Type-Erasure

Praktisches Type-Erasure in C++

Überblick

- Type-Erasure in C++
- Nichtintrusives Type-Erasure und Wertesemantik
- Beispiele zu Type-Erasure
- Gegenüberstellung: statisch vs. dynamisch
- gotcha mit any
- std::function und Konstanz-Korrektheit
- Beispiel aus Produktion: moveonly_function
- Design-Vorgaben
- Gory Details der Implementierung
- TMP und Tricks mit SFINEA
- Beispiel der Type-Erasure ohne dynamischer Bindung

Type-Erasure in C++

Definition

Unter Type-Erasure versteht man eine Praxis die Typinformation zu reduzieren oder vollständig zu entfernen.

Ziel solch einer Reduktion ist es eine einheitliche Behandlung von Objekten mit verschiedenen Typen zu ermöglichen.

Dabei wird die Operation auf dem Objekt mit reduziertem statischem Typ durchgeführt, die Wirkung dieser Durchführung ergibt sich aus der Operation auf dem dynamischen Typ des Objekts.

Dieses Konzept wird im Kontext der objektorientierter Programmierung als Laufzeitpolymorphismus bezeichnet.

Type-Erasure in C++

Zurück zu den Wurzeln

In C kann Type-Erasure sehr einfach mit Hilfe der Zeigerkonvertierung zu `void*` realisiert werden:

```
type * concrete_ptr = …;
…
void * erased_ptr = concrete_ptr;
qsort(erased_ptr, count, size, &compare_types);
```

Leider ist die Schnittstelle `void*` nicht besonders handlich, ohne jeglicher Typsicherheit und daher anfällig für Fehler.

Type-Erasure in C++

Ein besseres Interface – Objektorientiertes Type-Erasure

Type-Erasure kann in C++ mit Hilfe der Vererbung von Basisklassen mit virtuellen Funktionen realisiert werden.

Mit solch einem deduzierten Sprachkonstrukt lässt sich eine robuste Schnittstelle aufbauen, soweit Idiome zur Klassengestaltung von polymorphen Typen eingehalten werden.

Das Problem der Speicherverwaltung durch den Zeiger auf das Basisobjekt lässt sich durch Verwendung der intelligenten Zeiger wie z.B. `std::unique_ptr` ohne Mehraufwand lösen.

Type-Erasure in C++

Ein besseres Interface – Objektorientierte Type-Erasure

```
1. struct  drawable
2. {
3.     virtual ~drawable() = default;
4.     virtual void  draw(context cx, point p) = 0;
5.     drawable(const drawable&) = delete;
6.     drawable& operator=(const drawable&) = delete;
7.     …
8. };
9. …
10. class  rectangle final : public drawable
11. {
12.    void  draw(context cx, point p) override { … }
13.    …
14. };
15. …
16. std::unique_ptr<drawable>  shape = std::make_unique<rectangle>();
17. shape->draw(draw_context, position);
```

Nichtintrusives Type-Erasure und Wertesemantik

Probleme mit Vererbung

Bei Verwendung des objektorientierten Laufzeitpolymorphismus, ist man auf Referenzsemantik angewiesen.

In C++ wird aber oft die Wertesemantik bevorzugt (siehe Algorithmen, Container), daher ist man oft auf Adapter angewiesen.

Für alle fundamentale und third-party Typen muss eine Wrapperklasse geschrieben werden, die die Basisklasse vererbt und die Funktionalität durch Überschreibung virtueller Funktionen bereitstellt – auch dann wenn die Ausgangstypen Wertesemantik haben.

All dies kann mit Hilfe der Templates automatisiert werden.

Nichtintrusives Type-Erasure und Wertesemantik

Nichtintrusives Type-Erasure baut auf dem Duck-Typing Prinzip der Templates auf – dem Benutzer wird Minimum an Arbeit verlangt:

```
1.    void  draw_item(int  drawable, point p);
2.    void  draw_item(third_party_type  drawable, point p);
3.…
4.#include "any_drawable.h"
5.…
6.    auto  drawable1 = any_drawable{23};
7.    auto  drawable2 = any_drawable{third_party_type{…}};
8.    …
9.    draw_item(drawable1, point{100, 200});
10.   draw_item(drawable2, point{200, 200});
```

```
11#include "any_incrementable.h"
12…
13.   auto  incrementable1 = any_incrementable{42};
14.   ++incrementable1;
```

