Introducción a C++

Este texto pretende ser una introducción a la programación orientada a objetos y al lenguaje C++, para gente que ya conozca el lenguaje C.

Este texto ha sido escrito por Nacho Cabanes. Si quiere conseguir la última versión, estará en mi página web:

www.nachocabanes.com

Para cualquier sugerencia, no dude en contactar conmigo a través de mi web.

Revisión actual: 0.04

Contenido

(Revisión actual: 0.04)

(Página dejada en blanco intencionadamente para cuando el índice sea más completo)

Revisiones de este texto hasta la fecha:

- 0.04, de 12-Jul-06. Ampliado hasta 13 apartados, 37 páginas.
- 0.03, de 28-Abr-06. Ampliado hasta 10 apartados, 28 páginas.
- 0.02, de 27-Abr-06. Incluye 7 apartados, 19 páginas.
- 0.01, de 26-Abr-06. Primera versión disponible, con 3 apartados, 12 páginas.

1. ¿Qué es la programación orientada a objetos?

El lenguaje C++ es un lenguaje creado con la base del lenguaje C, pero ampliado para permitir Programación Orientada a Objetos (también hay alguna otra diferencia, que ya iremos viendo). Por eso, vamos a empezar viendo las generalidades de la Programación Orientada a Objetos y después iremos particularizando para el caso de C++...

Hasta ahora estamos estado "cuadriculando" todo para obtener algoritmos: tratábamos de convertir cualquier cosa en funciones y variables que pudieramos emplear en nuestros programas.

Pero no todo lo que nos rodea es tan fácil de cuadricular. Supongamos por ejemplo que tenemos que describir una puerta desde nuestro programa. En la zona de declaración de variables detallaríamos datos como su tamaño o su color. Pero no basta con eso. De hecho, eso no es lo más importante: lo que hace que una puerta sea una puerta es el hecho de que se pueda abrir y se pueda cerrar. Por tanto, si queremos simular o controlar una puerta desde nuestro programa, deberíamos crear también las funciones AbrirPuerta y CerrarPuerta en alguna otra parte de nuestro fuente.

No está mal, pero es antinatural: una puerta es un conjunto: no podemos separar su color de su tamaño, o de la forma en que debemos abrirla o cerrarla. Sus características son tanto las físicas (lo que hasta ahora llamábamos variables) como sus comportamientos en distintas circunstancias (lo que para nosotros eran las funciones). Todo ello va siempre unido, formando un OBJETO.

Por otra parte, si tenemos que explicar a alguien lo que es el portón de un garaje, y ese alguien no lo ha visto nunca, pero conoce cómo es la puerta de su casa, le podemos decir "se parece a una puerta de una casa, pero es más grande para que quepan los coches, está hecha de metal en vez de madera...". Las dos cosas son puertas: se trata de dos objetos que pertenecen a la misma CLASE.

Finalmente, conviene recordar que "abrir" no se refiere sólo a una puerta. También podemos hablar de abrir una ventana o un libro, por ejemplo.

Pues con esta discusión hemos comentado casi sin saberlo las tres características más importantes de la Programación Orientada a Objetos (OOP):

- ENCAPSULACION: No podemos separar los comportamientos de las características de un objeto. Los comportamientos del objeto serán funciones, que en OOP llamaremos MÉTODOS. Las características del objeto serán variables, como las que hemos usado siempre, que en OOP llamaremos ATRIBUTOS. La apariencia de un objeto en C++, como veremos un poco más adelante, recordará a un registro o "struct", que contendrá funciones además de datos.

- HERENCIA: Unos objetos pueden "heredar" métodos y atributos de otros. Esto hace más fácil definir objetos nuevos a partir de otros que ya teníamos anteriormente (como ocurría con el portón y la puerta) y facilitará la reutilización de los programas, pudiendo aprovechar buena parte de los anteriores... si están bien diseñados.

- POLIMORFISMO: Un mismo nombre de un método puede hacer referencia a comportamientos relativamente distintos (como abrir una puerta o abrir un libro). En C++ el polimorfismo llega incluso al nivel de los operadores: podremos "redefinir" el operador + para que sume matrices, ficheros o cualquier tipo de objeto que nos interese[1].

  (Nota: en C++ es frecuente llamar también "variables de instancia" o "variables miembro" a los atributos, y "funciones miembro" a los métodos).

Comentado esto, vamos a empezar a ver ejemplos en C++ para tratar de fijar estos primeros conceptos y de ver la sintaxis de este lenguaje. A partir de unos primeros ejemplos sencillos, iremos profundizando paulatinamente.

2. Creando un objeto con C++

Vamos a empezar por hacer algo frecuente: crear un objeto "pantalla", que nos aisle totalmente del hardware, del sistema operativo y del compilador que estemos utilizando. Así, luego podremos definir un objeto "ventana" (por ejemplo), que muestre la información accediendo al objeto "pantalla".

Si después queremos que nuestro programa funcione en otro entorno distinto, bastará con crear una nueva versión del objeto "pantalla", que internamente trabaje de forma distinta, pero externamente responda a los mismos mensajes que el objeto "pantalla" inicial.

Así, todo el resto del programa funcionará en el nuevo entorno sin ningún cambio adicional.

En nuestro caso, nos centraremos en un objeto que nos oculte el acceso directo a la pantalla, y que estará particularizado (inicialmente) para el compilador Turbo C++, bajo MsDos. Después del fuente están los comentarios sobre él.

[1] Realmente, este tipo de polimorfismo recibe el nombre de "sobrecarga de operadores", como ya veremos.

```
//
//  pant01.cpp
//
//  Introducción a C++
//    Por Jose Ignacio Cabanes
//  Primer ejemplo (en un sólo fuente)
//  Versión para TC++ (MsDos)
//    Probado con Borland C++ 5.5 FCLT
//
////////////////////////////////////////////

#include <conio.h>

// ----------------------------------------------------------
// CLASE  : PantallaTexto
// Oculta el funcionamiento de la pantalla en modo texto.

class PantallaTexto {
  char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
  char maxY, minY;     // Coordenadas máxima y mínima (vertical)

  public:
  void Borra();             // Borra toda la pantalla
  void Escribe(int X,
    int Y, char *texto);    // Escribe un mensaje en ciertas coordenadas
  int LeeTecla();           // Espera una tecla y la devuelve
  int LeerMaxX();           // Devuelve coordenada X máxima
};

// ========================================================================
//  A continuación viene el desarrollo de las funciones (métodos)

// ----------------- PantallaTexto - Borra -----------------
// Borra toda la pantalla

void
PantallaTexto::Borra() {
  clrscr();
}

// ----------------- PantallaTexto - Escribe ---------------
// Escribe un mensaje en ciertas coordenadas

void
PantallaTexto::Escribe(int X, int Y, char *texto) {
  gotoxy(X, Y);
  cputs(texto);
}


// ----------------- PantallaTexto - LeeTecla --------------
// Espera una tecla y la devuelve

int
PantallaTexto::LeeTecla() {
  return getch();
}


// ----------------- PantallaTexto - LeerMax ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMaxX() {
  return maxX;
}


// ========================================================================
//  Finalmente: un programa sencillo de ejemplo

main() {
  PantallaTexto pant;    // Definimos un objeto de la clase "PantallaTexto"

  pant.Borra();
```

```
  pant.Escribe(30,14, "Hola");
  pant.LeeTecla();
  return 0;
}
```

Como este fuente es el primero que hacemos, hay muchas cosas que comentar:

- Ya en la primera línea, aparece texto después de una doble barra (//). Se trata de un **comentario** hasta fin de línea. Es un tipo de comentarios que existe en C++ y no en C: el comentario comienza después de // y termina cuando acaba la línea.

- Después definimos una **clase** de objetos: la clase PantallaTexto. Para eso usamos la palabra "class".

- Esta clase tiene ciertas variables (atributos), como por ejemplo las coordenadas máxima y mínima, que nos pueden interesar para poder saber si cierto texto cabrá en la pantalla, o para centrarlo.

- También será capaz de hacer ciertas cosas: borrarse o mostrar un texto, por ejemplo. Hemos añadido también la posibilidad de leer una tecla, que no está estrictamente relacionada con la pantalla, pero que nos va a resultar cómoda para hacer una pausa cuando queramos.

