2-gsl-arrays

June 3, 2024

1 About arrays

Just as raw pointers are ambivalent, ordinary C arrays pose a number of problems in C ++, including uncontrolled implicit conversions to raw pointers.

The GSL introduced a new type of array, better controlled, at no extra cost, which is now known as `std::array`.

It also introduces a first kind of "view", `gsl::span`, a concept more and more popular in many languages.

1.1 New fixed-size `std::array`

The GSL discussed two new types of arrays: * `stack_array<T>`: an array allocated on the stack, with a fixed size, evaluable at compile time. * `dyn_array<T>` : an array allocated on the heap, because its size is only known at runtime.

Their elements are modifiable, unless the programmer instantiates the templates with a `const` type.

The class `gsl::stack_array<T>` has been integrated into the standard C++11 library, as `std::array<T>`

Regarding `gsl::dyn_array<T>`, it has not been implemented as such, but we can use `std::valarray` for a similar purpose (as long as we do not use the mathematical functions together with a non-numeric parameter type).

1.1.1 Improving management of fixed-size arrays

The `std::array` has **no extra cost** as compared to ordinary arrays. They respond to the **same interface**, except that it is necessary **to specify the type of the elements and the size of the array**. As a bonus, each element receives the default value of its type when this value is not supplied at construction.

[1]:
```
%%file tmp.gsl-arrays.cpp

#include <iostream>
#include <array>

int main() {
    std::array<int,5> arr { 1, 2, 3, 4 } ;
    for ( auto elem : arr )
```

```
      std::cout<<elem<<" " ;
    std::cout<<std::endl ;
}
```

```
Writing tmp.gsl-arrays.cpp
```

[2]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
 ↪gsl-arrays.exe
```

[3]:
```
!./tmp.gsl-arrays.exe
```

```
1 2 3 4 0
```

1.1.2 Possible check of indices

One can access the elements of a `std::array` via the function `at()`, which checks the validity of the index (with extra cost), or via the traditional operator `[]`, which does not perform the check.

[4]:
```
%%file tmp.gsl-arrays.cpp

#include <iostream>
#include <array>

int main()
 {
  std::array<double, 5> array_double { 1.2, 3.4, 5.6 } ;

  std::cout << "4th value = " << array_double[3] << std::endl ;
  array_double[4] = 9.9 ;
  array_double[10] = 1.1 ; // Undefined behavior - no exception

  std::cout << "5th value = " << array_double.at(4) << std::endl ;
  array_double.at(10) = 1.1 ; // Exception std::out_of_range
 }
```

```
Overwriting tmp.gsl-arrays.cpp
```

[5]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
 ↪gsl-arrays.exe
```

[6]:
```
!./tmp.gsl-arrays.exe
```

```
4th value = 0
5th value = 9.9
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 10) >= _Nm (which is 5)
Aborted (core dumped)
```

1.1.3 Use of standard algorithms

The usual algorithms of the standard library can be used, as with any other standard container.

[7]:
```
%%file tmp.gsl-arrays.h

#include <array>
#include <iostream>

template <std::size_t N>
using DoubleArray = std::array<double,N> ;

template <std::size_t N>
void print( DoubleArray<N> const & values )
 {
  for ( double value : values )
   { std::cout << value << " " ; }
  std::cout << std::endl ;
 }
```

```
Writing tmp.gsl-arrays.h
```

[8]:
```
%%file tmp.gsl-arrays.cpp

#include "tmp.gsl-arrays.h"
#include <algorithm>

int main()
 {
  DoubleArray<5> a1 { 1.2, 3.4, 5.6 } ;
  DoubleArray<5> a2 { 0.1, 0.2, 0.3, 1.4, 0.5 } ;
  DoubleArray<5> a3 ;

  std::transform(a1.begin(),a1.end(),a2.begin(),a3.begin(),
    []( double v1, double v2 ){ return v1+v2 ; } ) ;
  print(a3) ;

  std::sort(a3.begin(),a3.end()) ;
  print(a3) ;
 }
```

```
Overwriting tmp.gsl-arrays.cpp
```

[9]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
 ↪gsl-arrays.exe
```

[10]:
```
!./tmp.gsl-arrays.exe
```

```
1.3 3.6 5.9 1.4 0.5
0.5 1.3 1.4 3.6 5.9
```

1.2 To reference a sequence of values: `gsl::span<T>`

An instance of `span<T>` (previously called `array_view<T>`) denotes a sequence of modifiable instances of `T` (unless `T` is a `const` type). We are talking about a **non-owner view** in the sense that the instance of `span<T>` is built on top of another array, and only provides a simplified and standardized access to its elements.

1.2.1 Construction

This type is the equivalent of a pointer and a size, and is based on the assumption that the **elements of the sequence are stored contiguously**. We can therefore only build a `gsl::span<T>` on top of arrays which comply with this memory contiguity, such as `std::vector<T>`, `std::array<T>`, `std::string` or a raw array `T[N]`.

[12]:
```
%%file tmp.gsl-arrays.h

