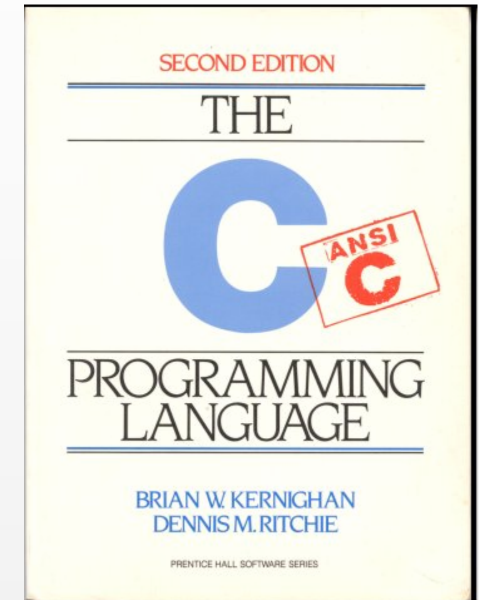
# Elementos de C para Sistemas Embebidos

Andrés Djordjalian <andres@indicart.com.ar>
Seminario de Sistemas Embebidos
Facultad de Ingeniería de la U.B.A.

# Estándares de C

- **Cada estándar es un "dialecto" diferente**
  - O sea, en grandes términos es lo mismo, pero existen diferencias que afectan la portabilidad.
- **"K&R C" (1978)**
  - La primera estandarización no fue institucional, sino que ocurrió cuando la comunidad adoptó como estándar la descripción hecha por Kernighan y Ritchie en su libro clásico de 1978.




- **"ANSI C" (o "ISO C" o "C90") (1990)**
  - Corresponde al estándar ISO/IEC 9899:1990
  - Es el más popular en la actualidad (el K&R C es obsoleto)
  - Es una descripción más exhaustiva que la del K&R C
- **"C99" (1999)**
  - Es la revisión de 1999 del estándar anterior
  - No todos los compiladores actuales lo soportan al 100%
  - Tiene elementos tomados de C++
    - …que los compiladores C/C++ suelen soportar aunque no sean 100% compatibles con C99

# Tamaño de los Tipos de Datos

- Los compiladores C tienen cierta libertad para elegir el tamaño de los tipos de datos comunes
  - Lo suelen hacer en base a consideraciones de eficiencia, que dependen de cada procesador
  - Tienen que respetar ciertos requisitos:
    1. `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)`
    2. `char` por lo menos de tamaño suficiente para el conjunto de caracteres básico
    3. `short` al menos de 16 bits
    4. `long` al menos de 32 bits
- En programación de bajo nivel, a veces queremos explicitar el tamaño exacto del dato
  - Por ejemplos, en casos como estos:
    - Entrada/Salida
    - Optimización de la utilización de memoria
    - Rutinas en Assembly embebidas
  - ¿Definir siempre un tamaño exacto? ¿O sólo a veces?
    - Es un tema polémico.

# Tipos de Datos de Tamaño Fijo

- El C99 incluye los tipos de datos:
  - `int8_t, int16_t, int32_t, int64_t` (con signo)
  - `uint8_t, uint16_t, uint32_t, uint64_t` (sin signo)
- `int8_t` es un entero de 8 bits incluyendo signo, `int16_t` es uno de 16 bits, etc.
- Estas definiciones están en un header llamado `stdint.h`
- Ejemplo:

```
#include <stdint.h>

int16_t g_posicion = 0;

int16_t leer_posicion(void) {
        unsigned int intentos = 0;
        etcétera…
```

- Si el compilador no es C99 compatible, podemos usar los mismos tipos definiéndolos con **`typedef`**

# typedef

- **`typedef`** le da un nombre al tipo de dato que indiquemos (que puede ser compuesto):

```
typedef <tipo de dato> <nombre>
```

- Ejemplos:

```
typedef signed char     int8_t;
typedef unsigned char   uint8_t;
```

- Así, podemos definir esos tipos en **`stdint.h`**, y usarlos aunque nuestro compilador no sea C99 compatible
  - Si **`stdint.h`** no existe, lo creamos
  - Pero muchos compiladores para embebidos ya lo traen
  - Usamos tipos de datos estándar de los cuales sabemos la longitud para el procesador en cuestión
    - Como con **`char`** en el ejemplo de arriba
- Por otro lado, aprovechen las sentencias **`typedef`** para clarificar el código

