String and Character Literals (C++)

Visual Studio 2015

For the latest documentation on Visual Studio 2017, see Visual Studio 2017 Documentation.

C++ supports various string and character types, and provides ways to express literal values of each of these types. In your source code, you express the content of your character and string literals using a character set. Universal character names and escape characters allow you to express any string using only the basic source character set. A raw string literal enables you to avoid using escape characters, and can be used to express all types of string literals. You can also create std::string literals without having to perform extra construction or conversion steps.

C++

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string literals

int main()
{
    // Character literals
    auto c0 =   'A'; // char
    auto c1 = u8'A'; // char
    auto c2 =  L'A'; // wchar_t
    auto c3 =  u'A'; // char16_t
    auto c4 =  U'A'; // char32_t

    // String literals
    auto s0 =   "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 =  L"hello"; // const wchar_t*
    auto s3 =  u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 =  U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 =   R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char*, encoded as UTF-8
    auto R2 =  LR"("Hello \ world")"; // const wchar_t*
    auto R3 =  uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
    auto R4 =  UR"("Hello \ world")"; // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 =   "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string
    auto S2 =  L"hello"s; // std::wstring
    auto S3 =  u"hello"s; // std::u16string
    auto S4 =  U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 =   R"("Hello \ world")"s; // std::string from a raw const char*
    auto S6 = u8R"("Hello \ world")"s; // std::string from a raw const char*, encoded as
UTF-8
    auto S7 =  LR"("Hello \ world")"s; // std::wstring from a raw const wchar_t*
    auto S8 =  uR"("Hello \ world")"s; // std::u16string from a raw const char16_t*, encoded
as UTF-16
    auto S9 =  UR"("Hello \ world")"s; // std::u32string from a raw const char32_t*, encoded
as UTF-32
```

```
}
```

String literals can have no prefix, or `u8`, `L`, `u`, and `U` prefixes to denote narrow character (single-byte or multi-byte), UTF-8, wide character (UCS-2 or UTF-16), UTF-16 and UTF-32 encodings, respectively. A raw string literal can have `R`, `u8R`, `LR`, `uR` and `UR` prefixes for the raw version equivalents of these encodings. To create temporary or static std::string values, you can use string literals or raw string literals with an `s` suffix. For more information, see the String literals section below. For more information on the basic source character set, universal character names, and using characters from extended codepages in your source code, see Character Sets.

Character literals

A *character literal* is composed of a constant character. It is represented by the character surrounded by single quotation marks. There are five kinds of character literals:

- Ordinary character literals of type `char`, for example `'a'`
- UTF-8 character literals of type `char`, for example `u8'a'`
- Wide-character literals of type `wchar_t`, for example `L'a'`
- UTF-16 character literals of type `char16_t`, for example `u'a'`
- UTF-32 character literals of type `char32_t`, for example `U'a'`

The character used for a character literal may be any character, except for the reserved characters backslash ('\'), single quotation mark ('), or new line. Reserved characters can be specified by using an escape sequence. Characters may be specified by using universal character names, as long as the type is large enough to hold the character.

Encoding

Character literals are encoded differently based their prefix.

- A character literal without a prefix is an ordinary character literal. The value of an ordinary character literal containing a single character, escape sequence, or universal character name that can be represented in the execution character set has a value equal to the numerical value of its encoding in the execution character set. An ordinary character literal that contains more than one character, escape sequence, or universal character name is a *multicharacter literal*. A multicharacter literal or an ordinary character literal that can't be represented in the execution character set is conditionally-supported, has type int, and its value is implementation-defined.
- A character literal that begins with the L prefix is a wide-character literal. The value of a wide-character literal containing a single character, escape sequence, or universal character name has a value equal to the numerical value of its encoding in the execution wide-character set unless the character literal has no representation in the execution wide-character set, in which case the value is implementation-defined. The value of a wide-character literal containing multiple characters, escape sequences, or universal character names is implementation-defined.
- A character literal that begins with the u8 prefix is a UTF-8 character literal. The value of a UTF-8 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value if it can be represented by a single UTF-8 code unit (corresponding to the C0 Controls and Basic Latin Unicode block). If the value can't be represented by a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing more than one character, escape sequence, or universal character name is ill-formed.
- A character literal that begins with the u prefix is a UTF-16 character literal. The value of a UTF-16 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value if it can be represented by a single UTF-16 code unit (corresponding to the basic multi-lingual plane). If

the value can't be represented by a single UTF-16 code unit, the program is ill-formed. A UTF-16 character literal containing more than one character, escape sequence, or universal character name is ill-formed.

- A character literal that begins with the U prefix is a UTF-32 character literal. The value of a UTF-32 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value. A UTF-8 character literal containing more than one character, escape sequence, or universal character name is ill-formed.

Escape Sequences

There are three kinds of escape sequences: simple, octal, and hexadecimal. Escape sequences may be any of the following:

Value	Escape sequence	Value	Escape sequence
newline	\n	backslash	\\
horizontal tab	\t	question mark	? or \?
vertical tab	\v	single quote	\'
backspace	\b	double quote	\"
carriage return	\r	the null character	\0
form feed	\f	octal	\ooo
alert (bell)	\a	hexadecimal	\xhhh

The following code shows some examples of escaped characters using ordinary character literals. The same escape sequence syntax is valid for the other character literal types.

C++

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

int main() {
    char newline = '\n';
    char tab = '\t';
    char backspace = '\b';
    char backslash = '\\';
    char nullChar = '\0';

    cout << "Newline character: " << newline << "ending" << endl; // Newline character:
                                                                  //  ending
    cout << "Tab character: " << tab << "ending" << endl; // Tab character : ending
    cout << "Backspace character: " << backspace << "ending" << endl; // Backspace character
: ending
    cout << "Backslash character: " << backslash << "ending" << endl; // Backslash character
: \ending
    cout << "Null character: " << nullChar << "ending" << endl; //Null character:  ending
}
```

