Richard Kaiser
https://www.rkaiser.de/

C++17 Polymorphic Memory Resources (pmr) and STL Containers for Embedded Applications

[Talk at the https://www.embo.io/]

You can download this manuscript from ***https://www.rkaiser.de/downloads/***

For many embedded C++ applications, compliance with the AUTOSAR or Misra rules is required. Among them is **AUTOSAR Rule A18-5-5**:

> *Memory management functions shall ensure the following:*
> *(a) deterministic behavior resulting with the existence of worst-case execution time,*
> *(b) avoiding memory fragmentation,*
> *(c) avoid running out of memory,*
> *(d) avoiding mismatched allocations or deallocations,*
> *(e) no dependence on non-deterministic calls to kernel.*

This rule has far-reaching consequences, because per default the C++ standard library containers allocate their memory with *new* and free it with *delete*. These calls

- do not have deterministic execution times.
- can cause memory fragmentation.

Since *new* and *delete* violate A18-5-5, the default STL containers must not be used in applications requiring AUTOSAR compliance. This holds for many embedded applications.

With the allocators available since C++17 in the namespace *std::pmr* (polymorphic memory resources) these requirements can often be satisfied. This means that, for the first time in the history of C++, the **containers and algorithms** of the C++ standard library can be used in applications that require AUTOSAR Rule A18-5-5.

1.1 Introduction

1.1.1 About Me

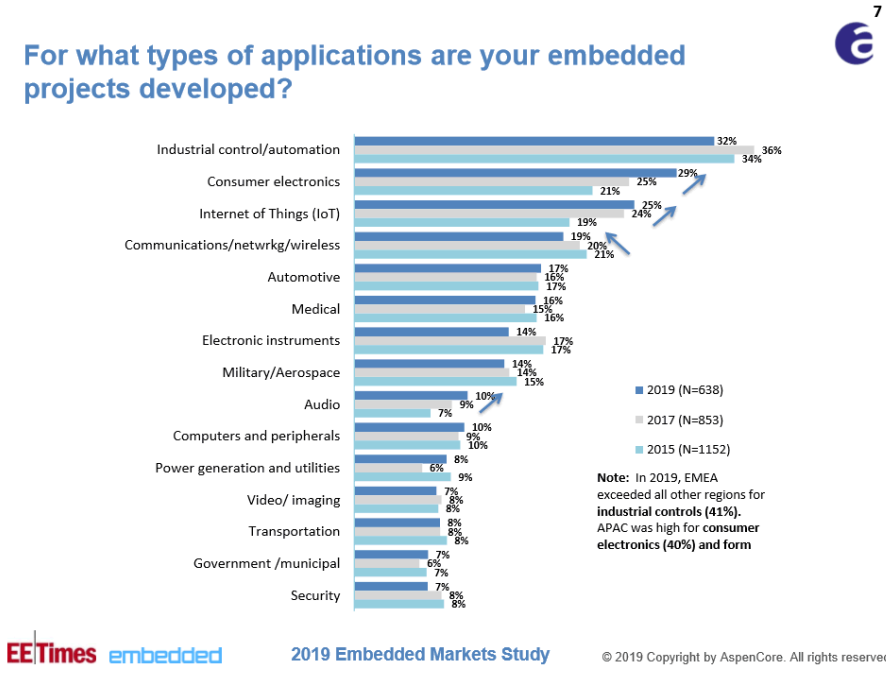
I am a freelancing C++ trainer since more than 30 years, software developer and author of several books about C++. Until 2017, I was professor at the Baden-Württemberg Cooperative State University. Please take a look at my website:

https://www.rkaiser.de/

1.1.2 Embedded Applications

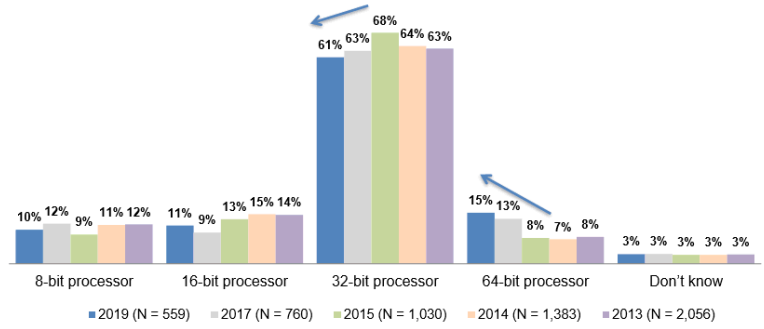
Although the term "embedded application" gives the impression of a clear and unambiguous demarcation, this is by no means the case. If you ask 5 people what this term means to them, you will get 10 different answers.

