# Bloque 1

## Información digital

Prácticas de
Fundamentos de Computadores y Redes
**27 de enero de 2017**

SESIÓN 1

# Representación de números enteros

## Objetivos

Esta práctica analizará la forma en la que se representan números naturales y enteros en un lenguaje de alto nivel, en concreto en C++. Además, se realizarán diversas operaciones con datos numéricos para entender las repercusiones que tienen los errores de desbordamiento en un programa.

## Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación de números naturales y enteros (binario natural, hexadecimal, signo-magnitud y complemento a 2).
- Conocer el lenguaje de programación C++.

## Desarrollo de la práctica

## 1. Codificación de enteros y naturales

Para realizar esta práctica se va a utilizar como herramienta el Visual Studio. Para simplificar se proporciona un proyecto mínimo ya creado. Descarga del Campus Virtual el fichero `1-Enteros` y descomprímelo.

- ❏ Abre el proyecto haciendo doble clic sobre el fichero de solución (extensión `sln`).
- ❏ Una vez que está abierto el proyecto compila y ejecuta el programa pulsando la combinación de teclas CTRL-F5.

La salida que proporciona este programa es sorprendente: dice que `a` vale 23 y `b`, -5, pero que `a` es menor que `b`. Para entender por qué ocurre esto, en primer lugar hay que codificar estos números. La codificación depende del tipo con el que estén declarados:

- ❏ La variable `a` está declarada como `unsigned int`, lo que significa, en la versión de Visual Studio empleada en prácticas, que es un número natural de 32 bits codificado en binario natural. ¿Cuál deberá ser la codificación de `a`? Indica el resultado en hexadecimal[1].
- ❏ La variable `b` está declarada como `int`, lo que significa, en la versión de Visual Studio empleada en prácticas, que es un número entero de 32 bits codificado en complemento a 2. ¿Cuál deberá ser la codificación de `b`?[2]

1

2

Para analizar la codificación de las variables en memoria es necesario ejecutar el programa en modo depuración y pausar su ejecución. La ejecución de un programa se puede pausar utilizando puntos de interrupción, que permiten interrumpir la ejecución del programa en cualquier línea. Realiza los siguientes pasos:

- ❏ Sitúa el cursor en la línea en la que se imprime el mensaje por pantalla. Pulsa F9 para colocar en esa línea un punto de interrupción. El Visual Studio indica que en esa línea hay un punto de interrupción dibujando un círculo rojo en el margen izquierdo. Pulsando con el botón izquierdo del ratón sobre el círculo rojo se elimina el punto de interrupción, y volviendo a pulsar se vuelve a crear.
- ❏ Ejecuta el programa en modo depuración pulsando F5. Fíjate que para ejecutar el programa en modo depuración no hay que pulsar la tecla *Control*.
- ❏ El programa se detendrá justo en la línea donde se colocó el punto de interrupción. La flecha amarilla en el margen izquierdo indica la próxima sentencia que se va a ejecutar.
- ❏ En este momento pasando el ratón por encima del nombre de una variable en el programa se puede obtener su valor.
- ❏ Otra forma de inspeccionar el valor de las variables es la ventana denominada «Automático», donde se muestran automáticamente variables utilizadas cerca del código ejecutado. También existe la posibilidad de escoger las variables que queremos ver utilizando la ventana «Inspección». Sigue estos pasos para abrir esta ventana: pulsa en el menú *Depurar*, a continuación pulsa en *Ventanas*, y dentro de *Inspección* pulsa en *Inspección 1*.
- ❏ Escribe en el inspector de variables el nombre de las variables `a` (en una línea) y `b` (en otra línea) para ver el valor que almacenan; podrás ver su valor en decimal. Pulsa con el botón derecho del ratón en cualquier parte de la ventana *Inspección 1* y selecciona la opción *Presentación hexadecimal*. El contenido de las variables debería coincidir con la codificación que habías indicado. Si no es así, repasa la codificación y pregunta a tu profesor si no encuentras el problema.

El programa ha determinado que `a` es menor que `b` debido a que al realizar la comparación se ha encontrado con que `a` es una variable sin signo y `b` con signo y ha decidido interpretar la codificación de los dos números como variables sin signo. Esto es así porque C hace tipados (*castings*) automáticos y en el caso de una expresión que mezcle números con y sin signo, decide interpretarlos ambos como números sin signo. Para ver qué valor tiene `b` interpretado sin signo, deberás tomar su codificación e interpretarla como binario natural en lugar de complemento a 2. Como es un número muy grande, vamos a utilizar la calculadora de Windows para comprobar su valor:

- ❏ Pulsa el botón de Windows y en «Buscar programas y archivos» escribe `calc` y pulsa Enter. Se abrirá la calculadora de Windows.
- ❏ En el menú `Ver` escoge `Programador`.
- ❏ Pon la calculadora en modo hexadecimal pulsando sobre `Hexa` en la parte izquierda.
- ❏ Escribe la codificación de `b`.
- ❏ Pulsa sobre `Dec` para que pase el número hexadecimal a decimal.

