

Introducción a la Programación Orientada a Objetos - Herramientas de C++

Informática II
R2004 - 2017

Introducción - ¿Cómo venimos programando?

Características

- **Separación** de las funciones en archivos distintos en función de la capa a la que pertenecen.
- **Separación** de las funciones en archivos distintos en relación con su funcionalidad (teclado, display, ADC, UART, etc.)
- Manejo de la **visibilidad** de los datos (variables locales static en lugar de globales, variables globales static).
- Generación de una **interfaz** para comunicar una capa con otra.

Aplicación

Primitivas

Drivers

Ventajas

- Puedo modificar un bloque del código sin modificar el programa entero (siempre que mantenga la **interfaz**)
- Es más fácil de reutilizar y adaptar.
- No tengo que conocer el funcionamiento de las capas del programa que no estoy realizando.
- Una vez **encapsuladas** las funciones y variables puedo utilizarlas para diferentes proyectos.

“Encapsulando” en C - caso Display

.H

```
#include "RegslPC17xx.h"
#include "KitInfo2.h"

#define CANT_DIGITOS 6

void BarridoDisplays (void);

extern volatile uint32_t valordisplay[CANT_DIGITOS];
extern volatile uint32_t flag1seg;

uint32_t tabla7seg[] = { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x67};

void BarridoDisplays ( void )
{
    static uint32_t nro_digito = 0;
    static uint32_t val_digito;
```

Datos y funciones que necesita mi bloque de código para funcionar

Interfaz

(funciones y datos “públicos” que pueden ver otros bloques de código si incluyen mi .h)

Datos “privados” que solo puede ver mi bloque de código (no me interesa que lo vean otros)

.C

```
//Tomo el valor que tengo que mostrar
val_digito = valordisplay [nro_digito];

//Lo pongo en el bus de datos:
SetPIN( BCD_A , val_digito & 0x01 );
SetPIN( BCD_B , ( val_digito >> 1 ) & 0x01 );
SetPIN( BCD_C , ( val_digito >> 2 ) & 0x01 );
SetPIN( BCD_D , ( val_digito >> 3 ) & 0x01 );

SetPIN( SEGMENTO_DP , 0 );

//Mando un pulso de clock (para correr el digito:
SetPIN ( DGT_CLK , 1 );
SetPIN ( DGT_CLK , 0 );

//Incremento el digito a mostrar (para la proxima):
nro_digito++;
```

Además de las funciones de utilización del Display (todas las que considere que el usuario podría necesitar), genero la función de INICIALIZACIÓN, que me asegura que todas las variables comiencen en el valor deseado

C++ como “evolución” del C - Clases

El C++ aparece como un lenguaje en donde una de las características estructurales es la encapsulación de datos.

Para lograr esto posee algunas características distintas al C:

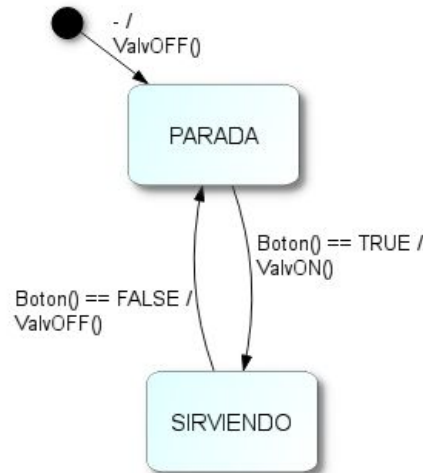
```
class Display{  
public:  
    int bufferDisplay [CANT_DIGITOS];  
    void BarridoDisplay(void);  
  
    void Display(){  
        InitDisplay();  
    }  
  
private:  
  
    int tabla7Seg [] = { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x67};  
    const int CANT_DIGITOS = 6;  
  
    void InitDisplay();  
  
};
```

Una clase contiene **PROPIEDADES** (variables) y **MÉTODOS** (funciones), que describen un subsistema dentro de nuestro programa. La **INSTANCIA** de una clase es un **OBJETO**.

Al momento de **INSTANCIARSE** el **OBJETO** se llama a un método **CONSTRUCTOR**, que típicamente es el método que inicializa mi objeto, dándole a sus **PROPIEDADES** los valores adecuados para su funcionamiento.

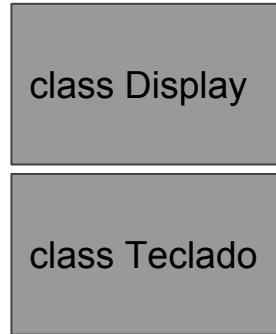
Paradigmas de programación: POE vs. P00

En la programación orientada a EVENTOS focalizamos la atención en el análisis de los eventos que pueden sucederse en el tiempo, y las transiciones y acciones que tiene nuestro programa a medida que se suceden dichos eventos.