This is because embedded applications are very multi-faceted. They cover a wide range of applications running under very different hardware platforms. Often, they are control systems that have to run reliably for a long time.



66

My current embedded project's main processor is a:



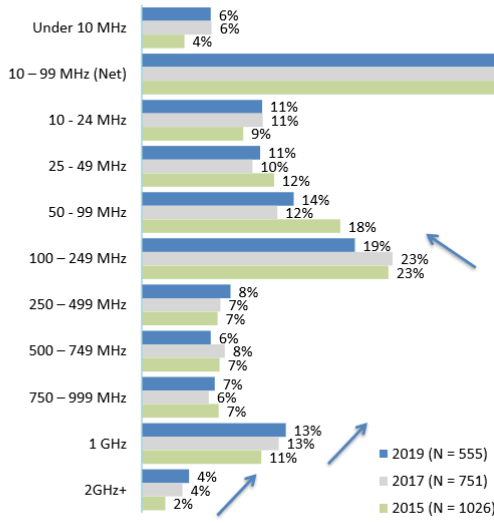
71% of **EMEA** users use 32-bit chips as their main processor.

Additional chips to the main processor	
Primarily 8-bit processors	19%
Primarily 16-bit processors	15%
Primarily 32-bit processors	55%
Primarily 64-bit processors	12%

EETimes embedded 2019 Embedded Markets Study

67

My current embedded project's main processor clock rate is:



The average processor clock rate was:
462 MHz in 2019
445 MHz in 2017
397 MHz in 2015
428 MHz in 2014

EETimes embedded 2019 Embedded Markets Study

1.1.3 Tracking *new* and *delete*

Before we introduce pmr allocators in section 1.2, we first show how to use overloaded *new* and *delete* operators to monitor calls to *new*- and *delete*.

For this purpose, global variables are defined that count each call of *new*- and *delete*

```
int new_counter = 0;    // Number of new calls
int delete_counter = 0; // Number of delete calls
size_t allocated_mem = 0; // Allocated memory in bytes

void reset_counter()
{
  new_counter = 0;
  delete_counter = 0;
  allocated_mem = 0;
}
```

and which are displayed, for example, as follows:

```
void new_delete_summary()
{
  std::cout << std::dec << "#new: " << new_counter << " #delete: "
   << delete_counter << " #bytes: " << allocated_mem << std::endl;
  reset_counter();
}
```

If you define overloaded *new* and *delete* operators, they will be called every time you call *new* and *delete*. The overloaded operators must have this signature:

```
void* operator new(std::size_t sz)
{ // Like the predefined new operator, this operator should only
  // call malloc and throw an exception if necessary
  void* ptr = std::malloc(sz);
  if (ptr)
  {
    new_counter++;      // The only differences from the predefined
    allocated_mem += sz;//                              new-Operator
    return ptr;
  }
  else throw std::bad_alloc{};
}

void operator delete(void* ptr) noexcept
{
  delete_counter++;
  std::free(ptr);
}
```

Of course, here you can log not only the number of calls of *new* and *delete*.

Example: After calling

```
void one_explicit_new_and_delete()
{
  int* pi = new int;

  delete pi;
}
```

you get

```
#new: 1 #delete: 1 #bytes: 4
```

For STL containers, *new* and *delete* are called implicitly. After some *push_back*s

```
void vector_with_implicit_heap_allocations()
{
  std::vector<int> v;
  for (int i = 0; i < 10; i++)
    v.push_back(i);
}
```

you get

```
#new: 7 #delete: 7 #bytes: 152
```

We will occasionally look at the contents of memory using a function like *show_memory*:

```
void showmemory(unsigned char* buffer, std::size_t buffer_size,
                                        const char* headline = "")
{
  if (headline != "")
    std::cout << headline << std::endl;
  std::cout << "&buffer=0x" << std::hex << (unsigned int)(buffer)
        << " " << std::dec << buffer_size << " bytes" << std::endl;
  int i = 0;
  while (i < buffer_size)
  {
    int first = i;
    int last = i + std::min(10, int(buffer_size - first));
    std::cout << "&=" << std::setw(2) << std::hex <<
                                          std::size_t(first);

    std::cout << " asc: ";
    for (int k = first; k < last; k++)
    {
      if ((buffer[k] >= 32) and (buffer[k] <= 127))
        std::cout << buffer[k];
      else
        std::cout << ".";
    }
```

```
      i = i + 10;
      std::cout << std::endl;
    }
    std::cout << std::endl;
  }
```

