Tipos aritméticos en C++

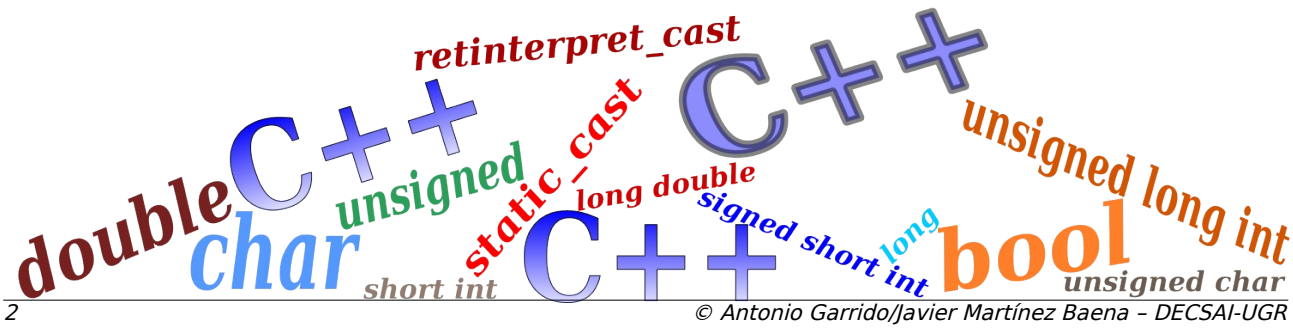
Antonio Garrido / Javier Martínez Baena

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Metodología de la Programación

Grado en Ingeniería Informática

Índice de contenido



1. Objetivos

El objetivo de esta práctica es que el alumno conozca con más precisión las características de los tipos aritméticos en C++. Recordemos que estos tipos se clasifican en:

1. Tipos integrales: booleanos, carácter y enteros.
2. Tipos en coma flotante.

En esta práctica profundizaremos en las características de estos tipos de datos, completando los conocimientos que se han visto en clase de teoría. El alumno deberá revisarlo y comprender los distintos conceptos y detalles que se muestran. De esta forma, podrá entender mejor el comportamiento de sus programas. Después de realizar esta práctica, el alumno deberá haber asimilado:

1. Los distintos tipos y sus relaciones, teniendo en cuenta sus distintos tamaños y precisiones. Aunque no será habitual usar *sizeof* y *numeric_limits*, el alumno debe saber que existen métodos que nos permiten consultar las características de los distintos tipos.
2. La aritmética en punto flotante implica una aproximación de la recta real y, por tanto, el cálculo con valores aproximados.
3. Existen los valores especiales para un número no representable (*NaN)* e infinito (*Inf*) para los tipos en punto flotante. Aunque no los usaremos explícitamente, el alumno debe conocer que pueden aparecer en sus programas y entender a qué se refieren.
4. El lenguaje ofrece un grupo de herramientas que permiten el formateo de la salida.
5. Es importante conocer en qué consisten las conversiones. De especial interés son las que involucran a booleanos y caracteres, pues son tipos que, en principio, no parecen relacionados con el resto de aritméticos.
6. Se puede usar la conversión explícita para que el compilador haga, exactamente, lo que el programador desea.

2. Rango y precisión

El rango y precisión de un tipo depende de la plataforma en la que se esté desarrollando. El estándar C++ no garantiza más que los mínimos siguientes:

Tipo	*Tamaño mínimo*
char	1
short int	2
int	2
long int	4
float	4
double	8
long double	8

Además, garantiza que esos tamaños están "ordenados". Por ejemplo, el tamaño de un tipo *float* no puede ser mayor que el de un tipo *double*. Se podrían listar todos los tipos de datos: **`signed char, char, unsigned char, signed short int, signed int, signed long int, unsigned short int, unsigned int, unsigned long int, float, double, long double,`** incluyendo modificadores de signo, aunque un tipo con signo o sin él ocupa lo mismo. Observe que en el caso de los enteros, el no incluir modificador equivale a añadir "**`signed`**".

