

PROGRAMACIÓN ORIENTADA A OBJETOS USANDO C++

1ª EDICIÓN

3ª SESIÓN

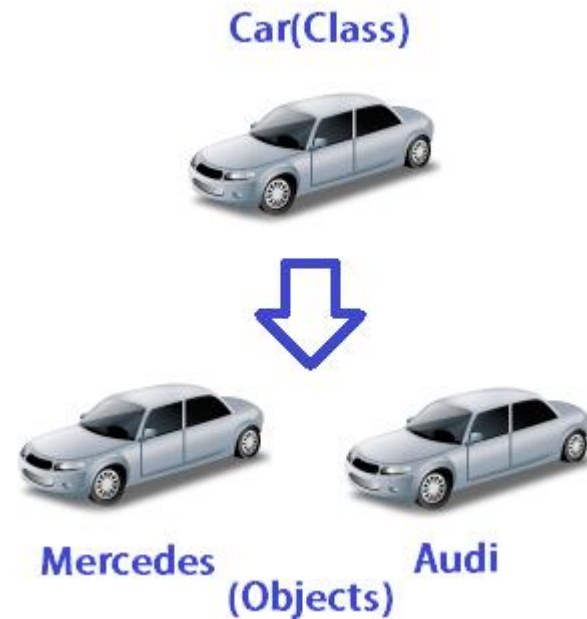
3

Curso 2015-2016

ÍNDICE

1. Clases y Objetos
2. Herencia
3. Sobrecarga de Operadores
4. Polimorfismo
5. Abstracción
6. Otros

1- CLASES Y OBJETOS



1- CLASES Y OBJETOS

- El principal objetivo de C++ es incluir la orientación a objetos al lenguaje de programación C.
- Una clase se utiliza para especificar la forma de un objeto.
- Además proporciona métodos para manipular estos objetos.

1- CLASES Y OBJETOS

- Los datos y funciones de una clase se llaman miembros. Por ejemplo, función miembro o variable miembro.
- Cuando se define una clase, se define un modelo para un tipo de datos (objeto).
- Se define cómo es el objeto y qué operaciones puede realizar.

1- CLASES Y OBJETOS

- Una clase se define con la palabra reservada ***class*** seguida por el nombre de la clase.
- Después entre llaves el contenido de la clase.
- Finalmente se termina con punto y coma.
- Es muy frecuente colocar un guión bajo antes del nombre de las variables miembro de nuestra clase. Es un convenio muy extendido. Ejemplo:

int _peso;

1- CLASES Y OBJETOS

- Vamos a ver un ejemplo de una clase Coche:

```
class Coche {  
    public:  
        int _peso;  
        int _n_puertas;  
        int _potencia;  
        double _precio;  
        string _marca;  
        string _modelo;  
};
```

1- CLASES Y OBJETOS

- La palabra reservada ***public*** determina el acceso a los datos y funciones de la clase. Un miembro ***public*** puede ser modificado y consultado desde cualquier sitio desde dentro o fuera de la clase.
- También existen la palabras reservada ***private*** y ***protected*** para definir el acceso a los miembros de la clase que serán explicados después.
- Miembro hace referencia tanto a variables como a funciones de la clase.

1- CLASES Y OBJETOS

- ¿Cómo definir objetos con la clase que hemos creado?
- Ahora mismo tenemos la clase Coche que representa un modelo para construir coches, ahora el siguiente paso es crear coches (objetos).
- Inicialmente vamos a crear coches sin atributos (vacíos).

1- CLASES Y OBJETOS

- La forma de declarar objetos de tipo Coche es exactamente igual que cuando declaramos una variable. Ejemplo:

Coche coche1;

Coche coche2;

Coche coche3;

- Cada uno de estos Coches (objetos) crea una copia de las variables miembros de la clase Coche.

1- CLASES Y OBJETOS

- ¿Cómo acceder a los miembros de un objeto?
- Para el caso de los objetos Coche, la idea es consular o modificar sus parámetros (miembros del objeto).
- Se realiza con el operador de dirección: el punto (.) ó la flecha (→) si los manejamos con punteros.

