

EJEMPLO

```
// Ejemplo de conversión de un tipo ya definido a un tipo definido
// por el usuario.

#include <iostream.h>

class clase
  {
```

```
    int x;

    public:
      clase (void) { x = 0; }
      clase (int i) { x = i; }
      int devx (void) { return x; }
      void imprx (void) { cout << devx () << '\n'; }
  };

void main (void)
{
  clase c1;
  clase c2 (1);
  clase c3 = clase (2); // conversión de tipo explícita
  clase c4 = 3; // conversión de tipo implícita: clase c4 = clase (3);
  clase c5 = c4; // no hay conversión de tipo

  c1.imprx (); // imprime  0
  c2.imprx (); // imprime  1
  c3.imprx (); // imprime  2
  c4.imprx (); // imprime  3
  c5.imprx (); // imprime  3
}
```

También es posible hacer conversiones de un tipo definido por el usuario a un tipo definido ya.

La forma de hacer esto es incluir funciones miembros de la forma:

```
operator tipo () { ... }
```

EJEMPLO

```
// Ejemplo de conversión de un tipo definido por el usuario a un
// tipo definido ya

#include <iostream.h>

class clase
  {
    int x;

    public:
      clase (int i) { x = i; }
      operator int () { return x; }
  };

inline void impr (int ent) { cout << ent << '\n'; }

void main (void)
{
  clase c1 = clase (2); // conversión de tipo explícita
  clase c2 = 3; // conversión de tipo implícita: clase c4 = clase (3);
  clase c3 = c1; // no hay conversión de tipo

  int a = 1;          // no hay conversión de tipo
  int b = int ('a'); // conversión de tipo explícita
  int c = 'A';        // conversión de tipo implícita: int c = int ('A');
  int d = int (c2);  // conversión de tipo explícita
  int e = c3;         // conversión de tipo implícita: int e = int (c3)
  int f = int (clase (4)); // dos conversiones de tipo explícita
```

```
  impr (a); // imprime  1
  impr (b); // imprime  97 . Nota: el valor ASCII de 'a' es 97.
  impr (c); // imprime  65 . Nota: el valor ASCII de 'A' es 65.
  impr (d); // imprime  3
  impr (e); // imprime  2
  impr (f); // imprime  4
}
```

ALGORITMO DE SELECCION DE FUNCION SOBRECARGADA

1. **Usa un ajuste exacto si se encuentra.**
   Un short, un char, o una constante 0 es un ajuste exacto para int.
   Un float es un ajuste exacto para double.

2. **Intenta conversiones implícitas de tipos estándares y usa cualquier ajuste.**
   Estas conversiones no deben perder información.
   Dichas conversiones incluyen conversiones de puntero y las siguientes extensiones: int a long y int a double.

3. **Intenta conversiones definidas por el usuario y usa un ajuste para el cual hay un único conjunto de conversiones.**

EJEMPLO

```
// Ejemplo de conversiones de tipos y de selección de funciones
// sobrecargadas

#include <iostream.h>

class clase
  {
    int x;

    public:
      clase (int i = 0) { x = i; }
      operator int () { return x; }
  };

inline void impr (int ent) { cout << ent << '\n'; }

inline int max (int a, int b) { return a > b ? a : b; }

// conversión implícita: { return int (a) > int (b) ? a : b; }
inline clase max (clase a, clase b) { return a > b ? a : b; }

void main (void)
{
  clase c1 = 2;         // conversión implícita: clase c1 = clase (2);
  clase c2 = clase (3); // conversión explícita

  impr (max (4, 3));   // imprime  4

  // conversión explícita
  impr (int (max (c1, c2))); // imprime  3 .

  // conversión implícita: impr (int (max (c1, c2)));
  impr (max (c1, c2)); // imprime  3 .
```

```
  // conversión explícita en max
  impr (max (int (c1), 10)); // imprime  10 .

  // conversión explícita en max
  impr (max (c1, clase (10))); // imprime  10 .

/*
  impr (max (c1, 10));

  Esta sentencia provoca el error de compilación:
  "Ambigüedad entre max(clase,clase) y max(int,int)."

  Se viola la tercera regla de las tres que acabamos de mencionar ya
  que el compilador no sabe qué función escoger entre max(clase,clase)
  y max(int,int) para realizar la conversión de tipos.
*/

  // conversión implícita: impr (int (max (c1, c2)) + int (max (5, 6)));
  impr (max (c1, c2) + max (5, 6)); // imprime  9

  // conversión implícita: int (c2).
  // La conversión explícita int (2) es redundante.
  impr (int (2) + 3 * c2); // imprime  11

  // conversiones implícitas: impr (int (c1 = clase (int (c2) + 2)));
  impr (c1 = c2 + 2); // imprime  5
}
```

ESPECIFICADOR friend

La palabra clave friend (amigo/a) es un especificador de función o de clase. **El especificador friend permite que una función no miembro de una clase X, u otra clase Y, puedan acceder a los miembros privados de la clase X.**

FUNCION friend

La declaración de la función amiga debe aparecer en el interior de la definición de la clase. La declaración de la función amiga debe empezar con la palabra clave friend. Esta declaración puede aparecer en la parte pública o en la parte privada de la clase sin afectar a su significado.

EJEMPLO

```
// Ejemplo de función friend

#include <iostream.h>

class clase
  {
    friend void asig (clase& cl, int i);
    friend int dev (clase cl) { return cl.x; }

    int x;

    public:
      clase (int i = 0) { x = i; }
```

```
      int devx (void) { return x; }
  };

inline void asig (clase& cl, int i) { cl.x = i; }

void main (void)
{
  clase c (5);
  cout << dev (c) << '\n'; // imprime  5
  asig (c, -2);
  cout << c.devx () << '\n'; // imprime  -2
}
```

FUNCION MIEMBRO friend

Las funciones miembros de una clase Y pueden ser funciones amigas de otra clase X. En este caso es necesario utilizar el operador de resolución de ámbito en la declaración de la función amiga dentro de la clase X.

EJEMPLO

```
// Ejemplo de función miembro friend

#include <iostream.h>

class clase_B; // declarac. (no definic.) para que sea conocida en clase_A

class clase_A
  {
    public:
      int dev (clase_B cl);
  };

class clase_B
  {
    friend int clase_A::dev (clase_B cl);

    int x;

    public:
      clase_B (int i = 0) { x = i; }
      int devx (void) { return x; }
  };

inline int clase_A::dev (clase_B cl) { return cl.x; }

void main (void)
{
  clase_B c1 (5);
  clase_A c2;
  cout << c1.devx () << '\n'; // imprime  5
  cout << c2.dev (c1) << '\n'; // imprime  5
}
```

CLASE friend

Si todas las funciones miembros de una clase son funciones amigas de otra

clase, se dice que una clase es amiga (friend) de otra. La declaración es
  friend class nombre_de_clase
en lugar de
  friend declaracion_de_funcion

EJEMPLO

```
// Ejemplo de clase friend

#include <iostream.h>

class clase_B; // declarac. (no definic.) para que sea conocida en clase_A

class clase_A
  {
    friend clase_B;

    int x;

    public:
      clase_A (int i = 0) { x = i; }
      int devx (void) { return x; }
  };

class clase_B
  {
    public:
      int dev (clase_A cl) { return cl.x; }
  };

void main (void)
{
  clase_A c1 (5);
  clase_B c2;
  cout << c1.devx () << '\n'; // imprime  5
  cout << c2.dev (c1) << '\n'; // imprime  5
}
partir
```

SOBRECARGA DE OPERADORES

Los operadores se pueden sobrecargar al igual que las funciones. La utilización de operadores sobrecargados conduce a programas más cortos y más legibles.
Hemos utilizado anteriormente la palabra clave **operator** para definir una función miembro que realiza la conversión de tipos. También puede usarse esta palabra clave para sobrecargar operadores de C++.
Aunque se sumen nuevos significados a los operadores, su asociatividad y precedencia se conserva. Tampoco se puede cambiar la sintaxis de operadores, por ejemplo, no es posible definir un operador unario % o uno binario !.

Los operadores que no pueden ser sobrecargados son:
  - operador de selector de componente (.)
  - operador de referencia a través de un puntero a un miembro (.*)
  - operador de resolución/acceso de ámbito (::)
  - operador condicional (?:)

Los operadores de autoincremento y autodecremento, ++ y --, se pueden sobrecargar pero sus significados en notación prefija y postfija es el mismo en la forma sobrecargada.

Todos los demás operadores se pueden sobrecargar, éstos incluyen:

- operadores aritméticos
- operadores lógicos
- operadores de comparación
- operadores de igualdad
- operadores de asignación
- operadores de bits
- operador de indexado []
- llamada a función ()
- operadores new y delete

No es posible definir nuevos tokens de operadores, pero se puede usar la notación de llamada a función cuando este conjunto de operadores es inadecuado. Por ejemplo, usa pow() en vez de **.

El ; no se puede sobrecargar puesto que no es un operador.

FUNCION OPERADOR (operator)

El nombre de una función operador es la palabra clave operator seguida por el operador. Por ejemplo, operator<<. Una función operador es declarada como cualquier otra función y por lo tanto puede ser llamada como cualquier otra; el uso del operador es solamente una forma corta de escribir una llamada explícita de una función de operador. Por ejemplo, definida una clase complex en la cual se ha definido un operador de suma, las dos inicializaciones siguientes son sinónimas.

```
void f (complex a, complex b)
{
  complex c = a + b;            // forma corta
  complex d = operator+ (a, b), // llamada explícita
}
```

OPERADORES BINARIOS Y UNARIOS

Hay dos formas de sobrecargar los operadores: haciendo que sean funciones miembros o haciendo que sean funciones amigas.

Un operador binario puede ser definido bien usando una función que toma un argumento o bien usando una función amiga que toma dos argumentos. Así, para cualquier operador binario @, a@b puede ser interpretado bien como a.operator@(b) o bien como operator@(a,b). Si ambos están definidos, a@b es un error.

Un operador unario, ya sea prefijo o postfijo, puede ser definido bien usando una función miembro que no toma ningún argumento o bien usando una función amiga que toma un argumento. Así, para cualquier operador @, tanto a@ como @a puede ser interpretado bien como a.operator@() o bien como operator@(a). Si ambos están definidos, a@ y @a son errores. Considerar este ejemplo:

```
class X
  {
    // funciones amigas:
    friend X operator- (X);     // menos unario
    friend X operator- (X,X);   // menos binario
    friend X operator- ();      // error: no hay operando
    friend X operator- (X,X,X); // error: ternario

    // funciones miembros (con primer argumento implícita: this):
    X *operator& ();   // unario & (dirección de)
    X operator& (X);   // binario & (and a nivel de bits)
```

```
      X operator& (X,X); // error: ternario
    };
```

EJEMPLO

```
// Ejemplo de operadores sobrecargados.

#include <iostream.h>

class clase
  {
    friend clase operator- (clase c1, clase c2);
    friend clase operator- (clase cl) { return 0 - cl; }
    friend clase operator/ (clase, clase);

    private:

    int x, y;

    public:

    clase (int xx = 0, int yy = 0) { x = xx; y = yy; }
    int getx (void) { return x; }
    int gety (void) { return y; }
    clase operator+ (clase cl) { return *this - -cl; }
    clase operator+ (void) { return *this; }
    clase operator* (clase);
  };

inline clase operator- (clase c1, clase c2)
{
  return clase (c1.getx () - c2.getx (), c1.gety () - c2.gety ());
}

inline clase operator/ (clase c1, clase c2)
{
  return clase (c1.getx () / c2.getx (), c1.gety () / c2.gety ());
}

inline clase clase::operator* (clase cl)
{
  return clase (this->getx () * cl.getx (), this->gety () * cl.gety ());
}

inline void escribir (clase c)
{
  cout << c.getx () << ' ' << c.gety () << '\n';
}

void main (void)
{
  clase c1 (5), c2 (2, 3);

  escribir (c1 - c2);           // imprime  -3 -3
  escribir (c1 + c2);           // imprime  7 3
  escribir (-c1 - +c2);         // imprime  -7 -3
  escribir (c1 / c2 + c2 * c1); // imprime  12 0

  clase c3 = c1 + c2 + c2;
  escribir (c3 * c3);           // imprime  81 36
}
```

SIGNIFICADOS PREDEFINIDOS PARA LOS OPERADORES

No se hace ninguna asunción acerca del significado de un operador definido por el usuario. En particular, puesto que un = sobrecargado no se asume para implementar la asignación del segundo operando al primer operando, no se testea para asegurar que el primer operando es un lvalue (un lvalue es una dirección que aparece en el lado izquierdo de una sentencia de asignación y a la cual se le puede dar un valor). El significado de algunos operadores ya construidos están definidos para ser equivalentes a alguna combinación de otros operadores sobre los mismos argumentos. Por ejemplo, si a es un int, ++a significa a+=1, y esto significa a su vez a=a+1. En los operadores definidos por el usuario tales equivalencias no son ciertas (a no ser que el usuario defina estos operadores de esa forma). **Los operadores = y & tienen significados predefinidos cuando se aplican a objetos de clases.** No hay una forma elegante de "indefinir" estos dos operadores. Puede ser, sin embargo, inhabilitados para una clase X. Se puede declarar, por ejemplo, X::operator& () sin proporcionar una definición para él. Si en algún lugar se toma la dirección de un objeto de la clase X, el enlazador detectará la ausencia de la definición. En algunos sistemas esta técnica no puede ser usada porque el enlazador es tan "inteligente" que detecta que hay una función declarada y no definida aun cuando tal función no se use. Por ello, una solución alternativa es definir la función X::operator&() para que provoque un error en tiempo de ejecución.

UNA CLASE DE NUMEROS COMPLEJOS (clase complex)

Las funciones de operadores, al igual que todas las funciones, se pueden sobrecargar. Por ejemplo:

```
  class complex
    {
      double re, im;

      public:

      complex (double r, double i) { re = r; im = i; }

      friend complex operator+ (complex, complex);
      friend complex operator+ (complex, double);
      friend complex operator+ (double, complex);

      friend complex operator- (complex, complex);
      friend complex operator- (complex, double);
      friend complex operator- (double, complex);
      complex operator- ();

      friend complex operator* (complex, complex);
      friend complex operator* (complex, double);
      friend complex operator* (double, complex);

      // ...
    };

  void f (void)
  {
    complex a (1, 1), b (2, 2), c (3, 3), d (4, 4), e (5, 5);
    a = -b - c;
    b = c * 2.0 * c;
    c = (d + e) * a;
  }
```

Para evitar escribir tres funciones sobrecargadas para cada

operador binario, declaramos un constructor que, dado un double cree un complex. Por ejemplo:

```
  class complex
    {
      // ...

      complex (double r) { re = r; im = 0; }
    };
```

Ahora podemos hacer:

```
  complex z1 = complex (23);
  complex z2 = 23;
```

donde, tanto z1 como z2, serán inicializados llamando a complex(23,0).

La clase complex la podemos declarar de esta nueva forma:

```
  class complex
   {
     double re, im;

     public:

     complex (double r, double i = 0) { re = r; im = i; }

     friend complex operator+ (complex, complex);
     friend complex operator* (complex, complex);
   };
```

y deberían ser legales las operaciones que involucren variables complex y constantes enteras. Una constante entera será interpretada como complex con la parte imaginaria cero. Por ejemplo,

```
  a = b * 2
```

significa

```
  a = operator+ (b, complex (double (2), double (0)))
```

Una conversión definida por el usuario es aplicada implícitamente únicamente si es única.

Un objeto construido por el uso explícito o implícito de un constructor es automático y será destruido en la primera oportunidad, normalmente inmediatamente después de la sentencia en la cual fue creado.

OPERADORES # Y ## DEL PREPROCESADOR

No se pueden sobrecargar los operadores # y ## porque son operadores del preprocesador de C++, no del lenguaje C++. Los operadores almohadilla (#) y doble almohadilla (##) ejecutan sustituciones y fusionado de tokens en la fase de exploración del preprocesador.

El operador unario # convierte (sustituye) el token de su operando en string. El operador binario ## fusiona dos tokens (operando izquierdo y operando derecho en un sólo token.

El símbolo # también se utiliza para indicar una directiva del prepro-

cesador, pero en este caso no actúa como operador.

El siguiente ejemplo muestra cómo se utilizan los operadores # y ## del preprocesador.

EJEMPLO

```
// Ejemplo de los operadores # y ## del preprocesador de C++.

#include <iostream.h>

#define impr_suma(x)  cout << #x " + " #x " = " << x + x << "\n"
#define var(x,y) (x##y)

void main (void)
{
  int x1;
  var (x, 1) = 5;
  cout << "Ejemplo" " de los " "operadores # y ## del
          "preprocesador:\n";
  impr_suma (x1);
  impr_suma (100);
}

/*
SALIDA DEL PROGRAMA:

Ejemplo de los operadores # y ## del preprocesador:
x1 + x1 = 10
100 + 100 = 200
*/
```

OBJETOS GRANDES

Cuando se pasa un objeto como **argumento por valor** a una función, se está pasando, en realidad, una copia de ese objeto. En objetos de cierto tamaño es deseable evitar tanto traspaso de información. Esto se consigue pasando los objetos como **argumento por referencia** a las funciones. Para ello utilizamos el operador de referencia **&**.

ASIGNACION E INICIALIZACION

Consideremos una clase string muy simple:

```
  struct string
    {
      char *p;
      int size; // tamaño del vector apuntado por p

      string (int sz) { p = new char[size = sz]; }
      string () { delete p; }
    };
```

Un string es una estructura de datos compuesta de un puntero a un vector de caracteres y el tamaño de ese vector. El vector es creado por el constructor y borrado por el destructor. Sin embargo, tal y como está cons-

truida la clase string puede causar problemas. Por ejemplo:

```
void f (void)
{
  string s1 (10);
  string s2 (20);
  s1 = s2;
}
```

asignará dos vectores de caracteres, pero la asignación s1 = s2 hará que el puntero p de s1 apunte al mismo sitio que el puntero p de s2. Cuando sean invocados los destructores s1 y s2, se destruirá dos veces el mismo vector con resultados desastrosos impredecibles. La solución a este problema es definir apropiadamente la asignación de los objetos string.
La asignación por defecto entre objetos es la copia de elementos de uno a otro. Definamos un nuevo operador de asignación a nuestra clase string para resolver el problema planteado:

```
struct string
  {
    char *p;
    int size; // tamaño del vector apuntado por p
    string (int sz) { p = new char[size = sz]; }
    string () { delete p; }
    void operator= (string&);
  };

void string::operator= (string& a)
{
  if (this == &a) return; // previene s = s
  delete p;
  p = new char[size = a.size];
  strcpy (p, a.p);
}
```

Con la nueva definición de string aseguramos que el ejemplo anterior (función f()) trabajará correctamente. Sin embargo, todavía hay un problema latente que lo podemos observar si hacemos una pequeña modificación en f():

```
void f (void)
{
  string s1 (10);
  string s2 = s1;
}
```

Ahora sólo un string es construido pero dos son destruidos. Los operadores de asignación definidos por el usuario no son aplicados a objetos no inicializados. Un rápido vistazo a string::operator=() muestra porqué esto es así: el puntero p contendría un valor indefinido (aleatorio). Consecuentemente, debemos definir otra función miembro adicional para hacer frente a la inicialización.
La nueva definición de string para tener en cuenta la inicialización es:

```
struct string
  {
    char *p;
    int size; // tamaño del vector apuntado por p
    string (int sz) { p = new char[size = sz]; }
    string () { delete p; }
    void operator= (string&);
  };

void string::string (string& a)
{
```

```
    p = new char[size = a.size];
    strcpy (p, a.p);
  }
```

Para un tipo X, el constructor X(X&) proporciona la inicialización para un objeto del mismo tipo X. Las operaciones de asignación y de inicialización son diferentes; pero sólo tiene especial importancia tal diferencia cuando una clase X tiene un destructor que realiza tareas no triviales, tales como desasignación de memoria, en cuyo caso es deseable evitar la copia elemento a elemento de objetos:

```
  class X
    {
      // ...
      X (algo);       // constructor: crea objetos
      X (X&);         // constructor: copia en inicialización
      operator= (X&); // asignación: borrado y copia
      X ();          // destructor: borrado
    };
```

Hay dos casos más en los que un objeto es copiado: como un argumento de función y como un valor devuelto por una función. En ambos casos se crea una variable que es inicializada con el objeto del argumento o con el objeto a devolver; en ambos casos se llamará, por tanto, a la función X(&X) si ésta está definida:

```
  string g (string arg)
  {
    return arg;
  }

  void main (void)
  {
    string s = "asdf";
    s = g (s);
  }
```

Claramente, el valor de s debería ser "asdf" después de la llamada a g(). El argumento arg se convierte en una copia del valor de s llamando a string::string(string&). El valor devuelto por g() es una copia del valor de arg creada llamando a string::string(string&); esta vez, la variable inicializada es una variable temporal, la cual es asignada a s. Tales variables temporales son destruidas, usando string::~string(), tan pronto como es posible.

