

# Problemas sobre representación de caracteres (Ejemplos resueltos)

March 1, 2023

## Contents

<b>1 Problemas</b>	<b>1</b>
<b>2 Apéndice: literales de caracteres y literales de cadena en C++ (C++20)</b>	<b>9</b>
2.1 Basic Character Set y Execution Character Set . . . . .	9
2.2 Posibles formas de literales de caracteres en C++ . . . . .	10
2.3 Literales de cadenas . . . . .	11

## 1 Problemas

**Nota:** Antes de la resolución de este problema se recomienda leer el apéndice de este documento que es un extracto de la referencia de C++20 en lo que se refiere a literales de caracteres y literales de cadena.

**Problema 1.** Al final de este enunciado tienes un programa C++ donde se definen diferentes literales de cadena con las codificaciones UTF8 y UTF16. Las cadenas s1 y s2 están codificadas en UTF8 y la cadena s4 en UTF16.

1. Abre la página web <https://www.onlinegdb.com/> y configura el lenguaje de programación para C++20. Esta página web es un editor-compilador-depurador en línea que nos permitirá estudiar en profundidad el funcionamiento del código y la forma en la que se representan los datos en la memoria.
2. Copia el código que se incluye al final de esta página en el editor de onlinegdb y ejecútalo. Observa las cadenas que se muestran en la terminal. Las cadenas s1 y s2 incluyen símbolos no representables en US-ASCII. ¿Cuáles son?. Busca en la página web <https://symbl.cc/en/unicode/blocks/> los valores de los puntos de código Unicode de cada uno de los caracteres de las cadenas s1 y s2. La cadena s4, incluye un símbolo adicional, el símbolo musical de la clave de sol. Observa como se ha introducido este símbolo en la cadena utilizando el universal character name de 32 bits en el código `\U0001D11E`. Comprueba en la tabla Unicode que efectivamente este punto de código se corresponde con el símbolo de la clave musical. Razona por qué la impresión en el terminal de las cadenas s1, s2, s3 y s4 es correcta.
3. El array s5 es tipo `wchar_t` (o `char` amplio) que en g++ de linux se corresponde con 32 bits, mientras que para el compilador de Microsoft es 16 bits. El array es asignado a un literal que comienza con la letra L, indicando precisamente que debe codificarse cada carácter en `wchar_t`. Observa la salida en la terminal. Razona los problemas que hacen que se haya impreso incorrectamente la cadena en la terminal.
4. Utiliza ahora el botón Debug para iniciar el depurador. Introduce el comando start para ir a la primera línea del código. Ahora ve pulsando next para ir avanzando línea por línea. Recuerda que la línea que se muestra después de haber pulsado next es la siguiente línea en ejecutarse. Ejecuta al menos hasta que se hayan asignado las variables s1 y s6. Ahora observa los bytes en la memoria para estos punteros. Por ejemplo, para s1 puedes introducir: `x/25xb s1` y se mostrarán los 25 bytes a partir de s1

. Anota en un papel la lista de bytes que forman las cadenas s1 y s6. Ahora repasa los apuntes y trata de obtener tú a mano la codificación UTF8 para la cadena s1 y UTF16 para la cadena s6, y compara los resultados con lo devuelto por el depurador. Deberías obtener los mismos bytes.

