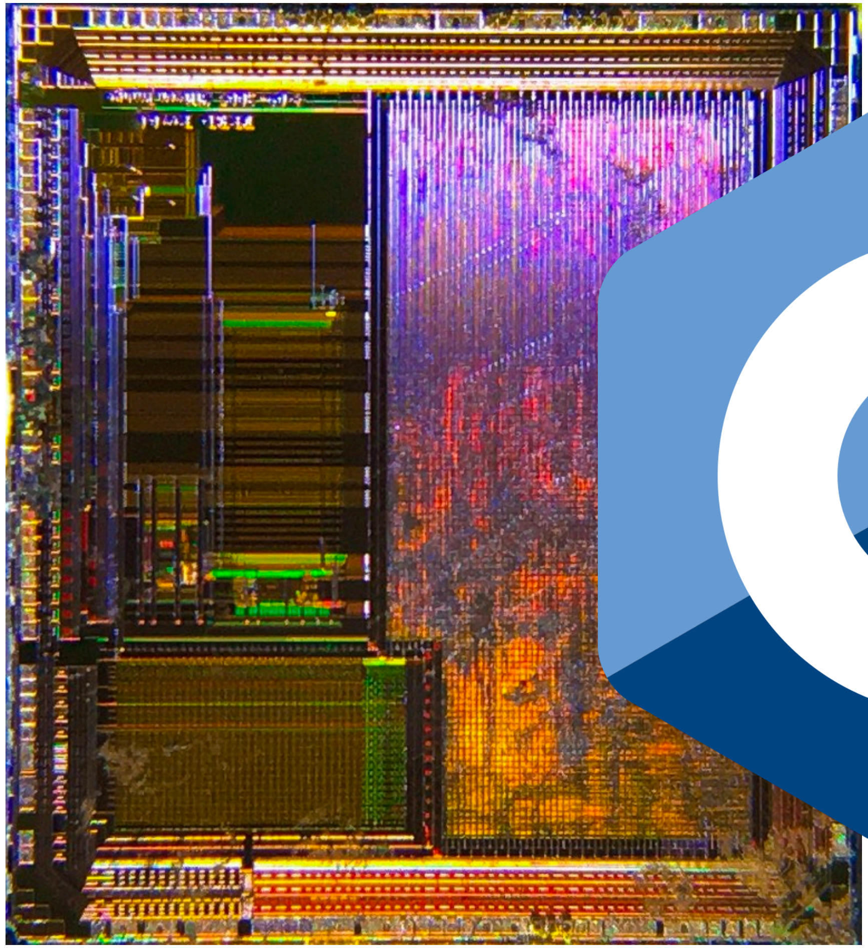
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

MASTER OF SCIENCE
IN ENGINEERING

Embedded Real-Time Software

object oriented programming on Microcontroller C++,
memory pool, containers, RAII, string class, custom literals and closure of oop



10th October 2023 , Prof. Dominique-Stephan Kunz

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Self study from last time

03_1_CPP_autoconsole

Create new the project: CPP_autoconsole_03_01.

Create a variable of type auto an increment it from 0 to 255;