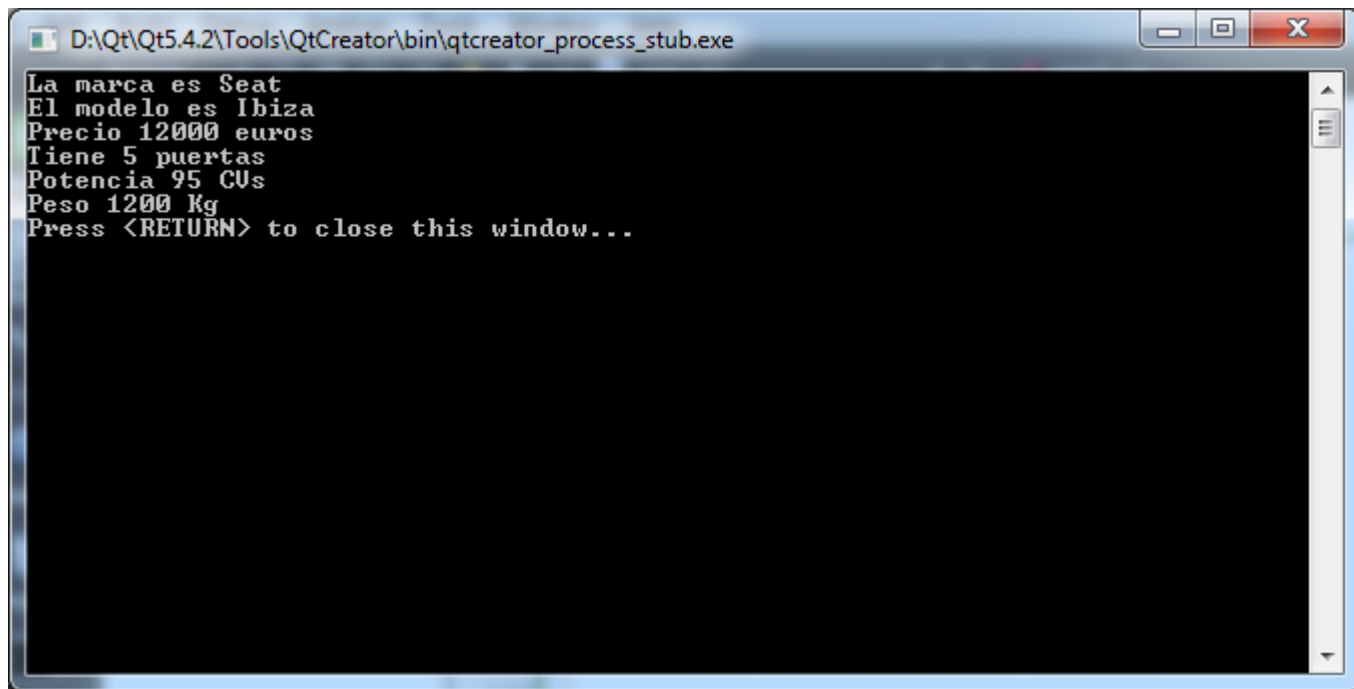
1- CLASES Y OBJETOS

- Vamos a ver un ejemplo para modificar la información del **coche1** y mostrarla:

```
coche1._marca = "Seat";  
coche1._modelo = "Ibiza";  
coche1._n_puertas = 5;  
coche1._peso = 1200;  
coche1._potencia = 95;  
coche1._precio = 12000;
```

```
cout << "La marca es " << coche1._marca << endl;  
cout << "El modelo es " << coche1._modelo << endl;  
cout << "Precio " << coche1._precio << " euros" << endl;  
...
```

1- CLASES Y OBJETOS



A screenshot of a Windows-style application window titled "D:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe". The window has a black background and white text. The text displays the following information: "La marca es Seat", "El modelo es Ibiza", "Precio 12000 euros", "Tiene 5 puertas", "Potencia 95 CVs", "Peso 1200 Kg", and "Press <RETURN> to close this window...". The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a vertical scrollbar on the right side.

```
D:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe
La marca es Seat
El modelo es Ibiza
Precio 12000 euros
Tiene 5 puertas
Potencia 95 CVs
Peso 1200 Kg
Press <RETURN> to close this window...
```

1- CLASES Y OBJETOS

- Añadir **funciones a las clases** para que los objetos puedan usarlas.
- Existen dos formas de añadir estas funciones: directamente dentro de la clase o definirla fuera de la clase. Vamos a verlas:
 - Añadir una función dentro de la clase. Ejemplo:

```
class Coche{  
    public:  
  
    ...  
    double getPotenciaPeso( void ){  
        return _peso/_potencia;  
    }  
};
```

1- CLASES Y OBJETOS

- Definir la función fuera de la clase. Debemos poner la definición dentro de la clase, y se usa exactamente igual que antes:

```
class Coche{  
    public:  
  
        ...  
        double getPotenciaPeso( void );  
  
};  
  
double Coche::getPotenciaPeso( void ){  
    return _peso / _potencia;  
}
```

- Utilizar este método que hemos creado:
 coche1.getPotenciaPeso();

1- CLASES Y OBJETOS

- La ocultación de datos es una de las características más importantes de la programación orientada a objetos.
- De esta forma podremos ocultar ciertas partes del programa y proporcionar el acceso a los datos como nosotros queramos.
- Los miembros de una clase pueden ser: **public**, **protected** o **private**.

1- CLASES Y OBJETOS

- Como hemos dicho antes, los miembros públicos pueden ser accedidos desde cualquier sitio.
- Los miembros **private** solo son accesibles desde dentro de la clase o por funciones amigas que veremos más adelante. Si no ponemos nada, los miembros serán **private**. Ejemplo:

```
class Coche{  
    ...  
    private:  
        int _n_chasis;  
};
```

```
class Coche{  
    ...  
    int _n_chasis;  
};
```

1- CLASES Y OBJETOS

- Si las variables miembro son «**private**» necesitamos crear métodos «**public**» si queremos modificar o consultar estas variables.
- Normalmente se usan las funciones **get()** y **set()**.
- Ejemplo:

public:

```
int getPotencia( void ) { return _potencia; }
```

```
void setPotencia( int potencia) { _potencia = potencia; }
```

1- CLASES Y OBJETOS

- Como hemos dicho antes, la palabra reservada **private** se puede omitir porque por defecto son de este tipo.
- En el ejemplo anterior no tendremos acceso a la variable «**_potencia**» desde fuera de la clase.
- Normalmente nuestras variables son **private** y después creamos una función **public** que sea la que modifique o consulte de forma adecuada el contenido.
- También es frecuente poner como **private** funciones intermedias que no queremos que se usen desde fuera.

1- CLASES Y OBJETOS

- Los miembros de clase **protected** son muy parecidos a los **private**, salvo una excepción que si son accesibles desde las clases derivadas (hijo) que veremos más adelante.
- Los miembros de la clase **protected** son heredados por las clases derivadas (hijo), es decir, tiene acceso a estos miembros como si fueran suyos.

1- CLASES Y OBJETOS

- El **constructor** es una función miembro especial que se ejecuta cuando creamos un nuevo objeto de la clase.
- El **constructor** debe tener exactamente el mismo nombre que la clase y no devuelve nada.
- Los **constructores** son útiles para inicializar nuestras variables miembro.
- Vamos a ver como funciona el **constructor vacío**.