2.1. Tamaños

El operador *sizeof* del lenguaje C++ nos permite obtener el tamaño de un determinado tipo u objeto. En un programa podemos escribir este operador con el nombre de un tipo entre paréntesis para obtener el tamaño[1] de dicho tipo.

> ***Ejercicio** 1: Operador sizeof*
>
> Escriba un programa que imprima, ordenadamente, los tamaños de los distintos tipos para la plataforma en la que está trabajando.

Puede comprobar con una simple modificación que, si a los tipos que ha escrito le antepone alguna modificación de signo (*signed* o *unsigned*), obtiene exactamente los mismos tamaños.

2.2. Límites numéricos

Si queremos que nuestro programa tenga información concreta sobre los límites exactos que presentan los distintos tipos, podemos usar la biblioteca estándar de C++.

En primer lugar, debemos tener en cuenta que en C++ también tenemos disponibles las mismas constantes que se usan en C. Así, los archivos de cabecera *limits.h* y *float.h* están disponibles en C++ como *climits* y *cfloat*.

En C++ podemos usar la plantilla *numeric_limits* para obtener esta información, previa inclusión del archivo de cabecera *limits*. Aunque entender cómo funciona esta plantilla es un tema avanzado, podemos usarla para implementar un programa simple que nos informe de algunos valores usados en nuestro sistema. Para ello, tenga en cuenta que la forma en que podemos usarla tiene el formato:

```
numeric_limits<TIPO>::MIEMBRO
```

donde *TIPO* es el nombre del tipo que queremos consultar y *MIEMBRO* se refiere a la información que queremos obtener. Algunos miembros que podríamos consultar son:

1. Para tipos entero y en coma flotante los valores del mínimo y máximo del rango: min(), max().
2. Para tipos en coma flotante la diferencia entre uno y el menor valor representable mayor que uno: epsilon().

> ***Ejercicio** 2: Límites numéricos*
>
> Escriba un programa que imprima los valores de los miembros que se han listado para los distintos tipos: **`signed short int, signed int, signed long int, unsigned short int, unsigned int, unsigned long int, float, double, long double.`**

> ***Ejercicio** 3: Rango limitado de enteros*
>
> Escriba un programa que declare un objeto de tipo entero, le asigne el valor máximo, le sume uno, y finalmente lo imprima. A continuación, le asigne el mínimo, le reste uno, y lo imprima. ¿Qué resultados obtiene?

1 *El tamaño corresponde al número de "char" que ocupa. Es decir, el tamaño de char es 1 y el del resto un múltiplo de lo que ocupa éste. En nuestro caso, en el que char ocupa un byte, podemos pensar en número de bytes.*

3. Aritmética en punto flotante

Analiza el siguiente programa:

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    char son_iguales;
    double x;
    cout << "Dime un número y comprobaré si se cumple "
            "que la raíz cuadrada al cuadrado da el número: ";
    cin >> x;
    son_iguales = (sqrt(x)*sqrt(x)==x) ? 'S' : 'N';
    cout << "Son iguales: " << son_iguales << endl;
}
```

Para ello, compruebe el resultado de su ejecución para distintos números. Por ejemplo, puede comprobarlo para el número 2 y el 4.

> ***Ejercicio** 4: Precisión de números en coma flotante*
>
> En el programa anterior es posible obtener valores que no son iguales. ¿Por qué?. Modifica el programa anterior para que funcione correctamente en todos los casos.

3.1. Valores especiales

Los números en coma flotante se almacenan habitualmente siguiendo el estándar IEEE-754[2]. Dicha representación permite almacenar algunos valores especiales como:

- *NaN* (*Not a Number*). Se usa cuando hay que almacenar algo que no se corresponde con un número real válido.
- *Inf*. Representa el valor "infinito".

Podemos usar *numeric_limits* para trabajar con estos valores especiales. En particular, pueden ser interesantes las siguientes propiedades:

- *quiet_NaN()*. Devuelve el valor *NaN*.
- *infinity()*. Devuelve el valor *Inf*.
- *is_iec559*. Devuelve *true* si el compilador está usando el estándar IEEE-754 / IEC-559. Si lo cumple, entonces tenemos garantizado que la comparación entre un *NaN* y cualquier otra cosa (incluido el propio *NaN*) es *false*.

El siguiente programa ilustra estos aspectos:

```
#include <iostream>
#include <cmath>
#include <limits>
using namespace std;

int main()
{
    double n1 = numeric_limits<double>::quiet_NaN();
    double n2 = numeric_limits<double>::infinity();
    double n3 = -numeric_limits<double>::infinity();
    double n4 = 2.7;

    cout << "n1 vale " << n1 << endl;
    cout << "n2 vale " << n2 << endl;
    cout << "n3 vale " << n3 << endl;
    cout << "n4 vale " << n4 << endl;
```

2 http://www.carlospes.com/curso_representacion_datos/06_01_estandar_ieee_754.php

```
    cout << "Raíz de -1 vale " << sqrt(-1.0) << endl;     // NaN
    cout << "0.0/0.0 vale " << 0.0/0.0 << endl;           // NaN
    cout << "1e1000 vale " << 1e1000 << endl;             // Inf
    cout << "-1e1000 vale " << -1e1000 << endl;           // -Inf

    cout << "Representación según el estándar IEEE-754 / IEC-559 : " <<
            (numeric_limits<double>::is_iec559 ? "Si" : "No") << endl;

    cout << "n1 es NaN : " << (n1==n1 ? "No es NaN" : "Si es NaN") << endl;
    cout << "n4 es NaN : " << (n4==n4 ? "No es NaN" : "Si es NaN") << endl;

    cout << "n2 es Inf : " <<
           (n2==numeric_limits<double>::infinity() ? "Si es Inf" : "No es inf") <<
           endl;
    cout << "n4 es Inf : " <<
           (n4==numeric_limits<double>::infinity() ? "Si es Inf" : "No es inf") <<
           endl;
}
```

***Ejercicio** 5: Valores especiales de coma flotante*

Pruebe el programa anterior, revisando los valores que va obteniendo junto con las correspondientes líneas que los producen.

4. E/S formateada

En ocasiones podemos cambiar el formato o apariencia de lo que mostramos en consola mediante *cout*. En particular podemos hacer uso de lo que se conocen como "manipuladores de formato". Se definen en el fichero de cabecera *iomanip*. Aquí presentamos algunos a modo de ejemplo:

- *dec*. Permite mostrar números en base decimal (activo por defecto).
- *hex*. Permite mostrar números en base hexadecimal.
- *oct*. Permite mostrar números en base octal.
- *setw(x)*. Indica el ancho (número de caracteres) que debe ocupar en consola un determinado dato (sólo afecta al siguiente dato).
- *setfill(c)*. Indica el carácter que se usará para rellenar el espacio que ocupa un dato (por defecto es un espacio).
- *left/right*. Permite justificar un dato a izquierda o derecha.
- Puedes consultar la lista completa de manipuladores en la dirección: http://www.cplusplus.com/reference/iostream/manipulators

El siguiente programa ilustra el uso de algunos de estos manipuladores:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int x;
    cout << "Dime un número: ";
    cin >> x;
```

```
    cout << "Decimal    : " << dec << x << endl;
    cout << "Octal      : " << oct << x << endl;
    cout << "Hexadecimal: " << hex << x << endl;

    cout << dec;

    cout << "8 posiciones            : " << setw(8) << x << endl;
    cout << "8 posiciones (just dcha): " << setw(8) << right << x << endl;
    cout << "8 posiciones (just izda): " << setw(8) << left << x << endl;

    double y;
    cout << "Dime otro número: ";
    cin >> y;

    cout << "Con 2 decimales: " << fixed << setprecision(2) << y << endl;
    cout << "Con 6 decimales: " << fixed << setprecision(6) << y << endl;
}
```

