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

Polymorphic Memory Resources (pmr) und STL Container für Embedded Anwendungen

Vortrag am Di. 18.5.2021 auf der Advanced C++ Developers Conference, https://adcpp.de/21/agenda. Sie können dieses Manuskript herunterladen von

https://www.rkaiser.de/downloads/

Für viele embedded C++-Anwendungen wird die Einhaltung der AUTOSAR- oder Misra-Regeln verlangt. Die aktuelle Version sind die

„Guidelines for the use of the C++14 language in critical and safety-related systems“ (https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf),

die die MISRA Rules für C++ aus dem Jahr 2008 auf C++14 aktualisieren. Die nächste Version dieser Regeln soll wieder von MISRA kommen und C++20 abdecken.

Zu den Autosar C++14 Regeln gehört die **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.

Diese Regel hat weitreichende Konsequenzen, da die Container der Standardbibliothek in der Voreinstellung den Speicher für ihre Elemente mit *new* allokieren und mit *delete* wieder freigeben. Diese Aufrufe

- haben kein deterministisches Zeitverhalten
- können zu einer Speicherfragmentierung führen.

Da *new* und *delete* die Regel A18-5-5 verletzen, dürfen die STL-Container in vielen embedded Anwendungen nicht eingesetzt werden.

Mit den seit C++17 im Namensbereich *std::pmr* (polymorphic memory resources) verfügbaren Allokatoren können diese Einschränkungen oft vermieden werden. Damit stehen zum ersten Mal in der Geschichte von C++ die Container und Algorithmen der Standardbibliothek auch für viele eingebettete Anwendungen zur Verfügung.

1.1 Einleitung

1.1.1 Über mich

Ich bin seit über 30 Jahren freiberuflicher C++ Trainer, Software Entwickler und Autor von mehreren C++-Büchern. Bis 2017 war ich Professor an der Dualen Hochschule Baden-Württemberg. Mehr finden Sie auf meiner Internetseite

https://www.rkaiser.de/

1.1.2 Embedded Anwendungen

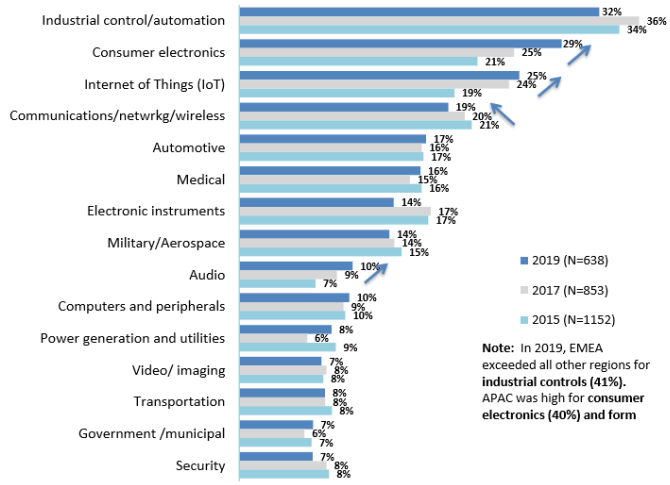
Obwohl der Begriff "embedded Anwendung" den Eindruck einer klaren und eindeutigen Abgrenzung erweckt, ist das keineswegs der Fall. Wenn man 5 Leute fragt, was dieser Begriff für sie bedeutet, bekommt man oft 10 verschiedene Antworten.

Das liegt daran, dass embedded Anwendungen einen sehr großen Einsatzbereich abdecken und auf sehr unterschiedlichen Hardware-Plattformen laufen. Oft sind das Steuerungen, die lange und zuverlässig laufen müssen.

Die folgenden Übersichten sollen diese Vielfalt illustrieren:

7

For what types of applications are your embedded projects developed?

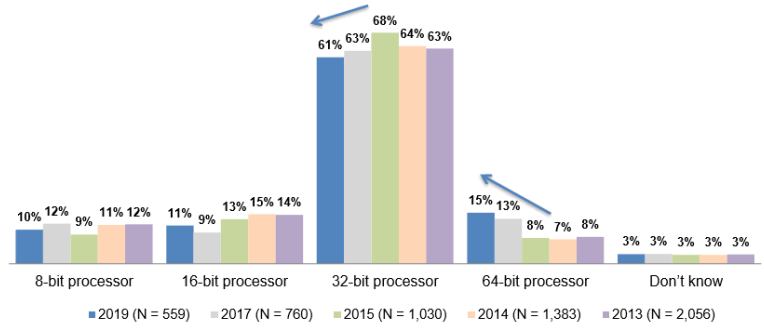


Note: In 2019, EMEA exceeded all other regions for **industrial controls (41%).** APAC was high for **consumer electronics (40%) and form**