1- CLASES Y OBJETOS

- Vamos a ver un ejemplo con la clase Coche.

```
int n_coches =0;
class Coche{
    ...
    public:
        Coche( void );
    ...
};
Coche::Coche( void ){
    cout << "Estamos creando un coche..." << endl;
    _peso = 0; _n_puertas = 0; _potencia = 0;
    _precio = 0; _marca = "Desconocida";
    _modelo = "Desconocido";
}
```

1- CLASES Y OBJETOS

- También se puede crear un constructor con cualquier número de parámetros de esta forma:

```
int n_coches = 0;
class Coche{
    ...
    public:
        Coche( int peso, int n_puertas);
    ...
};

Coche::Coche( int peso, int n_puertas ){
    cout << "Estamos creando un coche..." << endl;
    _peso = peso; _n_puertas = n_puertas;
    _potencia = 0; _precio = 0;
    _marca = "Desconocida"; _modelo = "Desconocido";
}
```

1- CLASES Y OBJETOS

- Por defecto cuando creamos un objeto se llamará al constructor vacío, pero si tenemos otros constructores también podemos usarlos así:

```
int n_coches = 0;
```

```
Coche coche1; n_coches ++;
```

```
Coche coche3(1500,5); n_coches ++;
```

```
cout << "Peso " << coche3._peso << " Kg" << endl;
```

```
cout << "Tiene " << coche3._n_puertas << " puertas" << endl;
```

```
cout << "El numero de coches es " << n_coches << endl;
```


1- CLASES Y OBJETOS

- El **destructor** es una función miembro de la clase especial que es ejecutada cuando el objeto está fuera de la zona donde fue definido.
- El **destructor** debe tener el mismo nombre que la clase con (~) delante.
- En algunos casos interesa modificar el **destructor** que tenemos por defecto para realizar otras acciones.
- El **destructor** no tiene argumentos de entrada.

1- CLASES Y OBJETOS

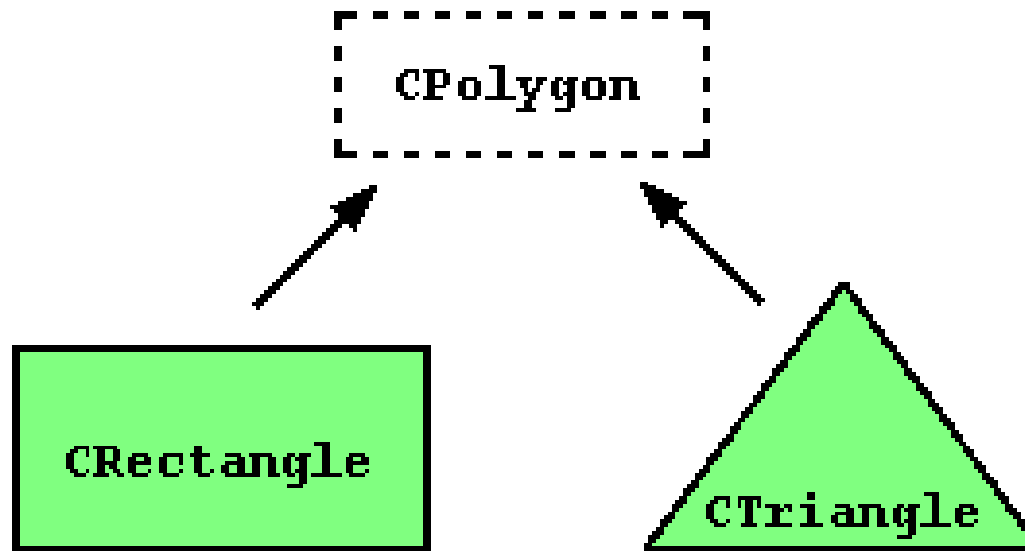
- Vamos a ver un ejemplo:

```
int n_coches = 0;
class Coche{
    ...
    public:
        ~Coche( void );
    ...
};
...
Coche::~~Coche( void ){
    cout << "Estamos eliminando un coche..." << endl;
}
```

EJERCICIO

- Implementa la clase «**Vehiculo**» para que contenga las siguientes variables miembro: `_peso`, `_n_ruedas`, `_n_chasis`, `_potencia`, `_velocidad_max`, `_n_puertas`, `_marca` y `_modelo`. Poner estas variables «public».
- Implementar un constructor que acepte de entrada el nuevo valor de todas las variables miembro.
- Implementar un constructor vacío y un destructor. Utilizar una variable global para contar el número de vehículos que hay en cada momento.

2- HERENCIA



2- HERENCIA

- Uno de los conceptos más importantes en la Programación Orientada a Objetos es la herencia.
- La herencia nos permite definir una clase cogiendo contenido de otra clase que ya tengamos.
- Esta estrategia permite reusar y reducir código.
- La clase existente se suele llamar **base** o padre, y la nueva clase **derivada** o hija.