Beispiele zu Type-Erasure

	Duck-Typing	**nützliches Interface**	**Wertesemantik**	**fehleranfällig zu implementieren**
void*	Ja	Nein	Nein	Nein
Objektorientierte Vererbung	Nein	Ja (Referenzsemantik)	Nein	Nein
boost::variant	Nein	Ja (Visitorbasiert)	Ja	Ja
std::any (TS) boost::any, …	Ja	Ja? (reguläres Interface)	Ja	Ja
std::exception_ptr	Ja	Ja	Nein	Ja
std::function, any_allocator (n3495) any_iterator, boost/type_erasure adobe/poly, …	Ja	Ja	Ja	Ja

Gegenüberstellung: statisch vs. dynamisch

Kompilierzeit Polymorphismus

keine Laufzeiteinbüße

nichtintrusives Duck-Typing

header-only

Robustheit (Concepts?)

Type-Erasure

aufteilbar in entkoppelte Komponenten

schnelle Übersetzungszeit

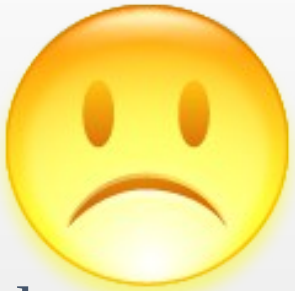
kleine Programmgröße

Anwendbar an binären Schnittstellen (?)

gotcha mit any

Bei Verwendung von `std::any`, oder einer anderer Klasse mit ähnlicher Funktionalität, können ownership Zyklen konstruiert werden:

```
1. auto  vec_any = std::vector<std::any> (1);
2. std::any & valid_ref = vec_any[0];
3. std::any  owner = std::move(vec_any);
4. valid_ref = std::move(owner);
```



Solche Zyklen sind bei unique-ownership nicht üblich.

gotcha mit any

Wer ist dran schuld?

Mit der move-Anweisung der 4. Zeile wird der owner mit dem gewrappten std::vector Objekt in das any Objekt (über valid_ref referenziert) verschoben, der sich selbst in dem std::vector Objekt aufhält.

Weil das std::vector Objekt vec_any in der 3. Zeile in den moved-from Zustand bereits versetzt wurde, ist nichts mehr für die Bereinigung des any Objektes und des gewrappten std::vectors verantwortlich.

In dem oben aufgeführten Programmabschnitt wird Speicher geleakt, obwohl jede Anweisung für sich harmlos erscheint.

Solche ownership Zyklen sollten vom Typsystem eines high-level Konstrukts abgefangen werden. Bei Type-Erasure wird aber das Typsystem bei Initialisierung gezielt abgeschaltet, um das Ausgangsobjekt zu wrappen.

Solche Fehlgriffe können allgemein nicht auf Bibliotheksebene abgefangen werden.

std::function und Konstanz-Korrektheit

Mit der Schablonenklasse `std::function` bietet die Standardbibliothek von Haus aus ein Werkzeug zur Type-Erasure mit Wertesemantik für aufrufbare Typen (Funktoren genannt).

Über den Typparameter wird die Aufrufsignatur übergeben.

`std::function` hat ein nützliches Interface, bietet Kleinobjekt-Optimierung und ist vor allem bei Lambda-Objekten nützlich, wenn diese zurückzugeben werden:

```
1.    #include<functional>
2.    …
3.    std::function<int()>  get_dec_generator(int  start)
4.    {
5.        int x = start - 10;
6.        return  [x]() mutable { return  x += 10; };
7.    }
```

std::function und Konstanz-Korrektheit

Ist dieses C++ Programm wohlgeformt?

```
1. #include    <iostream>
2. #include    <functional>
3.
4. int  main()
5. {
6.     int  x = 0;
7.     const std::function<int()>  cfunc = [x]() mutable { return ++x; };
8.     std::cout  << cfunc()  <<  std::endl;
9.     std::cout  << cfunc()  <<  std::endl;
10.}
```

std::function und Konstanz-Korrektheit

Ist dieses C++ Programm wohlgeformt?

```
1. #include    <iostream>
2. #include    <functional>
3.
4. int  main()
5. {
6.     int  x = 0;
7.     const std::function<int()>  cfunc = [x]() mutable { return ++x; };
8.     std::cout  << cfunc()  <<  std::endl;
9.     std::cout  << cfunc()  <<  std::endl;
10.}
```

