Buscando la paz interior...

twiter: @rleon1961

TEMA:

Programación OOP

fuente: www.prometec.net

Versión en LaTeX:
Ing. Ricardo De León López

Índice

1. Clases y Objetos

1.1. Las Clases en Arduino

Antes de que empecemos a hablar sobre Clases y Objetos, es importante insistir en que, la OOP no es tanto un lenguaje de programación diferente, sino más bien, una manera diferente de organizar tus programas y tus ideas, de acuerdo con unos principios guía que nos permiten modelar nuestro código de un modo distinto a como lo hemos hecho hasta ahora.

La OOP consiste en organizar tus programas de otra forma, que nos evite los problemas que mencionábamos en la sesión anterior, pero seguimos usando C++ con algunos añadidos.

Para definir las Clases, existen una serie de reglas y de nuevas instrucciones, pero por lo demás el lenguaje sigue siendo el de siempre.

La diferencia, es que ahora vamos a empezar definiendo unos entes abstractos que llamamos Clases y que son la base de la OOP.

En esta sesión daremos los primeros pasos con las Clases y su terminología Veremos cómo definir Clases y Objetos y veremos cómo acceder a las propiedades o variables miembros de la Clase y sus métodos o funciones miembros.

Escribiremos un par de programas completos que involucren Clases y veremos cómo usarlas.

Así que poneros cómodos, sujetad el temblor de rodillas y vamos a lío.

1.2. Nuestro primer programa con clases

Hay que empezar por algún sitio y para ello nada mejor que con un pequeño programa de muestra como un contador (Que parece que se ha convertido en una norma general).

¿Cómo harías un contador general, en programación estructurada?

Pues una variable global que ponemos a 0 y que vamos incrementando en función de la necesidad.

Pero esto ilustra bastante bien el problema de que si quiero 6 contadores voy a necesitar 6 variables globales, con el riesgo que supone que algún memo nos las borre en el momento más inoportuno por cualquier razón estúpida que solo el comprende, así que ni hablar de esto.

La idea en OOP es crear una Clase que nos permita definir Objetos tipo Contador que se pueda reutilizar y que nos permita mezclar diferentes contadores en un mismo programa sin riesgo de catástrofe.

Una de las ideas básicas tras la OOP es encapsular los datos y las funciones (O propiedades y métodos) de nuestro programa en un contenedor

común, y más importante aún, aplicamos el principio de: "Esconder los datos y mostrar los métodos o funciones".

Iremos hablando más de esto, pero de entrada conviene destacar que si escondemos los datos, pero proporcionamos las funciones que trabajan con ellos vamos a reducir drásticamente la posibilidad de que alguien nos la líe.

Por eso cuando definimos Clases, veremos que hay partes que son públicas y otras que son privadas (Y si no se especifica lo contrario son privadas. Volveremos a esto)

La sintaxis para definir la Clase contador que nos ocupa:

```
class Contador
  {
       private:
             ............
       public:
             ............
  } ;
```

Debajo de la cláusula "private:" viene las variables y funciones ocultas al exterior. Solo pueden ser invocadas desde el interior de la clase, es decir no se pueden ejecutar por una llamada exterior. Y lo contrario ocurre con lo que definamos tras la cláusula "public:"

Vamos a definir una variable privada llamada N, que llevará la situación del contador, y después necesitaremos los métodos necesarios para trabajar con ella.

En principio vamos a definir tres funciones públicas: Una que ponga el contador a un valor dado antes de nada, Otra que sirva para incrementar el contador, y otra tercera para que nos entregue el valor del contador en un momento dado. Debajo de la cláusula "private:" viene las variables y funciones ocultas al exterior. Solo pueden ser invocadas desde el interior de la clase, es decir no se pueden ejecutar por una llamada exterior. Y lo contrario ocurre con lo que definamos tras la cláusula "public:"

Vamos a definir una variable privada llamada N, que llevará la situación del contador, y después necesitaremos los métodos necesarios para trabajar con ella.

En principio vamos a definir tres funciones públicas: Una que ponga el contador a un valor dado antes de nada, Otra que sirva para incrementar el contador, y otra tercera para que nos entregue el valor del contador en un momento dado.

Nuestra clase podría ser algo así: (Fijaros en el";" al final)

```
class Contador
  {  private:
         int N ;

     public:
          void SetContador( int n)
             {  N = n ;    }

          void Incrementar()
             {  N++ ; }

          int GetCont()
             { return (N) ;}
  } ;
```

Dentro de las llaves de la Clase definimos las funciones y variables que necesitamos como hasta ahora, y en este caso son de lo más simples. La variable N se comporta como si fuera una variable global pero solo dentro del ámbito de la Clase (Encapsulación), y como está definida como prívate es inaccesible desde el exterior (Cualquier intento de llegar a ella causará un ladrido del compilador).

```
Este es el principio básico de encapsulación: Lo que pasa en la Clase,
se queda en la Clase. Y para acceder a ello, se debe declarar expresamente
como público..
```

Pues eso es todo, no era para tanto después de todo ¿No?

¿Y cómo se usa nuestra flamante primera Clase? Pues como otras que habéis usado antes. Primero se instancian tantas ocurrencias de la clase como queramos. Vamos a usar dos:

```
Contador C1,C2 ;
```

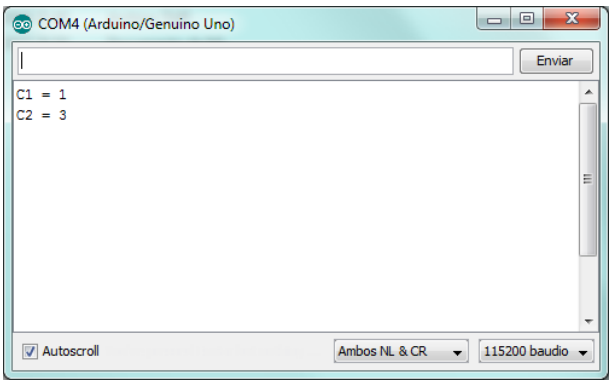
Podemos montar un programa que sea algo así:

```
void loop()
   {  C1.SetContador(0);
      C1.Incrementar() ;
      Serial.print("C1 = ") ; Serial.println( C1.GetCont() ) ;

      C2.SetContador(0);
      C2.Incrementar() ; C2.Incrementar() ; C2.Incrementar() ;
```

```
    Serial.print("C2 = ") ; Serial.println( C2.GetCont() ) ;

    Serial.flush();   exit(0);      // Abandonar el programa
}
```

Usamos el "." para referir la funcion a la que queremos llamar, con el Objeto al que se le aplica, como hemos visto antes en otros programas aunque sin entrar en muchos detalles.

Una vez creados un par de contadores, lo primero que hacemos es ponerlos a 0 (Con SetContador()), después hacemos un par de llamadas a Incrementar y cuando nos hartemos imprimimos el valor del contador. Aquí tenéis el resultado: Consola



No ganaremos premios con este programa, pero a cambio ilustra muy bien algunos conceptos básicos de la programación con Clases.

El primero es que una cosa es la definición de la Clase y otra distinta la instanciación. La clase es Contador pero el compilador no asigna memoria hasta que creamos par de instancias de la misma: C1 y C2. Ahora si que se crean los objetos.

Una Clase, pero tantas ocurrencias como sean precisas, que no se mezclan, son distintos objetos.

Hemos escondido las variables miembros, pero proporcionamos las funciones o métodos necesarios para manejar los objetos, y va a ser difícil que alguien enrede las variables globales porque no existen. ¿Qué te parece?

Podemos crear tantos contadores independientes como queramos, con absoluta certeza de que cada uno está aislado delos demás.

1.3. Refinando nuestra Clase: Constructores

La Clase anterior esconde una bomba de relojería, porque el que la use tiene que ser consciente de que por cada instancia que creemos de Contador, necesitamos una instrucción que la inicialice:

```
Contador C1 ;
C1.SetContador(0);
```

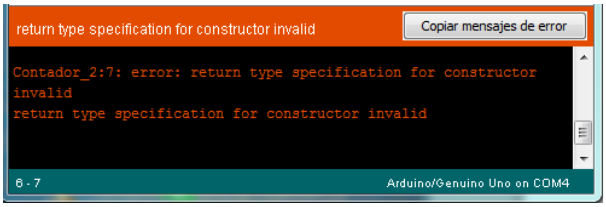
Y esto no solo es un asco, sino que además es peligroso, ya que un despiste de no inicializar, y el valor de N es impredecible. Sería conveniente que el contador se pusiese a 0 al crearse, ya que la mayor parte de los contadores empiezan en 0 y nos despreocupamos de olvidos.

Podemos definir una función que se ejecuta siempre que se crea un objeto, y es tan habitual que tiene nombre. Se le llama Constructor de la Clase, y para ello basta con llamarla igual que la Clase (Sin tipo):

```
class Contador
   {  private:
          int N ;

      public:
          Contador( )                 // Constructor
             { N = 0 ; }

          void SetContador( int n)
             {  N = n ;    }

          void Incrementar()
             {  N++ ; }

          int GetCont()
             { return (N) ;}
  } ;
```

Una peculiaridad de los constructores es que no tienen un tipo definido, otra de las razones por las que el compilador sabe que es un constructor. Si intentas esto de abajo, el compilador protestará amargamente:

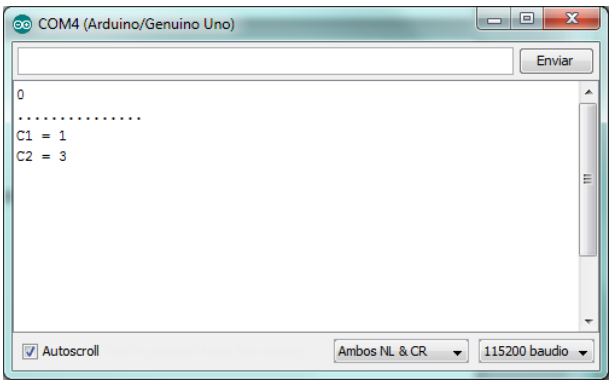
```
public:
     void  Contador( )                // Constructor
```



Usando el constructor, podemos reescribir el programa anterior así, sin problemas:

```
void loop()
   { Serial.println( C1.GetCont() ) ;
     Serial.println("..............");

     C1.Incrementar() ;
     Serial.print("C1 = ") ;
     Serial.println( C1.GetCont() ) ;

     C2.Incrementar() ; C2.Incrementar() ; C2.Incrementar() ;
     Serial.print("C2 = ") ; Serial.println( C2.GetCont() ) ;

     Serial.flush();   exit(0);
  }
```

Aquí está el resultado:

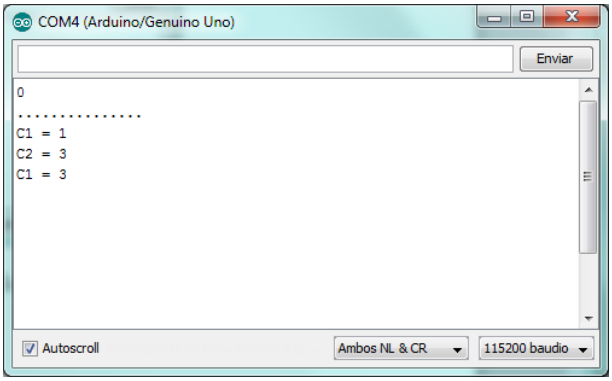


Como veis el constructor inicializa a 0 el valor interno al crear el objeto y nos podemos olvidar tranquilamente de obligaciones.