**Nota:** en la codificación UTF16 hay que hacer una resta del punto de código con el valor 0x10000 que en binario complemento a 2 es 00010000000000000000. Esta resta puedes hacerla fácilmente como una suma binaria con -(0x10000) que se representa en binario complemento a 2 como 11110000000000000000

```
#include <iostream>
#include <codecvt>
#include <string>
#include <locale>

int main() {

    // UTF-8
    // Default for g++ in linux systems
    const char s1[] {"Halag\u00fce\u00f1o"};
    const char s2[] {"Halagüeño"};
    const char8_t s3[] {u8"Halag\u00fce\u00f1o"};
    const char8_t s4[] {u8"Halag\u00fce\u00f1o\U0001D11E"};

    //Streaming utf8 chars
    std::cout<<" Literal_de_cadena_con_universal_character_names:_"<<s1<<std::endl;
    std::cout<<" Literal_de_cadena_sin_universal_character_names:_"<<s2<<std::endl;
    std::cout<<" Literal_de_cadena_con_universal_character_names_UTF8
convertido_stream_de_bytes:_"<<reinterpret_cast<const char*>(s3)<<std::endl;
    std::cout<<" Literal_de_cadena_con_universal_character_names_UTF8_convertido_a_
stream_de_bytes:_"<<reinterpret_cast<const char*>(s4)<<std::endl;

    // UTF-16

    const wchar_t s5[] {L"Halag\u00fce\u00f1o\U0001D11E"};

    const char16_t s6[] {u"Halag\u00fce\u00f1o\U0001D11E"};
    //Streaming out wide chars
    std::wcout<<s5<<std::endl;
    std::cout<<reinterpret_cast<const char*>(s6)<<std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> converter;
    std::u16string s7 {u"Halag\u00fce\u00f1o\U0001D11E"};
    std::string s8 = converter.to_bytes(s7);
    std::cout<<" Literal_cadena_utf-16_y_u16string_convertido_a_utf-8:_"<<s8<<std::
endl;
    return 0;
}
```

*Proof.* Solución

Para el primer apartado, simplemente abrimos la página web:

<https://www.onlinegdb.com/>

configuramos C++20 y copiamos el código en el editor, como se muestra en la figura 1.

Contestamos ahora las cuestiones del apartado segundo.

```

1 #include <iostream>
2 #include <codecvt>
3 #include <string>
4 #include <locale>
5
6 int main(){
7
8 // UTF-8
9 // Default for g++ in linux systems
10 const char s1[] {"Halag\u00fce\u00f1o"};
11 const char s2[] {"Halagüño"};
12 const char8_t s3[] {u8"Halag\u00fce\u00f1o"};
13 const char8_t s4[] {u8"Halag\u00fce\u00f1o\u000D\u0011E"};
14
15
16 //Streaming utf8 chars
17 std::cout<<"Literal de cadena con universal character names: "<<s1<<std::endl;
18 std::cout<<"Literal de cadena sin universal character names: "<<s2<<std::endl;
19 std::cout<<"Literal de cadena con universal character names UTF8 convertido stream de bytes: "<<reinterpret_cast<const char*>(s
20 std::cout<<"Literal de cadena con universal character names UTF8 convertido a stream de bytes: "<<reinterpret_cast<const char*>
21
22
23 // UTF-16
24
25 const wchar_t s5[] {L"Halag\u00fce\u00f1o\u000D\u0011E"};
26
27 const char16_t s6[] {u"Halag\u00fce\u00f1o\u000D\u0011E"};
28 //Streaming out wide chars
29 std::wcout<<s5<<std::endl;
30 std::cout<<reinterpret_cast<const char*>(s6)<<std::endl;
31
32 std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> converter;

```

Figure 1: Configurando la herramienta online onlinegdb y copiando el código.

```

Literal de cadena con universal character names: Halagüño
Literal de cadena sin universal character names: Halagüño
Literal de cadena con universal character names UTF8 convertido stream de bytes: Halagüño
Literal de cadena con universal character names UTF8 convertido a stream de bytes: Halagüño
Halagüño
H
Literal cadena utf-16 y u16string convertido a utf-8: Halagüño

...Program finished with exit code 0
Press ENTER to exit console.

```

Figure 2: Salida del programa en la terminal de la herramienta.

La ejecución del código produce en la terminal de la herramienta la salida que se muestra en la figura 2. Observen que las cadenas `s1` y `s2` producen el mismo resultado, esto es la salida correcta para la palabra “Halagüeño”. La razón es la siguiente:

- La terminal de la herramienta funciona correctamente para la codificación UTF8. Recuerden que UTF8 es una de las posibles serializaciones de Unicode.
- La terminal de la herramienta incluye las fuentes necesarias para generar los glifos correctos. Esto puede parecer evidente para este ejemplo donde no hay caracteres muy particulares, pero en otras circunstancias es un factor a analizar si un carácter no acaba mostrándose.
- El editor de la herramienta codifica el texto introducido en UTF8, pero en la primera fase de la compilación, el fuente es traducido al denominado basic character set, que es en esencia US-ASCII. Los caracteres no representables en US-ASCII son sustituidos por su Universal Character Name (por ejemplo `\uxxxx` o `\Uxxxxxxxx`)
- Los strings `s1` y `s2` son inicializados con un literal de cadena que se compone de chars.
- Un literal cadena, tras su compilación está formado por las unidades de código resultantes de la aplicación de la codificación que le corresponde (ver apéndice).
- El stream de salida `std::cout` de C++ envía la secuencia de chars a la terminal. Por lo tanto ésta recibe un stream de chars. Los valores de estos chars son las unidades de código de la codificación UTF-8 porque el execution character set del compilador utilizado es UTF-8. La terminal está configurada para interpretar la codificación UTF-8 y también los representará correctamente ya que dispone de las fuentes necesarias para ello.
- La cadena `s1` y `s2` reciben literales equivalentes por lo siguiente. En `s1` los caracteres “ü” y “ñ” son representados por el universal character name y la cadena completa es codificada en UTF-8 que es el execution character set del compilador. En `s2`, sin embargo, el programador escribió directamente los caracteres y el editor de texto guardó los mismos en el código fuente utilizando UTF-8. Sin embargo, el compilador antes de empezar tuvo que traducir este código UTF8 al basic character set y sustituir los caracteres no representables (estos dos), por su universal character name. Por eso `s2` es equivalente a `s1`.

Los caracteres que no están incluidos en US-ASCII son “ü” y “ñ”. Estos caracteres sí tienen una representación en Extended - ASCII, con los códigos 252 (0xFC) y 241 (0xF1) respectivamente. El Extended ASCII fue introducido en 1981 por la compañía IBM.


El Extended ASCII coincide con los puntos de código del U0000 al U00FF de Unicode. Estos dos caracteres están concretamente en el bloque Latin 1 Supplement que va del U0080 al U00FF. Así los puntos de código serían:

- **ü**: U+00FC
- **ñ**: U+00F1


Como hemos visto, las cadenas `s1` y `s2` representan el mismo texto: “Halagüeño”, siendo los puntos de código:


- **H**: U+0048
- **a**: U+0061
- **l**: U+006C
- **a**: U+0061
- **g**: U+0067

- **ü**: U+00FC
- **e**: U+0065
- **ñ**: U+00F1
- **o**: U+006F

Pasemos ahora al símbolo . Este símbolo está en el bloque “Musical Symbols” de Unicode (rango 1D100 a 1D1FF), con el punto de código U+1D11E. Este carácter está incluido en el literal de cadena que se asigna a s4. El literal de cadena es más concretamente de tipo UTF8, por eso se escribe con el prefijo u8 delante:

```
u8"Halag\u00fce\u00f1o\u0001D11E"
```


Esto significa que el compilador g++ tomará los caracteres que componen la cadena y los codificará bajo el esquema de serialización UTF8 (en este caso), incluido el símbolo , que en el código se expresa mediante un universal character name de 8 cifras hexadecimales.

La impresión de las cadenas s5 y s6 no es correcta. La razón es que nuestra terminal no trabaja con la codificación UTF32 o UTF16 sino con la UTF8. Por eso es necesario utilizar un proceso de conversión de UTF16 a UTF8 para poder imprimir, como vemos en la última parte del código. Además la terminal tiene las fuentes adecuadas para representar los glifos, incluido el de la clave musical .

Las cadenas basadas en wide chars, wchar\_t, como s5, son “peligrosas” porque dependiendo de la implementación el número de bytes destinado a un wchar\_t puede variar. En este caso son 4 bytes (32 bits). El compilador asigna cada carácter de la cadena a una palabra de 32 bits. Por ejemplo, si el código de H es 0x48, en la cadena larga, el mismo carácter es asignado a 0x00000048.

La cadena después es enviada a la salida std::wcout, destinada precisamente a enviar un stream de wcout\_t a la terminal. Así, podemos suponer que lo que la terminal recibe en relación al carácter H es 0x48, 0x00, 0x00, 0x00.

Vemos que la terminal imprimir correctamente el caracter H. La razón es que el código UTF8 0x00 representa el caracter NULL que no tiene ningún efecto en la terminal, mientras que el código UTF8 0x48 coincide con el caracter H.