1.2 Allocators and polymorphic memory resources

The main features of polymorphic allocators are shown below using examples like

```
void vector_with_heap_memory(int n)
{
  std::vector<std::string> container;
  // work with the container
  for (int i = 0; i < n; ++i)
  { // No small string optimization (SSO) is desired here:
    container.push_back("A string with more than 16 chars");
  }
}
```

Here strings with more than 16 characters are stored in the *vector* so that the characters are stored on the heap. In Visual C++, for smaller strings, the characters are stored in the string and thus on the stack ("small string optimization" - SSO).

These examples use

```
std::pmr::vector
```

instead of *std::vector*. Although at first glance this looks like a separate *vector* class in the *pmr* namespace, it is nothing else as the *std::vector* class with a special allocator:

```
namespace pmr {
  template <class T>
  using vector = std::vector<T, polymorphic_allocator<T>>;
} // namespace pmr
```

The standard library containers (except *std::array*) are constructed in such a way that the functions for allocating and freeing memory are encapsulated in an allocator which is passed as a template parameter. By default, this is *std::allocator<T>*, which allocates memory with *new* and frees it with *delete*:

```
template<typename T, typename Allocator = std::allocator<T>>
class vector;
```

As for *vector*, there are pmr variants for all other sequential and associative containers. The container classes from *std::pmr* differ from those from the namespace *std* only in that the allocator *polymorphic_allocator* is used instead of *std::allocator*.

The pmr classes are available only since C++17 and are not yet very widely used. Therefore, thorough tests are recommended.

1.2.1 Containers that are using a *monotonic_buffer_resource*

The following examples are not limited to *std::vector*, but can be applied to any sequential and associative container except *std::array*.

With the class *std::pmr::monotonic_buffer_resource*, available after

```
#include <memory_resource>
```

you can allocate memory to a container, which is then used by the container instead of heap memory. This way, when **working with the container, *new* or *delete* are never called**.

1. This **memory** is passed to the constructor

   ```
   monotonic_buffer_resource(void* buffer, std::size_t buffer_size);
   ```

   with its address and size. This can be, for example, an array created on the stack:

   ```
   void vector_with_stack_memory(int n)
   {  // The only difference are the first three statements:
     std::array<unsigned char, 100'000> memory; // local definition
     // use memory as memory for the vector and the strings:
     std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                               memory.size() };
     std::pmr::vector<std::pmr::string> container{&pool}; // see 7.

     // work with the container
     for (int i = 0; i < n; ++i)
     {
       container.push_back("A string with more than 16 chars");
     }
   } // #new: 0 #delete: 0 #bytes: 0
   ```

   If the memory of *memory* is sufficient for all operations with the container, no more *new* and *delete* take place. But if it is not sufficient, the additional memory is allocated with *new* and *delete* on the heap. See also 5.

   Stroustrup reports in "A tour of C++", ch. 13.6 on an application where the switch to polymorphic memory resources (essentially changing only the first three lines as above) reduced the memory requirement from 6 gigabytes to 300 megabytes.

   std::string is among the containers for which polymorphic allocators are available. This allows to **use strings without heap allocations**:

```
void use_strings_without_heap_allocations()
{
  std::pmr::monotonic_buffer_resource string_pool{ memory.data(),
                                                   memory.size() };
  std::pmr::string s1("This is a string", &string_pool);
  std::pmr::string s2("This is another string", &string_pool);
  s1 += s2;
  int p=s1.find("stringThis");
} // #new: 0 #delete: 0 #bytes: 0
```

2. Instead of locally defined memory, **globally defined memory** can also be used. Since the memory for global variables is allocated at the start of the program, one can prevent stack overflow with it:

```
std::array<unsigned char, 100'000> memory; // global definition

void vector_with_global_memory(int n)
{ // The definition of memory shifted out of the function.
  // everything else the same as vector_with_stack_memory
  std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                            memory.size() };
  std::pmr::vector<std::pmr::string> container{ &pool };

  // work with the container
} // frees the memory of pool in memory
```

3. The "monotonic" in the name *monotonic_buffer_resource* comes from the fact that the allocated **memory grows monotonically**: a monotonic_buffer_resource is always released as a whole when it leaves the scope. Deleting individual elements of the container does delete those elements from the container and invokes their destructor. However, the memory of the *monotonic_buffer_resource* is not released (as with *delete*). This has the effect to avoid memory fragmentation.

