Sistemas Informáticos Industriales

Apuntes de

Introducción a C++.
Programación orientada a objetos.

Grado en Electrónica y Automática
Departamento de Informática de Sistemas y Computadores
Escuela Técnica Superior de Ingeniería del Diseño

introduccion_POO_en_Cpp_r10.odt

Contenido

2 INTRODUCCIÓN A C++. PROGRAMACIÓN ORIENTADA A OBJETOS

2.1 INTRODUCCIÓN

En este tema se pretenden mostrar los conceptos básicos relacionados con la programación orientada a objetos en C++.

El objetivo es que estos conocimientos nos permitirán aprovechar la gran cantidad de librerías y "clases" puestas a nuestra disposición por terceras partes.

En particular, se pretenden aprovechar las tecnologías basadas en *componentes visuales,* que permiten incorporar infinidad de capacidades y de una manera sencilla a nuestras herramientas de desarrollo de software.

En el curso pasado, la utilización de componentes prediseñados para Embarcadero/Borland C++ Builder han permitido desarrollar programas con funcionalidad y aspecto profesional sin necesidad de conocer internamente o diseñar los componentes utilizados.

Estos componentes son *objetos* en la jerga C++. Es importante introducir las bases de la programación orientada a objetos para comprender mejor la filosofía de desarrollo de aplicaciones C++ y poder llegar a diseñar aplicaciones más complejas.

Éste apartado no pretende enseñar a programar con objetos, sino dar unas pinceladas sobre el tema que permitan comprender lo que son y poder aprovecharlos.

Esta unidad está pensada para trabajarla de forma lineal sin necesidad de acudir a otras fuentes ni materiales adicionales.

En puntos concretos se intercalan actividades que permitirán practicar los conocimientos adquiridos.

En caso de dificultad en la resolución de actividades sí se recomienda acceder a fuentes externas, por ejemplo, a la bibliografía recomendada.

2.2 OBJETIVOS

Comprender la filosofía de una programación orientada a objetos y sus ventajas e inconvenientes.

Aprender a definir e implementar clases C++.

Aprender a crear y utilizar objetos a partir de la clase.

Aprender a derivar clases.

2.3 ESQUEMA

El esquema de la unidad coincide con el desarrollo ordenado de los objetivos y que se plasman en la correspondiente tabla de contenidos.

2.4 CONCEPTOS BÁSICOS

Estructurar un programa en funciones repartidas en módulos puede no ser suficiente cuando éste empieza a ser muy complejo. Aunque las funciones se repartan correctamente en módulos, es difícil recordar que hace cada función y cómo se relaciona con las otras.

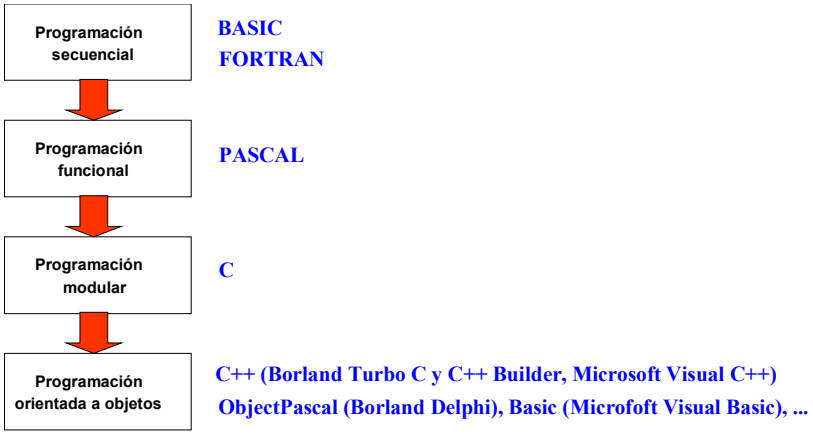


Figura 2-1. Evolución de las técnicas de programación

La programación orientada a objetos pretende resolver éste y otros problemas, simplificando la forma de escribir y mantener el código de las aplicaciones, y facilitando su reutilización.

Burdamente, un objeto en C++ es una colección de ***miembros*** (equivalente a variables) y ***métodos*** (equivalente a funciones), todos ellos ***encapsulados*** (privados y ocultos) de manera que no son accesibles desde el exterior a menos que, en su definición, se permita dicho acceso.