Por contra vemos que los caracteres ü y ñ **no** se imprimen correctamente. Veamos por qué. El carácter ü está representado en la cadena por 00fc. El literal de cadena tiene el prefijo L, lo que implica que para la codificación se utiliza el execution wide-character set, que en g++ depende del tamaño asignado a wchar\_t que es 32, así que aquí se usa UTF32. Por esto, lo que hace el compilador es que el universal character name 00fc se traduzca a los siguientes cuatro bytes de forma directa: 0xfc, 0x00, 0x00, 0x00. Igual que para la “H” los c’odigos 0x00 son interpretados como NULL y no tienen efecto. Pero a diferencia de la “H”, el c’odigo UTF8 0xfc no se corresponde con el caracter “ü”, ya que la codificación UTF8 del punto de código U+00FC es diferente. Por esta misma razón, no se imprime correctamente ñ ni .

Pasemos al último apartado. Activamos el depurador con el botón debug, ordenamos el comienzo de la depuración con el comando **start** y vamos avanzando en la ejecución de las líneas con el comando **next**. Recuerda que la línea que se imprime en la terminal cuando pulsamos next, aún no se ha ejecutado y tenemos que pulsar next para ejecutarla. En la figura 3 vemos la salida del depurador después de haber ejecutado la asignación de s1 y haber explorado la memoria con el comando:

```
x/25xt s1
```

En la tabla 1 se muestra la disposición en memoria de esta cadena conforme a lo que muestra el depurador, que es la coficación UTF8 de dicha cadena como vamos a comprobar ahora.

La codificación UTF8 se basa en la tabla 2. Todos nuestros caracteres, salvo “ü” y “ñ” están dentro del rango de la primera fila de la tabla de codificación para UTF8. Esto significa básicamente que el punto de código Unicode y el código UTF8 coinciden. Sin embargo esto no es así para los otros dos caracteres.

Por ejemplo, el punto de código para “ü” es U+00FC. Esto nos sitúa en la segunda fila de la tabla de codificación para UTF8. Esta fila nos indica que hacen falta dos bytes para codificar este caracter.

```

(gdb) n
10    const char s1[] {"Halag\u00fce\u00f1o"};
(gdb) n
11    const char s2[] {"Halagüeño"};
(gdb) x/25xb s1
0x7fffffffefeb80: 0x48    0x61    0x6c    0x61    0x67    0xc3    0xbc    0x65
0x7fffffffefeb88: 0xc3    0xb1    0x6f    0x00    0x00    0x00    0x00    0x00
0x7fffffffefeb90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffefeb98: 0x00
(gdb)

```

Figure 3: La variable s1 en la memoria mostrando la codificación UTF-8 de la cadena Halagüeño.

Dirección	Byte	Caracter
0x7fffffffefeb8b	0x00	null
0x7fffffffefeb8a	0x6f	o
0x7fffffffefeb89	0xb1	ñ
0x7fffffffefeb88	0xc3	ñ
0x7fffffffefeb87	0x65	e
0x7fffffffefeb86	0xbc	ü
0x7fffffffefeb85	0xc3	ü
0x7fffffffefeb84	0x67	g
0x7fffffffefeb83	0x61	a
0x7fffffffefeb82	0x6c	l
0x7fffffffefeb81	0x61	a
0x7fffffffefeb80	0x48	H

Table 1: Contenido de la memoria para la variable s1 codificada en UTF8. Las direcciones crecen de la parte inferior de la tabla a la parte superior

Rango del punto de código	Formato de los bytes
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Table 2: Reglas de composición de los bytes UTF-8 para los puntos de código UNICODE.

```

(gdb) n
27   const char16_t s6[] {u"Halag\u00fce\u00f1o\u0001D11E"};
(gdb) n
29   std::wcout<<s5<<std::endl;
(gdb) x/25xb s6
0x7fffffff90: 0x48  0x00  0x61  0x00  0x6c  0x00  0x61  0x00
0x7fffffff98: 0x67  0x00  0xfc  0x00  0x65  0x00  0xf1  0x00
0x7fffffffaa0: 0x6f  0x00  0x34  0xd8  0x1e  0xdd  0x00  0x00
0x7fffffffaa8: 0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
(gdb)

```

Figure 4: Representación en memoria de la cadena s6

Primero escribimos el punto de código en binario (los 11 bits menos significativos que como máximo pueden ser diferentes de 0 en este rango:

El código binario es:

0001111100

Los seis bits menos significativos son:

111100

Los usamos para completar el segundo byte (en la tabla los bytes más a la derecha son los menos significativos):

10111100=0xBC

Ahora completamos el primer byte con los cinco bits más significativos:

11000011=0xC3

Así pues, los dos bytes que resultan de la codificación son: 0xC3, 0xBC (el primero alberga los bits más significativos y el segundo los menos significativos). Finalmente, observen como se disponen en memoria siguiendo una ordenación Big Endian: en la dirección más baja aparece 0xC3 y en la dirección más alta tenemos a 0xBC.