EE Times embedded 2019 Embedded Markets Study

66

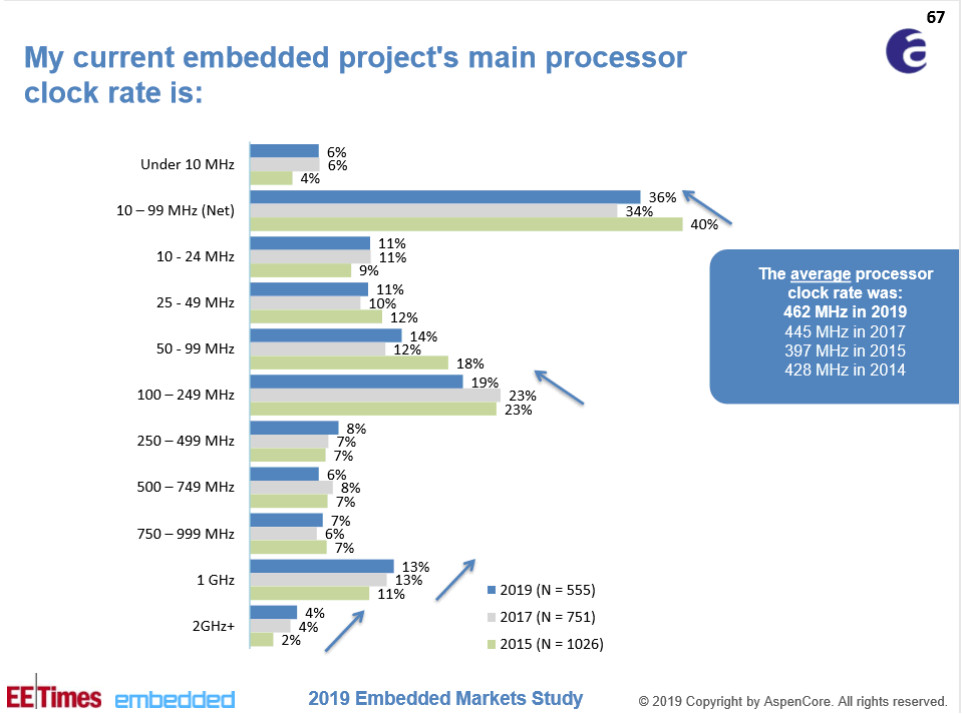
My current embedded project's main processor is a:



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%

EE Times embedded 2019 Embedded Markets Study



1.1.3 *new* und *delete* tracken

Bevor wir in Abschnitt 1.2 die pmr-Allokatoren vorstellen, wird zunächst gezeigt, wie man mit überladenen *new*- und *delete*-Operatoren die Aufrufe von *new*- und *delete* überwachen kann.

Dazu werden globale Variable definiert, die jeden Aufruf von *new*- und *delete* mitzählen

```
int new_counter = 0;       // Anzahl der new Aufrufe
int delete_counter = 0;    // Anzahl der delete Aufrufe
size_t allocated_mem = 0; // Allokierter Speicher in Bytes

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

und die z.B. folgendermaßen angezeigt werden:

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

Definiert man überladene *new*- und *delete*-Operatoren, werden diese bei jedem Aufruf von *new* und *delete* aufgerufen. Die überladenen Operatoren müssen diese Signatur haben:

```
void* operator new(std::size_t sz)
{ // Dieser Operator soll wie der vordefinierte new-Operator nur
  // malloc aufrufen und gegebenenfalls eine Exception auslösen
  void* ptr = std::malloc(sz);
  if (ptr)
  {
    new_counter++; // Die einzigen Unterschiede zum vordefinierten
    allocated_mem += sz;                          // new-Operator
    return ptr;
  }
  else throw std::bad_alloc{};
}

void operator delete(void* ptr) noexcept
{
  delete_counter++;
  std::free(ptr); // Das macht der vordefinierte delete-Operator
}
```

Selbstverständlich kann man hier nicht nur die Anzahl der Aufrufe protokollieren.

Beispiel: Nach dem Aufruf

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

  delete pi;
}
```

erhält man mit *new_delete_summary*

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

Bei STL-Containern werden *new* und *delete* implizit aufgerufen. Nach dem Aufruf

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

erhält man

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

Den Inhalt von Speicherbereichen werden wir gelegentlich mit einer Funktion wie *show_memory* anschauen:

```
void show_memory(unsigned char* buffer, std::size_t buffer_size,
                                          const char* headline = "")
{ // Zeigt die buffer_size Bytes ab der Adresse buffer an
  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 Allokatoren und polymorphic memory resources

Die wichtigsten Besonderheiten der polymorphen Allokatoren werden im Folgenden an Beispielen wie diesem gezeigt.

