

Variables y Punteros en C++

No hay duda de que los punteros son una de las características más definitorias de C y constituyen una de sus grandes fortalezas, a la vez que uno de sus mayores problemas, especialmente para las personas que dan sus primeros pasos en programación.

La experiencia demuestra que el concepto de puntero, es extraño, e inicialmente provoca la típica reacción de rechazo. Además tienen la dudosa virtud de ser potencialmente peligrosos en nuestros programas.

Bien utilizados, son excepcionalmente útiles para resolver cierto tipo de problemas, pero aprender a manejarlos puede generar serios dolores de cabeza, especialmente cuando tratamos de depurar un programa que se niega a funcionar como debe.

- *Hace unos años equivocarse en un programa con los punteros solía acabar colgando la PC sin previo aviso, lo que tenía gracia siempre y cuando lo hubiésemos salvado a tiempo (algo a lo que nos acostumbrábamos con rapidez)*
- *Actualmente las PCs con sistemas operativos modernos, Tanto Windows como Linux y Mac, aíslan los programas de la maquina con lo que no es fácil dinamitar el entorno de trabajo, pero es relativamente fácil bloquear el programa.*
- *Claro que en Arduino no tenemos un SO que nos evite el destrozo, así que será relativamente fácil colgarlo.*

Previo a adentrarnos en la teoría de punteros, deberemos hablar de las variables y de cómo se almacenan en la memoria.

Las variables en C++

La memoria del Arduino, como la de la PC, está numerada, a través de las llamadas **direcciones**. Cada una de las posibles posiciones de memoria tiene una **dirección única** que debe ser especificada cuando queremos leer o escribir su valor.

El Arduino modelo UNO tiene 32k de Flash para almacenar los programas y 2k de RAM para almacenar las variables.

Por eso cuando definimos una variable mediante una instrucción, el compilador le asigna una posición en la RAM de un byte si es un tipo de dato char o byte, de dos posiciones para un tipo

de dato int y de 4 posiciones si es un long, de modo que los tipos más largos se puedan almacenar en posiciones consecutivas que los acomoden. Si declaramos una variable así:

```
int j;
```

Simplemente le informamos al compilador utilizaremos una variable llamada *j*. Si la definimos como

```
j = 10;
```

Lo que le decimos al compilador es que vamos a usar la variable *j* que definimos antes y que debe asignarle ese valor de 10 que se especificó.

- Por esa razón, declarar y definir una variable no es lo mismo, a pesar de que muchas veces usamos los términos indistintamente.

Pero si pensamos un momento en lo que tiene que hacer el compilador, veremos que hay dos conceptos distintos. Por una parte está la definición de la variable *j*; el compilador tiene que crear una tabla donde anote que existe una variable llamada *j* y por otro lado tiene que asignar físicamente una o más posiciones de memoria para contener el valor de esa variable.

Si el compilador le asigna a la variable *j* la posición de memoria 2020, y graba en ella el valor que hemos pedido de 10, tendremos

NOMBRE	DIRECCIÓN DE MEMORIA	CONTENIDO
j	2020	10

Es decir, que para el compilador la variable *j* esta almacenada en la posición de memoria 2020 y el contenido de esa posición (o de la variable, que es lo mismo) es 10.

C++ nos exige que **declaremos** las variables antes de usarlas porque deberá reservar espacio para ellas del tamaño adecuado, y cuando las definimos o asignamos valores, escribe el valor en la dirección de memoria que la tabla le indica.

Por lo tanto, la cuestión es comprender que una cosa es la **dirección de memoria** que contiene el valor y otra distinta es el **contenido** de esa dirección de memoria.

Esto nos da una idea de porque los resultados son impredecibles cuando escribimos un valor de tipo long en una dirección de memoria que corresponde a un tipo int. Como el tipo long consume 4 bytes, cuando lo intentemos ubicar en el espacio reservado para un tipo int, de 2

bytes, va a pisar el contenido de las siguientes posiciones de memoria sin saber si están en uso o no. Probemos este ejemplo:

```
void setup()
{ Serial.begin(9600); }
void loop()
{
  int v ;
  long L = 100000 ;
  v = L ;
  Serial.println(v);
}
```

Uno esperaría que el compilador se diera cuenta de esto y lo considerara un error, pero C++ ignora el tema y nos informa tranquilamente que el resultado de v es de -31.072, que es lo que sale de los dos últimos bytes del 100.000 en binario y con signo.