- Estas funciones (métodos) son **públicas** (lo que se indica con la palabra "public"): se podrá acceder a ellas desde fuera de la definición del objeto (desde un programa que use algún objeto de la clase PantallaTexto, por ejemplo). Por el contrario, las variables (atributos) son **privadas**: no se podrán leer ni modificar salvo desde los propios métodos del objeto. Esto DEBERIA HACERSE SIEMPRE ASI, para que el programa que use nuestro objeto sea independiente de cómo el objeto es internamente. (Nota: más adelante veremos más detalles sobre las partes públicas y privadas de una clase).

- Por eso, hemos definido el atributo "maxX" como "char", para que ocupe poco espacio en memoria, pero el método "LeerMaxX" devuelve un valor "int": si después nos damos cuenta de que no nos basta con un "char", porque trabajamos con pantallas capaces de mostrar muchísimo texto, redefinimos el dato "maxX" y lo convertimos en "int". El programa internamente se ha corregido, pero al usuario que emplea nuestro objeto pantalla no le afecta el cambio: antes leía un número entero y ahora sigue leyendo un número entero.

- Para detallar el **funcionamiento interno** de cada función (método), tenemos que indicar el nombre de la clase y el nombre del método, separados por dos símbolos de "dos puntos" (el conjunto :: se llama "operador de resolución de ámbito"):

```
void PantallaTexto::Borra()
```

- Ya en el programa de ejemplo, definimos un objeto de esta clase y **accedemos** a sus métodos empleando el nombre del objeto y el nombre del método, separados por un punto:

```
pant.Borra();
```

En C++, a esto se le suele llamar "Pasar un **mensaje** a un objeto": en nuestro caso, estamos pasando al objeto "pant" el mensaje de que se borre, a lo que el objeto responde siguiendo los pasos que se le han indicado en el método "Borra" (en nuestro caso, llamar a clrscr).

De hecho, si fueramos puristas, deberíamos haber creado incluso un objeto "aplicación", de modo que el programa fuese un conjunto de objetos que se envían mensajes unos a otros e interactuan.

3. Constructores y destructores

Este primer fuente ya hace algo, pero queda mucho por mejorar: nos podría interesar comprobar si un cierto texto va a caber en la pantalla, con algo como

```
if (pant.LeerMaxX() > 40)
  pant.Escribe(45,8, "Hola");
```

(si la pantalla tiene más de 40 letras en la horizontal, escribir el texto). Esto todavía no lo podemos hacer, porque no tenemos el valor del tamaño horizontal de la pantalla. Quien ya haya programado en C, se puede sentir tentado a dale un valor inicial haciendo

```
class PantallaTexto {
     char maxX=80, minX=1;      // Coordenadas máxima y mínima (horizontal)
    [...]
```

pero esto no es válido en C++. Eso sí, tenemos otra opción más elegante: podemos definir un método que se ponga en funcionamiento nada más crearse el objeto, y que será el encargado de dar valores iniciales a las variables, de reservar memoria dinámica si fuera necesario, etc. Este es el método **CONSTRUCTOR**, que se definiría de la siguiente forma:

```
PantallaTexto::PantallaTexto() {
    maxX = 80;   minX = 1;
    maxY = 25;   minY = 1;
}
```

(es decir, tiene como nombre el mismo que la clase a la que pertenece, y no devuelve valor de ningún tipo.

Si hubieramos reservado memoria dinámica (no es nuestro caso), también nos interesaría utilizar un método **DESTRUCTOR**, que liberase esta memoria cuando el objeto se deja de utilizar, y que tendría un aspecto parecido a esto:

```
PantallaTexto::~PantallaTexto() {
    // Aquí liberaríamos el espacio reservado, etc.
}
```

(es decir, el mismo nombre que la clase pero precedido por ~).

Pues ahora ya vamos a crear nuestra clase "PantallaTexto", pero esta vez un poco más completa.

En C++, lo habitual (y lo que se debe hacer, siendo purista) es dividir la definición de un objeto en dos partes:

- Las variables (atributos) y los prototipos de las funciones (métodos) se suelen definir en un **fichero de cabecera** (los que en C tenían extensión **.h**, y que en C++ sería preferible indicar con extensión .hpp ó .hh, aunque la mayoría de gente no lo hace).

- Los detalles concretos de cómo trabaja cada función (el desarrollo) se definen en un fichero fuente normal (los que en C tenían extensión **.c**, y que en C++ tendrán extensión .cpp ó .cc ó .C).

A pesar de ello, en este ejemplo todavía incluiremos todo junto en un único fichero, para tener una visión de conjunto algo mejor, y será a partir de siguiente cuando empecemos a hacer las cosas con más corrección.

Eso sí, aprovecharemos para ampliar un poco las funcionalidades de la clase: ahora no sólo podremos leer el valor máximo de X, sino también el mínimo, y los valores correspondientes de Y.

```
//
//  pant02.cpp
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//
//  Segundo ejemplo (en un sólo fuente)
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#include <conio.h>

// ------------------------------------------------------------
// CLASE  : PantallaTexto
// Oculta el funcionamiento de la pantalla en modo texto.

class PantallaTexto {
  char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
  char maxY, minY;     // Coordenadas máxima y mínima (vertical)

  public:
  PantallaTexto();          // Método constructor
  void Borra();             // Borra toda la pantalla
  void Escribe(int X,
    int Y, char *texto);    // Escribe un mensaje en ciertas coordenadas
  int LeeTecla();           // Espera una tecla y la devuelve
  int LeerMaxX();           // Devuelve coordenada X máxima
  int LeerMinX();           // Devuelve coordenada X mínima
  int LeerMaxY();           // Devuelve coordenada Y máxima
  int LeerMinY();           // Devuelve coordenada Y mínima
  ~PantallaTexto();          // Método destructor
};

// ============================================================================
//  A continuación viene el desarrollo de las funciones (métodos)
```

```
// ----------------- PantallaTexto -------------------------
// Método constructor: valores iniciales

PantallaTexto::PantallaTexto() {
  maxX = 80;   minX = 1;
  maxY = 25;   minY = 1;
}


// ----------------- PantallaTexto - Borra -----------------
// Borra toda la pantalla

void
PantallaTexto::Borra() {
  clrscr();
}

// ----------------- PantallaTexto - Escribe ---------------
// Escribe un mensaje en ciertas coordenadas

void
PantallaTexto::Escribe(int X, int Y, char *texto) {
  gotoxy(X, Y);
  cputs(texto);
}


// ----------------- PantallaTexto - LeeTecla --------------
// Espera una tecla y la devuelve

int
PantallaTexto::LeeTecla() {
  return getch();
}


// ----------------- PantallaTexto - LeerMaxX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMaxX() {
  return maxX;
}


// ----------------- PantallaTexto - LeerMinX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMinX() {
  return minX;
}


// ----------------- PantallaTexto - LeerMaxY ---------------
// Devuelve la coordenada vertical máxima

int
PantallaTexto::LeerMaxY() {
  return maxY;
}


// ----------------- PantallaTexto - LeerMinY ---------------
// Devuelve la coordenada vertical mínima

int
```

```
PantallaTexto::LeerMinY() {
  return minY;
}

// ----------------- PantallaTexto -------------------------
// Método destructor: no usado esta vez

PantallaTexto::~PantallaTexto() {
  // Aquí liberaríamos el espacio reservado, etc.
}


// ========================================================================
//  Finalmente: un programa sencillo de ejemplo

int main() {
  PantallaTexto pant;     // Definimos un objeto de la clase "PantallaTexto"

  pant.Borra();
  pant.Escribe(30,14, "Hola");
  pant.LeeTecla();
  return 0;
}
```

4. Separando la interfaz y la implementación.

Una vez que lo hemos visto funcionar todo junto, vamos a hacerlo usando tres ficheros distintos:

- En uno tendremos el "resumen" de la clase, también llamado "interfaz de la clase": los nombres (y los tipos) de cada atributo y cada método.
- En otro estará la "implementación" de la clase: los detalles sobre lo que hace cada método.
- Finalmente, en un tercer fuente estará el "main" de un pequeño programa de prueba.

