



CIRCUITOS DIGITALES Y MICROCONTROLADORES 2025

Facultad de Ingeniería
UNLP

Programación de Microcontroladores
con Lenguaje C

Ing. José Juárez

Motivación del uso del [Lenguaje C \(Wiki\)](#)

- En conceptos de arquitectura aprendimos a programar en el lenguaje ensamblador porque tiene una relación directa con el código máquina que se genera y ejecuta en una determinada arquitectura de procesador y por lo tanto es el más eficiente en el uso de la misma.
- Sin embargo, el aspecto más importante en la elección de un lenguaje de programación de microcontroladores, es la organización de la solución de un problema. Un programa en ensamblador resultante de una mala o nula organización y en aplicaciones complejas puede ser más ineficiente que un programa en C bien organizado.
- C es un Lenguaje estructurado de Alto Nivel y estandarizado (ANSI C o C89 / ISO C o C90, última revisión ISO/IEC C23) pero contiene características de bajo nivel que lo hacen ideal para programar a nivel del hardware.
- Esto facilita la programación, el entendimiento, el mantenimiento y la portabilidad del código a otras arquitecturas de microcontroladores.
- C permite reducir el tiempo de desarrollo por medio de la reutilización de código (bibliotecas)

Motivación del uso del [Lenguaje C \(Wiki\)](#)

- Con el lenguaje C se facilita la modularización de código y la encapsulación de los datos permitiendo el desarrollo de proyectos en equipos de trabajo.
- Hoy en día existen compiladores de C para casi todas las arquitecturas de microcontroladores del mercado, gestionado por el fabricante, de uso libre o de compañías de terceros (third party companies)
- Los compiladores poseen optimizadores para aprovechar al máximo las ventajas de cada arquitectura.
- Los compiladores permiten insertar (y reutilizar) líneas de assembler en un código C así como los compiladores de C++ permiten incluir al C.
- Si se requiere un nivel de abstracción mayor para una determinada aplicación de usuario puede utilizarse C++, con drivers de bajo nivel escritos en C o en assembler.
- En sistemas embebidos es indispensable contar con herramientas que facilitan la depuración de errores (debugging) y el análisis de las aplicaciones (Profiling/tracing)
- Hay mucho soporte académico disponible (bibliografía, foros, cursos, wiki)

Eficiencia en el uso de recursos

- La eficiencia mas alta en el uso de los recursos del MCU (uso de memoria ROM, RAM y tiempo de ejecución) se logra en lenguaje assembler (siempre y cuando sea un programador experimentado), sin embargo un buen compilador de C puede acercarse a la misma eficiencia si esta optimizado para esa arquitectura en particular ya que utiliza el 100% del set de instrucciones y los modos de direccionamiento.
- El código en C, en aplicaciones sencillas, es generalmente mas extenso (ocupa mayor espacio de la memoria de programa) y es más lento que el código assembler (contiene código máquina adicional), sin embargo a medida que los proyectos son más complejos el tamaño del código tiende a equilibrarse.
- Si la velocidad de ejecución de cierta porción de código es crítica, la misma puede realizarse en assembler, por ejemplo: el cambio de contexto entre tareas en el ámbito de un RTOS o los de entrada/salida (I/O) a máxima velocidad.

Desventajas:

- La gestión de memoria es responsabilidad del programador y es compleja la tarea de brindar abstracción y seguridad respecto a otros lenguajes de alto nivel.
- Los compiladores de C tienen “un costo” o tienen limitaciones en el tamaño de código compilado (los ensambladores son gratuitos y sin restricciones)
- Los compiladores son programas y poseen “bugs” lo que implica que hay que mantenerse actualizado en las versiones de los mismos.

Resumiendo

- Características del desarrollo de programas en microcontroladores
 - Limitada capacidad memoria de programa (ROM) y de datos (RAM)
 - Limitado espacio para la pila (stack)
 - Programación orientada al Hardware
 - Conocimiento de la temporización de las tareas, rutinas, interrupciones, etc
 - Uso de identificadores y palabras claves específicas: @, interrupt, _attribute_ ...
 - Diferentes tipos de modificadores de almacenamiento: near, far, rom, section ...
 - Optimización en función del arquitectura 8, 16, 32bits (algunos compiladores se apartan del ANSI-C)
- La programación eficiente de microcontroladores requiere de:
 - El conocimiento de la arquitectura del MCU
 - El conocimiento de las herramientas de depuración y análisis
 - El uso de los tipos de datos “nativos” y su direccionamiento
 - El uso de modularización y abstracción de la aplicación
 - El uso “responsable” de las bibliotecas estándar y del soporte externo
 - El uso “responsable” de la asignación dinámica de memoria

