Arrays

Mittwoch, 29. Mai 2024 20:56

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

Back to basics.

C-style arrays have one interesting property - they decay to the pointer - either implicitly: passing it to the function that expects pointer (by value), or explicitly: calling the std::decay::type conversion traits facility, that strip away references and cv qualifiers, and perform array-to-pointer conversion.
This is due to fact that arrays reserve the contiguous block of memory, that can be addressed with the very first element of the array: Address of an array is address of its first element.

This means, if we have the function that takes as an argument a *pointer* - only to test the correctness for the given argument type (without the body)

```
template <typename T>
constexpr void func([[maybe_unused]] T* ptr) noexcept {}
```

both expressions will be correct - for the very same reason

```
auto ptr = new int{1};
int arr[] = {1,2,3,4};

func(ptr);
func(arr);
```

Furthermore, we can introduce another helper function - that compares at compile time equality of two types

```
template <typename T1, typename T2>
void is_equal() noexcept
{
    constexpr bool same = std::is_same_v<T1, T2>;
    std::cout << std::boolalpha << same << '\n';
}
```

We can pretend that we don't know - how the expressions will be deduced - by using appealing *auto* feature

```
auto ptr = new int{1};
auto arr = new int[3]{1,2,3};


is_equal<decltype(ptr), decltype(arr)>();
```

For *ptr* should be obvious - int*.

What about the *arr* type?
This may not be so obvious - at least not to beginners.
@hint The *CppInsights* can be useful tool to get insights into generated code

Let's break it down.
Calling the operator ::new T[N] - we allocate the contiguous block of the memory on the heap, construct each element in the memory - either by calling default constructor of T, or initialize each element with default value (for primitive types).
@digression In case that T has no default constructor, it must be called with additional argument(s):

```
template <typename T, std::size_t N, typename...Args>
requires std::constructible_from<T, Args...> && (N >0)
auto make_scope_arr(Args&&...args)
{
    auto* arr = new T[N]{std::forward<Args>(args)...};
    return CScope<T, std::default_delete<T[]>>(arr);
}
```

#Example: https://godbolt.org/z/Goq8afMx9

The operator ::new() will return the pointer - the address of the first element in the array.
In other words - it will be deduced to the **int***.

So, this check will print '*true*', same as

```
is_equal<decltype(arr), int*>();
```

Array type

Let's talk more about the arrays.
What would be the output of the fallowing expressions

```
int a[] = {1,2,3,4};
is_equal<decltype(a), int[]>();
```

C-style array as any primitive types - belongs to the language arsenal.
C-style array is **static** one `- it can't be resized in memory.`
Not only that - the number of elements in array - its size, is embedded into type definition.
Therefore - this will print "*false*".

The proper type would actually be

```
is_equal<decltype(a), int[4]>();
```

Or - to emphasize the importance of the size

```
is_equal<decltype(a), int[size(a)]>();
```

Where we used the well-known utility method for determine the size of the array

```
template<typename T, std::size_t N>
constexpr auto size(T (&)[N]) noexcept { return N; }
```

std::array

The importance of the size - the fact that it's part of the type definition,
becomes more obvious, with the *std* version of the array: **std::array**

```
template<class T, std::size_t N> struct array;
```

`It's compile-time construct, that will be created on the stack.`

Be aware that, although effectively represent the same data structure, they
are different types

```
std::array<int, 4> arr2;
is_equal<decltype(a), decltype(arr2)>(); // false
```

Code to play with
https://godbolt.org/z/dxGebz9Tn

Multidimensional arrays

Arrays are one of the most valuable data structure, especially for mathematical models, where we quite oft need to represent the data as multidimensional arrays - matrices.
@digression For those who worked with MATLAB - the entire language is designed to be comfortable to work with these kind of models, where you can easily invert, transpose or rotate the matrices.

Assume that we have a two-dimensional array

```
constexpr int A[3][2] = {
    {1, 2},
    {3, 4},
    {5, 6}
};

static_assert(A[2][0] == 5 && A[2][1] == 6, "Wrong initialization");
```

How would we do the same, using C++ *std::array* structure?
Here is the code in rescue:

```
template <typename T, std::size_t ROWS, std::size_t COLUMNS>
using table_type = std::array<std::array<T, COLUMNS>, ROWS>;
```

Notice the extra parenthesis in initialization ("*array of arrays*")

```
constexpr table_type<int, 3, 2> t =
{// outer std::array - ROWS
    { // inner arrays - COLUMNS
        {1, 2},
        {3, 4},
        {5, 6}
    }
```

```
};

static_assert(t[2][0] == 5 && t[2][1] == 6, "Wrong initialization");
```

Notice how complexity increases - even when it comes to simple initialization.

As a valuable reminder: the type maybe two-dimensional array - but the memory itself remains (as always) linear.
This is performance significant: to choose proper traversing strategy around the multi-dimensional arrays -
to utilize on the *space locality*, and avoid *cache misses*.

Let's demonstrate this on the concreate example: drawing the chess board

Instead of having two-dimensional array - we can simplified it with the array of fields: actually the colors of the fields

```
using enum RGBColors;

constexpr std::uint8_t ROWS = 8;
constexpr std::uint8_t COLUMNS = 8;
std::array<RGBColors, ROWS * COLUMNS> board;
```

We can iterate as with two-dimensional array - utilizing on the *space locality*

```
for (std::uint8_trow = 0; row < ROWS; ++row)
{
    for (std::uint8_t col = 0; col < COLUMNS; ++col)
    {
        const bool white = (is_odd(row) && not is_odd(col)) || (not is_odd(row) && is_odd(col));
        board[row * COLUMNS + col] = white ? WHITE : BLACK;
    }
}
```

@note If we were decided to switch the logic: to use rows as inner loop, the way how we access the block of the memory would be highly inefficient - since we would jump with offset of COLUMNS size, which would cause a lot of cache misses - and reloading from the main memory

Code to play with
https://godbolt.org/z/j9747KMYv

Links
std::decay - cppreference.com