Como verás, es un número muy grande, mucho mayor que 23 (el valor de `a`) y por eso se escribe el mensaje de que `b` es mayor que `a`. En Visual Studio, recompila la aplicación con *Compilar→Recompilar solución* (no vale una compilación normal porque como no se ha cambiado el código, no vuelve a compilar) y muestra la ventana denominada «Lista de errores» (*Ver→Lista de errores*). Podrás ver que, durante la compilación, se generó un aviso («warning») indicando que no coincidían los tipos de las dos variables comparadas. Recuerda que debes considerar los avisos como errores a no ser que sepas qué consecuencias tienen y que en ese caso concreto no son un error.

## 2. Almacenamiento en memoria

Vamos a ver cómo se almacena la codificación de estas variables en memoria. Para ello realiza los siguientes pasos:

- ❏ En primer lugar es necesario determinar la dirección de memoria en la que se almacenan las variables. Ejecuta el programa en modo depuración pulsando F5 y añade al inspector de variables `&a` (en una nueva línea) y `&b` (en otra línea). El operador `&` se utiliza para calcular la dirección a partir de la cual está almacenada una variable[1]. ¿En qué dirección se almacena `a`? 3 ¿Y `b`? 4
- ❏ A continuación se examinará el contenido de la memoria en las direcciones indicadas. Para ello pulsa en el menú *Depurar*, a continuación pulsa en *Ventanas*, y dentro de *Memoria* pulsa en *Memoria 1*. Esta acción abrirá el inspector de memoria.
- ❏ En el campo dirección del inspector de memoria introduce la dirección de la variable `a`. Los datos se muestran en hexadecimal agrupados por bytes, por tanto los 32 bits que contienen la codificación de la variable `a` son los cuatro primeros grupos (4 bytes son 32 bits). Sin embargo, para poder obtener el valor de la codificación hay que tener en cuenta que los datos en memoria se almacenan en orden inverso siguiendo el criterio *little-endian*. Esto quiere decir que si se quiere almacenar en memoria el dato `12AB34CD`, la secuencia de bytes que se observará será `CD 34 AB 12`. Teniendo esto en cuenta comprueba que la codificación de `a` coincide con lo que habías respondido previamente.
- ❏ Siguiendo el mismo procedimiento, comprueba que la codificación de `b` coincide con tu respuesta anterior.
- ❏ Para continuar con la ejecución del programa pulsa de nuevo F5. Como no hay más puntos de interrupción, el programa terminará su ejecución.

3

4

[1] El prefijo `0x` indica que el número que aparece a continuación está representado en hexadecimal.

- ❏ En el código C++, cambia el valor que se asigna a la variable `a` por 64 y el valor que se asigna a la variable `b` por -1. Codifica manualmente estos datos y repite el proceso anterior para verificar que coinciden con la codificación que se hace de estas variables en el programa.

## 3. Desbordamiento

El desbordamiento se produce cuando se realiza una operación aritmética y el resultado no es representable para el tipo de formato y número de bits con el que se está trabajando. Realiza los siguientes pasos para ver un ejemplo:

- ❏ Modifica el programa anterior para que se lean por teclado dos números enteros (con signo), `a` y `b`.
- ❏ Suma `a` y `b`, almacenando el resultado en otra variable entera (con signo) denominada `c`. Imprime el valor de `c` por pantalla.
- ❏ Compila y ejecuta el programa pulsando Ctrl-F5. Introduce valores para `a` y `b`, y verifica que funciona correctamente.
- ❏ Vuelve a ejecutar el programa pero esta vez introduce el valor dos mil millones para `a` (un 2 y nueve 0) y lo mismo para `b`. ¿Cuál es el resultado?[5]
- ❏ Razona el resultado que se ha producido. Si no tienes clara la causa deberías consultar con tu profesor.

5

Se podría pensar que una forma de evitar el problema del desbordamiento es utilizar un mayor número de bits. Sin embargo, siempre se podrían encontrar dos números tal que su suma no fuera representable para ese número de bits. Por tanto, **el desbordamiento es un problema que no se puede evitar, solo detectar**.