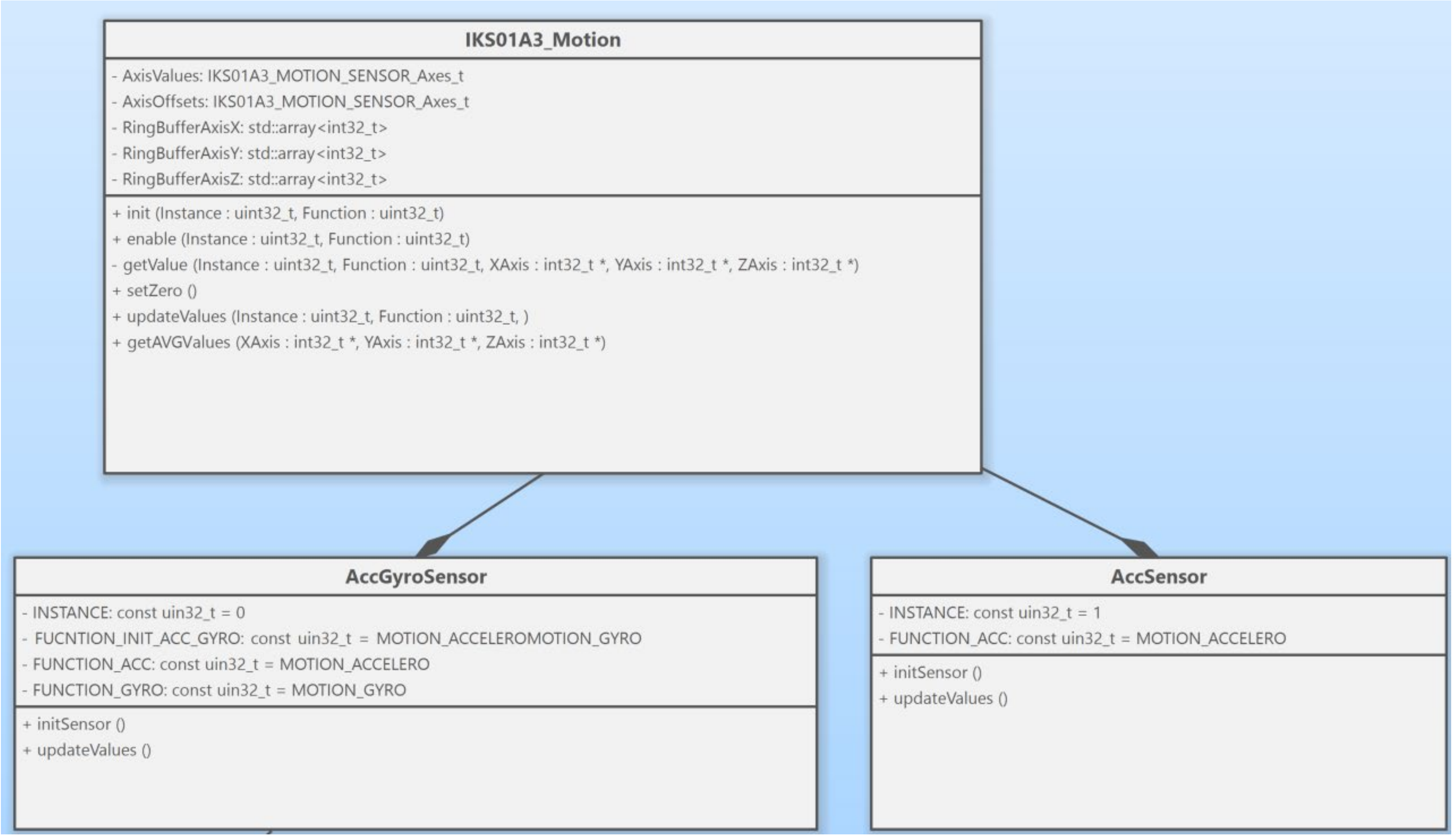
Print the value on the console.

03_2_CPP_ IKS01A3

Create new the project:
CPP_IKS01A3_03_02.

Implement the classes according to the concept.

Output the values of the acceleration sensor on the console.



03_3_CPP_ IKS01A3_ArrayAvg

Extend the project: CPP_IKS01A3_03_02 to CPP_IKS01A3_03_03 .

