PROGRAMACION EN C++

¿QUE ES LA PROGRAMACION ORIENTADA A OBJETOS?

En la programacion estructurada todos los programas tienen las estructuras secuencial, repetitiva o condicional.Tambien se utilizan los TAD (Tipos Abstractos de Datos) para por ejemplo una pila o un arbol.

```
typdef struct{
  int x,y;
  int color;
  }punto;

struct punto a,b;
```

luego se implementan las funciones de este TAD (pila_vacia, pila_llena).

En C++ se definen los TAD y las funciones o procedimientos y datos dentro de un mismo conjunto llamado **class** (clase).En el ejemplo, el typedef struct punto seria el equivalente en C de la class de C++ y las variables a y b de los objetos en C++

CLASES (CLASS)

Antes de poder definir un objeto debemos definir la clase a la que pertenece (igual que en el ejemplo anterior debemos definir antes la estructura punto para luego poder definir las variables a y b). La forma general de describir una clase seria mas o menos:

```
class nombre_clase  {
        datos y funciones privados;
public:
        datos y funciones publicos;
        funcion constructora;
        funcion destructora;
};
```

Los datos y funciones privados son los que no se puede acceder a ellas desde las funciones que son miembros de la clase (que estan definidas en ella), se comportan igual que las variables definidas localmente en una funcion en C normal. En cambio, los datos y las funciones publicas son accesibles por todas las funciones

del programa (igual que si estuviesen definidas las varibles globalmente en C normal).

Por defecto, en una clase se define todos los datos y funciones privados, a menos de que le especifiquemos las que son publicas con la instruccion public.

Para saber si una funcion debe ser definida publica o privada, debemos ver si el resto del programa necesita “conocer como funciona” dicha funcion. Si la respuesta es “si” entonces la funcion debe ser publica, en caso contrario debe ser privada. Como despues de leer este rollo no te habras enterado (con esta explicacion no me extraña), un ejemplo:

-Vamos a crear la clase CRONOMETRO:

```
class CRONOMETRO{
        struct time tm;        // Variable que coge la hora del sistema
        int andando;
        void dif_tm(time tr, time ts, time *dif);
public:
      void Arranca(void);
     void Muestra(void);
     void Para(void);
};

CRONOMETRO p; (p sera un objeto de la clase cronometro);
```

La funcion dif_tm es privada porque al resto del programa no le interesa acceder a ella, en cambio es las funciones *Arranca*, *Muestra* y *Para* si pueden acceder a ella porque necesitan saber la diferencia entre dos tiempos (sobre todo la funcion Muestra, que es la que muestra dicha diferencia). Para llamar a una funcion de la clase desde una parte del programa que no pertenece a dicha clase, se utiliza *nombre_objeto.funcion;*

FORMATOS DE CIN Y COUT

En C++ se empieza a utilizar el concepto de fujos de E/S y no son mas que una clase (son dos clases: CIN y COUT). Las salidas con formatos de COUT:

```
cout <<formato<<variable;    // equivale a      printf("formato",variable);
```

dec = %d (enteros)
hex = hexadecimal
oct = octal
endl = "\n" (en CIN y COUT se puede seguir poniendo "\n")

ends = '\0' (inserta el fin de cadena, un NULL, no el caracter cero)
setw (num) fija la anchura de un campo en n bytes;

cout<<setw(6)<<variable

setprecision(n) = fija el numero de decimales que queremos (n).

todas estas cosas estan en la libreria <iomanip.h>

***Sacar un numero en tres formatos: hexadecimal, octal y decimal:**

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

int num;
cin<<num;
cout<<"el numero decimal es: "<<dec<<num<<endl;  (endl="\n")
cout<<"el numero octal es: "<<dec<<oct<<endl;    (endl="\n")
cout<<"el numero decimal es: "<<dec<<hex<<endl;  (endl="\n")
```

CONSTRUCTOR Y DESTRUCTOR

Lo mas normal es que un objeto tenga que reinicializarse, para hacerlo usamos el contructor (tiene el mismo nombre de la clase); no es necesario que se creen (en este caso el compilador utiliza el constructor explicito que inicializa los dato en cero o NULL, segun sea el tipo de dato); cuando se encuentra una declaracion de un objeto, el compilador llama al contructor.No se debe llamar explicitamente al contructor. El formato de la funcion constructora es

nombre_clase (parametros);

Podemos definir la funcion constructora para que inicialice ciertas variables, pedir memoria dinamica, etc.El destructor tampoco se debe llamar explicitamente, y el compilador lo llama cuando antes de terminar el programa (cuando llega a la ultima llave del main) .Si no existe el destructor, el compilador llama a uno por defecto. Su formato es:

~nombre_clase (parametros);

Si hemos reservado memoria dinamica en la funcion constructora, podemos aprovechar y liberarla en la destructora.

OPERADOR ::

Sirve para especificar a que clase pertenece la funcion que estamos tratando, es decir:

```
class fichero{
        char *fich1,*fich2;
        FILE *entrada,*salida;                  } datos y prototipos de funciones
        void abrir_ficheros(void);              } privados
        char encriptar_desencriptar(char);
public:
        void ordenar(void);
       fichero(void);                           } prototipos de las funciones publicas
        ~fichero(void)
}

fichero a;     // Definimos un objeto a de la clase fichero

void fichero::operar(void)       // Tenemos que especificar a que clase pertenece
{
 ....
 ....
}
fichero::fichero(void)             // Declaracion de la funcion constructora
{
 ...  // Aqui va lo que hace la funcion constructora: incializar variables
 ...  // (solo si es a un valor <> 0, reservar memoria, etc
}
fichero::~fichero(void)          // Declaracion de la funcion destructora
{
 ...
 ...
}
```

Para llamar a las funciones es igual que en C normal, no hace falta poner "*nombre_clase::* ", pero si en las definiciones de las funciones que estan fuera de la clase.

MANEJO DE FICHEROS

El manejo de ficheros en C++ es diferente que en C normal. Igual que C normal solo hay un tipo de flujo (definido como FILE *), en C++ hay tres tipo de flujo:

```
ifstream fich_ent;     // Definimos fich_ent como flujo de entrada
ofstream fich_sal;     // Definimos fich_sal como flujo de salida
fstream fich_entrada_salida;   // Definimos fich_entrada_salida como flujo de
                               // entrada y salida.
```

Igual que en C normal una vez definido el flujo (con la instruccion FILE *), tenemos varias funciones para manejar archivos:

*void open(char *nombrearchivo, int modo, int acceso);*

Esta funcion abre un fichero. Hay que pasarle unos parametros:

nombrearchivo: nombre del archivo a abrir.
modo: dice como se abre el archivo, puede ser uno o varios de estos:

-**ios::app** todo lo que escribimos en el archivo se añade al final (pero hay que poner el ptro al final del archivo con fseek)

-**ios::ate** añadiendo esto a ios::app ya nos pone automaticamente el ptro al final del archivo

-**ios::binary** abre el fichero para operaciones binarias, pero no convierte el valor leido a caracter (si lee el numero 65 no nos devuelve 'A' sino 65)

-**ios::in** abre el fichero para operaciones de entrada (es decir para leer datos del fichero). Solo se puede utilizar con un flujo de entrada (ifstream)

-**ios::out** abre el fichero para operaciones de escritura (para escribir datos en el fichero). Solo se puede utilizar con un flujo de salida (ofstream)

-**ios::nocreate** hace que se produzca un fallo si la funcion open() si el archivo especificado no exista

-**ios::noreplace** igual que la anterior provoca un fallo en open() si el archivo ya existia

-**ios::trunc** destruye el contenido del archivo, si existia previamente, y deja la longitud del archivo en cero

Para unir varios modos se utiliza la "o logica" (||).

acceso: indica de que tipo es el archivo al cual vamos a acceder.

Hay cinco tipos de acceso::

0 Archivo normal
1 Archivo de solo lectura
2 Archivo oculto
4 Archivo del sistema
8 Bit de archivo activo (solo se usa en entornos multitarea para saber el archivo activo)

Todo este rollo se ve mejor con un ejemplo:

```
#include <iostream.h>
#include  <fstream.h>
ifstream entrada;  // Definimos un flujo de entrada
ofstream salida;  //Definimos un flujo de salida

entrada.open("pepe.dat", ios::in, 0); // Abrimos el fichero en modo lectura
salida.open("pepito.dat", ios::out,0); // Abrimos el fichero en modo escritura
entrada.close(); // Cerramos pepe.dat
salida.close(); //Cerramos pepito.dat
```

Todas estas definiciones y funciones estan definidas en <fstream.h>

CONSTRUCTORES PARAMETRIZADOS

Son los contructores de siempre, pero a los que le pasamos un parametro. Pueden ser utiles en algunos casos (por ejemplo para reservar la memoria justa para una cola, en vez de definir un numero fijo de elementos). Bueno, vamos a ver como se define una funcion contructora parametrizada:

```
class punto{
        int *coordenadas   // Aqui guardamos las coordenadas del punto
 public:
        punto(int);         // Constructor parametrizado al que le pasamos el
                            //  numero de coordenadas del punto ( 2 o 3 )
        ~punto(void);     // Funcion destructora
 };

 punto::punto(int num_coord)
 {
```

```
  coordenadas=new int [num_coord];  // Reservamos memoria para num_coord
 }                                  // coordenadas ( que seran 2 o 3 )

punto::~punto(void)
{
  delete (coordenadas); // Liberamos la memoria que reservamos en la funcion
 }                      // constructora
```

Para llamar a las funciones constructoras parametrizadas, hay dos formas:

```
clase nombre_objeto=clase ( parametros );      punto inicio=punto ( 3 );
clase nombre_objeto ( parametros );            punto inicio ( 3 );
```

La forma mas comoda de hacer la llamada, por lo menos para mi, es la segunda. En el listado anterior, podemos definir un objeto (un punto) de dos o tres coordenadas. Con los constructores parametrizados, le decimos de que tipo va a ser el punto (bidimensional o tridimensional), de otra manera tendriamos que definir el punto tridimensional, ahorrando asi 1 byte. Pues vaya mierda, ahorrar 1 byte de memoria!, direis. Pero si con nuestros grandes conocimientos de C hacemos un programa que calcule figuras, de unos 20.000 puntos cada una, si la figura es tridimensional, no ahorraremos nada; pero si es bidimensional ahorraremos 20.000 bytes (19.5 K). No esta mal, verdad?.

SOBRECARGA DE OPERADORES

Esto se utiliza para hacer que un operador (+, -, *, /, etc) haga mas cosas que sumar numeros enteros, decimales, etc. Podemos hacer que nos sume matrices, vectores, etc. Si definimos el operador + para que sume matrices, no dejara de hacer lo que hacia antes (sumar enteros, decimales, etc).

La forma para definir la funcion sobrecargada es:

```
tipo_a_devolver nombre_clase::operator( parametros );
```

Vamos a ver un ejemplo para sumar dos vectores:

```
class vector{
      int x, y, z;
   public:
      vector vector::operator+( vector );
};
```

```
vector vector::operator+( vector p )
{
  x=p.x+x;
  y=p.y+y;
  z=p.z+z;
  return( *this );
 }

void main(void)
{
  vector a, b, c;

  c=a+b;
 }
```

Este ejemplo tiene mas “cosillas” que los anteriores. Cuando en el *main* hacemos la llamada *a+b*, el compilador lo transforma en *a.operator+(b)*, cuando vamos a la funcion, el vector p es una copia del b (o sea p.x=b.x, p.y=b.y, p.z=b.z) y las coordenadas x, y, z son las del vector a, que es el objeto con el cual llamamos a la funcion. El objeto por defecto es el que queda a la izquierda del operador y el objeto que normalmente se suele pasar es el de la derecha. Lo de *this* es muy sencillo, this es un puntero al objeto por defecto, es decir el objeto que llama a la funcion. En el ejemplo anterior, al llamar a la funcion +, el objeto que llama a la funcion es a, por lo que this apuntara a “a”. Como la funcion devuelve un vector, y this es un puntero a un objeto tipo vector tendremos que devolver el contenido de this (*this=a). Si la llamada a la funcion es del tipo *d=a+b+c*, el resultado de la suma de los vectores b y c, quedaria guardado en b, pudiendo afectar a calculos posteriores que se hagan con este vector.

Para evitar esto, habria que declarar un vector auxiliar local a la funcion, quedando de la forma:

```
vector vector::operator+( vector p )
{
  vector h;


  h.x=p.x+x;
  h.y=p.y+y;
  h.z=p.z+z;
  return( h );
 }
```

FUNCIONES AMIGAS

Son funciones que no pertenecen a la clase (estan definidas en la parte publica de la clase), pero que tienen acceso a la parte privada de la clase. La forma de definir una funcion amiga es:

friend tipo_objeto_devuelve nombre_funcion (parametros)

La utilidad de estas funciones, son por ejemplo, en el caso de que queramos sumar un entero con un vector, siendo la suma de la forma entero+vector. Si no utilizamos las funciones amigas, si sumamos 1 al vector a (1+a), el compilador la transforma en 1.operator+(a), pero esto no funcionaria porque 1 es un entero y no un objeto.
Aparte de esto, la funcion debe poder acceder a la parte privada de la clase vector, que son las componentes x, y, z de cada vector (si no, no podriamos sumar). Con todo esto, la declaracion de las funciones amigas seria:

```
class vector{
        int x,y,z;
  public:
        friend vector operator+(int, vector);
};

vector operator+(int num, vector p)
{
 vector h;

 h.x=p.x+num;
 h.y=p.y+num;
 h.z=p.z+num;
 return(h);
}
```

Como no pertenece a la clase al hacer la llamada en el main, se llama como una funcion normal en C, ni en la definicion de la funcion poner vector:: .

CONSTRUCTORES DE COPIA

Cuando inicializamos un vector utilizando otro y ese objeto trabaja con punteros (asigna memoria dinamicamente, abre ficheros, etc) el compilador lo que hace es

copiar bit a bit el objeto. Si estamos utilizando punteros nos encontramos con que hay dos punteros que apuntan a la misma zona de memoria, y esto puede dar problemas (o sea puede el segundo objeto sobreescribir el contenido del puntero

del primer objeto). Como siempre, despues del rollo viene el ejemplo:

```
class libro{
        char *titulo;
        int año;
   public:
        void Introduce_libro(libro);
        libro(void);
};
 void libro::libro(void)
 {
  titulo=new char [100];
 }
 void main(void)
 {
  ejemplo libro1;

  ejemplo libro2=libro1;
 }
```

En este ejemplo nos encontramos con que libro1.titulo y libro2.titulo apuntan a la misma zona de memoria, con lo que al pedir el titulo del segundo libro, lo vamos a guardar en el mismo sitio que el primero, perdiendo el titulo del primer libro. Para evitar esto utilizamos los constructores de copia, la forma de definirlos es:

```
nombre_clase (const nombre_clase &objeto);      libro (const libro &p);
```

añadiendo esto el codigo quedaria igual en a parte publica de la clase, pero habria que definir la funcion constructora de copia:

```
 void libro::libro(const libro &p)
 {
  titulo=new char [100];
  año=p.año;
 }
```

Con esta solucion el tiulo de libro1.titulo no comparte el mismo sitio de memoria que libro2.titulo. Las llamadas a este constructor las hace ya el compilador cuando encuentra una instruccion del tipo libro libro2=libro1.

SOBRECARGA DE INSERTORES Y EXTRACTORES

Cuando definimos una clase (por ejemplo un vector de tres componentes) y

queremos mostrar las componentes de cada vector, tenemos que definir una funcion

de lectura y otra de escritura para leer y mostrar datos de este tipo. Pero ahora se acabo el definirlas ! (igual que en el anuncio de la tele). Para no tener que definirlas usamos la sobrecarga de insertores y extractores. Los extractores son los churros >> y los insertores son los << . La forma de definir la sobrecarga de insertor es:

```
friend ostream &operator(ostream &stream, tipo_clase objeto)
```

Se debe definir como funcion amiga porque si no no podria acceder a las partes privadas de la clase (es decir a los componentes del vector) y no podria mostrar ni guardar dichas componentes. Las dos funciones devuelve un flujo con los datos para que pueda ser utilizado por cin o cout.

```
ostream & operator<<(ostream &stream, tipo_clase objeto)
{
 // Aqui ponemos lo queramos que haga
 return stream;
}
```

La forma del prototipo del extractor es similar:

```
friend istream &operator>>(istream &stream, tipo_clase &objeto)
```

y la definicion de la funcion tambien:

```
istream &operator>>(istream &stream, tipo_clase &objeto)
{
 // Aqui va el codigo
 return stream;
}
```

En esta funcion pasamos objeto por referencia (con el simbolo &) y esto es porque como vamos a guardar los datos en dicho objeto. Mientras ejecutemos la funcion el objeto va a contener los valores que le metemos nosotros, pero al finalizar como es una copia del objeto original, esta se destruiria y se perderian los valores. Todo esto, aunque parezca mentira tiene una utilidad. Sirve para que cuando nosotros ponemos en el main cout<<b siendo b un vector de tres componentes, nos muestre las tres componentes del vector (igual que hace cuando le ponemos cout << a siendo a un entero). Las funciones sobrecargadas de insercion y extraccion no pueden ser miembros de la clase, pueden ser como mucho funciones amigas (para poder acceder a los datos privadas de dicha clase) o funciones independientes.

SOBRECARGA DE OPERADORES MONARIOS

Son los que axtuan sobre un solo dato (no como por ejemplo la suma, que se necesitan dos tipos de datos para sumar, o sea son ++, —, etc). Los operadores monarios, al igual que los binarios se pueden sobrecargar. Hay dos formas de sobrecargar un operador, con una funcion miembro de la clase y como una funcion amiga.

Para sobrecargar un operador con una funcion miembro de la clase, basta con ir al apartado de SOBRECARGA DE OPERADORES y leerlo. Para definirla con unafuncion amiga, hacemos lo mismo (vamos al apartado de las FUNCIONES AMIGAS). Hay que recordar que si la funcion sobrecargada es definida como miembro, y es un operador binario, hay que pasarle parametros; si es un operador monario no se le pasa ningun parametro. En caso de ser una funcion amiga, y ser un operador binario, hay que pasarle dos parametros; si es operador monario, se le pasa uno. Si despues de todo esto, sigues leyendo, igual te aclaras mas con un cuadro explicativo (explica poco, pero bueno):

Numero de parametros a pasar a las funciones a sobrecargar	*OPERACION*	*OPERADOR MONARIO*	*OPERADOR BINARIO*
Funcion miembro de la clase	a+b a++	—— a.operator++(void)	a.operator+(b) ——
Funcion amiga de la clase	a+b a++	—— operator ++(a)	operator+(a,b) ——

Una ultima cosa, debemos, siempre que sea posible, usar las funciones amigas, porque lo que el C++ trata de hacer es la encapsulacion (tener juntas en la parte privada de la clase los datos y las funciones que manejan, siendo estos inaccesibles desde fuera de la clase).

SOBRECARGA DE [ ]

Este operador tambien se puede sobrecargar (los unicos que no se pueden sobrecargar son ., ::, ?, sizeof).Como ya sabemos cuando escribimos:

x [i] el compilador lo convierte en *x.operator[ ](i)*

Hay una serie de restricciones que debemos de tener en cuenta:

-No se puede construir operadores propios, solo se pueden sobrecargar los operadores definidos en C
-No se puede cambiar la preferencia o asociatividad de los operadores
-No se puede cambiar un operador binario para funcionar con un unico objeto

FUNCIONES DE CONVERSION

Si queremos asignar una matriz a un entero, podemos hacer dos cosas: sobrecargar la funcion =, o declarar una funcion de conversion. EL problema de definir la funcion sobrecargada =, es que si en el main hay una instruccion del tipo *cout<<(int)b* no funcionaria, porque no sabria como asignar una matriz a un entero, debido a que

esto lo tenemos definido en la funcion sobrecargada =.La forma de definir una funcion de conversion seria:

```
operator tipo_dato();
```

Todo esto se ve mejor con un ejemplo:

```
class vector{
      int componentes[MAX];
      int orden;
   public:
      ...
      ...
      operator float();
};

vector::operator float()
{
 int i;
 float f;

 for(i=0;i<orden;i++)
  f=f+componentes[i];
 return (f/orden);
}
```

En este caso al hacer *cout<<(float)a* nos devolveria la media aritmetica de la suma de los componentes del vector. Las funciones de conversion deben ser funciones miembro de la clase.

Si todavia no has tirado estos apuntes a la basura, felicidades. Pero, ya sabes, despues de navidades mas C++!!

HERENCIA

La herencia permite la reutilizacion de codigo ya escrito. No necesitamos el codigo fuente, a partir del obj podemos hacer una clase derivada y completarla con mas funciones. Es algo parecido al arbol de directorios del DOS.

Clase base (clase de la que partimos)
^
Clase derivada (clase que heredan las caracteristicas de la clase base)
^
Clase derivada

La sintaxis es:

```
class base {
        ...
        ...
};
class derivada:base {
        ...
        ...
};
```

Lo que hereda una clase derivada de una clase base son funciones y datos (miembro) publicos (privados no).Desde la clase derivada se tiene acceso a los datos y funciones publicas de la base.La clase base no conoce a la clase derivada.

DATOS PROTEGIDOS (Protected): los datos o funciones que esten definidos asi, son accesibles desde las funciones de las clases derivadas de ella

```
class base{
  public:
        int a;
  protected:
        int b;
  private:
        int c;
};
```

```
class derivada:base{
        ...
        ...
public:
        void suma(void);
  };
```

```
void main(void)
{
 base y;
derivada z;

 y.a=3;

 }
```

Desde la clase derivada pordemos acceder a a, y tambien a b, pero no a c, porque

es privado.Los publicos son accesibles desde cual1quier sitio, pero los protegidos solo desde las funciones de la clase derivadas

TIPOS DE ACCESO A LA CLASE BASE:

```
class base{
  ...
  ...
};                    public
class derivada:   private     base{};
                      protected
```

PUBLIC : los miembros publicos de la clase base son miembros publicos de la clase derivada los miembros protegidos de la clase base son protegidos de la clase derivada

PRIVATE : todos los miembros publicos y protegidos de la clase base son miembros privados de la clase derivada

PROTECTED : los miembros publicos y protegidos de la clase base son protegidos de la clase derivada

En los tres casos los miembros privados siguen siendolo, son inaccesibles.

Especificador de acceso a la clase	Accesible desde la propia clase	Accesible desde la clase derivada	Accesible desde el exterior
PUBLIC	SI	SI	SI
PROTECTED	SI	SI	NO
PRIVATE	SI	NO	NO

Independientemente de la forma de acceso a la clase, no se heredan:

- Constructores y destructores (cada clase tendra los propios)
- Funciones amigas
- Sobrecarga del operador =

Un objeto de la clase derivada se considera un objeto tambien de la clase base, pero no al reves.Cuando hay clases derivadas el orden de ejecucion de constructores: contructor base, constructores miembro, constructor de propio miembro

Dado clase B derivada de A : constructor de la clase A, constructores de los datos miembros de la clase B, constructor de la clase B

```
class C{ };

class A{
    A(void);
};

class B:A{
    C z;
  public:
    B(void)
};
```

El orden de llamada a los constructores es: constructor de A, constructor de C, constructor de B.

```
#include <iostream.h>
class A{
  public:
    A(){cout<<"Constructor A\n";}
};

class B{
  public:
    B(){cout<<"Constructor B\n";}
};

class C{
  public:
    C(){cout<<"Constructor C\n";}
};

class D:A{
```

```
  public:
    B c;
    C d;
    D(){cout<<"Constructor D\n";}
};

void main(void)
{
 D d;
}
```

En este otro caso el orden seria: constructor de A, constructor de B, constructor de C y constructor de D.

```
class padre{
   int valor;
 public:
   padre(){valor=0;}
   padre(int v){valor=v;}
};

class hija:public padre{
   int total;
 public:
   hija(int t){total=t;}
};

void main(void)
{
 hija a;
}
```

Da un error, para que no lo de hay que cambiar en class hija::public cambiar la linea del constructor con parametros por hija(int t=0):padre(t){total=t}

Esto es porque los constructores no se drivan, desde la clase derivada hija no tiene acceso al constructor de la clase base (padre). La forma de llamar a un constructor especifico es ponerselo en la llamada del constructor de la clase derivada. Hay que recordar que las llamadas a los constructores se hacen de la siguiente forma: primero al de la clase base, luego a los de las derivadas; mientras que las llamadas a los destructores ocurre juestamente lo contrario: primero el de las clases derivadas y luego el de la clase base.

LIGADURA DINAMICA Y ESTATICA

Es un concepto que C se refiere a la asociacion que hay entre la llamada a una funcion y el cuerpo de la funcion que finalmente se ejecuta.En tiempo de compilacion se sabe que cuando llama a una funcion, va a ejecutar ese cuerpo.

En C++ como puede haber mas de una funcion con el mismo nombre (sobrecarga), se produce lo que se llama la ligadura dinamica, lo que hace es que el compiolador elige en tiempo de ejecucion que funcion se va a ejecutar finalmente.El compilador sabe a que funcion debe llamar viendo a que puntero esta referenciada.

Ejemplo de ligadura en C normal:

```
switch(a)
{
 case 1: sumar(); break
 case 2: restar(); break
 case 3: salir();
}

void sumar(void);
void restar(void);
void salir(void);
```

-Dinamica: se dara en C++,implementandose a traves de las funciones virtuales no genera un codigo de un salto incondicional a una funcion concreta del programa, sino que deja esa decision para cuando se ejecute el programa. Sera en tiempo de ejecucion cuando sepa a que funcion debe llamar (debido a que puede haber la sobrecarga de funciones).Esto lo sabe mediante el puntero this, si apunta a un objeto de clase A, llamara a la funcion de la clase A

Hasta ahora la sobrecarga es un ejemplo de ligadura estatica (haciamos la llamada como a.sumar(), el compilador sabe a que funcion estamos llamando). Pero a partir de ahora vamos a usar punteros a objetos.Vamos a ver un ejemplo con clases derivadas y clases bases

```
class A
{
 void sumar(void);
}

class B: public A
{
```

```
 void sumar(void);
}

A *a;
B b;

a=&b; // Si se puede hacer, se puede asignar un objeto derivado a uno base. (1)
b=&a; // Incorrecto, no se puede asignar un objeto base a un objeto derivado
```

Cuando hacemos esto (1), suprimimos todo lo de la clase derivada y nos quedamos con lo que tenga la clase base.

```
A c;
A *a;
B b;

a=&b;
a->sumar(); //Llama a la funcion de la clase derivada
a=&c;
c->sumar(); // Llama a la funcion de la clase base;
```

FUNCIONES VIRTUALES

Veamos un ejemplo y despues pasemos a explicarlas:

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

class A{
  public:
   A(void){};
   virtual void sumar(void);
};

class B:public A{
   public:
   B(void):A(){};
   void sumar(void);
};


void A::sumar(void)
{
```

```
 cout<<endl<<"Es la suma de la clase A.";
}

void B::sumar(void)
{
 cout<<endl<<"Es la suma de la clase B.";
}

void main(void)
{
 A c;
 A *a;
 B b;
 a=&b;
 a->sumar(); //Llama a la funcion de la clase derivada
 a=&c;
 a->sumar(); // Llama a la funcion de la clase base;
}
```

El resultado de este programa seria:

```
Esta es la suma de la clase A
Esta es la suma de la clase A
```

Este programa lo que hace es declarar una clase base (A) y una derivada (B), y un puntero a un objeto de la clase derivada (*a). Como ya vimos en la parte de ligadura, a partir de ahora para llmar a las funciones, ya no hace utilizaremos el formato de ligadura estatica (nombre_objeto.funcion(), por ejemplo a.sumar()); sino que declararemos un puntero de la clase base, e iremos haciendo que dicho puntero apunte a las direcciones de las clases derivadasy llamando a las funciones asi: ptro->funcion() (por ejemplo a->sumar()).