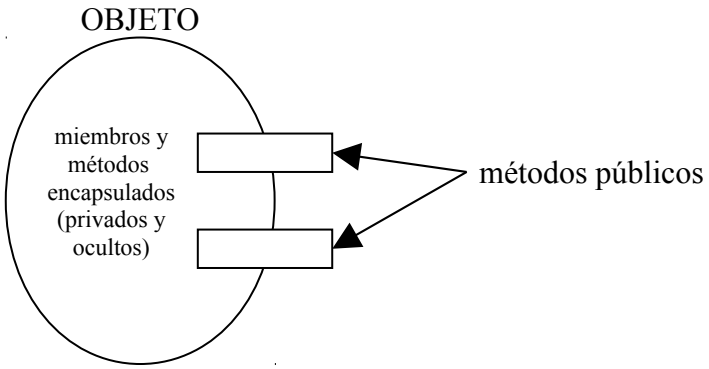


Figura 2-2. Representación gráfica de un objeto

Para tener un objeto es necesario definir primero el “tipo de objeto” o ***clase***, y después ***instanciar*** (crear) tantos objetos de una clase como se quiera.

Los miembros contenidos en cada instancia de la clase serán independientes unos de otros (datos en variables locales) .

Los tipos de datos que ya se conocen de C, en C++ son clases. Por ejemplo, el tipo int será una clase, y la creacíon de variables se corresponderá con la instanciación de objetos de esa clase.

tipo "int" es la clase

int a,b; es crear dos objetos “a” y “b”

Otro símil que se suele emplear es el de la factoría. LA clase es la factoría, y los objetos son lo que la factoría crea. En la Figura 2-3 se muestra gráficamente la idea de clase y objeto.

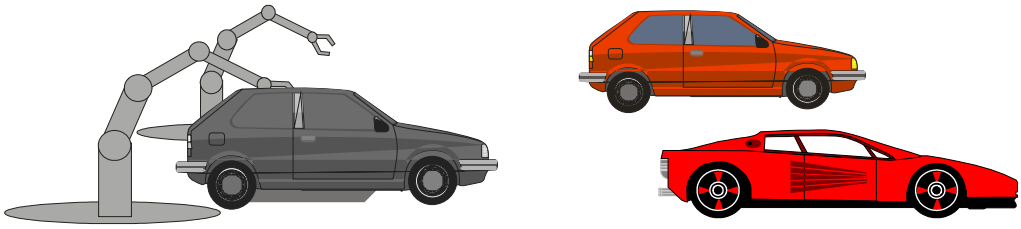


Figura 2-3. La factoría de la clase "coche" (izquierda) y dos objetos instancia de la clase (derecha)

2.5 CREANDO UNA CLASE

Para crear una clase es necesario primero dar la ***definición*** de la clase.

En dicha definición se declaran los miembros (variables) y los métodos (prototipos de funciones) que formarán parte del objeto.

Tiene la siguiente forma general:

es una clase — nombre de la clase

```
class TObjeto
{
public:
    TObjeto(parametros); //constructor
    ~TObjeto();          //destructor
    int var_ejemplo;
    double funcion_ejemplo (int valor);

private:
    …

protected:
    …
```

miembros y métodos accesibles desde el exterior

miembros y métodos privados (sólo accesibles por el propio objeto)

miembros y métodos accesibles por clases derivadas

```
}
```

Hay que destacar dos métodos opcionales, el ***constructor*** y el ***destructor***.

Un ***constructor*** es un método al que se llama cuando se crea una instancia de la clase (objeto). Su función principal suele ser inicializar los datos del objeto. En caso de que no sea necesario inicializar nada no hace falta proveer un constructor.

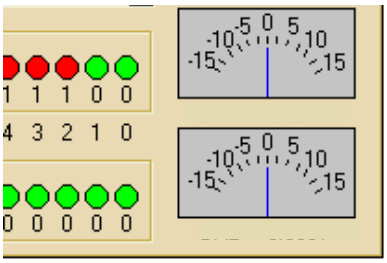
Un ***destructor*** es un método al que se llama cuando se destruye al objeto. Su propósito principal suele ser liberar los recursos requeridos por el objeto, por ejemplo, memoria dinámica solicitada, ficheros abiertos, etc.