Ya que estamos, es interesante destacar que podemos hacer esto, ya que son objetos del mismo tipo:

```
C1 = C2 ;
Serial.print("C1 = ") ;
Serial.println( C1.GetCont() ) ;
```

Con este resultado:



Por ultimo, me gustaría hablar de la forma de inicializar el Constructor, que está escrito de una forma de lo más sencilla y comprensible, así que alguien tenía que complicarlo un poco (A mí no me miréis) y para darle mayor prestancia es muy habitual escribirlo de otra forma:

```
public:
      Contador( ) : N(0)   { }              // Constructor
```

Donde N(0) es el nombre de la variable miembro a inicializar y el numero entre paréntesis, es el valor deseado. En el caso de que fueran varias las variables internas a las que queremos pasar valores cuando el objeto se crea, la sintaxis a usar es:

```
public:
      Contador( ) : N(0) , M(4) , P(44)              // Constructor
            {}
```

Se me escapan las razones por las que algo así ha llegado a ser lo habitual, pero os hartareis a verlo si revisáis las librerías de Arduino, así que yo con informaros cumplo.

```
Parece que hay variables que pueden ser inicializadas, pero que son
problemáticas para asignarse por programa (Como por ejemplo las constantes)
y por eso algunos recomiendan seguir este último método siempre que sea
posible.
```

1.4. Definiendo fuera las funciones miembros

Cuando las clases y las funciones miembro son tan pequeñas y sencillas como en este caso, la forma que hemos visto de definirlas puede valer, pero en seguida se quedará corta.

Por eso podemos declarar las funciones y variables miembros en la declaración de Clase, y definirlas fuera para mayor comodidad y evitar errores de sintaxis complicados de detectar.

Vamos a reescribir la clase Contador así:

```
class Contador
   {  private:
         int N ;

      public:
         Contador( ) ;                    // Constructor
         void SetContador( int n) ;       // Declaracion de funcion externa
         void Incrementar() ;             // Declaracion de funcion externa
         int GetCont() ;                  // Declaracion de funcion externa
   } ;
// ---------------------------------------
   void Contador::SetContador( int n)
        {  N = n ;    }

   void Contador::Incrementar()
        {  N++ ; }

   int Contador::GetCont()
        { return (N) ;}
```

Declaramos las funciones miembros dentro de la Clase (Para informar al compilador), pero no incluimos su código aquí, porque sería muy confuso en cuanto crezcan de tamaño (Pero fijaros que ahora hay un punto y coma al final de las declaraciones que antes no había)

En cualquier otro lugar podemos definir esas funciones sin más que hacer referencia a la Clase a la que pertenecen usando el operador '::' (Scope Operator u Operador Ámbito) y el compilador entiende que son miembros de la clase que precede al operador.

Este operador le indica al compilador, que estas funciones o variables son miembros de la clase, y solo pueden invocarse de acuerdo a las condiciones que se especifican en la declaración de la Clase (Que debe coincidir con esta claro está).

Si editáis cualquiera de las librerías de Arduino, encontrareis que ésta es la forma habitual de programar las clases y librerías (Pero mucho ojo, con cambiar nada por la cuenta que os tiene)

```
En algún momento tendremos que hablar de cómo se organizan las librerías
en diferentes módulos y ficheros, pero aun es un poco pronto.
```

1.5. Clases, Objetos y uso de memoria

Con lo que hemos visto hasta ahora, parece que aunque la definición de la Clase es única. Cuando instanciamos los objetos de esa clase, cada uno recibe una asignación de memoria suficiente para contener todas las variables y funciones miembros.

Me gusta que penséis así porque ayuda conceptualizar los objetos, pero me temo que en la realidad las cosas son un poco diferentes, aunque no mucho. Es verdad que cuando creamos un objeto el compilador asigna memoria para contener todas las propiedades de ese objeto en concreto, ya que es lo que diferencia a un objeto de otro de la misma Clase.

Pero no es verdad que se asignen copias de los métodos de la clase a cada instancia. Y el motivo es que no hace falta, ya que el compilador sabe que las funciones miembros son comunes y esto no causa ningún problema y además nos permite ahorra memoria que es algo a tener en cuenta.

El motivo de hacer este comentario, no es tanto volveros locos (Que siempre mola) como presentar otra posible directiva a tener en cuenta cuando definimos una Clase que ya conocemos: static.

Si recordáis, cuando en una función definíamos una variable como static, se creaba una sola vez y persistía disponible, por muchas veces que entráramos en la función, a diferencia de las variables normales que se creaba y destruían cada vez que ejecutamos la función.

Cuando declaramos una propiedad miembro de una clase como static, el compilador crea una única variable para ella, que es compartida por todas las instancias de los objetos de esa Clase, rompiendo así la regla de que cada instancia tiene su propio juego de memoria y propiedades.

¿Y porque vamos a querer hacer algo tan extraño?

Bueno la vida es complicada y a veces hacen falta excepciones. Por ejemplo, si por algún motivo necesitamos saber cuántas instancias de un objeto se han creada en un momento dentro del programa, podemos usar una variable static para interrogar a cualquier objeto de la clase, ya que comparten el campo static y eso es algo que sería imposible de hacer de ninguna otra manera que se nos ocurra.

```
Si alguien va a decir algo que incluya las palabras variable y global,
```

que se ponga inmediatamente de cara a la pared el próximo cuarto de hora, y haga severo acto de contricción.

Veamos un pequeño ejemplo:

```
class Contador
   {    private:
           int N ;
           static int Num ;

        public:
           Contador( ) ;                 // Constructor
           void SetContador( int n) ;    // Declaracion de funcion externa
           void Incrementar() ;          // Declaracion de funcion externa
           int GetCont() ;               // Declaracion de funcion externa
   } ;
```

Añadimos una variable static llamada Num que llevara la cuenta del numero de contadores que vamos a crear. He modificado ligeramente las funciones miembros :

```
Contador::Contador( )                     // Constructor
    { N = 0 ;
      ++Num ;
    }
void Contador::SetContador( int n)
    {  N = n ;
       ++Num ;
    }
void Contador::Incrementar()
    {  N++ ; }

int Contador::GetCont()
    { return (N) ;}

int Contador::Num_Objetos()
    { return(Num) ;   }
```

Básicamente he modifica el Constructor del objeto para que incremente la variable static Num, incrementándola cada vez que se ejecute (O sea cada vez que se cree un objeto de esta Clase) y añadido un nuevo método, Num_Objetos(), que nos devuelve el valor de Num.

Si usamos un programa como este:

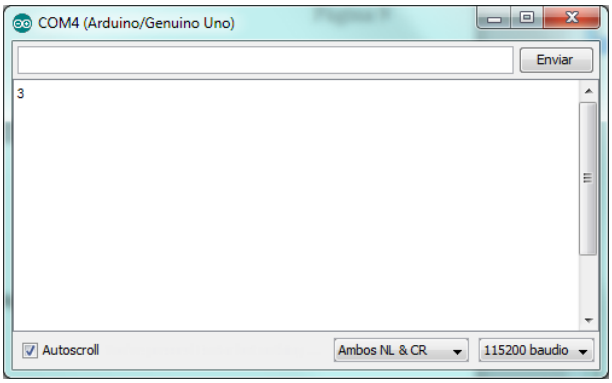
```
void loop()
   {  Serial.println(C1.Num_Objetos());
      Serial.flush();   exit(0);
   }
```

Vamos a tener una sorpresa en la salida:

```
Contador_5.cpp.o: In function ‘Contador::Contador()’:
C:\Program Files (x86)\Arduino/Contador_5.ino:16: undefined reference
to ‘Contador::Num’
Contador_5.cpp.o: In function ‘loop’:
C:\Program Files (x86)\Arduino/Contador_5.ino:38: undefined reference
to ‘Contador::Num’
Contador_5.cpp.o:C:\Program Files (x86)\Arduino/Contador_5.ino:38:
more undefined references to ‘Contador::Num’ follow
collect2.exe: error: ld returned 1 exit status
Error de compilación
```

El motivo está en que Num no ha sido incializado en ningún sitio y para corregirlo podemos hacer:

```
int Contador::Num = 0
```



Hay que tener un poco cuidado cuando definimos una variable static asociada a una clase, ya que hay que asignarla solo una vez, y fuera de las funciones miembros porque de lo contrario podemos encontrarnos con comportamientos extraños.

1.6. Haciendo resumen

Bueno yo creo que para esta primera sesión sobre objetos puede valer ya. He procurado mostrar con el ejemplo más sencillo que se me ha ocurrido, que programar con objetos es conceptualmente distinto del modo procedural, pero que tampoco es para tanto.

En lugar de resolver problemas pensando en funciones, buscamos un modelo a partir de objetos a los que vamos definiendo métodos y propiedades, de una manera muy parecida a como lo haríamos a base de funciones estructuradas.

La peculiaridad es que encapsulamos esas funciones y propiedades en un objeto abstracto que las contiene y aísla del exterior.

Para quienes podéis pensar que es una manera extraña y más trabajosa de hacer lo mismo, me gustaría haceros alguna consideración.

En primer lugar, ciertamente puede haber algo más de trabajo en planificar y diseñar las Clases, cuando el programa a desarrollar es pequeño, pero en cuanto el programa crece la ventaja se invierte, porque defino la clase una vez y la utilizo las veces que requiera.

En un ejemplo en el que el número de instancias de un objeto crezca, la ventaja a favor de la OOP es abismal. Menor código, mejor encapsulado, disminución de errores.

Está también la cuestión de la reutilización del código, que con una clase es automática, mientras que con una colección de funciones hay que andar con tiento.