Como ya vimosel resultado del programa anterior era la llamada a la funcion sumar de la clase A dos veces. El problema es que si ahora no sabe distinguir entre dos funciones sobrecargadas ¿para que sirve todo esto?. La solucion esta en declarar dichas funciones sobrecargadas como virtuales. Como en el programa anterior no estan definidas como virtuales, no comprueba los tipos, pero añadiendo virtual a la funcion sumar de la clase base, lo hace.

Resultado con virtual:

```
Esta es la suma de la clase B
Esta es la suma de la clase A
```

Con solo definir un ptro de tipo base y unos objetos de clase derivada, en funcion de quien apunte dicho ptro llamara a unas funciones u otras.

La sintaxis de las funciones virtuales es la siguiente:

-Se decide cuales son las funciones que van a tener un proposito parecido o el mismo en todas las clases.

-Se declara dicha funcion como virtual solamanete en la clase base, de tal forma que al definir una funcion commo virtual en la clase base, nos permite redefinir dicha funcion en las clases derivadas.

Una funcion que se declare virtual en la base, si esa funcion no es redefinida en alguna de las clases derivadas, da error de compilacion (solo si se trata de funciones virtuales puras). La solucion si no redefinimos, es declararla como virtual en la clase derivada, para asi evitar el error de compilacion.

El uso de las funciones virtuales tiene varias restricciones:

-El prototipo de la funcion virtual debe coincidir tanto en la clase base como en la redefinicion en la derivada (tanto en el numero y el tipo de prarametros) como en lo que devuelve

-Una funcion virtual debe ser miembro de la clase en la cual esta definida, pero puede ser amiga de otras clases.

Los constructores no pueden ser virtuales, ni necesitan serlo, pero los destructores pueden y deben ser virtuales. Hay que recordar que si una clase no sobreescribe una funcion virtual, el compilador usa la primera redefinicion de la clase siguiendo el orden inverso de la derivacion

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

class base{
   public:
     base(){cout<<endl<<"base constructora de A";}
     /*virtual*/ ~base(){cout<<endl<<"base destructora de A";}
     // Si no ponemos el virtual, no hace comprobacion de tipos.
};

class der:public base{
    public:
     der(){cout<<endl<<"constructor derivado de B";}
```

```
    ~der(){cout<<endl<<"base destructora de B";}
};

void main(void)
{
 base *p=new der;


 delete p; // El uso de delete con un objeto, llama al destructor del objeto.
}
```

CLASES ABSTRACTAS

Son las herramientas mas potentes para representar el mundo real, podremos representar cosas como casas, figuras geometricas.La forma de hacerla, no se diferencian en las normales solo en que debe tener, al menos una funcion virtual pura.

VIRTUAL PURA

Son funciones especiales, son similares a las virtuales.Supongamos que definimos una clase figura_3D; y dos clases derivadas: esfera y cubo. Los datos de la clase base son tres puntos (numero minimo de coordenadas para tener una figura en 3D seria un punto); y una funcion hallar_volumen. ¿Pero como podemos definir la funcion hallar_volumen, si para cada figura en 3D cambia la formula de hallarlo (p.e. cuadrado el lado al cubo (l^3) y para la esfera es 4/3 ¶ r^3)?.La solucion es declararlo como una funcion virtual pura, obligando a que en cada clase derivada tenga que ser redefinida (con la formula de hallar el volumen correspondiente). Las funciones virtuales puras lo que haria seria como guardar sitio, pero teniendo en cuenta que este sitio debe ser llenado en una clase derivada. Sintacticamente es:

```
virtual void nada(void)=0; // es diferente que poner {}
```

Las funciones virtuales puras al definirlas en la base, se definen nula (=0), no damos ninguna especificacion hasta que las redefinamos en las clases derivadas. La diferencia mas drastica entre las puras y las normales es que no se pueden llamar.

```
class A
{
 virtual void nada(void)=0;
 ...
}
void main(void)
```

```
{
 A a;
 a.nada(); // Da un error de compilacion, no se puede llamar.
}
```

Si en una derivada no queremos implementar en una clase derivada, la declaramos otra vez como derivada pura, pero debemos declararla en otras clases derivadas de esta ultima

CLASES ABSTRACTAS:

Estas clases no se deben utilizar para crear objetos, estan hechas para servir de

base, lo que haciamos hasta ahora (ser bases de ptros. A *a;)

Cuando tengamos que hacer un programa, debemos hacer:

-Clase abstracta de figura geometrica (define caracteristicas comunes a todas las figuras geometricas)

-Definir clases derivadas concretas (punto, linea, circulo)


PROGRAMACION EN WINDOWS

LA API DE WINDOWS

API (Aplication Programs Interface): es el equivalente a las interrupciones del MS-DOS, son un conjunto de funcion que permiten dibujar ventanas, dibujar un boton. A su vez la API llama a las interrupciones del DOS.

La API de windows son mas de 600 funciones en C normal. Para acceder a ella los compiladores utilizan librerias de objetos (el Borland C++ usa ObjectWindows), permitiendo acceder a la API de manera mas facil y reducida (son cien y pico clases).

Los programas hasta ahora en DOS, cuando un programa necesitaba hacer algo, llamaba a una interrupcion, servicio de BIOS o de DOS, nosotros llamabamos al sistema operativo para hacer operaciones. En windows esto cambia y es windows quien llama al programa (mensajes), cuando movemos al raton en windows, windows avisa al programa del cambio. El programa queda en memoria en espera de que windows le comunique mensajes.

Evento: se produce el hecho de mover el raton.
Mensaje: informacion que nos informa del evento.

LA CLASE TApplication

Para crear una aplicacion, lo primero que debemos hacer es crear un objeto aplicacion, siendo el constructor de la clase TApplication de la forma:

TApplication(const char far *Nombre=0);

siendo Nombre el titulo que aparece en la barra de la ventana. Por ejemplo:

TApplication prueba(“EJEMPLO”);

crea un objeto del tipo TApplication con el titulo EJEMPLO. Pero aparte de definir un objeto del tipo de aplicacion, hay que poner a funcionar el paso y recepcion de los mensajes (medio de comunicacion entre windows y nuestra aplicacion). La funcion que realiza esto es la funcion Run

FUNCION RUN

La funcion Run hace lo siguiente:

-Comprueba si ya se esta ejecutando una instancia del programa (si esta ejecutandose varias veces), para ello consulta la variable hRevInstance, que es un dato miembro de la clase TApplication(=0 ninguna instancia, n hay n copias del programa ejecutandose).

-Si no hay ninguna instancia, RUN llama a una funcion que se llama InitApplication() (funcion miembro de la clase aplication). Esta funcion es virtual pura y no hace nada, a menos que queramos redefinirla para hacer algo.

- Despues de InitApplication, llama a la funcion InitInstance(), que a su vez:

- llama a la funcion InitMainWindow(). Esta funcion es miembro de la clase y es la que construye un objeto ventana. La ventana que crea es normal, sin nada, no pone ni titulo, ni nada. Si queremos una ventana con menus, botones, etc, entonces tenemos que redefinir la funcion InitMainWindow.

- hace que la ventana se vea en pantalla.

Veamos esto con un ejemplo:

```
#include <owl\applicat.h>

// Como usamos objectvision, la main se llamara OwlMain (si programas con
// la API directamente se llamara WinMain

int OwlMain(int, char *[])
{
 TApplication a("HOLA"); // Clase incluida y definida crea un objeto de tipo aplicacion
                                    // bucle mensajes, funcion ventana, etc ...
 a.Run(); // Funcion miembro de la clase TApplication, empieza la ejecucion
          // (inicializa el proceso de paso de mensajes entre windows y la
            // aplicacion)
```

```
 return a.Status; //Dato miembro de la clase TApplication ( de una clase base
                  // TAplication. Contiene un dato de 16 bits que dice si se
                    // se ha creado bien la aplicacion}
}
```

El resultado de este programa seria mas o menos :

A continuacion, veremos los principales campos de la clase TApplication:

CAMPO hPrevInstance

En esta variable guardamos el numero de instancias de un programa que actualmente esten ejecutandose. Bien, pero ¿que es una instancia?, una instancia es una copia del programa. Es decir, al ejecutar por primera vez un programa, el campo hPrevInstance valdra 0, si es la segunda vez que ejecutemos el programa hPrevInstance valdra 1, etc.

Para moder tener varias instancias en memoria en windows es necesario que solo se utilice un solo segmento de datos (como ocurre en el modelo small de memoria). El modelo de memoria de windows es large (para evitar la limitacion del DOS de los 640k), este modelo usa solo un segmento de datos. En el momento que especificamos una variable de tipo FAR, le estamos diciendo es que creamos un nuevo segmento de datos, y entonces no podemos tener varias instancias en windows.

Igual con un ejemplo te enteras:

```
#include <owl\applicat.h>

int OwlMain(int, char *[])
{
 TApplication a;  // objeto del tipo TApplication

 if (!a.hPrevInstance)       // Si todavia no se ha ejecutado...
        a.Run();             // ... lo hacemos
 else  // Si actualmente esta ejecutandose...
```

```
        ::MessageBox(NULL,"Una sola instancia","Error",MB_OK);  //...mostramos una
                                                                // ventana con el mensaje de error
 return a.Status;
}
```

El resultado al ejecutar por primera vez este programa, es la figura 1, pero si en la segunda vez o posteriores sera la figura 2

Figura 1

Figura 2

FUNCION MessageBox

Para crear una caja de dialogos(MsgBox), seria:

```
::MessageBox(NULL, "mensaje", "titulo", botones);
```

Cuando queremos llamar a una funcion de la API anteponemos ::, sino llama a funciones de ObjectVision, no a la de la API.

CAMPO nCmdShow

Este campo nCmdShow (int) sirve para especificar el aspecto de la ventana (maximizada, minimizada, en forma de icono). Si ponemos:

```
nCmdShow=SW_SHOWMAXIMIZED; (la ventana que se cree estara mazimizada)
```

estamos diciendo que la ventana estara maximizada. Los posibles valores que puede tomar son:

SW_HIDE: esconder la ventana
SW_SHOW: volver a hacerla visible
SW_SHOWMAXIMIZED: ventana maximizada
SW_SHOWMINIMIZED: ventana minimizada

Ahora pasemos a ver las funciones miembro de la clase TApplication:

FUNCION InitMainWindow

La funcion miembro *virtual void InitMainWindow()* es protegida (solo se puede llamar desde un objeto de TAplication o una derivada) no se puede llamar, y crea la ventana marco (es el tipo de marco de la ventana). Define todas las caracteristicas interiores y exteriores de la ventana. Ademas de crear la ventana marco crea todos

los
controles asociados a la ventana marco. Donde se va a dibujar los datos y mostrar mensajes es la ventana cliente o ventana aplicacion. Hay que redefinir la funcion si queremos que por ejemplo no aparezca el boton de minimizar, etc

FUNCION CanClose

virtual BOOL CanClose() devuelve un tipo de datos booleano(TRUE o FALSE). Esta funcionse ejecuta automaticamente cuando cerramos la ventana (Alt+F4, etc). Si cuando salgamos queremos que muestre un mensaje de despedida, redefinimos esta funcion. Para que windows cierre la aplicacion CanClose debe devolver TRUE, si la funcion devolvemos FALSE, windows no cierra la aplicacion.

FUNCION IdleAction

BOOL IdleAction (long contador) se ejecuta automaticamente cuando la aplicacion no esta haciendo nada. Hay que redefinir esta funcion para hacer algo en los tiempos muertos. Contador indica el numero de veces que se ha ejecutado esta funcion desde el ultimo cambio (mover raton).

Veamos un ejemplo de esta funcion:

```
#include <owl\applicat.h>

class p: public TApplication{
  public:
                p():TApplication("Segundo Plano"){};
        protected:
                BOOL IdleAction(long);
};



BOOL p::IdleAction(long p)
{
 ::MessageBox(NULL,"Segundo Plano","Estoy en segundo Plano",MB_OK);
 return TRUE;
}

int OwlMain(int, char*[])
{
 p a;

  a.Run();
 return a.Status;
}
```

Este programa lo que hace es redefinir la funcion *IdleAction* para que cuando el programa no este ejecutando nada, muestre una ventana como esta:

esto puede ser util para ejecutar cosas en segundo plano o, como en el Word, aprovechar el momento en que no se esta haciendo nada para avisar de que se debe grabar el documento.

LA CLASE TWindow

Con esta clase definimos objetos del tipo ventana. Si creamos un objeto derivado de esta clase, podemos tambien redefinir una tabla de eventos propios. El constructor de la clase es el siguiente:

```
TWindow(TWindow *Madre, char far titulo=0, TModule *modulo=0);
```

Madre: es un puntero a la ventana madre, que sera NULL para la ventana marco de la aplicacion.

titulo: puntero al titulo de la ventana
modulo: informacion que debemos proporcionar si la ventana se crea a partir de una DLL

Uno de los principales campos de la clase TWindow es:

CAMPO Attr

Contiene las caracteristicas de la ventana. A su vez, Attr se divide en una serie de campos:

- Style: caracteristicas generales de la ventana, pudiendo ser uno de los siguientes valores:

WS_BORDER: ventana de borde delgado, impide modificar el tamaño de la ventana.
WS_CAPTION: indica la presencia de la barra de titulo.
WS_HSCROLL: barra de desplazamiento horizontal
WS_MAXIMIZEBOX: casilla de maximizar
WS_MINIMIZE: la ventana sera lo mas pequeña posible
WS_OVERLAPPED: la ventana podra cubrir otras y a su vez ser cubierta
WS_VSCROLL: barra de desplazamiento vertical
WS_THICKFRAME: borde grueso

Para poder activar una caracteristica seria de la forma:

```
Attr.Style=WS_OVERLAPPED|WS_CAPTION|WS_BORDER;
Attr.Style|=WS_VSCROLL
```

y para desactivar alguna de las caracteristicas:

```
Attr.Style &=~WS_MAXIMIZEBOX;
Attr.Style &=~(WS_MAXIMIZEDBOX|WS_MINIMIZEDBOX);
```

Ahora pasemos a ver las funciones miembro de la clase TWindow:

FUNCION Show

La funcion *void Show (int ShowCmd)* modifica la apariencia de una ventana.El argumento que le pasamos puede valer:

SW_HIDE: esconde la ventana
SW_SHOW: vuelve a hacer visible la ventana
SW_SHOWMAXIMIZED: amplia la ventana al maximo
SW_SHOWMINIMIZED: minimiza la ventana a un icono

FUNCION SetCaption

La funcion *void SetCaption(const char far *titulo)* modifica el titulo de la ventana, poniendo como titulo la cadena apuntada por titulo.

FUNCION SetBkgndColor

La funcion *void SetBkgndColor(DWORD color)* cambia el color de fondo de la ventana. Los colores disponibles son:

TColor::Black	negro	*TColor::LtBlue*	azul claro
TColor::LtGray	gris claro	*TColor::LtMagenta*	magenta claro

TColor::Gray	gris oscuro	*TColor::LtCyan*	cian claro
TColor::LtRed	rojo claro	*TColor::White*	blanco
TColor::LtGreen	verde claro		

LA CLASE TFrameWindow

Aqui definimos la ventana marco de la aplicacion, que posee el borde exterior y la ventana aplicacion. El constructor de la clase es:

*TFrameWindow(TWindow *madre, const char far titulo=0, TWindow *clientWnd=0)*

en donde

madre: es el puntero a la ventana madre
titulo: puntero al titulo de la ventana.
clientWnd: es el puntero a la ventana de la aplicacion

para crear la ventana marco y la ventana aplicacionse crean (en InitMainWindow):

MainWindow=new TFrameWindow(NULL, "Creacion de ventana", new TWindow)

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dc.h>

class b: public TApplication{
        public:
                void InitMainWindow();
                b():TApplication(){};
};

class TWinApp: public TWindow{
        public:
                TWinApp():TWindow(0,0,0)
                {
                 SetBkgndColor(TColor::LtBlue);
                }
};




void b::InitMainWindow()
{
 MainWindow=new TFrameWindow(NULL, "Ventana ejemplo de SetBkgndColor", new
TWinApp);
```

```
 /* Si queremos modificar la ventana añadimos: */
                        MainWindow->Attr.X=320; // Coordenada X=320
                        MainWindow->Attr.Y=240; // Coordenada Y=240
                        MainWindow->Attr.W=100; // Ancho 100 pixels
                        MainWindow->Attr.H=100; // Alto 100 pixels
                        MainWindow->Attr.Style=WS_OVERLAPPEDWINDOW|
                                                              WS_HSCROLL|
                                                              WS_VSCROLL;


}

int OwlMain(int, char*[])
{
 b prueba;

 prueba.Run();
 return prueba.Status;
}
```

Con lo que el resultado que daria, seria mas o menos:

TABLAS DE EVENTOS

Para tratar un evento es necesario escribir una funcion (o de paso redefinimos una que ya existe) y tenemos que crear una tabla de eventos a tratar que este asociada a la clase. La definicion de la tabla de eventos en la clase es de la forma:

DECLARE_RESPONSE_TABLE(nombre_clase)

en donde:

nombre_clase: es el nombre de la clase (dificil de suponer, eh?)

y definiendo la tabla de eventos de la forma:

```
DEFINE_TABLE_RESPONSEn(nombre_clase)
 ...
  Aqui van los eventos seprados por comas
END_RESPONSE_TABLE;
```

n: numero de clases bases de donde deriva

Veamos un ejemplo:

```
class prueba: public TWindow
{
 ...
 DECLARE_RESPONSE_TABLE(prueba);
 ...
};

 DEFINE_RESPONSE1(prueba)  // Como prueba solo deriva de la clase TWindow,
 ...                       // entonces ponemos 1
END_RESPONSE_TABLE;
```

FUNCION GetSystemMetrics

Esta funcion *GetSystemMetrics(nIndex)* devuelve distintas informaciones segun el valor de *nIndex*. Los principales valores de Nindex son:

SM_CXSCREEN: anchura de la pantalla
SM_CYSCREEN: altura de la pantalla

Veamos un ejemplo de todo esto:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dc.h>


class b: public TApplication{
        public:
                void InitMainWindow();
                b():TApplication(){};
};

class TWinApp: public TWindow{
        public:
                TWinApp():TWindow(0,0,0)
                {
```

```
                SetBkgndColor(TColor::LtBlue);
                }
};

class Marco: public TFrameWindow{
        public:
                void EvGetMinMaxInfo(MINMAXINFO far &);
                void EvSize(UINT  , TSize &);
                DECLARE_RESPONSE_TABLE(Marco);
                Marco(TWindow *p, char far *t, TWindow *cliente): TFrameWindow
                 (p,t,cliente){};
};

void Marco::EvSize(UINT marca, TSize &a)
{
 TFrameWindow::EvSize(marca,a); //Lamamos para que redibuje la ventana
 switch(marca)
 {
 case SIZE_MAXIMIZED: ::MessageBox(NULL,"Ventana Maximizada",”Aviso”,
                                        MB_OK);
                        break;


  case SIZE_MINIMIZED: ::MessageBox(NULL,"Ventana minimizada!","Aviso",
                                        MB_OK);
                        break;

  case SIZE_RESTORED: ::MessageBox(NULL,"Ventana con tamaño original",
                                         "Aviso", MB_OK);
                         break;
 }
}

void Marco::EvGetMinMaxInfo(MINMAXINFO far &a)
{
 a.ptMaxSize.x=300;
 a.ptMaxSize.y=200;
 a.ptMaxPosition.x=100;
 a.ptMaxPosition.y=100;
 a.ptMaxTrackSize.x=GetSystemMetrics(SM_CXSCREEN);
 a.ptMaxTrackSize.y=GetSystemMetrics(SM_CYSCREEN);
}

DEFINE_RESPONSE_TABLE1(Marco,TFrameWindow)
        EV_WM_GETMINMAXINFO,
        EV_WM_SIZE,
END_RESPONSE_TABLE;
```

```
void b::InitMainWindow()
{
 //TWindow *VentanaAplicacion;
 //VentanaAplicacion=new TWindow(0,0,0);
 MainWindow=new Marco(NULL, "Ventana marco a medida", new TWinApp);
 /* Si queremos modificar la ventana añadimos: */
                        MainWindow->Attr.X=GetSystemMetrics(SM_CXSCREEN)/4;
                // Coordenada X=320
                        MainWindow->Attr.Y=GetSystemMetrics(SM_CYSCREEN)/4;
                // Coordenada Y=240
                        MainWindow->Attr.W=200; // Ancho 100 pixels
                        MainWindow->Attr.H=100; // Alto 100 pixels
                        MainWindow->Attr.Style=WS_OVERLAPPEDWINDOW|
                                                                WS_HSCROLL|
                                                                WS_VSCROLL;