En la práctica utilizaremos [Microchip Studio IDE](#) que cuenta con AVR 8-bit GNU Toolchain que incluye el compilador C/C++, ensamblador y binutils (GCC and Binutils), Standard C library (AVR-libc) y GNU Debugger (GDB)

Empecemos repasando los tipos de datos en C

• Tipo		#bits	Rango de representación	bytes
signed char		8bits	-128 a +127	1
unsigned char		8bits	0 a 255	1
short int		16bits	-32768 a 32767	2
unsigned short		16bits	0 a 65535	2
int		???	(depende del compilador)	
long		32bits	- 2147483648 a 2147483647	4
unsigned long		32bits	0 a 4294967295	4
float	(IEEE32)	32bits	1.17549x10-38 a 3.40282x10+38	4
double	(IEEE64)	64bits	2.22507x10-308 a 1.79769x10+308	8

- Notar que la cantidad de bits asignada a un `int` puede variar de compilador en compilador manteniéndose la relación:

`sizeof(char) < sizeof(short) <= sizeof(int) <= sizeof(long)`

Tipos de datos en C

- A partir de C99 se incluye las definiciones de tipo para mejorar la compatibilidad.
- Incluir en el proyecto: `#include <stdint.h>` en la cual se definen:
 - `typedef signed char` `int8_t;`
 - `typedef unsigned char` `uint8_t;`
 - `typedef short int` `int16_t;`
 - `typedef unsigned short int` `uint16_t;`
 - `typedef long int` `int32_t;`
 - `typedef unsigned long int` `uint32_t;`
 - `typedef long long int` `int64_t;`
 - `typedef unsigned long long int` `uint64_t;`
- Si `stdint.h` no se encuentra disponible, definir los tipos en el proyecto.
- ¿para qué sirve `typedef`?

Tipos de datos en C

- **Primer lección** : Utilizar el tipo de dato óptimo para la arquitectura del uC
- **Arquitecturas de 8 bits:**
 - el tipo de dato más adecuado es **signed char** o **unsigned char**
 - Operaciones en ALU de 8bits en un ciclo de reloj
 - Operaciones con datos enteros de mayor precisión (16 o 32 bits) deben realizarse por código (generalmente incluido como funciones de biblioteca del compilador)
 - El uso de datos **float** o **double** en arquitecturas con ALU de punto fijo solo debe hacerse cuando es estrictamente necesario. Las operaciones con este tipo de dato se realizan mediante rutinas de biblioteca y consumen muchos recurso del uC (ciclos de reloj y bytes de memoria).
- **Arquitecturas de 32 bits:**
 - el tipo de dato más adecuado es **long int** o **unsigned long int**
- **Segunda lección** : Utilizar el signo adecuado.
 - **Si no importa el signo utilizar unsigned.**
 - Los modificadores **unsigned** / **signed** indican al compilador que instrucciones de assembler debe utilizar.
 - Muchas arquitecturas son más eficientes cuando operan con números sin signo.
 - Si la variable es **signed** se utilizarán instrucciones que operan sobre **Ca2**, caso contrario no.

Alcance de las variables en C

- **Locales:**

- Son variables que viven dentro del ámbito de una función { } cuando está se está ejecutando
- Desde el punto de vista del almacenamiento en memoria, las variables locales se alojan en los registros CPU o en la memoria de pila (stack)
- Por esta razón, solo la función donde están declaradas conoce su ubicación, haciéndolas invisible a otras funciones.
- Una vez finalizada la función son descartadas liberando memoria.
- Un beneficio adicional de las variables locales es que permite modularidad y por tanto la reutilización de código

- **Globales:**

- Son variables que pueden ser accedidas por desde cualquier parte del programa.
- Se declaran fuera de toda función y son alojadas en direcciones fijas de memoria RAM (puede requerir modos de direccionamiento extendido oara especificar la dirección de memoria)
- Cualquier función puede acceder y modificar su contenido, incluso en un servicio de interrupción
- Las variables globales se inicializan durante la ejecución del código de "startup"

Modificadores y tipo de almacenamiento

- **Static:**

- Una variable local declarada como static posee una dirección fija de memoria RAM que se conserva durante toda la ejecución del programa. De esta manera su valor se mantiene entre las sucesivas llamadas a la función.
- Una variable global declarada como static se convierte en una variable “privada” y su alcance se limita solo al archivo .c donde está definida.
- Una función declarada como static se convierte en una función “privada” y su alcance se limita solo al archivo .c donde está definida.

