
16.1. Character Encodings and Character Sets

At the beginning of the age of computer science the character set of computers was limited to the characters of the English alphabet. Today in the area of globalization, there are character set standards of up to 32 bits, with more than 1 million different character values.[2] As a consequence, there are different standards and approaches to deal with characters in different countries and cultures.

[2] Current 32-bit character sets use the values up to `0x10FFFF`, which are 1,114,111 values.

16.1.1. Multibyte and Wide-Character Text

Two different approaches are common to address character sets that have more than 256 characters: multibyte representation and wide-character representation:

1. With *multibyte representation*, the number of bytes used for a character is variable. A 1-byte character, such as an ISO-Latin-1 character, can be followed by a 3-byte character, such as a Japanese ideogram.
2. With *wide-character representation*, the number of bytes used to represent a character is always the same, independent of the character being represented. Typical representations use 2 or 4 bytes. Conceptually, this does not differ from representations that use just 1 byte for locales where ISO-Latin-1 or even ASCII is sufficient.

Multibyte representation is more compact than wide-character representation. Thus, the multibyte representation is normally used to store data outside of programs. Conversely, it is much easier to process characters of fixed size, so the wide-character representation is usually used inside programs.

In a multibyte string, the same byte may represent a character or even just a part of the character. During iteration through a multibyte string, each byte is interpreted according to a current "shift state." Depending on the value of the byte and the current shift state, a byte may represent a certain character or a change of the current shift state. A multibyte string always starts in a defined initial shift state. For example, in the initial shift state, the bytes may represent ISO-Latin-1 characters until an escape character is encountered. The character following the escape character identifies the new shift state. For example, that character may switch to a shift state in which the bytes are interpreted as Arabic characters until the next escape character is encountered.

16.1.2. Different Character Sets

The most important character sets are:

- **US-ASCII**, a 7-bit character set standardized since 1963 for teleprinters and other devices, so that the first 16 values are "nonprintable characters," such as carriage-return, horizontal tab, backspace, or a bell. This character set serves as base for all other character sets, and usually the values between `0x20` and `0x7F` have the same characters in all other character sets.
- **ISO-Latin-1** or **ISO-8859-1** (see [*ISOLatin1*]), an 8-bit character set, standardized since 1987 to provide all characters of the "Western Europe" languages. Also, this character set serves as base for all other character sets, and usually the values between `0x20` and `0x7F` and from `0xA0` to `0xFF` have the same characters in all other character sets.
- **ISO-Latin-9** or **ISO-8859-15** (see [*ISOLatin9*]), an 8-bit character set, standardized since 1999 to provide an improved version of all characters of the "Western Europe" languages by replacing some less common symbols with the euro sign and other special characters.
- **UCS-2**, a 16-bit fixed-sized character set, providing the 65,536 most important characters of the *Universal Character Set* and *Unicode* standards.
- **UTF-8** (see [*UTF8*]), a multibyte character-set using between one and four *octets* of 8 bits to represent all characters of the *Universal Character Set* and *Unicode* standards. It is widely used in the world of the World Wide Web.
- **UTF-16**, a multibyte character-set using between one and two *code units* of 16 bits to represent all characters of the *Universal Character Set* and *Unicode* standards.
- **UCS-4** or **UTF-32**, a 32-bit fixed-sized character set, providing all standardized characters of the *Universal Character Set* and *Unicode* standards. Note that UTF-16 and UTF-32 might have a *byte order mark* (*BOM*) at the beginning of the whole character sequence to mark whether *big-endian* (default) or *little-endian* byte order is used. Alternatively, you can explicitly specify UTF-16BE, UTF-16LE, UTF-32BE, or UTF-32LE.

Figure 16.1 shows the different hexadecimal encodings of an example character sequence, using ordinary ASCII characters, the German umlaut `ä`, and the euro symbol €. Here, UTF-16 and UTF-32 use no byte order marks. A byte order mark would have the value `0xFEFF`.