EJEMPLO DE SOBRECARGA DE ALGUNOS OPERADORES

```
#include <iostream.h> // cout
#include <stddef.h>   // size_t

class clase
  {
    public:

    int x, y, sum, dif, prod;

    clase (int i, int j): x (i), y (j), sum (i + j),
                          dif (i -j), prod (x * y) {}

    void escr (const char *s)
    {
      cout << "\n" << s << ": x = " << x << " y = " << y << " sum = " << sum
           << " dif = " << dif << " prod = " << prod;
    }
```

```
    operator int () { return dif; }

    int operator () (void) { return prod; }

    int operator [] (int ind)
      { return ind == 0 ? sum : ind == 1 ? prod : 0; }

    // cambia prioridad de operador , para la clase
    clase operator, (clase) { return *this; }

    void *operator new (size_t)
    {
      cout << "\nERROR: no se puede aplicar el operador new a un objeto "
              "de clase.";
      return 0;
    }

    void operator delete (void *) { }
  };

void main (void)
{
  clase cl1 (1, 2), cl2 (2, 3), cl3 (3, 4);
  int a = cl1, b = cl2 (), c = cl3 [1];

  cout << "\n** Inicializando: clase cl1 (1, 2), cl2 (2, 3), cl3 (3, 4);"
          "\n                 int a = cl1, b = cl2 (), c = cl3 [1];";
  cl1.escr ("cl1");
  cl2.escr ("cl2");
  cl3.escr ("cl3");
  cout << "\na = " << a << " b = " << b << " c = " << c;

  a = (b, c, 5);
  cl1 = (cl2, cl3, clase (5, 5));

  cout << "\n** Ejecutando: a = (b, c, 5); cl1 = (cl2, cl3, clase (5, 5));";
  cout << "\na = " << a;
  cl1.escr ("cl1");

  cout << "\n** Ejecutando: clase *pclase = new clase;";
  clase *pclase = new clase (0, 0);
  cout << "\n** Ejecutando: delete *pclase;";
  delete pclase;

  cout << "\n";
}

/*

SALIDA DE ESTE PROGRAMA:

** Inicializando: clase cl1 (1, 2), cl2 (2, 3), cl3 (3, 4);
                  int a = cl1, b = cl2 (), c = cl3 [1];
cl1: x = 1 y = 2 sum = 3 dif = -1 prod = 2
cl2: x = 2 y = 3 sum = 5 dif = -1 prod = 6
cl3: x = 3 y = 4 sum = 7 dif = -1 prod = 12
a = -1 b = 6 c = 12
** Ejecutando: a = (b, c, 5); cl1 = (cl2, cl3, clase (5, 5));
a = 5
cl1: x = 2 y = 3 sum = 5 dif = -1 prod = 6
** Ejecutando: clase *pclase = new clase;
ERROR: no se puede aplicar el operador new a un objeto de clase.
** Ejecutando: delete *pclase;

*/
```

LECCIÓN 6

HERENCIA SIMPLE

Esta lección describe los conceptos de herencia y clases derivadas en C++. Herencia es el mecanismo de derivar una nueva clase de otra vieja clase. Esto es, una clase puede adquirir todas las características de otra clase. A la nueva clase se le llama clase derivada y a la vieja clase se le llama clase base. Las clases derivadas proporcionan un mecanismo simple, flexible y eficiente para especificar una interface alternativa para una clase y para definir una clase añadiendo facilidades a una clase que ya existe sin reprogramación ni recompilación. La herencia es una característica importante de la programación orientada a objeto.

En los primeros estándares de C++, una clase derivada sólo podía heredar la descripción de una clase base (herencia simple). En los nuevos estándares, una clase derivada puede tener varias clases bases (herencia múltiple). En esta lección estudiaremos la herencia simple y en la siguiente veremos la herencia múltiple.

INDICE DE ESTA LECCION

****

En esta lección se va a estudiar los siguientes puntos:

- **Clases derivadas:** son clases que derivan de otras clases llamadas bases.
- **Modificadores de acceso public, private y protected:** estudio completo.
- **Punteros a clases derivadas:** clase derivada es un subtipo de clase base.
- **Funciones virtuales:** funciones declaradas con el especificador virtual.
- **Funciones puras:** son clases virtuales sin definición en clases bases.
- **Clases abstractas:** son las clases que tienen al menos una función pura.
- **Jerarquía de clases:** una clase derivada puede ser también una clase base.

****

CLASES DERIVADAS

Una clase puede ser derivada de una clase existente usando la foma:

```
(class | struct) nombre_clase_derivada: [public | private] nombre_clase_base
  {
    declaracion_de_los_miembros
  };
```

donde (class | struct) quiere decir que en ese lugar debe ir obligatoriamente la palabra clave **class** o la palabra clave **struct**; y [public | private] quiere decir que en ese lugar puede ir opcionalmente los modificadores de acceso **public** o **private**. Si no aparece ninguno de los dos modificadores de acceso, se considerará, por defecto, public para struct y private para class

Por ejemplo,

```
  struct d : b { ...
```
significa
```
  class d : public b { public: ...
```
y viceversa,
```
  class b : b { ...
```
significa
```
  struct b: private b { private: ...
```
Ejemplo de clase derivada:

```
  class empleado
    {
      public:
        char *nombre;
        short edad;
        short departamento;
        int salario;
        void imprimir (void);
    };

  class director: public empleado
    {
      public:
        int num_empleados;
        short tipo_director;
    };
```

La clase director es una clase derivada de la clase empleado, y por consiguiente, la clase empleado es una clase base para la clase director. La clase director tiene todos los miembros públicos de la clase empleado (nombre, edad, etc.) que son añadidos a los dos que ya posee la clase director. Por lo tanto, la clase empleado tiene cinco miembros y la clase director tiene siete.

Si no utilizáramos la herencia, haríamos:

```
  class director
    {
      public:
        empleado empl;
        int num_empleados;
        short tipo_director;
    };
```

Esta segunda versión de la clase director no es equivalente a la primera. En la primera versión, la clase director tiene siete miembros, en la segunda tiene tres, donde el miembro empl tiene a su vez cinco miembros. Desde el punto de vista lógico, en esta segunda versión no podemos decir que un director es un empleado, sino más bien que empleado es una parte de director. Hay otras diferencias como las conversiones de punteros a las clases empleado y director que se explicarán más adelante.

En el ejemplo de los directores y empleados, la forma natural de implementarlo en C++ es utilizando la herencia puesto que un director es un empleado y posee todas las características de los empleados.

MODIFICADORES DE ACCESO public, private Y protected

Los miembros de una clase pueden adquirir atributos de acceso de dos formas: por defecto, o a través del uso de los especificadores de acceso public, private y protected. La forma de utililizar estos especificadores es:

public: <declaraciones>
private: <declaraciones>

protected: <declaraciones>

Los especificadores de acceso (también llamados modificadores de visibilidad) pueden ser usados, dentro de una declaración de clase, en cualquier orden y en cualquier frecuencia. El estilo usual es:

```
class clase
  {
    private: // opcional
    ...
    protected:
    ...
    public:
    ...
  };
```

Si un miembro es **public**, puede ser usado por cualquier función. En C++, los miembros de un struct o union son public por defecto. Se puede cambiar el acceso de struct por defecto con private y protected; no se puede cambiar el acceso de union por defecto.

Si un miembro es **private**, sólo puede ser usado por funciones miembros y amigas de la clase en la que está declarada. Los miembros de una clase son private por defecto.

Si un miembro es **protected**, su acceso es el mismo que para private. Pero además el miembro puede ser usado por funciones miembros y amigas de las clases derivadas de la clase declarada, pero sólo en objetos del tipo derivado.

Las declaraciones de **friend** no son modificadas por estos especificadores de acceso.
Una clase base puede ser public o private con respecto a una clase derivada. Si la clase base es public, los miembros public y protected de la clase base conservan la misma visibilidad en la clase derivada. Si la clase base es private, los miembros public y protected de la clase base son miembros private en la clase derivada. Una clase base en C++ puede ser public o private, pero no protected. El siguiente esquema resume todo lo dicho en este párrafo:

Acceso en clase base	Modificador de acceso	Acceso heredado de base
public	public	public
private	public	no accesible
protected	public	protected
public	private	private
private	private	no accesible
protected	private	private

```
// Ejemplo sobre los especificadores de acceso en la herencia de C++

class base
{
   private:
     int base_priv;

   protected:
     int base_prot;

   public:
     int base_publ;
};

class derpubl : public base
{
   private:
```

```
     int derpubl_priv;

   protected:
     int derpubl_prot;

   public:
     int derpubl_publ;
};

class derpriv : private base
{
   private:
     int derpriv_priv;

   protected:
     int derpriv_prot;

   public:
     int derpriv_publ;
};

void main (void)
{
  base cbase;
  derpubl cderpubl;
  derpriv cderpriv;

  cbase.base_priv = 0;       // error
  cbase.base_prot = 0;       // error
  cbase.base_publ = 0;       // correcto

  cderpubl.base_priv = 0;    // error
  cderpubl.base_prot = 0;    // error
  cderpubl.base_publ = 0;    // correcto
  cderpubl.derpubl_priv = 0; // error
  cderpubl.derpubl_prot = 0; // error
  cderpubl.derpubl_publ = 0; // correcto

  cderpriv.base_priv = 0;    // error
  cderpriv.base_prot = 0;    // error
  cderpriv.base_publ = 0;    // error
  cderpriv.derpriv_priv = 0; // error
  cderpriv.derpriv_prot = 0; // error
  cderpriv.derpriv_publ = 0; // correcto
}
```

La clase derivada tiene sus propios constructores, los cuales invocarán a los constructores de la clase base.

Hay una sintaxis especial para pasar argumentos desde el constructor de la clase derivada al constructor de la clase base:

cabecera_funcion: nombre_clase_base (lista_de_argumentos)

En esta sintaxis nombre_clase_base es opcional pero es aconsejable ponerlo siempre.

```
// Ejemplo de clase derivada y clase base con constructores.

#include <iostream.h>

class base
{
   private:
     int b;
```

```
   public:
     base (int i = 0) { b = i; }
     int devb (void) { return b; }
};

class derivada : public base
{
  private:
    int d;

  public:
    derivada (int i = 0): base (i) { d = i; }
    derivada (int i, int j);
    int devd (void) { return d; }
};

derivada::derivada (int i, int j): base (j) { d = i; }

void main (void)
{
  base cb (2);
  derivada cd (3, 4);
  cout<<cb.devb()<<' '<<cd.devd()<<' '<<cd.devb(); // imprime  2 3 4
}
```

Los miembros de una clase derivada pueden tener los mismos nombres de su clase base, pero serán miembros diferentes aunque tengan los mismos nombres.

En este caso se puede distinguir entre los miembros de una clase y otra utilizando el operador de resolución de ámbito (::).

```
// Ejemplo de miembros con el mismo nombre en clases base y derivada.

#include <iostream.h>

class base
{
   protected:
     int x;

   public:
     base (int i = 0) { x = i; }
     int dev (void) { return x; }
};

class derivada : public base
{
  protected:
    int x;

  public:
    derivada (int i = 0): base (3 * i) { x = i; }
    int dev (void) { return x; }
    int f (void) { return x + derivada::x + base::x; }
    int g (void) { return dev() + derivada::dev() + base::dev(); }
};

void main (void)
{
  derivada clase (2);
  cout << clase.dev() << ' ' << clase.derivada::dev() << ' '
       << clase.base::dev() << ' ' << clase.f () << ' ' << clase.g ();
  // imprime:  2 2 6 10 10
```

```
}
```

El orden de las llamadas de los constructores es: constructor de la clase base, constructores de los miembros, y constructor de la propia clase derivada.

Los destructores son invocados en el orden contrario: destructor de la propia clase derivada, destructor de los miembros, y destructor de la clase base.

```
// Ejemplo sobre el orden de llamada de los constructores y los
// destructores.

#include <iostream.h>

class a
  {
    public:
      a () { cout << "\nConstructor de la clase a"; }
      ~a () { cout << "\nDestructor de la clase a"; }
  };

class b
  {
    public:
      b () { cout << "\nConstructor de la clase b"; }
      ~b () { cout << "\nDestructor de la clase b"; }
  };

class c : public a
  {
    private:
      b claseb;
    public:
      c () { cout << "\nConstructor de la clase c"; }
      ~c () { cout << "\nDestructor de la clase c"; }
  };

void main (void)
{
  c clasec;
}

/*
  SALIDA DEL PROGRAMA:

  Constructor de la clase a
  Constructor de la clase b
  Constructor de la clase c
  Destructor de la clase c
  Destructor de la clase b
  Destructor de la clase a
*/
```

Hemos dicho que cuando una clase derivada tiene una clase base privada, todos los miembros protegidos y públicos de la clase base son privados en la clase derivada. A veces es deseable que algunos de estos miembros sean públicos o protegidos en la clase derivada pero manteniendo la clase base privada. La forma de hacerlo se ilustra en el siguiente ejemplo.

```
// Ejemplo sobre la conversión de la visibilidad de los miembros de
// la clase base dentro de la clase derivada.

class b
  {
```

```
    public: int x, y, z, f(), g(), h();
  };

class d : private b
  {
    private:   int b::x, b::f; // opcional: esta línea es redundante
    protected: int b::y, b::g; // y y g son convertidos a protected
    public:    int b::z, b::h; // z y y son convertidos a public
  };

void main (void)
{
  b cb;
  d cd;
  cb.x = cb.y = cb.z = 0; // correcto
  cd.x = 0;               // error: d::x no es accesible
  cd.y = 0;               // error: d::y no es accesible
  cd.z = 0;               // correcto
}
```

De la misma forma, **un miembro transmitido públicamente puede ser convertido explícitamente a private o protected.**

```
// Ejemplo sobre la conversión de la visibilidad de miembros transmitidos
// públicamente de la clase base a la clase derivada.

class b
  {
    private:   int x, f();
    protected: int y, g();
    public:    int z, h();
  };

class d : public b
  {
    public:    int b::x, b::f;// x y f convertidos de private a public
    private:   int b::y, b::g;// y y g convertidos de protected a private
    protected: int b::z, b::h;// z y h convertidos de public a protected
  };

void main (void)
{
  b cb;
  d cd;

  cb.x = 0; // error: b::x no es accesible
  cd.x = 0; // correcto

  cb.z = 0; // correcto
  cd.z = 0; // error: d::z no es accesible
}
partir
```

PUNTEROS

Una clase derivada públicamente es un subtipo de su clase base.

Si una clase derivada tiene una clase base pública, entonces un puntero a la clase derivada puede ser asignado a una variable de tipo puntero a la clase base sin hacer uso de la conversión de tipo explícita. Por ejemplo:

```
  void main (void)
  {
    class base { /* ... */ };
    class derivada: public base { /* ... */ };
    derivada d;
    base *pb = &d;        // conversión implícita
    derivada *pd = pb;    // error: un base* no es un derivada*
    pd = (derivada *) pb; // conversión explícita
  }
```

En otras palabras, un objeto de una clase derivada puede ser tratado como un objeto de su clase base cuando se manipula a través de punteros. Lo contrario no es cierto.

```
\         \         \         \         /         /         /         /
  \         \         \         \      /         /         /         /
    \         \         \         \  /         /         /         /
      \         \         \                 /         /         /
        \         \         \               /         /         /
\         \         \         \         /         /         /         /
  \     Cuando la clase base es privada para la clase derivada,      /
    \   no se hace conversión implícita de derivada* a base*.      /
      \ Una conversión implícita no puede ser ejecutada en este  /
        caso porque un miembro público de la clase base puede
      / ser accedido a través de un puntero a la clase base pero \
    /   no a través de un puntero a la clase derivada.            \
  /         /         /         /      \         \         \         \
/         /         /         /         \         \         \         \
        /         /         /              \         \         \
      /         /         /                  \         \         \
    /         /         /         /  \         \         \         \
  /         /         /         /      \         \         \         \
/         /         /         /         \         \         \         \
// Ejemplo de puntero a clases base y derivada.

class base
  {
    int b1;
    public: int b2; // b2 es un miembro público de base
  };

class derivada: base
  {
    // b2 NO es un miembro público de derivada
  };

void main (void)
{
  derivada d;

// Para probar error, escribir en esta línea: #define PROBAR_ERROR

#ifdef PROBAR_ERROR
  d.b2 = 2;         // error: b2 de clase base privada
  base *pb = &d;    // error (base privada)
#else
  base *pb;
#endif

  pb->b2 = 2;       // correcto
  pb = (base *) &d; // correcto: conversión explícita
  pb->b2 = 2;       // correcto
}
\---------------------------------|---------------------------------/
||\-------------------------------|-------------------------------/||
||||\-----------------------------|-----------------------------/||||
||||||\---------------------------|---------------------------/||||||
```

```
||||||||\------------------------|------------------------/||||||||
||||||||||\----------------------|----------------------/||||||||||
|||||  Entre otras cosas, el ejemplo visto muestra que usando   |||||
|||||  conversión de tipo explícita se pueden romper las reglas |||||
-----  de protección. Está claro que no se recomienda hacer tal -----
|||||  cosa en un programa a menos que se tenga la intención de |||||
|||||  hacerlo ilegible.                                        |||||
||||||||||/----------------------|----------------------\||||||||||
||||||||/------------------------|------------------------\||||||||
||||||/--------------------------|--------------------------\||||||
||||/----------------------------|----------------------------\||||
||/------------------------------|------------------------------\||
/--------------------------------|--------------------------------\
```

FUNCIONES VIRTUALES

Hemos explicado que cuando se invoca a una función miembro sobrecargada, el compilador selecciona la función de acuerdo a un algoritmo de ajuste de tipos. Por ejemplo:

```
class c
  {
    private: int x;
    public:  void asig (int i) { x = i; }
             void asig (void) { x = 0; }
  };
```

También hemos dicho que una función derivada puede tener miembros con los mismos nombres que los miembros de la clase derivada. En este caso, utilizamos el operador :: para seleccionar el miembro. Por ejemplo:

```
class b                                  class d: public b
  {                                        {
    private: int x;                          private: int x;
    public:  void asig (int i=0) {x=i;}      public:  void asig (int i=0)
  };                                           { x=i; b::asig (i); } };
```

En los dos casos mencionados la función miembro seleccionada para ser invocada es elegida en tiempo de compilación.
En algunos casos interesa seleccionar dinámicamente (en tiempo de ejecución) la función miembro apropiada entre funciones de la clase base y funciones de la clase derivada. La palabra clave **virtual** es un especificador de función que proporciona este mecanismo, pero sólo puede ser usada para modificar declaraciones de funciones miembros.

Una función virtual debe estar definida (no sólo declarada). Se invoca igual que las demás funciones. El prototipo de la función virtual en la clase derivada debe ser igual que su prototipo en la clase base. Es ilegal redinir una función virtual de tal manera que difiera sólo en el tipo devuelto. Si dos funciones con el mismo nombre tienen diferentes argumentos, C++ las considera diferentes, y el mecanimo de función virtual es ignorado. El especificador virtual sólo es necesario ponerlo en la declaración de la función de la clase base, no en la declaración de la función en la clase derivada.

Las funciones virtuales deben ser miembros de alguna clase, pero no pueden ser miembros estáticos (static). Una función virtual puede ser amiga (friend) de otra clase. Una última restricción: los constructores no pueden ser virtuales, pero los destructores sí pueden serlo.

```
// Primer ejemplo de funciones virtuales.

#include <iostream.h>

class b
  {
```

```
    public:

    virtual void vf1 (void) { cout << "b::vf1 "; }
    virtual void vf2 (void) { cout << "b::vf2 "; }
    virtual void vf3 (void) { cout << "b::vf3 "; }
    void f (void)           { cout << "b::f ";   }
  };

class d: public b
  {
    public:

    virtual void vf1 (void) // especificador virtual es legal pero
    { cout << "d::vf1 "; }  // redundante

    void vf2 (int)          // no virtual, puesto que usa una lista de
    { cout << "d::vf2 "; }  // argumentos diferente

    /*
    char vf3 (void) { }     // ilegal: sólo cambia el tipo devuelto
    */

    void f (void) { cout << "d::f "; }  // no virtual
  };

void main (void)
{
  d cd;           // declara un objeto d
  b *pcb = &cd;  // conversión estándar de d* a b*

  pcb->vf1();    // llama a d::vf1()
  pcb->vf2();    // llama a b::vf2() ya que d::vf2 tiene args. diferentes
  pcb->vf3();    // llama a b::vf3()
  pcb->f();      // llama a b::f() (no virtual)
  pcb->b::vf1(); // llama a b::vf1()
}

/*
  SALIDA DE ESTE PROGRAMA:

  d::vf1 b::vf2 b::vf3 b::f b::vf1
*/
// Segundo ejemplo de funciones virtuales.

#include <iostream.h>

class b
  {
    public:

    int i;
    b (void) { i = 1; }
    virtual void imprimir_i (void) { cout << "\ni de clase b: " << i; }
  };

class d1: public b
  {
    public:

    d1 (void) { i = 2; }
    void imprimir_i (void) { cout << "\ni de clase d1: " << i; }
  };

class d2: public b
  {
    public:
```

```
    int i;
    d2 (void) { i = 3; }
    void imprimir_i (void)
    { cout << "\ni de clase d2: " << i << "; b::i de clase d2: " << b::i; }
  };

void main (void)
{
  b cb;
  b *pcb = &cb;
  d1 cd1;
  d2 cd2;

  pcb->imprimir_i (); // imprime:  i de clase b: 1
  pcb = &cd1;
  pcb->imprimir_i (); // imprime:  i de clase d1: 2
  pcb = &cd2;
  pcb->imprimir_i (); // imprime:  i de clase d2: 3; b::i de clase d2: 1
}
```

FUNCIONES PURAS

En la sección anterior hemos dicho que las funciones virtuales permiten a las clases derivadas proporcionar diferentes versiones de una función de la clase base; y también habíamos dicho que las funciones virtuales deben estar definidas en la clase base al igual que todas las funciones miembros. Pues bien, hay una forma de no definir una función virtual en la clase base: declarándola pura.

La forma de declarar una función pura es añadiendo **= 0** al final de la declaración virtual. Ejemplo:

```
  class b
    {
      virtual void vf (int) = 0; // = 0 significa pura
    };
```

En una clase derivada de una clase base con funciones puras, cada función pura debe ser definida o redeclarada como pura.