- **Volatile:**

- Advierte al compilador que la variable puede cambiar su valor por acción externa al programa y NO debe aplicar optimizaciones sobre dicha variable
- Ejemplo 1: todas las variables que representan los registros del MCU deben ser volátiles
- Ejemplo 2: variables globales modificadas por una interrupción deben ser volátiles

- **Const:**

- Indica al compilador que la variable es READ_ONLY y por lo tanto no puede modificarse en otra parte del programa (evita errores de escritura indeseados).
- El compilador puede almacenar este tipo de variables en ROM, optimizando el uso de la memoria RAM (uso de PROGMEM).

Modificadores y tipo de almacenamiento

- **Extern:**

- Indica que la variable está definida en otro archivo .c del proyecto (o módulo externo) pero se desea utilizarla en el archivo actual.
- se aplica solo a variables globales (públicas)

- **Register:**

- Sugiere al compilador que la variable local sea colocada en algún registro CPU para optimizar su tiempo de acceso.

Veremos ejemplos cuando avancemos con la práctica

Control de Flujo de un programa

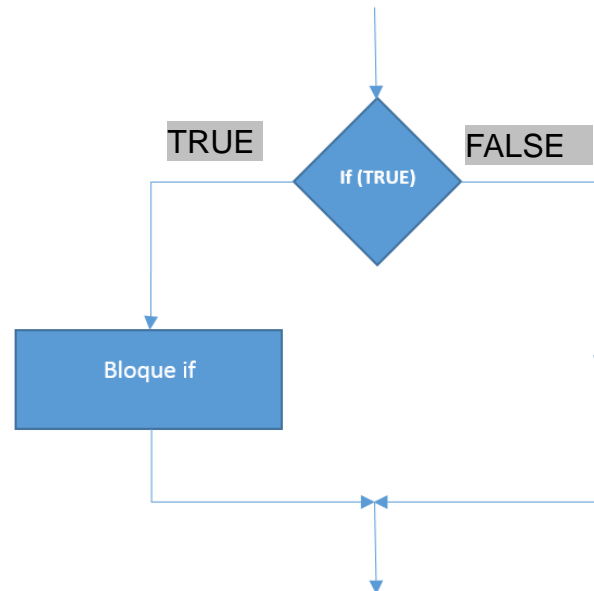
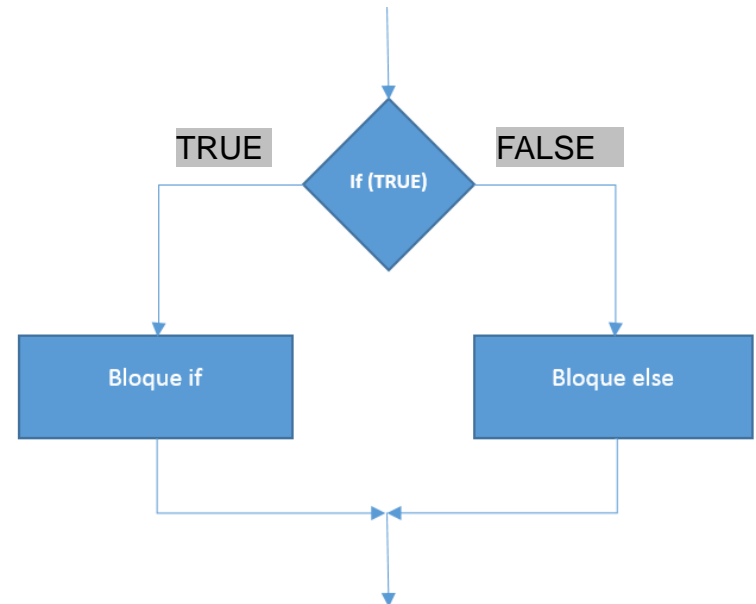
- Bifurcación condicional:

```
If (condición==TRUE)
{
    caso true
}
else {
    caso false
}
```

Podemos omitir el else:

```
If (condición==TRUE)
{
    ...
}
```

- ¿Cómo se formula una condición?
- ¿Qué es TRUE? ¿Qué es FALSE?



