charN_t incremental adoption: Casting pointers of UTF character types

Document #:	P2626R0
Date:	2022-08-09
Programming Language C++	
Audience:	SG-16, LEWG, EWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose a set of functions to cast to and from pointers of char8_t, char16_t, char32_t.

Tony table

Before

```
const wchar_t* str = L"Hello 🌎 ";

// The innocent: ill-formed.
const char16_t*  u = static_cast<const char16_t*>(str);

// The 10x developer: UB.
const char16_t*  u = reinterpret_cast<const char16_t*>(str);

// The C approach: UB in C++
const char16_t*  u = (const char16_t*)(str);

// The ranger: O(n), not contiguous
auto v = std::wstring_view(str) | std::views::transform(std::bit_cast<char16_t, wchar_t>);

// Download more RAM: O(n), allocates
auto v = std::wstring_view(str)
    | std::views::transform(std::bit_cast<char16_t, char16_t>) | std::to<std::vector>;

// The abstract Matrix: still UB
const char16_t* u =  std::launder(reinterpret_cast<const char16_t*>(str));

// The expert: not constexpr
const char16_t* u =  std::start_lifetime_as_array(reinterpret_cast<const char16_t*>(str),
                                                   std::char_traits<wchar_t>::length(str));
```

After

```
// constexpr
// no-op
// Explicit about the semantics
// Not UB
constexpr std::u16_string_view v = std::cast_as_utf_unchecked(L"Hello"sv);
```

Motivation

`char8_t`, `char16_t`, `char32_t` are useful to denotes UTF code units, and offer distinct types from `char`, `wchar_t`.

Indeed, existing practices use `char` to represent bytes, code units in the narrow encoding, code units in an arbitrary encoding, or UTF-8 code units.

It is therefore important that we had types to signal UTF code units and code units sequences such that:

- library authors can signal that an interface expects UTF
- library users can have some assurance that their UTF data will be properly interpreted and preserved.

This is necessary in C++ because the narrow and wide encodings are not guaranteed to be UTF-8 - and often are not -, and character types are not distinct from bytes or integer types.

We can lament that there should be a single character type and that it should be UTF-8. Unfortunately, there are too much code and systems for which this wishful assumption doesn't hold.

But of course, as `char16_t` and `char32_t` were adopted in C++11 and `char8_t` in C++20 - 15 to 25 years after the UTF encodings were standardized-, there exist a large body of code and projects that represent UTF-data by other means: Either using `char` and `wchar_t`, or an unsigned integer type.

There exist no way to pass a `charN_t` code units sequences to such pre-existing interface without either making a copy or triggering undefined behavior. This impossibility to pass or to extract `charN_t*` to/from these legacy interfaces is one of the issues causing the seemingly poor adoption of these types, and the persistence of the problems they aimed to solve - and which very much still exist.

Design

We propose 2 sets of functions:

- `std::cast_as_utf_unchecked`: cast a pointer of `std::byte`, `char`, `wchar_t` or unsigned integer type of size `N` to a pointer of `charN_t`.
- `std::cast_utf_to<T>`: cast a pointer of `charN_t` to a pointer of `T` (which can be `std::byte`, `char`, `wchar_t` or an unsigned integer type of size `N`).

These functions have semantics somewhat similar to `std::start_lifetime_as_array` in that they end the lifetime of the source objects, and start the lifetime of new objects with the same value but a different type. They are also, unlike `std::start_lifetime_as_array`, `constexpr`. As such, they do require some compiler support in the form of a magic built-in.

Valid casts

	char8_t	char16_t	char32_t
char	x		
unsigned char	x		
uint_least16_t		x	
uint_least32_t			x
wchar_t	!	!	!
std::byte	x		

! : implementation-defined

Note that `charN_t` is defined to have the size of `uint_leastN_t` on all platforms (which will be N bits almost everywhere, but theorically there could be some extra padding bits, which don't affect anything).