Repetimos el proceso con el punto de código que representa a la “ñ”, que es U+00F1. Está en el mismo rango de la tabla de decodificación, así que sacamos los 11 bits que lo conforman:

00011110001

El byte codificado menos significativo alberga los 6 bits menos significativos con el prefijo 10:


10110001=0xB1

El byte codificado más significativo alberga los 5 bits más significativos con el prefijo 110:

11000011=0xC3

Los dos bytes serían 0xC3 y 0xB1 que se disponen en memoria siguiendo un esquema Big Endian, en la dirección más baja 0xC3 y en la más alta 0xB1.

Terminemos este apartado comprobando los códigos de la cadena s6. Seguimos avanzando con el depurador línea por línea con **next** hasta haber ejecutado la asignación de s6. Después examinamos la memoria haciendo **x/25xb s6**. En la figura 4 pueden ver el resultado:

En la tabla 3 pueden ver más claramente la disposición de los bytes. Se trata de una codificación UTF16 y por tanto los puntos de código del plano básico multi-lingüe, se codifican directamente con 16 bits (2 bytes). Este plano va de U+0000 a U+FFFF de modo que todos los caracteres de nuestro ejemplo, salvo  están dentro. Vemos que se sigue una disposición Little Endian donde el byte menos significativo se coloca primero en la secuencia (dirección más baja).

Dirección	Byte	Caracter
0x7fffffffffeaa6	0x00	null
0x7fffffffffeaa5	0xdd	♩
0x7fffffffffeaa4	0x1e	♩
0x7fffffffffeaa3	0xd8	♩
0x7fffffffffeaa2	0x34	♩
0x7fffffffffeaa1	0x00	o
0x7fffffffffeaa0	0x6f	o
0x7fffffffffea9f	0x00	ñ
0x7fffffffffea9e	0xf1	ñ
0x7fffffffffea9d	0x00	e
0x7fffffffffea9c	0x65	e
0x7fffffffffea9b	0x00	ü
0x7fffffffffea9a	0xfc	ü
0x7fffffffffea99	0x00	g
0x7fffffffffea98	0x67	g
0x7fffffffffea97	0x00	a
0x7fffffffffea96	0x61	a
0x7fffffffffea95	0x00	l
0x7fffffffffea94	0x6c	l
0x7fffffffffea93	0x00	a
0x7fffffffffea92	0x61	a
0x7fffffffffea91	0x00	H
0x7fffffffffea90	0x48	H

Table 3: Contenido de la memoria para la variable s4\_data codificada en UTF16. Las direcciones crecen de la parte inferior de la tabla a la parte superior



Por otra parte, el carácter  $\text{U+1D11E}$  requiere de cuatro bytes o dos códigos de 16 bits. La razón es que su punto de código  $\text{U+1D11E}$  está fuera del plano básico multi-lingüe. Para encontrar los bits de estas palabras de 16 bits, debemos comenzar tomando los 20 bits del punto de código:

$\text{U+1D11E}=0001-1101-0001-0001-1110$

Ahora hay que restarle la cantidad  $0x10000$ . Esta resta equivale a realizar la suma binaria con el resultado de  $\text{NEG}(0x10000)=0xF0000$ . A continuación tienen la suma binaria:

1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	1	0	0	0	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	1	1	1	1	0

Los 20 bits del resultado son:

$0000-1101-0001-0001-1110$

El prefijo para la segunda palabra del par subrogado (la menos significativa) es  $110111$ . Los 10 bits menos significativos del resultado de la operación anterior van a continuación para formar:

$1101-1101-0001-1110=0xDD1E$

El prefijo para la primera palabra del par subrogado (la más significativa) es  $110110$ . Los 10 bits más significativos de la resta irían a continuación para formar:

$1101-1000-0011-0100=0xD834$

La palabra más significativa del par subrogado va primero en memoria (Big Endian). Los bytes de la palabra se almacenan conforme a Little Endian porque este es el Endianess de la máquina, primero el  $0x34$  y luego el  $0xD8$ .