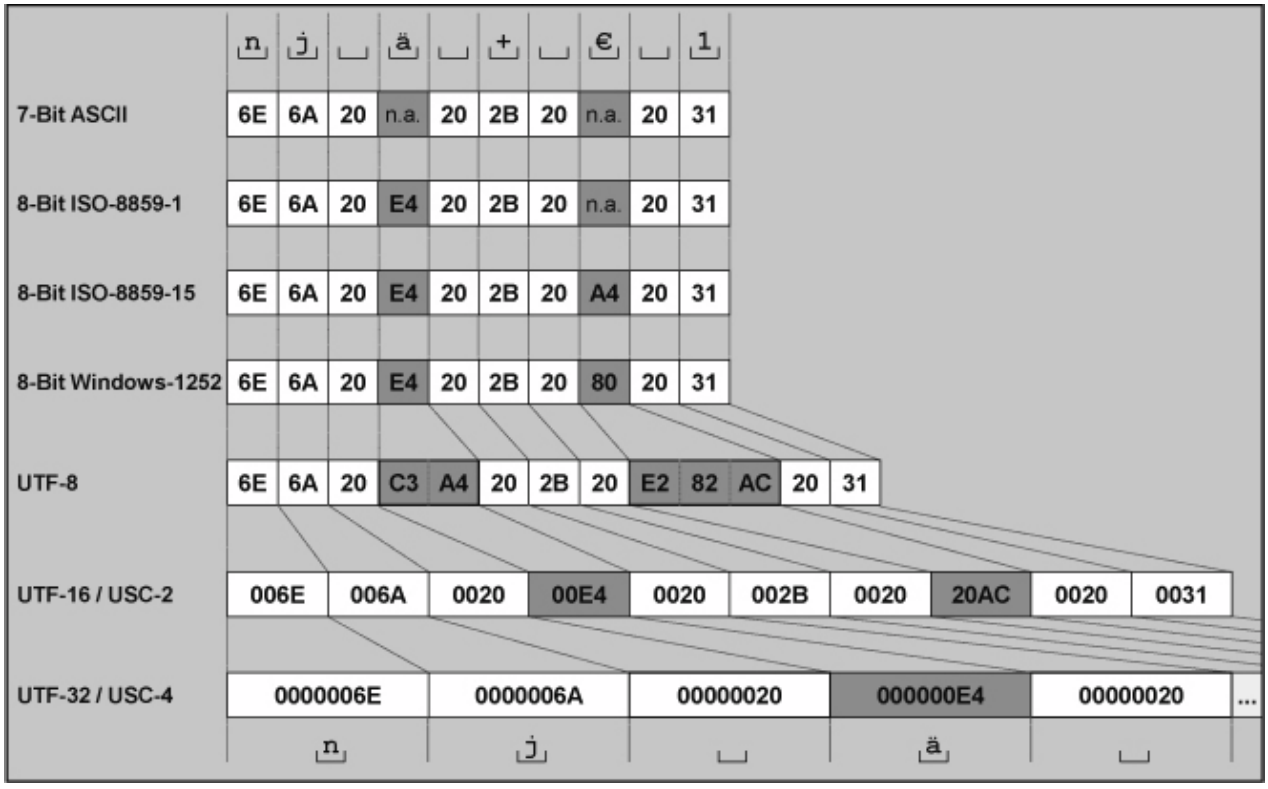

Figure 16.1. Hexadecimal Character Codes of Different Character Sets

Note that UTF-16 and UCS-2 almost match for the characters up to the value `0xFFFF`. Only for very special characters not available in UCS-2, UTF-16 uses two code units of 16 bits given that UCS-2 is a multibyte character set.

16.1.3. Dealing with Character Sets in C++

C++ provides different character types to deal with these character sets:

- **char** can be used for all character sets up to 8 bits, such as US-ASCII, ISO-Latin-1, and ISO-Latin-9. In addition, it can be used for octets of UTF-8.
- **char16_t** (provided since C++11) can be used for UCS-2 and as code unit for UTF-16.
- **char32_t** (provided since C++11) can be used for UCS-4/UTF-32.
- **wchar_t** is the type for the values of the largest extended character set among all supported locales. Thus, it is usually equivalent to `char16_t` or `char32_t`.

All these types are keywords, so it is possible to overload functions with all these types. Note, however, that the support of `char16_t` and `char32_t` is limited. Although character traits provide the ability to deal with Unicode strings, no overloads for these types for I/O are provided.