Se parecerá mucho a lo que tendríamos si simplemente partiéramos el fuente en tres trozos, pero aparecerá alguna novedad…

Veamos en primer lugar cómo quedaría el fichero de cabecera:

```
//
//  pant03.h
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//
//  Tercer ejemplo (en dos fuentes)
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#ifndef _PANT3_H
#define _PANT3_H

// ---------------------------------------------------------
// CLASE  : PantallaTexto
// Oculta el funcionamiento de la pantalla en modo texto.

class PantallaTexto {
```

```
  char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
  char maxY, minY;     // Coordenadas máxima y mínima (vertical)

  public:
  PantallaTexto();          // Método constructor
  void Borra();             // Borra toda la pantalla
  void Escribe(int X,
    int Y, char *texto);    // Escribe un mensaje en ciertas coordenadas
  int LeeTecla();           // Espera una tecla y la devuelve
  int LeerMaxX();           // Devuelve coordenada X máxima
  int LeerMinX();           // Devuelve coordenada X mínima
  int LeerMaxY();           // Devuelve coordenada Y máxima
  int LeerMinY();           // Devuelve coordenada Y mínima
  ~PantallaTexto();          // Método destructor
};

#endif
```

Los cambios no son grandes: poco más que incluir un bloque "#ifndef... #endif" para evitar que este fichero se puda incluir varias veces si usamos a la vez varios fuentes que dependan de él.

El cuerpo de las clase quedaría en otro fichero, que tiene que incluir el anterior (indicado entre comillas, por tratarse de un fichero nuestro, no uno original del compilador):

```
//
//  pant03.cpp
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//
//  Tercer ejemplo (en dos fuentes)
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#include <conio.h>
#include "pant03.h"


// ===========================================================================
//  A continuación viene el desarrollo de las funciones (métodos)


// ----------------- PantallaTexto -------------------------
// Método constructor: valores iniciales

PantallaTexto::PantallaTexto() {
  maxX = 80;   minX = 1;
  maxY = 25;   minY = 1;
}


// ----------------- PantallaTexto - Borra -----------------
// Borra toda la pantalla

void
PantallaTexto::Borra() {
  clrscr();
}

// ----------------- PantallaTexto - Escribe ---------------
// Escribe un mensaje en ciertas coordenadas
```

```
void
PantallaTexto::Escribe(int X, int Y, char *texto) {
  gotoxy(X, Y);
  cputs(texto);
}


// ----------------- PantallaTexto - LeeTecla --------------
// Espera una tecla y la devuelve

int
PantallaTexto::LeeTecla() {
  return getch();
}


// ----------------- PantallaTexto - LeerMaxX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMaxX() {
  return maxX;
}


// ----------------- PantallaTexto - LeerMinX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMinX() {
  return minX;
}


// ----------------- PantallaTexto - LeerMaxY ---------------
// Devuelve la coordenada vertical máxima

int
PantallaTexto::LeerMaxY() {
  return maxY;
}


// ----------------- PantallaTexto - LeerMinY ---------------
// Devuelve la coordenada vertical mínima

int
PantallaTexto::LeerMinY() {
  return minY;
}

// ----------------- PantallaTexto -------------------------
// Método destructor: no usado esta vez

PantallaTexto::~PantallaTexto() {
  // Aquí liberaríamos el espacio reservado, etc.
}
```

Y el programa que lo usara podría ser

```
//
//  usapant.cpp
//
//  Introducción a C++
//    Jose Ignacio Cabanes
```

```
//
//  Tercer ejemplo (en dos fuentes)
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#include <conio.h>
#include "pant03.h"

// ========================================================================
//  Finalmente: un programa sencillo de ejemplo

main() {
  PantallaTexto pant;     // Definimos un objeto de la clase "PantallaTexto"

  pant.Borra();
  pant.Escribe(30,14, "Hola");
  pant.LeeTecla();
  return 0;
}
```

Este fuente (formado realmente por 3 ficheros) se puede compilar con Turbo C++ 1.01 para MsDos tecleando el nombre del compilador y a continuación el de los dos fuentes que no son ficheros de cabecera (el fichero de cabecera no es necesario indicarlo, porque se incluye automáticamente):

```
TCC  USAPANT.CPP  PANT03.CPP
```

Obtenemos un fichero llamado USAPANT.EXE.

De igual modo, podríamos compilarlo con Borland C++ 5.5 para Windows tecleando desde el símbolo del sistema la orden:

```
BCC32  USAPANT.CPP  PANT03.CPP
```

5. Extensiones de los ficheros

Ya hemos hablado de las **extensiones** de los ficheros. Vamos a concretar un poco más...

Estamos usando la extensión .cpp para los fuentes, pero hemos comentado que existen otras:

- **cpp** es la extensión estándar en MsDos, es decir que si a un compilador de C y C++ para MsDos le decimos que compile un fichero cuya extensión es .cpp, sabrá que se trata de un fuente en C++ y no en C.

- **cc** es la extensión estándar en Unix (si intentamos compilar un fuente con extensión cpp desde una versión antigua de Unix, es posible que el ordenador no reconozca esa extensión como una propia de C++, e intente compilarlo como si se tratase de un fuente de C, de modo que no funcionará).

- **C** (en mayúsculas) es otra extensión que se puede usar en Unix, pero que es menos aconsejable porque existen muchos sistemas operativos que no distinguen entre mayúsculas y minúsculas, lo que dará problemas de portabilidad (no se podrá distinguir los fuentes en C, con extensión ".c", de los de C++, con extensión ".C", si los llevamos a uno de estos sistemas; es el caso de MsDos y de Windows).

Las extensiones .hpp (bajo MsDos) y .hh (bajo Unix) son aconsejables para indicar que se trata de un fichero de cabecera de C++, pero no es imprescindible, de modo que muchos programadores usan de todas formas la extensión .h, que es el mismo criterio que seguiremos en este texto.

6. Especificadores de acceso

Aclaremos un poco más sobre los **especificadores de acceso:**

Hemos comentado que en el fichero .h los atributos son privados y los métodos son públicos. Está claro por qué son públicos los métodos: los hemos precedido con la palabra **public**. Así, desde cualquier otro objeto o función se podrá acceder a ellos.

Los atributos son privados (sólo podrán ser leídos o modificados por los métodos del propio objeto), y esto lo podemos conseguir precediéndolos con la palabra **private**. No lo hemos hecho así porque ese es el valor por defecto en C++: si no indicamos lo contrario, se asume que los atributos y métodos son privados.

Existe un tercer modo de acceso, que se denota con la palabra **protected**. Los atributos y/o métodos que definamos como protegidos serán accesibles sólo por los métodos de este objeto y por sus descendientes (dentro de poco veremos la herencia).

Hay una excepción a todo esto: podremos declarar funciones "amigas" (**friend**), que sí tendrán acceso a los miembros privados y/o protegidos, pero es algo que no trataremos... al menos por ahora.

7. Heredando y ampliando la clase base.

Ahora podemos plantearnos mejorar esta clase de objetos para "pantalla de texto", y añadirle una serie de posibilidades extras, o modificar ligeramente su comportamiento actual.

Pues aquí entra en juego una de las posibilidades más interesantes de la Programación Orientada a Objetos: la **reutilización**. No necesitamos partir de cero, sino que podemos crear una clase que tome como base la que ya existe, "heredando" de ella todas sus características. Podremos redefinir las que no nos interesen, o añadir otras nuevas.

Por ejemplo, vamos a crear una clase "pantalla de texto" mejorada, que nos permita además escribir un texto centrado en cierta línea de la pantalla.

Igual que antes, crearemos un fichero de cabecera, otro de desarrollo y otro que pruebe el resultado.