Para detectar el desbordamiento cuando se ha realizado una suma con números enteros basta con que algunas de las siguientes dos condiciones sea cierta:

1. Que `a` y `b` sean positivos y `c` sea negativo.
2. Que `a` y `b` sean negativos y `c` sea positivo (mayor o igual que cero).

- ❏ Modifica el programa anterior para que, en caso de que la suma provoque desbordamiento, no se imprima el resultado, sino un mensaje indicándolo.

El problema del desbordamiento en sí no se puede evitar. Sin embargo, hay ciertas operaciones que, dependiendo de la forma en la que se realizan, pueden o no generar problemas de desbordamiento. El cálculo de la media aritmética es una de ellas.

- ❏ Añade al programa anterior una nueva variable entera, `d`.
- ❏ Asigna a `d` la media entre `a` y `b` e imprime por pantalla el valor de `d`.
- ❏ Ejecuta el programa e introduce nuevamente tanto para `a` como para `b` el valor dos mil millones (un 2 y nueve 0).

- ❏ Si has calculado la media como $(a + b)/2$ el resultado te habrá salido incorrecto. El problema en este caso no es que el resultado de la operación genere desbordamiento, ya que la media de dos números iguales es ese mismo número y, por tanto, representable. El problema en este caso es que se produce desbordamiento en una operación intermedia, en concreto en la suma que se realiza antes de la división entre dos.
- ❏ Una forma alternativa de calcular la media sería mediante la operación $a/2 + b/2$. Matemáticamente es equivalente a la forma de calcular la media anterior, pero en este caso se evita el problema del desbordamiento en las operaciones intermedias. Modifica el cálculo de la media con esta expresión.
- ❏ Ejecuta de nuevo el programa con los mismos datos y observa los resultados.

Utilizando los conocimientos adquiridos responde a la siguiente pregunta: ¿qué valor se imprimirá por pantalla al ejecutar el siguiente fragmento de código?[6]

6

```
1 const int UN_MILLON = 1000000;
2 int contador = 0;
3 for (int i = 0; i < 3000*UN_MILLON; i++)
4   contador++;
5 cout << "contador " << contador << endl;
```

- ❏ Añade el fragmento de código al programa y verifica que tu respuesta fue correcta.

## 4. Ejercicios adicionales

- ✏ Crea un programa que realice la suma acumulada de todos los números entre 1 y un millón que son múltiplos de 3. Para acumular la suma, utiliza variables de los siguientes tipos: `int` (entero de 32 bits), `short` (entero de 16 bits) y `long long` (entero de 64 bits). Compara y analiza los resultados.
- ✏ Crea un programa que lea por pantalla dos números enteros e imprima su diferencia. En caso de que dicha operación aritmética produzca un desbordamiento imprime un mensaje de error indicándolo.

SESIÓN 2

# Representación de números reales y caracteres

## Objetivos

Esta sesión analizará la forma en la que se representan números reales y caracteres en un lenguaje de alto nivel, en concreto en C++. Además, se realizarán diversas operaciones con datos numéricos para entender las repercusiones que tienen los errores de redondeo y desbordamiento en un programa.

## Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación numéricos de números reales (IEEE-754).
- Conocer los sistemas de codificación de caracteres.
- Conocer el lenguaje de programación C++.

## Desarrollo de la práctica

## 1. Codificación de números reales

Vamos a ver cómo se codifican números reales. En primer lugar, codifica el número -27.625 en el formato IEEE 754. ¿Cuál es la codificación en hexadecimal?[1]

[1]

Vamos a comprobarlo con un programa. Sigue estos pasos:

- ❏ Descarga el fichero `2-Reales` del Campus Virtual y descomprímelo. Abre el fichero de solución en Visual Studio.
- ❏ Modifica el programa insertando al principio del `main` la declaración de una variable llamada `f` de tipo `float`.
- ❏ Añade a continuación una sentencia que asigne a la variable `f` el valor `-27.625`.
- ❏ Añade un mensaje que imprima por pantalla el valor de la variable.

- ❏ Ejecuta el programa y verifica que funciona correctamente.
- ❏ Inserta un punto de ruptura en la sentencia que muestra el mensaje e inicia la depuración.
- ❏ Muestra la ventana de memoria e introduce en el campo dirección `&f`. En los cuatro primeros bytes deberás ver la codificación de f. Teniendo en cuenta que está guardada usando la convención *Little Endian*, comprueba que coincide con la codificación que habías hecho tú.