***Ejercicio** 6: Uso de manipuladores de formato*

A continuación tienes un programa que muestra dos listados de datos.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Nombre" << "Apellidos" << "Edad" << "Estado" << endl;
    cout << "Javier" << "Moreno" << 20 << "S" << endl;
    cout << "Juan" << "Espejo" << 8 << "S" << endl;
    cout << "Antonio" << "Caballero" << 53 << "C" << endl;
    cout << "Jose" << "Cano" << 27 << "C" << endl;

    cout << endl;

    cout << 123.456 << 26.467872 << 876.3876 << endl;
    cout << 17.26734 << 0.22 << 18972.1 << endl;
    cout << 456.5 << 2897.0 << 2832.3 << endl;

}
```

Modifíquelo de manera que, únicamente añadiendo manipuladores de formato, obtengamos una salida como la que se muestra a continuación

```
Nombre    Apellidos   Edad Estado
Javier    Moreno        20      S
Juan      Espejo         8      S
Antonio   Caballero     53      C
Jose      Cano          27      C

____123.46_____26.47____876.39
_____17.27______0.22__18972.10
____456.50___2897.00___2832.30
```

5. Conversiones

Cuando en una expresión mezclamos distintos tipos de datos, el compilador se encarga de analizar la expresión y realizar las conversiones que sean necesarias. Estas conversiones se denominan implícitas, y las realiza el compilador de forma automática.

5.1. El tipo booleano

El tipo *bool* es un tipo que no existe en *C*, y se añade al lenguaje *C++*. En el primero, el manejo de valores de tipo booleano (*true/false*) se realiza mediante algún tipo integral, codificando el valor *false* como cero y el valor *true* como distinto de cero. Por ejemplo, si prescindimos del tipo *bool,* podríamos escribir:

```
#include <iostream>

using namespace std;

int main()
{
  int es_menor;
  es_menor= 2<3;
  cout << "Resultado de 2<3: " << (es_menor?"Si":"No") << endl;
}
```

En C++ se pueden usar las mismas expresiones para representar un tipo booleano, por lo que debemos tener en cuenta esta relación para poder interpretar correctamente el comportamiento de nuestros programas.

> ***Ejercicio** 7: Conversión de bool y enteros*
>
> Considere el siguiente programa:
>
> ```
> #include <iostream>
>
> using namespace std;
>
> int main()
> {
>   cout << (4<1<5?"Ordenados":"Desordenados") << endl;
> }
> ```
>
> Compruebe su comportamiento. ¿Qué está ocurriendo?

Aunque nosotros siempre usaremos el tipo *bool* para representar booleanos, debemos tener en cuenta que se pueden realizar conversiones implícitas entre los distintos tipos. Además, el compilador puede usar otros tipos de datos como valores booleanos considerando que el valor cero corresponde a falso y distinto de cero a verdadero.

5.2. El tipo carácter

El tipo carácter en un tipo *integral*, y se puede usar como un entero de menor tamaño. Podemos considerar 3 tipos: *char, signed char* y *unsigned char*, aunque el primero de ellos se comportará como uno de los otros dos.

Es posible "mezclar" este tipo con un tipo entero, de forma que el compilador puede convertir caracteres en enteros y enteros en caracteres. Tenga en cuenta que:

- Si en una expresión hay que operar un carácter con un entero, el compilador convierte el carácter a entero.

- Si asignamos un entero a un carácter, la asignación tendrá sentido si el valor del entero está en el rango válido del carácter.

De hecho, puede consultar las funciones que ofrece el fichero de cabecera *cctype* (*http://www.cplusplus.com/reference/clibrary/cctype*), donde puede comprobar que las funciones, aun estando pensadas para trabajar con caracteres, realmente trabajan con datos enteros.

Aunque *char* sea un "entero pequeño", *cout* distingue si lo que recibe es un carácter o un entero. Así, si tiene que imprimir el valor de una variable de tipo *int* sabe que debe imprimir los dígitos numéricos que corresponden a dicho valor, mientras que si recibe un *char*, sabe que debe imprimir el carácter correspondiente de la tabla *ASCII*.

> ***Ejercicio** 8: Caracteres y número ASCII asociado*
>
> Escriba un programa que recibe como entrada un número del 0 al 25. Como resultado deberá escribir la letra (el 0 indica la 'a' y el 25 la 'z'), su correspondiente mayúscula y los dos valores *ASCII* correspondientes.

5.3. Conversiones explícitas

Hasta ahora, siempre hemos considerado las conversiones como una tarea que realiza de forma automática el compilador. Sin embargo, es posible hacer una conversión explícita, es decir, indicar al compilador que queremos que convierta el valor de una expresión a un determinado tipo. En este caso, diremos que hacemos un *casting* (moldeado).

La forma más simple de hacer un casting es escribiendo

