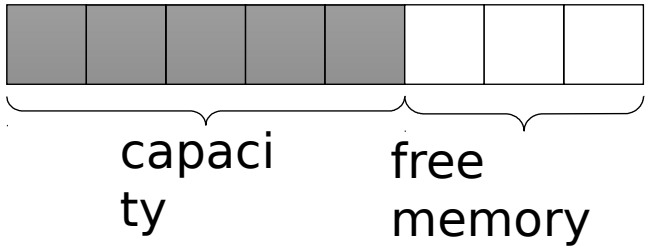
P0894 – realloc() for C++

Victor Dyachenko, Antony Polukhin

2018-11

The problem

- Contiguous containers (std::string & std::vector) have to relocate all the elements every time they increase the capacity. Even when adjacent memory block of required size is available.



- C language uses realloc() which has behaviour inappropriate for C++ object model

Solution

1. Add an optional function to allocator’s interface:

```
bool resize_allocated(pointer p, size_type cur_size, size_type
new_size);
```

2. Add similar wrapper function to std::allocator_traits that just always returns `false` if allocator doesn’t define such function:

```
static bool resize_allocated(Alloc &a, pointer p, size_type cur_size, size_type new_size);
```

3. Make std::string & std::vector use this function.

What has the call to do?

```
bool resize_allocated(pointer p, size_type cur_size, size_type
new_size);
```

- Check if the buffer can be expanded/shortened.
- Return `false` if not.
- Resize the buffer and return `true` otherwise.

What has the container to do?

- Just use the resized memory buffer if the call succeeds (`true` is returned).
- Fallback to the current buffer relocation approach when the call fails (`false` is returned).

Expected performance impact

- No performance impact is expected when the used allocator doesn’t define resize_allocated() function. The code in the container effectively becomes if(false) { … } which can be eliminated by optimizer.
- If used allocator defines such function and the call succeeds we eliminate new buffer allocation call + elements copy/move calls + old buffer deallocation call.
- If used allocator defines such function and the call fails we have an additional call + boolean check overhead (which expected to be compensated by successful calls).

Backward compatibility impact

- The feature is a pure library extension of std::allocator_traits interface. Just one additional customization point.
- We can decide don’t touch std::allocator interface at all or allow implementation to define or not to define resize_allocated() function. The first approach is fully backward compatible but users don’t get the feature “for free” – they have to manually replace the allocator.

Intended ship vehicle

- We target the nearest Standard update (C++20) because the earlier the STL containers can use this feature the earlier we can get support in the system allocators (like jemalloc, etc).

Bonus feature #1

- Combine with P0401 - Extensions to the Allocator interface (Jonathan Wakely). Add some feedback from allocator – allow allocator to tell the actual size of the allocated block via an additional output parameter:

```
bool resize_allocated(pointer p, size_type cur_size, size_type &new_size);
```

Bonus feature #2

- Pass the preferred and the minimum requested size. The allocator has to try to satisfy the preffered size first. If failed then try to satisfy the less size (bonus #1 is

```
bool resize_allocated(pointer p, size_type cur_size, size_type &new_size);
bool resize_allocated(pointer p, size_type cur_size, size_type &preferred_size, size_type min_size);
```

- The idea was borrowed from [N2045 - Improving STL Allocators]

std::allocator

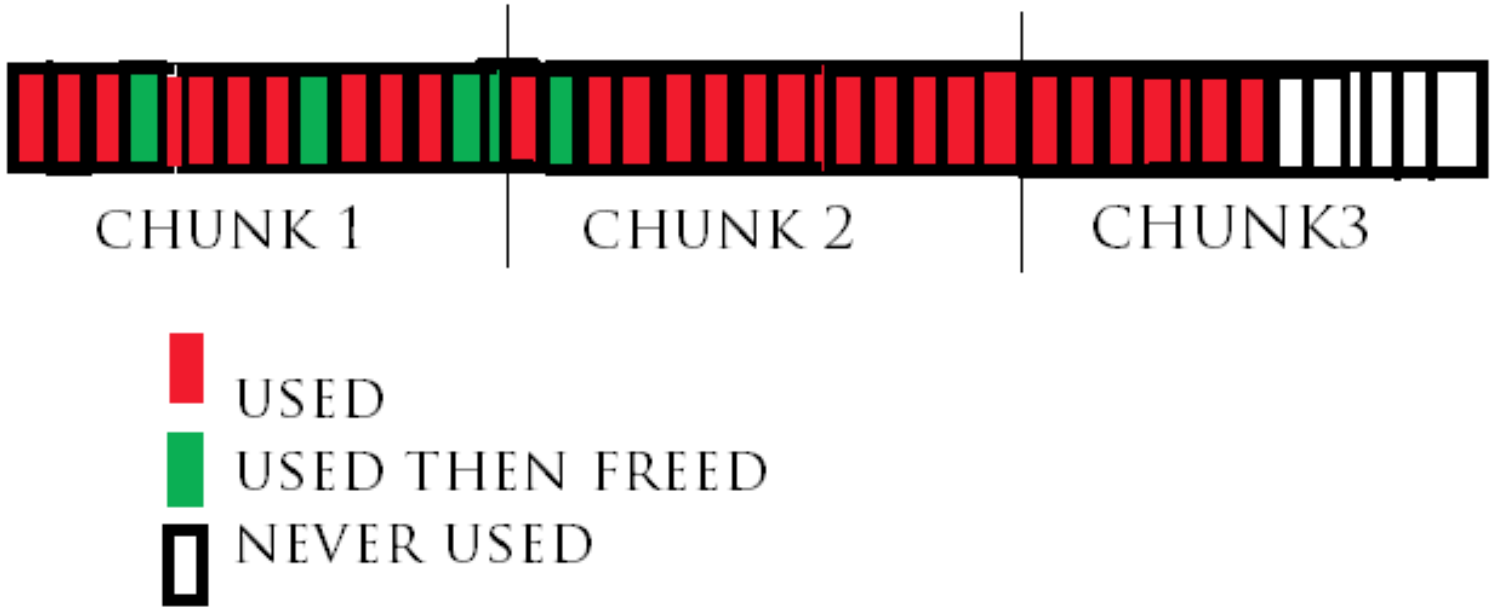
~~std::allocator~~

pool_allocator / monotonic_allocator

```
template <class T>
using pool_vector
    = std::vector<T, pool_allocator<T, N>>;
```

pool_allocator / monotonic_allocator

```
template <class T>
using pool_vector
    = std::vector<T, pool_allocator<T, N>>;
```



stack_vector

```
template <class T, size_t N> struct stack_allocator {
    aligned_storage_t<T> d[N];

    bool resize_allocatoed(...); // throws if buffer is small
};

template <class T, size_t N>
using stack_vector
    = std::vector<T, stack_allocator<T, N>>;
```

small_vector

```
template <class T, size_t N> struct small_allocator {
    aligned_storage_t<T> d[N];

    bool resize_allocatoed(...); // `new` if buffer is small
};

template <class T, size_t N>
using small_vector
    = std::vector<T, small_allocator<T, N>>;
```

small_string

```
template <class T, size_t N> struct small_allocator {
    aligned_storage_t<T> d[N];

    bool resize_allocatoed(...); // `new` if buffer is small
};

template <size_t N> using small_string
    = std::basic_string<char, char_traits<char>,
small_allocator<char, N>>;
```

small_deque

```
template <class T, size_t N> struct small_allocator {
    aligned_storage_t<T> d[N];

    bool resize_allocatoed(...); // `new` if buffer is small
};

template <class T, size_t N>
using small_deque
    = std::deque<T, small_allocator<T, N>>;
```