`span` and `string_view` overloads

For convenience, we propose `span` and `basic_string_view` overloads of both `std::cast_as_utf_unchecked` and `std::cast_utf_to`. These functions take their parameter by rvalue reference, as the objects denoted by their range are ended. Forcing a move adds *some* safety or at least some signaling of what is happening. The goal is to make the interface as usable and ergonomic as possible.

Semamtic cast and Unicode sandwich

`std::cast_as_utf_unchecked` isn't just casting the type of the string. It is also meant to reflect that the string value will now be semantically a UTF-8 sequence. The type change, the representation does not, but the value and its domain do. The name and the precondition - that the sequence must indeed represent UTF data - reflect that domain change.

Naming

The names were picked to

- Make it clear that it's a cast
- Be the same for all utf character types, for the sake of genericity.
- Make it clear that passing UTF data to a function that do not expect may not be a safe operation.

- Leaving the room for a different function which does check for the validity of the UTF sequence and return an `std::expected` for example. This is not explored in this paper, as to not getting ahead and conflicting with the work done by `P1629R1`.

Headers

The basic pointer interface is tentatively in `<utility>`. This is not great, not terrible. There isn't a more suitable header - maybe `cuchar`, but there isn't anything C++ specific in there yet.

Why only supporting ranges of code units and not individual code units?

`std::bit_cast` is perfectly suitable for that purpose.

What about C?

in C, `charN_t` are aliases for the corresponding unsigned integer type, so the concern does not apply.

Future work

- We should consider a validating variant of `std::cast_as_utf_unchecked` (`std::cast_as_utf`) once the standard library has a better grasp on how to model encoding errors.
- `charN_t` types are poorly supported in the standard library. We should notably support them in format.

Alternative (not) considered

Other solutions have been proposed to this problem:

- Relaxing aliasing rules
- Introducing special overloading rules
- Introducing magic in `static_cast`
- Deprecating/Removing `char8_t` from the standard.

However, these solutions do not address how their application would solve the problems that led to the introduction of these character types in the first place (see P0482R6 [2] for the original motivation), or would introduce poorly understood complexities, while hiding an operation, which, by its dangerouns nature, should be explicit.

Implementation

I prototyped a crude implementation in clang, but in the absence of existing support for `std::start_lifetime_as_array`, the implementation is probably not entirely correct. It does however illustrate the feature and the constexpr support, which is the one novelty compared to `std::start_lifetime_as_array`. A prototype can be found on Compiler Explorer, demonstrating constexpr usage and usage with an existing interface - In this case `iconv`.

Existing practices

- Many Win32 functions deal in `wchar_t*` pointers, but the encoding is UTF-16, so it should be possible to exchange `char16_t*` with these interfaces, but isn't.
- `iconv` can convert to and from UTF-8 but its interface expects `char` pointers.
- `QString::fromUtf8` expects `char` pointers.
- ICU can use `char16_t` or `uint16_t`. ICU offers cast functions that are one of the inspirations for this paper [Documentation].
- Any resemblance with Rust's `std::str::from_utf8_unchecked` is not entirely coincidental.

Wording

� Fundamental types [basic.fundamental]

Type `char` is a distinct type that has an implementation-defined choice of "`signed char`" or "`unsigned char`" as its underlying type. The three types `char`, `signed char`, and `unsigned char` are collectively called *ordinary character types*. The ordinary character types and `char8_t` are collectively called *narrow character types*. For narrow character types, each possible bit pattern of the object representation represents a distinct value. [*Note:* This requirement does not hold for other types. — *end note*] [*Note:* A bit-field of narrow character type whose width is larger than the width of that type has padding bits; see **??**. — *end note*]

Type `wchar_t` is a distinct type that has an implementation-defined signed or unsigned integer type as its underlying type.

Type `char8_t` denotes a distinct type whose underlying type is `unsigned char`. Types `char16_t` and `char32_t` denote distinct types whose underlying types are `uint_least16_t` and `uint_least32_t`, respectively, in <cstdint>.

The types `char8_t`, `char16_t` and `char32_t` are collectively called *utf character types*.