Claro está, que en el caso de que el valor de L sea inferior a lo que cabe en un entero con signo, no notaríamos el problema, aparte de que acaba de sobrescribir posiciones de memoria contiguas que a su vez pueden ser de otra variable o peor aún, un puntero a una función, con lo que tendríamos una variable que mientras su valor es inferior a 2^{15} (un int son 15 bits de datos y uno de signo) , no pasa nada pero que cuando pase por ahí nos devuelve valores incongruentes, y ya veremos cómo cuesta averiguar qué es lo que está sucediendo.

El caso es aún peor si hemos sobrescrito la dirección de una función, porque entonces, en algún momento el programa intentará ejecutar una función con un salto a una dirección que es sencillamente basura y acabamos de conseguir un cuelgue completo del programa y del Arduino.

Resumiendo, los tipos de variables son:

- **char**, se utilizan para almacenar caracteres, ocupan un byte.
- **byte**, pueden almacenar un número entre 0 y 255.
- **int**, ocupan 2 bytes (16 bits), y por lo tanto almacenan número entre 2^{15} y $2^{15}-1$, es decir, entre -32,768 y 32,767.
- **unsigned int**, ocupa también 2 bytes, pero al no tener signo puede tomar valores entre 0 y $2^{16}-1$, es decir entre 0 y 65,535.
- **long**, ocupa 32 bits (4 bytes), desde -2,147,483,648 a 2,147,483,647.
- **unsigned long**, ocupa 32 bits (4 bytes), desde 0 a 4,294,967,295.

- **float**, son números decimales que ocupan 32 bits (4 bytes). Pueden tomar valores entre $-3.4028235E+38$ y $+3.4028235E+38$.
- **double**, también almacena números decimales, pero disponen de 8-bytes (64 bit), para el caso del DUE. En el UNO y otros modelos, ocupa 32 bits (4 bytes), esto es, equivalente al tipo **float**.

Ámbito de las variables

El concepto de ámbito está ligado a lo que se conoce como “alcance” o visibilidad de la variable. Destacamos las del tipo “global” y “local”.

Ámbito global

```
int a= 5;
```

```
void setup()
{
  // initialize Serial
  Serial.begin(9600); // baudrate 9600
  Serial.println(String(a));
  Serial.println("fin setup");
}
```

```
void loop()
{
  a = a + 1;
  Serial.println(String(a));
  delay(1000);
}
```

Decimos que la variable *a* es global porque se puede acceder a ella desde cualquier parte, es decir, estamos accediendo a su valor desde las funciones *setup* y *loop*.

Ámbito local

Son variables que solo *existen* dentro del ámbito en el que han sido declaradas. Para simplificar el concepto, diremos que un ámbito es lo que está entre llaves. Si las utilizamos fuera de su *ámbito* tendremos un error de compilación. Al existir solo dentro de su ámbito en un programa se podría repetir el mismo nombre de variable en distintos ámbitos. Vamos a ver algunos ejemplos:

```
void setup()

{
  int a= 5; //la variable a solo existe dentro de la función setup
  // initialize Serial
  Serial.begin(9600); // baudrate 9600
```

```

    Serial.println(String(a));
    Serial.println("fin setup");
}
void loop()
{
    //al compilar daría un error de compilación, porque a no existe en loop
    a = a + 1;
    Serial.println(String(a));
    delay(1000);
}

```

Al compilar este programa obtendremos un error de código porque la variable *int a* es de ámbito local, y solo existe dentro de la función *setup*. Por esa razón no la podemos usar en la función *loop*.

En cambio, en el siguiente código el programa compilará ya que la variable *int a* ha sido declarada tanto en la función *setup* como en la función *loop*.

```

void setup()
{
    int a = 5; //la variable a solo existe dentro de la función setup
    // initialize Serial
    Serial.begin(9600); // baudrate 9600
    Serial.println(String(a));
    Serial.println("fin setup");
}

void loop()
{
    int a = 0;
    a = a + 1;
    Serial.println(String(a));
    delay(1000);
}

```

Observar que la variable *a* del *loop* se crea y se destruye en cada iteración, por lo que siempre se inicializa a 0 y se le añade 1... Por lo tanto, siempre vale 1 dentro del *loop*.

Los punteros en C++

Una vez comprendida la diferencia entre la dirección de una variable y su contenido estamos en condiciones de entender los **punteros** (pointers en inglés). Un **puntero** es, simplemente, un tipo de dato que contiene la **dirección física** de algo en el mapa de memoria.