Si una función virtual es definida en la base, no necesita ser redefinida en las clases derivadas. Las llamadas simplemente llamarían a la función de la clase base.

```
// Ejemplo de funciones puras.

#include <iostream.h>

class b
  {
    public:

    void f1 (void)              { cout << "\nb::f1()"; }
    virtual void f2 (void)      { cout << "\nb::f2()"; }
    virtual void f3 (void)      { cout << "\nb::f3()"; }
    virtual void f4 (void) = 0;
    /*
    virtual void f5 (void) = 0; // ERROR: no definida en ninguna clase
                                // derivada
    */
  };

class d: public b
```

```
  {
    public:

    void f1 (void)  { cout << "\nd::f1()"; }
    void f2 (int)   { cout << "\nd::f2()"; }
    void f3 (void)  { cout << "\nd::f3()"; }
    void f4 (void)  { cout << "\nd::f4()"; }
  };

void main (void)
{
  d cd;
  b *pcb = &cd;

  pcb->f1();    // imprime:  b::f1()
  pcb->f2();    // imprime:  b::f2()
  pcb->f3();    // imprime:  d::f3()
  pcb->f4();    // imprime:  d::f4()
  /*
  pcb->d::f1(); // error: d no es una clase base de b
  */
  pcb->b::f3(); // imprime:  b::f3()
}
```

CLASES ABSTRACTAS

Una clase abstracta es una clase que tiene al menos una función virtual pura.

Una clase abstracta sólo puede ser usada como una clase base para otras clases.

No se puede crear ningún objeto de una clase abstracta.

Una clase abstracta no puede ser usada como el tipo de un argumento ni como el tipo devuelto por una función.

Las referencias a una clase abstracta sí están permitidas, esto permite que no sea necesario un objeto temporal en la inicialización.

```
    *
   ***
  *****
   ***
    *
```

```
// Ejemplo de clase abstracta.

class ca // clase abstracta
  {
    private:
      int x;
    public:
      void asigx (int i) { x = i; }
      int devx (void) { return x; }
      virtual void imprx (void) = 0; // función virtual pura
  };

ca c;        // ERROR: intento de crear un objeto de una clase abstracta
ca *p;       // CORRECTO: puntero a una clase abstracta
ca f (void); // ERROR: el tipo devuelto no puede ser una clase abstracta
void g (ca); // ERROR: el tipo del argumento de una función no puede ser
```

```
             // una clase abstracta
ca& h (ca&); // CORRECTO: el tipo del valor devuelto y el tipo del
             // argumento de una función pueden ser una referencia
             // a una clase abstracta
ca* l (ca*); // CORRECTO:  el tipo del valor devuelto y el tipo del
             // argumento de una función pueden ser un puntero a una
             // clase abstracta
```

Supongamos que d es una clase derivada con la clase abstracta b como clase base inmediata. Entonces, para cada función virtual fv en b, la clase d debe, o bien proporcionar una definición para fv, o bien redeclarar fv como pura.

```
// Ejemplo de definición y redeclaración de funciones puras:
class b
  {
    public:
      virtual void f (void) = 0;
      virtual void g (void) = 0;
      virtual void h (void) = 0;
  };
class d: public b // clase d derivada de clase abstracta b
  {
    public:
      void f (void) { } // función f definida: no realiza ninguna acción
      void g (void);    // función g debe estar definida en algún sitio
      void h (void);    // función h redeclarada como pura
  };
Las funciones miembros pueden ser llamadas desde un constructor
de una clase abstracta, pero llamar a una función virtual pura,
directa o indirectamente, desde tal constructor, provoca un error  /
en tiempo de ejecución; lo cual, por otra parte, es lógico.      /
/        /        /        /        /        /        /        /
       /        /        /        /        /        /        /
     /        /        /        /        /        /        /
   /        /        /        /        /        /        /        /
/        /        /        /        /        /        /        /
       /        /        /        /        /        /        /
     /        /        /        /        /        /        /
   /        /        /        /        /        /        /        /
  /        /        /        /        /        /        /        /
/        /        /        /        /        /        /        /
       /        /        /        /        /        /        /
     /        /        /        /        /        /        /
   /        /        /        /        /        /        /        /
// Ejemplo de llamada a una función pura no definida (aunque sí
// declarada)

class b
  {
    public:

    int x;
    b (int i = 0) { asig (i); }
    virtual void asig (int i) = 0;
  };

class d: public b
  {
    void asig (int i) { x = i; }
  };

void main (void)
{
  d cd;
}

/*
```

```
  SALIDA DE ESTE PROGRAMA:

  Pure virtual function called

  NOTA:

  El mensaje de error anterior puede variar entre distintas
  implementaciones de C++; este programa ha sido compilado
  con Borland C++ 2.0. Recalcar que el error se produce en
  tiempo de ejecución, no en tiempo de compilación.
*/
```

Las funciones virtuales pagan un precio por su versatilidad: cada objeto en la clase derivada necesita llevar un puntero a una tabla de funciones, necesario para seleccionar la función correcta en tiempo de ejecución.

Por lo tanto, cuando no sea estrictamente necesario el uso de funciones virtuales, es preferible utilizar el operador de resolución de ámbito

```
nombre_de_clase::nombre_de_funcion_miembro
```

para seleccionar la función miembro a invocar.

JERARQUIA DE CLASES

```
===Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë===
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Ì==========================================================================¹
| Una clase derivada puede ser a su vez clase base de otra clase derivada. |
Ì==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==Ë==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==Î==¹
============================================================================
// Ejemplo de jerarquía de clases.

#include <iostream.h>

class c1 // clase base inmediata de c1 y no inmediata de c3
{
   public:

   int x;
   c1 (int i = 0) { x = i; }
};

class c2: public c1 // clase derivada de c1 y base de c3
  {
    public:
```

```
    int x;
    c2 (int i = 0): c1 (2 * i) { x = i; }
  };

class c3: public c2 // clase derivada inmediata de c2 y no inmediata de c1
  {
    public:

    int x;
    c3 (int i = 0): c2 (3 * i) { x = i; }
  };

void main (void)
{
  c3 c (4);

  cout << c.c1::x << ' '; // imprime: 24
  cout << c.c2::x << ' '; // imprime: 12
  cout << c.c3::x << ' '; // imprime: 4
  cout << c.x << ' ';     // imprime: 4
}
```

LECCIÓN 7

HERENCIA MULTIPLE

En la lección anterior estudiamos la herencia simple **(una clase derivada puede tener únicamente una clase base). En esta lección estudiaremos la** herencia múltiple **(una clase derivada puede tener más de una clase base).**

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

- Forma general de declarar una clase.
- Jerarquía circular.
- Ambigüedad en los miembros.
- Herencia virtual: clases virtuales.
- Orden de llamada a los constructores y los destructores.
- Destructores virtuales.
- Orden de ejecución de un programa en Turbo C++.
- Algunos ejemplos curiosos.

FORMA GENERAL DE DECLARAR UNA CLASE

Cuando una clase derivada tiene más de una clase base se habla de herencia múltiple.

La forma general de declarar una clase derivada es:

```
  <palabra_clave_de_clase> <nombre_de_clase>: <lista_de_clases_bases>
  {
    <lista_de_miembros>
  }
```

donde:

- <palabra_clave_de_clase> es la palabra clave class, struct o union.
- <nombre_de_clase> debe ser un nombre único dentro de su ámbito.
- <lista_de_clases_bases> es la lista de clases bases separadas por comas, cada clase base en esta lista puede ir precedida por la palabra clave public o private.
- <lista_de_miembros> consiste en la declaración de los datos miembros y las funciones miembros.

```
// Ejemplo de herencia múltiple.

#include <iostream.h>

class b1
{
   public:

   int x;
   b1 (int i = 0) { x = i; }
};

class b2
  {
    public:

    int x;
    b2 (int i = 0) { x = i; }
  };

class b3
  {
    public:

    int x;
    b3 (int i = 0) { x = i; }
  };

class d: public b1, public b2, private b3
  {
    public:

    int x;
    d (int i = 0): b1 (2 * i), b2 (3 * i), b3 (-2 * i) { x = i; }
  };

void main (void)
{
  d cd (4);

  cout << cd.b1::x << ' '; // imprime: 8
  cout << cd.b2::x << ' '; // imprime: 12
  cout << cd.d::x << ' ';  // imprime: 4
```

```
  cout << cd.x << ' ';     // imprime: 4
}
```

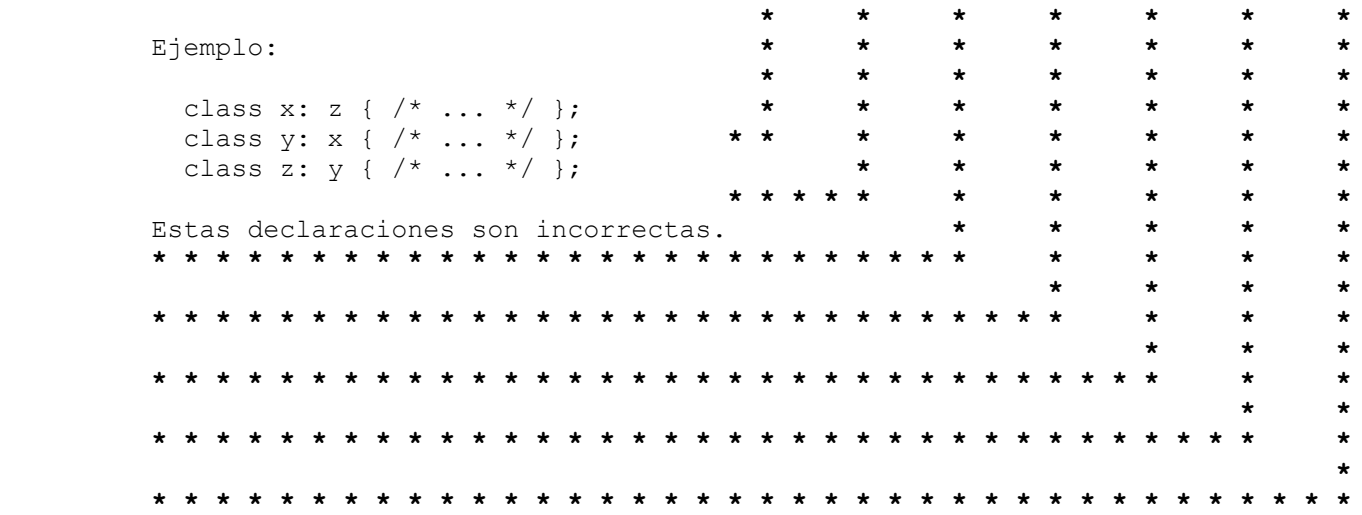
JERARQUIA CIRCULAR

La jerarquía circular se da cuando una clase A es base de una clase B (no necesariamente base inmediata) y la clase B es base de la clase A (no necesariamente base inmediata). Lógicamente, este tipo de jerarquía es ilegal.

Ejemplo:

```
  class x: z { /* ... */ };
  class y: x { /* ... */ };
  class z: y { /* ... */ };
```

Estas declaraciones son incorrectas.



AMBIGÜEDAD EN LOS MIEMBROS

La ambigüedad en los miembros se da cuando una clase deriva miembros con idéntico nombre de diferentes clases y no hay ningún miembro con ese nombre en la clase derivada. Ilustremos esto con un ejemplo:

```
class x { public: int f (void); };
class y { public: int f (void); };
class z: public x, public y { public: void g (void) { x::f() + y::f(); } };

void main (void)
{
   z clase;
   clase.f (); // ERROR: ¨clase.x::f() o clase.y::f()?
}
```

Hay dos formas de solucionar este problema:

1) Usando el operador de resolución de ámbito :: para invocar a la función f() que deseamos. De hecho, esto se ha hecho en g().

2) Añadir la función miembro f() a la clase derivada z. Por ejemplo:
   class z: ... { ... void f (void) { ... } ... };

HERENCIA VIRTUAL

Las clases virtuales se utilizan cuando una clase base se pasa más de una vez a una misma clase derivada, lo cual puede ocurrir en la herencia múltiple.

Una clase base no puede ser especificada más de una vez en una clase derivada:

```
class B { ...};
class D : B, B { ... };  // Ilegal
```

Sin embargo, una clase base se puede pasar indirectamente a una clase derivada más de una vez:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }  // CORRECTO
```

En este caso, cada objeto de clase Z tendrá dos sub-objetos de clase B. Si esto causa probemas, se puede añadir la palabra clave virtual al especificador de clase base. Por ejemplo,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B es ahora una clase base virtual, y la clase Z tiene solamente un sub-objeto de clase B.

ORDEN DE LLAMADAS A LOS CONSTRUCTORES Y LOS DESTRUCTORES

Los constructores de la clase base son llamados en el orden que fueron declarados:

```
class Y { ... };
class X: public Y { ... };
X clase;
```

Orden de llamadas:

```
Y (); // constructor de clase base
X (); // constructor de clase derivada
```

Para el caso de clases bases múltiples:

```
class X: public Y, public Z { ... };
X clase;
```

los constructores son llamados en el orden de declaración:

```
Y (); // primero de la lista
Z ();
X ();
```

Los constructores para las clases bases vituales son invocados antes que los de cualquier otra clase base no virtual.

Si la jerarquía contiene múltiples clases bases virtuales, los constructores de las clases bases virtuales son invocados en el orden en el cual fueron declarados. Los constructores de las clases bases no virtuales son invocados antes que el constructor de la clase derivada.

Si una clase virtual es derivada de una base no virtual, esa base no virtual será llamada primero, para que la clase base virtual sea construida apropiadamente.

Por ejemplo, el código

```
  class X: public Y, virtual public Z { ... };
  X clase;
```

produce el orden:

```
  Z (); // inicialización de la clase base virtual
  Y (); // clase base no virtual
  X (); // clase derivada
```

En el caso que una jerarquía de clases contiene múltiples instancias de una clase base virtual, esa clase base es sólo construida una vez. Si, sin embargo, existen las instacias virtual y no virtual de la clase base, el constructor de la clase es invocado una sola vez para todas las instancias virtuales y una vez para cada ocurrencia no virtual de la clase base.

Un ejemplo más complejo:

```
  class base;
  class base2;
  class nivel1: public base2, virtual public base;
  class nivel2: public base2, virtual public base;
  class nivelalto: public nivel1, virtual public nivel2;
  nivelalto clase;
```

El orden de los constructores es:

```
  base();      // clase base virtual más alta en la jerarquía
               // base es construida una sóla vez
  base2();     // base no virtual de nivel2 virtual
  nivel2();    // clase base virtual
  base2();     // base no virtual de nivel1
  nivel1();    // base no virtual de nivel1
  nivelalto(); // clase derivada
```

Antes se dijo que las clases bases son inicializadas en el orden de declaración. Con los miembros ocurre lo mismo, son inicializados también en orden de declaración, independientemente de la lista de inicialización.

```
  class X
  {
    private:
      int a, b;
    public:
      X (int i, int j): a (i), b (a + j) { }
  };
```

Con esta clase, una declaración de X x (1, 1) resulta en un asignamiento de 1 a x::a y de 2 a x::b.

```
  class base
  {
    private:
      int x;
    public:
      base (int i): x (i) { }
  };

  class derivada: base
  {
    private:
      int a;
    public:
      derivada (int i): a (i * 10), base (a) { } //Ojo! Argumento pasado a
  };                                             //base no está inicializado
```

Con esta clase, una llamada a d (1) no resultará en un valor de 10 para el miembro x de la clase base. El valor pasado al constructor de la clase base será indefinido.

Cuando se quiera una lista de inicializadores en un constructor no inline, no colocar la lista en la declaración de la clase. En lugar de ello, ponerlo en el punto en el cual la función es definida.

```
  derivada::derivada (int i): a (i)
  {
    ...
  }
// Ejemplo sobre el orden de llamadas a los constructores y los destructores.

// Sería interesante para el usuario que intentara averiguar cuál es la
// salida antes de mirarla al final de esta ventana. Por ejemplo, observando
// el código fuente, podría escribir la salida del programa en un papel, y
// una vez hecho esto, comprobarla con la salida real que está escrita al
// final de esta ventana.

#include <iostream.h>
#include <conio.h>

#define declarar_clase(NOMBRE_CLASE)                        \
  class NOMBRE_CLASE                                        \
    {                                                       \
      public:                                               \
        NOMBRE_CLASE ()  { cout << "\nC "#NOMBRE_CLASE; } \
        ~NOMBRE_CLASE () { cout << "\nD "#NOMBRE_CLASE; } \
    }

declarar_clase (base1);
declarar_clase (base2);
declarar_clase (miembro1);
declarar_clase (miembro2);

class derivada1: public base2, public base1
  {
    public:
      miembro1 m1_1, m1_2;
      miembro2 m2;
      derivada1 ()  { cout << "\nC derivada1"; }
      ~derivada1 () { cout << "\nD derivada1"; }
  };

class derivada2: public base1, virtual public base2
  {
    public:
      derivada2 ()  { cout << "\nC derivada2"; }
      ~derivada2 () { cout << "\nD derivada2"; }
};

class derivada3: virtual public derivada2, virtual public base2
  {
    public:
      miembro1 m1;
      derivada3 ()  { cout << "\nC derivada3"; }
      ~derivada3 () { cout << "\nD derivada3"; }
};

class derivada4: virtual base2, derivada3
  {
    public:
      miembro1 m1_1;
      derivada3 d3;
      miembro1 m1_2;
```

```
      derivada4 ()  { cout << "\nC derivada4"; }
      ~derivada4 () { cout << "\nD derivada4"; }
  };

void main (void)
{
  {
    cout << "\n*** Declaración de una clase derivada1";
    derivada1 d1;
  }
  getch ();

  {
    cout << "\n*** Declaración de una clase derivada2";
    derivada2 d2;
  }
  getch ();

  {
    cout << "\n*** Declaración de una clase derivada3";
    derivada3 d3;
  }
  getch ();

  {
    cout << "\n*** Declaración de una clase derivada4";
    derivada4 d4;
  }
  getch ();
}

/*
SALIDA DEL PROGRAMA:

*** Declaración de una clase derivada1
C base2
C base1
C miembro1
C miembro1
C miembro2
C derivada1
D derivada1
D miembro2
D miembro1
D miembro1
D base1
D base2
*** Declaración de una clase derivada2
C base2
C base1
C derivada2
D derivada2
D base1
D base2
*** Declaración de una clase derivada3
C base2
C base1
C derivada2
C miembro1
C derivada3
D derivada3
D miembro1
D derivada2
D base1
D base2
*** Declaración de una clase derivada4
```

```
C base2
C base1
C derivada2
C miembro1
C derivada3
C miembro1
C base2
C base1
C derivada2
C miembro1
C derivada3
C miembro1
C derivada4
D derivada4
D miembro1
D derivada3
D miembro1
D derivada2
D base1
D base2
D miembro1
D derivada3
D miembro1
D derivada2
D base1
D base2
*/
```

DESTRUCTORES VIRTUALES

Un destructor puede ser declarado como virtual. Esto permite a un puntero que apunta al objeto de una clase base llamar al destructor correcto en el caso de que el puntero haga referencia actualmente a un objeto de una clase derivada. El destructor de una clase derivada de una clase con un destructor virtual es asimismo virtual.

Ejemplo:

```
  class color
    {
      public:
        virtual ~color (); // destructor virtual para color
    };

  class rojo: public color
    {
      public:
        ~rojo (); // destructor para rojo es virtual también
    };

  class rojointenso: public rojo
    {
      public:
        ~rojointenso (); // destructor de rojointenso es también virtual
    };

  color *paleta[3];

  paleta[0] = new rojo;
  paleta[1] = new rojointenso;
  paleta[2] = new color;
```

Esto produce:

```
  delete paleta[0]; // El destructor para rojo es llamado,
                    // seguido por el destructor para color.

  delete paleta[1]; // El destructor para rojointenso es llamado,
                    // seguido por rojo y color.

  delete paleta[2]; // El destructor para color es invocado.
```

Sin embargo, en el caso que ningún constructor fuera declarado virtual, delete paleta[0], delete paleta[1], y delete paleta[2] sólo llamarían al destructor para clase color. Esto destruiría incorrectamente los dos primeros elementos, los cuales eran actualmente de tipo rojo y rojointenso.

Los constructores no pueden ser virtuales.

ORDEN DE EJECUCION DE UN PROGRAMA EN TURBO C++

El orden de ejecución del final de un programa en Turbo C++ (o Borland C++) es el que sigue:

- Las funciones atexit son ejecutadas en el orden que fueron insertadas.

- Las funciones #pragma exit son ejecutadas en el orden de sus códigos de prioridad.

- Los destructores para las variables globales son llamados.

El orden de ejecución al comienzo de un programa en Turbo C++ (o Borland C++) es justo el contrario que el orden de terminación, es decir, invirtiendo los tres puntos anteriores.

Cuando se llama a exit dentro de un programa, los destructores no son llamados para las variables locales en el ámbito actual. Las variables globales son destruidas en su orden normal.

Si se llama a abort en cualquier lugar de un programa, ningún destructor es llamado, ni incluso para las variables con un ámbito global.

La función atexit(), cuyo prototipo se encuentra en el fichero stdlib.h, fue discutida en el tutor de C. Sin embargo, las directivas #pragma startup y #pragma exit no se explicaron, por este motivo se estudian a continuación:

```
--------------------------------------------------
  #pragma startup <nombre_de_función> [priordidad]
  #pragma exit <nombre_de_función> [prioridad]~
==================================================
```

La directiva #pragma startup permite al progrma especificar funciones que deben ser ejecutadas al principio del programa (antes que se llame a la función main).

La directiva #pragma exit permite al programa especificar funciones que deben ser ejecutadas al final del programa (justo después de que el programa termina con _exit).

<nombre_de_funcion>

```
==================
<nombre_funcion> debe ser una función previamente declarada que no toma
ningún argumento y devuelve void.

Debe estar declarada como

  void func (void);

El nombre de la función debe ser definido (o declarado) antes de la línea
en la que se encuentra el pragma.


[prioridad]
===========
El parámetro opcional prioridad es un entero en el rango de 64 a 255.