# Conversión Automática de Tipos

- C permite realizar operaciones entre variables de diferente tamaño, o combinando con y sin signo

Ejemplo:

```
unsigned char a = 200;
int b = -20;

a = a+b;
```

- Los compiladores generalmente cambian cada tipo a uno de más bits que incluye todos los valores posibles del original
  - En el ejemplo anterior, **`a`** es convertido a **`int`** durante la operación, y el resultado es pasado de vuelta a **`unsigned char`**, quedando igual a 180, como resulta lógico
- Gracias a esto, el programador frecuentemente ni se da cuenta de que hay una conversión de tipos
  - Es más, muchos compiladores convierten tipos aunque no sea necesario, para simplificar la compilación

# Conversión Automática de Tipos

- Sin embargo, las reglas de C son más complejas que lo mostrado en la diapositiva anterior, porque tienen en cuenta la performance
- Ej.: Si **`a`** es **`unsigned int`** y la aritmética en **`long`** es ineficiente en este procesador, ¿qué conviene hacer?
  1. ¿Convertir todo a `long`?
  2. ¿O usar `int` a pesar de perderse rango?
  3. ¿O usar `unsigned int` a pesar de perderse el signo de `b`?

Ejemplo:

```
unsigned int a = 200;
int b = -20;

a = a+b;
```

- Los compiladores generalmente optan por la número 3, pero estas situaciones no están bien definidas
  - Por eso, traen **riesgos de errores y problemas de portabilidad**

# Conversión Automática de Tipos

- Recomendaciónes:

1. No mezclar valores con signo con valores sin signo
   - Si se lo hace, prescindir de la conversión automática lo más que se pueda
     - Por ejemplo, si **a** es **unsigned** y **b** es **signed**, en lugar de

       ```
       if (a+b < 0) { … };
       ```

       Escribir:

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

2. No hacer asignaciones que puedan perder información
   - En el primer ejemplo, además de no cumplir la regla anterior, estábamos sumando un **int** y poniendo el resultado en un **char**. Evitemos eso.

Ejemplo:

```
unsigned char a = 200;
int b = -20;

a = a+b;
```

# Operaciones Sobre Bits

- Para operar sobre bits individuales, escribamos las constantes en hexadecimal
  - Porque cada dígito hexa corresponde a 4 bits y se hace fácil de leer
    - Ej., 0xC8 = 11001000b 0xC8 = 11001000b
- Operadores de a bits:
  - AND: `&` • OR: `|` • NOT: `~` • XOR: `^`
  - Operan "bit a bit", así que no confundirlas con las operaciones lógicas `&&`, `||` y `!`
- Testeo de bits: `if (0x0020 & reg) { … };`
- Encendido de bits: `reg |= 0x0003;`
- Apagado de bits: `reg &= 0x00FF;`
- "Toggling" de bits: `reg ^= 0x0001;`
- Desplazamiento: `<< n` y `>> n`

# Punto Flotante

- En muchas aplicaciones (científicas, de procesamiento de señales, etc.) necesitamos representar números reales con:
  - **Rango amplio**
  - **Error de representación bajo en términos relativos**
  - Parte **fraccionaria** (no entera)
  - Sin que se necesiten muchos **bits** para representarlos
- La manera típica de hacerlo es mediante algún formato de punto flotante (o *floating point* o *coma flotante)*
- Consiste en descomponer el número así:

$$\text{número} = \text{mantisa} \times \text{base}^{\text{exponente}}$$

- La base y la posición de la coma en la mantisa están establecidos por el sistema de punto flotante que se haya elegido
  - Noten que, por lo tanto, sólo se necesitan guardar los dígitos de la mantisa y los del exponente, junto con sus respectivos signos, para representar el número en memoria

# Punto Flotante

número = mantisa x base $^{exponente}$

- Ej:
  - 18,375 = 1,8375 x $10^{1}$
  - -0,07514 = -7,5140 x $10^{-2}$
  - 982312014 ≅ 9,8231 x $10^{8}$
  - -0,00000000415147 ≅ -4,1514 x $10^{-9}$
- Noten que no usamos más que cinco dígitos para la mantisa y uno para el exponente (más sendos signos) y sin embargo logramos gran rango y bajo error relativo
- Podíamos haber establecido otra posición para la coma, por ej.:

  18,375 = 183,75 x $10^{-1}$ ≅ 0,0184 x $10^{-3}$ ≅ etc.