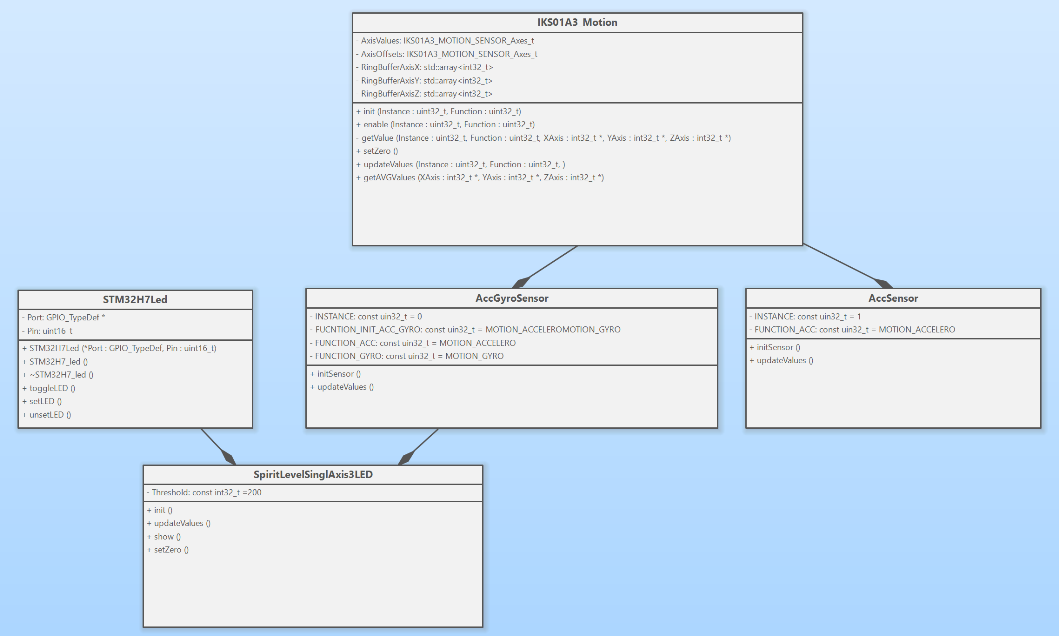
Now we take a container of the type array as a circular buffer.

The array should hold 6 values.

We form the mean value via the array with iterators.

Output this via the console.

03_4_CPP_ SpiritLevel



See detail next slide

Create new the project: CPP_SpiritLevel_03_04.

Implement the classes according to the concept.

The LED1, LED2 and LED3 serve as spirit levels.
To indicate whether the PCB is straight.

For this we extend the class STM32H7Led

Instead of a ring buffer of a container array (as in the project CPP_IKS01A3)
we now use a container of the type vector.