Cuando declaramos un puntero se crea una variable de tipo pointer, y cuando le asignamos el valor, lo que hacemos es apuntarlo a la dirección física de memoria, donde se encuentra algo concreto, sea un entero, un long o cualquier elemento que el compilador conozca.

Como en el Arduino UNO el mapa de memoria es de menos de 64k, los punteros que especifican una dirección de memoria se codifican con 16 bits o 2 bytes ($2^{16} = 65.536 > 32.768$). Una curiosidad de los punteros, es que o bien tienen una dirección de 16 bits en Arduino, o contienen basura, pero no hay más opciones. Porque aunque pueden apuntar a tipos de diferente longitud, la memoria en la que empiezan se sigue definiendo con 16 bits (aunque el tipo indica cuantos bytes hay que leer para conseguir el dato completo).

Naturalmente si tenemos una variable como `v` en Arduino, podemos conseguir la dirección en la que esta almacenada, con el operador `'&'`, sin más que hacer `&v`:

```
int v = 100 ;Serial.println( &v)
```

Lamentablemente, esto sí que generará una queja por parte del compilador diciendo que el tema es ambiguo y tenemos que hacer un cast de tipo de la siguiente manera:

```
int v = 100 ;  
Serial.println( (long)&v);
```

Que al momento de hacer la prueba responde diciendo 2290 pero puede ser otro.

- *Un cast consiste en forzar la conversión de un tipo en otro, y se efectúa precediendo a la variable que queremos forzar por el tipo que deseamos entre paréntesis.*

En realidad un **puntero** es sencillamente otro tipo de datos que **contiene una dirección de memoria** y cuando entendemos esto, comprendemos que podemos definir punteros, por si mismos, para usarlos de diferentes maneras.

Para declarar un puntero usamos el operador `'*'`, basta con declararlo precedido de un `*`:

```
int *p_data ;
```

Que significa un puntero a un int llamado `p_data`.

Aunque el puntero a un int es una **dirección**, lo mismo que un puntero a un long, es imprescindible indicarle al compilador a qué tipo de puntero vamos a apuntar, para que sepa cuantos bytes tiene que leer o escribir cuando se lo pidamos.

Si leemos un long donde hay un int, leeremos basura. Si escribimos un long donde hay un int acabamos de corromper el valor de otras posibles variables y estamos en el caso que definimos antes con las variables.

Veamos otro ejemplo:

```
void setup() {  
  Serial.begin(9600);  
}  
void loop() {
```

```

int a = 100;
int * puntero_a = &a; //el contenido de puntero_a es igual a la dirección de a

Serial.print("Valor contenido en a: ");
Serial.println(a);      //Imprime a
Serial.print("Direccion de memoria de la variable a: ");
Serial.println((unsigned int)&a);    //Imprime la dirección de a
Serial.print("El valor contenido en la direccion de memoria apuntada por el puntero_a: ");
Serial.println(*puntero_a);    //imprime el valor que hay en la dirección contenida en el
puntero
Serial.print("La direccion contenida en el puntero: ");
Serial.println((unsigned int)(puntero_a));    //imprime el valor contenido en el puntero
(dirección de a)
Serial.print("La direccion del puntero: ");
Serial.println((unsigned int)&puntero_a);    //imprime la dirección de puntero_a (dirección
donde está guardada la dirección de a)
Serial.println();

delay(3000);
}

```

Cuyo resultado en la consola serie fue: (pueden variar las direcciones de memoria):

Valor contenido en a: 100

Direccion de memoria de la variable a: 2294

El valor contenido en la direccion de memoria apuntada por el puntero puntero_a: 100