La palabra menos significativa del par subrogado va a continuación en memoria (Big Endian). Los bytes de la palabra se almacenan en orden Little Endian porque este es el Endianess de la máquina. Primero el  $0x1E$  y luego el  $0xDD$ .

□

## 2 Apéndice: literales de caracteres y literales de cadena en C++ (C++20)

Este apéndice es un extracto de la documentación oficial de C++ y salvo que se especifique lo contrario se refiere a la versión C++20. Hay que tener en cuenta que algunos de estos aspectos han cambiado en la versión C++23.

Para entender el funcionamiento de los caracteres y las cadenas de caracteres es necesario ver primero como el estándar C++20 organiza la codificación de los mismos.

### 2.1 Basic Character Set y Execution Character Set

Para empezar, nuestro código fuente estará escrito en un documento de texto realizado con un editor, el cual aplicará cierta codificación (por ejemplo UTF-8, Latin-9, u otra). Este texto así codificado, le llega al compilador, que obedeciendo al estándar de C++ deberá antes que nada, traducir los códigos de estos caracteres al conjunto denominado por el estándar “basic character set”, y que consiste en una parte de los 128 caracteres de la tabla US-ASCII.

Por lo tanto, surge la pregunta ¿qué pasa con los caracteres que se encuentren en un archivo de código fuente que no estén en el basic character set? En ese caso, el carácter es sustituido por el denominado universal-character-name.

El universal character name se define por el estándar como el patrón: `\uxxxx` o `\Uxxxxxxxx`, donde la `x` es una cifra hexadecimal. El conjunto de cifras decimales es el punto de código Unicode del carácter.

El estándar C++ establece además dos conjuntos de ejecución denominados: **execution character set** y **execution wide-character set**. Estos conjuntos son superconjuntos del basic literal character set y son los conjuntos que establecen una codificación binaria para los caracteres en las formas básicas de literales denominadas ordinarias tras la compilación y en la ejecución.

Antes de pasar a ver la diferencia entre ambos conjuntos hay que comprender el concepto de “unidad de código” o code unit en el contexto de la codificación de caracteres. El code unit es la celda de bits que forma el bloque básico de la codificación. Por ejemplo, en el caso de UTF-9 la unidad de código es 1 byte (aunque un carácter en UTF-8 pueda requerir varios bytes), mientras que la unidad de código de UTF-16 es una celda de 16 bits.

Entonces, la diferencia entre ambos conjuntos es que el execution wide-character set se suele orientar a la representación de caracteres cuya unidad de código requiere mayor número de bits que el execution character set. Mientras que el tipo de dato utilizado para literales destinados al execution char set es `char`, el correspondiente a los literales destinados al execution wide-char set es `wchar_t`.

Los compiladores suelen tener opciones para elegir el execution character set y el execution wide-character set de entre diferentes posibilidades disponibles. El compilador GNU CPP por ejemplo, tiene por defecto como execution character set, el esquema de serialización UTF-8 de Unicode, mientras que el execution wide character set puede ser UTF-16 o UTF-32 dependiendo del número de bytes de `wchar_t`. Este compilador, admite las opciones `-fexec-charset` y `-fwide-exec-charset` para especificar los conjuntos. Las opciones en su implementación en linux admiten cualquiera de las codificaciones soportadas por la librería `iconv`. Si ejecutas:

```
iconv --list
```

puedes ver todas las codificaciones soportadas en tu sistema.

## 2.2 Posibles formas de literales de caracteres en C++

Para entender un literal de carácter hemos de comenzar por ver qué es un c-char. Puede ser:

- Un basic-char: Un carácter del basic character set, es decir, la mayoría de los caracteres ASCII (entre U+0000 y U+007E), excepto la comilla simple, el carácter “\” o el carácter de nueva línea.
- Una secuencia escape definida en el estándar: como por ejemplo `\t` (tabulador) o `\’` (comilla simple).
- Un “Universal Character Name”: como `\u00F1` (punto de código con 4 cifras hexadecimales) o `\U0001D11E` (punto de código con 8 cifras hexadecimales).