   0 = prioridad más alta
100 = prioridad por defecto

-------------------------------------------
|    No usar las prioridades de 0 to 63.   |
| Ellas son usadas por las librerías de C. |
-------------------------------------------

Las funciones con prioridades más altas son las primeras que se llaman al
comienzo y las últimas que se llaman al final.
// Ejemplo sobre el orden de ejecución de un programa

#include <stdlib.h>
#include <iostream.h>

#define definir_funcion(nombre_funcion) \
  void nombre_funcion (void) { cout << "\nFunción "#nombre_funcion"()."; }

definir_funcion (f1);
definir_funcion (f2);
definir_funcion (f3);
definir_funcion (f4);
definir_funcion (f5);
definir_funcion (f6);

#pragma startup f1 80
#pragma startup f2 70

#pragma exit f5 90
#pragma exit f6

#define declarar_clase(nombre_clase)                                    \
class nombre_clase                                                      \
{                                                                       \
  public:                                                               \
    nombre_clase() { cout << "\nConstructor de clase "#nombre_clase"."; } \
    ~nombre_clase() { cout << "\nDestructor de clase "#nombre_clase"."; } \
}

declarar_clase (x);
x x;

void main (void)
{
  declarar_clase (y);
  y y;

  atexit (f3);
  atexit (f4);
}
```

```
/*
SALIDA DEL PROGRAMA:

Constructor de clase x.
Función f2().
Función f1().
Constructor de clase y.
Destructor de clase y.
Función f4().
Función f3().
Función f6().
Función f5().
Destructor de clase x.
*/
```

ALGUNOS EJEMPLOS CURIOSOS

```
// Miscélanea de ejemplos.

#ifdef __TCPLUSPLUS__ || __BCPLUSPLUS
#  pragma warn -aus  // evita aviso de valor asignado y nunca usado
#  pragma warn -par  // evita aviso de parámetro no usado nunca
#endif

class x
  {
    private:
      int i;
    public:
      x (void)      { /* ... */ }
      x (int x)     { /* ... */ }
      x (const x&) { /* ... */ }
  };

void f1 (void)
{
  x uno;           // invoca constructor por defecto
  x dos (1);       // usa constructor x::x (int)
  x tres = 1;      // llama a x::x (int)
  x cuatro = uno; // invoca a x::x (const x&) para la copia
  x cinco (dos);  // llama a x::x (const x&)
}

class base1
  {
    private:
      int x;
    public:
      base1 (int i) { x = i; }
  };

class base2
  {
    private:
      int x;
    public:
      base2 (int i): x (i) { }
  };

class derivada: public base1, public base2
  {
```

```
    private:
      int a, b;
    public:
      derivada (int i, int j): base1 (i * 2), base2 (i + j), a (i) { b = j; }
  };

void f2 (void)
{
  derivada derivada (-2, 3); // inicializa base1 con -4 y base 2 con 1
}

class y
  {
    public:
      int a, b;
    public:
      y (int i, int j): b (a + j), a (i) { } // a es inicializado antes que b
  };

void f3 (void)
{
   y y (1, 2); // a es inicializado con 1 y b es inicializado con 3
}

class z: virtual y
  {
    private:
      virtual y cy;
    public:
      z (int i, int j): cy (a, b), y (j, i) {} // y() llamado antes que cy()
  };

void f4 (void)
{
  z z (1, 1); // a es inicializado con 1 y b es inicializado con 2
}

void main (void)
{
  f1 ();
  f2 ();
  f3 ();
  f4 ();
}
```

LECCIÓN 8

ENTRADA/SALIDA EN C++

Esta lección describe la entrada y la salida en C++. La biblioteca de entrada/salida para el C, descrita por el fichero de cabecera stdio.h, está todavía disponible en C++. Sin embargo, es preferible utilizar las bibliotecas específicas para C++. En los primeros estándares de C++, el conjunto de clases para los flujos de E/S estaba descrita en el fichero

de cabecera stream.h. La utilización de este fichero se considera hoy día obsoleta. Para terminar la lección detallaremos dos ficheros de cabecera (bcd.h y complex.h) que no tienen nada que ver con la E/S pero que se incluyen en esta lección por ser ésta la única en que se describen los distintos ficheros de cabecera de C++.

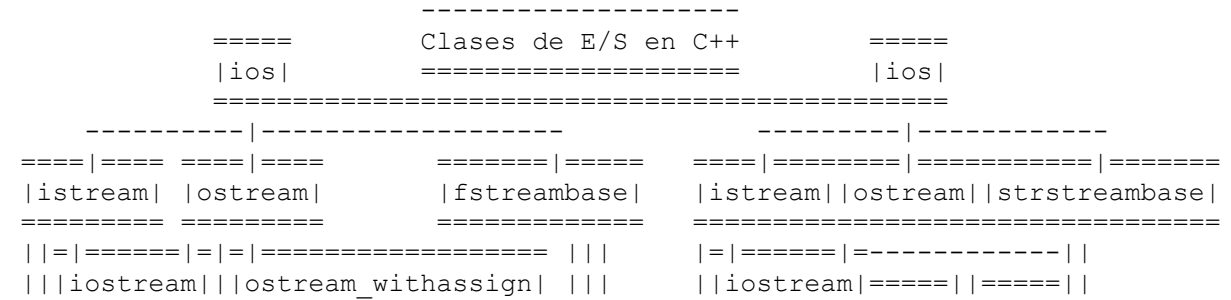
INDICE DE ESTA LECCION

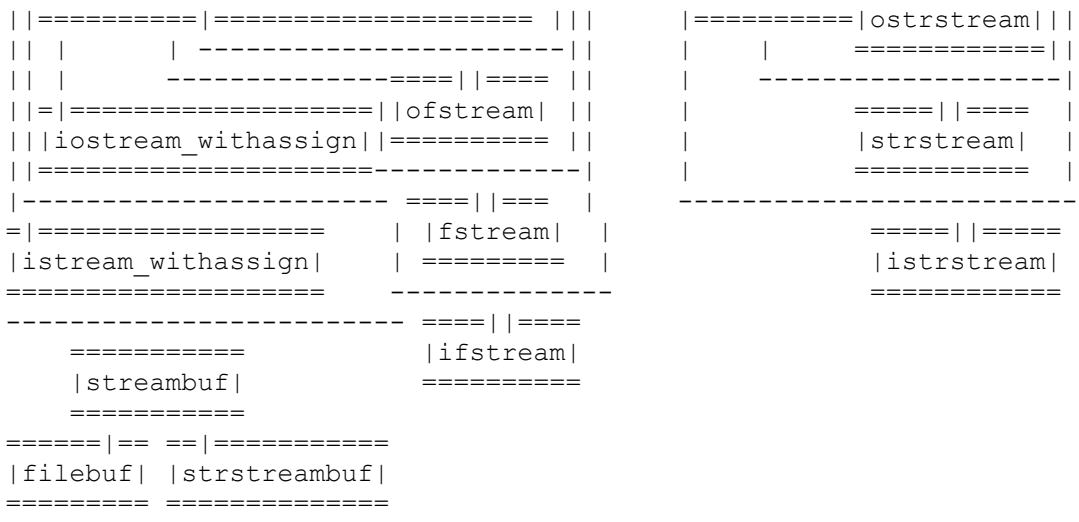
En esta lección se van a estudiar los siguientes puntos:

- Jerarquía de clases de E/S.
- Ficheros de cabecera: iostream.h, fstream.h y strstream.h.
- Clases definidas en iostream.h: streambuf, ios, istream, ostream, iostream, istream_withassign y ostream_withassign.
- Clases definidas en fstream.h: filebuf, fstreambase, ifstream, ofstream y fstream.
- Clases definidas en strstream.h: strstreambuf, strstreambase, istrstream, ostrstream y strstream.
- Clases bcd y complex.

ATENCION: Toda la información proporcionada en esta lección pertenece al Borland C++ 2.0. La presentación de los listados (resumidos) de los ficheros de cabecera es necesario tomarlos de algún compilador. En este caso se ha tomado de dicha versión que es con la que se ha desarrollado todo el tutor. La inclusión (resumida) de los ficheros de cabecera se ha presentado en esta lección para facilitar al usurio la comprensión de las clases declaradas en C++. Obviamente, el usuario no tiene la necesidad de examinar tales listados aunque sí es muy recomendable que lo haga para ver cómo están implementadas las clases explicadas. Las demás versiones de Borland C++ y de Turbo C++ así como de otros compiladores que no sean Borland International deberán tener descripciones similares de las clases y ficheros de cabeceras comentados. Si se comprende la jerarquía de clases de E/S explicada en esta lección sobre el compilador citado de Borland, no se debe tener mucha dificultad para comprender otra jerarquía de clases de E/S de otros compiladores.

JERARQUIA DE CLASES DE E/S





FICHEROS DE CABECERA

Hay tres ficheros de cabecera en la que están declaradas las clases anteriores:

iostream.h	declara los flujos básicos
fstream.h	declara las clases de flujos de C++ que soportan entrada y salida de ficheros
strstream.h	declara las clases de flujos de C++ para usarlas con arrays de bytes en memoria

CLASES DEFINIDAS EN IOSTREAM.H

Clases definidas en el fichero iostream.h, desde las más primitivas hasta las más especializadas:

streambuf	Proporciona métodos para los buffers de memoria.
ios	Manipula errores y variables de estado de los flujos.
istream	Manipula conversiones de caracteres formateadas y no formateadas de un streambuf
ostream	Manipula conversiones de caracteres formateadas y no formateadas de un streambuf
iostream	Combina istream y ostream para manipular operaciones bidireccionales en un solo flujo.
istream_withassign	Proporciona constructores y operadores de asignación para el flujo cin.
ostream_withassign	Proporciona constructores y operadores de asignación para cout, cerr y clog.

C++ proporciona estos objetos de flujos predefinidos:

cin	Entrada estándar, normalmente el teclado, se corresponde con stdin en C
cout	Salida estándar, normalmente el teclado, se corresponde con stdout en C

cerr	Salida de error estándar, normalmente la pantalla, se corresponde con stderr en C
clog	Una versión completamente buffereada de cerr (no tiene equivalente en C)

Se pueden redireccionar los flujos cin, cout, cerr y clog.

CLASES DEFINIDAS EN FSTREAM.H

filebuf	Especializa la clase streambuf para manipular ficheros.
fstreambase	Proporciona operaciones comunes a los flujos de ficheros.
ifstream	Proporciona operaciones de entrada sobre un filebuf.
ofstream	Proporciona operaciones de salida sobre un filebuf.
fstream	Proporciona operaciones simultáneas de entrada y salida sobre un filebuf.

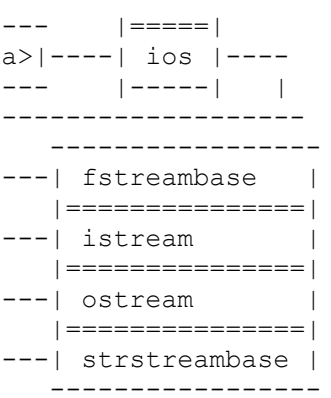
CLASES DEFINIDAS EN STRSTREAM.H

strstreambuf	Especializa la clase streambuf para formatos en memoria.
strstreambase	Especializa la clase ios para los flujos de string.
istrstream	Proporciona operaciones de entrada sobre un strstreambuf.
ostrstream	Proporciona operaciones de salida sobre un strstreambuf.
strstream	Proporciona operaciones simultáneas de entrada y salida sobre un strstreambuf.

Nota: en realidad, el fichero tiene el nombre de strstrea.h, ya que un nombre de fichero no puede tener más de 8 caracteres en el DOS.

CLASE ios

ios	Proporciona operaciones comunes a la entrada y la salida



Declarada en: iostream.h

Las clases derivadas de ios especializan la E/S con operaciones de formato de alto nivel.

```
Constructores
=============
Asocia un streambuf dado con el flujo:
  ios (streambuf *);


Funciones miembros
==================
bad         clear       eof         fail        fill        flags
good        precision   rdbuf       rdstate     setf        tie
unsetf      width


Definición
==========

class ios
  {
    public:

    // bits de estado de flujo
    enum io_state
      {
        goodbit  = 0x00,    // si está a 1, todo está correcto
        eofbit   = 0x01,    // 1 en fin de fichero
        failbit  = 0x02,    // falló la última operación de E/S
        badbit   = 0x04,    // se intentó una operación incorrecta
        hardfail = 0x80     // error irrecuperable
      };

    // modo de operación del flujo
    enum open_mode
      {
        in   = 0x01,        // abre para lectura
        out  = 0x02,        // abre para escritura
        ate  = 0x04,        // busca fin de fichero en la apertura original
        app  = 0x08,        // modo añadir: todas las adiciones se hacen al
                            // final del fichero
        trunc    = 0x10,    // trunca fichero si ya existe
        nocreate = 0x20,    // la apertura falla si el fichero no existe
        noreplace= 0x40,    // la apertura falla si el fichero ya existe
        binary   = 0x80     // fichero binario (no de texto)
      };

    // dirección de búsqueda en flujo
    enum seek_dir { beg=0, cur=1, end=2 };

    // indicadores de formato
    enum
      {
        skipws    = 0x0001, // salta espacios en blanco en la entrada
        left      = 0x0002, // salida ajustada a la izquierda
        right     = 0x0004, // salida ajustada a la derecha
        internal  = 0x0008, // relleno después de signo o indicador de base
        dec   = 0x0010,     // conversión decimal
        oct   = 0x0020,     // conversión octal
        hex   = 0x0040,     // conversión hexadecimal
        showbase  = 0x0080, // usa indicador de base en la salida
        showpoint = 0x0100, // fuerza coma decimal (salida flotante)
        uppercase = 0x0200, // salida hexadecimal en mayúsculas
        showpos   = 0x0400, // añade '+' a los enteros positivos
        scientific= 0x0800, // usa notación flotante 1.2345E2
        fixed     = 0x1000, // usa notación flotante 123.45
        unitbuf   = 0x2000, // vuelca todos los flujos después de la inserc.
        stdio     = 0x4000  // vuelca stdio y stderr después de la inserción
```

```
      };

    // constantes para el segundo parámetro de seft()
    static const long basefield;    // dec | oct | hex
    static const long adjustfield;  // left | right | internal
    static const long floatfield;   // scientific | fixed

    // constructor, destructor
    ios (streambuf *);
    virtual ~ios ();

    // para leer/poner/borrar indicadores de formato
    long flags ();
    long flags (long);
    long setf (long _setbits, long _field);
    long setf (long);
    long unsetf (long);

    // lee/pone anchura de campo
    int width ();
    int width (int);

    // lee/pone carácter de relleno
    char fill ();
    char fill (char);

    // lee/pone dígitos de precisión flotante
    int precision (int);
    int precision ();

    // lee/pone ostream atado a este flujo
    ostream * tie (ostream *);
    ostream * tie ();

    // obtiene estado actual del flujo
    int  rdstate ();         // devuelve el estado del flujo
    int  eof ();             // distinto de cero en fin de fichero
    int  fail ();            // distinto de cero si falló una operación
    int  bad ();             // distinto de cero si ocurrió un error
    int  good ();            // distinto de cero si no hay ningún bit de
                             // estado a 1
    void clear (int = 0);    // pone el estado del flujo
         operator void * (); // cero si estado con fallo
    int  operator! ();       // distinto de cero si estado con fallo

    streambuf * rdbuf ();        // obtiene el streambuf asignado

    // ...

    private:

    // ...

    // estas declaraciones previenen la copia automática de un ios
    ios (ios&);             // declarado pero no definido
    void operator= (ios&);  // declarado pero no definido

  };


Descripción de los métodos
==========================

-----
bad    Función miembro
=====
```

```
-----------------------------------
| ios | No cero si ocurrió un error |
|     |   int bad ()                |
------Á-----------------------------

-------
clear    Función miembro
=======
-------------------------------------------------
| ios | Pone el estado del flujo al valor dado: |
|     |   void clear (int = 0)                  |
------Á-------------------------------------------

-----
eof    Función miembro
=====
-----------------------------------
| ios | No cero en fin de fichero |
|     |   int eof ()              |
------Á---------------------------

------
fail    Función miembro
======
---------------------------------------
| ios | No cero si falló una operación |
|     |   int fail ()                  |
------Á---------------------------------

------
fill    Función miembro
======
-------------------------------------------------------------------------
| ios | Devuelve el carácter actual de relleno:                          |
|     |   char fill ()                                                   |
|     |------------------------------------------------------------------|
|     | Pone carácter de relleno; devuelve el que hay previamente:       |
|     |   char fill (char)                                               |
------Á-------------------------------------------------------------------

-------
flags    Función miembro
=======
-----------------------------------------------------------------
| ios | Devuelve los indicadores del formato actual:              |
|     |   long flags ()                                           |
|     |-----------------------------------------------------------|
|     | Pone los indicadores de formato para que sean idénticos   |
|     | al long dado; devuelve los indicadores previos:           |
|     |   long flags (long)                                       |
------Á-----------------------------------------------------------

------
good    Función miembro
======
-------------------------------------------------------------------------------
| ios | No cero si no hay ningún bit de estado a 1 (no aparecieron errores)     |
|     |   int good ()                                                           |
------Á-------------------------------------------------------------------------

-----------
precision    Función miembro
===========
-----------------------------------------------------------------------------
| ios | Pone la precisión de coma flotante; devuelve la precisión previa:     |
|     |   int precision (int)                                                 |
```

```
|     |-------------------------------------------------------------------|
|     | Devuelve la precisión actual de coma flotante:                    |
|     |   int precision ()                                                |
------Á-------------------------------------------------------------------
```

```
-------
rdbuf    Función miembro
=======
```

```
-------------------------------------------------------------------------
| ios         | Devuelve un puntero al streambuf asignado a este flujo: |
|             |   streambuf* rdbuf ()                                   |
|-------------+---------------------------------------------------------|
| fstream     | Devuelve el buffer usado:                               |
| fstreambase |   filebuf* rdbuf ()                                     |
| ifstream    |                                                         |
| ofstream    |                                                         |
--------------Á---------------------------------------------------------
```

```
---------
rdstate    Función miembro
=========
```

```
--------------------------------------
| ios | Devuelve el estado del flujo: |
|     |   int rdstate ()              |
------Á-------------------------------
```

```
------
setf    Función miembro
======
```

```
-----------------------------------------------------------------------
| ios | Pone a 1 los bits de _field correspondientes a los bits de      |
|     | _setbits:                                                       |
|     |   long setf (long _setbits, long _field)                        |
|     |-----------------------------------------------------------------|
|     | Pone a 1 los indicadores correspondientes al long dado;         |
|     | devuelve los indicadores previos:                               |
|     |   long setf (long)                                              |
------Á-----------------------------------------------------------------
```

```
-----
  tie    Función miembro
=====
```

```
-----------------------------------------------------
| ios | Devuelve el flujo antado, o 0 si no hay ninguno: |
|     |   ostream* tie ()                               |
|     |-------------------------------------------------|
|     | Ata otro flujo a éste y devuelve el flujo atado  |
|     | anterior, si lo hay:                            |
|     |   ostream* tie (ostream*)                       |
------Á-------------------------------------------------
```

Los flujos atados son aquéllos que son conectados de tal forma que cuando uno es usado, el otro es afectado de la misma forma.

Por ejemplo, cin y cout están atados; cuando se usa cin, se vuelca cout primero.

Cuando un flujo de entrada tiene caracteres para ser consumidos, o si un flujo de salida necesita más caracteres, el flujo atado es volcado en primer lugar automáticamente.

Por defecto, cin, cerr y clog están atados a cout.

```
--------
unsetf    Función miembro
========
```

```
-------------------------------------------------
| ios | Pone a 0 los bits correspondientes al long |
|     | dado; devuelve el long previo:            |
|     |   long unsetf (long)                      |
------Á-----------------------------------------
```

width Función miembro

```
-----------------------------------------------------------
| ios | Devuelve la anchura atual:                        |
|     |   int width ()                                    |
|     |---------------------------------------------------|
|     | Pone la anchura dada; devuelve la anchura previa: |
|     |   int width (int)                                 |
------Á---------------------------------------------------
```

Programa ejemplo

```
// Ejemplo de algunas funciones de la clase ios.

/*
  Los ejemplos se realizarán sobre el flujo cout que está predefinido en
  iostream.h. El objeto cout no es del tipo ios sino de un tipo derivado
  de la clase ios y por este motivo puede hacer uso de los miembros públicos
  y protegidos de la clase ios. Normalmente la clase ios se suele utilizar
  como clase base de otras clases derivadas y no directamente, por este
  motivo, vamos a probar las funciones de ios con el flujo cout.
*/

#include <iostream.h>
#include <conio.h>

inline void nl (void)
{
  cout << '\n';
}

void main (void)
{
  clrscr ();

  cout << 1;                                    // imprime  1
  int anchura_ant = cout.width (5);
  int ch_de_relleno_ant = cout.fill ('*');
  cout << 2;                                    // imprime  ****2
  cout.width (anchura_ant);
  cout.fill (ch_de_relleno_ant);
  cout << 3;                                    // imprime  3

  nl ();

  // la función width() sólo afecta a la siguiente salida
  const numhex = 0x0F;
  long indicadores = cout.flags ();
  cout.width (4);
  cout.fill ('#');
  cout << numhex << numhex;                     // imprime  ##1515
  cout.width (4);
  cout.flags (ios::hex);
  cout << numhex;                               // imprime  ####f
  cout.width (4);
  cout.setf (ios::uppercase | ios::showbase);
  cout << numhex;                               // imprime  #0XF
```

```
  cout.width (4);
  cout.setf (ios::left, ios::adjustfield);
  cout.width (4);
  cout << numhex;                               // imprime  0XF#
  cout.unsetf (ios::uppercase);
  cout.width (4);
  cout << numhex;                               // imprime  0xf#
  cout.flags (ios::dec | ios::showpos);
  cout.width (4);
  cout << numhex;                               // imprime  #+15
  cout.flags (ios::left);
  cout.width (4);
  cout << numhex; // imprime  ##15  con flags y  #+15  si se hubiese hecho
           // hecho cout.setf (ios::left) en vez de cout.flags (ios::left)
           // Esa es la diferencia principal entre flags() y setf()
  cout.flags (indicadores);
  cout << numhex;                               // imprime  15

  nl ();

  const float numfloat = 2.3456;
  cout << numfloat;                             // imprime  2.3456
  cout.precision (2);
  cout << ' ' << numfloat;                      // imprime   2.35
  cout.setf (ios::scientific);
  cout << ' ' << numfloat;                      // imprime   2.35e+00
  cout.setf (ios::fixed | ios::showpoint);
  cout << ' ' << float (int (numfloat));        // imprime  2.00

  nl ();

  // la siguiente línea imprime: <valor_no_cero> <valor_no_cero> 0
  // donde <valor_no_cero> es un valor distinto de 0. En la sentencia
  // cout << cout; se realiza la conversión implícita:
  // cout << (void *) cout; Esto es práctico para utilizarlo en expresiones
  // de condiciones como las del while: while (cout) ... Aunque con el cout
  // puede no ser muy útil sí lo es con los objetos de tipo fstream que
  // veremos más adelante
  cout << cout << ' ' << (void *) cout << ' ' << ! cout;

  // los modos de operación de y los bits y funciones de estado de los
  // flujos se probarán cuando se estudien las clases ifstream, ofstream
  // y fstream, que son derivadas de la clase ios y por lo tanto utilizan
  // todas estas características de la clase base ios

  getch ();
}
```

CLASE streambuf

```
-----------
  streambuf    Clase para manipular buffers.
===========
                                 ----------------
-----------   |===========|  --| filebuf      |
|<ninguna>|---| streambuf |--| |==============|
-----------   |-----------|  --| strstreambuf |
                                 ----------------
```

Declarada en: iostream.h

Normalmente, deberíamos usar las clases derivadas en nuestras aplicaciones, en vez de streambuf.