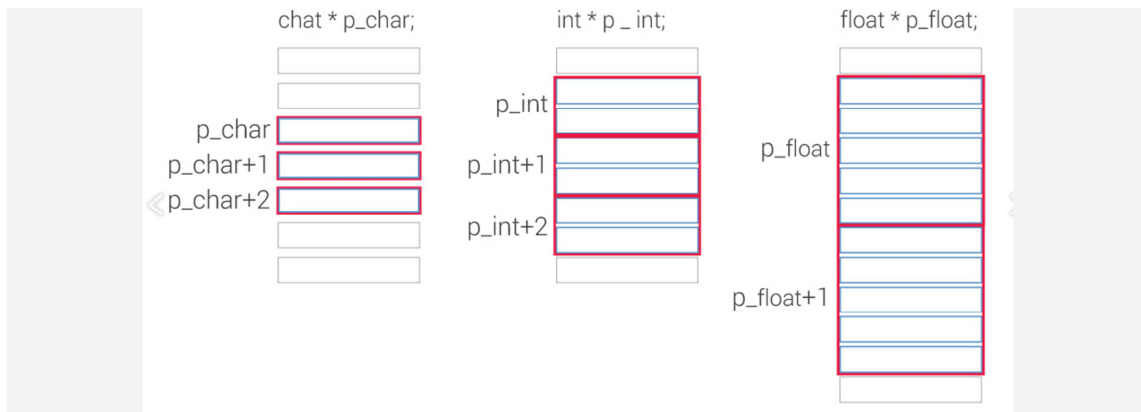
La direccion contenida en el puntero: 2294

La direccion del puntero: 2292.

Representando gráficamente:

	Valor Decimal	Valor Binario
	100	00000000 01100100
	2294	00001000 11110110
	2292	00001000 11110100

2290	01001100		} variable puntero_a
2291	11010011		
& puntero_a= 2292	00001000	}	} variable a
2293	11110110		
puntero_a= 2294	00000000		
2295	01100100		
2296	00011101		
2297	10101010		



- A los nombres de los punteros, se les aplican las mismas reglas que a los nombres de variables o funciones, pero conviene dejar claro que es un puntero para que quien lo lea, no se despiste y malinterprete el programa.
- Así es muy frecuente que al nombre de los punteros se les empiece por algo como “p_” o “ptr”, para simplificar la comprensión del código.

Ahora, para asignar la dirección de una variable a un puntero se procede como muestra el código a continuación:

```
int num = 5;
int *ptrNum;
ptrNum = &num;
```

Como `&num` nos da la dirección física donde se almacena `num`, basta con asignarla a `ptrNum` por las buenas. El operador “&” precediendo a una variable devuelve su dirección de memoria.

Ahora `ptrNum` apunta a la dirección donde está almacenada la variable `num`. ¿Podría usar esta información para modificar el valor almacenado allí? Desde luego:

```
int num = 5;
int *ptrNum;
ptrNum = &num;
*ptrNum = 7;
Serial.println( num);
```

Veremos que la respuesta en la consola al imprimir `num` es 7. Hemos usado un puntero para modificar el contenido en la celda a la que apunta usando el operador `*`.

Podemos asignar un valor a la posición a la que apunta un puntero, basta con referirse a él con el `*` por delante. Y si queremos leer el contenido de la posición de memoria a la que apunta un puntero usamos el mismo truco:

```
int num = 5;
int *ptrNum;
ptrNum = &num;
Serial.println( *ptrNum);
```


El resultado será 5.

Resumiendo:

- Un puntero es una variable que apunta a una dirección concreta de nuestro mapa de memoria.
- Para conocer la dirección concreta de donde algo está almacenado, basta con preceder el nombre de ese algo con el operador "&" y esa es su dirección, que podemos asignar a un puntero previamente definido (Del mismo tipo).
- Usamos el operador "*" precediendo al nombre del puntero, para indicar que queremos leer o escribir en la dirección a la que apunta, y no, cambiar el valor del puntero.

Mucho cuidado con lo siguiente. La instrucción

```
*ptrNum = 7;
```

Tiene todo el sentido del mundo, pues guarda un 7 en la dirección al que el valor de *ptrNum* apunta. Pero en cambio

```
ptrNum = 7 ;
```

Es absurda, porque acabamos de apuntar a la dirección de memoria número 7. Si escribimos algo en una posición de memoria cuyo uso desconocemos generará resultados imprevisibles.

¿Para qué sirven los punteros?