2- HERENCIA

- Una clase derivada (o hija) puede heredar de una clase base (padre) o varias.
- Para heredar necesitamos una lista de la clase(s) base desde las que se va a heredar.
- Además debemos definir que tipo de herencia (tipo acceso que tenemos) sobre la clase base. Puede ser «**public**», «**protected**» y «**private**». Que veremos más adelante.
- Se define así: *class nueva_clase : acceso clase_base1 , acceso clase_base2 , ...*

2- HERENCIA

- Vamos a ver un ejemplo:

// Clase base

```
class Forma {  
    public:  
        void setAnchura(int an){ _anchura = an; }  
        void setAltura(int al) { _altura = al; }  
    protected:  
        int _anchura; int _altura;  
};
```

// Clase derivada

```
class Rectangulo: public Forma {  
    public:  
        int getArea(){ return (_anchura * _altura); }  
};
```

// Clase derivada

```
class Triangulo: public Forma {  
    public:  
        int getArea(){ return (_anchura * _altura / 2); }  
};
```

2- HERENCIA

- Vamos a ver un ejemplo del uso:

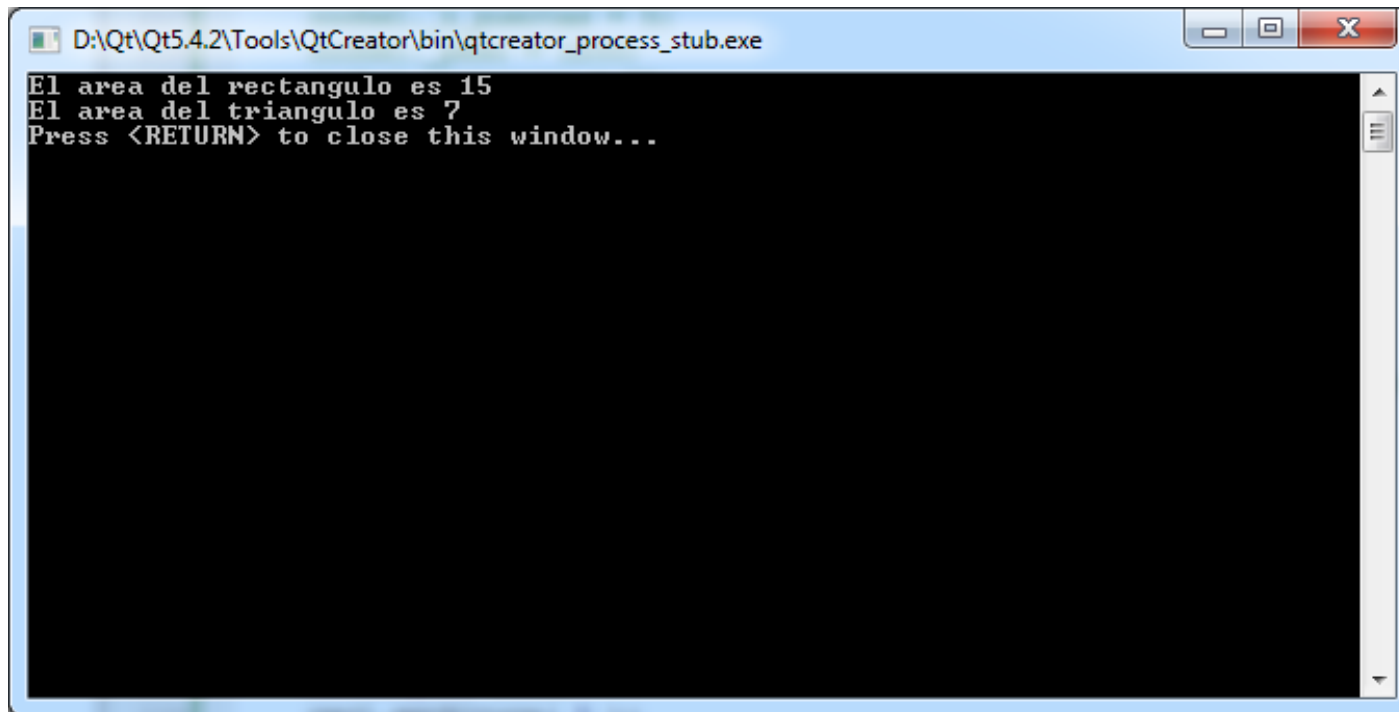
```
Rectangulo rec1;  
rec1.setAltura( 5 );  
rec1.setAnchura( 3 );
```

```
Triangulo tri1;  
tri1.setAltura( 5 );  
tri1.setAnchura( 3 );
```

```
cout << "El area del rectangulo es " << rec1.getArea() << endl;  
cout << "El area del triangulo es " << tri1.getArea() << endl;
```


2- HERENCIA

- Vamos a ver un ejemplo:



A screenshot of a Windows-style application window titled "D:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe". The window has a black background and white text. The text displayed is:

```
El area del rectangulo es 15  
El area del triangulo es 7  
Press <RETURN> to close this window...
```

2- HERENCIA

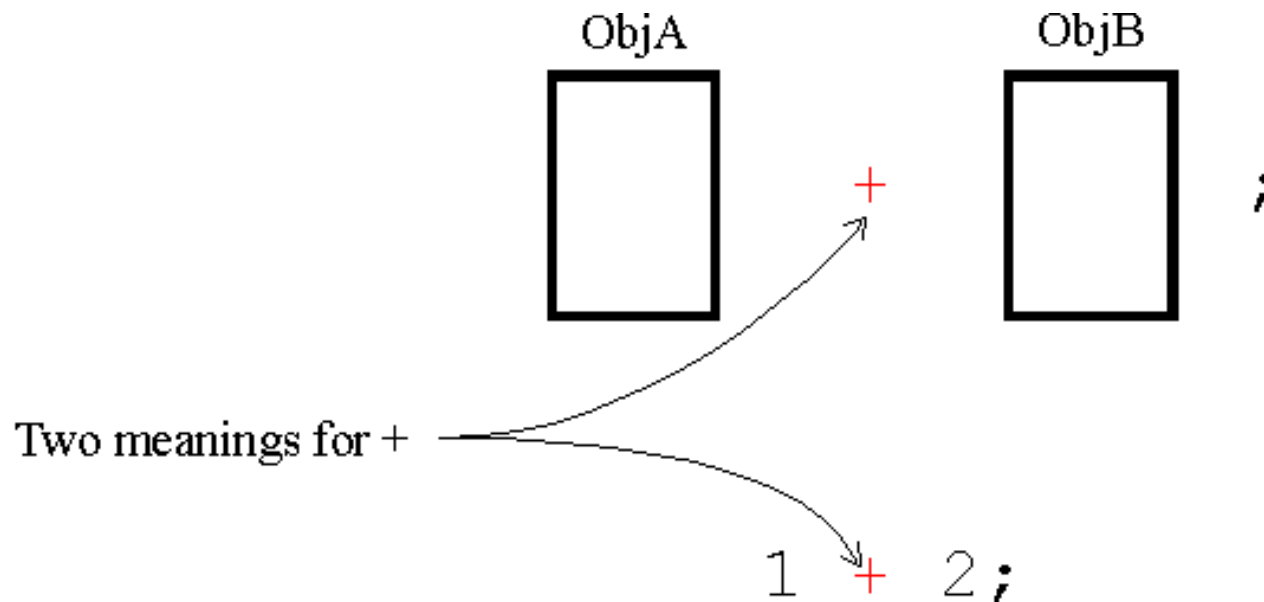
- Los tipos de herencia (tipo de acceso) funcionan así:

Inheritance	base class	derived class
<i>public</i>	public: • protected: • private:	public: • protected: • private:
<i>protected</i>	public: • protected: • private:	public: • protected: • private:
<i>private</i>	public: • protected: • private:	public: • protected: • private:

EJERCICIO

- Implementar las clases «Coche» y «Camion» que hereden de la clase vehículo.
- Implementar los métodos «calcularImpuesto()» y «obtenerVelocidadMaxAutovia()».
 - El impuesto es calculado así: $_potencia * _peso * _n_ruedas / 10000$. Los camiones pagan el doble.
 - Tener en cuenta que los coches tienen una velocidad máxima de 120 y los camiones de 100 en autovía. Si la velocidad máxima del vehículo es inferior dar su velocidad.
- ¿Es necesario implementar el constructor y destructor? Si lo es, implementarlos.

3- SOBRECARGA DE OPERADORES



3- SOBRECARGA DE OPERADORES

- C++ permite especificar los operadores para las clases.
- Se pueden sobrecargar más de un operador.
- Para sobrecargar un operador usamos la palabra reservada **operator** seguida del operador que queremos sobrecargar.
- La sobrecarga de operadores debe tener un argumento de retorno.

3- SOBRECARGA DE OPERADORES

- La definición es así:
 - `Coche operator+(Coche & c1);` ó
 - `Coche operator+(Coche c1);`
- Vamos a ver un ejemplo de sobrecarga de operadores con la clase `Coche` del ejercicio anterior.
- Vamos a sobrecargar la suma diciendo que cuando sumamos dos coches creamos un nuevo coche con la suma de peso y potencia.

3- SOBRECARGA DE OPERADORES

```
class Coche : public Vehiculo{  
    ...  
    public:  
        Coche operator+(Coche c1);  
    ...  
};  
  
Coche Coche::operator+(Coche c1){  
    Coche c;  
    c._peso = _peso + c1.getPeso();  
    c._potencia = _potencia + c1.getPotencia();  
    return c;  
}
```

3- SOBRECARGA DE OPERADORES

Coche c2(1200, 4, 5680, 110, 180, "Seat", "Altea");

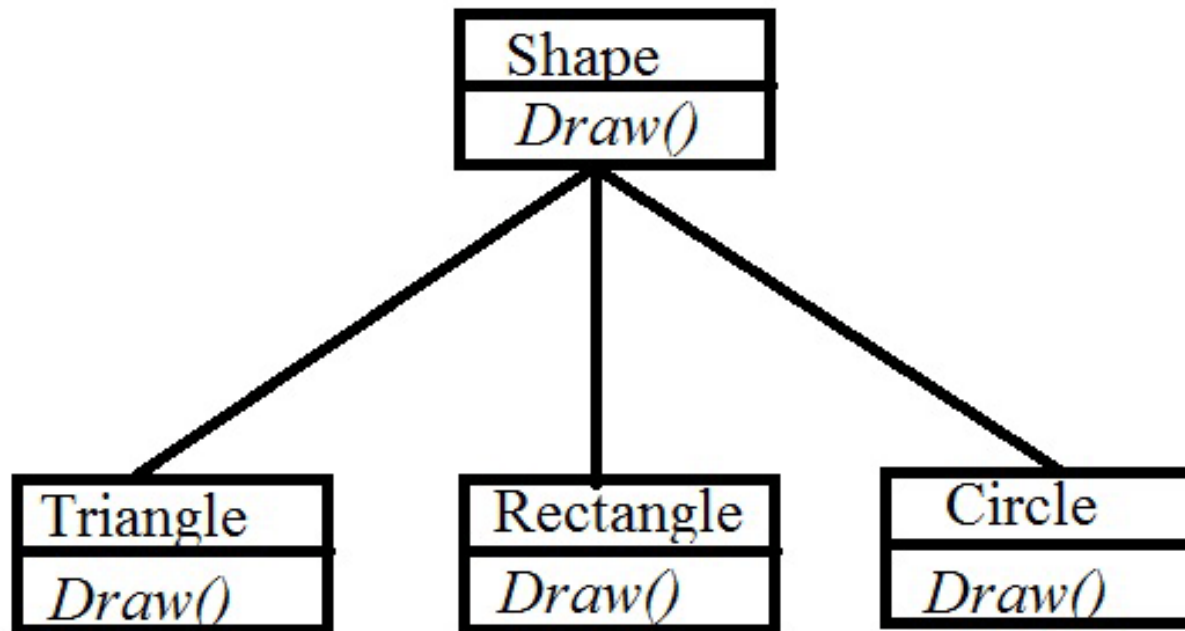
Coche c3(1200, 4, 1200, 110, 180, "Seat", "Altea");

cout << "El peso de c2 y c3 " << (c2+c3).getPeso() << endl;

EJERCICIO

- Implementar el operador resta para la clase Coche. Sabiendo que la resta de dos coches da otro coche como resultado con la diferencia en potencia, peso, n_chasis y velocidad_max.