```
void vector_with_heap_memory(int n)
{
  std::vector<std::string> container;
  // arbeite mit dem container
  for (int i = 0; i < n; ++i)
  { // Hier ist keine small string optimization (SSO) erwünscht:
    container.push_back("Ein string mit mehr als 16 Zeichen");
  }
} // mit n=10: #new: 17 #delete: 17 #bytes: 1392
```

Dabei werden Strings mit mehr als 16 Zeichen im *vector* abgelegt, damit die Zeichen auf dem Heap gespeichert werden. In Visual C++ werden bei kleineren Strings die Zeichen im String und damit auf dem Stack abgelegt ("small string optimization" - SSO).

Diese Beispiele verwenden

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

anstelle von *std::vector*. Obwohl das auf den ersten Blick wie eine eigene *vector*-Klasse im Namensbereich *pmr* aussieht, ist es nichts anderes als die Klasse *std::vector* mit einem speziellen Allokator:

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

Die Container der Standardbibliothek (außer *std::array*) sind so konstruiert, dass die Funktionen zur Anforderung und Freigabe von Speicher in einem Allokator gekapselt sind, der als Template-Parameter übergeben wird. In der Voreinstellung ist das *std::allocator<T>*, der Speicher mit *new* anfordert und mit *delete* wieder frei gibt:

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

Wie für *vector* gibt es auch für alle anderen sequenziellen und assoziativen Container pmr-Varianten. Die Containerklassen aus *std::pmr* unterscheiden sich von denen aus dem Namensbereich *std* nur dadurch, dass anstelle von *std::allocator* der Allokator *polymorphic_allocator* verwendet wird.

Die pmr-Klassen stehen erst seit C++17 zur Verfügung und werden bisher nur selten eingesetzt. Deswegen werden gründliche Tests empfohlen.

1.2.1 Container mit einer *monotonic_buffer_resource*

Die folgenden Beispiele sind nicht auf *std::vector* beschränkt, sondern lassen sich auf beliebige sequentielle und assoziative Container außer *std::array* übertragen.

Mit der nach

```
#include <memory_resource>
```

verfügbaren Klasse *std::pmr::monotonic_buffer_resource* kann man einem Container Speicher zuordnen, der dann vom Container anstelle von Speicher auf dem Heap verwendet wird. Dadurch muss bei der Arbeit mit dem Container **kein *new* und *delete*** aufgerufen werden.

1. Dieser **Speicher** wird im Konstruktor

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

   mit seiner Adresse und Größe angegeben. Das kann z.B. ein Array sein, das auf dem Stack angelegt wurde:

   ```
   void vector_with_stack_memory(int n)
   {  // Der einzige Unterschied sind die ersten drei Anweisungen:
     std::array<unsigned char, 100'000> memory; // lokale Definition
     // verwende memory als Speicher für den vector und die strings:
     std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                               memory.size() };
     std::pmr::vector<std::pmr::string> container{&pool}; // siehe 7.

     // arbeite mit dem container
     for (int i = 0; i < n; ++i)
     {
       container.push_back("Ein string mit mehr als 16 Zeichen");
     }
   }  // #new: 0 #delete: 0 #bytes: 0
   ```

   Falls der Speicher von *memory* für alle Operationen mit dem Container ausreicht, findet kein einziges *new* und *delete* mehr statt. Falls er aber nicht ausreicht, wird weiterer Speicher von einer als *upstream* bezeichneten *memory_resource* angefordert. Siehe dazu 5.

   Stroustrup berichtet in „A tour of C++", Kap. 13.6 von einer Anwendung, bei der die Umstellung auf pmr-Container den Speicherbedarf von 6 Gigabyte auf 300 Megabyte reduziert hat. Dabei wurden im Wesentlichen nur wie oben die ersten drei Zeilen geändert.

   Zu den Containern, für die polymorphe Allokatoren verfügbar sind, gehört auch *std::string*. Damit kann ***std::string*** und seine Elementfunktionen auch in Anwendungen verwendet werden, in denen kein *new* und *delete* zulässig ist:

```
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("hi");
} // #new: 0 #delete: 0 #bytes: 0
```

2. Anstelle von lokal definiertem kann auch **global definierter Speicher** verwendet werden. Da der Speicher für globale Variable beim Start des Programms reserviert wird, kann der Speicherbedarf für das Array beim Aufruf der Funktion nicht zu einem Stack overflow führen:

```
std::array<unsigned char, 100'000> memory; // globale Definition

void vector_with_global_memory(int n)
{ // Die Definition von memory nach außen verschoben.
  // Alles andere ist gleich wie bei vector_with_stack_memory
  std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                            memory.size() };
  std::pmr::vector<std::pmr::string> container{ &pool };

  // arbeite mit dem container
} // gibt den Speicher von pool in memory frei
```

3. Das „monotonic“ im Namen von *monotonic_buffer_resource* kommt daher, dass der reservierte **Speicher monoton wächst**: Eine *monotonic_buffer_resource* wird immer beim Verlassen ihres Gültigkeitsbereichs als Ganzes freigegeben. Eine Löschung von einzelnen Elementen des Containers löscht diese Elemente zwar aus dem Container und ruft ihren Destruktor auf. Der Speicher der *monotonic_buffer_resource* wird dadurch aber nicht (wie mit *delete*) freigegeben. Dadurch wird eine Fragmentierung des Speichers vermieden.

   Beispiel: Legt man Objekte der Klasse