   Example: If you store objects of the class

```
class NoAllocs// This class does not contain any members
{   // that are using an allocator (see section 1.2.4)
  int i;
public:
  NoAllocs(int n) :i(n){ }
};
```

   in a *pmr::vector* to which *memory* was assigned with a *monotonic_buffer_resource* they are stored in *memory*:

```
void store_NoAllocs(int n)
{ // use the global memory:
  std::pmr::monotonic_buffer_resource pool{
                          memory.data(), memory.size() };
  std::pmr::vector<NoAllocs> container{ &pool };
  // container.reserve(2 * n);
```

```
  for (int i = 0; i < n; ++i)
  {
    container.push_back(NoAllocs(i + 'A'));
  }
} // free the memory of pool in memory
```

With the ascii output of *show_memory* from section 1.1.3 you can see the data in *memory*:

```
&= 0: A...A...B...A...B...C...A...B...C...D...
&=28: A...B...C...D...E...F...A...B...C...D...
&=50: E...F...G...H...I...A...B...C...D...E...
&=78: F...G...H...I...J...K...L...M...A...B...
...
```

Here you can see how the capacity of the vector is increased by a factor of 1.5 each time the previous capacity is exhausted. Since the memory is not released again with a *monotonic_buffer_resource*, the old data remain in the memory.

If you had reserved memory with

```
container.reserve(n);
```

the capacity would be sufficient for n objects from the beginning. Then there would have been no need to copy the data to a new, larger area again and again from the area that had previously become too small:

```
&= 0: A...B...C...D...E...F...G...H...I...J...
&=28: K...L...M...N...O...P...Q...R...S...T...
```

These examples also show in particular that a *pmr::vector* requires more memory than one might expect at first glance for the number of elements to be stored:

– If the capacity of the *vector* is exhausted and new memory is to be allocated from *memory*, the old memory is not freed anymore
  A call like *reserve* may allocate more memory than is specified.

4. Since a *monotonic_buffer_resource* is freed when it leaves the scope, the **memory can be reused**. This can reduce memory requirements compared to *new* and *delete* and simplify and speed up memory management:

   Example: If, after calling *store_NoAllocs* from 3., you call the function

```
void reuse_memory(int n)
{ // use the global memory:
  std::pmr::monotonic_buffer_resource pool{
                          memory.data(), memory.size() };
  std::pmr::vector<NoAllocs> container{ &pool };
  container.reserve(2 * n);
```

```
  for (int i = 0; i < n; ++i)
  {
    container.push_back(NoAllocs(i + 'K'));
  }
} // frees the memory from pool in memory
```

which differs only by 'K' instead of 'A' from *store_NoAllocs*, the values in *memory* are overwritten from the beginning:

```
&= 0: K...L...M...N...O...P...Q...R...S...T...
```

A pool can be shared by multiple containers:

```
std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                          memory.size() };
std::pmr::list<std::pmr::string> lst{ &pool };
std::pmr::vector<int> v1{ &pool };
std::pmr::vector<std::pmr::string> v2{ &pool };
```

5. As already mentioned in 1., a pmr container uses the memory pool as long as there is free space. If the pool runs out of memory, more memory is allocated with *new* and *delete*.

   However, there are applications where the memory fragmentation caused by *new* and *delete* cannot be accepted. Then you can use the additional argument

   ```
   std::pmr::null_memory_resource()
   ```

   in the definition of a *monotonic_buffer_resource*

   ```
   void Throw_an_Exception_when_running_out_of_memory(int n)
   {
     std::pmr::monotonic_buffer_resource pool {memory.data(),
                memory.size(),std::pmr::null_memory_resource()};
     std::pmr::list<std::pmr::string> container{ &pool };
     // work with the container
   }
   ```

   to **throw an exception when memory is** needed but **not available** in the memory pool. This way you can determine if the memory pool is too small.