4- POLIMORFISMO



4- POLIMORFISMO

- La palabra polimorfismo significa tener muchas formas.
- Normalmente el polimorfismo ocurre con la herencia de clases.
- En C++ el polimorfismo significa que la función indicada de la clase base pueden ser implementadas por las clases derivadas.

```
virtual int area(){  
    ...  
};
```

- Vamos a ver un ejemplo para entenderlo mejor.

4- POLIMORFISMO

```
class Forma {  
    protected:  
        int _anchura, _altura;  
    public:  
        Forma( int a=0, int b=0) {  
            _anchura= a; _altura= b;  
        }  
        virtual int area() {  
            cout << "Area en Forma" << endl;  
        }  
};
```

4- POLIMORFISMO

```
class Rectangulo: public Forma{
    public:
        Rectangulo( int a=0, int b=0):Forma(a, b) { }
        int area () {
            cout << "Area clase Rec" << endl;
            return (_anchura * _altura);
        }
};
```

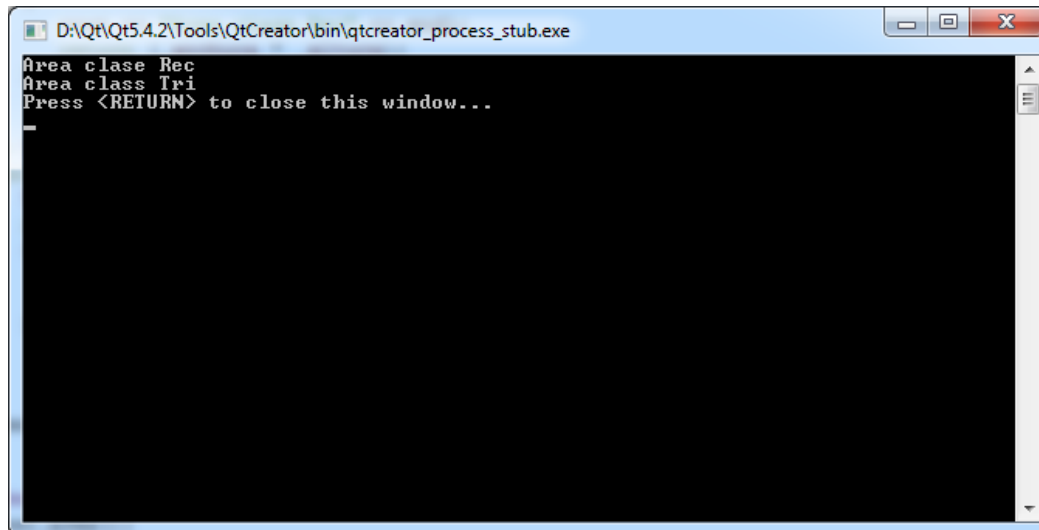
```
class Triangulo: public Forma{
    public:
        Triangulo( int a=0, int b=0):Forma(a, b) { }
        int area () {
            cout << "Area class Tri" << endl;
            return (_anchura * _altura / 2);
        }
};
```

4- POLIMORFISMO

Si ahora ejecutamos esto en el main():

```
Rectangulo rec(10,20);  
rec.area();
```

```
Triangulo tri(10,20);  
tri.area();
```

A screenshot of a Qt Creator console window. The title bar shows the file path "D:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe". The console output displays the following text: "Area class Rec", "Area class Tri", and "Press <RETURN> to close this window...". A single line of output, "-", is visible below the instructions.

```
D:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe  
Area class Rec  
Area class Tri  
Press <RETURN> to close this window...  
-
```

EJERCICIO

- Utilizando la clase «Forma» que hemos visto antes y las clases «Rectangulo» y «Triangulo». Añadir a la clase base una función polimórfica «perimetro», y que las clases derivadas la implementen de forma adecuada.
 - Suponer que el triangulo es isósceles para calcular su perímetro.

4- POLIMORFISMO. INTERFACES

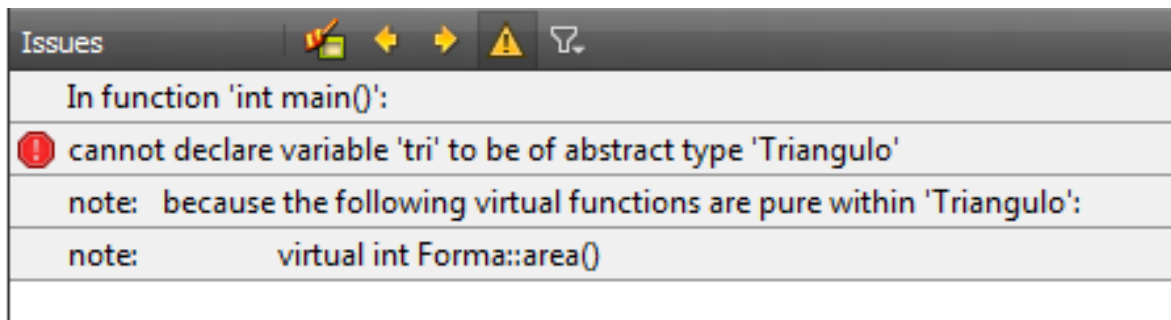
- Una interfaz describe el comportamiento o forma de una clase de C++ que no tiene una implementación.
- La interfaces en C++ también usan el concepto de **abstract class** que podría ser algo confuso.
- La diferencia para crear una interfaz es terminar la función con «= 0» y no tener la implementación.