```
class NoAllocs // Diese Klasse enthält kein Element, das
{   // einen Allokator verwendet (siehe Abschnitt 1.2.4)
  int i;
public:
  NoAllocs(int n) :i(n){ }
};
```

   in einem *pmr::vector* ab, dem mit einer *monotonic_buffer_resource* der Speicher *memory* zugeordnet wurde, werden diese in *memory* abgelegt:

```
void store_NoAllocs(int n)
{ // verwende das globale 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'));
  }
} // Hier wird der Speicher von pool in memory freigegeben
```

Mit der Ascii-Ausgabe von *show_memory* von Abschnitt 1.1.3 sieht man die Daten 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...
...
```

Hier sieht man, wie die Kapazität des *vector* jedes Mal um den Faktor 1,5 vergrößert wird, wenn die bisherige Kapazität nicht mehr ausreicht. Da der Speicher bei einer *monotonic_buffer_resource* nicht wieder freigegeben wird, bleiben die alten Daten im Speicher erhalten.

Hätte man Speicher mit

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

reserviert, wäre die Kapazität von Anfang an für n Objekte ausreichend. Dann müssten nicht immer wieder die Daten aus dem vorher zu klein gewordenen Bereich in einen neuen, größeren Bereich kopiert werden:

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

Diese Beispiele zeigen insbesondere auch, dass ein *pmr::vector* mehr Speicher benötigt als man für die zu speichernde Anzahl von Elementen auf den ersten Blick vielleicht erwartet:

- Falls die Kapazität des *vector* erschöpft ist und neuer Speicher von *memory* angefordert wird, wird der alte Speicher nicht mehr freigegeben
- Ein Aufruf wie *reserve* kann z.B. mehr Speicher anfordern als angegeben wird.

4. Da eine *monotonic_buffer_resource* beim Verlassen des Gültigkeitsbereichs freigegeben wird, kann der **Speicher wiederverwendet** werden. Das kann den Speicherbedarf im Vergleich zu *new* und *delete* reduzieren und die Speicherverwaltung vereinfachen und beschleunigen:

   Beispiel: Ruft man nach *store_NoAllocs* aus 3. die Funktion

```
void reuse_memory(int n)
{ // verwende das globale 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'));
    }
  } // gibt den Speicher von pool in memory frei
```

auf, die sich nur durch 'K' anstelle von 'A' von *store_NoAllocs* unterscheidet, werden die Werte in *memory* ab dem Anfang überschrieben:

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

Ein pool kann auch von mehreren Containern gemeinsam verwendet werden:

```
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. Wie schon in 1. erwähnt wurde, verwendet ein pmr-Container den memory pool, solange dieser noch freien Platz hat. Falls dieser aber erschöpft ist, wird weiterer Speicher von einer als *upstream* bezeichneten *memory_resource* angefordert, die als letztes Argument im Konstruktor angegeben werden kann.

   Gibt man keine solche *memory_resource* an, wird die sogenannte default resource verwendet (die man auch mit *get_default_resource* erhält) und die Speicher mit *new* auf dem heap anfordert. Der Grund für diese Vorgehensweise ist, dass pmr-Container und ihre Allokatoren vor allem mit dem Ziel einer hohen Geschwindigkeit entworfen wurden und nicht mit dem Ziel, *new* und *delete* zu vermeiden. Der mit *new* angeforderte Speicher kann ein großes dynamisches Array sein, das von der *monotonic_buffer_resource* wie das globale oder lokale Array unter 1. und 2. verwaltet wird.

   Für embedded Anwendungen, die *new* und *delete* vermeiden sollen, kann man als Argument für *upstream* eine

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

   bei der Definition einer *monotonic_buffer_resource*

   ```
   void Exception_auslösen_falls_Speicher_nicht_reicht(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 };
     // arbeite mit dem container
   }
   ```

verwenden. Damit wird eine **Exception** ausgelöst, **wenn Speicher** angefordert wird, aber im memory pool **keiner mehr verfügbar** ist. So kann man durch Tests feststellen, ob der memory pool zu klein ist.

6. Polymorphe Allokatoren enthalten einen Zeiger auf die *memory_resource* und werden deshalb auch als Allokatoren mit einem Zustand (**stateful allocators**) bezeichnet:

```
template <typename T>
class polymorphic_allocator  // nur ein Auszug
{
  memory_resource* res;
public:
  polymorphic_allocator() noexcept:res(get_default_resource()){};
  // das ist absichtlich kein explicit Konstruktor:
  polymorphic_allocator(memory_resource* r) :res(r) {}
};
```

Ein Vorteil eines solchen Zeigers wird in 7. gezeigt.

Die nicht polymorphen Allokatoren aus der Zeit vor C++17 haben dagegen keinen solchen Zeiger. Da dieser Zeiger in jedes Objekt aufgenommen wird, sind Objekte mit einem polymorphen Allokator etwas größer:

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

7. In 1. wurden sowohl der Container als auch die Elemente mit polymorphen Allokatoren definiert:

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

Da hier für die *pmr::string*-Elemente kein Speicher angegeben ist, stellt sich die Frage, welchen Speicher diese verwenden sollen. Die Antwort ist: Die **Elemente verwenden denselben Speicher wie der Container**, in dem sie enthalten sind. Das wird mit dem Zeiger auf die *memory_resource* erreicht, der in einem polymorphen Allokator enthalten ist (siehe 6.). Diese *memory_resource* wird vom umgebenden Typ (hier *pmr::vector*) an einen enthaltenen Typ (hier *pmr::string*) weitergegeben. Intern wird das mit Hilfe eines *std::scoped_allocator_adaptor* erreicht.

Deshalb verwenden in

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

sowohl *std::pmr::vector* als auch die *std::pmr::string*-Elemente den Speicher von *pool*.

Diese Weitergabe funktioniert auch rekursiv mit Containern, die Container enthalten:

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

Hier muss man aber wirklich darauf achten, dass man das *pmr::* bei den Elementen nicht vergisst. Verwendet man *std::string*, werden die Daten für die Strings mit *new* angelegt:

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

Wie man eigene Klassen definieren kann, die als Elemente eines Containers den Allokator des Containers übernehmen, wird in Abschnitt 1.2.4 gezeigt.

1.2.2 Benchmarks: pmr-Container sind oft 3-5 mal schneller

Die folgenden Benchmarks zeigen, dass pmr-Container oft um den Faktor 3 bis 5 schneller sind als gewöhnliche Container, bei denen oft *new* und *delete* aufgerufen wird. Dabei wurde gemessen, wie lange 100.000 *push_back*-Aufrufe dauern, bei denen 4, 100 und 1000 Bytes abgelegt werden:

Benchmarks VS 2019.8, x64, /O2 Release 100.000 Operationen	*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 / 17 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 Operationen	*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 11.1, x64, -O3, 100.000 Operationen	*pmr::vector / std::vector*	*pmr::deque / std::deque*	*pmr::list / std::list*
1. push_back int	0,2 / 0,5 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	80 / 100 ms	9 /35 ms	10 / 35 ms
4. wie 3., mit *reserve*	8 / 40 ms	–	–

John Lakos präsentiert in ("Local ('Arena') Memory Allocators (part 2 of 2)" CppCon 2017: https://www.youtube.com/watch?v=CFzuFNSpycI) ausführlichere und differenziertere Benchmarks, bei denen monotone Allokatoren oft auch um den Faktor 10 schneller sind. Jason Turner berichtet in C++ Weekly - Ep 222 - 3.5x Faster Standard Containers With PMR! von ähnlichen Verbesserungsfaktoren, ebenso Pablo Halpern in dem Vortrag „Getting Allocators out of Our Way"

1.2.3 Eine erste Zusammenfassung: pmr-Container bei Embedded Anwendungen

Die bisherigen Ausführungen haben gezeigt, dass pmr-Container ohne heap angelegt werden können und so eine Speicherfragmentierung verhindern. Damit werden die Anforderungen a), b) und e) der **AUTOSAR Rule A18-5-5** erfüllt:

(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.

Da die pmr-Container zur Standardbibliothek gehören, kann man davon ausgehen, dass

(d) avoiding mismatched allocations or deallocations

erfüllt wird. In vielen Anwendungen kann man außerdem noch genügend Speicher zur Verfügung stellen und eine Obergrenze für den Speicherbedarf garantieren, damit auch noch

(c) avoid running out of memory

sichergestellt ist. Dann werden alle Anforderungen der AUTOSAR Rule A18-5-5 und vieler anderer Kodierregeln (z.B. Misra, JSF usw.) erfüllt.

Das ist aber noch nicht alles: Mit den pmr-Containern stehen außerdem auch viele **Algorithmen** der Standardbibliothek zur Verfügung. Diese sind oft ausgefeilter und effizienter als selbstgeschriebene Algorithmen.

Damit können die **Container und Algorithmen der Standardbibliothek** von C++ das erste Mal in der Geschichte von C++ in Anwendungen eingesetzt werden, in denen die AUTOSAR Rule A18-5-5 verlangt wird.

Das ist ein großer Fortschritt gegenüber der **Zeit vor C++17 und pmr**: Da man die Container nicht einsetzen konnte, standen auch die Algorithmen oft nicht zur Verfügung. Diese Funktionalität musste dann durch eigene Algorithmen und Datenstrukturen hergestellt werden, was mit einem beträchtlichen Entwicklungs- und Testaufwand verbunden ist. Es ist oft auch nicht einfach, mit eigenen Containern und Algorithmen die Effizienz der Standardbibliothek zu erreichen.

1.2.4 Eigene Klassen pmr-fähig machen ϴ

Als Nächstes wird gezeigt, was man bei der Definition von selbstdefinierten Klassen beachten muss, damit Objekte dieser Klassen den polymorphen Allokator verwenden, wenn sie in einem pmr-Container abgelegt werden.

Bei den Datenelementen einer Klasse, die keine Allokatoren verwenden, ist das einfach: Hier muss man überhaupt nichts beachten. Der gesamte Speicherplatz für diese Datenelemente eines Objekts wird vom Allokator des Containers verwaltet.

Beispiel: Die Klasse *NoAllocs* enthält nur ein *int*-Element:

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

Da *int* keinen Allokator verwendet, wird der gesamte Speicherplatz für ein Objekt dieser Klasse vom Allokator des pmr-Containers verwaltet. Legt man Objekte dieser Klasse in einem pmr-Container ab, dem eine *monotonic_buffer_resource* zugeordnet wurde,

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

werden alle Objekte vollständig im memory pool abgelegt. Das wurde auch schon im Beispiel von Abschnitt 1.2.1, 3. gezeigt:

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

Wenn Datenelemente einer Klasse dagegen Allokatoren verwenden, wird der Speicher für diese Datenelemente von diesen Allokatoren verwaltet. Das gilt auch dann, wenn die Allokatoren keine pmr-Allokatoren sind und die Objekte der Klasse in einem pmr-Container abgelegt werden.

Beispiel: Legt man Objekte der Klasse