Esta definición suele formar parte de un fichero de cabecera.

Ejemplo:

El siguiente fragmento de una imagen de una aplicación para Windows muestra dos símiles de medidores analógicos de 120º. El aspecto de los dos medidores es muy similar, simplemente cambia la posición dentro de la ventana de la aplicación y el valor que representan.

En este caso, el escribir código independiente para cada medidor es tener que hacer 2 veces el mismo trabajo, así que se ha optado por crear una clase y definir 2 objetos de dicha clase con las características necesarias. Imagínese el trabajo de programación que se puede ahorrar si se pretendiese tener, por ejemplo, 20 medidores.



Se propone crear una clase con el nombre clase `Tmedidor120` y cuyos métodos públicos serán:

- Constructor de la clase:

```
Tmedidor120(
        TCanvas *cnv,  // puntero al Canvas del objeto sobre el que dibuja
        int ctrx,      //  posición del centro del medidor en pixels
        int ctry);     //  posición del punto de partida de la aguja
```

- Método para actualizar la medida representada

```
void valor(double v);  // v es el valor a representar
```

- Método para redibujar el medidor

```
void redibuja(void);
```

Un posible código para la definición del objeto es:

```
//-------------------------------m120b.h-------------------------------------
#include <vcl.h>

#ifndef m120bH
#define m120bH
//---------------------------------------------------------------------------

class Tmedidor120
{
public:

        Tmedidor120(TCanvas *cnv, int ctrx, int ctry);
        void valor(double v);
        void redibuja(void);

private:

        TCanvas *canvas;
        int centrox, centroy;
        double medida;

        void modarg_xy(double modulo, double argumento, int *x, int *y);
        void dibujanumero(AnsiString texto, int radio, double grados);
};

#endif
```

Dicha definición se suele colocar dentro de un fichero de cabecera al que se le añaden macros de preprocesador (empiezan por #).

Actividad

Crea la clase `TGlass` (tipo vaso) con las siguientes características.

- En el constructor se debe especificar el tamaño del vaso (capacidad) en centímetros cúbicos.
- Definir el método `PourLiquid` (echar líquido) de manera que permita especificar una cantidad de líquido vertida en el vaso.
- Definir el método `Drink` (beber) de manera que permita consumir una parte del líquido especificada como parámetro.
- Definir el método `GetRemaining` (ver cuanto queda) de manera que devuelva la cantidad de líquido en el vaso.
- Dos miembros privados que representen la capacidad del vaso y la cantidad de líquido actual en el vaso.

Ahora podemos pasar a la implementación del objeto. Esta suele estar en un módulo que incluirá la cabecera previamente definida.

La implementación contendrá los cuerpos de los métodos de la definición según la siguiente sintaxis:

```
TObjeto::TObjeto (parametros)
```

```
{
      ...
}

TObjeto::~TObjeto()
{
      ...
}

double TObjeto::funcion_ejemplo (int valor)
{
      ...
}
```

cuerpo del método constructor (si lo hay)

cuerpo del método destructor (si lo hay)

cuerpo de un método ejemplo

es decir, al nombre de los métodos se les antepone el nombre de la clase seguido de `::` al que se le denomina operador de resolución de ámbito.

Dentro de los métodos se escribirá el código deseado como si de funciones C normales se tratase, y se podrá acceder a los miembros del objeto como si de variables globales se tratase.

Ejemplo:

Esta es la implementación del medidor de 120º

```
//-------------------------------m120b.cpp-----------------------------------
#include "m120b.h"
#include <math.h>

#define radio_aguja 48

#define color_fondo     clSilver
#define color_aguja     clBlue
#define color_texto     clBlack
#define color_borde     clBlack
#define color_division  clBlack

// rango de medida
#define r_min  -15.0
#define r_max   15.0

// --- constructor de la clase
Tmedidor120::Tmedidor120(TCanvas *cnv, int ctrx, int ctry)
{
        // hacer una copia de los parámetros
        canvas = cnv;
        centrox = ctrx;
        centroy = ctry;

        // suponemos que uan medida inicial
        medida=0.0;

        // pintamos el control
        redibuja();
}


// --- metodo para pintar el medidor
void Tmedidor120::redibuja(void) {
```

```
        AnsiString s;
        int h, l;
        int x,y;
        int pos;

        //dibujar el rectángulo del medidor
        canvas->Brush->Style = bsSolid;
        canvas->Brush->Color = color_fondo;
        canvas->Pen->Color = color_borde;
        canvas->Rectangle(centrox-radio_aguja,centroy-
radio_aguja,centrox+radio_aguja,centroy);

        canvas->Font->Size=6;
        canvas->Font->Color=color_texto;
        canvas->Font->Name="MS Sans Serif";


        // dibujar los números
        pos = radio_aguja - 6;
        dibujanumero("0",pos,90.0);
        dibujanumero("5",pos,70.0);
        dibujanumero("10",pos,50.0);
        dibujanumero("15",pos,30.0);
        dibujanumero("-5",pos,110.0);
        dibujanumero("-10",pos,130.0);
        dibujanumero("-15",pos,150.0);

        // dibujar las divisiones
        pos = radio_aguja - 14;
        double gra = 30.0;
        canvas->Pen->Color = color_division;
        for (int i=0;i<13;i++) {
                modarg_xy(pos,gra,&x,&y);
                canvas->MoveTo(centrox+x,centroy-y);
                modarg_xy(pos-4,gra,&x,&y);
                canvas->LineTo(centrox+x,centroy-y);
                gra += 10.0;
        }

        // dibujar la aguja
        valor(medida);
}

// --- método para actualizar el medidor
void Tmedidor120::valor(double v)
{
        int x, y;
        double posicion;
        double ra;

        ra = radio_aguja - 19;
        //borrar aguja antigua
        posicion = 120+30-(120/(r_max-r_min))*(medida-r_min);
        modarg_xy(ra, posicion, &x, &y);
        canvas->Pen->Color = color_fondo;
        canvas->MoveTo(centrox,centroy-2);
        canvas->LineTo(centrox+x,centroy-y);
        medida = v;
        //dibujar la nueva aguja
        posicion = 120+30-(120/(r_max-r_min))*(medida-r_min);
        modarg_xy(ra, posicion, &x, &y);
        canvas->Pen->Color = color_aguja;
        canvas->MoveTo(centrox,centroy-2);
```

```
        canvas->LineTo(centrox+x,centroy-y);
}

// a partir de un modulo y un argumento en grados devuelve la posición x y
correspondiente
void Tmedidor120::modarg_xy(double modulo, double argumento, int *x, int *y)
{
        const double grarad = 3.141592/180;
        *x=cos(grarad*argumento)*modulo;
        *y=sin(grarad*argumento)*modulo;
}

// dibuja texto en la posición indicada a partir de un rádio y unos grados
void Tmedidor120::dibujanumero(char *texto, int radio, double grados)
{
        int x,y,h,l;

        modarg_xy(radio, grados, &x,&y);
        h= canvas->TextHeight(texto);
        l= canvas->TextWidth(texto);
        canvas->TextOut(centrox+(x-l/2),centroy-(y+(h/2)),texto);
}
```

Actividad

Implementa la clase `TGlass`.

Supón el vaso vacío inicialmente.

2.6 CREANDO OBJETOS

Crear objetos es, básicamente, lo mismo que crear variables. Como cualquier tipo de dato C, estos se pueden definir y crear al escribir el programa (tiempo de compilación) o dinámicamente cuando sea necesario (tiempo de ejecución).

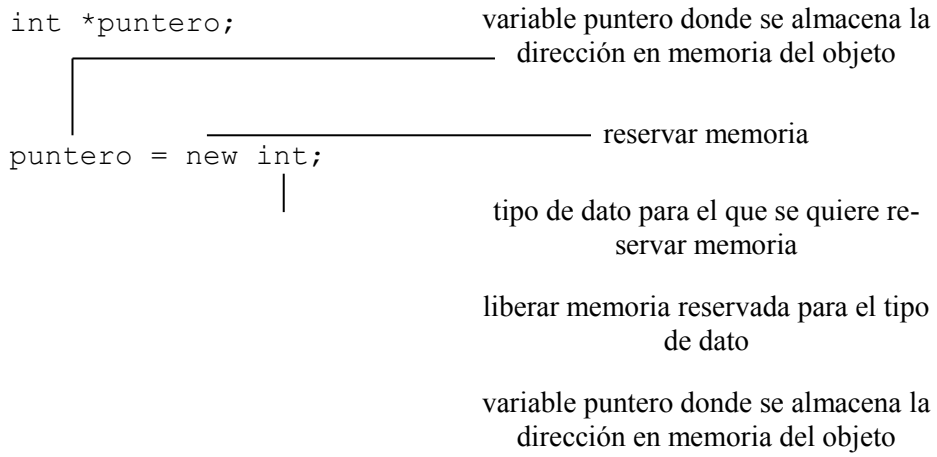
Por ejemplo, la siguiente sentencia crea dos objetos de la clase `TObjeto`.

```
TObjeto uno(parametros), dos(parametros);
```