El fichero de cabecera será:

```
//
//  pant04.h
//
//  Introducción a C++
//  Jose Ignacio Cabanes
//
//  Cuarto ejemplo: Pantalla mejorada
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#ifndef _PANT4_H
#define _PANT4_H

#include "pant03.h"

// ------------------------------------------------------
// CLASE  : PantallaTM
// Pantalla en modo texto, mejorada
// ------------------------------------------------------
class PantallaTM: public PantallaTexto {
  public:
     void EscribeCent(
        int Y, char *texto);    // Escribe un mensaje centrado
};

#endif
```

Sólo hay una novedad: la clase PantallaTM es un caso concreto de "pantalla de texto", un descendiente de ésta. Por eso la declaramos diciendo que proviene de PantallaTexto (el indicador de acceso "public" es para que los métodos y atributos sigan siendo como en la clase "padre": en nuestro caso concreto, los atributos eran y seguirán siendo privados, y los métodos eran y seguirán siendo públicos).

El fichero de desarrollo de la clase "PantallaTM" será

```
//
//  pant04.cpp
//
//  Introducción a C++
//  Jose Ignacio Cabanes
//
//  Cuarto ejemplo: Pantalla mejorada
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#include <string.h>
#include "pant04.h"

// ========================================================================
//  A continuación viene el desarrollo de las funciones (métodos)

// ----------------- PantallaTM -------------------------
// Escribe un mensaje centrado en una línea

void
PantallaTM::EscribeCent( int Y, char *texto) {
     Escribe( (LeerMaxX() - LeerMinX() - strlen(texto)) / 2, Y, texto);
}
```

Para acceder a los valores máximo y mínimo de las coordenadas X e Y, hemos empleado las funciones LeerMaxX() y similares, dado que no estamos en la clase original PantallaTexto, sino en una que desciende de ella, por lo que no podemos acceder directamente a los atributos. Esta es una de las formas de trabajar "altamente recomendadas" en Programación Orientada a Objetos: la **ocultación de los detalles**, para que los cambios que podamos hacer a los atributos de una clase no afecten a las clases que se deriven de ella.

Si quisiéramos acceder directamente a los atributos (no es nada recomendable, es preferible la opción anterior), no deberíamos haberlos declarado como "private", sino como "**protected**", como ya se comentó anteriormente, para que las clases que "heredan" de la primera puedan acceder a ellos:

```
class PantallaTexto {
protected:
     char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
     char maxY, minY;     // Coordenadas máxima y mínima (vertical)

public:
    PantallaTexto();          // Método constructor
    // ...
```

con lo que el cuerpo del método EscribeCent quedaría

```
    Escribe( maxX - minX - strlen(texto)) / 2, Y, texto);
```

El ejemplo que usa esta clase "PantallaTM" podría ser:

```
//
//  usapantm.cpp
//
//  Introducción a C++
//  Jose Ignacio Cabanes
//
//  Cuarto ejemplo: Pantalla mejorada
//  Versión para TC++ (MsDos)
//    (este fuente se debe usar con
//    "pant03.cpp" y "pant04.cpp")
//
////////////////////////////////////////////

#include "pant04.h"

//  Programa sencillo de ejemplo

int main() {
  PantallaTM pant;     // Objeto de la clase "PantallaTexto" mejorada

  pant.Borra();
  pant.Escribe(30,14, "Hola");
  pant.EscribeCent(11, "Centrado");
  if (pant.LeerMaxX() > 40)
    pant.Escribe(45,8, "Hola");
  return 0;
}
```

Para compilarlo, nuevamente le indicamos al compilador el nombre de los ficheros fuente que debe combinar para crear el ejecutable:

```
bcc32 usapantm.cpp pant04.cpp pant03.cpp
```

En este caso, y al igual que ya hicimos con el ejemplo anterior, indicamos primero USAPANTM porque queremos que sea éste el fichero que dé nombre al ejecutable.

En la mayoría de compiladores, existe una posibilidad más cómoda que ésta de escribirle uno a uno los nombres de los fuentes implicados: podemos crear un "proyecto" e indicarle qué ficheros van a ser los necesarios para construirlo. Esto ayudará incluso a que el compilador sólo recompile los fuentes que realmente hayan cambiado, en vez de volver a crear todos, y así se gana en rapidez y comodidad.

8. Redefiniendo la clase base para otro sistema

Si queremos "adaptar" nuestra clase "PantallaTexto" para otro sistema (compilador y/o sistema operativo), tenemos dos opciones:

- Crear una nueva clase que herede de la básica pero que redefina ciertas cosas.
- Crear una nueva clase base alternativa, en la que modifiquemos lo que nos interese.

En nuestro caso, vamos a crear una versión para Linux, usando "ncurses". Como nuestre clase base incluye cosas que son específicas de MsDos, no nos bastará con redefinir parte, porque aun así nuestro compilador protestaría, diciendo que no conoce "conio.h", ni órdenes como "gotoxy".

Por eso, vamos a usar la segunda opción: crear una nueva clase base. Sólo necesitamos cambiar unas pocas cosas:

- El constructor deberá encargarse de entrar al modo texto mejorado con "initscr()".
- De igual modo, el destructor deberá salir con "endwin()".
- Para borrar la pantalla usaremos "clear()".
- Para escribir, usaremos "move" en vez de "gotoxy", "printw" en lugar de "cprintf", y al final actualizaremos la información en pantalla con "refresh()".

El resto de la clase no debería cambiar. Con todo esto, el fuente (el inicial, que conserva todo en un único fichero) quedaría así:

```
//
//  pant02L.cpp
```

```
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//  Segundo ejemplo (en un sólo fuente)
//  Versión para GCC (Linux)
//
////////////////////////////////////////////

#include <curses.h>

// ------------------------------------------------------------
// CLASE  : PantallaTexto
// Oculta el funcionamiento de la pantalla en modo texto.

class PantallaTexto {
  char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
  char maxY, minY;     // Coordenadas máxima y mínima (vertical)

  public:
  PantallaTexto();          // Método constructor
  void Borra();             // Borra toda la pantalla
  void Escribe(int X,
    int Y, char *texto);    // Escribe un mensaje en ciertas coordenadas
  int LeeTecla();           // Espera una tecla y la devuelve
  int LeerMaxX();           // Devuelve coordenada X máxima
  int LeerMinX();           // Devuelve coordenada X mínima
  int LeerMaxY();           // Devuelve coordenada Y máxima
  int LeerMinY();           // Devuelve coordenada Y mínima
  ~PantallaTexto();          // Método destructor
};

// ====================================================================
//  A continuación viene el desarrollo de las funciones (métodos)


// ----------------- PantallaTexto -------------------------
// Método constructor: valores iniciales

PantallaTexto::PantallaTexto() {
  maxX = 80;   minX = 1;
  maxY = 25;   minY = 1;
  initscr();
}


// ----------------- PantallaTexto - Borra -----------------
// Borra toda la pantalla

void
PantallaTexto::Borra() {
  clear();
}

// ----------------- PantallaTexto - Escribe ---------------
// Escribe un mensaje en ciertas coordenadas

void
PantallaTexto::Escribe(int X, int Y, char *texto) {
  move(Y, X);
  printw(texto);
  refresh();
}


// ----------------- PantallaTexto - LeeTecla --------------
// Espera una tecla y la devuelve

int
```

```
PantallaTexto::LeeTecla() {
  return getch();
}


// ----------------- PantallaTexto - LeerMaxX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMaxX() {
  return maxX;
}


// ----------------- PantallaTexto - LeerMinX ---------------
// Devuelve la coordenada horizontal máxima

int
PantallaTexto::LeerMinX() {
  return minX;
}


// ----------------- PantallaTexto - LeerMaxY ---------------
// Devuelve la coordenada vertical máxima

int
PantallaTexto::LeerMaxY() {
  return maxY;
}


// ----------------- PantallaTexto - LeerMinY ---------------
// Devuelve la coordenada vertical mínima

int
PantallaTexto::LeerMinY() {
  return minY;
}

// ----------------- PantallaTexto ------------------------
// Método destructor: no usado esta vez

PantallaTexto::~PantallaTexto() {
  // Aquí liberaríamos el espacio reservado, etc.
  endwin();
}


// ====================================================================
//  Finalmente: un programa sencillo de ejemplo

main() {
  PantallaTexto pant;  // Definimos un objeto de clase "PantallaTexto"

  pant.Borra();
  pant.Escribe(30,14, "Hola");
  pant.LeeTecla();
  return 0;
}
```

Para compilar este fuente desde Linux, deberíamos añadir la opción "-lncurses" para enlazar también las bibliotecas "ncurses" de acceso a pantalla.

