Doc. no. P0442R0
Date: 10/16/2016
Project: Programming Languages C++
Audience: Core Language Work Group, Evolution Working Group
Reply-to: Farideddin Mehrabi <farid.mehrabi@gmail.com>

ABSTRACT

Bult-in arrays of C++ are among various legacy C features that give C++ the power to manage low-level system resource management. In case of arrays the resource is memory. C-style array has special properties - such as decaying to pointers, call-by-refrence semantics etc - which make them subtle and difficult to use in most simple form. This proposal is intended to introduce a new distinct built-in array type with more robust semantics, specially regarding construction and literals.

Full-Featured Value-Semantic Arrays

REFERENCES

Some addressed issues include:

P0259 :

FIXED_STRING: A COMPILE-TIME STRING
by **Michael Price** & **Andrew Tomazos**

N4236:

A COMPILE-TIME STRING LIBRARY TEMPLATE WITH UDL OPERATOR TEMPLATES
by **Michael Price**

N4121:

COMPILE-TIME STRING: STD::STRING_LITERAL<N>
by **Andrew Tomazos**

P0373:

PROPOSAL OF FILE LITERALS
by **Andrew Tomazos**

Orthogonal papers include:

P0341:

PARAMETER PACKS OUTSIDE OF TEMPLATES
by **Mike Spertus**

CONTENTS

1. THE AGE-OLD WOUND

C-style arrays features:

- Are passed by reference in function calls rather than by values
- There are no array literal constants, 'std::initializer_list' is used instead
- Are not constexpr-able because nor is 'std::initializer_list'.
- Decay to pointers
- The highest order dimension of C-styke array can be implicit:

```
element_type array_obj [] [dim2] [dim1];
```

that is because the array can decay to pointers.

1.1 DECAYING VS TYPE-ERASURE

- **Type-Erasure** is a well-known method of converting static polymorphism to dynamic polymorphism. The mechanism is simple: capture the runtime features of underlying type in a (partialy) type-agnostic handle and keep the - otherwise lost – metadata using either the bult-in polymorphism features (the 'virtual' keyword), or some manual library hack(eg. Type index no. in 'std::variant').
- **Decaying** is some unavoidable implicit convertion to a (partialy) type-agnostic handle and permanently lose the important meta-data. Keeping track of the the - otherwise lost – metadata is burdened to user code.
  In case of array decay, the handle is the pointer, and metadata is the element-count or simply size of the array.

1.1.1 'Character Array' and 'Character String'

In C/C++ as well as many other programming languages, 'character string' has been considered and implemented as type-erasure on 'character array'(erase the number of characters from the type and keep with data). In contrast to many other languages C (and inherently C++) have **tried** to keep the implementation of string in the library rather than the core language (because of diversity and complexty of available implementations).

1.1.1.1 The Scar: String literals

Robust programming without literals – specifically without string literals – is impossible. But since there is no built-in string type in C/C++ and array syntax and semantics are too limiting, c-style string literals have become an ever-lasting singularity in the core language; **C-style 'string literal' is implemented via type-erasure on character array** (yes, null-termination is an implicit way of keeping metadata about size of the string) in a language that does not define character string in its core. Interestingly, std::string does not follow trend of C in its implementation; it stores size and data explicitly in separate encapsulated members – rather than the traditional null-terminated sequence of characters. The new C++-style array and string literal syntax discussed in the present paper are meant to bypass such complexities by providing the programmer with maximum metadata available at compile-time, because dynamic polymorphism is an always present approach that is easily implemented troughout type-erasure (**size-erasure).**

1.1.1.1.1 C++ style character array literal

Since C/C++ doesn't intend to supply 'string' as a built-in class, in the era of template meta programming, we can introduce 'C++ style character array literal' as syntax suger to simplify initialization of C++-style character array which is not supposed to decay to pointers.

'C++ style character array literal' - In contrast to C-style string literals - won't encourage any specific implementation of size-erasure on character arrays.

1.1.2 Array Literal(Initializer)

Since - back in good old 'C' days -array Initializers did not seem as crucial as string literals, for relatively long period C++ lacke true Array Literals. Of course it has always been possible to initialize a named array upon its declaration/definition, but what about an unnamed array?

```
auto &  char_6 = "12345" ; // OK: char (& char_6) [6];
auto &  int_6 = {1,2,3,4,5,6} ; // ERROR: NOT int (& int_6) [6];
```

1.1.2.1 Latest Surgery: 'std::initializer_list'

The problem with array literals punched into the face by the imergence and flourishment of generic container types who needed some sort of unnamed array literal for initialization. The solution came out to be a magical size-erased array which was considered part of the library - while not providing any implementation regarding its own initialization!!!

1.2 CONSEXPR ARRAY/STRING

There have been demands in template/meta programming to have array/string constexpr values as none-type template parameters. But array/string literals(initializer_list/null-terminated strings) are inherently not constexpr-able for known solid reasons.