La principal diferencia con crear una variable es que, al crear los objetos, se llama al constructor de la clase y es necesario pasarle los parámetros requeridos.

Es muy habitual que los objetos se creen dinámicamente en tiempo de ejecución según las necesidades de la aplicación. Para ello se utilizarán los operadores C++ `new` y `delete`, que son equivalentes a las funciones `malloc()` y `free()` de C.

Su sintaxis es:



```
delete puntero;
```

En el caso de que se emplee `new` para crear objetos, se llama automáticamente al constructor de la clase.

Lo mismo se aplica a `delete`, al borrar el objeto, se llama automáticamente al destructor.

Por ejemplo:

```
TObjeto *puno, *pdos;
puno = new TObjeto(parámetros);
pdos = new TObjeto(parámetros);
…
delete puno;
delete pdos;
```

Con esto se consigue el escenario mostrado en la figura 2-4.

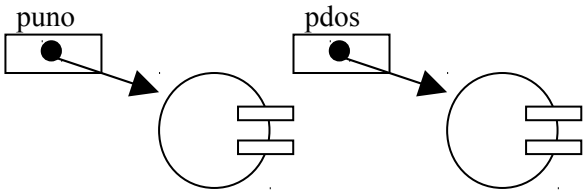


Figura 2-4. Dos objetos apuntados con punteros.

Ejemplo:

Para crear los dos objetos del medidor de 120º que se muestran en una imagen previa se ha hecho:

```
#include "m120b.h"
Tmedidor120 *m1,*m2;

// codigo asociado a la creación del formulario principal
void __fastcall Torm1::FormCreate(TObject *Sender)
{
    m1 = new Tmedidor120(TForm->Canvas,342,249,48,-15,15);
    m2 = new Tmedidor120(TForm->Canvas,342,312,48,-15,15);
}
```

Destacar que los punteros están fuera de la función para que los objetos creados puedan ser accedidos desde otras funciones.

Actividad

Haz un programa principal que cree 2 vasos estáticos y uno dinámico de la capacidad que quieras.

2.7 UTILIZANDO LOS OBJETOS

Los objetos se utilizan llamando a los métodos públicos.

En función de si el objeto se instancia creándolo en tiempo de compilación o, dinámicamente, en tiempo de ejecución, la sintaxis para acceder a los métodos cambia. Se usará el operador `.` en el primer caso y el operador `->` en el segundo, igual que se hace para acceder a los miembros de una estructura.

Por ejemplo,

```
double a;
TObjeto uno(parámetros), *pdos;
pdos = new TObjeto(parámetros);
…
a = uno.funcion_ejemplo(3);
…
a = pdos->funcion_ejemplo(3);
```

Ejemplo:

La siguiente línea llama al método público `valor()` del medidor de 120º que realiza una actualización del valor mostrado por el medidor

```
m1->valor(3.5);
```

Actividad

Amplía la actividad anterior de manera que se eche algo de líquido a los vasos, se beba y se haga un recuento de la cantidad total de líquido que queda (la suma del líquido en cada vaso).

2.8 CLASES DERIVADAS. HERENCIA

Una característica importante de la programación orientada a objetos es la posibilidad de derivar una clase a partir de otra. Esta característica permite crear clases complejas a partir de otras más sencillas, con lo que se facilita la reutilización de código.

Se denomina ***derivación*** a crear una clase a partir de otra. Al tipo de la clase de partida se le llama ***base*** o ***ascendiente***, mientras que al nuevo tipo se le llama ***derivado*** o ***descendiente***.