```
Constructores
=============
Crea un objeto buffer vacío:
  streambuf ()

Usa el array y el tamaño dados como el buffer:
  streambuf (char *, int)


Funciones miembros
==================
in_avail      out_waiting   sbumpc        seekoff       seekpos
sgetc         sgetn         setbuf        snextc        sputbackc
sputc         sputn         stossc


Definición
==========

class _CLASSTYPE streambuf
  {
    public:

    // constructores and destructores
    streambuf ();            // hace un streambuf vacío
    streambuf (char *, int); // hace un streambuf con un array de
                             // caracteres dado
    virtual ~streambuf();

    // usa el array de caracteres proporcionado para el buffer si es posible
    virtual streambuf * setbuf (signed char *, int);
    // AVISO: esta función no es virtual; no la vuelvas a definir igual
    streambuf * setbuf (unsigned char *, int);

    // obtener (extraer) caracteres
    int sgetc ();                       // obtiene próximo carácter
    int snextc ();                      // avanza y devuelve próximo carácter
    int sbumpc ();                      // devuelve carácter actual y avanza
    void stossc ();                     // avanza a próximo carácter
    int sgetn (char *, int);            // obtiene los próximos n caracteres
    virtual int do_sgetn (char *, int); // implementación de sgetn
    virtual int underflow ();           // llena buffer vacío
    int sputbackc (char);               // devuelve carácter a la entrada
    virtual int pbackfail (int);        // implementación de sputbackc
    int in_avail ();                    // número de caracteres disponibles
                                        // en buffer

    // poner (insertar) caracteres
    int sputc (int);                  // pone un carácter
    int sputn (const char *, int);    // pone n caracteres de string
    virtual int do_sputn (const char * s, int n); // implementación de sputn
    virtual int overflow (int = EOF); // vuelca buffer y hace más sitio
    int out_waiting ();               // número de caracteres no volcados

    // mueve puntero en flujo
    virtual streampos seekoff (streamoff, seek_dir,
                               int = (ios::in | ios::out));
    virtual streampos seekpos (streampos, int = (ios::in | ios::out));
    virtual int sync ();

    // ...
  };
```

Descripción de los métodos
==========

---
in_avail Función miembro
==========

streambuf	Devuelve el número de caracteres restantes en el buffer de entrada: int in_avail ()

---
out_waiting Función miembro
==========

streambuf	Devuelve el número de caracteres restantes en el buffer de salida: int out_waiting ()

---
sbumpc Función miembro
==========

streambuf	Devuelve el carácter actual del buffer de entrada, entonces avanza: int sbumpc ()

---
seekoff Función miembro
==========

filebuf	Mueve el puntero de fichero relativo a la posición actual: virtual long seekoff (long, seek_dir, int)
streambuf	Mueve el puntero de lectura y/o escritura (el tercer argumento determina cuál de los dos o ambos) relativo a la posición actual: virtual long seekoff (long, seek_dir, int = (ios::in | ios::out))
strstreambuf	Mueve el puntero relativo a la posición actual: virtual long seekoff(long, seek_dir, int)

---
seekpos Función miembro
==========

streambuf	Mueve el puntero de lectura y/o escritura a una posición absoluta: virtual long seekpos(long, int = (ios::in | ios::out))

---
sgetc Función miembro
==========

streambuf	Devuelve el próximo carácter en el buffer de entrada: int sgetc10 ()

---
sgetn Función miembro

```
=======
-------------------------------------------------------------------------
| streambuf | Obtiene los próximos n caracteres del buffer de entrada: |
|           |   int sgetn (char*, int n)                                |
------------Á------------------------------------------------------------

--------
setbuf    Función miembro
========
-----------------------------------------------------------------
| filebuf      | Especifica el buffer a usar:                    |
| strstreambuf |   virtual streambuf* setbuf (char*, int)        |
|--------------+-------------------------------------------------|
| fstreambase  | Usa un buffer especificado:                     |
|              |   void setbuf (char*, int)                      |
|--------------+-------------------------------------------------|
| streambuf    | Conecta a un buffer dado:                       |
|              |   virtual streambuf* setbuf (signed char*, int) |
---------------Á-------------------------------------------------

--------
snextc    Función miembro
========
-------------------------------------------------------------
| streambuf | Avanza a (y devuelve el próximo carácter de) el |
|           | buffer de entrada:                              |
|           |   int snextc ()                                 |
------------Á------------------------------------------------

-----------
sputbackc    Función miembro
===========
-------------------------------------------------
| streambuf | Devuelve un carácter a la entrada: |
|           |   int sputbackc (char)             |
------------Á-----------------------------------

-------
sputc    Función miembro
=======
-----------------------------------------------------------
| streambuf | Put one character into the output buffer: |
|           |   int sputc(int)                          |
------------Á--------------------------------------------

-------
sputn    Función miembro
=======
-----------------------------------------------------------
| streambuf | Pone n caracteres en el buffer de salida: |
|           |   int sputn (const char*, int n)          |
------------Á--------------------------------------------

--------
stossc    Función miembro
========
-------------------------------------------------------------------
| streambuf | Avanza al próximo carácter en el buffer de entrada: |
|           |   void stossc ()                                    |
------------Á----------------------------------------------------


Programa ejemplo
================

// Este ejemplo prueba algunos métodos (funciones miembros) de la clase
```

```
// streambuf.

/*
  Los ejemplos se realizarán accediendo al streambuf de los flujos cout y
  cin a través de la función miembro rdbuf() de la clase ios. Los objetos
  cout y cin no son del tipo ios sino de un tipo derivado de la clase ios
  y por este motivo pueden hacer uso de los miembros públicos y protegidos
  de la clase ios.
*/

#include <iostream.h>

#define nl << "\n"

void main (void)
{
  cout << char ((cin.rdbuf())->sputbackc ('X'))  nl; // imprime  X
  cout << (cin.rdbuf())->in_avail ()             nl; // imprime  1
  cout << char ((cin.rdbuf())->sgetc())          nl; // imprime  X
  cout << char ((cout.rdbuf())->sputbackc ('X')) nl; // imprime  X
  cout << (cout.rdbuf())->out_waiting ()         nl; // imprime  0
  cout << char ((cout.rdbuf())->sputc('X'))      nl; // imprime  XX
}
```

CLASE istream

```
---------
  istream    Proporciona entrada formateada y no formateada de un streambuf
=========
-------   |=========|
| ios |---| istream |----
-------   |---------|   |
   -----------------------
   |   ----------------------
   |---| ifstream           |
   |   |====================|
   |---| iostream           |
   |   |====================|
   |---| istream_withassign |
   |   |====================|
   ----| istrstream         |
       ----------------------

Declarada en: iostream.h


Constructores
=============
Asocia un streambuf dado con el flujo:
  istream (streambuf *)


Funciones miembros
==================
gcount    get       getline   ignore    peek      putback   read
seekg     tellg


Definición
==========

class istream : virtual public ios
  {
```

```
public:

// constructor y destructor
istream (streambuf *);
virtual ~istream ();

// ...

// pone/lee la posición del puntero de lectura
istream& seekg (streampos);
istream& seekg (streamoff, seek_dir);
streampos tellg ();

int sync ();

/*
 * Operaciones de extracción no formateadas
 */
// extrae caracteres colocándolos en un array
istream& get (signed char *, int, char = '\n');
istream& get (unsigned char *, int, char = '\n');
istream& read (signed char *, int);
istream& read (unsigned char *, int);

// extrae caracteres colocándolos en un array hasta encontrar el
// carácter de terminación dado como máximo
istream& getline (signed char *, int, char = '\n');
istream& getline (unsigned char *, int, char = '\n');

// extrae caracteres colocándolos en un streambuf hasta encontrar el
// carácter de terminación dado como máximo
istream& get (streambuf&, char = '\n');

// extrae un solo carácter
istream& get (unsigned char&);
istream& get (signed char&);
int      get ();

int      peek ();        // devuelve próximo carácter sin extracción
int      gcount ();      // número de caracteres no formateados en la
                         // última extracción
istream& putback (char); // devuelve un carácter a la entrada

// extrae y desecha caracteres pero para al encontrar el delimitador
istream& ignore (int = 1, int = EOF);

/*
 * Operaciones de extracción formateadas
 */
istream& operator>> (istream& (*_f) (istream&));
istream& operator>> (ios& (*_f) (ios&));
istream& operator>> (signed char *);
istream& operator>> (unsigned char *);
istream& operator>> (unsigned char&);
istream& operator>> (signed char&);
istream& operator>> (short&);
istream& operator>> (int&);
istream& operator>> (long&);
istream& operator>> (unsigned short&);
istream& operator>> (unsigned int&);
istream& operator>> (unsigned long&);
istream& operator>> (float&);
istream& operator>> (double&);
istream& operator>> (long double&);

// extrae de este istream, insertando en streambuf
```

```
    istream& operator>> (streambuf *);

    protected:

    istream ();
    void eatwhite ();  // extrae espacios blancos consecutivos

    private:

    // ...
  };
```

Descripción de los métodos
==========

----
gcount Función miembro
========

istream	Devuelve el número de caracteres extraídos en la última operación: int gcount ()

-----
get Función miembro
=====

istream	Extrae el próximo carácter o EOF: int get () Extra caracteres colocándolos en el char * dado hasta que el delimitador (tercer parámetro) o el fin-de-fichero es encontrado, o hasta que (len - 1) bytes han sido leídos: istream& get (signed char*, int len, char = '\n') istream& get (unsigned char*, int len, char = '\n') þ Un nulo de terminación es siempre colocado en el string de salida; el delimitador nunca es colocado. þ Falla sólamente si no es extraído ningún carácter. Extra un solo carácter colocándolo en la referencia a carácter dada: istream& get (unsigned char&) istream& get (signed char&) Extrae caracteres colocándolos en el streambuf dado hasta que se encuentra el delimitador: istream& get (streambuf&, char = '\n')

----
getline Función miembro
=========

istream	Igual que get, excepto que también se extrae el delimitador: istream& getline (signed char*, int, char = '\n') istream& getline (unsigned char*, int, char = '\n')

----
ignore Función miembro
========

istream	Salta un máximo de n caracteres en el flujo de entrada;

```
|          | para cuando encuentra el delimitador:                    |
|          |   istream& ignore (int n = 1, int delim = EOF)           |
----------Á-----------------------------------------------------------
```

```
------
peek    Función miembro
======
-------------------------------------------------------
| istream | Devuelve próximo carácter sin extracción: |
|          |   int peek ()                             |
----------Á-------------------------------------------
```

```
---------
putback    Función miembro
=========
-----------------------------------------
| istream | Devuelve un carácter al flujo: |
|          |   istream& putback (char)      |
----------Á---------------------------------
```

```
------
read    Función miembro
======
-------------------------------------------------------------------------------
| istream | Extrae un número dado de caracteres colocándolos en un array:     |
|          |   istream& read (signed char*, int)                               |
|          |   istream& read (unsigned char*, int)                             |
|          |                                                                   |
|          | Usa gcount() para determinar el número de caracteres leídos       |
|          | realmente si ocurre un error.                                     |
----------Á-------------------------------------------------------------------
```

```
-------
seekg    Función miembro
=======
-------------------------------------------------------------------------
| istream | Mueve a una posición absoluta (normalmente el argumento       |
|          | es el valor devuelto por tellg):                              |
|          |   istream& seekg (long)                                       |
|          |-----------------------------------------------------------|
|          | Mueve a una posición relativa a la posición actual:           |
|          |   istream& seekg (long, seek_dir)                             |
|          |                                                               |
|          | Usa esta definición para seek_dir:                            |
|          |   enum seek_dir { beg, cur, end };                            |
----------Á---------------------------------------------------------------
```

```
-------
tellg    Función miembro
=======
-------------------------------------------------
| istream | Devuelve la posición actual del flujo: |
|          |   long tellg ()                        |
----------Á----------------------------------------
```

Programa ejemplo
================

```
// Este ejemplo prueba todos los métodos (funciones miembros) de la clase
// istream.

/*
  Los ejemplos se realizarán sobre el flujo cin que está predefinido en
  iostream.h. El objeto cin no es del tipo istream sino de un tipo derivado
  de la clase istream y por este motivo puede hacer uso de los miembros
```

```
  públicos y protegidos de la clase istream.
*/

#include <iostream.h>
#include <conio.h>

void main (void)
{
  const tam = 100;
  char cad1[tam], cad2[tam];

  cout << "Introduce dos líneas:\n";
  cin.getline (cad1, tam).getline (cad2, tam);
  cout << "Primera línea: " << cad1 << "\nSegunda línea: " << cad2;

  cout << "\n\nIntroduce una serie de caracteres terminados con * que "
          "serán ignorados: ";
  cin.ignore (tam, '*');

  cout << "\nIntroduce 5 caracteres: ";
  cin.read (cad1, 5);
  cout << "Cadena leída: " << (cad1[cin.gcount()] = 0, cad1);

  cout << "\nIntroduce un carácter:\n";
  cout << "Carácter leído sin extracción: " << char (cin.peek ());
  cout << "\nCarácter leído con extracción, devuelto y leído de nuevo: "
       << char (cin.putback(cin.get()).get());

  cout << "\n\nPosición actual en flujo cin leída, puesta y leída de nuevo: "
       << cin.seekg (cin.tellg ()).tellg ();

  cout << "\n\nPulsa una tecla para finalizar.";
  getch ();
}

/*
SALIDA:


Introduce dos líneas:
abc
d e f
Primera línea: abc
Segunda línea: d e f

Introduce una serie de caracteres terminados con * que serán ignorados: as
df
g
h j*

Introduce 5 caracteres: qwert
Cadena leída:
qwer
Introduce un carácter:
Carácter leído sin extracción: t
Carácter leído con extracción, devuelto y leído de nuevo: t

Posición actual en flujo cin leída, puesta y leída de nuevo: 11974

Pulsa una tecla para finalizar.


OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) Observando el programa nos damos cuenta de que parte de la salida
mostrada corresponde a caracteres introducidos por teclado y visualizados
```

en la pantalla.

2) En la función read, al leer 5 caracteres, lee primero el carácter nueva línea que se encuentra en el buffer pues se pulsó la tecla ENTER después del *.

3) La función read no lee el carácter t pues le hemos dicho que lea solamente 5 caracteres. Por lo tanto, el carácter t queda en el buffer de entrada y la función get, que sigue a la read, lee el carácter que se encuentra en el buffer en vez de pedir un carácter por teclado.

4) El número 11974 es distinto en cada ejecución del programa. Las funciones seekg() y tellg() son realmente útiles al leer ficheros de discos.
*/

CLASE ostream

```
---------
  ostream    Proporciona salida formateada y no formateada a un streambuf.
=========

-------   |=========|
| ios |---| ostream |----
-------   |---------|   |
   ------------------------
   |   ----------------------
   |---| iostream           |
   |   |====================|
   |---| ofstream           |
   |   |====================|
   |---| ostream_withassign |
   |   |====================|
   ----| ostrstream         |
       ----------------------
```

Declarada en: iostream.h

Constructores
=============
Asocia un streambuf dado con el flujo:
  ostream (streambuf *)

Funciones miembros
==================
flush   seekp   put   tellp   write

Definición
==========

```
class ostream : virtual public ios
  {
    public:

    // constructores y destructores
    ostream (streambuf *);
    virtual ~ostream ();

    // ...

    ostream& flush ();
```

```
    // pone/lee la posición del puntero de escritura
    ostream&  seekp (streampos);
    ostream&  seekp (streamoff, seek_dir);
    streampos tellp ();

    /*
     * Operaciones de inserción no formateadas
     */
    ostream& put (char);                          // inserta el carácter
    ostream& write (const signed char  *, int);   // inserta el string
    ostream& write (const unsigned char  *, int); // inserta el string

    /*
     * Operaciones de inserción formateadas
     */
    // inserta el carácter
    ostream& operator<< (signed char);
    ostream& operator<< (unsigned char);

    // para lo siguiente, inserta representación del valor númerico
    ostream& operator<< (short);
    ostream& operator<< (unsigned short);
    ostream& operator<< (int);
    ostream& operator<< (unsigned int);
    ostream& operator<< (long);
    ostream& operator<< (unsigned long);
    ostream& operator<< (float);
    ostream& operator<< (double);
    ostream& operator<< (long double);

    // inserta string terminado en nulo
    ostream& operator<< (const signed char *);
    ostream& operator<< (const unsigned char *);

    // inserta representación del valor del puntero
    ostream& operator<< (void *);

    // extrae de streambuf, insertando en este otream
    ostream& operator<< (streambuf *);

    // manipuladores
    ostream& operator<< (ostream& (*_f) (ostream&));
    ostream& operator<< (ios& (*_f) (ios&));

    protected:

    // ...

    private:

    // ...
  };


Descripción de los métodos
==========================

-------
flush    Función miembro
=======
----------------------------------
| ostream | Vuelca el flujo:    |
|         |   ostream& flush () |
----------Á-----------------------
```

```
-------
seekp    Función miembro
=======
-------------------------------------------------------------------------
| ostream | Mueve a una posición absoluta (normalmente el argumento |
|         | es un valor devuelto por tellp):                        |
|         |   ostream& seekp (long)                                 |
|         |---------------------------------------------------------|
|         | Mueve a una posición relativa a la posición actual:     |
|         |   ostream& seekp (long, seek_dir)                       |
|         |                                                         |
|         | Usa esta definición para seek_dir:                      |
|         |   enum seek_dir { beg, cur, end };                      |
----------Á-------------------------------------------------------------

-----
put    Función miembro
=====
-------------------------------------
| ostream | Inserta el carácter:   |
|         |   ostream& put (char)  |
----------Á-------------------------

-------
tellp    Función miembro
=======
-----------------------------------------------------
| ostream | Devuelve la posición actual del flujo: |
|         |   long tellp ()                        |
----------Á-----------------------------------------

-------
write    Función miembro
=======
-------------------------------------------------------------
| ostream | Inserta n caracteres (incluido nulos):          |
|         |   ostream& write (const signed char*, int n)    |
|         |   ostream& write (const unsigned char*, int n)  |
----------Á--------------------------------------------------


Programa ejemplo
================

// Este ejemplo prueba todos los métodos (funciones miembros) de la clase
// ostream.

/*
  Los ejemplos se realizarán sobre el flujo cout que está predefinido en
  iostream.h. El objeto cout no es del tipo ostream sino de un tipo derivado
  de la clase ostream y por este motivo puede hacer uso de los miembros
  públicos y protegidos de la clase ostream.
*/

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

void main (void)
{
  cout.flush () << "Volcando flujo.";

  cout << "\nPosición actual en flujo cout leída, puesta y leída de nuevo: "
       << cout.seekp (cout.tellp ()).tellp ();

  const char *p = "\nInsertando esta cadena y el carácter X.";
```

```
  const char lp = strlen (p);
  cout.write (p, lp - 1).put (p[lp - 1]) << "\n";

  getch ();
}

/*
SALIDA:


Volcando flujo.
Posición actual en flujo cout leída, puesta y leída de nuevo: 2090
Insertando esta cadena y el carácter X.



OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) El número 2090 es distinto en cada ejecución del programa. Las
funciones seekp() y tellp() son realmente útiles al leer ficheros
de discos.
*/
```

CLASE iostream

```
----------
  iostream   Permite entrada y salida sobre un flujo.
==========
-----------                     -------------
| istream |--  |==========|  --| fstream   |
|=========| |--| iostream |--| |===========|
| ostream |--  |----------|  --| strstream |
-----------                     -------------

Declarada en: iostream.h

La clase iostream es simplemente una mezcla de sus clases bases, permitiendo
entrada y salida sobre un flujo.


Constructores
=============
Asocia un streambuf dado con el flujo:
  iostream (streambuf *)


Funciones miembros
==================
Ninguna


Definición
==========

class iostream : public istream, public ostream
  {
    public:

    iostream (streambuf *);
    virtual ~iostream ();

    protected:
```

```
    iostream ();
  };
```

Programa ejemplo
==========

```
// Este ejemplo prueba algunos métodos (funciones miembros) de la clase
// iostream.