Type `bool` is a distinct type that has the same object representation, value representation, and alignment requirements as an implementation-defined unsigned integer type. The values of type `bool` are `true` and `false`.

� Utility components [utility]

� Header <utility> synopsis [utility.syn]

The header utility contains some basic function and class templates that are used throughout the rest of the library.

```
namespace std {
    // [..]

     // [utility.underlying], to_underlying
   template<class T>
   constexpr underlying_type_t<T> to_underlying(T value) noexcept;

    // [utility.utf.cast]
    template <class From>
    constexpr see below cast_as_utf_unchecked(From* ptr, size_t n) noexcept;
    template <class To, class From>
    constexpr see below cast_utf_to(From* ptr, size_t n) noexcept;

    // [utility.unreachable], unreachable
    [[noreturn]] void unreachable();
    // [...]
}
```

� UTF cast utilities [utility.utf.cast]

Let *COPY_CV(From, To)* denote the type To with the same cv-qualifiers as From.

For a type T, if there exist a *utf character type* ([basic.fundamental]) U that has the same size and alignement as T, then *UTF_TYPE(T)* denotes U.

Otherwise *UTF_TYPE(T)* does not denote a type.

```
template <class From>
constexpr COPY_CV(From, UTF_TYPE(From))* cast_as_utf_unchecked(From* ptr, size_t n) noexcept;
```

Constraints:

- remove_cv_t<From> denotes an integral type or byte.
- remove_cv_t<From> does not denote an *utf character type*.
- *UTF_TYPE(From)* denotes a type.

Preconditions:

[ptr, ptr + n) is valid range, and a valid code unit sequence in the UTF encoding associated with *UTF_TYPE(From)*.

Effects: The lifetime of each object O in the range [ptr, ptr + n) is ended and an object of type *COPY_CV(From, UTF_TYPE(From))* with the same object representation as O is implicitely created at the address of O.

Returns: A pointer to the first object in the range `[ptr, ptr + n)`

```
template <class To, class From>
constexpr COPY_CV(From, To)* cast_utf_to(From* ptr, size_t n) noexcept;
```

Constraints:

- `remove_cv_t<From>` denotes a *utf character type*.
- `To` denotes an integral type or `byte`.
- `To` does not denote an *utf character type*.
- `From` and `To` have the same size and alignment.
- `same_as<To, remove_cv_t<To>>` is `true`.

Preconditions: `[ptr, ptr + n)` is valid range.

Effects: The lifetime of each object `O` in the range `[ptr, ptr + n)` is ended and an object of type *`COPY_CV(From, To)`* with the same object representation as `O` is implicitely created at the address of `O`.

Returns: A pointer of type to the first object in the range `[ptr, ptr + n)`.

� String view classes [string.view]

� Header `<string_view>` synopsis [string.view.synop]

```
#include <compare>              // see ??

namespace std {
    // ??, class template basic_string_view
    template<class charT, class traits = char_traits<charT>>
    class basic_string_view;

    inline namespace literals {
        inline namespace string_view_literals {
            // ??, suffix for basic_string_view literals
            constexpr string_view    operator""sv(const char* str, size_t len) noexcept;
            constexpr u8string_view  operator""sv(const char8_t* str, size_t len) noexcept;
            constexpr u16string_view operator""sv(const char16_t* str, size_t len) noexcept;
            constexpr u32string_view operator""sv(const char32_t* str, size_t len) noexcept;
            constexpr wstring_view   operator""sv(const wchar_t* str, size_t len) noexcept;
        }
    }

    template <class T>
    constexpr see below cast_as_utf_unchecked(basic_string_view<T> && v) noexcept;
    template <class To, class From>
    constexpr auto cast_utf_to(basic_string_view<From> && v) noexcept;
```

```
}
```

� UTF cast functions [string_view.utf.cast]

```
template <class T>
constexpr auto cast_as_utf_unchecked(basic_string_view<T> && v) noexcept;
```

Constraints:

- `remove_cv_t<From>` denotes an integral type or `byte`.
- `remove_cv_t<From>` does not denote an *utf character type*.
- `UTF_TYPE(From)` denotes a type ([utility.utf.cast]).