Los objetos se parece mucho a la forma en como pensamos en nuestro cerebro y eso nos ayuda a desarrollar mejores programas y más seguros.

Para programas muy pequeños quizás no compense, pero a medida que la complejidad crece, es más seguro dedicar un tiempo a esa planificación a la que tan reacios somos los amigos del "Tu dispara y pregunta luego".

2. Polimorfismo y Function Overloading

2.1. Centrando ideas

Con el título que tiene esta sesión, probablemente vamos a leerla 4 gatos, y sin embargo Polimorfismo es un nombre extraño para un concepto muy sencillo que nos resulta natural de entender. Ya hablamos algo de ello en las sesiones anteriores, pero vamos a tocar el tema un poco más en profundidad en esta (Sin miedo que no hay para tanto)

Y es que resulta que lleváis usando aspectos del concepto de Polimorfismo con desenvoltura, poco menos desde los primeros días que empezasteis a

programar vuestros Arduinos.

¿Qué no? Ya lo creo que sí, pero no os habéis dado cuenta porque el concepto es tan natural que ni siquiera solemos percibirlo, salvo haciendo un esfuerzo mental.

Y para que veáis que no os engaño vamos a empezar con algunos casos que deberían haberos disparado todas las alarmas y que sin embargo, os han parecido completamente normales desde el minuto uno.

Quizás empezando así, os daréis cuenta de que aunque no sabías como se llamaba, me creeréis cuando os digo que habéis estado usando el Polimorfismo de un modo natural desde que empezasteis con Arduino C++.

Por ejemplo lleváis mucho tiempo usando la función Serial.println(), que no es nada de sospechosa de veleidades extravagantes y sin embargo tiene un comportamiento sorprendente . ¿No veis nada raro en estas líneas?

```
Serial.println( 5) ;
Serial.println( 3.1416 ) ;
Serial.println(\ Buenos días") ;
```

Insisto, ¿No veis nada sospechoso ahí? Eso es porque estáis tan acostumbrados a ello que no es fácil ver la trampa.

Según lo que hemos aprendido hasta ahora, una función solo puede aceptar un tipo definido de parámetros. ¿Qué demonios es eso, de pasar a una función un int un float o un String según se me ocurra?

¿Si el parámetro es int... Porque me acepta que le pase un float o String? Aquí está pasando algo raro. ¿Por qué nuestro compilador, siempre tan amable el, no nos devuelve un ladrido diciendo que te den?

Lo lleváis haciendo desde siempre pero es imposible, ¿Por qué funciona? ¿Serias capaz de programar una función así? ¿A que no?

Y eso, queridos amigos, en una función que habéis estado usando hasta hartaros sin pensar ni por un momento que era imposible (¿Creías que os engañaba?)

El misterio está precisamente en una característica inherente a C++ y que no existía en C, y no es otra que una característica llamada function overloading.

2.2. Function Overloading

Ahora que he conseguido tu atención, podemos empezar a hablar en serio del Polimorfismo y de porque los println() anteriores funcionan, aunque todo indica que no deberían, porque va en contra de todo lo que hemos aprendido hasta ahora de las funciones.

Y el misterio está en que no existe una única función println(), sino que las líneas anteriores invocan 3 funciones completamente diferentes... que se llaman igual.

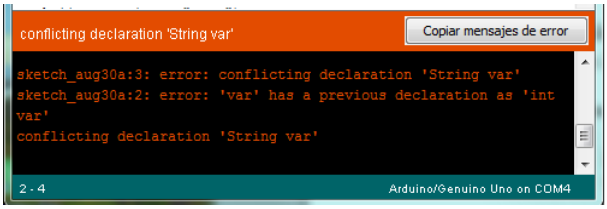
¡ Queee ¡ ¡Venga ya!

Normalmente aquí aparecen frases del tipo: "Todo el mundo sabe que dos funciones distintas no pueden llamarse igual, lo mismo que dos variables diferentes no pueden tener el mismo nombre".

Veamos. Si intento algo así:

```
int var = 0 ;
String var = "Buenos dias" ;
```

El compilador enseguida me pone firme y parece estar de acuerdo con la idea general:



Pero hagamos un intento diferente. Imaginaros una función llamada Duplica() que si le paso un int me devuelve el doble claro, y lo mismo se le paso un float. Pero imagínate que quiero que si le paso un String me devuelva otro String con la cadena inicial duplicada, ¿Parece natural, No? Fíjate que hasta en la redacción de este párrafo no hago diferencia entre ambas ideas.

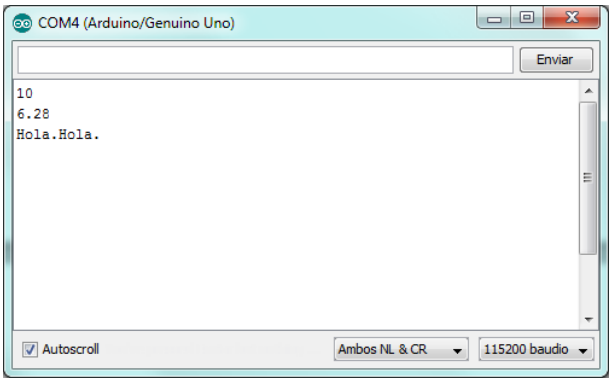
Pero... ¿Y el compilador que va a decir? Veamos, intentemos definir tres funciones así:

```
int Duplica( int j)
     { return (2 * j) ; }

float Duplica ( float n)
     {   return( 2 * n) ;   }

String Duplica( String s)
     {  return ( s + s) ;   }
```

Y luego podemos intentar esto:

```
Serial.println(Duplica(5));
  Serial.println(Duplica(3.1416 )) ;
  Serial.println(Duplica("Hola."));
```

Lo lógico es que el compilador diga que ni de coña se traga esto. "Las funciones tienen que llamarse distinto y punto". Pero resulta que no, ya ves. Siempre consiguen sorprendernos:



Resulta que el compilador de C++ (Que no el de C) acepta que diferentes funciones tengan el mismo nombre, a condición inexcusable de que él pueda diferenciarlas implícitamente por el numero o tipo de parámetros que requiere cada una, lo que suele llamarse firma. ¿Qué te parece?

A esta capacidad de definir varias funciones diferentes con el mimsos nombre, se le llama Function Overloading o sobrecarga de funcionesy de cada una de las funciones de igual nombre decimos que están sobrecargadas (Overloaded).

A pesar de tan extravagante comportamiento y de algo que nos parece tan extraño al primer bote, llevas mucho tiempo usándolo y te parece tan normal, porque nuestro cerebro abstrae los conceptos mayores y le parece normal que el ordenador haga esto (Aunque nos suele dejar flasheados descubrirlo)

De hecho la sobrecarga (Overloadding) de funciones es una operación tan intuitiva que nos permite desarrollar programas mucho más sencillos y menos proclives a error.

```
De no existir el Overloading, la función println() necesitaría al menos
3 funciones en sus lugar: Una para enteros, otra para float, otra apara Strings.
Pero recuerda que también hay Bytes, Uints, longs, doubles y .....
```

En cuanto nos recuperemos de la impresión sufrida, empezaremos a preguntarnos que si se puede hacer Overloading de funciones ... ¿Hay más cosas con las que se pueda hacer?

Y la respuesta es que si, y os habéis hartado a usarlo sin daros cuenta tampoco. ¿Adiviná is que puede ser? Os doy una pista: En el último programa usamos el Overloading de algo más que las funciones, pero de esto hablaremos en la próxima sesión.

De momento quiero volver a la clase Contador que definimos en la sesión previa, para darle más vueltas.

2.3. Jugando con la Clase Contador

En nuestra última sesión estuvimos jugando con una pequeña clase ejemplo que llamamos Contador. La definimos así:

```
class Contador
  {   private:
         int N ;

      public:
         Contador( ) ;                // Constructor
         void SetContador( int n) ;
         void Incrementar() ;
         int GetCont() ;
  } ;
```

Y luego definimos sus funciones miembros o Métodos.

```
Contador::Contador( )                // Constructor
             { N = 0 ; }

      void Contador::SetContador( int n)
             {  N = n ;    }

      void Contador::Incrementar()
             {  N++ ; }

      int Contador::GetCont()
             { return (N) ;}
```

Bien, a lo nuestro. No está mal para ser nuestra primera Clase, pero es manifiestamente mejorable. Por ejemplo, todos nuestros contadores se ponen a cero mediante el Constructor, lo que ha sido una mejora con respecto a tener que inicializarlo a mano, pero... ¿Qué hago si necesito un contador que empiece en digamos 129 o cualquier otro valor, claro?

Puedo usar el método SetContador(), pero nuestros amigos nos mirarán con desprecio por usar semejante solución, así que hay que discurrir algo más.

La solución elegante y que hará suspirar a los freakys de tus colegas es hacer un Overloading del Constructor, que lo acepta sin rechistar como cualquier otra función.

```
class Contador
   {  private:
         int N ;

      public:
         Contador( ) ;                // Constructor
         Contador( int k ) ;          // Constructor
         void SetContador( int n) ;
         void Incrementar() ;
         int GetCont() ;
   } ;
```

Y las funciones miembros podrían ser así:

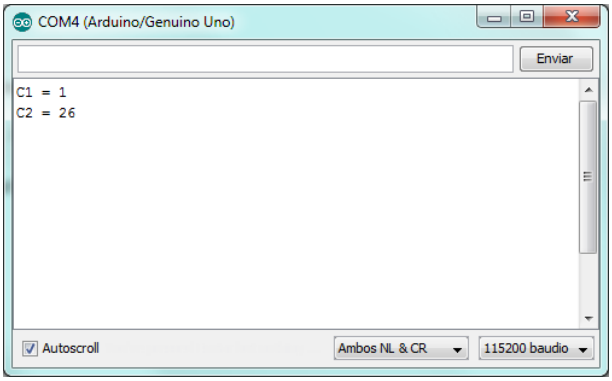
```
Contador::Contador( )              // Constructor
           { N = 0 ; }

      Contador::Contador( int k)         // Constructor
           { N = k ; }

      void Contador::SetContador( int n)
           {  N = n ;    }

      void Contador::Incrementar()
           {  N++ ; }

      int Contador::GetCont()
           { return (N) ;}
```

Hemos hecho un Overloading del Constructor de la Clase, Que dicho así suena muy raro, pero que traducido significa, que podemos declarar dos Constructores diferentes siempre y cuando le pasemos diferente firma parámetros (En numero o tipo). Si hacemos dos constructores, uno sin parametros y otro que acepte un int:

```
Contador C1, C2(23) ;
```

```
void loop()
{
     C1.Incrementar() ;
     Serial.print("C1 = "); Serial.println(C1.GetCont());

     C2.Incrementar() ; C2.Incrementar() ; C2.Incrementar() ;
     Serial.print("C2 = "); Serial.println(C2.GetCont());
}
```

Si no lleva parámetros ponemos a cero el contador interno, pero si recibe un parámetro hacemos que este sea el valor inicial del contador. ¿Qué fácil y hasta natural, No? Casi oigo como os crujen las neuronas. Las ideas involu-



cradas son sencillas una a una, pero al ir construyendo una idea sobre otra, puede haber que dar un paso atrás para coger perspectiva (Y aire).