6. Polymorphic allocators contain a pointer to the *memory_resource* and are therefore also called stateful allocators:

```
template <typename T>
class polymorphic_allocator  // only an excerpt
{
  memory_resource* res;
public:
  polymorphic_allocator() noexcept:res(get_default_resource()){};
  // this is by intention not an explicit constructor:
  polymorphic_allocator(memory_resource* r) :res(r) {}
};
```

One advantage of such a pointer is shown in 7.

The non-polymorphic allocators from before C++17, on the other hand, do not have such a pointer. Since this pointer is included in each object, objects with a polymorphic allocator are slightly larger:

```
sizeof(std::string); // 24
sizeof(std::pmr::string); // 28
```

7. In 1. both the container and the elements were defined with polymorphic allocators:

```
std::pmr::vector<std::pmr::string> container{&pool}; // see 7.
```

Since no memory is specified here for the *pmr::string* elements, the question arises as to which memory the string elements should use. The answer is: **the elements use the same memory as the container** they are contained in. This is achieved with the pointer to the *memory_resource* (see 6.) contained in a polymorphic allocator. This *memory_resource* is passed from the surrounding type (here *pmr::vector*) to a contained type (here *pmr::string*). Internally, this is achieved by using a *std::scoped_allocator_adaptor*.

Therefore in

```
std::pmr::vector<std::pmr::string> container{&pool};
```

both *std::pmr::vector* as well as the *std::pmr::string*-elements use the memory of *pool*.

This pass-through also works recursively with containers that contain containers:

```
std::pmr::vector<std::pmr::list<std::pmr::string>> c{ &pool };
```

Take care not to forget the *pmr::* for the elements. If you use *std::string*, these memory for the string data is allocated with *new*:

```
std::pmr::vector<std::string> c{&pool};  // Strings auf dem heap
```

How to define your own classes that use the allocator of the container if they are elements of a container is shown in section 1.2.4.

1.2.2 Benchmarks: pmr-Containers are often 3-5 times faster

The following benchmarks show that pmr containers are often faster by a factor of 3 to 5 than *std::* containers, where *new* and *delete* are often called. It was measured how long 100,000 *push_back* calls take, where 4, 100 and 1000 bytes are stored:

Benchmarks VS 2019.8, x64, /O2 Release 100.000 operations	*pmr::vector / std::vector*	*pmr::deque / std::deque*	*pmr::list / std::list*
1. push_back int	0,27 / 0,3 ms	0,4 / 6 ms	0,6 / 28 ms
2. push_back 100 Bytes	5 / 20 ms	1 / 30 ms	1,3 / 30 ms
3. push_back 1000 Bytes	70 / 230 ms	10 / 40 ms	10 / 40 ms
4. wie 3., mit *reserve*	12 / 50 ms	–	–

Benchmarks VS 2019.5, x64, /O2 Release 100.000 operations	*pmr::vector / std::vector*	*pmr::deque / std::deque*	*pmr::list / std::list*
1. push_back int	0,3 / 0,3 ms	0,4 / 1 ms	0,7 / 4 ms
2. push_back 100 Bytes	4 / 18 ms	1,5 / 6 ms	2 / 6 ms
3. push_back 1000 Bytes	70 / 200 ms	10 / 20 ms	10 / 15 ms
4. wie 3., mit *reserve*	12 / 40 ms	–	–

Benchmarks gcc 9.2, x64, -O3, 100.000 Operationen	*pmr::vector / std::vector*	*pmr::deque / std::deque*	*pmr::list / std::list*
1. push_back int	0,2 / 0,3 ms	0,2 / 0,4 ms	0,6 /25 ms
2. push_back 100 Bytes	3 / 9 ms	1 / 6 ms	1 / 27 ms
3. push_back 1000 Bytes	50 / 100 ms	9 /35 ms	10 / 35 ms
4. wie 3., mit *reserve*	12 / 40 ms	–	–

John Lakos presents more detailed and refined benchmarks in ("Local ('Arena') Memory Allocators (part 2 of 2)" CppCon 2017: https://www.youtube.com/watch?v=CFzuFNSpycI), where monotone allocators are often also faster by a factor of 10. Jason Turner reports similar improvement factors in C++ Weekly - Ep 222 - 3.5x Faster Standard Containers With PMR!, as does Pablo Halpern in the talk "Getting Allocators out of Our Way".

1.2.3 A first Summary: pmr Container for Embedded Applications

The previous tutorial has shown that pmr containers can be created without using the heap and thus prevent memory fragmentation. This satisfies requirements a), b) and e) of **AUTOSAR Rule A18-5-5**:

(a) deterministic behavior resulting with the existence of worst-case execution time,
(b) avoiding memory fragmentation,
(e) no dependence on non-deterministic calls to kernel.