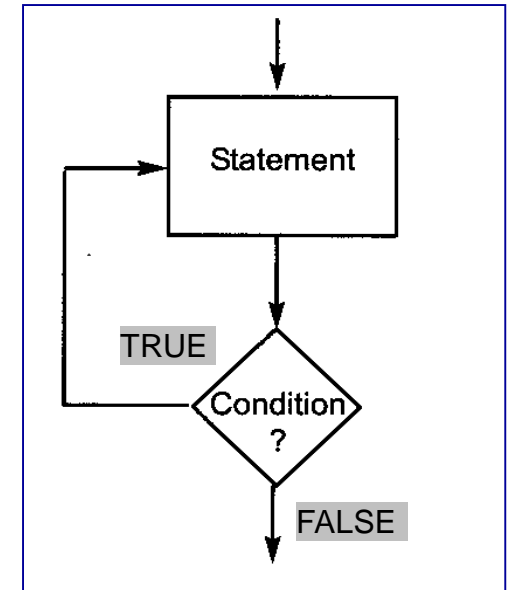
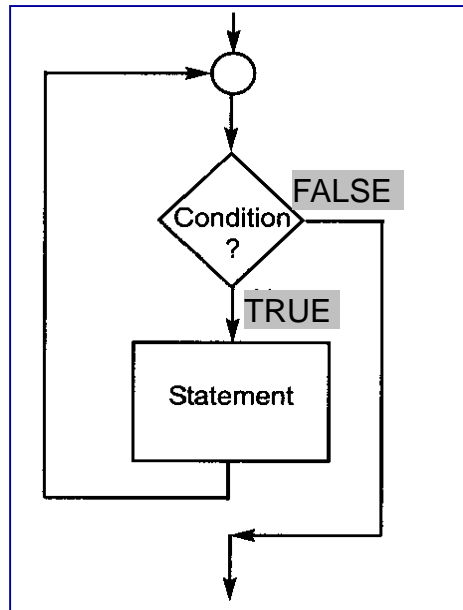
Control de Flujo de programa

- Repetición condicional:

```
do {  
  ...  
} while (condición==TRUE);
```

```
while (condición ==TRUE) {  
  ....  
}
```

¿ while (1) ?



Control de Flujo de programa

- Repetición:

```
for ( i=0; i<N; i++ )  
{  
    Se repite N veces...  
}
```

- Selección:

```
switch (x) {  
    case 1: ...  
        break;  
    case 2: ...  
        break;  
    default: ...  
        break;  
}
```

¡Está PROHIBIDO usar el GOTO!
No forma parte de las reglas de
programación estructurada.

Arquitectura de software: el main()

```
/* Inclusión de bibliotecas de código */
#include <xxxxx.h> // Registros del microcontrolador

/* Función main (principal):
es el punto de acceso al programa luego de encender el microcontrolador
o una operación de reset. */
int main (void)
{
    /* Setup:
    Parte de declaración de variables y configuración de de periféricos.
    Se ejecuta una única vez. */
    Sentencias...

    /* Loop:
    Este es el programa que se repetirá continuamente y realizará las
    tareas de procesamiento, por ejemplo, lectura de entradas, salidas. */
    while(1)
    {
        Sentencias...
    }

    /* Punto de finalización del programa:
    El microcontrolador no debe alcanzarlo ya que no existe un sistema
    operativo al cual retornar como ocurre en el caso de una PC al
    finalizar un programa */
    return 0;
}
```

Constantes en C

- Constantes

X+3;	←	Cte decimal 3
'F'	←	Caracter ASCII
"HOLA"	←	Cadena de caracteres ASCII

- Prefijos

0	←	octal
0b	←	binario
0x	←	hexadecimal

- Sufijos

U, L, UL, F

Ejemplo: **#define** CLK 16000000**UL**

Enumeraciones

Las Enumeraciones son un conjunto de constantes enteras que limitan el valor que una variable de este tipo puede tomar.

```
enum tag_name {  
    NOMBRE_0 [=CONST_0],  
    NOMBRE_1 [=CONST_1],  
    ...  
    NOMBRE_n [=CONST_n]  
};
```

- Ejemplo:

```
enum meses { ENE=1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC };
```

```
meses var1, var2 ;
```

← Declaro variables del tipo meses

```
var1 = FEB;
```

Operadores de C

- Aritméticos:

+ , - , * , / , % , ++ , --

- Asignación:

=

- Lógicos y de comparación:

&& , || , ! , < , >= , > , <= , != , ==

- Lógicos a nivel de bits:

& , | , ^ , ~ , << , >> ,

&= , |= , ^= , <<= , >>=

- Tamaño de:

`sizeof (var) ; sizeof (tipo de dato);`

Operadores Lógicos de bits en C

		AND	OR	EX-OR	Inverter
A	B	A&B	A B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

- Ejemplos:

```
unsigned char a=0x35, b=0x0F, c ;
```

```
c=a & b;
```

```
c=a | b;
```

```
c= a ^ b;
```

```
c= ~ a;
```

No confundir con los operadores de comparación: &&, ||, !=, !, <=, etc

A practicar

- 1) Realice una función en C, que reciba como parámetro un número N de 8 bits y lo envíe (a la salida estándar) de manera serie de a un bit cada 1s comenzando por el menos significativo.
- 2) Dado un número N (por la entrada estándar) de 8 bits sin signo, realice una función en C que devuelva al programa principal la suma de los números consecutivos de 1 hasta N. Analice el tipo de variable que retorna más conveniente.

Punteros

- Declaración:

char *p;



p es una variable puntero a un char

short *fp;



fp es una variable puntero a un int

void *ptr;



ptr es una variable puntero sin tipo definido aún

- Utilización:

char *p;

char a,b;

p=&a;



La dirección de a se asigna a p

*p=b;



b se asigna al contenido de la dirección p

- Ejemplo direccionamiento absoluto

unsigned char *reg_io;

reg_io=0x1010;








Se asigna la dirección de memoria de un registro para poder acceder al mismo

*reg_io=0xFF;

Punteros

- Operaciones con punteros:

<code>short*ptr;</code>		ptr es una variable puntero a un int (16bits)
<code>long *lptr;</code>		lptr es una variable puntero a un long (32bits)
<code>ptr = ptr +1;</code>		Sumar 1 al puntero equivale a moverse 2 bytes en la memoria de datos
<code>lptr = lptr +1;</code>		Sumar 1 al puntero equivale a moverse 4 bytes en la memoria de datos
<code>ptr++;</code> <code>--lptr;</code>		Uso de operadores unarios

Punteros: ejemplos

```
char c;
```

```
char *p;
```

```
c=*p++;    // asigna a c el valor apuntado por p, luego incrementa p
```

```
c=*++p;    // incrementa p, luego asigna a c el valor apuntado por p
```

```
c=++*p;    // incrementa el valor apuntado por p, luego lo asigna a c (p no se modifica)
```

```
c=(*p)++;  // asigna a c el valor apuntado por p, luego incrementa el valor apuntado por p (p no se modifica)
```

```
void * ptrv; //puntero a tipo de dato indefinido
```

```
char * p;
```

```
p = (char * ) ptrv;    //conversión al tipo de dato que corresponda
```

Punteros: ejemplos

Paso de parámetros a una función por referencia

- Definamos una función:

```
void swap (short *a , short *b)
```

```
{  
    short temp;  
    temp =*b;  
    *b=*a;  
    *a=temp;  
}
```

a y b son direcciones de memoria de operandos tipo short

el contenido de memoria b se guarda en temp

el contenido de a se guarda en b

Se copia temp donde apunta a

- utilización:

si v1=1000 y v2=500;

```
swap( &v1 , &v2 );
```

¿resultado?

Arreglos

- Declaración:

```
Short digitos[10];
```

```
char str[20];
```

- Inicialización:

```
Short tarray [5] = {12,15,27,56,-2};
```

```
char buffer[]="Hola";
```



String agrega '\0' al final

- Ejemplo:

```
char s[]="Esto es una prueba";
```

```
char i;
```

```
char *p;
```

```
void main (void)
```

```
{
```

```
    for (i=0 ; i<15 ; i++)
```

```
        putchar( s[i] );    //s[i]==*(s+i)
```

```
    p=s;
```

```
    for (i=0 ; i<15 ; i++)
```

```
        putchar(*p++);    //p se modifica
```

```
}
```

Arreglos: ejemplo multidimensional

```
#define TRUE 1
```

```
#define FALSE 0
```

```
char getkey (char * row , char * col);
```

← Prototipo de función

```
const char keys[4][3] = { '1', '2', '3',  
                          '4', '5', '6',  
                          '7', '8', '9',  
                          '*', '0', '#' };
```

← Definición de una matriz que representa el teclado

```
void main(void) {  
    char press, row , col;  
    while(1) {  
        if( ( press= getkey( &row , &col ) ) == TRUE);  
            putchar (keys[row][col]);  
    }  
}
```

← si se presionó una tecla la función getkey
modificará el valor de row y col para indicar cuál es
dicha tecla.