La primera respuesta es, que los argumentos que pasamos a las funciones se **pasan por valor**, es decir que una función no puede cambiar el valor de la variable que le pasamos, algo que ya vimos en la sección "[Ámbito de las variables](#)". Probemos esto:

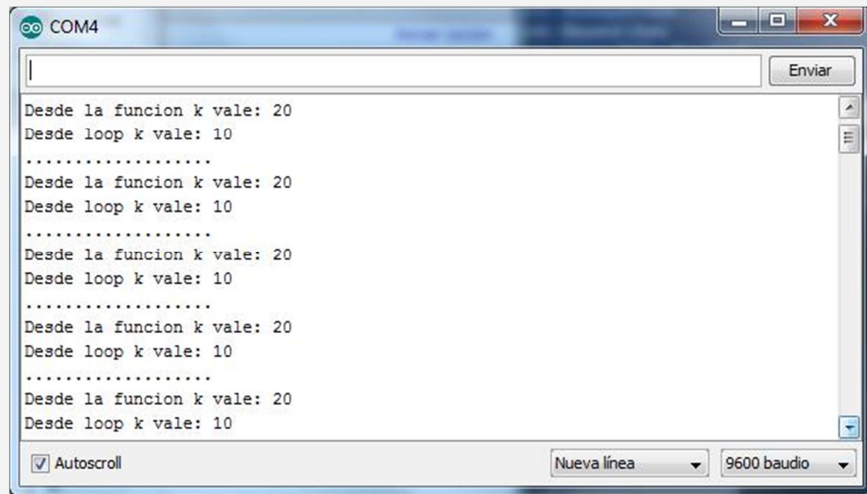
```
void setup()  
{ Serial.begin(9600);}
```

```
void loop()  
{ int k = 10;  
  Serial.print("Desde loop k vale: ");  
  Serial.println(k);  
  dobla(k);  
  Serial.println(".....");  
}
```

```
void dobla(int k)  
{ k = k*2 ;  
  Serial.print("Desde la función k vale: ");  
  Serial.println(k);  
}
```

}

El resultado es así:



Como la variable k del programa principal y la k de la función `dobla` son de ámbito diferente, no pueden influirse la una a la otra.

Cuando llamamos a la función `dobla(k)`, lo que el compilador hace es copiar el valor de k y **pasárselo por valor** a la función, pero se guarda mucho de decirle a la función, la **dirección de la variable k** . De ese modo aislamos el ámbito de las dos variables. Nada de lo que se haga en `dobla` influirá en el valor del k de la función principal.

A veces nos puede interesar que una función modifique el valor de la variable. Podríamos definir una variable global y con eso podríamos forzar a usar la misma variable para que modifique su valor. El problema es que a medida que los programas crecen, el número de variables globales puede crecer, y seguir las puede complicarse mucho.

Otra solución, es pasar a una función la dirección de la variable y ahora la función sí que puede modificar el valor de esta, Probemos esto:

```
int k = 10;
void setup()
{ Serial.begin(9600);}

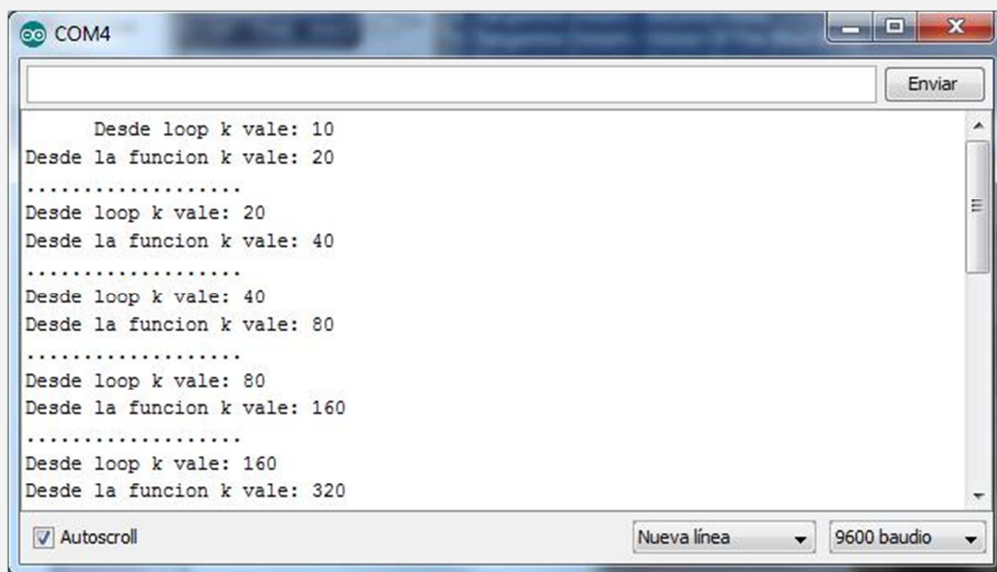
void loop()
{ Serial.print("Desde loop k vale: ");
  Serial.println(k);
  dobla( &k );      // Pasamos la dirección de k y no su valor
  Serial.println(".....");
}

void dobla(int *k)   // Avisamos a la función de que recibirá un puntero
{ *k = *k * 2 ;
  Serial.print("Desde la función k vale: ");
  Serial.println (*k);
```

}

- Hemos sacado la definición de *k* fuera del loop para que no inicialice el valor en cada vuelta (aunque hemos acabado definiendo una variable global que es lo que queríamos evitar).