Microsoft Specific

To create a value from an ordinary character literal (those without a prefix), the compiler converts the character or character sequence between single quotes into 8-bit values within a 32-bit integer. Multiple characters in the literal fill corresponding bytes as needed from high-order to low-order. To create a `char` value, the compiler takes the low-order byte. To create a `wchar_t` or `char16_t` value, the compiler takes the low-order word. The compiler warns that the result is truncated if any bits are set above the assigned byte or word.

C++

```
char c0    = 'abcd';    // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';    // C4305, C4309, truncates to '\x6364'
```

An octal escape sequence is a backslash followed by a sequence of up to 3 octal digits. The behavior of an octal escape sequence that appears to contain more than three digits is treated as a 3-digit octal sequence followed by the subsequent digits as characters; this can give surprising results. For example:

C++

```
char c1 = '\100';   // '@'
char c2 = '\1000';  // C4305, C4309, truncates to '0'
```

Escape sequences that appear to contain non-octal characters are evaluated as an octal sequence up to the last octal character, followed by the remaining characters. For example:

C++

```
char c3 = '\009';   // '9'
char c4 = '\089';   // C4305, C4309, truncates to '9'
char c5 = '\qrs';   // C4129, C4305, C4309, truncates to 's'
```

A hexadecimal escape sequence is a backslash followed by the character `x`, followed by a sequence of hexadecimal digits. An escape sequence that contains no hexadecimal digits causes compiler error C2153: "hex literals must have at least one hex digit". Leading zeroes are ignored. An escape sequence that appears to have hexadecimal and non-hexadecimal characters is evaluated as a hexadecimal escape sequence up to the last hexadecimal character, followed by the non-hexadecimal characters. In an ordinary or u8-prefixed character literal, the highest hexadecimal value is 0xFF. In an L-prefixed or u-prefixed wide character literal, the highest hexadecimal value is 0xFFFF. In a U-prefixed wide character literal, the highest hexadecimal value is 0xFFFFFFFF.

C++

```
char c6 = '\x0050'; // 'P'
char c7 = '\x0pqr'; // C4305, C4309, truncates to 'r'
```

If a wide character literal prefixed with `L` contains more than one character, the value is taken from the first character. Subsequent characters are ignored, unlike the behavior of the equivalent ordinary character literal.

C++

```
wchar_t w1 = L'\100';   // L'@'
wchar_t w2 = L'\1000';  // C4066 L'@', 0 ignored
wchar_t w3 = L'\009';   // C4066 L'\0', 9 ignored
wchar_t w4 = L'\089';   // C4066 L'\0', 89 ignored
```

```
wchar_t w5 = L'\qrs';   // C4129, C4066 L'q' escape, rs ignored
wchar_t w6 = L'\x0050'; // L'P'
wchar_t w7 = L'\x0pqr'; // C4066 L'\0', pqr ignored
```

END Microsoft Specific

The backslash character (\) is a line-continuation character when it is placed at the end of a line. If you want a backslash character to appear as a character literal, you must type two backslashes in a row (\\). For more information about the line continuation character, see Phases of Translation.