Estructuras

- Definición:

```
struct tag_name {  
    type memeber_1;  
    type memeber_2;  
    ...  
    type memeber_n;  
}
```

var1 y var2 son variables estructuras del tipo tag_name

var3 es un arreglo de 3 estructuras de ese tipo

ptr contendrá la dirección de una estructura del tipo especificado

- Declaración de variables:

```
struct tag_name var1 , var2 , var3[4] , *ptr;
```

- Utilización:

```
var1.member_1 = ... ;  
... = var2.member_2;
```

El operador “punto” permite acceder al campo interno de la estructura

Estructuras: ejemplo

```
struct Location {  
    shortx;  
    shorty;  
}
```

```
struct Part {  
    char name[20];  
    long int id;  
    struct Location bin;  
};
```

```
struct Part widget={"Window", 1, {0,0}};
```

```
short xpos, ypos;
```

Una estructura puede contener un conjunto heterogéneo de variables, por ejemplo otras estructuras

Inicialización entre llaves separando los campos internos por comas.

```
xpos=widget.bin.x;  
ypos=widget.bin.y;
```

```
widget.bin.x= xpos+10;  
widget.bin.y=ypos+23;
```

Estructuras y punteros

```
struct Part {  
    char name[20];  
    long int id;  
    struct Location bin;  
} widget[10];
```



Otra forma de declarar variables de ese
Tipo de estructuras

```
struct Part reg1, *ptr;
```

```
ptr=&reg1;
```



Si tengo la dirección de una estructura

```
ptr->id = 1234;
```



Puedo acceder a un campo interno
utilizado el operador “flecha”

Es equivalente a :

```
(*ptr).id = 1234;
```

Estructuras: typedef y campo de bits

```
typedef struct {  
    char name[20];  
    char age;  
    short room_num;  
} student;
```

← typedef define un nuevo tipo de dato

← Nombre del nuevo tipo de datos

```
student bob, Sally, John;
```

← Declaro variables de ese tipo

```
typedef struct {  
    unsigned short bit0:1;  
    unsigned short bit1:1;  
    ...  
    unsigned short bit15:1;  
} _PORT_IO;
```

← bit0 corresponde al LSB de un unsigned short

← bit1 corresponde al siguiente bit del mismo unsigned short

Es la manera de definir variables de tamaños menor a un byte.
Es útil para acceder a bits individuales de un registro de memoria

Directivas del pre-procesador

- Tarea para la práctica ...

#define

#include

#if, #elif, #else, #endif, #ifdef, #ifndef

#undef

#error

#warning

Cast: conversión entre distintos tipos de datos

- Las operaciones con tipos de datos diferentes requieren que el compilador realice Conversión Automática de tipo basado en reglas:

1-En expresiones con 2 o más operandos de diferentes tipos, el tipo dato mas corto es promovido al tipo de dato mas largo.

2-En expresiones con un operando signado y otro no signado, el signado se convierte a no signado

3-cuando un dato de tipo largo se asigna a uno de tipo corto el resultado es truncado

4-las constantes numéricas toman el valor más corto posible

- **Ejemplo 1:**

```
char a;
```

```
long b=1000;
```

```
a=b;    =>Warning! (b no es del mismo tipo que a)
```

Poner de forma explícita la conversión de tipo

```
a=(char) b; =>Sin Warning! (b no es del mismo tipo que a, pero se ha pedido una adecuación)
```


Cast: conversión entre distintos tipos de datos

- **Ejemplo 3:**

```
unsigned short a;  
signed short b = -1000;
```

```
a=b;    =>Sin Warning!
```

Sin embargo el programador puede dejar explícita la conversión

```
a=(unsigned short) b;
```

- **Ejemplo 4:**

```
if (a + b < 0) {...} //este tipo de condiciones debería evitarse
```

Conviene reemplazar por:

```
if( (b < 0) && (-b > a) ) {...}
```

- **Ejemplo 5:**

```
short z;  
short x=150;  
char y =63;
```

```
z= (y * 10 ) + x; =>sin warning !!!  
    Mult de 8-bit
```

=>Hacer una promoción explícita:

```
z=( (int) y * 10 ) + x;
```

O usar sufijos:

```
z=( y*10U ) + x;
```

Bibliografía

- *Los Microcontroladores AVR de ATMEL*. Felipe Espinoza. 2012. (CH3)
- *The AVR Microcontroller and Embedded System*. Mazidi. 2011. (CH7 y APENDICE D)
- Cualquier libro del Lenguaje C, apunte o recurso web que considere necesario.