En un caso general, si tenemos la clase desglosada en un fichero .h y un fichero .cpp, sólo sería necesario cambiar el fichero .cpp, porque el fichero .h no suele tener detalles que dependan del sistema.

9. Una clase que use la anterior.

Ahora vamos a crear una **nueva clase** de objetos que se utilice la anterior. Puede tratarse, por ejemplo, de una clase "ventana", que se encargue de mostrar ventanas en una pantalla de texto.

Cada ventana podrá tener un cierto tamaño, estar en cierta posición y contener una línea de texto. No he contemplado la posibilidad de que se puedan borrar las ventanas y el fondo de la pantalla quede restaurado como estaba, ni de que se pueda escribir más de una línea de texto dentro de la ventana (ni otras muchas).

Así, la cabecera podría ser ésta:

```
//
//  vent01.h
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//  Clase "Ventana"
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#ifndef _VENT1_H
#define _VENT1_H
#include "pant04.h"
// ----------------------------------------------------------
// CLASE  : Ventana
// Para dibujar en pantalla ventanas con una línea de texto.
// ----------------------------------------------------------
class Ventana {
    char maxX, minX;     // Coordenadas máxima y mínima (horizontal)
       char maxY, minY;     // Coordenadas máxima y mínima (vertical)
       char msg[80];        // El mensaje que mostrará la ventana
       char msgX, msgY;     // Coordenadas del mensaje en la ventana
       PantallaTM pant;     // La pantalla en la que escribir
  public:
       Ventana(PantallaTM p);          // Constructores: 3 distintos,
       Ventana(PantallaTM p, int x1,   //   según los parámetros
       int y1, int x2, int y2);
       Ventana(PantallaTM p, int x1,
       int y1, int x2, int y2, char* texto);
       int FijaCoords(int x1,     // Fija el tamaño
       int y1, int x2, int y2);
       void Escribe(int X,
       int Y, char *texto);     // Escribe mensaje en ciertas coords
       void Escribe(int Y,
       char *texto);            // Escribe un mensaje en una línea
       void Escribe(char *texto); // Escribe un mensaje centrado
       void Muestra();            // Muestra la ventana
};
#endif
```

Hay apenas una novedad interesante: hay 3 métodos constructores distintos y otros 3 métodos "Escribe" distintos, cada uno con distintos parámetros, y cada uno se comporta de forma ligeramente distinta. Es un ejemplo de **polimorfismo** (en este caso concreto, en que las 3 funciones tienen igual nombre y distintos parámetros, es un caso especial de polimorfismo llamado **sobrecarga**, se diría que la función Escribe "está sobrecargada").

El desarrollo de la clase "Ventana" podría ser éste:

```
//
//  vent01.cpp
//
//  Introducción a C++
//  Cuarto ejemplo
//    (usado por "ejcpp04.cpp")
//  Versión para TC++ (MsDos)
//
//  Curso de C
//  Jose Ignacio Cabanes
//
////////////////////////////////////////////

#include "vent01.h"
#include <string.h>

// ========================================================================
//  A continuación viene el desarrollo de las funciones (métodos)
//
// ----------------- Ventana --------------------------------
// Métodos constructores: valores iniciales

Ventana::Ventana(PantallaTM p) {
    maxX = 60;   minX = 20;
    maxY = 20;   minY = 6;
    msgX = 30;   msgY = 8;
    strcpy(msg,"");
    pant = p;
}

Ventana::Ventana(PantallaTM p, int x1, int y1, int x2, int y2) {
    maxX = x2;   minX = x1;
    maxY = y2;   minY = y1;
    msgX = 30;   msgY = 8;
    strcpy(msg, "");
    pant = p;
}

Ventana::Ventana(PantallaTM p, int x1, int y1, int x2, int y2,
  char *texto) {
    maxX = x2;   minX = x1;
    maxY = y2;   minY = y1;
    msgX = 30;   msgY = 8;
    strcpy(msg, texto);
    pant = p;
}
// ----------------- Ventana --------------------------------
// Fija las coordenadas, comprobando si son válidas

int
Ventana::FijaCoords(int x1, int y1, int x2, int y2) {
    if ((x1 < pant.LeerMinX()) ||
        (x2 > pant.LeerMaxX()) ||
        (y1 < pant.LeerMinY()) ||
        (y2 > pant.LeerMaxY()) ) return 0;
    maxX = x2;   minX = x1;
    maxY = y2;   minY = y1;
```

```
    return 1;
}

// ----------------- Ventana --------------------------------
// Escribe un mensaje en ciertas coordenadas

void
Ventana::Escribe(int X, int Y, char *texto) {
    msgX = X;
    msgY = Y;
    strcpy(msg, texto);
}

// ----------------- Ventana --------------------------------
// Escribe un texto centrado en una línea

void
Ventana::Escribe(int Y, char *texto) {
    char X = (maxX - minX - strlen(texto)) / 2;
    Escribe(X, Y, texto);
}
// ----------------- Ventana --------------------------------
// Escribe un texto centrado en la ventana

void
Ventana::Escribe(char *texto) {
    char Y = (maxY - minY) / 2;
    Escribe(Y, texto);
}
// ----------------- Ventana --------------------------------
// Muestra la ventana

void
Ventana::Muestra() {
    int i, j;
    // Borde superior
    for (i=minX; i<=maxX; i++)
      pant.Escribe(i, minY, "-");
    // Laterales que bajan
    for (i=minY+1; i<=maxY-1; i++) {
      pant.Escribe(minX, i, "|");
      pant.Escribe(maxX, i, "|");
    }
    // Borde inferior
    for (i=minX; i<=maxX; i++)
      pant.Escribe(i, maxY, "-");
    // Texto de la ventana
    pant.Escribe (minX+msgX, minY+msgY, msg);
}
```

Confío en que en este desarrollo no haya ninguna dificultad. Se apoya en parte en el objeto de clase "PantallaTm", y las rutinas "Escribe" se basan unas en otras. La rutina "Muestra" es sencilla aunque eso hace que tampoco el resultado sea muy lucido.

Finalmente, un ejemplo que usase las clases de objetos "Ventana" y "PantallaTM" podría ser:

```
//
//  usavent.cpp
//
//  Introducción a C++
//    Jose Ignacio Cabanes
//  Uso de clase Ventana
```

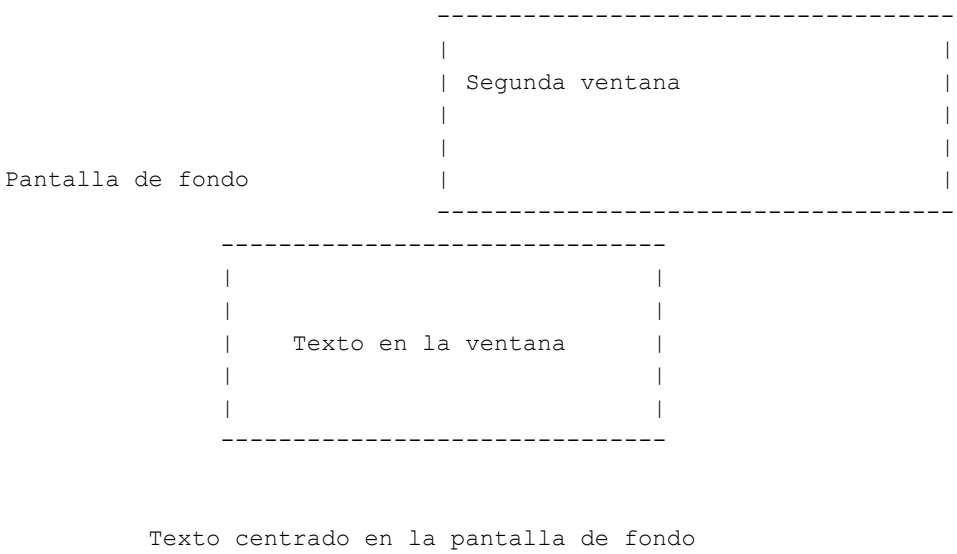
```
//  Versión para TC++ (MsDos)
//
////////////////////////////////////////////

#include "pant04.h"
#include "vent01.h"

//  Programa sencillo de ejemplo
PantallaTM pant;          // Objeto de la clase "PantallaTexto" mejorada
Ventana  vent1(pant);     // Primera ventana, vacía
Ventana  vent2(pant,      // Segunda, con coordenadas ya fijadas
    40, 5, 75, 11);
int main() {
    pant.Borra();
    pant.Escribe(10,10, "Pantalla de fondo");
    if (!vent1.FijaCoords(25, 12, 55, 18)) {
        pant.Escribe(10,20, "La ventana no cabe!");
        return 0;
    }
    vent1.Escribe("Texto en la ventana");
    vent1.Muestra();
    vent2.Escribe(2,2,"Segunda ventana");
    vent2.Muestra();
    pant.EscribeCent(23, "Texto centrado en la pantalla de fondo");
    return 0;
}
```

En este programa se ve la forma de utilizar los distintos métodos constructores para crear las ventanas, así como las dos llamadas a "Escribe" con distintos parámetros.