Recapitulemos.

- Definimos una clase llamado Contador que nos permite llevar la cuenta de lo que se nos ocurra.
- Pero no queremos tener que inicializar el contador cada vez que instanciamos un nuevo Objeto (Forma pija de decir que creamos un contador)
- Para evitarlo, definimos un Constructor, que se invoca siempre que creamos un Objeto del tipo Contador, poniendo el contador a 0.
- Pero esto, aunque no está mal, no mola porque si quiero cambiar el valor del contador tengo que invocar un método, y como somos vagos, no queremos aprender tonterías, y preferimos evitarlo.

- Para ello Hacemos un Constructor Overloading, o segundo constructor de modo que pueda aceptar un parámetro al instanciar el contador y poner a ese valor el contador interno.
- De ese modo si instanciamos el objeto sin parámetro, el contador arranca desde cero, pero si le pasamos un parámetro, inicia el contador desde ahí.
- mpresionamos a los colegas fijo (De ligar nada, no sirve para eso)

La potencia que este tipo de unión entre las Clases y el Overloading nos proporciona es impresionante, no tanto para quedarnos con los colegas, sino para hacer programas más sencillos y comprensibles.

En lugar de usar varias funciones que puedan hacer algo que para nosotros es lo mismo, nos basta con recordar una. Casi parece lo normal

Vale, esto va cogiendo buena pinta, pero vaya asco de contador que hemos hecho. Solo se incrementa. ¿Y si a mí me apetece que decremente porque voy a hacer una cuenta atrás, que?

Además C++ siempre ha tenido esa chulada del ++ o el – para variar el valor de una variable. ¿Podría hacer lo mismo con un objeto?

O más aún, Si tengo dos contadores ¿Puedo sumarlos y obtener un contador con la suma neta de los dos contadores? ¿Y podría restarlos?

Creo que ya adiviná is la respuesta. Desde luego que sí, mediante un Operator Overloading en lugar de un Function Overloading.

Pero esto, queridos amigos, será el tema de la próxima sesión que por hoy ya nos hemos complicado suficiente y conviene descansar el cerebro.

2.4. Algo más sobre el Polimorfismo

El function Overloading es un aspecto del Polimorfismo que nos permite manejar diferentes objetos con los mismos métodos o propiedades.

Los lectores avispados se habrán percatado que he evitado referirme directamente al Polimorfismo per se, porque entraríamos en aguas pantanosas rápidamente y no es el momento, ni probablemente sea yo el más indicado para esa discusión.

He preferido evitar el rigor conceptual en beneficio de una aproximación simple, presentando algunas ventajas más tangibles como el concepto de Funcion y Operator Overloading (Que veremos en la próxima sesión) y esquivar el tema central, porque requeriría otros conceptos adicionales que no hemos visto como la Herencia simple y múltiple, o las funciones virtuales, que serían complicadas de encajar con garantías en esta primera aproximación.

Por eso me contentaré con decir aquí simplemente, que el Polimorfismo es una cualidad abstracta de los objetos que nos permite usar un interface único, de métodos y propiedades, en una colección de objetos de distintos tipos o Clases.

Recordad el ejemplo que comentamos en alguna sesión previa, que existe un concepto abstracto llamado arrancar que nos resulta natural, para un motor eléctrico, de gasolina o de diésel.

En la forma en que nuestro cerebro procesa el mundo, las tres objetos comparten ese método común, y para nosotros es de lo más natural considerarlos iguales, por más que comprendemos muy bien que el procedimiento físico que arranca esos tres motores es completamente diferente.

Polimorfismo es un concepto abstracto que representa precisamente esa capacidad de modelizar diferentes sistemas físicos u Objetos, mediante métodos y propiedades comunes, en un concepto abstracto (Y jerárquicamente superior) de motor que comparten métodos como Arrancar, Frenar o Acelerar y propiedades como Potencia o Velocidad.

La Clase Motor en abstracto, es independiente de la tecnología que se emplea en un caso concreto y sigue siendo válida cuando se desarrollen otros tipos de motores en el futuro.

Si queréis profundizar en el tema, no tendréis dificultad en hallar documentación en Internet, pero os recomiendo que si esta es vuestro primera aproximación a la OOP, evitéis hacerlo hasta que hayáis asentado e interiorizado bastante más el asunto.

2.5. Resumen de la sesión

- Confiamos en que el Polimorfismo y Overloading parezcan un poco menos amenazantes ahora que sabeís lo que son.
- Vimos como una sobrecarga de funciones como el Constructor, nos ayuda a escribir programas más sencillos de comprender y usar.
- Hemos programado algún ejemplo de Function Overloading y parece que no era para tanto.

3. Unary Operator Overloading

3.1. Operator Overloading

En la última sesión estabamos construyendo una Clase, Contador, que nos sirviera como ejemplo de lo que podemos hacer. Vimos cómo definir la

sintaxis y sobre todo nos centramos en el Function Overloading, ya que nos daba una ventaja importante de cara a usar un nombre único de función, para varis cosas que en principio serian diferentes.

La ventaja de esto es que resulta mucho más fácil de recordar y más sencillo de utilizar porque encaja bien con nuestra forma de procesar las ideas.

Pero una vez que abrimos la caja de Pandora con el Overloading, resulta muy complicado cerrarla, porque en cuanto te acostumbras a la idea, empiezas a hacerte muchas preguntas raras, del tipo de ¿Y qué más puedo sobrecargar? Y aquí es cuando la cosa se lía.

Porque no solo se pueden sobrecargar las funciones, sino también los operadores para que hagan cosas diferentes en función del tipo de los operadores. No creo que tenga que insistir mucho para que me creáis si os digo que la suma de dos enteros no se parece (A nivel de procedimiento) a la de dos float, y lo mismo pasa con +, -, * y / por poner un caso.

Los operadores invocan distintos procedimientos en función del tipo de los operandos, y nunca es más evidente que cuando hacemos:

```
String Duplica( String s)
  {    return ( s + s) ;  }
```

En donde el símbolo de la suma significa concatenar dos Strings. Todos estos operadores están sobrecargados por C++, para que podamos usarlos sin pensar en ello, y que se comporten como parece que es lo normal. (Pero no podemos por ejemplo hacer s1-s2, porque ¿Qué sentido tendría?)

De hecho, cuando definimos una nueva Clase, lo que estamos haciendo es crear un nuevo tipo de datos, tipo en el sentido de int, long, etc. y dentro de cada clase podemos hacer el Overloading de los operadores que nos interesen, para indicarle al compilador, como debe ejecutarse la operación que representa el símbolo del operador.

Por eso vamos a dedicar esta sesión a ver la forma y el modo de realizar el Operator Overloading, pero os prevengo, sentaros cómodos y a ser posible relajados, porque el tema hay que irlo dosificando sin prisa.

Pero antes me gustaría hablaros de los operadores unarios y binarios. En C++, se consideran dos grandes familias de operadores, los que se aplican a un solo elemento (Unary Operator u Operador unitario) y los que se aplican a 2 elementos (O Binary Operator, Operador Binario).

- También existe un operador terciario que no nos conviene mencionar en este momento.

En la primera categoría, Unary Operators, están los operadores de incrementar y decrementar ++ y –, tanto en su versión prefijo como sufijo (++i, i++) y además la negación y el símbolo negativo – cuando se aplica a un número para cambiarle el signo. En la categoría de Binary Operators tenemos +, -, *, /,

Esto es importante porque vamos a empezar viendo como se hace el Operator Overload de los Unary Operators (*No corráis cobardes*).

3.2. Unary Operator Overload

Volvamos a nuestra flamante nueva Clase de Contador, para usarla como base. Podemos reescribirla así:

```
class Contador
   { private:
         int N ;

     public:
         Contador( ) : N(0) {}                    // Constructor
         Contador(int k ) : N(k) {}               // Constructor
         void SetContador( int n) ;
         void Incrementar() ;
         int GetCont() ;
   } ;

void Contador::SetContador( int n)
     {  N = n ;    }
void Contador::Incrementar()
     {  N++ ; }
int Contador::GetCont()
     { return (N) ;}
```

Hemos reescrito los constructores para tener una notación más compacta. Bien no está mal. Podemos inicializar los objetos de Contador, con y sin parámetro, lo que es un avance y nos permite escribir tal y como veíamos en la última sesión algo así:

```
Contador C1, C2(23) ;
```

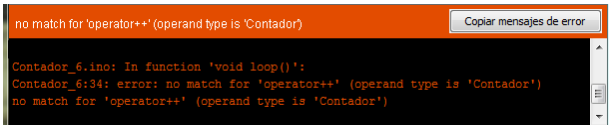
Lo que resulta bastante fácil de leer, y cómodo de usar, pero ya que estamos (Ay Dios) nos preguntamos si se podrían hacer algunas cosas normales en C++ como esto:

```
++C2 ;
```

En lugar de nuestra forma actual:

```
C2.Incrementar() ;
```

Que es como un poco raro de leer. ¿Sería posible? Intentadlo y veréis lo que dice el compilador. Recordad que dijimos que crear una Clase es como crear



un nuevo tipo de datos. El compilador sabe cómo aplicar el operador ++ a un int, pero no tiene ni idea de cómo usarlo con un Contador... salvo que se lo expliquemos claramente, con un Operator Overload.