Universal character names

In character literals and native (non-raw) string literals, any character may be represented by a universal character name. Universal character names are formed by a prefix \U followed by an eight-digit Unicode code point, or by a prefix \u followed by a four digit Unicode code point. All eight or four digits, respectively, must be present to make a well-formed universal character name.

C++

```
char u1 = 'A';          // 'A'
char u2 = '\101';       // octal, 'A'
char u3 = '\x41';       // hexadecimal, 'A'
char u4 = '\u0041';     // \u UCN 'A'
char u5 = '\U00000041'; // \U UCN 'A'
```

Surrogate Pairs

Universal character names cannot encode values in the surrogate code point range D800-DFFF. For Unicode surrogate pairs, specify the universal character name by using \UNNNNNNNN, where NNNNNNNN is the eight-digit code point for the character. The compiler generates a surrogate pair if required.

In C++03, the language only allowed a subset of characters to be represented by their universal character names, and allowed some universal character names that didn't actually represent any valid Unicode characters. This was fixed in the C++11 standard. In C++11, both character and string literals and identifiers can use universal character names. For more information on universal character names, see Character Sets. For more information about Unicode, see Unicode. For more information about surrogate pairs, see Surrogate Pairs and Supplementary Characters.

String literals

A string literal represents a sequence of characters that together form a null-terminated string. The characters must be enclosed between double quotation marks. There are the following kinds of string literals:

Narrow String Literals

A narrow string literal is a non-prefixed, double-quote delimited, null-terminated array of type `const char[n]`, where n is the length of the array in bytes. A narrow string literal may contain any graphic character except the double quotation mark ("), backslash (\), or newline character. A narrow string literal may also contain the escape sequences listed above, and universal character names that fit in a byte.

C++

```
const char *narrow = "abcd";

// represents the string: yes\no
```

```
const char *escaped = "yes\\no";
```

UTF-8 encoded strings

A UTF-8 encoded string is a u8-prefixed, double-quote delimited, null-terminated array of type `const char[n]`, where n is the length of the encoded array in bytes. A u8-prefixed string literal may contain any graphic character except the double quotation mark (″), backslash (\), or newline character. A u8-prefixed string literal may also contain the escape sequences listed above, and any universal character name.

C++

```
const char* str1 = u8"Hello World";
const char* str2 = u8"\U0001F607 is O:-)";
```

Wide String Literals

A wide string literal is a null-terminated array of constant `wchar_t` that is prefixed by 'L' and contains any graphic character except the double quotation mark ("), backslash (\), or newline character. A wide string literal may contain the escape sequences listed above and any universal character name.

C++

```
const wchar_t* wide = L"zyxw";
const wchar_t* newline = L"hello\ngoodbye";
```

char16_t and char32_t (C++11)

C++11 introduces the portable `char16_t` (16-bit Unicode) and `char32_t` (32-bit Unicode) character types:

C++

```
auto s3 = u"hello"; // const char16_t*
auto s4 = U"hello"; // const char32_t*
```

Raw String Literals (C++11)

A raw string literal is a null-terminated array—of any character type—that contains any graphic character, including the double quotation mark ("), backslash (\), or newline character. Raw string literals are often used in regular expressions that use character classes, and in HTML strings and XML strings. For examples, see the following article: Bjarne Stroustrup's FAQ on C++11.

C++

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
const char*       raw_utf8  = u8R"(An unescaped \ character)";
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character)";
```

A delimiter is a user-defined sequence of up to 16 characters that immediately precedes the opening parenthesis of a raw string literal and immediately follows its closing parenthesis. For example, in `R"abc(Hello"\()abc"` the delimiter sequence is `abc` and the string content is `Hello"\(`. You can use a delimiter to disambiguate raw strings that contain both double quotation marks and parentheses. This causes a compiler error:

C++

```
// meant to represent the string: )”
const char* bad_parens = R"()")";  // error C2059
```

But a delimiter resolves it:

C++

```
const char* good_parens = R"xyz()")xyz";
```

You can construct a raw string literal in which there is a newline (not the escaped character) in the source:

C++

```
// represents the string: hello
//goodbye
const wchar_t* newline = LR"(hello
goodbye)";
```

std::string Literals (C++14)

std::string literals are Standard Library implementations of user-defined literals (see below) that are represented as "xyx"s (with a `s` suffix). This kind of string literal produces a temporary object of type std::string, std::wstring, std::u32string or std::u16string depending on the prefix that is specified. When no prefix is used, as above, a std::string is produced. L"xyz"s produces a std::wstring. u"xyz"s produces a std::u16string, and U"xyz"s produces a std::u32string.

C++

```
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };
wstring str3{ L"hello"s };
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

The s suffix may also be used on raw string literals:

C++

```
u32string str6{ UR"(She said "hello.")"s };
```

std::string literals are defined in the namespace `std::literals::string_literals` in the <string> header file. Because `std::literals::string_literals`, and `std::literals` are both declared as inline namespaces,

`std::literals::string_literals` is automatically treated as if it belonged directly in namespace `std`.

