Inconvenientes de los tipos de datos básicos del lenguaje C

Diseño Basado en Microprocesadores

Víctor Manuel Sánchez Corbacho

Dpto. de Automática, Electrónica, Arquitectura y Redes de Computadores

2016

Contenido

Tipos de datos básicos

- Tipos de datos enteros.

Tipo	¿Con signo?	Variante sin signo	Variante con signo
`char`	**Depende**	`unsigned char`	`signed char`
`short`	Sí	`unsigned short`	`signed short`
`int`	Sí	`unsigned int`	`signed int`
`long`	Sí	`unsigned long`	`signed long`
`long long`	Sí	`unsigned long long`	`signed long long`

- El tipo `char` tiene signo o no según el compilador.
- Tipos de datos flotantes.

Tipo	
`float`	(Siempre con signo)
`double`	(Siempre con signo)

En algunas arquitecturas existe el tipo `long double`.

Tamaños de los tipos de datos

- Uno de los inconveniente del lenguaje C:
  - Los tamaños concretos de los tipos de datos no están estandarizados.
- Sólo se garantiza que
  - (tamaño `char`) $\leq$ (tamaño `short`) $\leq$ (tamaño `int`) $\leq$ (tamaño `long`) $\leq$ (tamaño `long long`)
  - (tamaño `char`) $\leq$ (tamaño `float`) $\leq$ (tamaño `double`) $\leq$ (tamaño `long double`)
  - Tamaño `char` $\geq$ 8 bits.
  - Tamaño `int` $\geq$ 16 bits.
  - Tamaño `long` $\geq$ 32 bits.
  - Tamaño `long long` $\geq$ 64 bits.
- La razón es que se crearon muchos compiladores de C diferentes antes de la publicación de un estándar internacional. El estándar tuvo que *acomodar* a todos.
- Hay que consultar la documentación del compilador usado.

Tamaños de tipos básicos en distintos compiladores

- Tamaños típicos de tipos básicos en compiladores de 16 bits y 32 bits:

Tipo	Compilador de 16 bits	Compilador de 32 bits
`char`	8 **¿Con signo?**	8 **¿Con signo?**
`short`	16	16
`int`	**16**	**32**
`long`	32	32
`long long`	64	64
`float`	32	32
`double`	64	64

- Además, hay tipos enteros del mismo tamaño en cada compilador.
- Todo esto puede provocar:
  - Errores al elegir el tipo correcto para las variables.
  - Errores al analizar el funcionamiento de un programa.
  - Problemas al portar un programa a otra arquitectura.

Tipos de datos de tamaños concretos

- Para evitar estos problemas **es mejor no usar los tipos de datos básicos de C**.
- En su lugar, pueden usarse los tipo enteros definidos en `stdint.h`:

Tipo	Bits	Signo	Rango
`int8_t`	8	Sí	–128 a 127
`uint8_t`	8	No	0 a 255
`int16_t`	16	Sí	–32768 a 32767
`uint16_t`	16	No	0 a 65535
`int32_t`	32	Sí	–2147483648 a 2147483647
`uint32_t`	32	No	0 a 4294967295
`int64_t`	64	Sí	-2^{32} a $2^{32}-1$
`uint64_t`	64	No	$2^{64}-1$

- El tipo `char` sigue usándose, pero sólo para almacenar caracteres ASCII.
  - Necesario por compatibilidad con código ajeno (ej.: la biblioteca estándar usa `char`).
  - Para datos de 8 bits que no sean caracteres ASCII se usará `int8_t` o `uint8_t`.

El fichero `stdint.h` en diferentes compiladores

- Extracto del fichero `stdint.h` en un compilador de 32 bits.

```
typedef signed char            int8_t;
typedef signed short int       int16_t;
typedef signed int             int32_t;
typedef signed long long int   int64_t;

typedef unsigned char          uint8_t;
typedef unsigned short int     uint16_t;
typedef unsigned int           uint32_t;
typedef unsigned long long int uint64_t;
```

- Extracto del fichero `stdint.h` en un compilador de 16 bits.

```
typedef signed char            int8_t;
typedef signed int             int16_t;
typedef signed long int        int32_t;
typedef signed long long int   int64_t;

typedef unsigned char          uint8_t;
typedef unsigned int           uint16_t;
typedef unsigned long int      uint32_t;
typedef unsigned long long int uint64_t;
```

Definiciones de `float32_t` y `float64_t`

- Los tipos `float` y `double` suelen coincidir con los tipos IEEE-754 de precisión simple (32 bits) y doble (64 bits).
- Pero a veces se prefiere definir los tipos `float32_t` y `float64_t` para que queden claros sus tamaños.

```
typedef float  float32_t;
typedef double float64_t;
```

Tipo booleano

- El lenguaje C no tiene un tipo de dato específico para valores booleanos.
- Los valores booleanos son representados mediante valores enteros:
  - Un valor entero **igual a cero** es **falso**.
  - Cualquier valor entero **distinto de cero** es **verdadero**.
- Usar enteros para codificar booleanos puede causar dudas al interpretar el significado de una variable, parámetro o valor de retorno de una función.

Ejemplos

```
int buscar(int dato_a_buscar, int *array, unsigned tamano_array);
```

¿Significa que `buscar` retorna `0` si no se encuentra y `1` si se encuentra, u otra cosa?

```
unsigned int estado_caldera;
```

¿Significa que si `estado_caldera == 0` está “apagada” y si es `!= 0` está “encendida” o hay más opciones?

Tipo bool_t

- Para evitar ambigüedades es mejor usar un tipo específico para denotar valores booleanos.
- Una forma de definirlo es **(hay otras formas de hacerlo)**:

```
typedef unsigned char bool_t;
#define FALSE 0
#define TRUE  1
```

- Algunos prefieren llamarlo `bool` en lugar de `bool_t`.
- Ahora es fácil reconocer valores booleanos:

```
bool_t buscar(int dato_a_buscar, int *array, unsigned tamano_array);
bool_t estado_caldera;
```

Asignar y comprobar un `bool_t`

Asignación

```
bool_t estado_caldera = FALSE;
...
estado_caldera = TRUE;
```

Comprobación

```
if (estado_caldera) ...

if (!estado_caldera) ...
```

No hacerlo así:

```
if (estado_caldera == TRUE) ...
```

Porque no está garantizado que `estado_caldera` sólo valga `0` o `1`.