Ist das Verhalten wohldefiniert?

std::function und Konstanz-Korrektheit

Die Typsicherheit der Konstanz-Korrektheit kann umgangen werden, ohne `mutable` oder `const_cast` zu benutzen:

```
1. struct  const_incorrect
2. {
3.     int  x = 0, *ptr_x = &x;
4.     void  inc_x() const { ++*ptr_x;  /* ++x;  wäre nicht wohlgeformt */ }
5. };
6. …
7.     const const_incorrect  cx;
8.     cx.inc_x();     /* hier wird der Versuch gemacht, ein konstantes
9.                        Unterobjekt zu modifizieren */
```

Beispiel aus Produktion: moveonly_function

Bedarf für weiteren Funktorencontainer

Gelegentlich ist man nicht aufs Kopieren angewiesen, würde aber moveonly-Typen übergeben können.

In C++11 gibt es zwar keine moveonly Lambda-Objekte, ein Workaround kann mit wenig Aufwand auf Bibliotheksebene entwickelt werden.

In C++14 gibt es dafür einen deduzierten Sprachkonstrukt (siehe „generalized lambda captures”).

Primäre Anwendungsbeispiele sind Schnittstellen von Threads oder Tasks.

Beispiel aus Produktion: moveonly_function

Schablonenentwurf	charakteristische Funktionalität
`std::function`	Der Aufrufoperator ist const-qualifiziert, es werden aber auch Funktoren akzeptiert dessen Aufrufoperator nicht const-qualifiziert ist. Objektinstanzen der Klasse lassen sich kopieren. Moveonly-Typen werden nicht akzeptiert.
`dynamic_function`	Objektinstanzen der Klasse lassen sich kopieren. Moveonly-Typen werden akzeptiert, schmeißen aber beim Kopieren. Es gibt sowohl den const-qualifizierten als auch den nichtqualifizierten Aufrufoperator. Der const-qualifizierter schmeißt beim Aufruf, wenn der Ausgangsfunktor solchen nicht bereitstellt.
`moveonly_function`	Der Aufrufoperator ist nicht const-qualifiziert. Akzeptiert sowohl kopierbare als auch moveonly-Typen. Kann nicht kopiert werden.
`moveonly_constfunction`	Der Aufrufoperator ist const-qualifiziert. Akzeptiert nur Funktoren, dessen Aufrufoperator auch const-qualifiziert ist.

Design-Vorgaben

1. Wertesemantik
2. Kleinobjekt Optimierung
3. Konstanz-Korrektheit
4. Nichtschmeißende Konstruktion (Zeile 7., 8.)
5. Überladefähigkeit bei Konstruktion (Zeile 4., 5.)

```
6.  void  use_functor(moveonly_function<void(int)>);
7.  void  use_functor(moveonly_function<void(double)>);
8.  …
9.      use_functor([](int){});
10.     use_functor([](double){});
11.     …
12.     use_functor({std::nothrow, [](int){}});
13.     use_functor({std::nothrow, [](double){}});
```

Design-Vorgaben

Gory Details der Implementierung

Zuerst werden die Hilfsschablone `func_storage` und das Konzeptinterface `f_interface` definiert.

```
1.template<class result_type, class... param_types> struct f_interface;
2.
3. template<class result_type, class... param_types>
4. struct func_storage
5. {
6.     using func_interface = f_interface<result_type, param_types...>;
7.     static const uint max_size = 64;
8.     static const uint max_align = 32;
9.     func_interface * m_impl_base;
10.    byte_t m_storage[max_size + max_align - sizeof(func_interface*)];
11. };
```

Gory Details der Implementierung

```
1.template<class  result_type, class...  param_types>
2.struct  f_interface
3.{
4.    using  func_storage_t = func_storage<result_type, param_types...>;
5.    using  destructive_mover = void abicall
6.        (func_storage_t& source, func_storage_t& dest);
7.    using  deleter_type = void abicall(func_storage_t&);
8.    virtual  result_type  abicall  invoke(param_types... params) = 0;
9.    virtual  bool  abicall  is_engaged() = 0;
10.    virtual  deleter_type* abicall  get_deleter() = 0;
11.    destructive_mover * destroy_mover;
12.    f_interface(const f_interface&) = delete;
13.    f_interface& operator=(const f_interface&) = delete;
14.    f_interface() = default;
15.protected:
16.    ~f_interface(){}
17.};
```