  - Pero lo típico es usar un sistema ***normalizado*** con un y sólo un dígito entero

# Punto Flotante

- El ejemplo es en base 10. En un circuito digital normalmente se usa base igual a 2
- ¿Cómo se escribe 18,375 en binario?
  - $18{,}375_{10} = 10010{,}011_2$
    - Porque las posiciones a partir de la coma valen $2^{-1}$, $2^{-2}$, $2^{-3}$, etc.
      - O sea, 0,5; 0,25; 0,125; etc.
- Ejemplos en binario y base igual a 2:
  - **18,375 = 1,0010011b x $2^{100b}$**
  - **-0,07514 ≅ -1,1110101b x $2^{-100b}$**
  - **982312014 ≅ 1,1101010b x $2^{11101b}$**
  - **-0,00000000415147 ≅ -1,0001110b x $2^{-11100b}$**
- Noten que el 1 entero es redundante (sólo el 0 no lo tiene), así que no hace falta guardarlo
  - …siempre que exista alguna convención para guardar un cero
  - Cuando no se lo guarda, a ese 1 se le llama *implícito* u *oculto*

# Estándar IEEE 754

- Se usa muy extensivamente. Define varios tipos de punto flotante, principalmente:

1. Simple precisión
   - Normalmente es el típico tipo **`float`** de lenguaje C
   - Usa 32 bits en total para cada número:
     - 1 para el signo
     - 23 para la mantisa (sin contar el 1 implícito)
     - 8 para el exponente (en lugar de ser signado tiene un *bias* de 127)
2. Doble precisión
   - Normalmente es el tipo **`double`** de lenguaje C
   - Usa 64 bits:
     - 1 para el signo
     - 52 para la mantisa (sin contar el 1 implícito)
     - 11 para el exponente (con *bias* igual a 1023)

- Los dos son normalizados y usan base igual a 2

# Punto Flotante

- Las operaciones con punto flotante no son tan básicas como las de enteros. Ej.:
  - Para sumar, se alinean las mantisas, se suman, se toma el exponente más grande, y se normaliza el resultado
  - Para multiplicar, se multiplican las mantisas, suman los exponentes, y normaliza el resultado
  - Etcétera
- Cuando un programador necesita números con parte fraccionaria, frecuentemente los define como `float` (o algo similar) y listo
  - Se logra un error de representación pequeño para un rango muy grande, así que sólo excepcionalmente hay que analizar si es lo suficientemente bueno (ej., aplicaciones científicas)
  - El compilador y las librerías implementan todo, usando operaciones enteras o de la unidad de punto flotante (*floating-point unit* o FPU) si es que se cuenta con este hardware

# Números no enteros en Sist. Emb.

- Sin embargo, muchas de esas veces no se requiere el amplio rango y el bajo error relativo del punto fijo. Se paga así un precio que, en sistemas embebidos, puede ser caro:
  - Las operaciones de punto flotante **tardan más**, particularmente si no se cuenta con FPU, que es algo frecuente en S.E.
  - …y **consumen más energía**
  - Si se necesitan guardar muchos números, los 32 bits que ocupa cada `float` pueden resultar en un requisito excesivo de **memoria**
- En esos casos, puede ser mejor usar otro sistema para representar números con decimales: punto fijo
- La dificultad principal de usar punto fijo es asegurarse de que su error de representación sea aceptable
  - Por eso, es frecuente empezar con punto flotante y pasar a punto fijo recién en una etapa de optimización
    - …si es que se ve justificado el trabajo extra
  - Cuidado, el error de representación puede no ser problemático en los resultados finales pero sí en los intermedios
    - Por eso, no es fácil comprobar si es aceptable el punto fijo