Returns: `basic_string_view{cast_as_utf_unchecked(v.data(), v.size()), v.size()}`.

```
template <class To, class From>
constexpr auto cast_utf_to(basic_string_view<From> && v) noexcept;
```

Constraints:

- `remove_cv_t<From>` denotes a *utf character type*.
- `same_as<To, char> || same_as<To, wchar_t>` is `true`.
- `From` and `To` have the same size and alignment.

Returns: `basic_string_view{cast_utf_to<To>(v.data(), v.size()), v.size()}`.

� Views [views]

� General [views.general]

The header <span> defines the view `span`.

� Header `<span>` synopsis [span.syn]

```
namespace std {
    // constants
    inline constexpr size_t dynamic_extent = numeric_limits<size_t>::max();

    // ??, class template span
    template<class ElementType, size_t Extent = dynamic_extent>
    class span;

    template<class ElementType, size_t Extent>
    inline constexpr bool ranges::enable_view<span<ElementType, Extent>> = true;
    template<class ElementType, size_t Extent>
    inline constexpr bool ranges::enable_borrowed_range<span<ElementType, Extent>> = true;
```

```
// ??, views of object representation
template<class ElementType, size_t Extent>
span<const byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_bytes(span<ElementType, Extent> s) noexcept;

template<class ElementType, size_t Extent>
span<byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_writable_bytes(span<ElementType, Extent> s) noexcept;

template <class T>
constexpr see below cast_as_utf_unchecked(span<T> && v) noexcept;
template <class To, class From>
constexpr auto cast_utf_to(span<From> && v) noexcept;

}
```

� Views of object representation [span.objectrep]

```
template<class ElementType, size_t Extent>
span<const byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_bytes(span<ElementType, Extent> s) noexcept;
```

Effects: Equivalent to: `return R{reinterpret_cast<const byte*>(s.data()), s.size_bytes()};` where `R` is the return type.

```
template<class ElementType, size_t Extent>
span<byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent>
as_writable_bytes(span<ElementType, Extent> s) noexcept;
```

Constraints: `is_const_v<ElementType>` is `false`.

Effects: Equivalent to: `return R{reinterpret_cast<byte*>(s.data()), s.size_bytes()};` where `R` is the return type.

```
template <class T>
constexpr auto cast_as_utf_unchecked(span<T> && v) noexcept;
```

Constraints:

- `remove_cv_t<From>` denotes an integral type or `byte`.
- `remove_cv_t<From>` does not denote an *utf character type*.
- *`UTF_TYPE(From)`* denotes a type ([utility.utf.cast]).

Returns: `span{cast_as_utf_unchecked(v.data(), v.size()), v.size()}`.

```
template <class To, class From>
constexpr auto cast_utf_to(span<From> && v) noexcept;
```

Constraints:

- `remove_cv_t<From>` denotes a *utf character type*.

- `To` denotes an integral type or `byte`.
- `To` does not denote an *utf character type*.
- `From` and `To` have the same size and alignment.
- `same_as<To, remove_cv_t<To>>` is `true`.

Returns: `span{cast_utf_to<To>(v.data(), v.size()), v.size()}`.

Feature test macros

[Editor's note: Add a new macro in `<version>`, `<utility>`, `<span>`, and `<string_view>` : `__cpp_lib_utf_cast` set to the date of adoption] .

Acknowledgments

Matt Godbolt for hosting an implementation on Compiler Explorer.

Tom Honermann, Hubert Tong, Richard Smith, David Goldblatt for helping me have a better grasp on TBAA.

Timur Doumler for his work on `start_lifetime_as` (P2590R2 [1]).

The many people who voiced their desire for better usability of the `charN_t` types.

References

[1] Timur Doumler and Richard Smith. P2590R2: Explicit lifetime management. https://wg21.link/p2590r2, 7 2022.

[2] Tom Honermann. P0482R6: char8_t: A type for utf-8 characters and strings (revision 6). https://wg21.link/p0482r6, 11 2018.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4892