Unicode Identifiers And Reflection

D1953R0
Reply to: corentin.jabot@gmail.com
Audience: SG-7, SG-15

Abstract

SG-16 members are looking at extending the basic character set to support Unicode Identifiers.
SG-7 is designing tools to convert identifiers to string (as well as the reverse).

Therefore it will be necessary to be able to reflect (and reifere) on identifiers containing characters outside of the basic character sets.

We explore solutions

Unicode Identifiers

Extending the basic character set is an area of ongoing research, but the general direction is:

- Based on TR31
- Specified Normalization of identifiers at compile time (more likely NFC) - to ensure consistent behavior (and mangling) across translation units and implementations.
- Limited to (assumed) UTF-encoded files, because no one wants mojibake in their identifiers

The general motivation is not to encourage Unicode characters in identifiers but to ensure a consistent, reliable behavior across platforms.

However, the goal of that paper is not specified how Unicode identifiers should work but rather to open a discussion as to how they should be reflected upon.

C++ Text Model Primer

For people not familiar with the work of SG-16, here is briefly how C++ handle text

- Each token is converted from the “source character encoding” (which is determined by the compiler in an implementation-defined way - GCC and Clang assumes UTF-8 by default while MSVC uses UTF BOMs and user locale to determine the “source

character encoding" - Both GCC and MSVC provides flags to let their user override that behavior)
- To the internal character encoding, which is not specified but implied to be a Unicode encoding
- String literals are further converted to the _execution encoding_ whose character set is a subset of the internal character set.
- Both the source character set and the execution character set are supersets of the ***basic source character set*** which is the following abstract characters

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

We can make a few observations:

- Converting from the internal encoding to the execution encoding is lossy: the execution encoding might not be able to represent all the abstract characters the internal encoding can.
- Converting from execution to UTF-8 is lossless
- Converting from the source character set to the internal character-set is lossless. (C++ does not support abstract characters that are not part of the Unicode repertoire.

A Reflection primer

For people not familiar with the work of SG-7, here are a few facilities they are discussing

- `std::string name_of(meta::info)`
- `std::string display_name_of(meta::info)`
- Reification (conversion of a string to a c++ identifier): `[: "my_var" :] = 42;`

The problem

In the general case, the following code cannot work

```
std::string resumé;
auto mf = reflexpr(resumé);
consteval std::string name = meta::name_of(mf); //??? - 1
[: name :] = 42; // ?? - 1
```

1. If the execution encoding cannot represent `é`, `name_of` cannot return a meaningful value
2. If the program was not ill-formed in 1, it is now

If we allow Unicode identifiers, reflection and reification must support Unicode

Possible Solutions

- **`name_of` and `display_name_of` should return `u8string`** (or another `char8_t` based type)
- We should provide a method to compare identifiers to other character types and other identifiers, that would normalize

```
consteval compare_identifier(u8string_view, u8string_view);
consteval compare_identifier(u8string_view, string_view);
consteval compare_identifier(string_view, u8string_view);
```

Although a better alternative would be to rely on upcoming Unicode utilities to not duplicate runtime and compile time interface.

- Because some people may want to query at runtime the name of identifiers - when runtime reflection is built on top of dynamic reflection, for example, we should p**rovide a set of functions to convert u8 strings to ordinary and wide literals**

```
consteval const char* to_narrow_literal(u8string_view);
consteval const wchar_t* to_wide_literal(u8string_view);
```

This is done at compile-time and can be efficiently implemented as intrinsic (note that an overwhelming majority of identifier will continue to use the ASCII subset of Unicode).

At compile time. **lossy transcoding to narrow or wide execution encoding should be ill-formed by default.**

Note that many applications (for example I suspect that would be the case of the Qt framework) will want to store UTF-8 data at runtime rather than narrow encoding.

- Because narrow/wide -> internal encoding conversion never loses information, it makes sense to **allow reifying identifiers from any string literal types**, by transcoding to the internal encoding and NFC normalizing.

These set of solution have the advantages to:1

- Not duplicate the reflection API for different string types
- Work consistently for all identifiers and all platforms
- Leverage the fact that identifiers already need to encoded in a Unicode encoding
- Let applications to choose which encoding the strings obtained from the reflection API should used to build dynamic reflection facilities

References

P1240R1: [SG7] Scalable Reflection in C++ (by Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, Faisal Vali, Daveed Vandevoorde)

https://unicode.org/reports/tr31/ UNICODE IDENTIFIER AND PATTERN SYNTAX