P3655R5: cstring_view

From Croydon

Principles behind P3655

1) It is a non-allocating type that provides a type-system propagating annotation of having a null terminator at the end.

2) Type carries the at-creation checked existence of a null terminator at size() until its use. If it cannot guarantee it for some operation, it should fail to compile at the step where the guarantee is potentially lost.

3) Type works identical to `string_view` and `string` where possible, similar where feasible and fail to compile if neither of those can be done.

4) Principle of least surprise: Type fits between string and string_view. Calling functions that preserve null terminator return cstring_view, functions that do not return string_view. Converting from `string` to `string_view` gives the same result as conversion from `string` to `cstring_view` and on to `string_view`.

5) Type works in the way clients of the type will expect, in that size() and strlen(c_str()) are guaranteed to give the same result.

Principles behind P3655

1) It is a **non-allocating type** that provides a **type-system propagating annotation** of **having a null terminator at the end**.
2) Type carries the **at-creation checked** existence of a **null terminator at size()** until its use. If it cannot guarantee it for some operation, it should **fail to compile** at the step **where the guarantee is** potentially **lost**.
3) Type works identical to `string_view` and `string` where possible, similar where feasible and fail to compile if neither of those can be done.
4) Principle of least surprise: Type fits between string and string_view. Calling functions that preserve null terminator return cstring_view, functions that do not return string_view. Converting from `string` to `string_view` gives the same result as conversion from `string` to `cstring_view` and on to `string_view`.
5) Type works in the way clients of the type will expect, in that size() and strlen(c_str()) are guaranteed to give the same result.

- "cstring_view is a non-owning, size-aware C++ string type that preserves the null-termination invariant -- so null-terminated strings can be faithfully represented, safely processed, and passed to C APIs without copies."

1) Avoid allocation
2) Call C APIs
3) Representational fidelity
4) Type safe, rich, C++ API
5) Efficient

From Croydon

Embedded nulls

- cstring_view is a null *terminated* type.
- Embedded nulls have been used both for evil and for good:
  - “10.0.0.1\0.microsoft.com” to circumvent checks
  - “abc\0def\0ghi\0\0” to pass an array of strings
- Logically we know something is wrong when we get a call to c_str() with an embedded null

Embedded nulls

- No check
- Hardened precondition on constructor
- Precondition on constructor
- Hardened precondition on c_str()
- Precondition on c_str()
- Truncate in constructor

Peter Dimov, LEWG reflector

This just moves the same problem elsewhere.

```
void f( std::cstring_view );

void g( std::string const& fn )
{
    if( fn.ends_with( ".txt" ) )
    {
        f( fn );
    }
}
```

	No check	Hardened Precondition constructor	Precondition constructor	Hardened precondition c_str()	Precondition c_str()	Truncate in constructor
Prevents NUL into C-func	❌	✅	✅	✅	✅	✅
No overhead	✅	❌	⚠️	❌	⚠️	❌
Works like string (with c_str())	✅	❌	❌	❌	❌	❌
Works like string (other than c_str())	✅	❌	❌	✅	✅	❌
Help user find cause	❌	✅	✅	⚠️	⚠️	❌
Gives safety guarantee	❌	✅	❌	✅	❌	❌
Does not add UB	✅	✅	❌	✅	❌	✅
No silent value changes	✅	✅	✅	✅	✅	❌

Suggested resolution

1) No check

   Matches existing types, no performance impact

2) Hardened precondition on constructor, with escape hatch

   Catches error on first chance, provides safety benefits, does not add UB

3) Regular precondition on constructor

   Adds UB, no safety benefit, does not change meaning

4) Any other

   Adds UB, does not give safety benefit, catches late, changes meaning, ...