Para poner en marcha este programa, deberemos compilarlo creando un fichero de proyeto o indicando al compilador qué fuentes lo componen:

```
bcc32 usavent.cpp vent01.cpp pant04.cpp pant03.cpp
```

El resultado de este ejemplo tiene un aspecto parecido a éste:



Tampoco es demasiado vistoso, pero todavía somos principiantes en C++...

10. Otras diferencias entre C y C++

Hay muuuchos temas de Programación Orientada a Objetos que no hemos tratado, por ejemplo: sobrecarga de operadores (p.ej., redefinir el operador + para añadir texto a una ventana), herencia múltiple (que una clase herede métodos y atributos de más de un "padre"), etc. Algunos de estos temas "más avanzados" los veremos más adelante. Comentemos ahora otros aspectos más generales...

Primero repasemos alguna de las diferencias que ya hemos comentado.

- **Orientado a objetos**, con todos los cambios que ello implica, y en los que no voy a insistir.

- **Comentarios hasta fin de línea**, que se marcan con una doble barra //.

Y mencionemos alguna más:

- **Declaraciones después de instrucciones**. En C hay que declarar las variables locales al comienzo de una función; en C++ se pueden declarar después de sentencias ejecutables, lo que permite incluso declararlas justo en el momento de usarlas, con órdenes como

```
for (int i=1; i<=10; i++)
```

- Nuevas rutinas de **acceso a pantalla y teclado**: se lee del teclado con "cin >> variable" y se manda a pantalla con "cout << texto", como se muestra en el siguiente ejemplo:

```
#include <iostream.h>
char nombre[40];
main() {
  cout << "Teclea tu nombre\n";
  cin >> nombre;
  cout << "Así que te llamas " << nombre << ", verdad?\n";
  return 0;
}
```

- Los **struct** y **enum** se tratan como tipos. Así, se pueden definir variables de tipo "enum" y "struct" sin necesidad de repetir las palabras "enum" y "struct":

```
struct Complejo { float mod, arg };
struct Complejo c1;   // esto es lo que habría que hacer en C
Complejo c1;          // esto es válido en C++
enum Color { rojo, verde, azul };
enum Color unColor;   // esto es lo que habría que hacer en C
Color unColor;        // esto es válido en C++
```

- Se pueden dar **valores por defecto** a los parámetros de una función:

```
void Escribe(char *texto, int x=30, int y=10);
Escribe(saludo, 20, 5);  // Válido, escribe en (20,5)
Escribe(saludo, 20);     // Válido, escribe en (20,10)
Escribe(saludo);         // Válido, escribe en (30,10)
Escribe();               // No válido: no hay valor para "texto"
```

La restricción es que los valores por defecto deben corresponder a los parámetros que aparecen más a la derecha.

- Dispone de los operadores **new** y **delete** para reservar memoria dinámica, similares a malloc y free.

Por ejemplo, para crear dinámicamente una cadena de texto de 80 letras se haría

```
texto = new char[80];  // en vez de = (char *) malloc (80*sizeof(char))
```

y para eliminarla

```
delete texto;
```

- **Operador ::** para indicar el ámbito. Lo hemos utilizado ya para el desarrollo de los métodos de una clase, pero también tiene otros usos, como puede ser el de acceder a una variable global cuando existe una variable local con el mismo nombre:

```
int i = 5;
void funcion () {
  int i = 4;
  cout << "La variable local vale " << i;
  cout << "La variable global vale " << ::i;
}
```

- Se pueden declarar **referencias**:

```
int i = 0;
int &i2 = i;      // es un alias para i
i2 = 4;           // tiene el mismo efecto que i = 4
```

También se pueden pasar **parámetros por referencia**, que es más legible que pasar parámetros que sean punteros. Por ejemplo, para crear una función que calcule el triple de un número, lo podemos hacer de 3 formas:

Primera forma, pasando valores:

```
int tripleA( int n )   // Desarrollo de la función
{ return n * 3; }
int x, i = 5;
x = tripleA (i);       // Ahora x vale 15
```

Segunda forma, como puntero, pasando direcciones:

```
void tripleB( int *n )    // Desarrollo de la función
{ *n = (*n) * 3; }
int x, i = 5;
x = i;                    // Ahora x vale 5
tripleB (&x);             // Ahora x vale 15
```

Tercera forma, empleando referencias, más legible:

```
int tripleC( int& n )    // Desarrollo de la función
{ n = n * 3; }
int x, i = 5;
x = i;                   // Ahora x vale 5
tripleC (x);             // Ahora x vale 15
```

Y todavía más, pero como introducción es suficiente...

11. Más sobre entrada y salida en C++

Hemos visto la base de la escritura en pantalla y de la lectura desde teclado empleando C++: cout se usa para escribir en pantalla y cin para leer desde teclado. Los distintos textos que debe escribir **"cout"** se preceden con "<<", y las entradas de **"cin"** se preceden con ">>".

Pero en ese manejo básico han quedado muchas "lagunas", cosas que sabíamos hacer en C pero que no hemos visto cómo hacer en C++. Por ejemplo, no sabemos cómo mostrar un número con una cierta cantidad prefijada de cifras decimales, ni cómo leer una cadena de texto que contenga espacios en blanco, ni cómo acceder a ficheros.

Vayamos por partes:

Hemos usado dos "flujos" (streams) de datos: un flujo de entrada de datos (cin, que corresponderá al teclado) y un flujo de salida de datos (cout, la pantalla). Pero también existe otro flujo estándar, que no hemos empleado todavía, pero que es interesante conocer: es "**cerr**", que corresponde a la salida de los mensajes de error. Esta suele ser la pantalla (de modo que normalmente coincidirá con "cout"), pero en algunos sistemas operativos (como los Unix, entre ellos Linux) se pueden redirigir los mensajes de error a un fichero, de modo que se pueda analizar los errores independientemente de la salida habitual del programa por pantalla.

Por ejemplo, si manejamos bajo Unix una utilidad llamada "prueba", podemos hacer las siguientes cosas:

- Si tecleamos "prueba", la utilidad se pondrá en funcionamiento, y mostrará la información "normal" en pantalla, y los mensajes de error también.

- Si tecleamos "prueba > salida", la utilidad se pondrá en funcionamiento, pero la información que normalmente aparecería en pantalla no lo hará, sino que será "redirigida" a un fichero de texto llamado "salida".
- Si tecleamos "prueba 2> errores", la utilidad se pondrá en funcionamiento, y mostrará la información "normal" en pantalla, pero los mensajes de error serán "redirigidos" a un fichero de texto llamado "errores", que nosotros podríamos analizar después.

Este mismo funcionamiento estándar se puede conseguir desde los programas que nosotros creemos si enviamos los mensajes de error al flujo llamado "cerr" (eso sí, insisto en que esta característica debe permitirla el sistema operativo: lo permiten todos los Unix -Linux, FreeBSD, AIX, SCO, etc- pero no lo permite MsDos ni Windows).