La cosa está chupada. Para ello basta con redefinir el operador ++ para nuestra clase mediante la instrucción operator y nuestra clase quedaría:

```
class Contador
   {  private:
         int N ;

      public:
         Contador( ) : N(0) {}                    // Constructor
         Contador(int k ) : N(k) {}               // Constructor
         void SetContador( int n) ;
         int GetCont() ;
         void operator ++ ();                         // Aqui esta ++
   } ;

void Contador::SetContador( int n)  {  N = n ;    }
int  Contador::GetCont() { return (N) ;}
void Contador::operator ++ ()                         //  <---
     {  ++N }
```

En la que podéis ver que la línea clave es :

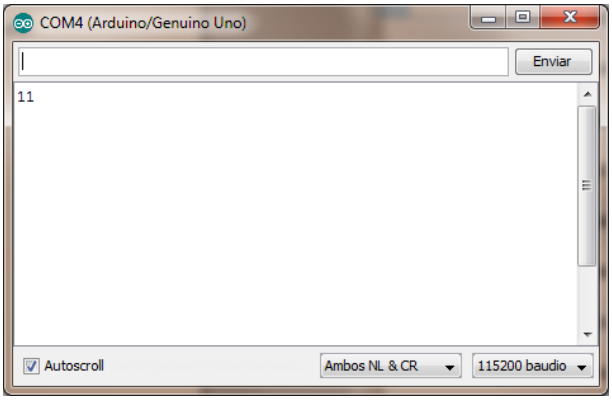
```
void operator ++ ();
```

Usamos la keyword "operator", para identificar el operador a definir y la definimos como void porque no devolvemos nada, simplemente incrementamos su valor.

Después hemos definido la función que el operador ++ aplicará y de paso eliminamos la función Incrementar() que aunque útil, era un asco de usar. Si ahora hacemos esto:

```
Contador C1(10)  ;
++C1 ;
Serial.println(C1.GetCont());
```

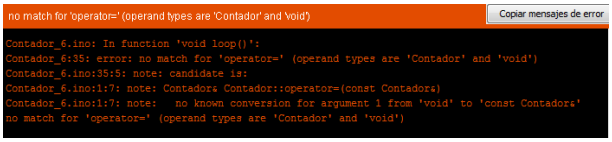
Obtendremos un bonito resultado de 11, como queríamos conseguir. ¿Y



podríamos hacer esto?

```
Contador C1 , C3(10) ;
C1 = ++C3 ;
```

Para nada, ¿ Porque? Piensalo un momento antes de seguir. Pues porque he-



mos definido como void el resultado del operador ++ y no podemos hacer que el resultado void, se asigne a un objeto de la Clase Contador, y naturalmente el compilador se pone atacado en cuanto lo ve.

Para resolver eso, vamos a necesitar que lo que devuelva el operador ++, sea un objeto de la Clase Contador, y para ello tenemos que definir la función así:

```
Contador Contador::operator ++()
     { return Contador (++N);  }
```

Y ahora si que es posible hacer:

```
Contador C1, C3(10) ;
C1 = ++C3 ;
Serial.println(C1.GetCont());
```

Que aunque lo hemos hecho con mucha facilidad, conviene fijarse en un par de cosas:

- De Una función puede devolver un objeto tranquilamente. Algo que hasta ahora no habíamos planteado pero que es bastante frecuente. En este caso es un objeto del tipo Contador.
- En este caso el objeto que devolvemos es un objeto temporal que ni siquiera tiene nombre y que se calcula sobre la marcha, para devolverlo a quien invoque el operador ++.En este caso se asigna a C1 y el Objeto temporal se desvanece sin más, sin haber llegado siquiera a bautizarlo.
- Para que este método que hemos usado funcione necesitamos haber hecho un Constructor Overloading que nos permita crear un objeto tipo y pasarle el valor que deseamos al crearlo.

Vale, es un buen momento para tomar aire y volver a leer despacio lo de arriba, porque aunque la operación es sencilla y parece sencilla tiene un fondo importante, y de nuevo, muchos conceptos mezclados.

3.3. Postfix Unary Operator Overload

Parece que estamos haciendo un concurso de títulos raros, pero las cosas son mas o menos así.

De acuerdo, hemos hecho un Overloading del Prefix Operator, es decir, que podemos escribir ++C1 (Con el operador en modo prefijo) pero si intentáis hacerlo con el modo postfix, o sufijo: C1++, recibiréis un simpático corte de mangas del compilador, porque la sintaxis anterior describe el modo prefix pero no el suffix.

Para definir el operador suffix necesitamos usar una sintaxis un tanto extraña, pero indolora:

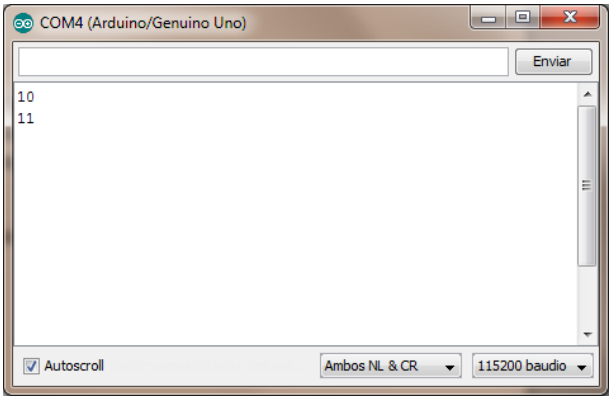
```
Contador Contador::operator ++ (int)
   { return Contador (N++);  }
```

Donde el int que le pasamos entre paréntesis solo significa que se refiere al postfix Operator. Es raro pero vete acostumbrando, C++ es así de maniático, y no tiene otro significado.

Ahora podemos hacer un nuevo programa

```
Contador C1, C3(10) ;
C1 = C3++ ;
Serial.println(C1.GetCont());
Serial.println(C3.GetCont());
```

El resultado es el que cabía esperar:



- Recordad que ++i, en prefix significa, primero incrementa y después usa el valor de i, mientras que i++, en postfix significa, que primero entregas el valor de i, y una vez que ha operado, incrementalo.

3.4. El Operador this

La solución que dimos en los últimos ejemplos, de devolver un Objeto temporal que debe ser primero creado y después destruido, funciona, (Lo que no es poco), pero tiene el inconveniente de que puede ser lento y consumir una memoria de la que rara vez estamos sobrados.

Así, que no se considera elegante, y menos para un procedimiento como devolver un Objeto, que es algo muy frecuente, y más si tenemos en cuenta, que en realidad, ya tenemos un Objeto del tipo Contador dispuesto y con el valor que queremos: C1, ¿Por qué no devolverlo directamente?

Por eso los señores que diseñan los compiladores C++ nos ofrecen una solución mucho más elegante: el operador "this".

El operador "this" es un puntero que se pasa a disposición de todas las funciones miembro de la clase, (Y eso incluye a todas los funciones de operadores sobrecargados), que apunta al objeto al que pertenecen.

- Y por eso, el compilador no necesitaba hacer copias de las funciones para todos los objetos de una clase, basta con aplicarlas apuntando a la dirección contenida en this.

Cuando instanciamos C1, cualquier función miembro que reclame el operador this, recibe un puntero a la dirección de memoria que almacena sus datos, que por definición es una la dirección del objeto C1 (No de la definición de la clase).

Si recordáis como trabajábamos con punteros, podremos escribir la función de Overloading del operador ++, de este modo (Coged aire):

```
const Contador &Contador::operator ++()
    { ++N;
      return *this ;
    }
```

Antes de nadie salga corriendo, dejad que me explique.

- Definimos la función operator ++ como tipo Contador porque va a devolver un objeto de este tipo. (Esta parte ya estaba dominada, recordad)
- La particularidad está en que avisamos al compilador con el símbolo &, de que lo que vamos a devolver es un puntero a un objeto de la clase Contador, y no un objeto.
- Tras incrementar N, ya hemos realizado la operación que buscábamos y el objeto presente, por ejemplo C1, ya tiene el valor adecuado.
- Y ahora devolvemos el puntero a nuestra propia instancia del Objeto con la referencia que indica el operador this y de ese modo nos ahorramos el trasiego de crear y eliminar objetos temporales.
- Lo de especificar la función como const, es para evitar que al pasar la referencia de nuestro objeto actual, haya posibilidad de modificarlo por error. No os olvidéis de esto por si las moscas.

Como es habitual, en cuanto se mentan los punteros, los jadeos de angustia se escuchan agónicos. Pero en serio, no os preocupéis, si ahora os resulta duro, es normal, las cosas tienen que asentarse y encontrar su sitio, así que no os agobiéis que requiere su tiempo.

Además bueno es C++ para estas cosas, pero recordad que si el tema os marea siempre podéis usar un objeto temporal que es mucho más sencillo de comprender y sino queréis nota sobra.

En algún momento tendremos que dedicar una sesión (O varias) a las cuestiones de punteros en profundidad, porque es algo que concede a C++ una potencia sin precedentes, pero por ahora es pronto y hay que ir poco a poco, que no quiero asustar a nadie.

3.5. Resumen de la sesión

- Vimos la diferencia entre operadores unitarios y binarios.
- Aprendimos a sobrecragar los operadores unitarios, tanto en prefijo como en sufijo.
- Vimos que podemos devolver objetos como retorno de una función.
- Presentamos el nuevo operador this.

4. Binary Operator Overload y conversión de tipos

4.1. Overloading Binary Operators

En la sesión previa vimos que eran y como hacer la sobrecarga de operadores unitarios, aquellos que se aplicaban a un único operando, como el incrementar y decrementar, por ejemplo. En esta sesión vamos a ver cómo hacemos para programar la sobrecarga de operadores binarios, aquello que como la suma +, o el producto *, involucran dos operadores para producir un resultado.

Vamos a partir de la Clase Contador, que llevamos mareando un tiempo, porque es una clase lo bastante sencilla para evitar complicaciones innecesarias y porque además es un ejemplo excelente, por lo simple, de lo que se puede hacer con una Clase, sin que el programa de ejemplo se estire hasta un punto en el que sea difícil de seguir la idea.

La historia hasta aquí: La definición de la Clase Contador:

```
class Contador
  {   private:
         int N ;

      public:
         Contador( ) : N(0) {}                    // Constructor
         Contador(int k ) : N(k) {}               // Constructor
         void SetContador( int n) ;
         int GetCont() ;
         const Contador &operator ++ ();
  } ;

void Contador::SetContador( int n)
```

```
        {  N = n ;    }

int Contador::GetCont()
        { return (N) ;}

const Contador &Contador::operator ++()   // Prefix Operator
         { ++N;
           return *this ;
         }
Contador Contador::operator ++ (int)      // Postfix Operator
         { return Contador (N++);  }
```

a disponemos en la clase, de dos Constructores diferentes, uno para inicializar a 0 se crea sin parámetros y otro que inicializa el contador interno en el caso de que se le suministre un valor.

Pero imagínate que queremos sumar contadores. ¿Por qué no? Podemos sumar los registros internos de dos contadores, de modo que el resultado sea otro contador con un valor de N interno igual a la suma de los dos operandos.