4- POLIMORFISMO. INTERFACES

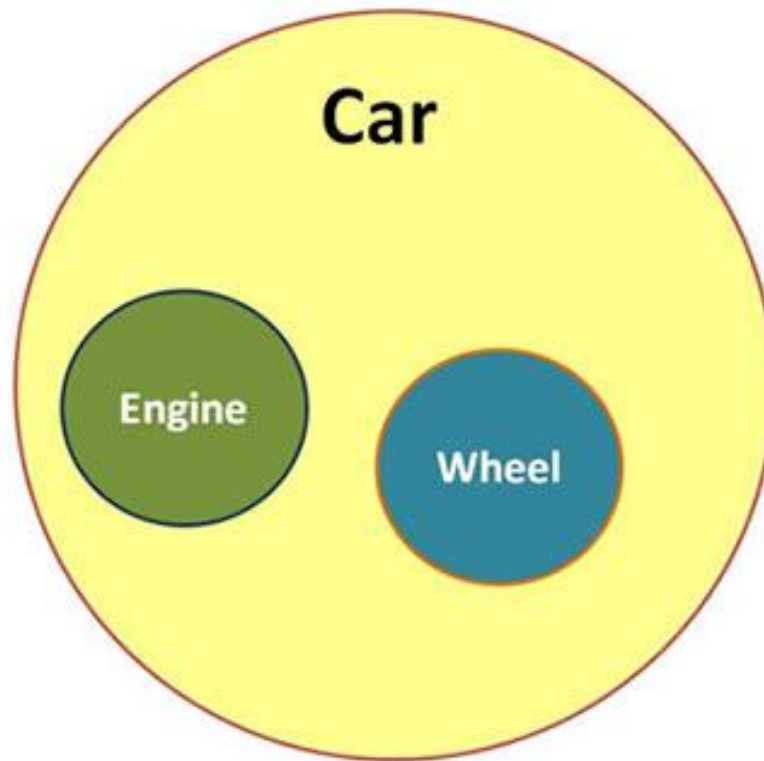
- Las diferencias de una interfaz respecto al concepto de polimorfismo son dos:
 - La clase interfaz no tiene una implementación en la clase base.
 - Todas las clases derivadas deben implementar la interfaz.
- El formato de una interfaz en C++ es así:
 - `virtual int nombreInterfaz() = 0;`

4- POLIMORFISMO. INTERFACES

- Por ejemplo si no implementamos la función área en la clase Triangulo.
- El compilador nos dará un error de este tipo:



5- ABSTRACCIÓN



5- ABSTRACCIÓN

- La abstracción hace referencia en dar solo la información esencial al mundo y ocultar los detalles.
- En programación, la abstracción sirve para separar la implementación (funcionamiento) de la interfaz.
- Viendo un ejemplo en la vida real se entiende mejor este concepto. Nosotros sabemos usar el coche con el volante, la palanca, los pedales, ...(esto sería la interfaz). Pero no tenemos porque saber como funciona por dentro las partes del coche (funcionamiento).

5- ABSTRACCIÓN

- La abstracción en términos de C++ provee un nivel superior de abstracción.
- La idea es crear métodos «**public**» que pueden ser usando desde fuera (interfaz). Y por otra parte, crear métodos y variables miembro como «**private**» donde no queramos mostrar el funcionamiento.

5- ABSTRACCIÓN

- Las ventajas de la abstracción de datos son:
 - Las clases internas («private») están protegidas del exterior. Por lo tanto, son invariantes a errores de usuario que podrían provocar problemas en el estado del objeto.
 - Además la implementación de las funciones internas («private») pueden ser modificadas para atender nuevos requisitos sin cambiar la forma de usarla (interfaz).
- Vamos a ver un ejemplo en C++.

5- ABSTRACCIÓN

```
class Sumador{  
    public:  
        // Constructor  
        Sumador(int i = 0) { _total = i; }  
  
        // Interfaz para usar desde fuera  
        void sumar(int numero) { _total += numero; }  
  
        // Interfaz para usar desde fuera  
        int getTotal() { return _total; };  
  
    private:  
        // Datos ocultos al exterior  
        int _total;  
};
```

5- ABSTRACCIÓN

Sumador sumador;

sumador.sumar(10);

sumador.sumar(25);

cout << "El total es " << sumador.getTotal() << endl;

EJERCICIO

- Modificar la clase «vehiculo» para ocultar las variables miembro.
- Implementar el acceso desde fuera («public») para poder consultar y modificar las variables miembro.

6- OTROS



6- OTROS. FUNCIÓN AMIGA

- Una función amiga «friend» significa que de una clase se puede acceder a todas las variables y funciones protegidas («private» and «protected») de la misma desde fuera.
- Puede haber funciones, clases y plantillas amigas.
- Para definir una función amiga, se usa la palabra reservada **friend** que es colocada delante de la función.
- Vamos a ver un ejemplo para entenderlo mejor.

6- OTROS. FUNCIÓN AMIGA

```
class Caja {  
    private:  
        double _anchura;  
  
    public:  
        // Nota: la función amiga no es una función miembro de Caja  
        friend void imprimirAnchura( Caja caja );  
  
        // Función miembro  
        void setAnchura( double anchura ) {  
            _anchura = anchura;  
        }  
};  
  
void imprimirAnchura(Caja caja){  
    cout << "La anchura de la caja es : " << caja._anchura << endl;  
}
```

6- OTROS. FUNCIÓN AMIGA

- Ahora dentro de nuestra función main():

// Creamos una caja...