Since the pmr containers are part of the standard library, it can be assumed that

(d) avoiding mismatched allocations or deallocations

is satisfied. In many applications, you can also provide enough memory and guarantee an upper limit for the memory requirements, so that also also

(c) avoid running out of memory

is satisfied. Then all requirements of AUTOSAR Rule A18-5-5 and many other coding rules (e.g. Misra, JSF, etc.) are met.

But that is not all: With the pmr containers, many **algorithms** of the standard library are also available. They are often more sophisticated and efficient than hand crafted algorithms.

This means that, for the first time in the history of C++, the **containers and algorithms** of the C++ standard library can be used in applications that require AUTOSAR Rule A18-5-5.

This is a big step forward compared to the **time before C++17** and pmr: Since you could not use the containers, the algorithms were often not available either. This functionality then had to be produced by custom algorithms and data structures, causing a considerable development and testing effort. It is also often not easy to achieve the efficiency of the standard library with hand crafted containers and algorithms.

1.2.4 pmr-aware custom types ϴ

Next, we will show what you have to consider when defining custom classes so that objects of these classes use the polymorphic allocator when placed in a pmr container.

For the data members of a class that don't use allocators, it's easy: here you don't have to consider anything at all. All the storage for these data members of an object is managed by the allocator of the container.

Example: The class *NoAllocs* contains only an *int* member:

```
class NoAllocs
{
  int i;
public:
  NoAllocs(int n) :i('A' + n){ }
};
```

Since *int* does not use an allocator, all memory for an object of this class is managed by the allocator of the pmr container. If you put objects of this class in a pmr container that has been allocated with a *monotonic_buffer_resource*,

```
std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                          memory.size() };
std::pmr::vector<NoAllocs> container{&pool};
```

all objects are stored in the memory pool. This has already been shown in the example of section 1.2.1, 3.:

```
&= 0: A...B...C...D...E...F...G...H...I...J...
&=28: K...L...M...N...O...P...Q...R...S...T...
```

On the other hand, if data members of a class use allocators, the memory for those data members is managed by those allocators. This is even true if the allocators are not pmr allocators and the objects of the class are stored in a pmr container.

Example: If you pass objects of the class

```
class StdStringAlloc
{// The allocator of std::string uses new and delete.
  int i;
  std::string str; // The characters of the string are
public:            // stored on the heap
  StdStringAlloc(int n, std::string_view sv):i{n}, str{sv}
  { // string_view, not string, so that no string-
    str = char(n) + str;           // constructor calling
  }                                // new is called
};
```

to a pmr container that uses a *monotonic_buffer_resource*,

```
std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                          memory.size() };
std::pmr::vector<StdStringAlloc> container{&pool};
```

only the 28 bytes (*sizeof(std::string)* = 24 + *sizeof(int)* = 4) of these objects are stored in the memory pool. The characters belonging to the strings are stored in the heap allocated by *new* and not in the memory pool. After the statements

```
const std::string non_sso_str ="-non-SSO-Str ####";//17,>16
for (int i = 0; i < n; ++i)
  container.push_back(StdStringAlloc(i+'A', non_sso_str));
```

you don't see anywhere in *memory* the characters of the stored string (with the ascii part of *show_memory* from section 1.1.3):

```
&= 0 asc: A...x.....................B...(.......
&=28 asc: ..............C.....................
&=50 asc: ....D.........................E.......
&=78 asc: ..................F...H..............
&=a0 asc: ........G.........................H...
```

If strings with 16 or less characters would have been stored,

```
const std::string sso_str = "-SSO-String 16"; // 14, <= 16
for (int i = 0; i < n; ++i)
  container.push_back(StdStringAlloc(i + 'A', sso_str));
```

these would have been stored inside the string because of the small string optimization (SSO) and therefore would be stored in *memory*:

```
&= 0 asc: A...A-SSO-String 16.........B...B-SSO-St
&=28 asc: ring 16.........C...C-SSO-String 16.....
&=50 asc: ....D...D-SSO-String 16.........E...E-SS
&=78 asc: O-String 16.........F...F-SSO-String 16.
```

To make objects of custom classes manage all data with the polymorphic allocators of the container when stored in a pmr container, the simplest way is to use

1. pmr types (e.g. *std::pmr::string*) for all data members that have an allocator.