Paradigmas de programación: POE vs. P00

En la programación orientada a OBJETOS vamos a focalizar la atención en crear las clases que sean necesarias para el funcionamiento de mi programa, cada una de ellas con las interfaces adecuadas, de manera de que el programa se reduzca a instanciar los objetos necesarios y comunicarlos de acuerdo a la lógica de mi sistema.



```
void main () {  
  
    Display D1;  
    Teclado T1;  
  
    D1.Show( T1.getTecla() );  
  
}
```

Diferencias del lenguaje C++

- Los archivos fuente de C++ tienen la extensión ***.cpp** (de C plus plus) en lugar de ***.c** que conocemos y usamos en lenguaje C
- El compilador identifica el lenguaje de programación utilizado mediante la extensión de los archivos fuente

Inclusión de encabezadores en C

```
#include <stdio.h>
```

```
#include "complejos.h"
```

Inclusión de encabezadores en C++

```
#include <iostream>
```

```
#include "complejos.h"
```



Flexibilidad para declarar variables

En C++ las variables locales pueden ser declaradas en cualquier parte del bloque de código

```
for (double suma = 0.0, int i = 0; i<n; i++)  
    suma += a[i];
```



Datos bool

- En C no existe un tipo de datos **bool** con valores verdadero o falso, hay que simularlo con variables char que adopten valores “0” o “**distinto de 0**” o “**negativo**” y “**positivo**”
- En C++ existen las variable tipo **bool** que pueden tomar los valores **TRUE** y **FALSE**

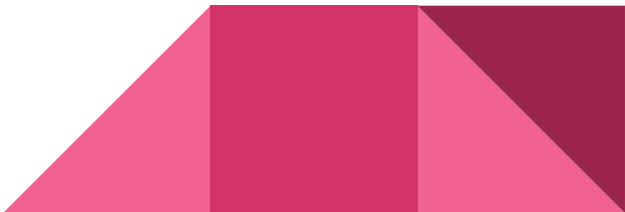


Constantes

Constantes en C con el preprocesador **Dificultades:**


- Haciendo `#define PI 3,1416` las constantes quedan fuera del ámbito del compilador
- No se realiza comprobación de tipos y no se puede obtener la dirección de **PI**
- **PI** dura desde el momento en que es definida hasta el final del archivo.

Constantes en C++ (se declara con la palabra clave **const**) **Ventajas:**

- Es lo mismo que una variable pero su valor NO puede cambiar.
 - Las constantes tienen ámbito => se la puede 'esconder' dentro de una función => no afecta el resto del programa.
 - Al compilar, se verifica la comprobación de tipos.
- 

Paso de valores por referencia

```
void main (void) {  
    int a, b = 4; // b vale 4  
    a = funcion (&b, 6);  
    //b vale posiblemente otro valor  
    -----  
}  
  
int funcion (int *x, int y) {  
    int z;  
    printf ("%d", z);  
    *x = z+y;  
    return z;  
}
```




VERDADERO Paso de valores por referencia

```
void main (void) {  
    int a, b = 4; // b vale 4  
    a = funcion (b, 6);  
    //b vale posiblemente otro valor  
    -----  
}  
  
int funcion (int &x, int y) {  
    int z;  
    printf ("%d", z);  
    x = z+y;  
    return z;  
}
```

x es un **ALIAS** de b

Para declarar una variable Alias se realiza con & en la declaración de la misma.



Visibilidad o Scope

La **visibilidad** o **scope** de una variable es la parte del programa en la que ella está definida y puede ser utilizada

En C++ la visibilidad de una variable puede ser:

1. local
2. a nivel de archivo (global) o
3. a nivel de **clase**.

Al tener declaradas una **variable global** y otra **variable local** del mismo nombre, la **variable global** está oculta por la **variable local**.

scope resolution operator: '::'

Este operador, antepuesto al nombre de una **variable global** que está oculta por una **variable local** del mismo nombre, permite acceder al valor de la variable global



Salida por pantalla y entrada por teclado

Los objetos **cin** y **cout** nos permiten manejar los streams de pantalla y teclado facilmente:

- `cout << variable << "texto" << endl`
- `cin >> variable`

Nos abstraemos del formato de las variables y los datos a imprimir/leer.

cin objeto de la clase ***istream*** y *cout* objeto de ***ostream*** ambas incluidas en la biblioteca ***iostream***



namespace

Es un conjunto de nombres (*identificadores*), agrupados en un '**espacio**' en el cual todos ellos (variables, funciones, identificadores) **son únicos**

```
namespace espacio_2D {  
    struct Punto {  
        int x;  
        int y;  
    };  
}  
namespace espacio_3D {  
    struct Punto {  
        int x;  
        int y;  
        int z;  
    };  
}
```

Un espacio de nombres queda definido por el uso de la palabra clave **namespace** seguida de llaves de apertura y cierre

para utilizar las variables, funciones o identificadores se debe incluir la siguiente línea:

using namespace espacio_2D;



Memoria dinámica

Las funciones de librería malloc() y free() de ANSI C son reemplazadas por los operadores de C++ **new** y **delete**. Su utilización es la siguiente:

- new **tipo_de_datos** o new **tipo_de_datos** [tam_vector]
- delete **tipo_de_datos** o delete [] **tipo_de_datos**

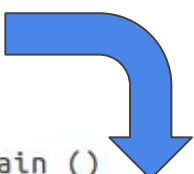
```
long * l, total = 0;
cout << "Cuantos números desea ingresar? ";
cin >> i;
→ l = new long[i];
if (l == NULL) exit (1);
for (n=0; n<i; n++) {
    cout << "Número: ";
    cin.getline (input,100);
    l[n]=atol (input); }
cout << "Usted ingreso: ";
for (n=0; n<i; n++) cout << l[n] << ", ";
→ delete[] l;
return 0; }
```


Constructores

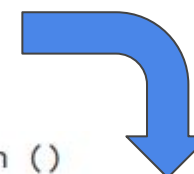
Las clases pueden tener un método **CONSTRUCTOR**. Este método es una función que será invocada en el momento en que se cree el objeto (o sea que se instancie la clase).

El constructor se diferencia del resto de los métodos por llamarse de igual manera que la clase.

```
class Punto {  
    private:  
    int x,y;  
    public:  
    Punto() {  
        x = 0;  
        y = 0;  
    }  
}  
  
void main ()  
{  
    Punto Punto1;  
    ...  
}
```



```
class Punto {  
    private:  
    int x,y;  
    public:  
    Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
}  
  
void main ()  
{  
    Punto Punto1(3,5);  
    ...  
}
```



Destruyores

De la misma manera, el **destructor** es un método que se invoca al momento en que deja de existir el objeto (ya sea porque se libera la memoria con el operador delete o porque se abandona el ámbito de pertenencia del mismo)

Se lo reconoce porque se llama de igual manera que la clase, pero con el operador ~ antecediendo al nombre:

```
class Punto {  
    private:  
        int x,y;  
  
    public:  
        Punto(int a, int b) {  
            x = a;  
            y = b;  
        }  
  
        ~Punto(){  
            cout << "Punto borrado: " << x << " ; " << y << endl;  
        }  
}
```



```
Punto *p;  
...  
p = new Punto(3,5);  
...  
delete p;  
...
```

**LOS CONSTRUCTORES Y DESTRUCTORES
NO PUEDEN RETORNAR VALORES!!**

Sobrecarga de funciones

Otra de las ventajas que ofrece C++ es la posibilidad de sobrecargar una función. Esto es, varios métodos pueden compartir el mismo nombre pero diferenciarse según:

- El TIPO de parámetros que reciban
- La CANTIDAD de parámetros que reciban.

Sin embargo, **no puede** haber dos métodos con igual cantidad y tipo de parámetros, pero con distinto valor de retorno.



Sobrecarga de funciones - constructores:

```
class Punto {  
    private:  
        int    x,y;  
    public:  
        Punto() {  
            x = 0;  
            y = 0;  
        }  
  
        Punto(int a, int b) {  
            x = a;  
            y = b;  
        }  
  
        Punto(const Punto &A) {  
            x = A.x;  
            y = A.y;  
        }  
  
        ~Punto(){  
            cout << "Punto borrado: " << x << " "; << y << endl;  
        }  
  
        int getX();  
        int getY();  
};
```



Constructor por defecto



Constructor parametrizado



Constructor "de copia"

```
Punto *p1,*p2,*p3;  
  
...  
p1 = new Punto(3,5);  
p2 = new Punto;  
p3 = new Punto(*p1);  
  
...  
delete p1;  
delete p2;  
delete p3;  
  
...
```

Un constructor DE COPIA siempre debe recibir una referencia a un objeto (para que no se llame nuevamente a un constructor de copia dentro de otro constructor de copia)

Lista inicializadora

```
class punto {  
    int x_;  
    int y_;  
public:  
    punto (int,int);  
    punto (const punto&);  
    void set_xy (int,  
int);  
}
```

Definición de un constructor parametrizado tradicional

```
punto::punto (int a, int b)  
{  
    x_ = a;  
    y_ = b;  
}
```

//Definición del constructor usando lista de inicializadores

```
punto::punto (int a, int b) : x_(a),y_(b){ }
```

miembros públicos y privados

```
class nombre_clase
{
    especificador_de_acceso_1:
        int miembro_1;
        int miembro_2;
        ...
    especificador_de_acceso_2:
        int miembro_n;
        ...
};
```

- **public:** estos miembros son accesibles cualquier parte de nuestro programa
- **private:** su acceso es restringido. Private es el estatus por defecto de acceso a los miembros de una clase
- **protected:** está vinculado al concepto de “herencia”. (lo veremos en detalle mas adelante)



buenas practicas de programacion - metodos set y get

Se recomienda en el diseño de las clases colocar las propiedades como privadas e incluir los siguientes métodos para acceso a ellas:

- Una función **consultora** (*get*), es una función que retorna un valor desde su objeto, pero no cambia el objeto (sus atributos)
- Una función **modificadora** (*set*), es una función capaz de modificar el “estado” (sus atributos) de un objeto