}

int OwlMain(int, char*[])
{
 b prueba;

 prueba.Run();
 return prueba.Status;
}
```

El resultado del programa es:

figura 1: al iniciar el programa
figura 2: al maximizar la ventana
figura 3: al minimizar la ventana

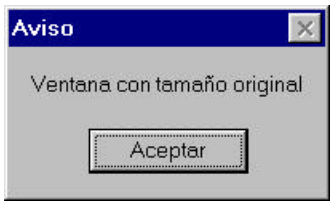


Figura 1

Figura 2

Figura 3

EVENTOS DEL RATON

Son los eventos que manda Windows a nuestro programa cuando hacemos click con

el raton, lo movemos, etc. Los valores posibles de dichos mensajes son :

WM_LBUTTONDOWN : boton izquierdo pulsado
WM_LBUTTONUP : boton izquierdo soltado
WM_MBUTTONDOWN : boton central pulsado
WM_MBUTTONUP : boton central soltado
WM_RBUTTONDOWN : boton derecho pulsado
WM_RBUTTONUP : boton derecho soltado
WM_MOUSEMOVE : cuando se mueve el raton

La forma de saber como se escribe el evento y la funcion de tratamiento, es facil sabiendo como se escribe el mesaje. El mensaje es igual que el evento, pero añadiendo EV_ al principio.La funcion de tratamiento es poner en minisculas todo el nombre, menos el inicio de las palabras y eliminando los subrayados.

Veamos unos ejemplos:

MENSAJE	EVENTO	FUNCION DE TRATAMIENTO
WM_LBUTTONUP	**EV_WM_LBUTTONUP**	**EvLButtonUp**
WM_MOUSEMOVE	**EV_WM_MOUSE**	**EvMouseMove**
WM_MBUTTONDOWN	**EV_WM_MBUTTONDOWN**	**EvMButtonDown**

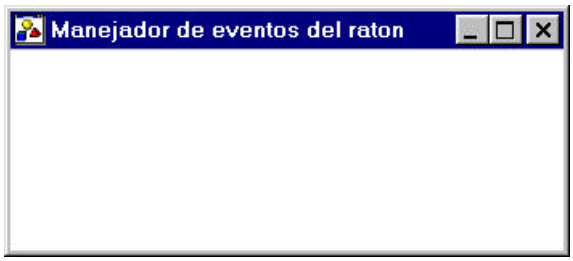
Hay que indicar que el boton derecho y el izquierdo se refieren a los de windows, y que es posible invertirlo, por lo que puede haber veces que el boton derecho del raton de windows sea el boton fisico izquierdo del raton.

Logicamente si queremos hacer un programa que responda a los eventos del raton, debemos definir la tabla de eventos.Como siempre, despues de la explicacion clara y concisa, vemos un ejemplo para enterarnos de algo:

```
#include<owl\applicat.h>
#include<owl\framewin.h>

class TWinApp: public TWindow
{
 void EvRButtonDown(UINT, TPoint &);
 void EvMButtonDown(UINT, TPoint &);
 void EvLButtonDown(UINT, TPoint &);
 public:
 TWinApp():TWindow(0,0,0){}
 DECLARE_RESPONSE_TABLE(TWinApp);
};
```

```
DEFINE_RESPONSE_TABLE1(TWinApp,TWindow)
  EV_WM_RBUTTONDOWN,
  EV_WM_LBUTTONDOWN,
  EV_WM_MBUTTONDOWN,
END_RESPONSE_TABLE;

class TApp: public TApplication
{
 void InitMainWindow()
 {
  MainWindow=new TFrameWindow(NULL,"Manejador de eventos del raton", new
  TWinApp);
 }
 public:
 TApp(): TApplication() {}
};

void TWinApp::EvRButtonDown(UINT a, TPoint &b)
{
 ::MessageBox(NULL,"Boton derecho pulsado","Aviso",MB_OK);
}

void TWinApp::EvMButtonDown(UINT a, TPoint &b)
{
 ::MessageBox(NULL,"Boton central pulsado","Aviso",MB_OK);
}

void TWinApp::EvLButtonDown(UINT a, TPoint &b)
{
 ::MessageBox(NULL,"Boton izquierda pulsado","Aviso",MB_OK);
}

int OwlMain(int, char * [])
{
 TApp b;

 b.Run();
 return b.Status;
}
```

El resultado del programa es el siguiente: cuando pulsamos la tecla izquierda, muestra la figura 1, si es la tecla central muestra la figura 2 y por ultimo si es la derecha muestra la figura 3.



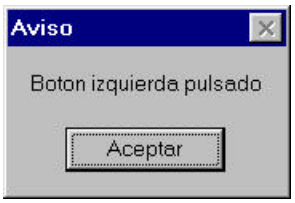


Figura 1

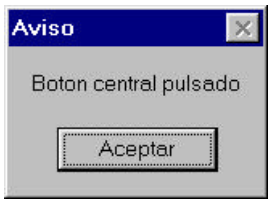


Figura 2

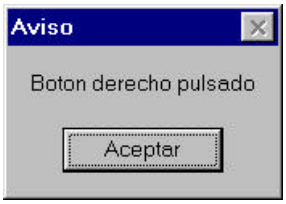


Figura 3

FUNCION SetCursorPos

Esta funcion SetCursorPos(x, y) pone el puntero del raton en el punto de coordenadas (x, y) relativas a la pantalla. Si queremos pasar de las coordenadas de pantalla a las coordenadas de la ventana, usamos dos funciones: ScreenToClient y ClientToScreen

FUNCION ScreenToClient

La funcion *ScreenToClient(TPoint &)* pasa las coordenadas de pantalla a coordenadas de la ventana. Tenemos que pasar a esta funcion un objeto de tipo TPoint, que tiene dos campos x e y. Al pasar ese objeto por referencia, modificara las coordenadas del objeto

FUNCION ClientToScreen

Esta funcion cuyo prototipo es *ClientToScreen(TPoint &)* pasa las coordenadas de la ventana a coordenadas de la pantalla.Ocurre exactamente que con la funcion anterior, guarda las nuevas coordenadas en el objeto del tipo TPoint.

Un ejemplo de estas dos funciones:

```
TPoint p;  // Declaramos p como un objeto TPoint ( de dos campos x e y )

p.x=10;  p.y=20;  // Coordenadas de la ventana ...
ClientToScreen(p); // ... las convertimos a coordenadas de la pantalla y ...
SetCursorPos(p.x, p.y); // ... ponemos el cursor del raton en las nuevas
                        // coordenadas
```