# Punto Fijo

- Se trata, básicamente, de usar una representación de enteros pero interponiendo un factor de escala
  - Ej.: Si convenimos que el factor de escala sea igual a 1000, para representar 12,34 guardamos 12340 en una variable entera
- Para calcular granularidad y rango, dividimos las de la representación entera por el factor de escala
- Ej., si usamos una representación entera signada de 16 bits (o sea, `int16_t`) y un factor de escala igual a 2048:
  - **Rango de `int16_t`:** desde -32768 hasta 32767
    - Recuerden que, en complemento a dos, hay un número negativo más (que los positivos) porque el complemento a uno del número cero se aprovecha como número negativo (o sea, -1)
  - **Rango de nuestro sistema: desde -16 hasta 15,9995**
    - Porque -32768/2048 = -16 y 32767/2048 ≈ 15,9995
  - **(Granularidad de `int16_t`:** 1)
  - **Granularidad de nuestro sistema: 1/2048 ≈ 0,0005**

# Punto Fijo: Aritmética

- Para multiplicar y dividir números de punto fijo por enteros, usamos las operaciones sobre enteros. Ej.:

```
typedef   int32_t  fxp24_8_t;

int16_t    a;
fxp24_8_t  x,y;

y = x * a;
x /= a;   //es igual que x=x/a
```

- Para sumar y restar en punto fijo, *si los factores de escala son iguales*, usamos la suma y resta de enteros. Ej:

```
fxp24_8_t      x,y,z;

x += y;
z = x - y;
```

# Punto Fijo: Cambios de Escala

- En algunos casos necesitamos cambiar un número de un factor de escala a otro. Por ej.:
  - Para igualar escalas **antes de sumar o restar**
  - Para extraer la **parte entera**
  - Para representar un entero en un sistema de punto fijo
- Si las escalas son potencias de dos, alcanza con hacer un simple desplazamiento (*shift*)
  - También funciona con números negativos porque, en C, el desplazamiento es de tipo *aritmético*
    - O sea que cuando un número signado se desplaza hacia la derecha, el bit que se ingresa por la izquierda es igual al bit más significativo del número desplazado
      - Recién en C99 esto es estándar, pero es común desde antes
  - Un caso particular es convertir desde o hasta enteros (porque estos tienen factor de escala = $2^0$):

```
int16_t     a;
fxp24_8_t   x;

y = x + (a<<8);
```

# Punto Fijo: Cambios de Escala

- Si hay riesgo de overflow, mejor chequear antes
  - Ej., (si **`a`** es **`int32_t`**):

```
if (-0x800000<=a && a<=0x7fffff)
     y = x + (a<<8);
else  …reportar el error, o hacer lo que haya que hacer
```

- Se puede elegir redondear o truncar
  - Ej., para extraer la parte entera de un número en punto fijo:

*Para truncar:*

```
a = x >> 8;
```

*Recordemos que, por ej., trunc(-0.1) = -1*

*…y para redondear:*

```
a = (x+128) >> 8;
```

- Si las escalas no son potencia de dos, habrá que recurrir a una multiplicación y/o división
  - Eso hace deseable que sean potencia de dos, pero **no es imprescindible**
    - En particular con ISAs que cuentan con modos eficientes de multiplicar por constantes, como por ejemplo ARM

# Punto Fijo: Aritmética

- Es común querer multiplicar o dividir dos números de punto fijo de un mismo tipo, y tener el resultado en el mismo tipo
- Para la multiplicación: si usamos la multiplicación de enteros, el factor de escala del resultado es el cuadrado del deseado
  - Así que, usualmente, se convierte y redondea o trunca después
- Para la división: si usamos la división de enteros, los factores de escala se cancelan
  - Así que podríamos multiplicar por el factor de escala (o desplazar) después de hacer la división
    - Pero estaríamos poniendo ceros en reemplazo de bits que habían sido truncados, perdiendo precisión
      - Por eso, es mejor multiplicar por el factor de escala **antes** de hacer la división (es decir, multiplicar el dividendo)
- En estos casos, no pasemos por alto el riesgo de *overflow*

# Punto Fijo: Actividad

**Se nos pide un sistema de punto fijo para representar números cuya precisión es de un milésimo, con un rango de al menos ±20**

A. Definir uno que no use más de 16 bits para cada número, y en el que pueda convertirse desde y hacia enteros con un desplazamiento (*shift*)

B. ¿Qué tipo (de dato) de C99 usamos?

C. ¿Cuál es la granularidad? ¿Y el rango?

D. Pasar el siguiente pseudocódigo a lenguaje C:
   1. Usar un `typedef` para crear el tipo de punto fijo
   2. Declarar dos variables de ese tipo (llamadas "`foo`" y "`bar`") y un `uint8_t` (llamado "`a`") que sea igual a 2
   3. Codificar, como corresponde, lo siguiente:

```
foo = a
bar = 10.5
foo = foo + bar/3
a = round(foo)
```