Gory Details der Implementierung

Nichtschmeißende inplace-Konstruktion wird in der Schablone `moveonly_function_impl` umgesetzt:

```
1. template<class func_type>
2. explicit moveonly_function_impl(nulltype, std::nothrow_t, func_type && functor) noexcept
3. {
4.   static_assert(!std::is_lvalue_reference<func_type>::value,
5.       "functor is not allowed to be copied into the function object!");
6.   static_assert(meta::is_noexcept_movable<func_type>::value,
7.       "functor must be noexcept movable!");
8.   using inplace_func_impl = func_inplace_impl<func_type>;
9.   const uint func_alignment = std::alignment_of<inplace_func_impl>::value;
10.  static_assert(func_alignment <= func_storage_t::max_align,
11.      "alignment of provided functor is too strict!");
12.  static_assert(sizeof(inplace_func_impl) <= func_storage_t::max_size,
13.      "size of provided functor is too large!");
14.  void * the_storage = m_functor.m_storage;
15.  size_t the_size = sizeof(m_functor.m_storage);
16.  m_functor.m_impl_base = new (std::align(func_alignment,
17.      sizeof(inplace_func_impl), the_storage, the_size)) inplace_func_impl{std::move(functor)};
18. }
```

Gory Details der Implementierung

```
1. template<class func_type>
2. explicit moveonly_function_impl(nulltype,
3.    storage::any_allocator && allocator, func_type && functor)
4. {
5.    using func_value_type = std::remove_const_t<std::remove_reference_t<func_type>>;
6.    using inplace_func_impl = func_inplace_impl<func_value_type>;
7.    const uint func_alignment = std::alignment_of<inplace_func_impl>::value;
8.    if (sizeof(inplace_func_impl) <= func_storage_t::max_size &&
9.        std::alignment_of<inplace_func_impl>::value <= func_storage_t::max_align)
10.   {
11.       void * the_storage = m_functor.m_storage;
12.       size_t the_size = sizeof(m_functor.m_storage);
13.       m_functor.m_impl_base = new(std::align(func_alignment, sizeof(inplace_func_impl),
14.           the_storage, the_size)) inplace_func_impl{std::forward<func_type>(functor)};
15.   }
16.   else
17.   {
18.       using alloc_func_impl = func_alloc_impl<func_value_type>;
19.       auto * new_storage = allocator.allocate<alloc_func_impl>(1u);
20.       m_functor.m_impl_base = new_storage;
21.       exceptional::throw_on_nullptr(new_storage, __func__, __FILE__, sizeof(alloc_func_impl));
22.       const auto & free_on_exception = create_scope_guard
23.           ([&]{ if(new_storage != nullptr) allocator.free(new_storage); });
24.       new (new_storage) alloc_func_impl{std::forward<func_type>(functor), allocator};
25.       new_storage = nullptr;
26.   }
27. }
```

Gory Details der Implementierung

Für eine korrekte Konstanz-Übertragung des Aufrufoperators sorgt der Schablonenparameter `const_invoke`.

```
template<class result_type, class... param_types, bool const_invoke>
class moveonly_function_impl<result_type(param_types...), const_invoke>
{
….
    template<class func_type>
    class func_inplace_impl
    {
        func_type m_func;
        result_type abicall invoke(param_types... params) override
        {
            return static_cast<std::conditional_t
                <const_invoke, const func_type&, func_type&>>
                (m_func)(static_cast<param_types&&>(params)...);
        }
   …
   };
…
};
```