/*
  Los ejemplos se realizarán accediendo al streambuf de los flujos cout y
  cin a través de la función miembro rdbuf() de la clase ios. Esta función
  devuelve streambuf * y uno de los constructores de la clase iostream es
  precisamente de ese tipo. El objeto cout no es del tipo ios sino de un
  tipo derivado de la clase ios y por este motivo puede hacer uso de los
  miembros públicos y protegidos de la clase ios.
*/

#include <iostream.h>

void main (void)
{
  int x;
  iostream ((iostream ((iostream (cout.rdbuf ()) << "Introduce x: ")
           .rdbuf ()) >> x).rdbuf ()) << "x es ";
  iostream ((iostream (cout.rdbuf ()) << x).rdbuf ()).flush ();
}

/*
SALIDA:


Introduce x: 5
x es 5
```

OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) El primer 5 de la salida mostrada es el número tecleado desde teclado y visualizado en pantalla.

2) Los pasos que se realizan en la sentencia

```
  iostream ((iostream ((iostream (cout.rdbuf ()) << "Introduce x: ")
           .rdbuf ()) >> x).rdbuf ()) << "x es ";
```

son lo siguientes:

Paso	Expresión	Tipo de la expr
1)	cout.rdbuf ()	streambuf *
2)	iostream (cout.rdbuf ())	iostream
3)	iostream (cout.rdbuf ()) << "Introduce x: "	ostream
4)	(iostream (cout.rdbuf ()) << "Introduce x: ").rdbuf ()	streambuf *
5)	iostream ((iostream (cout.rdbuf ()) << "Introduce x: ") .rdbuf ())	iostream
6)	iostream ((iostream (cout.rdbuf ()) << "Introduce x: ") .rdbuf ()) >> x	istream
6)	(iostream ((iostream (cout.rdbuf ()) << "Introduce x: ") .rdbuf ()) >> x).rdbuf ()	streambuf *
7)	iostream ((iostream ((iostream (cout.rdbuf ()) << "Introduce x: ").rdbuf ()) >> x).rdbuf ())	iostream
8)	iostream ((iostream ((iostream (cout.rdbuf ()) << "Introduce x: ").rdbuf ()) >> x).rdbuf ())	

```
             << "x es "                                          ostream
*/
```

CLASE istream_withassign

```
--------------------
  istream_withassign    istream con un operador de asignación añadido
====================
-----------    |====================|    -------------
| istream |----| istream_withassign |----| <ninguna> |
-----------    |--------------------|    -------------

Declarada en: iostream.h


Constructores
=============
Constructor nulo (llama a constructor de istream):
  istream_withassign ()


Funciones miembros
==================
Ninguna (Aunque el operador = está sobrecargado)


Definición
==========

class istream_withassign : public istream
  {
    public:

    // no inicialización
    istream_withassign ();

    virtual ~istream_withassign ();

    // obtiene buffer de ostream y hace inicialización completa
    istream_withassign& operator= (istream&);

    // asocia streambuf con flujo y hace inicialización completa
    istream_withassign& operator= (streambuf *);
  };


Programa ejemplo
================

// Este ejemplo prueba la clase istream_withassign

/*
  El flujo cin es de tipo istream_withassign.
*/

#include <iostream.h>

void main (void)
{
  const int longs = 256;
  char s[longs];

  cout << "\nIntroduce cadena: ";
```

```
  cin >> s;
  cout << "\nCadena introducida: " << s;
  cout << "\nIntroduce línea: ";
  cin.getline (s, longs);
  cout << "\nLínea introducida: " << s;

  cout << "\nIntroduce cadena saltando espacios blancos: ";
  cin.setf (ios::skipws);
  cin >> s;
  cout << "\nCadena introducida: " << s;
  while ((cin.rdbuf())->in_avail ()) // desecha caracteres que hay en buffer
    cin.get ();                      // para que próximo cin lea de teclado
  cout << "\nIntroduce cadena sin saltar espacios blancos: ";
  cin.unsetf (ios::skipws);
  cin >> s;
  cout << "\nCadena introducida: " << s;
}

/*
SALIDA:



Introduce cadena: Antonio Lebrón Bocanegra

Cadena introducida: Antonio
Introduce línea
Línea introducida:  Lebrón Bocanegra
Introduce cadena saltando espacios blancos:  x

Cadena introducida: x
Introduce cadena sin saltar espacios blancos:  x

Cadena introducida:


OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) Parte del texto mostrado en la salida anterior corresponde a texto
introducido por teclado y visualizado en pantalla. Esto se ve claramente
observando el código fuente. Por supuesto, tanto la salida como la
entrada se pueden redirigir puesto que cout y cin trabajan con la salida
y la entrada estándar.

2) La diferencia de usar cin >> s y de usar cin.getline () o cin.get ()
es que la primera versión para de leer cuando se encuentra un carácter
espacio (correspondiente a la macro isspace() definida en ctype.h), un
carácter de nueva línea o el carácter de fin de fichero, mientras que
las dos versiones anteriores paran de leer cuando nosotros queremos.
También se puede observar que la función getline() lee los caracteres del
buffer y no del teclado pues se encuentra el texto " Lebrón Bocanegra"
seguido de un carácter de nueva línea en el buffer.

3) En la segunda parte de la función main() hay algunos conceptos intere-
santes. El primero de ellos es que por defecto, el indicador ios::skipws
está puesto a 1, por lo tanto, la sentencia cin.setf (ios::skipws) del
programa es innecesaria. Cuando este indicador está a 1, todos los primeros
caracteres de espacios blancos en la entrada son saltados. Por ejemplo, en
la salida se ve que se introduce " x" mientras que se almacena "x" en la
cadena de caracteres s. Sin embargo, en la segunda lectura, con el indicador
ios::skipws puesto a 0, los caracteres iniciales blancos no son saltados, y
por lo tanto, al encontrarse en primer lugar con un espacio blanco (ASCII
32) para de leer y la cadena s queda vacía. También es importante significar
que en la primera lectura, no sólo se introduce en el buffer de entrada los
caracteres espacio blanco y x (" x") sino también un carácter de nueva línea;
por ello, se utilizar el bucle while, para desechar este caracter; el bucle
```

```
while también nos previene, por ejemplo, de la entrada " x y" desechando la
'y' y el carácter de nueva línea, ya que si no, la segunda lectura empezaría
leyendo el espacio blanco que hay entre la x y la y.
*/
```

CLASE ostream_withassign

```
--------------------
  ostream_withassign    ostream con un operador de asignación añadido
====================
-----------   |====================|   -----------
| ostream |---| ostream_withassign |---|<ninguna>|
-----------   |--------------------|   -----------

Declarada en: iostream.h


Constructores
=============
Constructor nulo (llama a constructor de ostream):
  ostream_withassign ()


Funciones miembros
==================
Ninguna (Aunque el operador = está sobrecargado)


Definición
==========

class ostream_withassign : public ostream
  {
    public:

    // no inicialización
    ostream_withassign ();

    virtual ~ostream_withassign ();

    // obtiene buffer de istream y hace inicialización completa
    ostream_withassign& operator= (ostream&);

    // asocia streambuf con flujo y hace inicialización completa
    ostream_withassign& operator= (streambuf *);
  };


Programa ejemplo
================

// Este ejemplo prueba la clase ostream_withassign.
// EN REALIDAD, ESTE EJEMPLO MUESTRA COMO SOBRECARGAR LOS OPERDORES >> Y <<
// EN UNA CLASE DEFINIDA POR EL USUARIO

/*
  El flujo cout es de tipo istream_withassign.
*/

#include <iostream.h>

class vect
  {
```

```
    private:
      int x, y;

    public:
      vect (int i = 0, int j = 0) { asig (i, j); }
      void asig (int i, int j) { x = i; y = j; }

      vect operator+ (void) { return *this; }
      vect operator- (void) { return vect (-x, -y); }

      friend vect operator+ (vect v1, vect v2)
        { return vect (v1.x + v2.x, v1.y + v2.y); }
      friend vect operator- (vect v1, vect v2)
        { return v1 + -v2; }

      friend ostream& operator<< (ostream& out, vect v)
        { return out << '[' << v.x << ',' << v.y << ']'; }

      friend istream& operator>> (istream& in, vect& v)
        { return in >> '[' >> v.x >> ',' >> v.y >> ']'; }
  };

void main (void)
{
  vect vec, vec1 (1, 2), vec2 (-3);

  cout << "\nvec1 = " << vec1 << "  vec2 = " << vec2;
  cout << "\n+vec1 - vec2 = " << vec1 - vec2;

  cout << "\nIntroduce vec en la forma [x,y]: ";
  cin >> vec;
  cout << "vec: " << vec;
}

/*
SALIDA:



vec1 = [1,2]  vec2 = [-3,0]
+vec1 - vec2 = [4,2]
Introduce vec en la forma [x,y]: [4,5]
vec: [4,5]
```

OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) El primer [4,5] es introducido desde teclado y visualizado en pantalla.

2) La línea

```
   cout << "vec: " << vec;
```

es equivalente a la línea:

```
  cout.operator<< ("vec: "); operator<< (cout, vec);
```

La direrencia de las dos llamadas anteriores a operator<< viene dada porque la primera es una función miembro (ver definición de clase ostream.h) y la segunda es una función amiga (ver definición de la clase vect más arriba).

Los operadores << y >> son dos operadores binarios que se pueden sobrecargar al igual que los demás operadores sobrecargables. De hecho, los operadores << y >> ya están sobrecargados en las clases istream y ostream para los tipos definidos por el sistema.

Cuando se hace:

```
  cout << "a" << 'b';
```

se está haciendo en realidad:

```
  (cout.operator<<("a")).operator<<('b');
*/
```

CLASE iostream_withassign

```
---------------------
  iostream_withassign    iostream con un operador de asignación añadido
=====================
------------    |====================|    -------------
| iostream |----| iostream_withassign |----| <ninguna> |
------------    |--------------------|    -------------
```

Declarada en: iostream.h

Constructores
=============
Constructor nulo (llama a constructor de iostream):

```
  iostream_withassign ()
```

Funciones miembros
==================
Ninguna (Aunque el operador = está sobrecargado)

Definición
==========

```
class iostream_withassign : public iostream
  {
    public:

    // no inicialización
    iostream_withassign ();

    virtual ~iostream_withassign ();

    // obtiene buffer de stream y hace inicialización completa
    iostream_withassign& operator= (ios&);

    // asocia streambuf con flujo y hace inicialización completa
    iostream_withassign& operator= (streambuf *);
  };
```

FICHERO iostream.h

```
------------
  IOSTREAM.H
============
```

Declara los flujos básicos.

Reemplaza al viejo fichero de cabecera stream.h.

```
Incluye
=======
mem.h

Clases
======
ios                   iostream              iostream_withassign
istream               istream_withassign    ostream
ostream_withassign    streambuf

Funciones
=========
allocate      bad           base          blen          cerr
cin           clear         cout          dbp           dec
do_sgetn      do_snextc     do_sputn      doallocate    eback
ebuf          egptr         endl          ends          eof
epptr         fail          fill          flags         flush
gbump         gcount        get           getline       good
gptr          hex           ignore        in_avail      oct
out_waiting   overflow      pbackfail     pbase         pbump
peek          precision     put           putback       read
sbumpc        seekg         seekoff       seekp         seekpos
setb          setbuf        setf          setg          setp
setstate      sgetc         sgetn         skip          snextc
sputbackc     sputc         sputn         stossc        sync
tellg         tellp         tie           unbuffered    underflow
unsetf        width         write         ws

Constantes, tipos de datos, y variables globales
================================================
adjustfield (const)   basefield (const)     floatfield (const)
io_state (enum)       open_mode (enum)      seek_dir (enum)
streamoff (typedef)   streampos (typedef)

Operadores sobrecargados
========================
    =    >>    <<

Contenido (abreviado):
======================
// ...

#ifndef __IOSTREAM_H
#define __IOSTREAM_H

#if !defined( __MEM_H )
#include <mem.h>    // obtiene memcpy y NULL
#endif

// ...

// Definición de EOF que debe coincidir con la que se hace en <stdio.h>
#define EOF (-1)

// extrae un carácter de int i, asegurando que zapeof(EOF) != EOF
#define zapeof(i) ((unsigned char)(i))

typedef long streampos;
typedef long streamoff;

class streambuf;
class ostream;

class ios
  {
```

```
    // ...
  };

class streambuf
  {
    // ...
  };

class istream : virtual public ios
  {
    // ...
  };

class ostream : virtual public ios
  {
    // ...
  };

class iostream : public istream, public ostream
  {
    // ...
  };

class istream_withassign : public istream
  {
    // ...
  };

class ostream_withassign : public ostream
  {
    // ...
  };

class iostream_withassign : public iostream
  {
    // ...
  };

/*
* Los flujos predefinidos
*/
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;

/*
* Manipuladores
*/
ostream& endl (ostream&);  // inserta nueva línea y vuelca
ostream& ends (ostream&);  // inserta nulo para terminar string
ostream& flush (ostream&); // vuelca el flujo
ios&     dec (ios&);       // pone base de conversión a decimal
ios&     hex (ios&);       // pone base de conversión a hexadecimal
ios&     oct (ios&);       // pone base de conversión a octal
istream& ws (istream&);    // extrae caracteres de espacios blancos

#endif
```

Programa ejemplo
==========

```
// Ejemplo sobre manipuladores.

#include <iostream.h>
```

```
inline ostream& may (ostream& out)
{
  return out.setf (ios::uppercase), out;
}

inline ostream& dosendl (ostream& out)
{
  return out << endl << endl;
}

void main (void)
{
  const int num = 10;
  const char sp = ' ';

  cout << dec << 10 << sp << hex << num << sp << may << num << sp
       << oct << num << dosendl << flush;

  int x;
  const char *msj = "Introduce valor de x: ";

  cout << msj;
  cin >> ws >> x;
  cout << "El valor de x es: " << x << ends;
}

/*
SALIDA:


10 a A 12

Introduce valor de x: 3
El valor de x es 3


OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) El primer 3 es introducido desde teclado y visualizado en pantalla.

2) Los manipuladores endl, ends, flush, dec, hex, oct y ws están declarados
en el fichero iostream.h. Los manipuladores may y dosendl son manipuladores
definidos por el usuario. El manipulador ends es útil cuando la salida se
dirige hacia un string en vez de a pantalla o a fichero.
*/

partir
```

CLASE filebuf

```
---------
  filebuf    Especializa streambuf para manipular ficheros.
=========
-------------   |=========|   -------------
| streambuf |---| filebuf |---| <ninguna> |
-------------   |---------|   -------------

Declarada en: fstream.h


Constructores
=============
```

```
Crea un filebuf que no es añadido a un fichero:
  filebuf ()

Crea un filebuf añadido a un fichero vía un descriptor de fichero:
  filebuf (int fd)

Crea un filebuf añadido a un fichero y usa el buffer de n caracteres
especificado:
  filebuf (int fd, char *, int n)


Funciones miembros
==================
attach    close     fd        is_open   open      seekoff   setbuf


Definición
==========

class  filebuf : public streambuf
  {
    public:

    static const int openprot;  // protección del fichero por defecto

    // constructores, destructor
    filebuf ();                    // crea un filebuf cerrado
    filebuf (int);                 // crea un filebuf añadido a fd
    filebuf (int _f, char *, int); // igual, con buffer especificado
    ~filebuf ();

    int is_open ();   // está el fichero abierto
    int fd ();        // cuál es el descriptor del fichero

    // abre fichero nombrado con modo y protección, lo añade a este filebuf
    filebuf * open (const char *, int, int = filebuf::openprot );

    filebuf * close ();      // vuelca y cierra fichero
    filebuf * attach (int);  // añade este filbuf al descriptor de fichero
                             // abierto

    /*
     * Ejecutan las funciones de streambuf sobre un filebuf
     * Los punteros de lectura y escritura son guardados juntos
     */
    virtual int overflow (int = EOF);
    virtual int underflow ();
    virtual int sync ();
    virtual streampos seekoff (streamoff, seek_dir, int);
    virtual streambuf * setbuf (char *, int);

    protected:

    // ...
  };


Descripción de los métodos
==========================

--------
attach    Función miembro
========
-------------------------------------------------------------------------------
| filebuf     | Añade este filebuf cerrado al descriptor de fichero abierto:|
|             |   filebuf* attach (int)                                     |
```

fstreambase	Conecta a un descriptor de fichero abierto: void attach (int)

close — Función miembro

filebuf	Vuelca y cierra el fichero. Devuelve 0 en caso de error: filebuf* close ()
fstreambase	Cierra el filebuf y fichero asociado: void close ()

fd — Función miembro

filebuf	Devuelve el descriptor de fichero o EOF: int fd ()

is_open — Función miembro

filebuf	Devuelve un valor distinto de cero si el fichero está abierto: int is_open()

open — Función miembro

filebuf	Abre el fichero dado y se conecta a él: filebuf* open (const char*, int mode, int prot = filebuf::openprot)
fstream	Abre un fichero para un fstream:
fstreambase	Abre un fichero para un fstreambase:
ifstream	Abre un fichero para un ifstream:
ofstream	Abre un fichero para un ofstream: void open (const char*, int, int = filebuf::openprot)

seekoff — Función miembro

filebuf	Mueve el puntero de fichero relativo a la posición actual: virtual long seekoff (long, seek_dir, int)
streambuf	Mueve el puntero de lectura y/o escritura (el tercer argumento determina cuál o si son ambos) relativo a la posición actual: virtual long seekoff (long, seek_dir, int = (ios::in | ios::out))
strstreambuf	Mueve el puntero relativo a la posición actual: virtual long seekoff (long, seek_dir, int)

setbuf — Función miembro

```
========
-------------------------------------------------------------------
| filebuf      | Especifica el buffer a usar:                      |
| strstreambuf |   virtual streambuf* setbuf (char*, int)          |
|--------------+---------------------------------------------------|
| fstreambase  | Usa un buffer especificado:                       |
|              |   void setbuf (char*, int)                        |
|--------------+---------------------------------------------------|
| streambuf    | Conecta a un buffer dado:                         |
|              |   virtual streambuf* setbuf (signed char*, int)   |
---------------Á---------------------------------------------------


Programa ejemplo
================

// Ejemplo sobre la clase filebuf.

#include <iostream.h>
#include <fstream.h>

void main (void)
{
  ofstream fich;

  fich.open ("PRUEBA.TXT");

  cout << "¨Está abierto?: " << (fich.rdbuf())->is_open() << endl
       << "Descriptor de fichero: " << (fich.rdbuf())->fd();

  fich.close ();
}

/*
SALIDA:


¨Está abierto?: 1
Descriptor de fichero: 8


OBSERVACIONES SOBRE LA SALIDA DEL PROGRAMA:

1) El número del descriptor de fichero es dependiente del sistema.

2) La clase ofstream la veremos un poco más adelante. En este ejemplo, el
objeto fich es un fichero de escritura. La clase filebuf no se suele usar
directamente sino a través de sus clases derivadas tal y como hemos hecho
en este programa.
*/
```

CLASE fstreambase

```
-------------
  fstreambase   Proporciona operaciones comunes a los flujos de fichero.
=============
                              ------------
                            --| fstream  |
-------   |=============|---  |==========|
| ios |---| fstreambase |-----| ifstream |
-------   |-------------|---  |==========|
                            --| ofstream |
                              ------------
```

Declarada en: fstream.h

Constructores

Crea un fstreambase que no es añadido a un fichero:
```
fstreambase ()
```

Crea un fstreambase, abre un fichero, y lo conecta a él:
```
fstreambase (const char*, int, int = filebuf::openprot)
```

Crea un fstreambase, lo conecta a un descriptor de fichero abierto:
```
fstreambase (int)
```

Crea un fstreambase conectado a un fichero abierto, usa el buffer especializado:
```
fstreambase (int _f, char*, int)
```

Funciones miembros

attach close open rdbuf setbuf

Definición

```
class fstreambase : virtual public ios
  {
    public:

    fstreambase ();
    fstreambase (const char *, int, int = filebuf::openprot);
    fstreambase (int);
    fstreambase (int _f, char *, int);
    ~fstreambase ();

    void open (const char *, int, int = filebuf::openprot);
    void attach (int);
    void close ();
    void setbuf (char *, int);
    filebuf * rdbuf ();

    protected:

    // ...

    private:

    // ...
  };
```

Descripción de los métodos

attach Función miembro

filebuf	Añade este filebuf cerrado al descriptor de fichero abierto: filebuf* attach (int)
fstreambase	Conecta a un descriptor de fichero abierto: void attach (int)

```
-------
close    Función miembro
=======
-------------------------------------------------------------------------
| filebuf     | Vuelca y cierra el fichero. Devuelve 0 en caso de error: |
|             |   filebuf* close ()                                      |
|-------------+----------------------------------------------------------|
| fstreambase | Cierra el filebuf y fichero asociado:                    |
|             |   void close ()                                          |
--------------Á----------------------------------------------------------

------
open    Función miembro
======
-------------------------------------------------------------------------
| filebuf     | Abre el fichero dado y se conecta a él:                  |
|             |   filebuf* open (const char*, int mode,                  |
|             |     int prot = filebuf::openprot)                        |
|-------------+----------------------------------------------------------|
| fstream     | Abre un fichero para un fstream:                         |
| fstreambase | Abre un fichero para un fstreambase:                     |
| ifstream    | Abre un fichero para un ifstream:                        |
| ofstream    | Abre un fichero para un ofstream:                        |
|             |   void open (const char*, int, int = filebuf::openprot)  |
--------------Á----------------------------------------------------------

-------
rdbuf    Función miembro
=======
-------------------------------------------------------------------------
| ios         | Devuelve un puntero al streambuf asignado a este flujo: |
|             |   streambuf* rdbuf ()                                    |
|-------------+----------------------------------------------------------|
| fstream     | Devuelve el buffer usado:                                |
| fstreambase |   filebuf* rdbuf ()                                      |
| ifstream    |                                                          |
| ofstream    |                                                          |
--------------Á----------------------------------------------------------

--------
setbuf    Función miembro
========
-----------------------------------------------------------------
| filebuf      | Especifica el buffer a usar:                   |
| strstreambuf |   virtual streambuf* setbuf (char*, int)       |
|--------------+------------------------------------------------|
| fstreambase  | Usa un buffer especificado:                    |
|              |   void setbuf (char*, int)                     |
|--------------+------------------------------------------------|
| streambuf    | Conecta a un buffer dado:                      |
|              |   virtual streambuf* setbuf (signed char*, int) |
---------------Á------------------------------------------------


Programa ejemplo
================

// Ejemplo sobre la clase fstreambase.