However, this is not enough. The following definitions are also necessary:

2. The public member *allocator_type*:

   ```
   using allocator_type = std::pmr::polymorphic_allocator<char>;
   ```

   Here exactly the name *allocator_type* must be used. However, the type specified in angle brackets is meaningless and can be replaced by any other type.

3. Initializing constructors called without an allocator argument should use an *allocator_type*-object for initializing the pmr elements. However, this is not necessary for the copy and move constructors if they are implemented.

4. For all constructors (including the copy and move constructors) overloaded variants that have an *allocator_type* parameter as an additional last parameter and use it for the pmr elements. The copy and move constructors should not be *noexcept*.

Example: If you store objects of the class

```
class PmrType
{
  int i; // an member without an allocator
  std::pmr::string pmr_str; // 1. not std::string
public:
  using allocator_type =
              std::pmr::polymorphic_allocator<char>; // 2.

  // initializing constructor(s):
  PmrType(int n, std::string_view sv,
                    allocator_type alloc = {}) : // 3.
                                  i{ n }, pmr_str{ sv }
  {
    pmr_str = char(i) + pmr_str;
  }
```

```
  // 4. copy/move with allocators:
  PmrType(const PmrType& c, allocator_type alloc)
    : pmr_str{ c.pmr_str, alloc }, i{ c.i }
  {}

  PmrType(PmrType&& c, allocator_type alloc)
    : pmr_str{ std::move(c.pmr_str), alloc }, i{ c.i }
  {}
};
```

in a pmr container,

```
std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                          memory.size() };
std::pmr::vector<PmrType> container{ &pool };
container.reserve(n);
const std::string non_sso_str ="-non-SSO-Str ####";//17,>16
for (int i = 0; i < n; ++i)
  container.push_back(PmrType(i + 'A', non_sso_str));
```

you can see all data in the memory of *memory*:

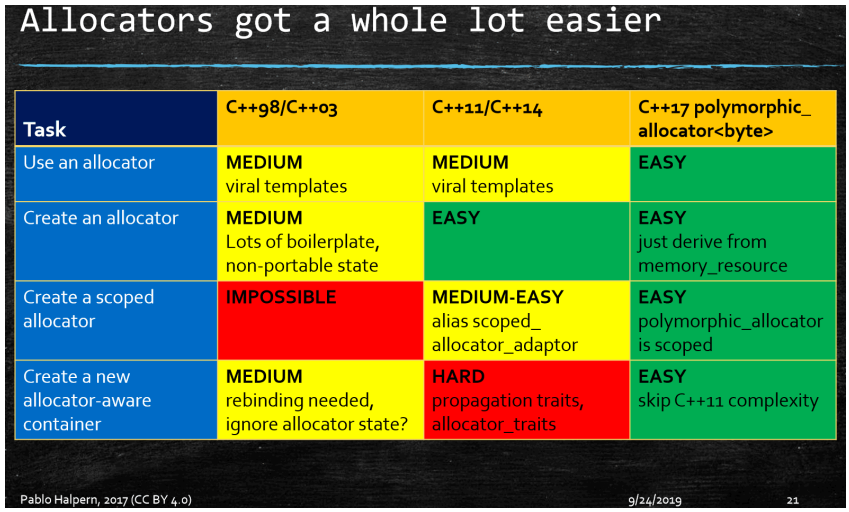
```
&= 0 asc: A...`./..s..------------........B...`./.
&=28 asc: .t..------------........C...`./..t..----
&=50 asc: --------........A-non-SSO-Str ####.-----
&=78 asc: --------B-non-SSO-Str ####.-------------
&=a0 asc: C-non-SSO-Str ####.-------------A...`./.
```

If you want to set a *pmr::string* in a class via a member function, you have to be careful not to forget the *pmr::*. You can also use a parameter of type *string_view*. However, you should avoid *std::string* or *std::string&*, since *std::string&* can also lead to the call of a *std::string* constructor and thus to a call of *new* if, for example, a *char** argument is passed.

1.3 Allocators in C++17

Before C++17, writing custom allocators was difficult and error-prone. In C++17 this has been greatly simplified. The pmr allocators are all derived from the abstract base class *memory_resource*. It consists of only a few member functions. If you want to develop your own allocators, you "only" have to derive your own class in which the functions *do_allocate*, *do_deallocate* and *do_is_equal* are overridden.