1.3 BY-REFERENCE SEMANTICS

In contrast to all other intrinsic and most library types, C-style arrays –unless wrapped in some other UDT- are passed by reference in function calls. This used to be bothersome before the introduction of library class 'std::array' that also alleviated the issue of decaying to pointers and index overrun.

2. PROPOSED SEMANTICS

A 'C++-style' array is:

- a complete value type
- copy-able from same type
- movable
- Assignable from same type
- Constexpr-able
- Does not decay to pointers.
- cv-qualification of the array is same as that of its elements (just like C-style arrays)
- Cannot be copied or assigned from an array of larger element count:
  Consider 'cpp_array<typename, size_t>' as an alise for a 'C++-style' array:

  ```
  cpp_array< source_type,source_count> source;
  cpp_array< target_type,target_count> target{source};
  ```

  the above snippet would compile if and only if :
  - 'source_type' is convertible to 'target_type' and
  - 'target_count' not larger than 'source_count' and
  - 'source_type{}' is implicit, well-define and accessible; so that excess elements are (quasi-)default constructed.

  Another option would be **exact match** between element types and/or counts.

Conjunction with C-style arrays:

- It can be implicitly copied to a c-style array.
- It can be explicitly "static_cast"ed to/from a reference of c-style array with complying dimensions and same basic element type:
- It can be safely "reinterpret_cast"ed to/from a reference of any array with same size, alignment and basic element type.
- It can be assigned and explicitly copied from a c-style array with complying dimensions and same basic element type.

3. PROPOSED SYNTAX

The only syntactic difference from C-style array is in object declaration and initialization. Indexing would be as before.

3.1 ARRAY OBJECT INSTANCE DECLARATION

Either:

```
element_type array_obj [{element_count}];
```

Or:

```
element_type array_obj [<element_count>];
```

Or:

```
element_type array_obj [ element_count ++ ];
```

Or:

```
element_type array_obj [ - element_count ];
```

The third syntax reflects C++-style of the array, while the fourth seems easier to implement -though it is more controversial, less readable/safe to permit negative size. But I prefer the first and second syntax. Any of the above syntaxes enjoys the advantage of easily chaining multiple dimensions of mixed C/C++ styles:

```
element_type array_obj [{dim1}] [dim2] [{dim1}];
```

3.2 'C++-STYLE' ARRAY CONSTEXPR INITIALIZER

3.2.1 Named array initialization syntax:

The most definite syntax is:

```
constexpr element_type array_obj [{N}] {value_1, …, value_N};
```

Type of '{value_1, …, value_N}' is proposed to be C++-style array of 'N' elements if and only if the values have a common type wich is the element type. So, what about the `'std::initializer_list'`? That is the subject of initializer_list section.

3.2.1.1 Array initialization syntax with implicit dimension

'C++-style' array does not decay to pointers; therefore in contrast to 'C-style 'arrays, the element count of 'C++-style' array can only be omitted if initializer immediately follows the declaration, so that the type is fully specified:

```
constexpr int array_int [{}] {1, 2, 3}; // OK: int array_int [{3}]
constexpr int array_int [{}]; // compile error: size unknown
```

3.2.2 Unnamed array initialization syntax

Considering lambda syntax, following proposition are made:

3.2.2.1 explicit-type array initializer:

```
auto & array_explicit      = element_type [{N}] {element_1, …, element_N};
```

3.2.2.2 Implicit-type array initializer:

Element type is deduced as the common type of all provided elements by means of narrowing:

```
auto & array_implicit_type = [{N}] {element_1, …, element_N};
```

size can be implicit in either of the above syntaxes:

```
auto & array_implicit_size = element_type [{}] {element_1, …, element_N};
auto & array_implicit      = [{}] {element_1, …, element_N};
```

It is assumed that simple curly braces ('{a,b,c…}') are reserved for built-in tuple (P0341) but with proper narrowing, arrays can be constructed from built-in tuples Character array initializer. In either case one can write:

```
auto list = {1, 2, 3};//OK: int list[{3}]
```

3.3 CHARACTER ARRAY INITIALIZER

Now that we are going to have constexprable arrays, we can declare a neat syntax substitution for P0259, N4236, N4121 ,et al. Single quoted **none-null terminated character literals** are proposed as constexpr C++ arrays:

```
constexpr char cppstr [{3}] =’123’;
```

The reason character literal and C-style string literal use different quoting is that double quotes implicitly pad one extra null byte to the end of array. But with C++-style arrays, no extra byte is padded and no syntax ambiguity is introduced.