```
class StdStringAlloc
{// Der Allokator von std::string verwendet new und delete.
  int i;
  std::string str; // Die Zeichen des Strings werden auf
public:            // dem Heap abgelegt.
  StdStringAlloc(int n, std::string_view sv):i{n}, str{sv}
  { // string_view, nicht string, damit hier kein string-
    str = char(n) + str;           // Konstruktor mit
  }                                // new aufgerufen wird
};
```

in einem pmr-Container ab, dem eine *monotonic_buffer_resource* zugeordnet wurde,

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

werden nur die 28 Bytes (*sizeof(std::string)* = 24 + *sizeof(int)* = 4) dieser Objekte im memory pool abgelegt. Die zu den Strings gehörenden Zeichen werden in dem von *new* zugeordneten Heap gespeichert und nicht im memory pool. Nach den Anweisungen

```
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));
```

sieht man im Speicher von *memory* nirgendwo die Zeichen des abgelegten Strings (mit dem ascii-Teil von *show_memory* von Abschnitt 1.1.3):

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

Hätte man Strings mit 16 oder weniger Bytes abgelegt,

```
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));
```

würden diese wegen der small string optimization (SSO) in den String aufgenommen und in *memory* abgelegt:

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

Damit Objekte selbstdefinierter Klassen beim Speichern in einem pmr-Container alle Daten mit den polymorphen Allokatoren des Containers verwalten, verwendet man am einfachsten

1. für alle Datenelemente, die einen Allokator haben, pmr-Datentypen (z.B. *std::pmr::string*).

Das reicht aber nicht aus. Es sind außerdem noch die folgenden Definitionen notwendig:

2. Das *public* Element *allocator_type*:

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

   Hier muss genau der Name *allocator_type* verwendet werden. Der in spitzen Klammern angegebene Datentyp ist aber ohne Bedeutung und kann durch jeden anderen Typ ersetzt werden.

3. Initialisierende Konstruktoren, die ohne Allokator-Argument aufgerufen werden, sollen ein *allocator_type*-Objekt für die Initialisierung der pmr-Elemente verwenden. Das ist aber für die copy- und move Konstruktoren nicht notwendig, falls sie implementiert werden.

4. Für alle Konstruktoren (auch die copy- und move-Konstruktoren) überladene Varianten, die einen *allocator_type*-Parameter als zusätzlichen letzten Parameter haben und diesen für die pmr-Elemente verwenden. Die copy- und move-Konstruktoren sollten kein *noexcept* haben.

Beispiel: Speichert man Objekte der Klasse

```
class PmrType
{
  int i; // ein Element ohne Allokator
  std::pmr::string pmr_str; // 1. nicht std::string
public:
  using allocator_type =
              std::pmr::polymorphic_allocator<char>; // 2.