Sigamos. La salida en pantalla se puede personalizar mediante una serie de **indicadores** ("flags", en inglés) o modificadores. Por ejemplo, uno de estos indicadores nos sirve para decir que queremos que los números enteros se muestren en **hexadecimal**: si escribimos

```
cout << 90 << "\n";
cout.setf(ios::hex);
cout << 90 << "\n";
```

el resultado de estas líneas de programa sería:

```
90
5A
```

En este caso hemos utilizado el indicador llamado "hex". Todos estos modificadores deberemos precederlos, como se ve en el ejemplo, por "ios::" (porque pertenecen a una clase llamada "ios", que es de la que derivan las clases istream y ostream que estamos usando "sin darnos cuenta" con cin y cout).

Los indicadores que tenemos disponibles son:

- dec: salida decimal para enteros (defecto).
- oct: salida octal para enteros.
- hex: salida hexadecimal al para enteros.
- left: la salida se alinea a la izquierda.
- rigth: la salida se alinea a la derecha.
- internal: se alinea el signo y los caracteres indicativos de la base por la izquierda y las cifras por la derecha.
- showpos: se muestra el signo (+) en los valores positivos.
- scientific: notación científica para coma flotante.
- fixed: notación normal para coma flotante.
- skipws: se descartan los blancos iniciales a la entrada.
- showbase: se muestra la base de los valores numéricos.
- showpoint: se muestra el punto decimal.
- uppercase: los caracteres de formato aparecen en mayúsculas.
- unitbuf: salida sin buffer (se vuelca cada operación).

Cuando queremos fijar uno de ellos, se hace como ya hemos visto:

```
cout.setf(ios::hex);
```

Si queremos fijar varios de ellos a la vez, podemos enlazarlos con la barra vertical (|) que representa la operación OR (suma a nivel de bits):

```
cout.setf(ios::left | ios::oct);
```

Esto es porque realmente cada uno de los **flags** corresponde a un bit de un cierto número entero:

```
enum {
  skipws=0x0001, left=0x0002, rigth=0x0004, internal=0x0008,
  dec=0x0010, oct=0x0020 hex=0x0040, showbase=0x0080,
  showpoint= 0x0100 uppercase=0x0200, showpos=0x0400, scientific=0x800,
  fixed=0x1000, unitbuf=0x2000
};
```

Nota: el valor que **devuelve** la función "setf" es la configuración que tenían anteriormente estos indicadores (un número de tipo "long"), de modo que podemos guardar dicho valor en una variable auxiliar, para después dejar las opciones de salida en pantalla tal y como estaban:

```
long configAnterior;
configAnterior = cout.setf(ios::hex); // Pasamos a hexadecimal, guardando config.
cout << 215;                          // Escribimos lo que queramos
...
cout.setf(configAnterior);            // Finalmente, dejamos config. antigua
```

Para **desactivar** un modificador concreto, se usa "unsetf" en lugar de "setf":

```
cout.unsetf(ios::hex);
```

También se puede usar la función **"flags"** para ver la condiguración actual (o para fijarla toda "de golpe", de modo que resulta más arriesgada que usar "setf"):

```
configActual = cout.flags();
```

Algunos de estos indicadores se pueden usar también en forma de **manipuladores**. Es el caso de hex:

```
cout << hex << 90 << "\n";
```

mientras que otros manipuladores disponibles no tienen correspondencia con ningún "flag". Por ejemplo, existe un manipulador llamado "endl" que da un salto de línea, de modo que la línea anterior también se podría haber escrito así:

```
cout << hex << 90 << endl;
```

Los manipuladores más habituales son:

- dec, hex y oct: fijar la base en que se mostrarán los enteros (decimal, hexadecimal u octal).
- ws: saltar los espacios en blancos iniciales.
- endl: saltar de línea ('\n') y vacíar el buffer de salida.
- flush: se vacía el buffer de salida.

También tenemos varias **funciones miembro** interesantes:

- **"width"** permite fijar la anchura mínima para un dato de salida (si el dato ocupa más, se toma mayor anchura automáticamente), y devuelve la anchura que estaba anteriormente prefijada. Esta anchura sólo es válida para el siguiente dato que se imprima.
- **"precision"** permite indicar el número de cifras de los datos númericos (por defecto son 6 cifras)y devuelve el valor de precisión anterior.
- **"fill"** elige el carácter de relleno para un dato de salida (que normalmente será un espacio en blanco) y devuelve el carácter de relleno anterior.

Un ejemplo de estas 3 funciones miembro sería:

```
cout << 123.43 << endl;
cout.width(12);
cout.precision(4);
cout.fill('@');
cout << 123.43  << endl;
```

que mostraría lo siguiente en pantalla

```
123.43
@@@@@@@123.4
```

12. Ficheros en C++.

Al igual que ocurre con la escritura en pantalla, a la hora de manejar los ficheros desde C++, podemos emplear las funciones que ya conocíamos de C, o bien emplear otras nuevas posibilidades que aporta C++. También al igual que ocurría con la pantalla, el manejo de ficheros se basará en flujos de entrada y salida.

Tenemos las clases fstream (fichero, en general), ifstream (fichero de entrada) y ofstream (fichero de salida), todas ellas definidas en **"fstream.h"**. Leeremos y escribiremos con << y >>, al igual que para la pantalla. Cerraremos un fichero con "close" (tanto si lo hemos abierto como para leer o para escribir) y comprobaremos si se ha terminado un fichero de entrada con "eof".

Vamos a ver un primer **ejemplo** que lo aplique, creando un fichero de texto:

```
//
//  fich01.cpp
//
//  Introduccion a C++
//  Ejemplo de ficheros (escritura)
//  Probado con BC++ 5.5 FCLT
//
//  Curso de C
//  Jose Ignacio Cabanes
//
////////////////////////////////////////////
#include <fstream.h>
main() {
  ofstream fichero("ejemplo.txt");
  fichero << "Hola" << endl;
  fichero << "Adios" << endl;
  fichero.close();
}
```

Debería ser muy fácil de seguir:

- Incluimos el fichero de cabecera "fstream.h".
- Definimos un fichero de salida, que tendrá por nombre físico "ejemplo.txt"
- Escribimos dos líneas de texto en el fichero.
- Cerramos en el fichero.

En un caso general, puede ocurrir que no sepamos el **nombre físico** del fichero en el momento de definir la variable "fichero", sino más tarde (por ejemplo, porque el usuario sea el que vaya a teclear el nombre del fichero con el que trabajar, o porque vaya a escoger dicho nombre en una ventana de diálogo). En ese caso, podemos usar la función miembro "open". Como ejemplo, vamos a leer el fichero que acabamos de crear:

```
//
//  fich02.cpp
//
//  Introduccion a C++
//  Ejemplo de ficheros (lectura)
```

```
//  Probado con BC++ 5.5 FCLT
//
//  Curso de C
//  Jose Ignacio Cabanes
//
////////////////////////////////////////////
#include <fstream.h>
#include <iostream.h>

main() {
  fstream fichero;
  char texto[200];
                              // Abro para lectura
  fichero.open("ejemplo.txt", ios::in);
  fichero >> texto;           // Leo una primera linea
  while (!fichero.eof())      // Mientras se haya podido leer algo
  {
    cout << texto << endl;    // Muestro lo que lei
    fichero >> texto;         // Y vuelvo a intentar leer
  }
  fichero.close();            // Finalmente, cierro
}
```

La estructura es ligeramente distinta, pero aun así, debería resultar fácil de seguir. Esta vez, el fichero lo hemos declarado como **"genérico"**, sin especificar si va a ser para lectura o escritura, de modo que este dato lo indicamos cuando realmente abrimos el fichero. Los modos de apertura que tenemos disponibles son:

- ios::in abre el fichero para lectura
- ios::out abre el fichero para escritura
- ios::append abre el fichero para añadir datos (al final, después de los que ya contenga)

Si hubiéramos declarado el fichero como **"ifstream"**, se daría por sentado que lo abrimos para leer, y no sería necesario indicarlo:

```
ifstream fichero;
// Abro para lectura
fichero.open("ejemplo.txt");
```

No hemos comprobado si el fichero realmente **se ha podido abrir**. Para conseguirlo, añadiríamos después de "open" algo parecido a esto, similar a lo que hacíamos en C estándar:

```
if (!fichero) {
  cerr << "No se ha podido abrir el fichero." << endl;
  exit(1);
}
```

Estas son las ideas básicas. Pero hay **más posibilidades**, que voy a comentar con menos detalle, pero prefiero que se sepa que existen, porque pueden ser útiles en muchos casos. Por ejemplo, podemos leer un bloque de datos de una determinada longitud, lo que será útil cuando manejemos ficheros binarios, o escribir una serie de bytes, o leer un dato de un flujo pero sin avanzar de posición.