No sería complicado definir una función miembro, que podemos llamar Suma, que devuelva un objeto Contador tras operar con dos contadores. Podríamos hacer algo así:

```
Contador  Contador::Suma( const Contador & C1 )
      {   return Contador  ( N + C1.GetCont() ) ; }
```

Que resulta muy fácil de escribir, pero un poco más pesado de digerir. La función miembro Suma toma una referencia a un Contador C1 genérico, (Forzada a const para evitar sustos) y devuelve un objeto tipo Contador con el resultado de sumar ambos valores internos (Sé que en cuanto hay unareferencia la cosa se complica y se escuchan gemidos).

- Para que esto funcione, tenemos que disponer de un Constructor que acepte crear un objeto de este tipo mediante la operación de la segunda línea, pero esto ya lo teníamos definido.

Aunque este método funcionará, su uso es más bien repelente :

```
Contador C1(), C2(23) ;
Contador C3 = C1.Suma(C2) ;
```

No tiene ningún problema, es simplemente que recordar esto es un asco y fácil de olvidar si la función era suma, Suma, Add o ....

Aquí somos gente elegante y lo que queremos hacer es algo más intuitivo y que no se nos olvidará nunca. Queremos escribir la suma así, como corresponde a cualquier programador que se precie:

```
Contador C3 = C1 + C2 ;
```

Mucho más presentable y elegante, ¿No? Y para eso está l**a sobrecarga de operadores binarios**, como el +. Es tan fácil de hacer como cualquiera de los ejemplos previos, y sería algo así mediante el uso de la clave operator como ya vimos:

```
class Contador
  {   private:
         int N ;

      public:
         Contador( ) : N(0) {}                  // Constructor
         Contador(int k ) : N(k) {}             // Constructor
         void SetContador( int n) ;
         int GetCont() ;
         const Contador &operator ++ ();
         Contador operator ++ (int) ;
         Contador operator + (  Contador &) ;  // Pasamos una referencia a un co
  } ;
```

Obviando las funciones que ya tenemos claras la novedad seria:

```
Contador  Contador::operator+ (  Contador & C1 )
           { return Contador  (  N + C1.GetCont() ) ; }
```

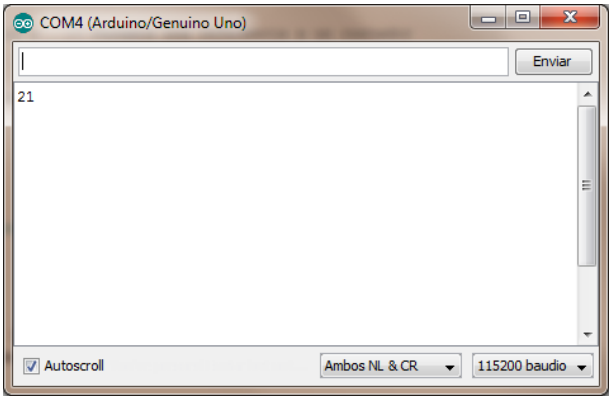
Ahora podemos hacer esto tranquilamente:

```
void loop()
   {    Contador C1, C2(10), C3(11) ;
        C1 = C2 + C3 ;

        Serial.println(C1.GetCont());
        Serial.flush();   exit(0);
   }
```

Con este resultado:

4.2. Comparator Overload

Podemos sobrecargar más operadores, por ejemplo el operador '¿' para comparar dos Contadores y realizar alguna operación en consecuencia. No parece descabellado hacer una comparación parecida a esto:




```
Contador C1, C2(12) ;
if ( C1 > C2 )
     Serial.println( \Mayor");
else
     Serial.println( \Menor");
```

Si lo intentamos por las buenas C++ estará encantado de darnos un corte de mangas e informarnos que no tiene ni pastelera idea de cómo usar este operador con objetos tipo Contador. Pero podemos especificárselo así:

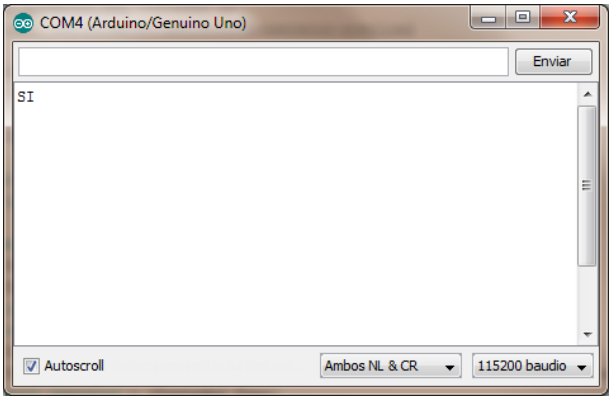
```
bool operator > (Contador Cont)
   {    if ( N > Cont.N)
             return true ;
        else
             return false ;
   }
```

Y ahora sí, nos permitirá hacer :

```
Contador C1 = 12, C2 = 6  ;
if (C1 > C2 )
    Serial.println("SI");
else
    Serial.println("NO");
```

Aquí tenéis el resultado:

Este es un momento tan bueno como cualquier otro para indicar que además de operadores unitarios y binarios existe un operador terciario (¿O trinario?) y para ilustrar su ejemplo podemos escribir la función anterior así:




```
bool operator > (Contador Cont)
   { return  ( N > Cont.N ) ? true : false ;  }
```

Para ser franco, aunque es un modo compacto de escribir condicionales, no os lo recomiendo porque va a despistar mucho a más de uno que tenga que leer el código, pero por si acaso os hago la anotación porque encontrareis, que quienes escriben las clases parecen tener un gusto especial por escribir cosas de este modo.

4.3. Consideraciones sobre la sobrecarga de operadores

No pretendemos en esta humilde introducción a la Programación Orientada a Objetos con C++, hacer un repaso exhaustivo de todos los operadores que se pueden sobrecargar, sino simplemente ver una primera aproximación con algunos ejemplos sencillos.

Por ejemplo, no sería complicado definir el operador – para hacer la resta de contadores y si os parece os lo dejo como ejercicio, pero solo aspiramos a daros una primera visión de porque es interesante hacer Operator y función Overloading en vuestros programas.

Aunque ahora os pueda parecer increíble, cuando empecéis a jugar con Clases, veréis que la sobrecarga viene sola a vuestros programas a poco que entendáis la idea, y es algo que simplifica mucho la comprensión de los programas.

De hecho el problema nos es que no la uséis, sino que los nuevos tienden a usarla en exceso y para cosa que no se deberían hacer.

Por ejemplo sería muy fácil redefinir el operator -, para hacer sumas pero naturalmente sería estúpido y exasperarías a quien fuera a usarlo. Mantened

la sobrecarga de operadores en un límite sensato y aplica la regla de que si dudas de si algo es sensato, entonces seguro que no lo es.

Cuando programes ten piedad de quien tenga que leer tu código (Que probablemente serás tú además) y utiliza la sobrecarga para hacer los programas más fáciles de leer, no para impresionar a tus colegas. Se trata de evitar leerse el manual y que la lectura del código se comprenda de modo natural.

En el ejemplo de suma que hemos hecho arriba a cualquiera sin leer un manual se le puede ocurrir intentar la suma del modo que lo hemos hecho.

4.4. Conversión de tipos de datos

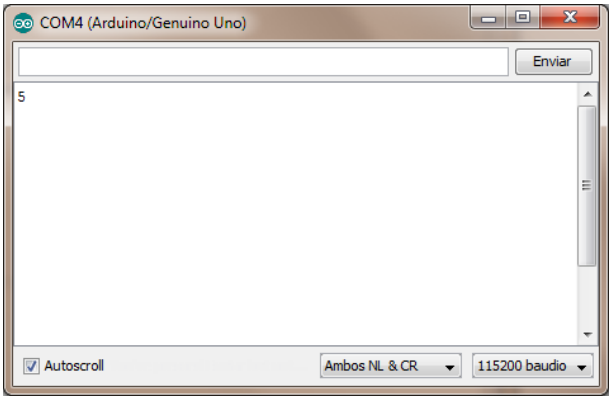
Hay un tema muy relacionado con los operator overloading que tiene que ver con la conversión de tipos en nuestros programas. Veamos porque.

Con lo visto hasta ahora podemos crear un objeto tipo contador directamente con o sin parámetro inicial, pero se nos puede ocurrir con facilidad preguntarnos si podríamos hacer algo como esto:

```
Contador C1 = 5 ;
```

Al primer bote, parece razonable. Nos ahorraríamos la función SetContador() que aunque útil, es poco practica por las mismas razones que dimos arriba, hay que sabérsela y no es evidente, pero la línea de aquí arriba seria de lo más fácil de entender, y nos acordaríamos seguro.

Pero... ¿Funcionaría? Después de todo, estamos forzando una conversión de un tipo int a un tipo Contador, y no hemos especificado como hacer esta conversión. Si lo intentáis obtendremos esto:



C++ nos vuelve a sorprender con una conversión automática y un buen ejemplo de cómo intenta entendernos, ¿Pero cómo ha sabido hacer la conversión de tipo?

La razón estriba en que ha supuesto que queremos convertir un int a Contador y es lo bastante astuto para buscar un constructor que requiera un único int como argumento, y lo ha encontrado. Por eso ha sabido cómo hacer una conversión de tipo automática.

O dicho de otro modo ha supuesto, mediante la firma de la función sobrecargada lo que pretendíamos, y lo ha aceptado sin rechistar, pero tened cuidado con las conversiones automáticas porque pueden daros más de una sorpresa.

Y para mayor sorpresa aun, esto otro también va a funcionar (por increíble que parezca) con un float:

```
Contador C1 = 5.7F ;
```

¿Pero cómo es posible? No existe un constructor a partir de un float.

Pues porque cuando C++ vea que le pasmaos un float, hará una conversión automática de float a int (Incluida en las conversiones automáticas de tipos) cortando la parte decimal y asignándoselo al Constructor con int de la Clase, y esto no es algo que pudieramos esperar, pero la vida está llena de sorpresas, ya ves.

Por tanto y en contra de todo pronóstico (Y con más de un ataque de nervios de cualquier teórico de los lenguajes de programación) la función:

```
void SetContador( int n) ;
```

Puede tranquilamente ser borrada sin consecuencias de nuestras funciones miembros, ya que el constructor mediante int es capaz de hacer su labor de modo automático. (Si querían elegancia aquí hay taza y media)

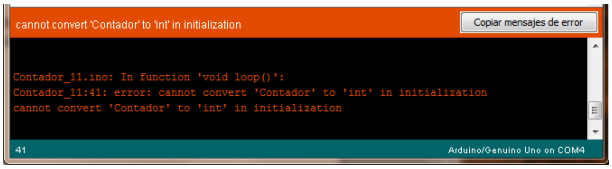
Y es que quien va a querer recordar el nombre de una función, cuando un sencillo símbolo de asignación es suficiente , es lo primero que cualquiera probaríamos para asignar valor al contador.