El resultado es:



```
Desde loop k vale: 10
Desde la funcion k vale: 20
.....
Desde loop k vale: 20
Desde la funcion k vale: 40
.....
Desde loop k vale: 40
Desde la funcion k vale: 80
.....
Desde loop k vale: 80
Desde la funcion k vale: 160
.....
Desde loop k vale: 160
Desde la funcion k vale: 320
```

Al pasarle la variable **por referencia**, la función `dobla()` **sí** que ha podido modificar el contenido de la variable, y en cada ciclo la dobla.

Cuando hablamos de las funciones, en las sesiones previas, dijimos que podíamos pasar a una función tantos parámetros como quisiéramos, pero que solo podía devolvernos un valor.

Pero con nuestro reciente dominio de los punteros podemos entender porque no nos preocupa. Podemos pasar tantas variables como queramos **por referencia** a una función, con **punteros**, de modo que no necesitamos que nos devuelva múltiples valores, ya que la función puede cambiar múltiples variables.

Punteros y Arrays

Vamos a probar este programa:

```
void setup()
{ Serial.begin(9600);}

void loop()
{ char h[] = { 'P','r','o','m','e','t','e','\n'};

  for (int i=0 ; i < 9 ; i++)
    Serial.print( h[i] );
  Serial.flush();
  exit(0);
}
```

El resultado es este:



- Si no utilizamos el Serial.Flush, probablemente no veremos el mensaje completo. La razón es que lo que enviamos por el puerto serie, se transmite en bloques y no carácter a carácter.
- Por eso, asegurarnos que todo se ha enviado usa el flush() antes de salir con exit(0).

Hemos utilizado *p* como un *array* de *char* y usado un loop para recorrerlo e imprimirlo. Nada nuevo en esto. Pero hagamos un pequeño cambio en el programa:

```
void setup()
{ Serial.begin(9600);}

void loop()
{ char h[] = { 'P','r','o','m','e','t','e','\n'};

  for (int i=0 ; i < 9 ; i++)
    Serial.print( *(h + i) );
  Serial.flush();
  exit(0);
}
```

Hemos cambiado la línea:

```
Serial.print( h[i] );
```

Por esta otra:

```
Serial.print( *(h + i) );
```

Y el resultado es... exactamente lo mismo. ¿Por qué?

Pues porque un *array* es una colección de datos, almacenada en posiciones consecutivas de memoria (y sabemos el tamaño de cada dato porque lo hemos declarado como char o int o lo que sea), y lo que el compilador hace cuando usamos el nombre del *array* con un índice como *h[i]* es apuntar a la dirección de memoria donde empieza el *array* y sumarle el índice multiplicado por la longitud en bytes, de los datos almacenados, en este caso 1 porque hemos declarado un char).

En realidad es otra forma de decir lo mismo. Como esto:

```
Serial.print(*( h + i * sizeof(char)));
```

Cuando usamos *h+i*, el compilador entiende que sumemos *i* al puntero que indica el principio del array, y por eso, al usar el operador *, busca el contenido almacenado en *h+i*, que es exactamente lo mismo que la forma anterior.

Así que si utilizamos un *array* sin índice, lo que en realidad estamos haciendo es pasar un puntero al comienzo en memoria del *array*, o sea, su dirección de inicio.

```
void setup()  
{ Serial.begin(9600);}
```

```
void loop()  
{ char h[] = { 'P','r','o','m','e','t','e','\n'};  
  char *ptr = h ;  
  for (int i=0 ; i < 9 ; i++)  
    Serial.print(*ptr++);
```

Pues más de lo mismo, pero en ese estilo típicamente críptico que caracteriza algunos de los aspectos más oscuros de C++. Declaramos sobre la marcha un puntero que apunta a *h* con:

```
char *ptr = h ;
```

Utilizamos el bucle para contar simplemente, pero **ptr* significa el contenido al que *ptr* apunta, o sea *h*, y después de imprimirlo, incrementamos *ptr*, con lo que apunta al siguiente char del *array*