Pablo Halpern, one of the leading authors of pmr, illustrated this with this graphic at his talk at CppCon 2017 (https://www.youtube.com/watch?v=v3dz-AKOVL8, https://github.com/phalpern):



1.4 What is polymorphic with pmr-Allocators? ϴ

A *polymorphic_allocator* uses virtual functions for the allocation of memory. In the examples so far, this feature has not been used explicitly. In the following, it is briefly shown what advantages polymorphic allocators

```
std::vector<T, polymorphic_allocator<T>>; // C++17
```

have, compared to the non-polymorphic ones from the time before C++17:

```
std::vector<T, typename Allocator = std::allocator<T>>; // vor C++17
```

First, let's start with an example to show the disadvantage of non-polymorphic allocators.

Since the type of a type argument affects the type of a generic type, generic types with different type arguments are different. Therefore, they cannot be assigned to each other, or used as arguments to a parameter in a function.

Example: Since the two classes

```
template <typename T>
class memory_resource_1{ };
```

```
template <typename T>
class memory_resource_2{ };
```

are two different types, with

```
template <typename T, typename MyAllokator>
class MyContainer{};
```

the two containers

```
MyContainer<int, memory_resource_1<int>> v1;
MyContainer<int, memory_resource_2<int>> v2;
```

have different types. That is why the assignment:

```
v1 = v2; // error - the types of v1 and v2 are different
```

is not allowed, nor is passing an argument of the type of v1 to a parameter of the type of v2.

To make such assignments possible, the allocators are derived from an abstract base class

```
template <typename T>
class memory_resource {
public:
  virtual void allocate(T x) = 0;
};

template <typename T>
class memory_resource_1 :public memory_resource<T>
{
  void allocate(T x) override {};
};

template <typename T>
class memory_resource_2 :public memory_resource<T>
{
  void allocate(T x) override {};
};
```

Then they are passed to the container via the constructor:

```
template <typename T, typename MyAllocator=memory_resource<T> >
class MyContainer
{
  MyAllokator* a_;
public:
  MyContainer(MyAllocator* a): a_(a){};
  void push_back(const T& x)
  {
    a_->allocate(x) ;
  }
};
```

This allows assignments of containers with different allocators:

```
memory_resource_1<int> m1;
memory_resource_2<int> m2;
MyContainer<int> v1(&m1);
MyContainer<int> v2(&m2);
v1 = v2; // the types of v1 and v2 are the same: MyContainer<int>
```

To a function like

```
void f(MyContainer<int>& t) {}
```

v1 and v2 can be passed as arguments:

```
f(v1); f(v2);
```

Since the call to *allocate* in *push_back* is virtual, this results in a call to *allocate* from memory_resource_1 in v1 and a call to allocate from *memory_resource_2* in v2.

It is also possible to change the allocator at runtime.

Even if you rarely make use of this compatibility: It is often used in the standard C++17 library, contributing to ease of use. In C++11 this was not possible.

1.5 Literatur

Pablo Halpern: Allocators: the good parts (CppCon 2017): https://youtu.be/v3dz-AKOVL8

Pablo Halpern: „Getting Allocators out of Our Way“ https://www.youtube.com/watch?v=RLezJuqNcEQ

Nico Josuttis: C++17 - The Complete Guide http://cppstd17.com/

Jason Turner “C++ Weekly - Ep 222 - 3.5x Faster Standard Containers With PMR!” https://www.youtube.com/watch?v=q6A7cKFXjY0

Jason Turner Ep 235: https://www.youtube.com/watch?v=vXJ1dwJ9QkI&feature=youtu.be

Jason Turner Ep 236: https://www.youtube.com/watch?v=2LAsqp7UrNs Construction Allocator-aware types

https://en.cppreference.com

Pablo Halpern: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1850.pdf

Pablo Halpern: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2554.pdf

Pablo Halpern 2013: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf

Pablo Halpern: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3916.pdf

https://stackoverflow.com/questions/22148258/what-is-the-purpose-of-stdscoped-allocator-adaptor von Jonathan Wakely

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2126r0.pdf

Pablo Halpern, Dietmar Kühl: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0339r5.pdf

Arthur O'Dwyer “An Allocator is a Handle to a Heap” (CppCon 2018): https://trshow.info/watch/IejdKidUwIg/cppcon-2018-arthur-o-dwyer-an-allocator-is-a-handle-to-a-heap.html