Caja caja;

// Modificamos la caja...

caja.setAnchura(10.0);

// Usamos la función amiga para imprimir...

imprimirAnchura(caja);

6- OTROS. FUNCIÓN INLINE

- En C++ la función **inline** es un concepto útil y potente. Es bastante usada en las clases.
- Cuando un función es **inline**, el compilador copia el contenido de la función en todos los puntos donde esta la llamada a la función.
- Cuando hacemos un cambio en la función debemos compilar todo el código, ya que el código donde aparezca la llamada cambiará.

6- OTROS. FUNCIÓN INLINE

- Las funciones **inline** suelen ser más rápidas en tiempo de ejecución.
- Se recomienda convertir a **inline** funciones cortas que son llamadas muchas veces.
- Vamos a ver un ejemplo para entenderlo mejor:
 - Creamos una función Suma **inline** y otra normal.

6- OTROS. FUNCIÓN INLINE

// Funcion inline

```
inline int SumaA(int x, int y) {  
    int result = x + y;  
    return result;  
}
```

// Funcion normal

```
int SumaB(int x, int y) {  
    int result = x + y;  
    return result;  
}
```


6- OTROS. FUNCIÓN INLINE

- ¿Cómo podemos medir el tiempo para saber que función es más rápida en tiempo de ejecución?
- Para medir el tiempo en C++ podemos usar esta librería:
 - *#include <ctime>*
- Después usando la función **clock()** podemos obtener la hora actual del sistema.
- Finalmente usando la constate **CLK_TCK** podemos convertir el tiempo transcurrido a segundos.

6- OTROS. FUNCIÓN INLINE

- Si hacemos una ejecución y medimos el tiempo no vamos a ver la diferencia ya que son tan rápidas que nos dará cero el resultado.
- Vamos a crear un bucle (por ejemplo 10Millones) de llamadas a las dos funciones suma.
- Primero a una y luego la otra.

6- OTROS. FUNCIÓN INLINE

```
double t_inicial = clock();
```

```
for (int i=1; i < 10000000 ; i++){  
    int a = SumaA(20,10);  
}
```

```
double t_final = clock();  
double seconds = (t_final-t_inicial)/CLK_TCK ;  
printf( "%.10f segundos \n", seconds);
```

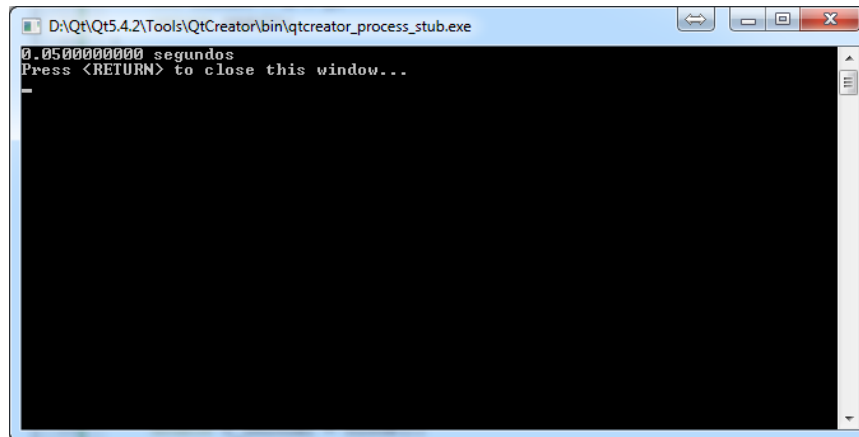
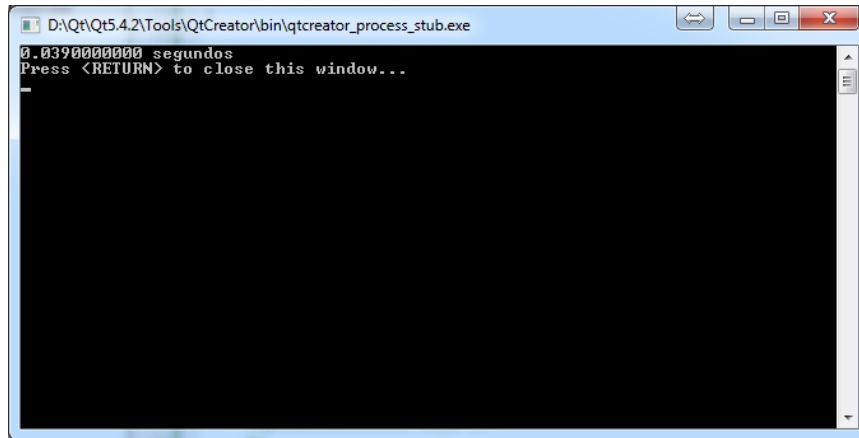
6- OTROS. FUNCIÓN INLINE

```
double t_inicial = clock();
```

```
for (int i=1; i < 10000000 ; i++){  
    int a = SumaB(20,10);  
}
```

```
double t_final = clock();  
double seconds = (t_final-t_inicial)/CLK_TCK ;  
printf( "%.10f segundos \n", seconds);
```

6- OTROS. FUNCIÓN INLINE



6- OTROS. FUNCIÓN INLINE

- El tiempo en realizar 10 Millones de sumaA() es de 0.039 segundos.
- El tiempo en realizar 10 Millones de sumaB() es de 0.050 segundos.
- El incremento en la velocidad usando la función inline es de $0.05/0.039 = 1.28$
- Un 28% más rápido solo añadiendo una palabra.