  // initialisierender Konstruktor:
  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 mit Allokatoren:
  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 einem 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));
```

sieht man im Speicher von *memory* alle Daten:

```
&= 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...`./.
```

Falls man einen *pmr::string* in einer Klasse über eine Elementfunktion will, muss man darauf achten, dass man das *pmr::* nicht vergisst. Man kann auch einen Parameter des Typs *string_view* verwenden. *std::string* oder *std::string&* sollte man aber vermeiden, da auch *std::string&* zum Aufruf eines *std::string*-Konstruktors und damit zum Aufruf von new führen kann, wenn z.B. ein *char**-Argument übergeben wird.

1.3 Memory Resourcen Θ

Für das zweite Argument in einer pmr-Containerklasse

```
std::vector<T, polymorphic_allocator<T>>;
```

kann man eine *memory_resource* übergeben, da der Konstruktor eine solche Resource implizit konvertiert:

```
template<class Tp = std::byte> // nur ein Auszug
class polymorphic_allocator {  // mit einem Konstruktor
public:
  polymorphic_allocator(std::pmr::memory_resource* r);
};
```

Pablo Halpern weist in seinem Vortrag „Get Allocators out of our way" ausdrücklich darauf hin, dass diese implizite Konversion beabsichtigt ist, da sie den Aufruf vereinfacht.

Der Standard stellt dafür neben der schon in Abschnitt 1.2 vorgestellten

```
class monotonic_buffer_resource : public std::pmr::memory_resource
{
// ...
};
```

die Klassen

```
class synchronized_pool_resource : public std::pmr::memory_resource
class unsynchronized_pool_resource :public std::pmr::memory_resource
```

sowie einige Funktionen zur Verfügung.

1.3.1 (un)synchronized_pool_resource

Die Klassen *synchronized_pool_resource* und *unsynchronized_pool_resource* verwalten ihre Elemente in Pools. In jedem dieser Pools werden Elemente bis zu einer bestimmten Größe abgelegt (also z.B. im Pool für Elemente, die <= 8 Bytes sind, alle Elemente mit bis zu 8 Bytes, im Pool für Elemente zwischen 9 und 16 Bytes alle Elemente mit 9 bis 16 Bytes usw.). Dabei wird jedes Element eines Pools in einem gleich großen Speicherbereich abgelegt (also z.B. im ersten Pool ein Element mit 4 Bytes in 8 Bytes, usw.). Falls die bisher reservierten Pools nicht mehr ausreichen, werden neue Pools angelegt.

Durch diese Strategie

– wird erreicht, dass die Elemente des Containers nahe beieinander liegen. Dadurch werden die CPU cache loads minimiert, was zu einer Steigerung der Geschwindigkeit führen kann.

– wird die Fragmentierung reduziert.

Diese beiden Klassen sind vor allem für knotenbasierte Container geeignet, die wachsen und schrumpfen können und bei denen alle Knoten gleich groß sind (*list*, *map*, *multmap* usw.).

Eine *synchronized_pool_resource* unterscheidet sich von einer *unsynchronized_pool_resource* im Wesentlichen nur dadurch, dass der Zugriff auf den Speicher von mehreren threads aus synchronisiert ist. Wenn der Zugriff auf den Speicher nur von einem thread erfolgt, sollte man die schnellere *unsynchronized_pool_resource* verwenden.

Falls man *new* und *delete* vermeiden will, kann eine (*un*)*synchronized_pool_resource* den Speicher einer *monotonic_buffer_resource* verwenden:

```
std::pmr::monotonic_buffer_resource pool{ memory.data(),
                                          memory.size() };
std::pmr::synchronized_pool_resource spool{ &pool };
```

Einer (*un*)*synchronized_pool_resource* kann man im Konstruktor als erstes Argument

```
struct pool_options {
    size_t max_blocks_per_chunk        = 0;
    size_t largest_required_pool_block = 0;
};
```

übergeben. Diese Elemente bedeuten:

max_blocks_per_chunk: Die maximale Anzahl von Blöcken, die auf einmal von der vorgeschalteten *memory_resource* zugewiesen werden, um den Pool aufzufüllen.

largest_required_pool_block: Die größte Zuteilungsgröße, die mit Hilfe des Pooling-Mechanismus erfüllt werden muss. Versuche, einen einzelnen Block zu allokieren, der größer als dieser Schwellenwert ist, werden direkt von der vorgeschalteten std::pmr::memory_resource allokiert.

Ist einer dieser beiden Werte gleich Null oder größer als ein von der Implementierung festgelegter Grenzwert, wird stattdessen dieser Grenzwert verwendet. Die Implementierung kann einen Durchgangsschwellenwert wählen, der größer ist als in diesem Feld angegeben.

Beispiel: Die Options

```
std::pmr::pool_options options;
options.max_blocks_per_chunk = 100;
options.largest_required_pool_block = 256;
```

können den Konstruktor einer (*un*)*synchronized_pool_resource*