# Declaración de Punteros

- El compilador lee lo siguiente:

```
uint16_t  *p_uint16;
```

  De esta forma:
  1. `uint16_t`
  2. `*`
  3. `p_uint16`

- Por eso, esto funciona:

```
uint16_t*  p_uint16;
```

- Pero no lo hagan, porque es peligroso si declaran varios punteros en la misma línea:

```
uint16_t*  p_uint16, p_otro;
```

  (`p_otro` queda declarado como `uint16_t`, no como puntero)

# const

- El modificador **`const`** indica que el código no debe cambiar el valor de la variable declarada
  - Ej., `int const foo = 10;`
  - Si por error escribimos código que la modifica, el compilador chilla, y podemos repararlo sin que cause más problemas
    - Por lo tanto, los **`const`** no son imprescindibles, pero **hacen que la verificación estática sea más potente**
      - Incluyendo la verificación estática simple que hace el compilador
- Usémoslo en argumentos que son pasados mediante punteros pero que la función no debe modificar
  - Ej., `char *strcpy(char *dest, char const *fuente)`
- Pero no lo usemos donde no haga falta
  - Ej., en argumentos pasados por valor, como:

    ```
    int son_iguales(char A, char B)
    ```

    Porque no hay problema si la función cambia el valor de **`A`** o **`B`**, dado que son copias de uso local.

# `const` y Punteros

- ¿Cuál es la diferencia entre estos dos?

```
const char *p;
char *const p;
```

- Respuesta:
  - El primer **`p`** es un puntero a un carácter constante
  - El segundo **`p`** es un puntero constante a un carácter
    - Para entenderlo, ayuda leer la declaración “de derecha a izquierda” en inglés
      - Por eso, se recomienda escribir `char const *p;` en lugar de la primera de las dos líneas (para el compilador es lo mismo)

- ¿Cómo se interpreta lo siguiente?

```
typedef char *TA;
const TA p;
```

- Respuesta: **`p`** es un puntero constante a un carácter
  - Mirarlo como que, en este caso, **`const`** quedó entre el declarador **`p`** y el especificador **`*`**, tal como pasaba en el segundo caso de arriba
    - Si la segunda línea se escribía `TA const p;` quedaba todo más claro

# Ejemplo

- Prototipo de una rutina de interrupción:

```
void IRQ_handler();
```

- Para instalar la rutina en la tabla de vectores de interrupción (*interrupt vector table*, o IVT), quisiéramos poder hacer algo simple y claro como:

```
IVT[6] = IRQ_handler;
```

- Para poder hacerlo, previamente declaramos:

```
typedef void (*pointer_to_handler)();
pointer_to_handler *const IVT =
        (pointer_to_handler *) 0x20;
```

  - La primera línea define a `pointer_to_handler` como un puntero a una función del tipo de las rutinas de interrupción.
    - Notar que, si no la hubiéramos rodeado de paréntesis, habríamos declarado una función que devuelve un puntero.
  - La segunda línea declara a `IVT` como un puntero constante, de valor 0x20 (o sea, el comienzo de la tabla, de acuerdo con el micro), siendo esa una posición que aloja un puntero capaz de apuntar a una rutina de interrupción.
    - `(pointer_to_handler *)` es un *type cast* para convertir 0x20 en un puntero del mismo tipo que `IVT`.
  - `IVT[6] = IRQ_handler;` ahora funciona, porque:
    - `IVT[0]` es lo mismo que `*IVT`, o sea, el contenido de la primera posición de la tabla, así que `IVT[6]` es el contenido del sexto puntero de la tabla.
    - El nombre de la función, sin los paréntesis, equivale a un puntero a la función.