#include <iostream.h>
#include <fstream.h>

void main (void)
{
  ofstream fich;
```

```
  fich.open ("PRUEBA.TXT");

  cout << (fich.rdbuf () == fich.fstreambase::rdbuf ());

  fich.close ();
}

/*
SALIDA:


1


OBSERVACIONES SOBRE EL PROGRAMA:

1) La clase fstreambase es clase base de las clases ifstream, ofstream y
fstream. Esta clase base no se suele usar directamente sino que sirve
simplemente como clase base de las tres clases mencionadas.

2) La función ofstream::rdbuf() está implementada como una función inline
que lo único que hace es realizar la llamada fstreambase::rdbuf().
*/
```

CLASE ifstream

```
----------
  ifstream   Proporciona operaciones de entrada sobre un filebuf.
==========
---------------
| fstreambase |---   |==========|    -------------
|=============|  |---| ifstream |----| <ninguna> |
|   istream   |---   |----------|    -------------
---------------

Declarada en: fstream.h


Constructores
=============
Crea un ifstream que no es añadido a un fichero:
  ifstream ()

Crea un ifstream, abre un fichero, y lo conecta a él:
  ifstream (const char*, int, int = filebuf::openprot)

Crea un ifstream, lo conecta a un descriptor de fichero abierto:
  ifstream(int)

Crea un ifstream, conectado a un fichero abierto y usa un buffer
especificado:
  ifstream(int fd, char*, int)

Funciones miembros
==================
open    rdbuf


Definición
==========

class ifstream : public fstreambase, public istream
```

```
  {
    public:

    ifstream ();
    ifstream (const char *, int = ios::in, int = filebuf::openprot);
    ifstream (int);
    ifstream (int _f, char *, int);
    ~ifstream ();

    filebuf * rdbuf ();
    void open (const char *, int = ios::in, int = filebuf::openprot);
  };
```

Descripción de los métodos
==========================

------
open   Función miembro
======

filebuf	Abre el fichero dado y se conecta a él: filebuf* open (const char*, int mode, int prot = filebuf::openprot)
fstream	Abre un fichero para un fstream:
fstreambase	Abre un fichero para un fstreambase:
ifstream	Abre un fichero para un ifstream:
ofstream	Abre un fichero para un ofstream:
	void open (const char*, int, int = filebuf::openprot)

----
rdbuf   Función miembro
=======

ios	Devuelve un puntero al streambuf asignado a este flujo: streambuf* rdbuf ()
fstream	Devuelve el buffer usado: filebuf* rdbuf ()
fstreambase	
ifstream	
ofstream	

Programa ejemplo
================

```
// Ejemplo sobre la clase ifstream.

#include <iostream.h>
#include <fstream.h>

inline void error_apertura (const char *nomfich)
{
  cout << "No se puede abrir el fichero " << nomfich << endl;
}

inline void error_cierre (const char *nomfich)
{
  cout << "No se puede cerrar el fichero " << nomfich << endl;
}

void main (void)
{
```

```
  // APERTURAS DE FICHEROS DE TEXTO

  // Forma 1 de abrir fichero:
  ifstream f1;
  f1.open ("FICH1.TXT");

  // Forma 2 de abrir fichero:
  ifstream f2 ("FICH2.TXT");

  // Forma 1 de comprobar apertura:
  if (f1.fail ()) // o también se puede hacer: if (! f1.good ())
    error_apertura ("FICH1.TXT");

  // Forma 2 de comprobar apertura:
  if (! f2)
    error_apertura ("FICH2.TXT");

  // APERTURA DE FICHERO BINARIO

  ifstream f3 ("FICH3.TXT", ios::binary);
  if (! f3)
    error_apertura ("FICH3.TXT");

  // CIERRE DE LOS FICHEROS ABIERTOS

  f1.close ();
  if (! f1)
    error_cierre ("FICH1.TXT");
  f2.close ();
  if (! f2)
    error_cierre ("FICH2.TXT");
  f3.close ();
  if (! f3)
    error_cierre ("FICH3.TXT");
}

/*
SALIDA:


No se puede abrir el fichero FICH1.TXT
No se puede abrir el fichero FICH2.TXT
No se puede abrir el fichero FICH3.TXT
No se puede cerrar el fichero FICH1.TXT
No se puede cerrar el fichero FICH2.TXT
No se puede cerrar el fichero FICH3.TXT



OBSERVACIONES SOBRE EL PROGRAMA:

1) Los errores se han producido al ejecutar este programa porque ninguno de
los tres ficheros de lectura (FICH1.TXT, FICH2.TXT y FICH3.TXT) existen en
disco.

2) Después de cada operación que se haga sobre un fichero se puede hacer la
comprobación de operación correcta. Ejemplo:
  ch = f1.get ();
  if (! f1)
    error ();

Todas las funciones de la clase istream (y de sus clases bases como ios)
están a disposición de la clase ifstream. De esta forma, podemos leer del
fichero utilizando las funciones como get(), getline(), read() o el operador
sobrecargado >>.
```

Aunque el usuario no tiene porqué saber cómo están implementadas las funciones de una clase sino solamente su interface, diremos que el operador ! está sobrecargado en la clase ios de la siguiente forma:

```
inline int ios::operator! () { return fail (); }
```

3) En la clase ifstream, si no se especifica el segundo parámetro de la función open() o bien del constructor, éste es por defecto: ios::in.

4) Todo lo dicho en estas observaciones también se puede aplicar a la clase ofstream que veremos a continuación, cambiando las funciones de lectura por funciones de escritura y cambiando el modo de operación por defecto ios::in al modo ios::out.
*/

CLASE ofstream

```
----------
  ofstream    Proporciona operaciones de salida sobre un filebuf.
==========
---------------
| fstreambase |---   |==========|   -------------
|=============|  |---| ofstream |---| <ninguna> |
| ostream     |---   |----------|   -------------
---------------
```

Declarada en: fstream.h

Constructores
=============
Crea un ofstream que no es añadido a un fichero:
```
  ofstream ()
```

Crea un ofstream, abre un fichero, y lo conecta a él:
```
  ofstream (const char*, int, int = filebuf::openprot)
```

Crea un ifstream, lo conecta a un descriptor de fichero abierto:
```
  ofstream (int)
```

Crea un ifstream, conectado a un fichero abierto y usa un buffer especificado:
```
  ofstream (int fd, char*, int)
```

Funciones miembros
==================
open   rdbuf

Definición
==========

```
class ofstream : public fstreambase, public ostream
  {
    public:

    ofstream ();
    ofstream (const char *, int = ios::out, int = filebuf::openprot);
    ofstream (int);
    ofstream (int _f, char *, int);
    ~ofstream ();

    filebuf * rdbuf ();
```

```
    void open(const char *, int = ios::out, int = filebuf::openprot);
  };
```

Descripción de los métodos
==========================

------
open   Función miembro
======

filebuf	Abre el fichero dado y se conecta a él: filebuf* open (const char*, int mode, int prot = filebuf::openprot)
fstream	Abre un fichero para un fstream:
fstreambase	Abre un fichero para un fstreambase:
ifstream	Abre un fichero para un ifstream:
ofstream	Abre un fichero para un ofstream: void open (const char*, int, int = filebuf::openprot)

----
rdbuf   Función miembro
=======

ios	Devuelve un puntero al streambuf asignado a este flujo: streambuf* rdbuf ()
fstream	Devuelve el buffer usado: filebuf* rdbuf ()
fstreambase	
ifstream	
ofstream	

Programa ejemplo
================

```
// Ejemplo sobre la clase ofstream.

#include <iostream.h>
#include <fstream.h>

inline void error_apertura (const char *nomfich)
{
  cout << "No se puede abrir el fichero " << nomfich << endl;
}

void main (void)
{
  ofstream f1 ("FICH1.TXT");
  if (! f1)
    error_apertura ("FICH1.TXT");

  ofstream f2 ("FICH2.TXT", ios::binary);
  if (! f2)
    error_apertura ("FICH2.TXT");

  ofstream f3 ("FICH3.TXT", ios::app);
  if (! f3)
    error_apertura ("FICH3.TXT");

  ofstream f4 ("FICH4.TXT", ios::app | ios::binary);
  if (! f4)
    error_apertura ("FICH4.TXT");
```

```
  ofstream f5 ("FICH5.TXT", ios::nocreate);
  if (! f5)
    error_apertura ("FICH5.TXT");

  ofstream f6 ("FICH6.TXT", ios::noreplace);
  if (! f6)
    error_apertura ("FICH6.TXT");

  ofstream f7 ("FICH7.TXT", ios::nocreate | ios::noreplace);
  if (! f7)
    error_apertura ("FICH7.TXT");

  ofstream f8 ("FICH8.TXT", ios::ate);
  if (! f8)
    error_apertura ("FICH8.TXT");

  ofstream f9 ("FICH9.TXT", ios::trunc);
  if (! f9)
    error_apertura ("FICH9.TXT");

  f1.close ();
  f2.close ();
  f3.close ();
  f4.close ();
  f5.close ();
  f6.close ();
  f7.close ();
  f8.close ();
  f9.close ();
}

/*
SALIDA:


No se puede abrir el fichero FICH5.TXT
No se puede abrir el fichero FICH7.TXT



OBSERVACIONES SOBRE EL PROGRAMA:

1) Se ha producido error de apertura en FICH5.TXT porque éste no existe
en el disco. Sin embargo, con el fichero FICH7.TXT siempre se producirá
un error de apertura: si existe el fichero, hay error debido a ios::
noreplace, y si no existe el fichero, hay error debido a ios::nocreate.
Por defecto, el fichero es abierto con modo de operación ios::out, esto
quiere decir, que tanto exista como no exista el fichero, se crea uno
nuevo; con los modos de operación se puede modificar esta situación. Lo
mismo ocurre con el modo texto que es el que está por defecto; para tra-
bajar con ficheros binarios, es necesario el indicador ios::binary. Con
el modo ios::app las escrituras en el fichero siempre se hacen al final
de éste; con el modo ios::ate se pueden hacer en cualquier lugar usando
el método seekp(). Si se abre el fichero con el modo ios::trunc se des-
carta el contenido actual del fichero si éste existe, o se crea un fi-
chero nuevo si no existe.

2) Podemos escribir en los ficheros abiertos anteriormente haciendo, por
ejemplo:
  f1 << 'x';
o utilizando los métodos de escritura tales como write() o put().
En resumen, podemos utilizar todas las funciones públicas de la clase
ostream, y por lo tanto, también de la clase ios. La otra clase base
de ostream es fstreambase.
*/
```

CLASE fstream

```
---------
  fstream   Proporciona entrada y salida simultánea sobre un filebuf.
=========
---------------
| fstreambase |---   |=========|   -------------
|=============|  |---| fstream |---| <ninguna> |
|  iostream   |---   |---------|   -------------
---------------
```

Declarada en: fstream.h

Constructores
=============
Crea un fstream que no es añadido a un fichero:
  fstream ()

Crea un fstream, abre un fichero, y lo conecta a él:
  fstream (const char*, int, int = filebuf::openprot)

Crea un fstream, lo conecta a un descriptor de fichero abierto:
  fstream (int)

Crea un fstream, conectado a un fichero abierto y usa un buffer especificado:
  fstream (int _f, char*, int)

Funciones miembros
==================
open   rdbuf

Definición
==========

```
class fstream : public fstreambase, public iostream
  {
    public:

    fstream ();
    fstream (const char *, int, int = filebuf::openprot);
    fstream (int);
    fstream (int _f, char *, int);
    ~fstream ();

    filebuf * rdbuf ();
    void open (const char *, int, int = filebuf::openprot);
  };
```

Descripción de los métodos
==========================

```
------
open    Función miembro
======
-------------------------------------------------------------------------
| filebuf     | Abre el fichero dado y se conecta a él:                 |
|             |   filebuf* open (const char*, int mode,                 |
|             |     int prot = filebuf::openprot)                       |
|-------------+---------------------------------------------------------|
| fstream     | Abre un fichero para un fstream:                        |
| fstreambase | Abre un fichero para un fstreambase:                    |
| ifstream    | Abre un fichero para un ifstream:                       |
```

| ofstream | Abre un fichero para un ofstream: |
| | void open (const char*, int, int = filebuf::openprot) |

---

rdbuf Función miembro

ios	Devuelve un puntero al streambuf asignado a este flujo:
	streambuf* rdbuf ()
fstream	Devuelve el buffer usado:
fstreambase	filebuf* rdbuf ()
ifstream	
ofstream	

Programa ejemplo

```
// Ejemplo sobre la clase fstream.

#include <iostream.h>
#include <fstream.h>

inline void error_apertura (const char *nomfich)
{
  cout << "No se puede abrir el fichero " << nomfich << endl;
}

void main (void)
{
  fstream f ("FICH.TXT", ios::in | ios::out);
  if (! f)
    error_apertura ("FICH.TXT");
  f.close ();
}

/*
SALIDA:
```

OBSERVACIONES SOBRE EL PROGRAMA:

1) Este programa no escribe nada porque no se produce error de apertura.

2) La clase fstream se suele utilizar para trabajar con ficheros de lectura y escritura.

3) Por supuesto, se pueden añadir todos los indicadores de modos de operación que se deseen, tales como ios::nocreate para que la operación de apertura falle si el fichero no existe, o ios::binary para abrir un fichero binario puesto que por defecto se abre el fichero en modo texto, o ios::noreplace para que la operación de apertura falle si el fichero ya existe, etc.
*/

FICHERO fstream.h

FSTREAM.H

```
===========
Declara las clases de flujos de C++ que soportan entrada y salida de
ficheros.

Incluye
=======
iostream.h

Clases
======
filebuf        fstream        fstreambase    ifstream       ofstream

Constantes, tipos de datos, y variables globales
================================================
_FSTREAM_H_


Contenido (abreviado):
======================

#ifndef __FSTREAM_H
#define __FSTREAM_H

#if !defined( __IOSTREAM_H )
#include <iostream.h>
#endif

class filebuf : public streambuf
  {
    // ...
  };

class fstreambase : virtual public ios
  {
    // ...
  };

class ifstream : public fstreambase, public istream
  {
    // ...
  };

class ofstream : public fstreambase, public ostream
  {
    // ...
  };

class fstream : public fstreambase, public iostream
  {
    // ...
  };

#endif


Programa ejemplo
================

En el segundo ejemplo que se encuentra en la tarea de ejemplos de esta
lección se encuentra un programa bastante instructivo que hace uso de
las clases ifstream y ofstream: programa que copia un fichero en otro
sitio del disco.
```

CLASE strstreambuf

```
--------------
  strstreambuf    Especializa streambuf para formatos en memoria.
==============
-------------   |==============|   -------------
| streambuf |---| strstreambuf |---| <ninguna> |
-------------   |--------------|   -------------
```

Declarada en: strstrea.h

Constructores

Crea un strstreambuf dinámico:

```
  strstreambuf ()
```

  Será asignada memoria dinámicamente conforme sea necesaria.

Crea un buffer dinámico con funciones de asignación y liberación especificadas.

```
  strstreambuf (void * (*)(long), void * (*)(void *))
```

Crea un strstreambuf dinámico, inicialmente se asigna un buffer de n bytes como mínimo:

```
  strstreambuf (int n)
```

Crea un strstreambuf estático con un buffer especificado:

```
  strstreambuf (char *, int, char *end)
```

  Si en no es nulo, él delimita el buffer.

Funciones miembros

```
freeze    str       setbuf    seekoff
```

Definición

```
class strstreambuf : public streambuf
  {
    public:

    strstreambuf ();
    strstreambuf (int n);
    strstreambuf (void * (*a) (long), void (*f) (void *));
    strstreambuf (signed char * _s, int, signed char * _strt = 0);
    strstreambuf (unsigned char * _s, int, unsigned char * _strt = 0);
    ~strstreambuf ();

    void freeze (int = 1);
    char * str ();
    virtual int doallocate ();
    virtual int overflow (int);
    virtual int underflow ();
    virtual streambuf * setbuf (char *, int);
    virtual streampos seekoff (streamoff, seek_dir, int);

    private:

    // ...
  };
```

Descripción de los métodos
==========

----
freeze Función miembro
========

strstreambuf	Si el parámetro de entrada es distinto de cero, no permite almacenar caracteres en el buffer; realiza la operación contraria pasando un cero: void freeze (int = 1)

-----
str Función miembro
=====

ostrstream strstream	Devuelve y congela (freeze) el buffer: char *str ()
	Debes liberar el buffer si es dinámico.
strstreambuf	Devuelve un puntero al buffer y lo congela (freeze): char *str ()

----
setbuf Función miembro
========

filebuf strstreambuf	Especifica el buffer a usar: virtual streambuf* setbuf (char*, int)
fstreambase	Usa un buffer especificado: void setbuf (char*, int)
streambuf	Conecta a un buffer dado: virtual streambuf* setbuf (signed char*, int)

----
seekoff Función miembro
=========

filebuf	Mueve el puntero de fichero relativo a la posición actual: virtual long seekoff (long, seek_dir, int)
streambuf	Mueve el puntero de lectura y/o escritura (el tercer argumento determina cuál de los dos o ambos) relativo a la posición actual: virtual long seekoff (long, seek_dir, int = (ios::in | ios::out))
strstreambuf	Mueve el puntero relativo a la posición actual: virtual long seekoff(long, seek_dir, int)

Programa ejemplo
==========

```
// Ejemplo sobre la clase strstreambuf.

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

```
void main (void)
{
  const int tam = 100;
  char buf[tam];
  ostrstream s (buf, tam);

  s << "abc" << ends;

  cout << buf << ' '<< (s.rdbuf())->str();
}

/*
SALIDA:


abc abc


OBSERVACIONES SOBRE EL PROGRAMA:

1) La clase ostrstream se explicará un poco más adelante pero era necesario
utilizarla aquí puesto que la clase strstreambuf no se suele utilizar direc-
tamente.

2) El método str() que se ejecuta es el de la clase strstreambuf. Como se
verá más adelante, este método lo posee la clase ostrstream, con lo cual
se podía haber hecho directamente: s.str(). Aunque la forma más fácil de
acceder al buffer es utilizando directamente el mismo buffer si se dispone
de él como es éste el caso: buf.

3) El manipulador ends es necesario, ya que si no se hace, no hay carácter
nulo en buf que delimite el final de la cadena de caracteres.
*/
```

CLASE strstreambase

```
---------------
  strstreambase    Especializa ios para flujos de strings.
===============
                                 --------------
                               --| istrstream |
-------   |===============|   | |============|
| ios |---| strstreambase |--+-| ostrstream |
-------   |---------------|   | |============|
                               --| strstream  |
                                 --------------
Declarada en: strstrea.h


Constructores
=============
Crea un strstreambase vacío:
  strstreambase ()

Crea un strstreambase con un buffer y posición de comienzo especificados:
  strstreambase (const char*, int, char *start)


Funciones miembros
==================
Ninguna (usa ios)
```

Definición
==========

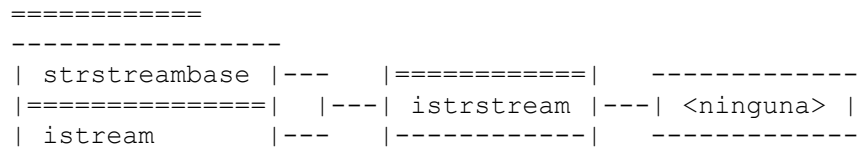
```
class strstreambase : public virtual ios
  {
    public:

    strstreambuf * rdbuf ();

    protected:

    strstreambase (char *, int, char *);
    strstreambase ();
    ~strstreambase ();

    private:

    strstreambuf buf;
  };
```

Programa ejemplo
================

```
// Ejemplo sobre la clase strstreambase.

#include <iostream.h>
#include <strstream.h>

void main (void)
{
  const int tam = 100;
  char buf[tam];
  ostrstream s (buf, tam);

  s << "abc" << ends;

  cout << buf << ' '<< (s.strstreambase::rdbuf())->str();
}

/*
SALIDA:


abc abc


OBSERVACIONES SOBRE EL PROGRAMA:

1) La clase ostrstream se explicará un poco más adelante pero era necesario
utilizarla aquí puesto que la clase strstreambase no se suele utilizar
directamente sino como clase base de las clases istrstream, ostrstream y
strstream.

2) El método rdbuf() que se ejecuta es el de la clase strstreambase. Como
se verá más adelante, este método lo posee la clase ostrstream, con lo cual
se podía haber hecho directamente: s.rdbuf().
*/
```

CLASE istrstream

```
------------
  istrstream    Proporciona operaciones de entrada sobre un strstreambuf.
```



Declarada en: strstrea.h

Constructores

Crea un istrstream con un string especificado (el carácter nulo nunca se extrae):

```
istrstream (const char *)
```

Crea un istrstream usando n bytes de un string especificado:

```
istrstream (const char *, int n)
```

Funciones miembros

Ninguna

Definición

```
class istrstream : public strstreambase, public istream
  {
    public:

    istrstream (char *);
    istrstream (char *, int);
    ~istrstream ();
  };
```