## 2. Redondeo

Los números reales se ven afectados por los mismos problemas que los números enteros en cuanto al desbordamiento, aunque la forma de detectarlo es diferente. Además, en la representación de números reales aparece un nuevo problema: el problema del redondeo. Este problema ocurre cuando un número real no tiene una representación exacta.

Para comprobar en qué consiste el problema de redondeo convierte el número 0.1 a binario. ¿Cuál es su codificación?[2]. El número en binario no tiene una representación exacta con lo que se tendrá que trabajar con aproximaciones.

2

A continuación se van a comprobar los problemas que pueden generar este tipo de errores:

- ❏ En primer lugar comenta todo el código del `main`.
- ❏ Añade dos variables de tipo `float` `f1` y `f2`.
- ❏ Asigna el valor 0.1 a `f1` y 0.3 a `f2`.
- ❏ Imprime por pantalla el valor de ambas variables.
- ❏ Compila y ejecuta el programa pulsando Ctrl-F5.
- ❏ Aparentemente se imprime el valor exacto, pero eso se debe a que por defecto se imprimen pocos decimales. Para imprimir los números reales con más decimales en primer lugar hay que añadir al programa un nuevo fichero de cabecera:

```
1 #include <iomanip>
```

  A continuación se puede indicar a `cout` que se quieren imprimir más decimales de la siguiente forma:

```
1 cout << "f1: " << setprecision(15) << f1 << endl;
2 cout << "f2: " << setprecision(15) << f2 << endl;
```

- ❏ Ejecuta de nuevo el programa y observa los resultados.

Ciertamente los errores de redondeo son muy pequeños. Sin embargo, si no se tienen en cuenta se pueden generar graves problemas.

- ❏ Añade al programa una sentencia condicional que imprima si `f1`$\times 3$ es igual a `f2` o si son distintos. Dado que estamos trabajando con números reales, todos los números de la expresión deben ser reales para que el compilador no aplique un tipado diferente: la expresión a utilizar debería ser `f1*3.0`.

❏ Ejecuta el programa y comprueba los resultados.

El problema es que los errores de redondeo cometidos al representar `f1` y hacer la operación `f1*3.0` son distintos a los errores cometidos al representar `f2`, lo que hace que las magnitudes comparadas sean ligeramente diferentes. Por este motivo para comparar dos números reales se utiliza siempre una expresión similar a la siguiente:

```
1 const float TOLERANCIA = 0.0000001;
2 if (fabs(f1*3.0 - f2) < TOLERANCIA)
3   cout << "Son iguales" << endl;
4 else
5   cout << "Son distintos" << endl;
```

❏ Añade el fragmento de código al programa y verifica que ahora la comparación da el resultado esperado. La función `fabs` calcula el valor absoluto de un número y está definida en el archivo de cabecera `math.h`. El valor de la tolerancia depende del tipo de datos que se manejen en la aplicación.

Un error de redondeo muy pequeño puede dar lugar a un error muy grande si se realizan muchas operaciones con números reales. Los pequeños errores se van acumulando y finalmente el error empieza a ser significativo.

Supongamos que se almacena sobre una variable de tipo `float` el número de euros que una persona tiene en el banco. ¿Qué ocurriría si dicha persona realiza 20 millones de ingresos de 10 céntimos cada uno?

❏ Añade al programa el siguiente fragmento de código que simularía el caso anterior:

```
1 float euros = 0;
2 const int UN_MILLON = 1000000;
3 for (int i = 0; i < 20*UN_MILLON; i++)
4   euros = euros + 0.1;
5 cout <<"euros = " << euros << endl;
```

❏ 20 millones multiplicado por 0.1 debería dar como resultado 2000000 (2 millones). Ejecuta el programa y observa el error que se produce.

❏ Piensa que ocurriría si en lugar de ingresar 0.1 euros se ingresase 1 euro cada vez. En este caso el número 1 tiene una representación exacta donde no se producen errores de redondeo. ¿Crees que el resultado será correcto?

❏ Haz los cambios oportunos y comprueba lo que ocurre en este caso.

❏ Nuevamente aparecen errores de redondeo. Aunque el número 1 tiene una representación exacta, no ocurre lo mismo con otros números que aparecen en los cálculos.

❏ Añade el siguiente código al programa:

```
1 float f3 = 19*UN_MILLON;
2 cout << "f3: " << setprecision(15) << f3 << endl;
3 f3 = f3+1;
4 cout << "f3 + 1: " << setprecision(15) << f3 << endl;
```

❏ Reflexiona sobre el resultado que va a producir el código que se acaba de añadir. Ejecuta el programa y analiza los resultados.