Eventhogh I don’t see any problem in allowing single element array being implicitly constructed from an element, in order to be conservative, single element character array is explicitly initialized form single character:

```
constexpr char cppstr [{1}] ={’1’};
```

Concatenation and prefixes can be applied (just like C-style literals):

```
constexpr wchar cppstr [{6}] = w’123’ w’456’;
constexpr char cpp_r_str [{7}] = R’(reg_exp)’;//compare R”(reg_exp)”
```

4. CHALLENGES

4.1 INITIALIZER_LIST

Roughly speaking the 'std::initializer_list' was originally introduced as a magic solution for the lack of array literals. Now the magic is gone and it can be implemented as some type-erasure (remove the fixed element count) on C++-style array (possibly via the 'alloca' function to avoid heap allocation):

```
template<class Elem>
struct initializer_list   :
	std::pair<const Elem* const, const Elem*const>
{
	typedef std::pair<const Elem* const, const Elem*const>     pair;
	typedef const Elem value_type;
	typedef value_type& reference;
	typedef value_type& const_reference;
	typedef size_t size_type;
	typedef value_type* iterator,pointer_type;
	typedef value_type* const_iterator;
private:
	initializer_list(pointer_type ptr, size_type sz)
		: pair {ptr,ptr+sz}
	{}
public:
	initializer_list()
		: pair {nullptr,nullptr}
	{}
	template<size_t N>
	initializer_list(value_type(&arr)[{N}])
		: initializer_list {new(on_stack) value_type [N]{ arr },N}
	{}
	~initializer_list() {
		for (auto ptr = end();ptr >= begin();)
			(--ptr)->~Elem();
		dispose(ptr);
	}
	iterator     begin()const {return (first);}
	iterator     end()  const  {return (second);}
	size_type    size() const  {return ((size_type)(second - first));}
};
```

5. EFFECTS ON STD LIBRARY

- It is good for container types to declare array conversion constructors, although initializer_list constructor can compensate for them.
- Type traits, iterator and basic meta type libraries (eg. 'tuple_size', 'tuple_element') should be declared to comply with 'C-style' arrays.
- 'string' library also needs an update to erase the type (element count) of 'C++-style' character array and provide some extra convenience access and convertion methods.
- 'array_view' needs to cover 'C++-style' arrays too.

6. SUBSIDIARY PROPOSALS

6.1 KEYWORD 'RECORD_FILE'

There has been P0373 to introduce some new mechanism that can convert a file on the compiler host to an array of bytes. This would simplify design of resource compilers in GUI libraries; they usually use one of the approaches:

- **Pre-compile action:** Convert the required binary file to C++ source file (Qt, emWin…)
- **Post-build Binary link:** link the binary file along with some magic metadata (VS…)

By introducing a standard keyword most of the job gets done. Writing some simple compile time parsers may also become possible. The proposed approach in present document follows similar rationale but with different syntax.

The main semantics difference is that P0373 divides files into binary and text files which is a unix/POSIX based categorization adopted by some other OS families, but on some file systems - in order to optimize storage and access - there is the ability to treat files of specific fixed-size record types differently. Division to text/binary categories is not proposed here and binary format is the default, but if needed this syntax may be modified to support extra parameters to state compile-time parsing options.

Another difference is that P0373 does not discuss the underlying container type to keep the constexpr result. Even if P0373 is preferable over 'record_file', its syntax may need to be modified to use singly-qouted literals.

6.1.1 Proposed syntax

The record type is a POD type that is passed as an explicit type parameter to the keyword. The keyword is supposed to be function-like with the following pseudo signature:

```
template <typename record_type = char, size_t M >
constexpr  record_type (record_file (char (&filepath) [{M}]) ) [{N}];
```

Where 'N' is the size of file in terms of record type. The keyword accepts a filepath of arbitrary length, resolve it according to project config, and converts it to a constexpr array of 'record_type's. Types other than 'char' family of types shall be considered none-portable (due to endian-ness and platform specific issues). Example:

```
//add a wallpaper to the executable (kept as signed char[{X}]):
auto constexpr picture = record_file <signed char> ('.\resource\photo.jpg');
//embed a welcome sound track in the executable (kept as char[{X}]):
char constexpr voice[{}] = record_file ('.\resource\track.mp3');
```

6.2 SINGLE-QUOTED " OPERATOR'' "

Although in presence of constexpr character array literals, user defined literals for other types can be implemented as constexpr converters, syntax suger might seem appealing.

This operator can be the constexpr counterpart of 'operarator"" ' for string literal definition. To emphasize the constexpr nature of this operator, the value is fed in broken braces. One typical specialization of the operator may look like:

```
constexpr auto operarator ''< char value_string[{N}], suffix_string > () {…};
```

example (literal for std::bitset<8>):

```
constexpr std::bitset<8> operarator ''< char value [{8}], 'b8' > () {
        static_assert(! all_1_0 (value), 'bad bitset<8> literal' );
        return  to_b8(value);
};
std::bitset<8> bits= 10101010b8;
```

7. THE HORIZON

Existence of constexpr array and string types as candidates for containing compile-time metadata, might have a positive impact on future proposals regarding meta-data, reflection or typeinfo on the long run. I cannot reference th large amount of proposals for reflection which are waiting on some form of constexpr array/string. Adoption of P0442 may lubricate the wheels for all those efforts in addition to string proposals addressed earlier.