```
(TIPO) EXPRESIÓN
```

donde *TIPO* es el tipo al que queremos convertir la expresión. Por ejemplo, si deseamos escribir la parte entera de un valor real, podemos hacer:

```
#include <iostream>

using namespace std;

int main()
{
    double x;
    cout << "Dime un número: ";
    cin >> x;
    cout << "Parte entera: " << (int) x << endl;
}
```

Tenga en cuenta que el *casting* es otro operador, y que tiene una prioridad alta, ya que está al nivel de los operadores unarios. Si tiene alguna duda, deberá poner la expresión entre paréntesis.

El operador de moldeado que hemos explicado es el que se usa en C, y que también está disponible en C++. Sin embargo, en C++ se incorporan nuevos operadores que son más seguros[3], ya que se diversifica con distintos tipos de *casting* de forma que el compilador conozca mejor nuestras intenciones. Aunque hay varios, simplemente comentaremos la sintaxis del único que usaremos en esta asignatura:

```
static_cast<TIPO> (EXPRESIÓN)
```

que convierte el valor resultante de la expresión al tipo *TIPO.*

3 *No entraremos en detalles sobre los distintos tipos de casting, ya que corresponden a un tema avanzado.*

***Ejercicio** 9: Caracteres y número ASCII asociado*

Escriba un programa que recibe como entrada un número del 0 al 25. Como resultado deberá escribir la letra (el 0 indica la 'a' y el 25 la 'z'), su correspondiente mayúscula y los dos valores ASCII correspondientes, habiendo declarado únicamente un objeto de tipo *int*.

***Ejercicio** 10: Limites del tipo char*

Escriba un programa que imprima los límites (mínimo y máximo) de los tres tipos de datos: `signed char, char, unsigned char`. Con este ejercicio podrá confirmar a qué tipo de dato corresponde el tipo char de su sistema.

6. Experimentando con la codificación

La mejor forma de entender que los datos se pueden codificar de distinta forma es con la práctica. Es conveniente que hagamos algunas pruebas para confirmar cómo se comporta nuestro sistema con algunos tipos de datos.

6.1. Codificación de caracteres

Vamos a trabajar fundamentalmente considerando una codificación *ISO-8859-15,* es decir, con la tabla ASCII extendida para Europa Occidental. Por tanto, cuando trabajemos con texto, la lectura de un objeto de tipo *char* implicará la lectura de un *byte* que corresponde a un carácter de la siguiente tabla.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGC	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Sin embargo, en la práctica podemos editar en otros formatos. En concreto, la codificación *UTF8* es la más habitual en *GNU/Linux*. En las secciones anteriores se ha trabajado especialmente con caracteres que corresponden a la primera parte de la tabla *ASCII*. Estos caracteres se codifican de forma idéntica en ambos casos, por lo que hemos evitado considerar ningún detalle relativo a la codificación. Ahora bien, ¿Qué ocurre con otros caracteres de la segunda mitad de la tabla?.

Los caracteres de la segunda mitad son un valor de 128 a 255 en la codificación ISO-8859-15, por lo que sólo necesitan de un byte. En código UTF8, se codifican como dos bytes.

> ***Ejercicio** 11: Analizando un archivo*
>
> Escriba un programa "*verbytes.cpp*" que lea desde la entrada estándar –cin– el contenido de un archivo y escriba en la salida estándar todos y cada uno de los valores de *char* que lea. El valor escrito será un número en el rango 0-255.
