Traits for [Un]bounded Arrays

Document #: WG21 P1357R0
Date: 2019–01–06
Audience: LWG[1]
Reply to: Walter E. Brown <webrown.cpp@gmail.com>
Glen J. Fernandes <glenjofe@gmail.com>

Contents

1 Introduction and background . . 1
2 Discussion and proposal 2
3 Sample implementation 2
4 Proposed wording 3
5 Acknowledgments 4
6 Bibliography 4
7 Document history 4

Abstract

This paper proposes two new type traits that have been demonstrated, in prior art, to provide a useful partition of the types for which the **`is_array`** trait holds. The proposed partition is analogous to that provided, when **`is_integral`** holds, by the **`is_signed/is_unsigned`** traits.

When ignorance gets started it knows no bounds.
— William Penn Adair "Will" Rogers

The function of vice is to keep virtue within reasonable bounds.
— Samuel Butler

I am not bound to win, but I am bound to be true. I am not bound to succeed, but I am bound to live by the light that I have.
— Abraham Lincoln

1 Introduction and background

The type traits known as *primary type categories*[2] serve to partition the universe of C++ types into fourteen mutually exclusive categories such that each type will fall into exactly one category. In addition, we have *composite type traits*[3] to identify useful combinations of these primary type categories. Finally, we have current and past examples[4] of traits that provide even more finely-grained categorization.

[1] At its Saturday post-San Diego session, LEWG reviewed and tentatively approved (9|3|2|1|0) a pre-publication draft of this paper. No substantive changes have been applied since then.

[2] Specified in [meta.unary.cat]: **`is_void`**, **`is_null_pointer`**, **`is_integral`**, **`is_floating_point`**, **`is_array`**, **`is_pointer`**, **`is_lvalue_reference`**, **`is_rvalue_reference`**, **`is_member_object_pointer`**, **`is_member_function_pointer`**, **`is_enum`**, **`is_union`**, **`is_class`**, and **`is_function`**.

[3] Specified in [meta.unary.comp]: **`is_reference`**, **`is_arithmetic`**, **`is_fundamental`**, **`is_object`**, **`is_scalar`**, **`is_compound`**, and **`is_member_pointer`**.

[4] Specified in [meta.unary.prop]: **`is_signed`** and **`is_unsigned`**, providing a partition of types for which **`is_integral`** holds. Also, the composite trait **`is_reference`** was orignally a primary trait before the introduction, by [N2028], of rvalue references and the corresponding then-new **`is_rvalue_reference`** trait.

This paper proposes to add two new type traits of this last kind, for each of which there is significant prior art. The proposed traits will provide a useful partition of types for which **`is_array`** holds.

2 Discussion and proposal

It is often necessary to distinguish types that are arrays of unknown bound (i.e., where **`T`** is of the form **`U[]`**) from types that are arrays of known bound (i.e., where **`T`** is of the form **`U[N]`**) instead of just any array type. For example, such distinctions are needed in the implementation of certain C++ standard library facilities:

- **`std::unique_ptr`** and **`std::make_unique`**: these support use with arrays of unknown bound, but intentionally prohibit use with arrays of known bound.
- **`std::make_shared`** and **`std::allocate_shared`**: these provide different overloads for arrays of known bounds as well as for arrays of unknown bound.

Beyond use in implementation, it would also be useful to have these traits in the standard library for the specifications of other facilities in the C++ standard. Several constraints could then be specified by a C++ expression using traits, instead of by the equivalent (but less succinct) English prose.

We therefore propose to add two new traits named **`is_bounded_array`** and **`is_unbounded_array`**. Other possible name pairs for these traits include (a) **`is_bound_array`** and **`is_unbound_array`**, (b) **`is_known_bound_array`** and **`is_unknown_bound_array`**, or (c) **`is_known_extent_array`** and **`is_unknown_extent_array`**.

Using these traits, wording for the following constraints in the C++ standard could be consistently simplified:

Wording of this form	Could take this form instead
`T` is not an array	**`! is_array_v<T>`**
`T` is an array of known bound	**`is_bounded_array_v<T>`**
`T` is not an array of unknown bound	**`! is_unbounded_array_v<T>`**
`T` is an array of unknown bound	**`is_unbounded_array_v<T>`**

No such rewording is herein proposed; we simply provide the above possibilities by way of example and of motivation.

3 Sample implementation

Implementing the proposed traits is straightforward:

```
template< typename >
struct is_bounded_array : false_type  { };
//
template< typename U, size_t N >
struct is_bounded_array<U[N]> : true_type  { };

template< typename T >
inline constexpr bool is_bounded_array_v  = is_bounded_array<T>::value;
```

```
template< typename >
struct is_unbounded_array : false_type  { };
//
template< typename U >
struct is_unbounded_array<U[]> : true_type  { };

template< typename T >
inline constexpr bool is_unbounded_array_v  = is_unbounded_array<T>::value;
```

4 Proposed wording[5]

4.1 Insert the following row into Table 35 — Standard library feature-test macros. If needed, adjust the Value placeholder entry so as to denote this proposal's date of adoption.

Macro name	Value	Header(s)
⋮		
`__cpp_lib_bounded_array_traits`	201902L	`<type_traits>`
⋮		

4.2 Augment [meta.type.synop] as shown:

```
namespace std {
  ⋮
  template<class T> struct is_signed;
  template<class T> struct is_unsigned;
  template<class T> struct is_bounded_array;
  template<class T> struct is_unbounded_array;
  ⋮
  template<class T>
    inline constexpr bool is_signed_v = is_signed<T>::value;
  template<class T>
    inline constexpr bool is_unsigned_v = is_unsigned<T>::value;
  template<class T>
    inline constexpr bool is_bounded_array_v
      = is_bounded_array<T>::value;
  template<class T>
    inline constexpr bool is_unbounded_array_v
      = is_unbounded_array<T>::value;

  ⋮
}
```

[5] Proposed additions and ~~deletions~~ are based on [N4778]. Editorial notes appear like this.

4.3 Augment Table 46 — Type property predicates as shown:

Template	Condition	Preconditions
⋮		
`template<class T>` `struct is_signed;`	If `is_arithmetic_v<T>`...	
`template<class T>` `struct is_unsigned;`	If `is_arithmetic_v<T>`...	
`template<class T>` `struct is_bounded_array;`	`T` is an array type of known extent ([dcl.array])	
`template<class T>` `struct is_unbounded_array;`	`T` is an array type of unknown extent ([dcl.array])	
`template<class T, class... Args>` `struct is_constructible;`	For a function type `T`...	`T` and all types...
⋮		

5 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments.

6 Bibliography

[N2028] Howard E. Hinnant: "Minor Modifications to the type traits Wording." ISO/IEC JTC1/SC22/WG21 document N2028 (mid-Berlin/Portland mailing), 2006–06–12. https://wg21.link/n2028.

[N4778] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4778 (pre-San Diego mailing), 2018–10–08. https://wg21.link/n4778.

7 Document history

Rev.	Date	Changes
-1	2018–11–09	• Draft uploaded to San Diego LEWG wiki for post-San Diego review.
0	2019–01–06	• Added footnote and adjusted audience to reflect favorable post-San Diego review. • Published as P1357R0, pre-Kona mailing.