```
std::pmr::synchronized_pool_resource pool(options);
```

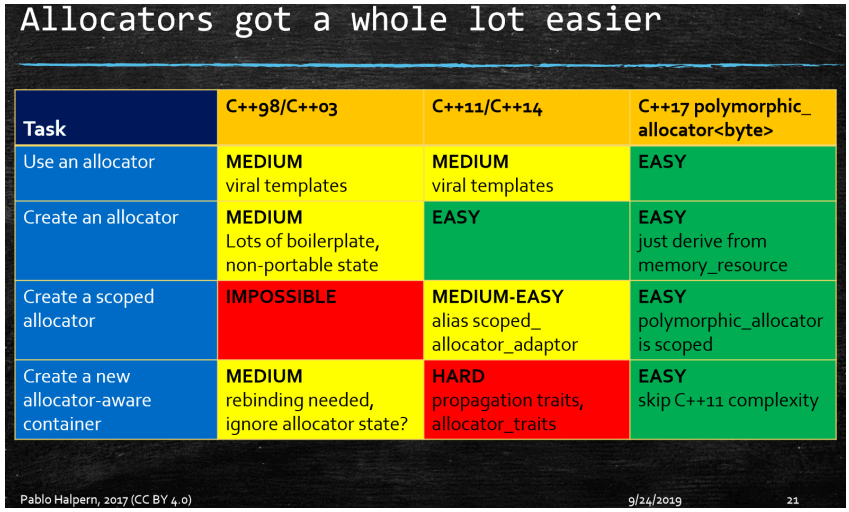
übergeben werden, die dann von einem map verwendet wird:

```
std::pmr::map<long, double> nodes( &pool );
```

1.4 Allokatoren in C++17

Vor C++17 war das Schreiben von eigenen Allokatoren schwierig und fehleranfällig. In C++17 wurde das stark vereinfacht. Die Allokatoren der pmr sind alle von der abstrakten Basisklasse *memory_resource* abgeleitet. Diese besteht nur aus wenigen Funktionen. Falls Sie eigene Allokatoren entwickeln wollen, müssen Sie „nur“ eine eigene Klasse ableiten, in der die Funktionen *do_allocate*, *do_deallocate* und *do_is_equal* überschrieben werden.

Pablo Halpern, einer der maßgeblichen Autoren der pmr, hat das auf seinem Vortrag auf der CppCon 2017 (https://www.youtube.com/watch?v=v3dz-AKOVL8, https://github.com/phalpern) mit dieser Graphik dargestellt:



1.4.1 Was ist an pmr-Allokatoren polymorph? ʘ

Ein *polymorphic_allocator* verwendet virtuelle Funktionen für die Allokation des Speichers. In den bisherigen Beispielen wurde diese Eigenschaft noch nicht explizit verwendet. Im Folgenden wird kurz gezeigt, welche Vorteile polymorphe Allokatoren

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

im Vergleich zu den nicht polymorphen aus der Zeit vor C++17 haben:

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

Beginnen wir zunächst mit einem Beispiel, das den Nachteil von nicht polymorphen Allokatoren zeigen soll.

Da sich der Datentyp eines Typ-Arguments auf den Datentyp eines generischen Typs auswirkt, sind generische Datentypen mit unterschiedlichen Typ-Argumenten verschieden. Sie können einander deswegen nicht zugewiesen werden, oder bei einer Funktion nicht als Argument für einen Parameter verwendet werden.

Beispiel: Da die beiden Klassen

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

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

zwei verschiedene Datentypen sind, haben mit

```
template <typename T, typename MeinAllokator>
class MeinContainer{};
```

die beiden Container

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

unterschiedliche Datentypen. Deswegen ist die Zuweisung:

```
v1 = v2; // error - die Typen von v1 und v2 sind verschieden
```

nicht zulässig, ebenso wenig wie die Übergabe eines Arguments des Typs von v1 an einen Parameter des Typs von v2.

Damit solche Zuweisungen möglich sind, werden die Allokatoren von einer abstrakten Basisklasse abgeleitet

```
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 {};
};
```

und dem Container dann im Konstruktor übergeben:

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

Damit sind Zuweisungen von Containern mit unterschiedlichen Allokatoren möglich:

```
memory_resource_1<int> m1;
memory_resource_2<int> m2;
MeinContainer<int> v1(&m1);
MeinContainer<int> v2(&m2);
v1 = v2; // die Typen von v1 und v2 sind gleich: MeinContainer<int>
```

An eine Funktion wie

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

können v1 und v2 als Argumente übergeben werden:

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

Da der Aufruf von *allocate* in *push_back* virtuell ist, führt das in *v1* zum Aufruf von *allocate* von *memory_resource_1* und in *v2* zum Aufruf von *allocate* von *memory_resource_2*.

Es ist außerdem möglich, den Allokator zur Laufzeit zu ändern.

Auch wenn Sie von dieser Kompatibilität nur selten Gebrauch machen: In der Standardbibliothek von C++17 wird sie oft verwendet und trägt so zu einer einfachen Verwendbarkeit bei. In C++11 war das nicht möglich.

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/w/cpp/memory/scoped_allocator_adaptor
Pablo Halpern: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1850.pdf
alt: 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
https://meetingcpp.com/mcpp/slides/2019/accu2019.191115.pdf
https://www.boost.org/doc/libs/1_62_0/doc/html/container/Cpp11_conformance.html

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