Note that since C++11, you can specify string literals using different character encodings (see Section 3.1.6, page 24, for details).

To support character and code conversions, the C++ standard library provides the following features:

- To convert `string`s into `wstring`s and vice versa, you can use the member functions `widen()` and `narrow()` of the `ctype<>` facet (see Section 16.4.4, page 891). Note that they also can be used to convert characters of the native character set into characters of a locale's character set, both using the character type `char`.
- To convert multibyte sequences into `wstring`s and vice versa, you can use the class template `wstring_convert<>` and the corresponding `codecvt<>` facets (see Section 16.4.4, page 901).
- Class `codecvt<>` (see Section 16.4.4, page 897) is also used by class `basic_filebuf<>` (see Section 15.9.1, page 791) to convert between internal and external representations when reading or writing files.
- To read or write multibyte character sequences, you can use class `wbuffer_convert<>` and the corresponding `codecvt<>` facets (see Section 16.4.4, page 903).

16.1.4. Character Traits

The different representations of character sets imply variations that are relevant for the processing of strings and I/O. For example, the value used to represent "end-of-file" or the details of comparing characters may differ for representations.

The string and stream classes are intended to be instantiated with built-in types, especially with `char` and `wchar_t`, and, since C++11, maybe with `char16_t` and `char32_t`. The interface of built-in types cannot be changed. Thus, the details on how to deal with aspects that depend on the representation are factored into a separate class, a so-called *traits class*. Both the string and stream classes take a traits class as a template argument. This argument defaults to the class `char_traits`, parametrized with the template argument that defines the character type of the string or stream:

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT>>
            class basic_string;
}

namespace std {
   template <typename charT,
             typename traits = char_traits<charT>>
            class basic_istream;
   template <typename charT,
             typename traits = char_traits<charT>>
            class basic_ostream;
   ...
}
```

The character traits have type `char_traits<>`. This type is defined in `<string>` and is parametrized on the specific character type:

```
namespace std {
   template <typename charT>
   struct char_traits {
      ...
   };
}
```

The traits classes define all fundamental properties of the character type and the corresponding operations necessary for the implementation of strings and streams as static components. Table 16.1 lists the members of `char_traits`.

Table 16.1. Character Traits Members

Expression	Meaning
`char_type`	The character type (the template argument for `char_traits`)
`int_type`	A type large enough to represent an additional, otherwise unused value for end-of-file
`pos_type`	A type used to represent positions in streams
`off_type`	A type used to represent offsets between positions in streams
`state_type`	A type used to represent the current state in multibyte streams
`assign(`*c1*`,`*c2*`)`	Assigns character *c2* to *c1*
`eq(`*c1*`,`*c2*`)`	Returns whether the characters *c1* and *c2* are equal
`lt(`*c1*`,`*c2*`)`	Returns whether character *c1* is less than character *c2*
`length(`*s*`)`	Returns the length of the string *s*
`compare(`*s1*`,`*s2*`,`*n*`)`	Compares up to *n* characters of strings *s1* and *s2*
`copy(`*s1*`,`*s2*`,`*n*`)`	Copies *n* characters of string *s2* to string *s1*
`move(`*s1*`,`*s2*`,`*n*`)`	Copies *n* characters of string *s2* to string *s1*, where *s1* and *s2* may overlap
`assign(`*s*`,`*n*`,`*c*`)`	Assigns the character *c* to *n* characters of string *s*
`find(`*s*`,`*n*`,`*c*`)`	Returns a pointer to the first character in string *s* that is equal to *c* or `nullptr` if no such character is among the first *n* characters
`eof()`	Returns the value of end-of-file
`to_int_type(`*c*`)`	Converts the character *c* into the corresponding representation as `int_type`
`to_char_type(`*i*`)`	Converts the representation *i* as `int_type` to a character (the result of converting EOF is undefined)
`not_eof(`*i*`)`	Returns the value *i* unless *i* is the value for EOF; in this case, an implementation-dependent value different from EOF is returned
`eq_int_type(`*i1*`,`*i2*`)`	Tests the equality of the two characters *i1* and *i2* represented as `int_type` (the characters may be EOF)

The functions that process strings or character sequences are present for optimization only. They could also be implemented by using the functions that process single characters. For example, `copy()` can be implemented using `assign()`. However, there might be more efficient implementations when dealing with strings.

Note that counts used in the functions are exact counts, not maximum counts. That is, string-termination characters within these sequences are ignored.

The last group of functions concerns the special processing of the character that represents endof-file (EOF). This character extends the character set by an artificial character to indicate special processing. For some representations, the character type may be insufficient to accommodate this special character because it has to have a value that differs from the values of all "normal" characters of the character set. C established the convention to return a character as `int` instead of as `char` from functions reading characters. This technique was extended in C++. The character traits define `char_type` as the type to represent all characters and `int_type` as the type to represent all characters plus EOF. The functions `to_char_type()`, `to_int_type()`, `not_eof()`, and `eq_int_type()` define the corresponding conversions and comparisons. It is possible that `char_type` and `int_type` are identical for some character traits. This can be the case if not all values of `char_type` are necessary to represent characters so that a spare value can be used for end-of-file.

`pos_type` and `off_type` are used to define file positions and offsets, respectively (see Section 15.9.4, page 799, for details).

The C++ standard library provides specializations of `char_traits<>` for types `char` and `wchar_t` and, since C++11, for `char16_t` and `char32_t`:

```
namespace std {
  template<> struct char_traits<char>;
  template<> struct char_traits<wchar_t>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32>;
}
```

The specialization for `char` is usually implemented by using the global string functions of C that are defined in `<cstring>` or `<string.h>`. An implementation might look as follows:

Click here to view code image

```
namespace std {
 template<> struct char_traits<char> {
  // type definitions:
  typedef char      char_type;
  typedef int       int_type;
  typedef streampos pos_type;
  typedef streamoff off_type;
  typedef mbstate_t state_type;

  // functions:
  static void assign(char& c1, const char& c2) {
     c1 = c2;
  }
  static bool eq(const char& c1, const char& c2) {
     return c1 == c2;
  }
  static bool lt(const char& c1, const char& c2) {
     return c1 < c2;
  }
  static size_t length(const char* s) {
     return strlen(s);
  }
  static int compare(const char* s1, const char* s2, size_t n) {
     return memcmp(s1,s2,n);
  }
  static char* copy(char* s1, const char* s2, size_t n) {
     return (char*)memcpy(s1,s2,n);
  }
  static char* move(char* s1, const char* s2, size_t n) {
     return (char*)memmove(s1,s2,n);
  }
  static char* assign(char* s, size_t n, char c) {
     return (char*)memset(s,c,n);
  }
  static const char* find(const char* s, size_t n,
                          const char& c) {
     return (const char*)memchr(s,c,n);
  }
  static int eof() {
     return EOF;
  }
  static int to_int_type(const char& c) {
     return (int)(unsigned char)c;
  }
  static char to_char_type(const int& i) {
     return (char)i;
  }
  static int not_eof(const int& i) {
     return i!=EOF ? i : !EOF;
  }
  static bool eq_int_type(const int& i1, const int& i2) {
     return i1 == i2;
  }
 };
}
```

See Section 13.2.15, page 689, for the implementation of a user-defined traits class that lets strings behave in a case-insensitive manner.

16.1.5. Internationalization of Special Characters

One issue with character encodings remains: How are special characters, such as the newline or the string termination character, internationalized? The class `basic_ios<>` has members `widen()` and `narrow()` that can be used for this purpose. Thus, the newline character in an encoding appropriate for the stream `strm` can be written as follows:

```
strm.widen('\n')   // internationalized newline character
```

The string-termination character in the same encoding can be created like this:

```
strm.widen('\0')   // internationalized string-termination character
```

See the implementation of the `endl` manipulator in Section 15.6.2, page 777, for an example use.

The functions `widen()` and `narrow()` use a locale object: more precisely, the `ctype<>` facet of this object. This facet can be used to convert all characters between `char` and some other character representations. It is described in Section 16.4.4, page 891. For example, the following expression converts the character `c` of type `char` into an object of type `charType` by using the locale object `loc` :

```
std::use_facet<std::ctype<charType>>(loc).widen(c)
```

The details of the use of locales and their facets are described in the following sections.