TMP und Tricks mit SFINEA

Um die Überladefähigkeit von `moveonly_function` mit verschiedenen Signaturen bei Konstruktion zu ermöglichen muss der deduzierter Typ des Parameters eingeschränkt werden.

```
template<class  func_type, class  func_val_type =
    std::remove_reference_t<func_type>, class = std::enable_if_t<!std::is_same
    <const func_val_type, const moveonly_constfunction>::value>,
    (func_val_type::*) (param_types...) const = &func_val_type::operator()>
moveonly_constfunction(func_type && functor) :
    m_func{nulltype{}, storage::any_allocator{storage::default_allocator{}},
    std::forward<func_type>(functor)}  {  }
```

```
template<class  func_type, class  func_val_type = std::remove_reference_t<func_type>,
    class = std::enable_if_t<!std::is_same<const func_val_type,
        const moveonly_function>::value  &&  !std::is_same<const func_val_type,
    const moveonly_constfunction<result_type(param_types...)>>::value  &&
    details_::callable_checker<func_val_type, result_type, param_types...>::result>>
moveonly_function(func_type && functor) : m_func{nulltype{}, storage::any_allocator
    {storage::default_allocator{}}, std::forward<func_type>(functor)}  {  }
```

TMP und Tricks mit SFINEA

DON’T PANIC

default template parameter als typedef um die Referenz zu entfernen

SFINEA mit std::enable_if sorgt für die Auflösung der Disambiguierung mit dem Copy-Konstruktor

```
1. template<class  func_type, class  func_val_type =
2.     std::remove_reference_t<func_type>, class = std::enable_if_t<!std::is_same
3.     <const func_val_type, const moveonly_constfunction>::value>,
4.     (func_val_type::*) (param_types...) const = &func_val_type::operator()>
5. moveonly_constfunction(func_type && functor) :
6.     m_func{nulltype{}, storage::any_allocator{storage::default_allocator{}},
7.     std::forward<func_type>(functor)}  {  }
```

In dieser Zeile wird der Funktortyp eingeschränkt, indem die Adresse dessen Aufrufoperators dem Zeiger auf eine Methode mit entsprechende Signatur zugewiesen wird.

```
8. template<class  func_type, class  func_val_type = std::remove_reference_t<func_type>,
9.     class = std::enable_if_t<!std::is_same<const func_val_type,
10.        const moveonly_function>::value  &&  !std::is_same<const func_val_type,
11.    const moveonly_constfunction<result_type(param_types...)>>::value  &&
12.    details_::callable_checker<func_val_type, result_type, param_types...>::result>>
13. moveonly_function(func_type && functor) : m_func{nulltype{}, storage::any_allocator
14.    {storage::default_allocator{}}, std::forward<func_type>(functor)}  {  }
```

Hier ist std::enable_if komplett für SFINEA Auflösung verantwortlich, details_::callable_checker übernimmt die Aufgabe zu überprüfen, ob sich der Funktortyp entweder const- oder nicht const- qualifiziert aufrufen lässt (oder

TMP und Tricks mit SFINEA

Die Einschränkung soll auch für die Konstruktoren, die nicht schmeißen können, durchgeführt werden.

```
1.template<class  func_type, class =
2.    std::enable_if_t<!std::is_lvalue_reference<func_type>::value>,
3.    result_type (func_type::*) (param_types...) const =
  &func_type::operator()>
4.moveonly_constfunction(std::nothrow_t, func_type && functor) noexcept :
5.    m_func{nulltype{}, std::nothrow, std::move(functor)}  {  }

6.template<class  func_type, class = std::enable_if_t
7.    <!std::is_lvalue_reference<func_type>::value  &&
8.    details_::callable_checker<func_type, result_type,
  param_types...>::result>>
9.moveonly_function(std::nothrow_t, func_type && functor) noexcept :
10.    m_func{nulltype{}, std::nothrow, std::move(functor)}  {  }
```

TMP und Tricks mit SFINEA

```
1.template<class  functor_type, class  result_type, class...  param_types>
2.class  callable_checker
3.{
4.    template<class  func_type,
5.        result_type (func_type::*) (param_types...) = &func_type::operator()>
6.    static  uint8_t  is_mutably_callable(int);
7.    template<class  func_type>
8.    static  uint16_t  is_mutably_callable(...);
9.    template<class  func_type,
10.        result_type (func_type::*) (param_types...) const = &func_type::operator()>
11.    static  uint8_t  is_const_callable(int);
12.    template<class  func_type>
13.    static  uint16_t  is_const_callable(...);
14.public:
15.    static  const bool  result = sizeof(is_const_callable<functor_type>(1)) == 1  ||
16.        sizeof(is_mutably_callable<functor_type>(1)) == 1;
17.};
```

Beispiel der Type-Erasure ohne dynamischer Bindung

Wenn mehrere Elemente des gleichen Typs über einen Container übergeben werden sollen, benutzt man häufig eine Referenz auf diesen Container als Ein/Ausgabeparameter.