#include <iostream>
#include <gsl/gsl>

void display( gsl::span<int const> data )
 {
  for ( auto e : data ) std::cout << e << ' ';
  std::cout << std::endl;
 }
```

```
Overwriting tmp.gsl-arrays.h
```

[13]:
```
%%file tmp.gsl-arrays.cpp

#include "tmp.gsl-arrays.h"
#include <vector>
#include <array>

int main()
 {
  std::vector<int> v { 1, 2, 3, 4, 5 };
  display(v) ;

  std::array<int, 5> a {1, 2, 3, 4, 5};
  display(a) ;

  int arr[] { 1, 2, 3, 4, 5 } ;
  display(arr) ;
}
```

```
Overwriting tmp.gsl-arrays.cpp
```

[14]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
↪gsl-arrays.exe
```

[15]:
```
!./tmp.gsl-arrays.exe
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

1.2.2 Static size

In the previous examples, the view size was determined at runtime. When this size is known in advance, it can be specified explicitly and benefit from an optimized implementation.

[16]:
```
%%file tmp.gsl-arrays.h

#include <iostream>
#include <gsl/gsl>

template <typename T, std::ptrdiff_t size = -1>
void display( gsl::span<T,size> data )
 {
   for( auto e : data)
    { std::cout << e << ' ' ; }
   std::cout << std::endl ;
 }
```

```
Overwriting tmp.gsl-arrays.h
```

[17]:
```
%%file tmp.gsl-arrays.cpp

#include "tmp.gsl-arrays.h"
#include <vector>
#include <array>

int main()
 {
  std::vector<int> v { 1, 2, 3, 4, 5 };
  display(gsl::span<int>{v}) ;

  std::array<int, 5> a {1, 2, 3, 4, 5};
  display(gsl::span<int,5>{a}) ;

  int arr[] { 1, 2, 3, 4, 5 } ;
  display(gsl::span<int,5>{arr}) ;
}
```

```
Overwriting tmp.gsl-arrays.cpp
```

[18]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
↪gsl-arrays.exe
```

[19]:
```
!./tmp.gsl-arrays.exe
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

In `main`, we help with explicit calls to `gsl::span` constructors, because the compiler cannot both deduce the type of `T` (int) and insert the implicit conversion into `gsl::span`.

Actually, when the size parameter of some `std::span` is not specified, it receives the default value `-1`, which means `dynamic_extent`.

1.2.3 Subviews

A part of a view can be extracted from the start, from the end, or between the start and end the index (excluded) **at runtime, or at compile time** if the indices are already known.

[20]:
```
%%file tmp.gsl-arrays.cpp

#include "tmp.gsl-arrays.h"
#include <vector>

int main()
 {
  std::vector<double> data { 1.1, 2.2, 3.3, 4.4, 5.5 } ;
  gsl::span<double> s{ data } ;

  // run time
  display(s.first(2)) ;
  display(s.last(2)) ;
  display(s.subspan(2,3)) ;
}
```

```
Overwriting tmp.gsl-arrays.cpp
```

[21]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
 ↪gsl-arrays.exe
```

[22]:
```
!./tmp.gsl-arrays.exe
```

```
1.1 2.2
4.4 5.5
3.3 4.4 5.5
```

[23]:
```
%%file tmp.gsl-arrays.cpp

#include "tmp.gsl-arrays.h"

int main()
 {
```

```
  std::array<int,5> data { 1, 2, 3, 4, 5 } ;
  gsl::span<int,5> s{ data } ;

  // compile time
  display(s.first<2>()) ;
  display(s.last<2>()) ;
  display(s.subspan<2,3>()) ;
}
```

```
Overwriting tmp.gsl-arrays.cpp
```

[24]:
```
!rm -f tmp.gsl-arrays.exe && g++ -std=c++17 -I./ tmp.gsl-arrays.cpp -o tmp.
↪gsl-arrays.exe
```

[25]:
```
!./tmp.gsl-arrays.exe
```

```
1 2
4 5
3 4 5
```

1.2.4 Some details

- `gsl::span` does not check its indices (before C++26), like `std::vector::at`, but can help debugging.
- You will not be able to construct a view from an instance of `initializer_list`, since it is an array of an ephemeral nature, whereas a view must rely on a durable underlying structure.
- The C++20 standard contains a `std::span` similar to the `gsl::span`.
- There is a high demand for multi-dimensional views, which Microsoft referred to as `multi_span` and others call `mdspan`. This does not appear yet in C++20.