Programa ejemplo

```
// Ejemplo sobre la clase istrstream.

#include <iostream.h>
#include <strstream.h>
#include <stdlib.h>

inline void error (const char *nomclase)
{
  cerr << "Error en clase " << nomclase;
  exit (1);
}

void main (void)
{
  char *buf = "Ejemplo de la clase istrstream.";

  istrstream s1 (buf);
  if (! s1)
    error ("s1");

  istrstream s2 (buf, 5);
  if (! s2)
    error ("s2");

  char ch, str[20];
```

```
  s1 >> ch >> str;
  cout << ch << str << endl;

  s2 >> ch >> str;
  cout << ch << str << endl;
}

/*
SALIDA:


Ejemplo
Ejemp


OBSERVACIONES SOBRE EL PROGRAMA:

1) Con la clase istrstream se trabaja de una forma similar que con la clase
ifstream, con la diferencia de que en istrstream se lee de un string y en
ifstream se lee de un fichero.
*/
```

CLASE ostrstream

```
------------
  ostrstream    Proporciona operaciones de salida sobre un strstreambuf.
============
-----------------
| strstreambase |---   |============|   -------------
|===============|  |---| istrstream |---| <ninguna> |
| istream       |---   |------------|   -------------
-----------------

Declarada en: strstrea.h


Constructors
============
Crea un ostrstream dinámico:
  ostrstream ()

Crea un ostrstream con un buffer de n bytes especificado:
  ostrstream (char*, int, int)

  Si el modo es ios::app o ios::ate, el puntero de lectura/escritura
  es posicionado en el carácter nul del string.


Funciones miembros
==================
pcount   str


Definición
==========

class ostrstream : public strstreambase, public ostream
  {
    public:

    ostrstream (char *, int, int = ios::out);
    ostrstream ();
```

```
    ~ostrstream ();

    char * str ();
    int pcount ();
  };
```

Descripción de los métodos
==========

----
pcount   Función miembro
========

ostrstream	Devuelve el número de bytes almacenados actualmente en el buffer: char *pcount ()

-----
str   Función miembro
=====

ostrstream strstream	Devuelve y congela (freeze) el buffer: char *str () Debes liberar el buffer si es dinámico.
strstreambuf	Devuelve un puntero al buffer y lo congela (freeze): char *str ()

Programa ejemplo
==========

```
// Ejemplo sobre la clase ostrstream.

#include <iostream.h>
#include <strstream.h>
#include <stdlib.h>
#include <string.h>

inline void error (const char *nomclase)
{
  cerr << "Error en clase " << nomclase;
  exit (1);
}

void main (void)
{
  const short tam = 100;
  char buf1[tam], buf2[tam], buf3 [tam], buf4[tam];

  strcpy (buf1, "Ejemplo de la clase ostrstream.");
  strcpy (buf2, "Ejemplo de la clase ostrstream.");
  strcpy (buf3, "Ejemplo de la clase ostrstream.");
  strcpy (buf4, "Ejemplo de la clase ostrstream.");

  ostrstream s1 (buf1, tam);
  if (! s1)
    error ("s1");

  ostrstream s2 (buf2, tam, ios::app);
  if (! s2)
    error ("s2");
```

```
  ostrstream s3 (buf3, strlen (buf3) + 2, ios::app);
  if (! s3)
    error ("s3");

  ostrstream s4 (buf4, tam);
  if (! s4)
    error ("s4");

  s1 << 'a' << "bc" << ends;
  s2 << 'a' << "bc" << ends;
  // s3 << 'a' << "bc" << ends; // no se añade ni c (de "bc") ni 0 (de ends)
  s4 << 'a' << "bc";

  cout << buf1 << endl << buf2 << endl << buf4 << endl;
}

/*
SALIDA:


abc
Ejemplo de la clase ostrstream.abc
abcmplo de la clase ostrstream.



OBSERVACIONES SOBRE EL PROGRAMA:

1) Con la clase ostrstream se trabaja de una forma similar que con la
clase ofstream, con la diferencia de que ostrstream escribe en un string
y ofstream escribe en un fichero.

2) El modo de operación (tercer parámetro de constructor ostrstream) por
defecto es ios::out.
*/
```

CLASE strstream

```
-----------
  strstream    Proporciona entrada y salida simultánea sobre un strstreambuf.
===========
-----------------
| strstreambase |---   |===========|   -------------
|===============|  |---| strstream |---| <ninguna> |
| iostream      |---   |-----------|   -------------
-----------------

Declarada en: strstrea.h


Constructores
=============
Crea un strstream dinámico:
  strstream ()

Crea un strstream con un buffer de n bytes especificado:
  strstream (char*, int n, int mode)

  Si el modo es ios::app o ios::ate, el puntero de lectura/escritura
  es posicionado en el carácter nul del string.


Funciones miembros
```

```
==================
str


Definición
==========

class strstream : public strstreambase, public iostream
  {
    public:

    strstream ();
    strstream (char *, int _sz, int _m);
    ~strstream ();

    char * str ();
  };


Descripción de los métodos
==========================

-----
str    Función miembro
=====
-------------------------------------------------------------------------
| ostrstream   | Devuelve y congela (freeze) el buffer:                 |
| strstream    |   char *str ()                                         |
|              | Debes liberar el buffer si es dinámico.                |
|--------------+--------------------------------------------------------|
| strstreambuf | Devuelve un puntero al buffer y lo congela (freeze):   |
|              |   char *str ()                                         |
---------------Á--------------------------------------------------------


Programa ejemplo
================

// Ejemplo sobre la clase strstream.

#include <iostream.h>
#include <strstream.h>

void main (void)
{
  const short tam = 100;
  char buf[tam];

  strstream str (buf, tam, ios::in | ios::out);

  const char sp = ' ';

  str << 2 << sp << 3.3 << sp << 'x' << sp << "abc";

  int i;
  float f;
  char c;
  char s[10];

  str >> i >> f >> c >> s;

  cout << i << sp << f << sp << c << sp << s << endl;
}

/*
SALIDA:
```

```
2 3.3 x abc


OBSERVACIONES SOBRE EL PROGRAMA:

1) Con la clase strstream se trabaja de una forma similar que con la clase
fstream, con la diferencia de que en strstream trabaja con un string y
fstream trabaja con un fichero.

3) Con el modelo de operación (ios::in | ios::out) podemos escribir y leer
simultáneamente sobre un mismo buffer.
*/
```

FICHERO strstream.h

```
------------
  STRSTREA.H
============
Declara las clases de flujos de C++ para trabajar con arrays de bytes en
memoria.

Incluye
=======
iostream.h

Clases
======
istrstream      ostrstream      strstream       strstreambase
strstreambuf

Constantes, tipos de datos, y variables globales
================================================
_STRSTREAM_H_


Contenido (abreviado):
======================

#ifndef __STRSTREAM_H
#define __STRSTREAM_H

#if !defined( __IOSTREAM_H )
#include <iostream.h>
#endif

class strstreambuf : public streambuf
  {
    // ...
  };

class strstreambase : public virtual ios
  {
    // ...
  };

class istrstream : public strstreambase, public istream
  {
    // ...
  };
```

```
class ostrstream : public strstreambase, public ostream
  {
    // ...
  };

class strstream : public strstreambase, public iostream
  {
    // ...
  };

#endif
```

Programa ejemplo
==========

```
// Ejemplo sobre las clases declaradas en el fichero strstream.h.

#include <iostream.h>  // para flujo cout
#include <strstream.h> // para clases istrstream y ostrstream

void main (void)
{
  char *buffer_fuente = "Ejemplo de las clases istrstream y ostrstream.";
  const int tam = 100;
  char buffer_destino[tam];

  istrstream flujo_fuente (buffer_fuente);
  ostrstream flujo_destino (buffer_destino, tam);

  char ch;
  while (flujo_destino && flujo_fuente.get (ch))
    flujo_destino.put (ch);
  flujo_destino << ends;

  cout << buffer_destino;
}

/*
SALIDA:


Ejemplo de las clases istrstream y ostrstream.


OBSERVACIONES SOBRE EL PROGRAMA:

1) Tal y como está el programa, el bucle while termina por la segunda
condición: flujo_fuente.get (ch). Si tam fuera 10 en vez de 100, entonces
el bucle while se saldría por la primera condición: flujo_destino. En
este segundo caso, sólo se copiarán 10 caracteres de buffer_fuente, pero
hay que tener en cuenta que en este caso no se guarda el carácter 0 (con
ends) en buffer_destino pues no cabría; esto significa que buffer_destino
no sería un string terminado con el carácter nulo.
*/
```

CLASE bcd Y CLASE complex

Además de todos los ficheros de cabecera que acabamos de ver, el C++ actual posee dos más: **bcd.h** y **complex.h**. En el primero está declarada la clase **bcd** para que podamos trabajar con números bcd (binary-code decimal) y en el segundo está declara-

da la clase **complex** para que podamos trabajar con números complejos.

Aunque estas dos clases no tienen mucho que ver con la entrada y la salida, que es el objeto de esta lección, las vamos a incluir en este lugar puesto que en esta lección es el único lugar en el que hemos visto los ficheros de cabecera específicos al C++, es decir, que no los posee el C ni pueden utilizarse en un programa de C.

FICHERO bcd.h

```
-------
  BCD.H
=======
Declara la clase bcd de C++, más los operadores sobrecargados para la clase
bcd y las funciones matemáticas bcd.


Funciones
=========
abs     acos    asin    atan    cos     cosh    exp     log     log10
pow     pow10   real    sin     sinh    sqrt    tan     tanh


Constantes, tipos de datos, y variables globales
================================================
_BCD_H
_BcdMaxDecimals
bcdexpo (enum)


Constructores
=============

-----
  bcd    Convierte número a decimal en código binario (bcd).
=====
Sintaxis:
   bcd bcd (int x);
   bcd bcd (double x);
   bcd bcd (double x, int decimales);

Devuelve el equivalente BCD de un número dado.


Operadores sobrecargados
========================
    +         -=
    -         *=
    *         /=
    /         <=
    ==        >=
    !=        <
    +=        >


Definición (abreviada)
======================
// ...

#ifndef __BCD_H
#define __BCD_H
```

```
#if !defined( __MATH_H )
#include <math.h>
#endif

#define _BcdMaxDecimals     5000

class bcd
  {
    public:

    // Constructores
    _Cdecl bcd();
    _Cdecl bcd(int x);
    _Cdecl bcd(unsigned int x);
    _Cdecl bcd(long x);
    _Cdecl bcd(unsigned long x);
    _Cdecl bcd(double x, int decimals = _BcdMaxDecimals);
    _Cdecl bcd(long double x, int decimals = _BcdMaxDecimals);

    // Manipuladores bcd
    friend long double _Cdecl real(bcd&);   // Devuelve la parte real

    // Funciones matemáticas del ANSI C sobrecargadas
    friend bcd _Cdecl abs(bcd&);
    friend bcd _Cdecl acos(bcd&);
    friend bcd _Cdecl asin(bcd&);
    friend bcd _Cdecl atan(bcd&);
    friend bcd _Cdecl cos(bcd&);
    friend bcd _Cdecl cosh(bcd&);
    friend bcd _Cdecl exp(bcd&);
    friend bcd _Cdecl log(bcd&);
    friend bcd _Cdecl log10(bcd&);
    friend bcd _Cdecl pow(bcd& base, bcd& expon);
    friend bcd _Cdecl sin(bcd&);
    friend bcd _Cdecl sinh(bcd&);
    friend bcd _Cdecl sqrt(bcd&);
    friend bcd _Cdecl tan(bcd&);
    friend bcd _Cdecl tanh(bcd&);

    // Funciones de operadores binarios
    friend bcd _Cdecl operator+(bcd&, bcd&);
    friend bcd _Cdecl operator+(long double, bcd&);
    friend bcd _Cdecl operator+(bcd&, long double);
    friend bcd _Cdecl operator-(bcd&, bcd&);
    friend bcd _Cdecl operator-(long double, bcd&);
    friend bcd _Cdecl operator-(bcd&, long double);
    friend bcd _Cdecl operator*(bcd&, bcd&);
    friend bcd _Cdecl operator*(bcd&, long double);
    friend bcd _Cdecl operator*(long double, bcd&);
    friend bcd _Cdecl operator/(bcd&, bcd&);
    friend bcd _Cdecl operator/(bcd&, long double);
    friend bcd _Cdecl operator/(long double, bcd&);
    friend int _Cdecl operator==(bcd&, bcd&);
    friend int _Cdecl operator!=(bcd&, bcd&);
    friend int _Cdecl operator>=(bcd&, bcd&);
    friend int _Cdecl operator<=(bcd&, bcd&);
    friend int _Cdecl operator>(bcd&, bcd&);
    friend int _Cdecl operator<(bcd&, bcd&);
    bcd& _Cdecl operator+=(bcd&);
    bcd& _Cdecl operator+=(long double);
    bcd& _Cdecl operator-=(bcd&);
    bcd& _Cdecl operator-=(long double);
    bcd& _Cdecl operator*=(bcd&);
    bcd& _Cdecl operator*=(long double);
    bcd& _Cdecl operator/=(bcd&);
```

```
    bcd& _Cdecl operator/=(long double);
    bcd  _Cdecl operator+();
    bcd  _Cdecl operator-();

   // Implementación
   private:

   // ...
  };

// ...

enum bcdexpo
  {
    ExpoZero,
    ExpoInf,
    ExpoNan,
  };

// ...

#endif  // __BCD_H
```

Descripción de los métodos
==========

La sintaxis de las funciones matemáticas del ANSI C aplicadas a objetos de la clase bcd es similar así que no se van a describir aquí pues ya se explicaron en el tutor de C; lo única diferencia está en que donde aparece el tipo double en la función correspondiente al ANSI C, aparace el tipo bcd en la correspondiente función matemática sobrecargada; por ejemplo, el prototipo del ANSI C

```
  double cos (double x);
```

se convierte en la función cos sobrecargada correspondiente a números complejos en

```
  bcd cos (bcd z);
```

Con los operadores sobrecargados para los objetos complejos ocurre lo mismo que las funciones matemáticas sobrecargadas para los objetos complejos.

Hay una función (método) nueva en la clase bcd que no tiene correspondencia con el tipo double:

---
real Devuelve la parte real del número bcd.

Sintaxis:

```
   bcd real (bcd x);
```

Programa ejemplo
==========

```
// Ejemplo de la clase bcd.

#include <iostream.h>
#include <bcd.h>

void main (void)
{
  bcd b1 (2), b2 (3.3);

  cout << "b1: " << b1                        // imprime:  b1: 2
       << endl
```

```
       << "b2: " << b2                        // imprime:  b3: 3.3
       << endl
       << "b1 + b2: " << b1 + b2              // imprime:  b1 + b2: 5.3
       << endl
       << "abs (b1 - b2): " << abs (b1 - b2) // imprime:  abs (b1 - b2): 1.3
       << endl
       << "(b1 = b2 - 1): " << (b1 = b2 - 1) // imprime:  (b1 = b2 - 1): 2.3
       << endl
       << "real (b2): " << real (b2)          // imprime:  real (b2): 3.3
       << endl;
}
```

FICHERO complex.h

```
-----------
  COMPLEX.H
===========
Declara las funciones matemáticas complejas de C++.


Incluye
=======
iostream.h   math.h


Funciones
=========
abs     acos    arg     asin    atan    conj    cos     cosh    exp
imag    log     log10   norm    polar   pow     pow10   real    sin
sinh    sqrt    tan     tanh


Constantes, tipos de datos, y variables globales
================================================
_COMPLEX_H


Constructores
=============

---------
  complex    Crea números complejos.
=========
Sintaxis:
   complex complex (double real, double imag = 0);

Devuelve el número complejo con las partes real e imaginaria dadas.


Operadores sobrecargados
========================
    +        +=
    -        -=
    *        *=
    /        /=
    ==       !=
    <<       >>


Definición (abreviada)
======================
// ...
```

```
#if !defined( __COMPLEX_H )
#define _COMPLEX_H

#if !defined( __MATH_H )
#include <math.h>
#endif

class complex
  {
    public:

    // Constructores
    complex(double __re_val, double __im_val=0);
    complex();

    // Manipuladores de complex
    friend double _Cdecl real(complex&);   // la parte real
    friend double _Cdecl imag(complex&);   // la parte imaginaria
    friend complex _Cdecl conj(complex&);  // el complejo conjugado
    friend double _Cdecl norm(complex&);   // el cuadrado de la magnitud
    friend double _Cdecl arg(complex&);    // el ángulo en el plano

    // Crea coordenadas polares de un objeto complex dado
    friend complex _Cdecl polar(double __mag, double __angle=0);

    // Funciones matemáticas del ANSI C sobrecargadas
    friend double  _Cdecl abs(complex&);
    friend complex _Cdecl acos(complex&);
    friend complex _Cdecl asin(complex&);
    friend complex _Cdecl atan(complex&);
    friend complex _Cdecl cos(complex&);
    friend complex _Cdecl cosh(complex&);
    friend complex _Cdecl exp(complex&);
    friend complex _Cdecl log(complex&);
    friend complex _Cdecl log10(complex&);
    friend complex _Cdecl pow(complex& __base, double __expon);
    friend complex _Cdecl pow(double __base, complex& __expon);
    friend complex _Cdecl pow(complex& __base, complex& __expon);
    friend complex _Cdecl sin(complex&);
    friend complex _Cdecl sinh(complex&);
    friend complex _Cdecl sqrt(complex&);
    friend complex _Cdecl tan(complex&);
    friend complex _Cdecl tanh(complex&);

    // Funciones de operadores binarios
    friend complex _Cdecl operator+(complex&, complex&);
    friend complex _Cdecl operator+(double, complex&);
        friend complex _Cdecl operator+(complex&, double);
    friend complex _Cdecl operator-(complex&, complex&);
    friend complex _Cdecl operator-(double, complex&);
        friend complex _Cdecl operator-(complex&, double);
    friend complex _Cdecl operator*(complex&, complex&);
        friend complex _Cdecl operator*(complex&, double);
    friend complex _Cdecl operator*(double, complex&);
    friend complex _Cdecl operator/(complex&, complex&);
        friend complex _Cdecl operator/(complex&, double);
    friend complex _Cdecl operator/(double, complex&);
    friend int _Cdecl operator==(complex&, complex&);
    friend int _Cdecl operator!=(complex&, complex&);
    complex& _Cdecl operator+=(complex&);
    complex& _Cdecl operator+=(double);
    complex& _Cdecl operator-=(complex&);
    complex& _Cdecl operator-=(double);
    complex& _Cdecl operator*=(complex&);
    complex& _Cdecl operator*=(double);
    complex& _Cdecl operator/=(complex&);
```

```
    complex& _Cdecl operator/=(double);
    complex _Cdecl operator+();
    complex _Cdecl operator-();

    // Implementación
    private:

    double re, im;
  };


// Funciones inline de complex

// ...

#endif  // __COMPLEX_H
```

Descripción de los métodos
==========

La sintaxis de las funciones matemáticas del ANSI C aplicadas a objetos de la clase compleja es similar así que no se van a describir aquí pues ya se explicaron en el tutor de C; lo única diferencia está en que donde aparece double en la función correspondiente al ANSI C, aparace el tipo complex en la correspondiente función matemática sobrecargada; por ejemplo, el prototipo del ANSI C

```
  double cos (double x);
```

se convierte en la función cos sobrecargada correspondiente a números complejos en

```
  complex cos (complex z);
```

Con los operadores sobrecargados para los objetos complejos ocurre lo mismo que las funciones matemáticas sobrecargadas para los objetos complejos.

Sí merece la pena explicar las nuevas funciones (métodos) aportados por la clase complex: real(), imag(), conj(), norm(), arg() y polar():

---
real Devuelve la parte real de un número complejo.
======
Sintaxis:

```
   double real (complex x);
```

---
imag Devuelve la parte imaginaria de un número complejo.
======
Sintaxis:

```
   double imag (complex x);
```

Los datos asociados a un número complejo están compuestos por dos números en coma flotante (double). La función imag() devuelve el que se considera parte imaginaria de los dos.

---
conj Devuelve el complejo conjugado de un número complejo.
======
Sintaxis:

```
   complex conj (complex z);
```

Devuelve el complejo conjugado del número complejo z.

---
norm Devuelve el cuadrado del valor absoluto.
======

```
Sintaxis:
   double norm (complex x);

La norma puede desbordarse por encima (overflow) si la parte real o la
parte imaginaria son suficientemente grandes.

-----
  arg    Da el ángulo de un número en el plano complejo.
=====
Sintaxis:
   double arg (complex z);

El eje real positivo tiene ángulo 0, y el eje imaginario positivo tiene
ángulo pi/2. Si z es 0, devuelve 0.

-------
  polar    Calcula el número complejo con la magnitud y el ángulo dados.
=======
Sintaxis:
   complex polar (double mag, double angulo);

El ángulo por defecto es 0. Estas dos declaraciones son iguales:

  polar (mag, angulo);

  complex (mag * cos (angulo), mag * sin (angulo));


Programa ejemplo
================

// Ejemplo de la clase complex.

#include <iostream.h>
#include <complex.h>

void main (void)
{
  complex z1 (2.2), z2 (3, 4);

  cout << "z1: " << z1;                   // imprime:  z1: (2.2, 0)
  cout << endl;
  cout << "z2: " << z2;                   // imprime:  z2: (3, 4)
  cout << endl;
  cout << "z1 + z2: " << z1 + z2;         // imprime:  z1 + z2: (5.2, 4)
  cout << endl;
  cout << "abs (z1-z2): " << abs (z1-z2); // imprime:  abs (z1-z2): 4.079216
  cout << endl;
  cout << "(z1 = z2-1): " << (z1 = z2-1); // imprime:  (z1 = z2-1): (2, 4)
  cout << endl;
  cout << "real (z2): " << real (z2);     // imprime:  real (z2): 3
  cout << endl;
  cout << "imag (z2): " << imag (z2);     // imprime:  imag (z2): 4
  cout << endl;
  cout << "conj (z1): " << conj (z1);     // imprime:  conj (z1): (2, -4)
  cout << endl;
}
```

```
    f f f f    i i i i    n     n    a a a a    l
    f             i       n n   n    a     a    l
    f f f         i       n  n  n    a a a a    l
    f             i       n   n n    a     a    l
    f          i i i i    n     n    a     a    l l l l
```