La sintaxis para crear una clase derivada es:

```
class TDerivada: [acceso] TBase
{
                                        } miembros y métodos repartidas
                                          en las secciones ya explicadas
}
```

La nueva clase ***heredará*** los miembros y métodos de la previa y se le añadirán los nuevos que se definan. La nueva clase podrá acceder a los miembros y métodos públicos y protegidos de la previa, pero no a los privados.

Por defecto, los miembros y métodos heredados pasarán a la sección privada de la nueva clase. Esto se puede cambiar añadiendo atributos de acceso a las clases de las que se deriva. Por ejemplo,

```
class TDerivada: public TBase
```

hará que los métodos públicos de TBase sean accesibles al usuario de la clase derivada, manteniéndose privados el resto de miembros.

Desde la implementación del constructor de la clase descendiente se podrá llamar al constructor de la clase ascendiente según la sintaxis,

```
TDerivada::TDerivada(parámetros):TBase(parámetros)
```

Con ello se logra que la base construya parte del objeto y la descendiente la otra parte.

Si en una clase descendiente se define de nuevo un método que ya existía en la clase base, los objetos creados con la clase descendiente utilizarán dicho método en lugar del heredado. Este mecanismo permite redefinir el comportamiento de las clases derivadas y, así, adaptarlas a las nuevas necesidades.

Cuando se redefine un método, el método original de la clase descendiente puede ser accedido desde la implementación utilizando la sintaxis,

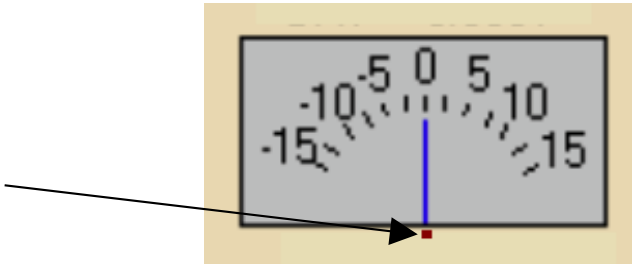
```
resolución de ámbito, "padre"
    |
TBase::funcion_ejemplo(3.5);
                  |
            método accedido
```

La herencia en programación orientada a objetos es una propiedad transitiva, lo que significa que se pueden derivar nuevas clases a partir de previas y los métodos de la clase base segui-

rán disponibles. Ello permite conseguir jerarquías de clases complejas a partir de clases relativamente sencillas.

Ejemplo:

Se quiere crear un medidor de 120º igual que el anterior pero con un pequeño punto rojo debajo de la aguja. Lo más práctico será utilizar la clase anterior y derivarla de manera que se le añada esa característica.



Téngase en cuenta que, en muchos casos, las clases padre no están disponibles en código fuente o es muy complejo, por lo que este método es el más aconsejable.

A continuación se propone un nuevo fichero de cabecera para la clase derivada, observar que se incluye la cabecera de la clase previa y se especifican los métodos a redefinir.

```
//------------------------ mi_m120.h ----------------------------------------

#ifndef mi_m120H
#define mi_m120H

#include "m120b.h"  //definición de la clase base

class TMiMedidor : Tmedidor120
{
public:
   TMiMedidor(TCanvas *cnv, int ctrx, int ctry);
   // metodos de la clase base a redefinir
   void redibuja(void);
   void valor(double v);
};

private:
   TCanvas *mi_cnv;
   int mi_ctrx, mi_ctry;
   double mi_valor;
   void pon_punto(void);

#endif
```

Y esta sería la implementación, que se pude almacenar en un archivo aparte como se hace aquí e incluir después en el proyecto que pretenda aprovecharlo.

```
//------------------------ mi_m120.cpp ---------------------------------------

#include "mi_m120.h"

// -- Constructor del nuevo medidor, llama también al constructor antiguo
TMiMedidor::TMiMedidor(TCanvas *cnv, int ctrx, int ctry)
```

```
:Tmedidor120(cnv, ctrx, ctry)
{
   // nos guardamos los parámetros especificados para la creación del objeto
   // pues no podemos consultarla del padre al tenerla privada
   mi_ctrx = ctrx;
   mi_ctry = ctry;
   mi_cnv = cnv;
   mi_valor = 0.0;
}

// -- Método que nos permitirá mostrar un punto rojo al comienzo de la aguja
indicadora
void TMiMedidor::pon_punto(void)
{
   mi_cnv->Pixel[mi_ctrx][mi_ctry] = clRed; //poner un puntito

}

// -- Método redefinido,
// así el objeto se usa igual que el padre y hace más cosas
void TMiMedidor::redibuja(void)
{
   Tmedidor120::redibuja();             // pintar el medidor
   pon_punto();                         //pintar punto
}

// -- Otro método redefinido, pensando en la ampliación para prácticas
void TMiMedidor::valor(double v)
{
   Tmedidor120::valor(v);  //pintar aguja
}
```