Dabei wird die Schnittstelle von dem Typ der Datenstruktur abhängig gemacht.

Falls die Elemente fortlaufend im Adressraum gespeicherten sind und man nur an deren Zugriff interessiert ist, kann die Abhängigkeit beseitigt werden, wenn stattdessen roher Zeiger mit zusätzlichem Integer-Parameter als Längeninformation benutzt wird.

Ohne Performanceeinbüße eines dynamischen Aufrufs virtueller Funktionen erzielt man Datenstrukturunabhängigkeit.

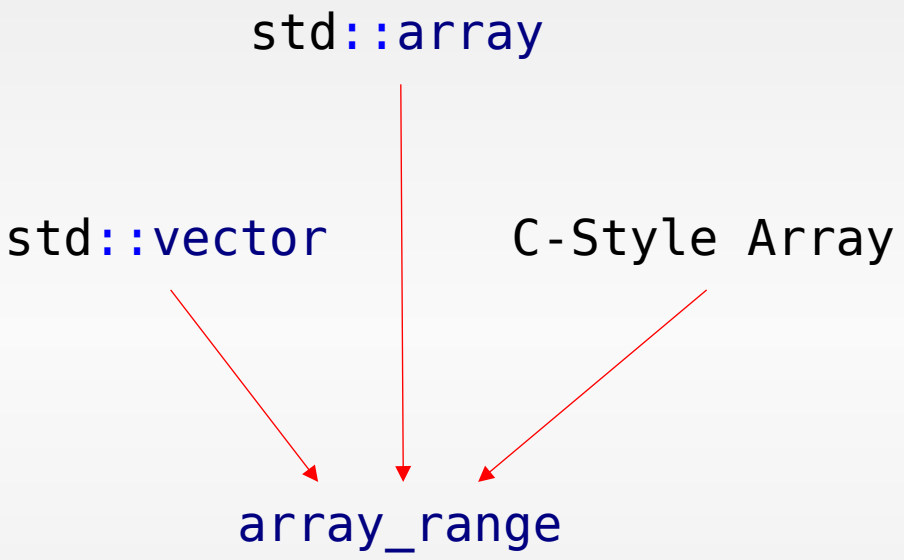
Beispiel der Type-Erasure ohne dynamischer Bindung

array_range als Ein/Ausgabeparameter

Die Schablone `array_range` stellt nichtintrusives Type-Erasure mit Referenzsemantik für Array-ähnliche Datenstrukturen zu Verfügung und lässt sich als Standardlayout-Typ implementieren.

Die Implementierung ist wesentlich kürzer und weniger fehleranfällig als bei `moveonly_function`.

Ähnliche Konzepte sind in den `array_view` TS und `string_view` TS umgesetzt.



Beispiel der Type-Erasure ohne dynamischer Bindung

```
1. template<class  xtype>
2. class  array_range
3. {
4.     xtype * m_ptr;
5.     size_t  m_length;
6. public:
7.     array_range(const array_range&) = default;
8.     array_range& operator=(const array_range&) = default;
9.
10.    array_range(array_range && range) noexcept :
11.        m_ptr{range.m_ptr}, m_length{range.m_length}   {  }
12.
13.    array_range(xtype * ptr, size_t  length) noexcept : m_ptr{ptr}, m_length{length}  {
    }
14.
15.    xtype& operator[](size_t  idx) const noexcept { return  m_ptr[idx]; }
16.
17.    xtype * begin() const noexcept { return  m_ptr; }
18.    xtype * end() const noexcept { return  m_ptr + m_length; }
19.    size_t  count() const noexcept { return  m_length; }
20.};
```

Referenzen

Type-Erasure allgemein:

https://akrzemi1.wordpress.com/2013/11/18/type-erasure-part-i/

http://talesofcpp.fusionfenix.com/post-16/episode-nine-erasing-the-concrete

http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil

Problem mit std::function:

http://compgroups.net/comp.lang.c++.moderated/std-function-and-const-correctness/2983990

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4159.pdf

array_view Proposal:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3851.pdf