>
> Para probar el programa, obtenga los valores que genera un archivo "*letras8859.txt*" que contenga las siguientes letras (en formato ISO-8859):
>
> *a e i o u A E I O U á é í ó ú Á É Í Ó Ú ñ Ñ ü Ü*
>
> Los valores obtenidos deberían coincidir con la información que presenta la tabla anterior.

El programa que ha creado en el ejercicio 11 también es válido para analizar un archivo en formato *UTF8*. Puede probar el resultado sobre un archivo "*letrasUTF8.txt*" de idéntico contenido[4], pero con esta codificación. Podrá comprobar que los caracteres que tenían códigos de la parte extendida ahora se codifican de otra forma.

Si analiza con cuidado los resultados de los valores numéricos que ha obtenido para ambas codificaciones, podrá entender que si tenemos un texto escrito en castellano con una de las dos codificaciones, se puede deducir a cuál de ellas corresponde. Esta operación la realizan muchos editores cuando abren un archivo, de forma que el usuario pueda trabajar con la misma codificación que contenía el archivo sin necesidad de saber nada sobre ello.

> ***Ejercicio** 12: Deduciendo codificación*
>
> Escriba un programa "*adivinacodigo.cpp*" que lea desde la entrada estándar –cin– el contenido de un archivo y escriba en la salida estándar la posible codificación. Para simplificar el problema, suponga que sólo analizamos las letras de un texto escrito en castellano.
>
> Recuerde los códigos de las letras:
>
> *a e i o u A E I O U á é í ó ú Á É Í Ó Ú ñ Ñ ü Ü*
>
> En codificación ISO-8859-15: 97, 101, 105, 111, 117, 65, 69, 73, 79, 85, 225, 233, 237, 243, 250, 193, 201, 205, 211, 218, 241, 209, 252 y 220.
>
> En codificación UTF8: 97, 101, 105, 111, 117, 65, 69, 73, 79, 85, (195/161), (195/169), (195/173), (195/179), (195/186), (195/129), (195/137), (195/141), (195/147), (195/154), (195/177), (195/145), (195/188), (195/156).
>
> Pruebe el resultado con los archivos de ejemplo "*texto1.txt*", "*texto2.txt*", "*texto3.txt*" y "*texto4.txt*".

Observe que si nos limitamos a los caracteres habituales en un texto en castellano no hay demasiadas diferencias entre ambas codificaciones.

4 *Puede usar la orden "iconv" de GNU/Linux para transformar cualquier fichero de una codificación a otra. Pruebe por ejemplo "iconv -l" y podrá ver la cantidad de codificaciones que podría procesar.*

***Ejercicio** 13: Conversor*

Escriba un programa "utf2iso8859*.cpp*" que lea desde la entrada estándar –cin– el contenido de un archivo codificado en UTF8 y escriba en la salida estándar el mismo archivo pero codificado en formato ISO-8859-15.

Para probar el programa, uso el archivo "textoUTF8*.txt*".

6.2. Codificación de enteros y reales: uniones

Para experimentar con tipos de dato entero y real, vamos a introducir las *uniones*, una forma especial de estructura. Para definir una estructura podemos usar la siguiente sintaxis:

```
union NOMBRE_UNION {
  tipo1 campo1;
  tipo2 campo2;
  …
  tipon campon;
};
```

Esta sintaxis es muy parecida a la de las estructuras. Define un nuevo tipo con el nombre de la *union*. Así, después de su definición, podemos declarar objetos con ese nuevo nombre. Por ejemplo, en el siguiente código:

```
union Mezcla {
    double x;
    int i;
};
Mezcla m;
```

se define un nuevo tipo – de nombre *Mezcla* – con dos miembros (de tipo *double* y tipo *int*) y a continuación declaramos el objeto *m* de ese tipo.

Las uniones están diseñadas para reservar un espacio de memoria tan grande como el más grande de los miembros que la componen. Cuando definimos una unión, no queremos almacenar tantos objetos como miembros hemos incluido en la definición. El objetivo es almacenar tan sólo uno de ellos. Por eso, el compilador sólo necesita reservar espacio suficiente para el mayor. Por ejemplo, en mi máquina el tipo *double* ocupa 8 bytes y el tipo *int* ocupa 4. El objeto *m* de tipo *Mezcla* anterior ocupa 8 bytes, es decir, el mayor de los tamaños.

El efecto de declarar m de tipo *Mezcla* es que el compilador busca una zona de memoria para la unión y la llama *m*. Una vez tenemos el objeto, podemos usar cualquiera de sus miembros para usar la zona de memoria como más nos convenga. Es decir, podemos decir que:

- Todos los miembros se sitúan en la misma dirección de memoria.
- Cada miembro o campo se refiere a una interpretación de los contenidos que hay en esa zona de memoria.
- En un momento dado, sólo nos interesa uno de los miembros de la estructura. Si almacenamos un dato conforme a uno de sus miembros, sólo sería portable usar el contenido de la unión con esa misma interpretación.

Las uniones serán útiles cuando queramos almacenar sólo un dato, aunque dependiendo de alguna condición, podría corresponder a uno de entre varios tipos. Por ejemplo, en la unión que hemos definido podemos almacenar un entero o un número real, dependiendo del campo al que nos refiramos. Si almacenamos un entero, no tiene sentido que preguntemos por el valor del campo *double*, ya que la única forma de garantizar que el dato tiene sentido es acceder a la unión como un entero.

Las uniones son poco habituales, especialmente en código a un nivel de abstracción alto. Muchas soluciones en base a uniones se pueden resolver de una forma más eficaz con otras técnicas. Es posible que si encuentra código de este tipo, sea a un nivel bajo, en casos en los

que se desea resolver un problema de una forma muy simple, con un mínimo de memoria.

En este guión proponemos su uso para explorar los contenidos de distintos tipos. La idea es que si almacenamos en la unión un dato por uno de sus campos, podemos acceder a otro de sus campos para ver los efectos que tendría ese contenido interpretado de otra forma.

***Ejercicio** 14: Interpretación de la memoria*

Escriba un programa "*interpretaciones.cpp*" para analizar algunos detalles de representación de su máquina. Antes de resolver el problema:

1. Determine qué tipo de dato de su máquina es un entero sin signo de 1 byte.
2. Determine qué tipo de dato de su máquina es un entero sin signo de 2 bytes.
3. Determine qué tipo de dato de su máquina es un entero con signo de 4 bytes.
4. Determine qué tipo de dato de su máquina es un número en coma flotante con 4 bytes.

Nos centraremos en analizar los efectos en un bloque de 4 bytes. Este bloque puede interpretarse de distintas formas:

- Como un entero con signo.
- Como un número real de 4 bytes.
- Como 2 números enteros sin signo de 16 bits, consecutivos.
- Como 4 números enteros sin signo de 8 bits, consecutivos.

El programa debe definir un tipo de dato que permita consultar un objeto de 4 bytes con cualquiera de las 4 interpretaciones. Para ello, defina una unión que tenga las 4 posibilidades. Use los tipos de datos necesario y definiciones que considere necesarios. El programa debe:

1. Imprimir en la salida estándar el tamaño de ese tipo de dato para confirmar que tiene 4 bytes.
2. Mostrar el contenido de esos 4 bytes en el caso de que:
   a) Se almacene el entero de 4 bytes 0.
   b) Se almacene el entero de 4 bytes 1.
   c) Se almacene el entero de 4 bytes -1.
   d) Se almacenen dos enteros de 2 bytes: el 0 y el 65535.
   e) Se almacenen dos enteros de 2 bytes: el 65535 y el 0.
   f) Se almacene el número real de 4 bytes 1.0.
   g) Se almacene el número real de 4 bytes -1.0.

Para mostrar el contenido, defina una función que recibe un objeto y escribe en la salida estándar las distintas interpretaciones: el entero, el flotante, los dos enteros de 16 bits, y los 4 enteros de 8 bits. Adicionalmente, deberá imprimir los 32 bits desde el más significativo al menos significativo.

7. Referencias

- Garrido, A. "*Fundamentos de programación en C++*". Delta publicaciones, 2006.
- http://www.carlospes.com/curso_representacion_datos/06_01_estandar_ieee_754.php.
- Manipuladores C++. http://www.cplusplus.com/reference/iostream/manipulators
- Funciones de caracteres. http://www.cplusplus.com/reference/clibrary/cctype