Por todos estos motivos, la utilización de números reales en los programas sin conocer sus limitaciones debidas a los errores de redondeo puede provocar grandes problemas. Es por tanto esencial para el desarrollo de programas robustos el conocimiento de la forma en la que se codifica la información.

## 3. Codificación de caracteres

Vamos a ver a continuación cómo se codifican caracteres en C++. Sigue estos pasos:

- ❏ Comenta el código de la función `main`.
- ❏ Define una variable `c1` de tipo `char` y asígnale el valor `'a'`. Fíjate que para indicar un carácter individual se utilizan las comillas simples.
- ❏ Define otra variable `c2` también de tipo `char` y haz que contenga el carácter `'ñ'`.
- ❏ Añade una sentencia que imprima los dos caracteres separados por un espacio.
- ❏ Ejecuta el programa y comprueba que la `a` se imprime correctamente pero la `ñ` no.

Vamos a analizar lo que está ocurriendo. En primer lugar, debes saber que el tipo `char` de C sirve para almacenar un byte. Vamos a ver en concreto qué secuencia de bits se almacena en las dos variables que hemos definido:

- ❏ Pon un punto de ruptura en la línea que imprime las variables e inicia la depuración con F5.
- ❏ Muestra la ventana de memoria y haz que muestre el valor de la variable `c1` en hexadecimal. ¿Cuál es?[3] Como puedes comprobar si buscas en Internet la tabla de códigos ASCII[1] ese valor hexadecimal se corresponde con la codificación ASCII de la letra `a`. Como es una letra del alfabeto inglés y forma parte del ASCII, también tendrá esa misma codificación en todas las extensiones del ASCII, incluyendo ISO-Latin-1, y en UTF-8.
- ❏ Muestra en la ventana de memoria el valor de la variable `c2` en hexadecimal. ¿Cuál es?[4] Este no puede ser un código ASCII porque los códigos ASCII sólo van hasta 127 (7Fh). En realidad es un código que se corresponde con una de las extensiones de ASCII. MS-DOS, el sistema operativo de Microsoft antecesor de Windows, tradicionalmente usaba en España una extensión llamada página de códigos 850[2]. Sin embargo, en la actualidad las aplicaciones usan distintas extensiones según estén configuradas, siendo una de las más comunes en España la denominada Windows-1252[3]. ¿Cuál de estas dos extensiones asigna a la `ñ` el código que se encuentra almacenado en memoria?[5]

3

4

5

[1] http://es.wikipedia.org/wiki/ASCII
[2] http://es.wikipedia.org/wiki/Página_de_códigos_850
[3] http://es.wikipedia.org/wiki/Windows-1252

Esa es la versión que está utilizando el editor de Visual Studio. Sin embargo, la consola que se muestra cuando ejecutas el programa está utilizando la otra extensión y, por esta razón, el carácter mostrado no se corresponde con la `ñ`, sino con el carácter que se corresponde con el número `F1h` en esa extensión.

Vamos a comprobar que es así:

- ❏ Finaliza la depuración.
- ❏ Modifica la sentencia de asignación de `c2` para que se le asigne el código hexadecimal correspondiente a la `ñ` en la extensión que utiliza la consola. ¿Cuál es?[6] Para asignar directamente un código hexadecimal puedes poner su valor en hexadecimal antecediéndolo de 0x. Por ejemplo, si quieres asignar el valor hexadecimal `5a` a la variable `var` tendrías que poner `var = 0x5a`.
- ❏ Ejecuta el programa y comprueba que ahora se imprime la `ñ` correctamente.

[6]

## 4. Ejercicios adicionales

✏ Es importante darse cuenta de que los errores con números reales que se han observado en esta práctica ocurrirán en la mayoría de los programas que trabajen con datos numéricos. A modo de ejemplo, abre una hoja Excel y añade la siguiente información a sus celdas:

- ❏ En la celda `A1` escribe 1,324
- ❏ En la celda `A2` escribe 1,319
- ❏ En la celda `A3` calcula la diferencia entre `A1` y `A2` de la siguiente forma: =A1-A2
- ❏ En la celda `A4` comprueba que la diferencia es 0.005 de la siguiente forma: =SI(A3=0,005; "Iguales"; "Diferentes")
- ❏ Interpreta el resultado. Deberías ser capaz de identificar el problema.

✏ ¿Se podría utilizar una variable de tipo `char` para almacenar cualquier carácter codificado con UTF-8?[7] ¿Por qué?

[7]