# Números Mágicos

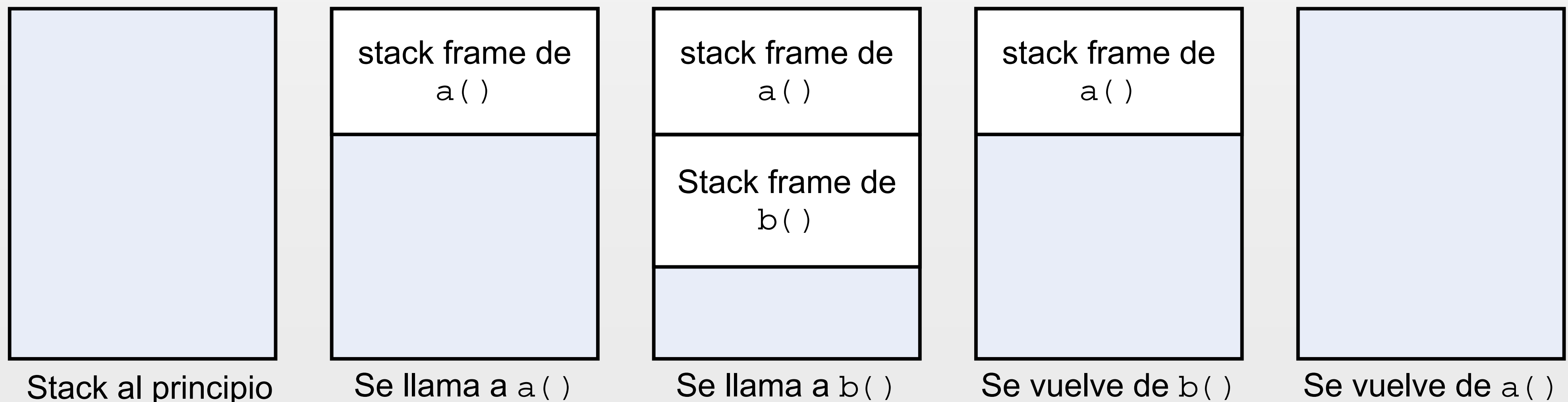
- A veces queremos escribir cosas como **`IVT[6]`** o **`estado = 8;`**
  - Pero ¿qué significan esos 6 y 8?
  - A estos números, que no se entiende de dónde salieron, se los llama *números mágicos*. Conviene evitarlos
- Usemos enumeraciones y constantes en lugar de números mágicos
  - Así les indicamos un significado y queda todo más legible. Ej.:

```
enum numero_de_interrupcion {
    in_comienzo, in_reset = in_comienzo,
    in_undef, in_SWI, in_pref_abort,
    in_data_abort, in_reserved, in_IRQ, in_FIQ,
    in_fin
};
```

  - Así, se puede escribir: `IVT[in_IRQ] = IRQ_handler;`
  - **`in_comienzo`** e **`in_fin`** están para poder hacer:
    `for (int n = in_comienzo; n != in_fin; n++) { … }`

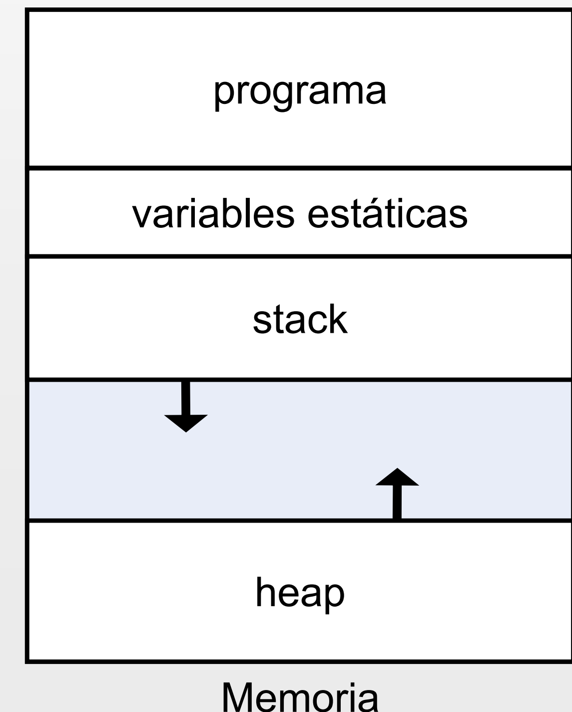
# Asignación de Memoria

- En la mayoría de las plataformas, al llamar a una función, se reserva una porción de memoria en el stack (llamada *stack frame*) para guardar:
  - La dirección de retorno
  - Los parámetros que se le pasan a la función
  - Las variables de la función que sólo viven mientras se ejecuta esta
    - A estas se las llama ***variables automáticas***