En C++20 hay 7 formas de literales de carácter que se construyen con c-char:

- **’c-char’**: Se denomina literal de carácter ordinario. El literal tiene tipo `char` y su valor viene determinado por el execution character set. Si el c-char especificado no tiene un código en el execution character set, el literal de este modo podría no ser soportado por el compilador, y de serlo tendría tipo `int` y un valor definido dependiendo de la implementación.
- **L’c-char’**: Se denomina literal de carácter amplio. El literal tiene tipo `wchar_t` y su valor viene determinado por el execution wide character set. Si el c-char especificado no tiene un código en el execution wide character set, el literal de este modo podría no ser soportado por el compilador, y de serlo tendría tipo `wchar_t` y un valor definido dependiendo de la implementación.
- **u8’c-char’**: Se denomina literal de carácter UTF-8. Existe desde C++17. En esta versión este literal tiene tipo `char`, mientras que en C++20 tiene tipo `char8_t` y su valor viene dado por el estándar IEC10646 siempre que el punto de código requiera solo una unidad de código de UTF-8.
- **u’c-char’**: Se denomina literal de carácter UTF-16. Existe desde C++11. Tiene tipo `char16_t` igual al punto de código en IEC10646, siempre que el punto de código sea representable con una sola unidad de código de UTF-16.

- **U‘c-char’**: Se denomina literal de carácter UTF-32. Este literal tiene tipo `char32_t` y su valor es igual al punto de código en el estándar ISO 10646. Si el c-char en los tres tipos anteriores no encaja en una unidad de código el literal se considera mal formado.
- **‘secuencia de c-char’**. Se denomina literal de multi-carácter ordinario. Se trata de varios c-char cuya traducción como valor se realiza a un tipo `int`. No siempre es soportado y la forma de hacerlo depende de la implementación del compilador.
- **L‘secuencia de c-char’**. Se denomina literal de multi-carácter amplio ordinario. Se trata de varios c-char cuya traducción como valor se realiza a un tipo `wchar_t`. No siempre es soportado y la forma de hacerlo depende de la implementación del compilador.

Además hay que tener en cuenta que una secuencia de escape octal o hexadecimal, puede ser utilizada para especificar directamente el código del carácter.

## 2.3 Literales de cadenas

Para entender el literal de cadena hemos de comenzar por ver qué es un s-char. Puede ser:

- Un basic-s-char: Un carácter del basic character set, es decir, la mayoría de los caracteres ASCII (entre U+0000 y U+007E), excepto la doble comilla, el carácter “\” o el carácter de nueva línea.
- Una secuencia escape definida en el estándar: como por ejemplo `\t` (tabulador) o `\’` (comilla simple).
- Un “Universal Character Name”: como `\u00F1` (punto de código con 4 cifras hexadecimales) o `\U0001D11E` (punto de código con 8 cifras hexadecimales).

Entonces tenemos 6 formas de literales de cadena:

- **“Secuencia de s-char”**: Se denomina literal de cadena ordinario. La codificación utiliza el execution character set. El tipo es `const char[N]`, donde *N* es el número de unidades de código empleadas incluyendo un terminador (carácter nulo `\0`).
- **L“Secuencia de s-char”**: Se denomina literal de cadena amplia. La codificación utiliza el execution wide character set. El tipo es `const wchar_t[N]`.
- **u8“Secuencia de s-char”**: Se denomina literal de cadena UTF-8. El tipo es `const char8_t[N]` en C++20 y `const char [N]` en versiones previas, siendo *N* el tamaño de la cadena en unidades de código de UTF-8.
- **u“Secuencia de s-schar”**: Se denomina literal de cadena UTF-16. El tipo es `const char16_t[N]`
- **U“Secuencia de s-schar”**: Se denomina literal de cadena UTF-32. El tipo es `const char32_t[N]`
- **Cadenas raw**: En un literal de cadena raw no se procesan las secuencias de escape. El formato es `R“delimitador(cadena)delimitador”` El delimitador es opcional, puede ser cualquier secuencia de caracteres mientras no contenga `\`, paréntesis o espacios. Dentro de la cadena no se puede incluir el terminador, es decir `)delimitador`. La cadena solo puede incluir caracteres del basic character set.