Para poder utilizar la nueva clase, nuestro proyecto de aplicación deberá incluir la cabecera de esta nueva clase y los módulos de la implementación (en forma de módulos fuente, módulos compilados .obj, o metidos dentro de una librería .lib).

Actividad

Define una nueva clase `TVasoCilindro` que permita especificar el tamaño del vaso como un diámetro de la base y una altura y emplee el vaso ya creado antes.

El constructor de la clase deberá tener el prototipo.

Tvasocilindro::Tvasocilindro(double altura, double radio):Tvaso(3.141592*rad*rad*alt)

Los métodos de la clase ascendiente deberán mantenerse públicos.

Añade el siguiente método público para suponer que se echan cubitos al vaso, reduciendo su capacidad. Supón que un cubito tiene 3 c.c.

void Tvasocilindro::EcharHielo(int num_cubitos).

Redefine el método Beber() de manera que, cada vez que se beba, se desperdicie una cantida adicional de 0,3 cc.

Actividad

Implementa la clase `TVasoCilindro`.

En la implementación se deberá usar:

```
Tvasocilindro::Tvasocilindro(double altura, double radio):Tvaso(3.141592*rad*rad*alt)
```

NOTA: Este nuevo método influirá en la variable "capacidad" de TVaso que se definió como "privada". Pása dicha variable a la sección "protected" para que pueda acceder a ella la nueva clase.

2.9 EL OPERADOR *THIS*

El operador `this` es equivalente a un puntero a los miembros (variables) de un objeto llamado mediante uno de sus métodos.

Cuando se crean instancias de objetos dinámicos, es imposible saber dónde se han creado sus miembros, pues no se conoce el nombre del objeto y solo se tiene un puntero a él. Se podría decir que `this` es equivalente a decir "yo mismo, aunque no sé como me llamo". Con ese this, será fácil acceder a mis propios miembros y métodos aunque no sepa mi nombre.

2.10 RESUMEN

En este punto se debe saber qué es una clase C++ y saber crear clases sencillas. También se debe ser capaz de crear instancias (objetos) de esas clase y saber emplearlos.

Crear clase e instanciar objetos se debe convertir en este punto en algo mecánico, así que no tiene ningún mérito. El mérito estará en comprender la potencia que va a aportar al desarrollador si sabe cuál es su utilidad.

La clase ejemplo desarrollada durante esta unidad se ha seleccionado precisamente para intentar que esa filosofía se vea en un “objeto” cotidiano como es un simple vaso para beber. De ahí a construir una “casa” no hay mucho.

Aquí se puede contar que C++ tiene ventajas e inconvenientes. En particular, la palicación de C++ a sistemas basados en microcontrolador no es eficiente debido a la sobrecarga de código máquina que implica este paradigma, así que, si se pretende desarrollar para microcontroladores, no es buena idea abandonar la filosofía C.

Una vez destapada la caja C++, es más fácil aprovechar una inmensidad de clases y librerías disponibles en Internet que permitirán hacer aplicaciones realmente impresionantes.

2.11 BIBLIOGRAFÍA COMENTADA

- El lenguaje de programación C++. Bjarne Stroustrup. Addison Wesley, 2001. ISBN: 84-7829-046-X

El manual de referencia del principal promotor de C++. No es nada recomendable para empezar, pero es una referencia obligada.

- Borland C++ Builder 2006”. Francisco Charte Ojeda. Anaya Multimedia. ISBN: 84-415-1988-9 .

El libro básico para la asignatura anterior tiene un apartado sobre C++ que está bastante bien y es corto. Puede ser un buen punto de partida.