Estas son algunas funciones miembro de iostream que nos servirán para cosas como esas:

- **put**(char c) escribe un carácter en un flujo de salida.

- **get**(char& c) lee un carácter de un flujo de entrada.
- **read**(char* s, int n) lee n bytes del flujo de entrada y los deposita en la cadena s (normalmente se usará para entrada binaria).
- **write**(const char* s, int n) escribe n bytes de la cadena s en un flujo de salida (normalmente se usará para salida binaria).
- **get**(char* s, int n, char c='\n') lee como máximo n caracteres del flujo de entrada (incluyendo el '\0') y los introduce en la cadena s, o hasta que encuentre el carácter de terminación (por defecto '\n', salto de línea), o el fin de fichero. No retira el carácter de terminación del flujo de entrada.
- **getline**(char* s, int n, char c='\n') lee como máximo n-1 caracteres del flujo de entrada, o hasta que encuentre el carácter de terminación (por defecto un final de línea) o hasta el fin de fichero. Retira el carácter de terminación del flujo de entrada, pero no lo almacena en la cadena s.
- **ignore**(int n=1, int delim=EOF) ignora o descarta los n caracteres siguientes de un flujo de entrada (o un solo carácter, si no se indica el valor de n), o hasta que encuentra un cierto carácter de terminación (por defecto el fin de fichero EOF).
- **peek**() lee un carácter del flujo de entrada pero sin retirarlo de dicho flujo.
- **putback**(char c) devuelve el carácter c al flujo de entrada (de modo que sería lo primero que se leería en la próxima operación de entrada).

Por otra parte, en la clase "fstream" tenemos otras funciones miembro que nos ayudarán a comprobar errores en la lectura o escritura:

- **good** () devuelve un valor distinto de cero si no ha habido ningún error.
- **eof**() devuelve un valor distinto de cero si se ha llegado al fin del fichero, como ya hemos visto.
- **bad**() devuelve un valor distinto de cero si ha habido un error grave de entrada/salida grave. No se puede continuar en esas condiciones.
- **fail**() devuelve un valor distinto de cero si ha habido cualquier error de E/S distinto de EOF. Después podemos llamar a bad() para comprobar si el error es grave o si se puede intentar proseguir la lectura. Si bad() devuelve 0, el error no es grave y la lectura puede proseguir después de llamar a la función clear().
- **clear**() resetea la situación de error (siempre que no sea grave), para poder seguir leyendo.

Además, también podemos comprobar si ha habido algún error en la forma que hemos empleado en el ejemplo anterior: cada función (open, read, etc) devolverá un valor **distinto de cero** cuando exista algún error, por lo que es habitual emplear construcciones como

```
  if (!fichero) {
     // No se ha podido abrir el fichero
  }
```

o como

```
if (!fichero.get(ch)) {
    // No se ha podido leer el siguiente dato.
 }
```

13. Los "strings" en C++.

Una de las características recientes que se ha añadido al estándar de C++ es la existencia de "strings" (cadenas de texto) como parte del lenguaje, con un manejo tan sencillo como lo es en otros lenguajes como Pascal o Basic.

En general, los compiladores del año 2000 o posteriores deberían permitir el manejo de cadenas, y la mayoría de los anteriores no lo harán. Eso sí, es posible que algún otro incluya alguna clase "string" no totalmente estándar, como las "AnsiString" de Borland C++ Builder.

Veamos su manejo primero con un ejemplo sencillo que nos introduzca los conceptos básicos:

```
//
//  string1.cpp
//
//  Introducción a C++
//  Cadenas en el nuevo estandar de C++
//  Probado con:
//    Borland C++ 5.5 FCLT
//
//  Curso de C
//  Jose Ignacio Cabanes
//
////////////////////////////////////////////

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

main()
{
 string mensaje;
 mensaje = "Hola";
 cout << mensaje;
}
```

La primera diferencia es la existencia de "named spaces" (espacios con nombre) dentro del nuevo estándar de C++. Se trata de una nueva forma de organizar los fichero de cabecera (ficheros ".h"), con la intención de que no sea un auténtico caos cuando tenemos muchos. Así, los ficheros de cabecera estándar están en el espacio llamado "std".

Se sigue pudiendo utilizar los fichero de cabecera de la forma clásica: podríamos escribir

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

aunque la forma recomendada (y que quiere decir lo mismo) es

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

El manejo básico de las cadenas no reviste mayor dificultad:

- Se declaran como cualquier otra variable en C y C++,: string cadena;
- Se les asigna valor con el signo = , como se hace con los números enteros o reales (o como las cadenas en otros lenguajes).
- Su valor se muestra en pantalla con "cout", igual que para las demás variables.

Así que veamos otro ejemplo un poco más complejo:

```
//
//  string2.cpp
//
//  Introducción a C++
//  Cadenas en el nuevo estandar de C++
//  Probado con:
//    Borland C++ 5.5 FCLT
//
//  Curso de C
//  Jose Ignacio Cabanes
//
////////////////////////////////////////////

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

main()
{
 string texto1, texto2 = "Hola ", texto3("Que tal");

 texto1 = texto2 + texto3 + " estas? ";
 cout << texto1 << "\n";
 string subcadena (texto1, 2, 6);  // 6 letras de texto1, desde la tercera
 cout << subcadena << "\n";
 string subcadena2;
 subcadena2 = texto1.substr(0, 5); // 5 letras de texto1, desde el comienzo
 texto1.insert(5, "Juan ");  // Inserto un texto en la posicion 6
 cout << texto1 << "\n";
 texto2.replace(1, 2, "ad");  // Cambio 2 letras en la posicion 2
 cout << texto2 << "\n";
 cout << "La longitud de texto1 es " << texto1.size() << "\n";
 cout << "La tercera letra de texto1 es " << texto1[2]
   << " o bien " << texto1.at(2) << "\n";
 if (texto2 == "Hada ")
   cout << "Texto 2 es Hada\n";
}
```

Casi todo se explica por sí solo. Aun así, vamos a dar un repaso rápido:

- Se puede crear una cadena sin valor inicial haciendo `string texto1;`
- Se le puede dar una valor inicial a la vez que se declara, haciendo `string texto2 = "Hola ";` o bien `string texto3("Que tal");`
- Se puede crear una cadena formada por varias, concateándolas (sumándolas), usando el signo +, así: `texto1 = texto2 + texto3 + " estas? ";`
- Se puede crear una subcadena a partir de un trozo de otra, la vez que se declara, así: `string subcadena (texto1, 2, 6);`
- O bien se puede extraer un fragmento posteriormente: `texto1.substr(0, 5);`
- Se puede insertar texto en el interior de una cadena: `texto1.insert(5, "Juan ");`
- O reemplazar ciertas letras por otras: `texto2.replace(1, 2, "ad");`
- Se puede saber el tamaño (cantidad de letras) de la cadena: `texto1.size()`
- Se puede acceder a una posición siguiendo el estándar de C: `texto1[2]`
- O bien usando la función "at": `texto1.at(2)`

- Se puede comprobar el valor de una cadena (el texto almacenado) con ==, así: `if (texto2 == "Hada ") ...`

El resultado de este ejemplo es:

```
Hola Que tal estas?
la Que
Hola Juan Que tal estas?
Hada
La longitud de texto1 es 25
La tercera letra de texto1 es l o bien l
Texto 2 es Hada
```

Esto es lo básico de las cadenas de texto en C++. Hay más (por ejemplo, un método "find" para ver si pertenece una cierta subcadena), pero eso queda propuesto como "ejercicio de investigación" para quien quiera profundizar más.