# Asignación de Memoria

- Los programas tienen más formas de acceder a RAM, aparte de las variables automáticas:
  - Las variables *estáticas* (que viven durante toda la ejecución del programa)
    - Estas son: las **globales**, y las declaradas con el modificador **`static`**, que vamos a ver en la próxima diapositiva.
  - Memoria asignada dinámicamente
    - Se la solicita y devuelve explícitamente, mediante **`malloc()`** y **`free()`**
    - Esta memoria conforma el ***heap***
      - (pronúnciese “jiip”)
- Normalmente, el stack crece en en sentido, y el heap en otro
  - Pero el *heap* puede estar fragmentado, porque no es una pila
    - O sea, se pueden liberar porciones que no hayan sido las últimas en ser asignadas
- Es lento pedir memoria del heap
  - A veces conviene que el programa la pida de a porciones grandes y la reparta él mismo



# static

- El modificador `static` hace que la variable declarada sea estática
  - Por lo tanto, **conserva su valor aunque se haya salido de la rutina donde estaba declarada**
  - Ej.,

```
uint16_t cuenta()
{
    static uint16_t foo = 0;

    return ++foo;
};
```

  - Notar que `foo` no deja de ser **local**
    - O sea, no puede ser accedida fuera de `cuenta()`
  - La inicialización de `foo` ocurre sólo una vez
- Si se declara como `static` una variable global, su ámbito se restringe al archivo (del código fuente) donde está declarada

# volatile

- El modificador `volatile` le indica al compilador que la variable declarada puede sufrir modificaciones que no estén explícitas en el código
  - Ej.: Registros de entrada • Variables globales modificadas por rutinas de interrupción
- De esa forma, el compilador evita hacer optimizaciones que podrían ser erróneas
- Ej., Supongamos que la variable `clock` se incrementa regular y automáticamente, y la usamos en este código:

```
uint32_t  volatile  clock;

foo = clock;
while (clock < foo+10) { … };
```

Si `clock` no fuera declarada como `volatile,` el compilador podría asumir que la condición del `while` ocurre siempre
  - …y entonces no generar código que la calcule, ejecutándose siempre el bloque del `while`

# MISRA C

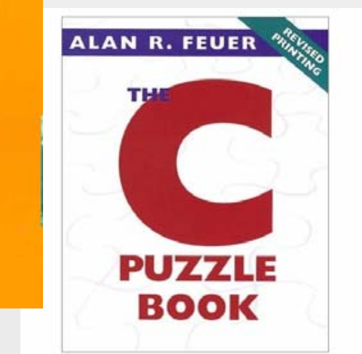
- Estándar para la producción de código C confiable y portable, en el contexto de sistemas embebidos
- Consiste de un conjunto de reglas
  - Algunas obligatorias, otras sugeridas
- Desarrollado por MISRA (Motor Industry Software Reliability Association)
- Originalmente pensado para la industria automotriz
  - Pero actualmente se usa en la aeroespacial, telecomunicaciones, electromedicina, defensa, ferrocarriles y otras
- Lanzado en 1998 (MISRA-C:1998)
  - La última versión es MISRA-C:2004
- Artículo introductorio:
  - http://www.eetimes.com/discussion/other/4023981/Introduction-to-MISRA-C

# Reglas de MISRA C

- Ejemplos:
  - Regla 30 (requerida): A todas las variables automáticas se les debe haber asignado un valor antes de leerlas
  - Regla 49 (sugerida): La comparación de un valor contra cero debe ser hecha explícitamente, a menos que el operando sea realmente booleano
  - Regla 50 (requerida): Cuando las variables de punto flotante sean comparadas entre sí, no se deben testear igualdades o desigualdades que sean *exactas*
  - Regla 104 (requerida): Los punteros a funciones deben ser constantes
- Pueden ver un resumen de todas (en inglés) en: http://computing.unn.ac.uk/staff/cgam1/teaching/0703/misra%20rules.pdf

# Conclusiones

- Conociendo bien C, en particular su sistema de declaración de variables, se puede escribir código que sea:
  - Más **legible**
  - Menos propenso a **errores**
  - Más **portable**
- Tengan en cuenta al punto fijo, para ganar en velocidad y utilización de la memoria
- Se puede agregar mucho a lo que dicen estas diapositivas
  - Consideren leer un libro sobre C intermedio o avanzado.




- Un sitio para visitar:
  - http://www.embeddedgurus.net/barr-code/