¿Y al revés funcionaria? Me refiero a si puedo hacer:

```
Contador C1 = 12 ;
int i = C1 ;
```

El compilador se pone atacado inmediatamente y nos dice que no tiene ni idea de cómo convertir un objeto tipo Contador en un int, lo que no resulta sorprendente.

Pero C++ dispone de un mecanismo para definirle esta conversión, y que en un caso tan sencillo como nuestra clase Contador, parece que lo lógico sería asignar al int el valor interno del contador.



- Mucho cuidado con estas conversiones que no siempre son tan evidentes y la cosa se puede complicar, pero aquí estamos para presentaros un ejemplo sencillo de algo que os puede hacer falta.

Para especificarle al compilador que queremos asignar a un int el valor contenido en N, podemos usar la instrucción :

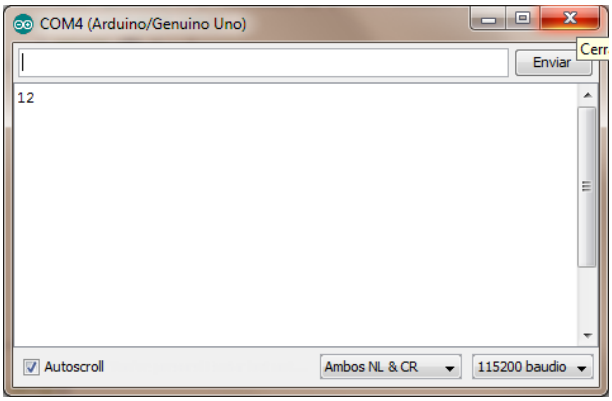
```
operator unsigned int()
    { return (N) ;   }
```

Donde os conviene fijaros en que la sintaxis es un poco extraña. Las conversiones de tipos no se declaran con tipo de retorno, aunque devuelva uno (A mí no me miréis)

Y ahora nuestro programa ha definido al compilador como convertimos de int a Contador (Mediante un Constructor) y como convertir a int un Contador, ¿Qué os parece?

```
Contador C1 = 12 ;
int i = C1 ;

Serial.println(i);
```

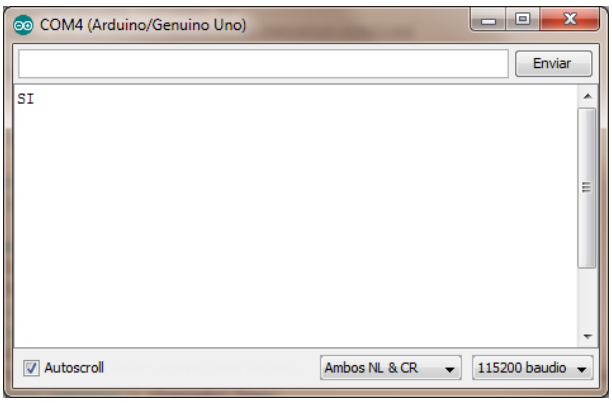
La salida es esta: No puedo resistirme aquí a una pequeña maldad, que con-



siste en que probéis algo aparentemente inocente. Con este mismo programa haced lo siguiente:

```
Contador C1 = 12, C2 = 6  ;
if (C1 > C2 )
    Serial.println("SI");
else
    Serial.println("NO");
```

Tal y como vimos en algún programa anterior la respuesta es, tal y como esperábamos: Nada de particular en esto ¿Noo? El problema radica en que en



este programa no hemos definido el operador ¿por ningún sitio. Comprobadlo sino me creéis:

```
class Contador
   {  private:
         int N ;

      public:
         Contador( ) : N(0) {}                    // Constructor
         Contador(int k ) : N(k) {}               // Constructor
         int GetCont() ;
         const Contador &operator ++ ();
         Contador operator ++ (int) ;
         Contador operator + (  Contador &) ;   // Pasamos una referencia a un c
         operator unsigned int()
             { return (N) ;   }
   } ;
```

¿Pero entonces cómo es posible que funcione?

De nuevo la respuesta está en las suposiciones con las que C++ trata de ayudarnos que van normalmente en la buena dirección pero que os pueden jugar malas pasadas si no estáis sobre aviso.

Como C++ sabe cómo convertir nuestros Contadores a una variable tipo int que conoce bien, ha supuesto que podría hacer la comparación así, en tanto en cuanto no le declaremos específicamente otra forma de hacer la conversión.

- Naturalmente, si le quitas la conversión de Contador a int, la comparación sobreentendida dejara de funcionar.

Así pues tened un poco de cuidado hasta que os acostumbréis, porque algunas de estas suposiciones pueden acabar siendo un tiro en el pie.

4.5. Resumen de la sesión

- Vimos cómo sobrecargar los operadores binarios.
- Hicimos un par de ejemplos con + y ¿.
- Presentamos las conversiones automáticas de tipos.
- Vimos como declarar conversiones de tipos en nuestras propias clases.

5. La Herencia en C++

5.1. La Herencia en C++

Hemos ido viendo en las sesiones previas la sintaxis para crear y trabajar con la OOP y las clases en un entorno de C++. Pero la programación orientada a objetos es mucho más que las clases, y en esta sesión veremos otra de sus mayores ventajas: La herencia.

Si recordáis hemos ido insistiendo en que una de las ventajas de definir Clases en nuestros programas era aprovechar una de las características de la OOP, llamada data hiding, que significa esconder las variables globales dentro de las propias clases en la medida de lo posible para evitar que un uso atolondrado de las mismas cause problemas globales de difícil detección.

Mientras que la programación estructurada separa por completo los datos y las funciones que los manejan, la idea básica detrás de la OOP es unir ambos conceptos en un único objeto llamado clase,

De ese modo, encapsulamos los datos y los métodos, dentro de las clases y mediante las instrucciones public y private, podemos limitar el acceso de ambas a programadores atolondrados y por ende, limitamos su capacidad de causar daños inadvertidos.

Pero la OOP dispone de más medios de limitar esos daños mediante otra capacidad que se llama Herencia, de no menor importancia, y a la que vamos a dedicar esta sesión.

Y para ello tenemos que volver a hablar del santo grial de la programación: La reusabilidad del código.

A medida que el software se iba haciendo más complicado y los proyectos más descomunales, cualquier sistema que nos permita usar un código que ya teníamos escrito y reusarlo, redunda en una mayor rapidez en el desarrollo y por tanto en mayores beneficios (La pasta manda, como siempre)

Pero copiar y pegar código no es una buena solución, porque al final siempre hay que modificarlo un poco para adaptarlo y la ventaja que teníamos de usar un software probado se pierde al modificarlo, ya que a la primera de cambio aparecen nuevos problemas de depuración con los que no se contaba y que rápidamente añaden horas y coste a un proyecto que parecía chupado hasta convertirlo en ruinoso.

Por eso, los directores de equipos de programación con cierta experiencia, son alérgicos a modificar programas probados (Y depurados con sangre) y acaban forzando a procedimientos de calidad que impidan estas prácticas, lo que está muy bien, pero al final, lo que te ahorras en errores te lo gastas en burocracia.

Y por eso la el concepto de herencia de la OOP, es una magnifica solución a este problema y ha sido adoptada por cualquier departamento de un cierto tamaño. Porque mejora la reusabilidad del código probado, y a la vez te impide tocarlo, evitando el interminable circulo sin fin, de modificar, depurar y vuelta empezar.

Veamos cómo.

5.2. La sintaxis de la Herencia en C++

Habíamos ido definiendo una clase propia, Contador, en las sesiones previas que vamos a usar para ilustrar el concepto de la herencia.

Suponed que ya tenemos probada y depurada la clase Contador y que ahora necesitamos una Clase nueva que en vez de ir creciendo sin fin sea un descontador, para que haga cuentas a cero desde el número que le damos, como para lanzar un cohete.

Podemos coger el código fuente de contador y modificarlo para incluir un decrementador del mismo, pero aquí nos vamos a encontrar con dos posibles problemas:

- Uno, en el mundo Arduino, da gusto porque todo el mundo regala su trabajo, lo que nos permite disponer del código fuente para hacer esto.

Pero en el mundo real, los programadores suelen querer cobrar por su trabajo (Aunque os resulte increíble) y no suelen darte el código fuente de sus programas, con lo que tenemos mal para modificarlos.

- DOS. Aun cuando dispongas del código fuente, el jefe que dirige el proyecto en el que trabajas, tiene a otros 45 programadores a su cargo y te dejará muy claro con un par de ladridos, lo que piensa de que modifiques programas que ya funcionan y se usan en otros sitios (Normalmente a gritos).

Motivo por el que si desear seguir cobrando tu cheque a fin de mes, te conviene buscar una solución alternativa. Y para eso está la herencia.

El método aprobado es derivar una nueva clase de una que ya existe. Esto hace que la clase derivada herede todas las características y métodos de la Clase Base sin tocarla y ahora podamos añadir lo que nos interese, garantizando que la Clase original permanece inalterada.

- Tened en cuenta que tocar una Clase en la que se apoyan otros programas, puede suponer un lío mayúsculo, ya que cualquier pequeña diferencia con el original puede suponer una miríada de problemas en otros programas que ya estaban probados y con los que ahora hay que volver a empezar.

Vamos a empezar definiendo nuestra clase Contador para después ver como derivamos una clase CountDown de ella. Empecemos definiendo una Clase de base sencilla:

```
class Contador
  {   private:
        int N ;

      public:
         Contador( ) : N(0) {}                    // Constructor
         Contador(int k ) : N(k) {}               // Constructor
         int GetCont() ;
  } ;

int Contador::GetCont()
    { return (N) ;}
const Contador &Contador::operator ++()      // Prefix Operator
    {  return Contador( ++N) ;   }
```

Queremos definir una nueva clase que se llame CountDown derivada de Contador y añadirle una función de decremento. Para ello lo primero es ver como derivamos una clase de otra. La sintaxis es esta:

```
Class CountDown :  public Contador          // Es una clase derivada
   {   public:
           Counter Operator {()
               { return Counter(--N) ;
   }
```

En la primera línea declaramos una nueva clase CountDown que deriva de Counter y es de acceso público, y despues definimos un prefix operator para decrementar la variable interna. Aunque la sintaxis es buena, el compilador no tragaría con esto. ¿Adivináis porque?