Size of String Literals

For ANSI char* strings and other single-byte encodings (not UTF-8), the size (in bytes) of a string literal is the number of characters plus 1 for the terminating null character. For all other string types, the size is not strictly related to the number of characters. UTF-8 uses up to four char elements to encode some *code units*, and char16_t or wchar_t encoded as UTF-16 may use two elements (for a total of four bytes) to encode a single *code unit*. This example shows the size of a wide string literal in bytes:

C++

```
const wchar_t* str = L"Hello!";
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

Notice that `strlen()` and `wcslen()` do not include the size of the terminating null character, whose size is equal to the element size of the string type: one byte on a char* string, two bytes on wchar_t* or char16_t* strings, and four bytes on char32_t* strings.

The maximum length of a string literal is 65535 bytes. This limit applies to both narrow string literals and wide string literals.

Modifying String Literals

Because string literals (not including std:string literals) are constants, trying to modify them—for example, str[2] = 'A'—causes a compiler error.

Microsoft Specific

In Visual C++ you can use a string literal to initialize a pointer to non-const `char` or `wchar_t`. This is allowed in C99 code, but is deprecated in C++98 and removed in C++11. An attempt to modify the string causes an access violation, as in this example:

C++

```
wchar_t* str = L"hello";
str[2] = L'a'; // run-time error: access violation
```

You can cause the compiler to emit an error when a string literal is converted to a non_const character pointer when you set the /Zc:strictStrings (Disable string literal type conversion) compiler option. We recommend it for standards-compliant portable code. It is also a good practice to use the `auto` keyword to declare string literal-initialized pointers, because it resolves to the correct (const) type. For example, this code example catches an attempt to write to a string literal at compile time:

C++

```
auto str = L"hello";
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

In some cases, identical string literals may be pooled to save space in the executable file. In string-literal pooling, the compiler causes all references to a particular string literal to point to the same location in memory, instead of having each reference point to a separate instance of the string literal. To enable string pooling, use the /GF compiler option.

End Microsoft Specific

Concatenating adjacent string literals

Adjacent wide or narrow string literals are concatenated. This declaration:

C++

```
char str[] = "12" "34";
```

is identical to this declaration:

C++

```
char atr[] = "1234";
```

and to this declaration:

C++

```
char atr[] =  "12\
34";
```

Using embedded hexadecimal escape codes to specify string literals can cause unexpected results. The following example seeks to create a string literal that contains the ASCII 5 character, followed by the characters f, i, v, and e:

C++

```
"\x05five"
```

The actual result is a hexadecimal 5F, which is the ASCII code for an underscore, followed by the characters i, v, and e. To get the correct result, you can use one of these:

C++

```
"\005five"     // Use octal literal.
"\x05" "five"  // Use string splicing.
```

std::string literals, because they are std::string types, can be concatenated with the + operator that is defined for basic_string types. They can also be concatenated in the same way as adjacent string literals. In both cases, the string encoding and the suffix must match:

C++

```
auto x1 = "hello" " " " world"; // OK
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix
auto x3 = u8"hello" " "s u8"world"s; // OK, agree on prefixes and suffixes
auto x4 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

String literals with universal character names

Native (non-raw) string literals may use universal character names to represent any character, as long as the universal character name can be encoded as one or more characters in the string type. For example, a universal character name representing an extended character cannot be encoded in a narrow string using the ANSI code page, but it can be encoded in narrow strings in some multi-byte code pages, or in UTF-8 strings, or in a wide string. In C++11, Unicode support is extended by the char16_t* and char32_t* string types:

C++

```
// ASCII smiling face
const char*     s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*  s2 = L"😉 = \U0001F609 is ;-)";

// UTF-8  encoded SMILING FACE WITH HALO (U+1F607)
const char*     s3 = u8"😇 = \U0001F607 is O:-)";

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t* s4 = u"😃 = \U0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t* s5 = U"😎 = \U0001F60E is B-)";
```

See Also

Character Sets
Numeric, Boolean and Pointer Literals
User-Defined Literals