1.3 Multiple dimensions with `std::mdspan<T>` (C++23)

Highly demanded for years, we now have `mdspan` which provides a multi-dimensional-like access to a contiguous area of memory. It is part of the C++ 23 standard library… but not yet implemented by GCC. The demonstration code below is based on the reference implementation bu the Kokkos team:

[26]:
```
%%file tmp.mdspan.cpp

#include <iostream>
#include <mdspan/mdspan.hpp>
#include <vector>
#include <format>

int main() {
    std::vector v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

    // View data as contiguous memory representing 2 rows of 6 ints each
    auto ms2 = Kokkos::mdspan(v.data(), 2, 6);
```

```
    // View the same data as a 3D array 2 x 3 x 2
    auto ms3 = Kokkos::mdspan(v.data(), 2, 3, 2);

    // Write data using 2D view
    for (std::size_t i = 0; i != ms2.extent(0); i++)
        for (std::size_t j = 0; j != ms2.extent(1); j++)
            ms2[i, j] = i * 1000 + j;

    // Read back using 3D view
    for (std::size_t i = 0; i != ms3.extent(0); i++) {
        std::cout<<std::format("--- slice {} ---", i)<<std::endl;
        for (std::size_t j = 0; j != ms3.extent(1); j++) {
            for (std::size_t k = 0; k != ms3.extent(2); k++)
                std::cout<<std::format("{} ", ms3[i, j, k]);
            std::cout<<std::endl;
        }
    }
}
```

```
Overwriting tmp.mdspan.cpp
```

[27]:
```
!rm -f tmp.mdspan.exe && g++ -std=c++23 -I./ -I./mdspan/include tmp.mdspan.cpp␣
↪-o tmp.mdspan.exe
```

[28]:
```
!./tmp.mdspan.exe
```

```
--- slice 0 ---
0 1
2 3
4 5
--- slice 1 ---
1000 1001
1002 1003
1004 1005
```

2 Take away (paraphrasing Etienne Boespflug)

- Better than a pointer/size pair, `gsl::span` is easy to pass as an argument and it responds to a container interface, sparing us the arithmetic of pointers.
- It explicitly shows that we are working with an intermediate view, rather than on the container itself.
- Better than a `const std::vector<T>&`, a `gsl::span<T>` argument can be copied at no extra cost, and can stay on top of any "continuous memory" container.
- BUT: beware not to carry your span somewhere where the underlying container would go out of scope.
- FRESH: mdspan is almost here !

3 Questions ?

4 Exercise

How about writing your own `my_span<T>` ? 1. Provide a `my_span<T>` constructor taking a `std::vector<T>` as input, add function `size()` and `operator[]`, which should make the code compilable. 2. Try to turn the constructor into a template, assuming that the object passed to the constructor has member functions `size()` and `operator[]`, check that the code also works if you substitute the `std::vector` with a `std::array`.

[26]:
```
%%file tmp.gsl-collections.cpp

#include <iostream>
#include <vector>

// ... PUT HERE YOUR my_span<T> IMPLEMENTATION ...

void print( my_span<int> data )
 {
   for( std::size_t i = 0 ; i < data.size() ; ++i )
     std::cout << data[i] << ' ' ;
   std::cout << std::endl ;
 }

int main()
 {
  std::vector<int> arr { 1, 2, 3, 4, 5 } ;
  print(arr) ;
  return 0 ;
}
```

```
Writing tmp.gsl-collections.cpp
```

[27]:
```
!rm -f tmp.gsl-collections.exe && g++ -std=c++17 -I./ tmp.gsl-collections.cpp␣
 ↪-o tmp.gsl-collections.exe
```

```
tmp.gsl-collections.cpp:7:6: error: variable or
field ‘print’ declared void
    7 | void print( my_span<int> data )
      |      ^~~~~
tmp.gsl-collections.cpp:7:13: error:

‘my_span’ was not declared in this scope
    7 | void print( my_span<int> data )
      |             ^~~~~~~
tmp.gsl-collections.cpp:7:21: error: expected
primary-expression before ‘int’
    7 | void print( my_span<int> data )
      |                     ^~~
```

```
tmp.gsl-collections.cpp: In function ‘int
main()’:
tmp.gsl-collections.cpp:17:3: error:
‘print’ was not declared in this scope; did you mean
‘printf’?
   17 |   print(arr) ;
      |   ^~~~~
      |   printf
```

[ ]:
```
!./tmp.gsl-collections.exe
```

4.1 Sources

- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#gsl-guidelines-support-library
- http://modernescpp.com/index.php/c-core-guideline-the-guidelines-support-library
- http://nullptr.nl/2018/08/refurbish-legacy-code/
- https://etienne-boespflug.fr/cpp/257-c20-stdspan/
- https://github.com/microsoft/GSL/blob/master/include/gsl/span
- http://codexpert.ro/blog/2016/03/07/guidelines-support-library-review-spant/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html
- https://github.com/kokkos/mdspan