Si te fijas en la definición de Contador, hemos definido N, el contador interno, como private, y eso significa que no permitirá el acceso a ninguna función externa a la clase Contador (Incluido CountDown), lo que nos hace imposible acceder desde la nueva clase derivada.

Pero que no cunda el pánico. Para que podamos acceder a propiedades o métodos internos desde clase derivadas (Pero no desde cualquier otro medio), necesitamos definirlo no como private, sino como protected:

```
class Contador
   {  protected:                          // Aqui esta el truco
         int N ;

      public:
         Contador( ) : N(0) {}           // Constructor
         Contador(int k ) : N(k) {}      // Constructor

         int GetCont()
            { return (N) ;      }

         Contador operator ++()
           {  return Contador( ++N) ;    }
   } ;
```

Al definir N como protected, significa que podemos acceder a esta variable desde clases derivadas de ella, pero sigue siendo imposible acceder desde un programa externo.

- Aquí surge un debate interminable, acerca de la seguridad de esto, ya cualquier cretino puede derivar una clase de la original y meter mano sin control a las propiedades y métodos protegidos.

- No seré yo quien intente mediar en semejante refriega, pero es evidente que una propiedad como protected es menos segura que como private, pero las ventajas compensan el riesgo en muchas ocasiones y en la vida no hay nada perfecto.

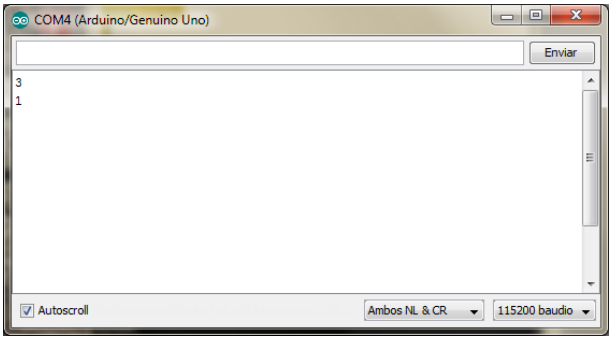
- Y por otro lado, si el cretino se dedica a hacer sandeces con la variable original, tiene la virtud de que el daño solo se lo hace a él y a su código, porque el de los demás que usan la clase original permanece virginal.

A esta capacidad de acceder a los miembros públicos o protected de una clase se le conoce genéricamente por accesibilidad.

Así pues, aquí os dejo copia del programa modificado con protected, y si ahora hacemos:

```
void loop()
   { CountDown C1 ;
     ++C1; ++C1; ++C1;
     Serial.println (C1.GetCont()) ;

     --C1 ; --C1 ;
     Serial.println (C1.GetCont()) ;
   }
```

Obtendremos una salida similar a esta: Que requiere una cierta explicación.



Definimos C1 con de la clase CountDown, lo que explica que pueda hacer el decremento mediante prefix Operator, pero... ¿Cómo se explica que pueda aplicar el prefix increment Operator ++, o el GetCont (), que no están definidos en su declaración?

Como ya habéis adivinado, una clase derivada hereda los métodos y propiedades, de la clase original, (Que sean public o private, claro) y podemos

usarlas sin problema, lo que le confiere una potencia inusitada para definir jerarquías conceptuales.

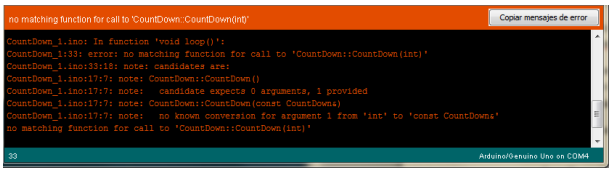
Pero hay otra cuestión sorprendente implícita en ese mismo programita. Y es que el compilador ha inicializado a 0 nuestra instancia C1, sin que exista un constructor en CountDown ¿Por qué?

La respuesta vuelve a ser que el compilador ha interpretado que al no haber constructor propio sin parámetros en la clase derivada, debe aplicar el constructor de la clase base, lo cual puede ser mucho suponer y causar problemas si no estáis advertidos.

¿Significa eso que puedo hacer entonces algo así, Para aprovecharme del segundo constructor de la clase base?

```
CountDown C1(25) ;
```

Ni de coña, y mucho cuidado con esto que es fuente de múltiples dolores de cabeza: El compilador puede usar un constructor por defecto sin parámetros,



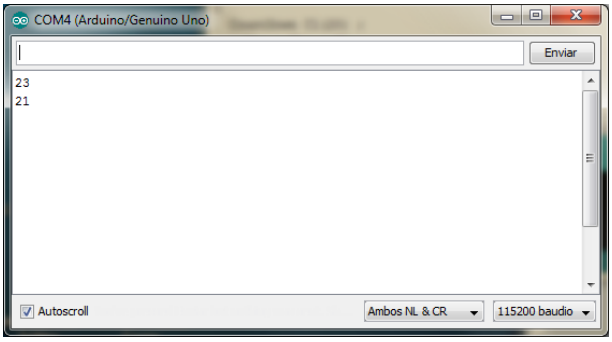
pero cualquier otro debe ser definido en la clase derivada independientemente, como por ejemplo así:

```
class CountDown : public Contador
  {   public:
         CountDown( )  :  Contador() {}          // Constructor
         CountDown(int k )  :  Contador(k) {}

         Contador operator -- ()
            {  return Contador( --N) ;       }
  } ;
```

Donde simplemente le especificamos al compilador que use los constructores disponibles en la clase base (O definir unos completamente nuevos), y así podamos crear C1 con un valor especificado:

```
CountDown C1(20) ;
```

Fijaros en la hasta ahora desconocida sintaxis de:




```
CountDown( )  :  Contador() {}
CountDown(int k )  :  Contador(k) {}
```

Donde especificamos al compilador que cuando se cree una instancia de CountDown, debe invocar el constructor de la clase base que le indicamos. La primera podríamos omitirla porque ya sabemos que el compilador proporcionara un constructor por defecto, pero es buena política definirlo aquí para evitar sobresaltos.

5.3. Function member Overriding

Vale, hemos visto que podemos definir nuevos constructores porque el compilador no aplicará per se mas que el default constructor sin parámetros, y también hemos visto que las funciones disponibles en la clase original están gentilmente a disposición de las clases derivadas, pero puede ocurrir que nos interese redefinir una de ellas para que funcione de otra manera en la nueva clase derivada.

De la misma forma que podíamos sobrecargar funciones a condición de que tuvieran diferentes firmas ¿Puedo redefinir métodos con el mismo nombre?

Y la respuesta es que naturalmente, y ni siquiera necesitamos diferente firma porque el compilador aplicará a cada instancia de una clase la función que le corresponda.

A esta capacidad de redefinir una función miembro con el mismo nombre se le llama Function Overriding (Que no estoy muy seguro de como traducirla al cristiano, porque sería algo así como invalidar u omitir la función)

Podemos forzar un Override de la función miembro GetCont (), en nuestra clase CountDown para hacer que nos devuelva el doble del valor interno del contador.

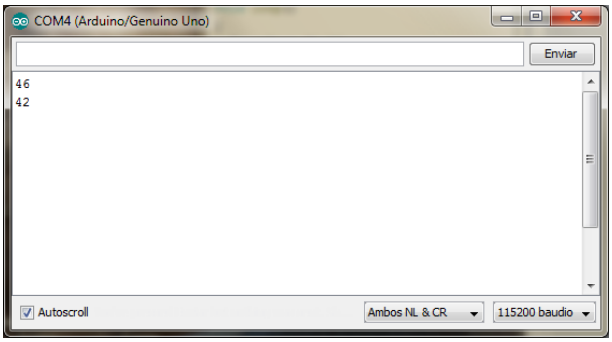
- Algo que resulta bastante estúpido e inútil, pero que nos permite demostrar el concepto con el material que tenemos a mano.

- Normalmente la función Overriding se utiliza para que una función de clases derivados se comporten de forma similar a pesar de ser diferentes, pero soy incapaz de imaginar ningún ejemplo sensato, con el programa que nos ocupa.

Para forzar un Override de la función miembro GetCount() podemos hacer algo así:

```
class CountDown : public Contador
   {  public:
         CountDown( ) : Contador() {}          // Constructor
         CountDown(int k ) : Contador(k) {}

         Contador operator -- ()
             {   return Contador( --N) ;    }

         int GetCont()
             {   return(2*N) ; }
   } ;
```

En donde simplemente creamos una nueva función miembro de CountDown con el mismo nombre y distinta ejecución. Y tal y como esperábamos este es el resultado



5.4. Consideraciones finales

El objetivo de estos humildes tutoriales es proporcionar una introducción a los conceptos detras de la OOP y allanar el camino de los que se inician en este nuevo modo de pensar en la programación, pero no pretende ser, de ningún modo, una clase doctoral o a fondo en ningún concepto.

Por eso nuestra intención no es ser exhaustivos, sino más bien presentar los conceptos básicos que os permitan desbrozar el camino inicial y permitiros seguir aprendiendo por vuestra cuenta.

Y naturalmente solo hemos desvelado la punta del iceberg en lo que se refiere a la programación orientada a objetos y especialmente en las cuestiones de herencia, porque se podría seguir hablando indefinidamente sobre el tema, pero creemos que ha llegado el momento de cortar aquí el tema OOP por ahora.

Por supuesto que hay infinidad de cuestiones como la herencia múltiple, el polimorfismo o las funciones virtuales que darían tema para una interminable y probablemente desierta disquisición que no nos conduciría a nada por ahora.

En mi experiencia no suele ser útil, describir soluciones a problemas que aún no habéis tenido, y antes de entrar en esos temas, deberéis trabajar y madurar los conceptos que hemos expuesto hasta aquí para desarrollar la experiencia necesaria para seguir avanzando, aquellos que decidáis seguir este camino.

Recuerdo aquel chiste en el que al salir de una clase de universidad, un alumno pregunta a otro que le ha parecido el profesor. Y este responde que debe ser una eminencia porque no ha entendido nada.

Nuestra aspiración es a ayudar a aprender y no a ganar puntos de cara a no sé muy bien que otros estamentos. Nuestros amigos saben ya, hace mucho, que no tenemos remedio y a pesar de todo (Increíblemente) siguen invitándonos a cañas de vez en cuando.

Pero sí que me gustaría aseguraros que no hay nada raro en la programación orientada a objetos que no podáis aprender. No es para tanto. Simplemente es otra manera conceptual de organizar tus programas, y con grandes ventajas si el proyecto crece.

5.5. Resumen de la sesión

- Hemos hecho una primera aproximación a la herencia en la OOP.
- Presentamos su sintaxis.
- Vimos los conceptos relativos a los constructores.
- Conocimos el function Overriding.