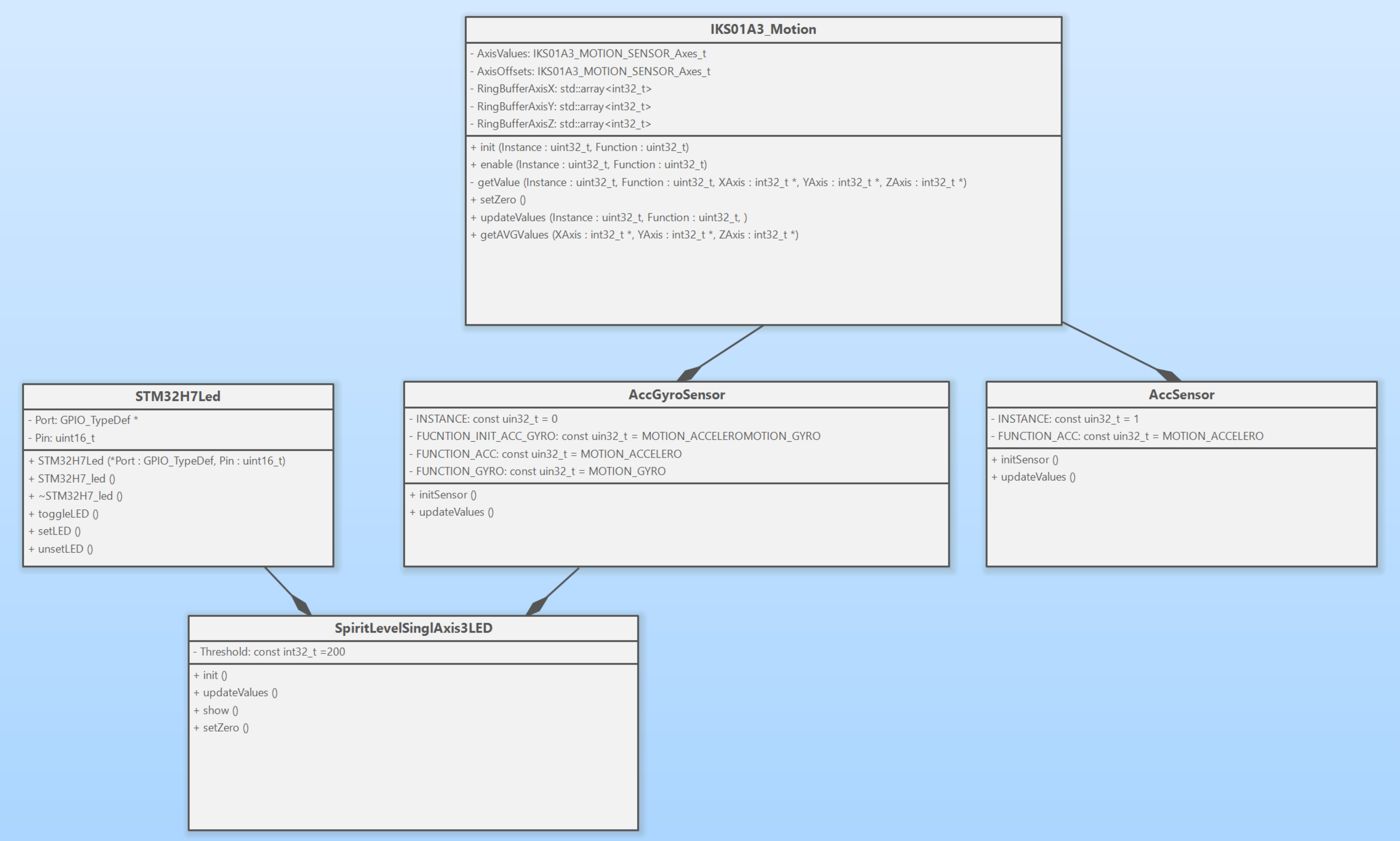
We use a custom allocator from Mr. Kormanyo's author of Real-Time C++.
We take the mean value of the accelerometer of the axis: Y of 10 values.

With the USER key we want to be able to make a zero offset of the sensor.

Set LEDS accordingly to the acc value of the Y-axis .

Output the value of the Y-axis on the console.

03_4_CPP_ SpiritLevel



Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Learning Objectives

- You will know the solution of the custom ring allocator useable for every memory segment.
- You will know how to create a string class object using the user defined memory allocator.
- You will learn what a memory pool is, how it is used, and you will use a simple one as an exercise.
- You will know what associative containers are and get to know some representatives.
- You will know what the RAII problem is and some countermeasures.
- You will get a short introduction to placement new and garbage collectors in C++.
- You will know how to create user defined literals and use them for constants with units.

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Custom memory allocator Part 2

Custom memory allocator from day 3

The allocator from Ch. K is programmed with the main buffer as static
→ position is always memory segment `data`

To overcome this an have a universal solution for stack, heap and data segment changes have to be made:

1. Copy `util_ring_allocator.h` to Copy `util_ring_allocator_std.h`, rename classes by adding suffix `std`
2. Move `buffer and get_ptr` from inside the methode `do_allocate` outside

```
// The ring allocator's memory allocation.
template<const std::uint_fast8_t buffer_alignment>
static void* do_allocate(size_type chunk_size)
{
  ALIGNAS(16) static buffer_type buffer;

  static std::uint8_t* get_ptr = buffer.data;

  // Get the newly allocated pointer.
  std::uint8_t* p = get_ptr;
```

For instance as `protected:` not `static` → make accessible

3. Remove `static` before method `do_allocate` → make allocatable wherever

Custom memory allocator from day 3

With these changes the allocator can be used in memory segment: `data`, `stack` and `heap`:
(ring_allocator_std is the new adapted general class) depending on instantiation

data
(old)
stack
heap

```
std::vector<int32_t , util::ring_allocator_std<int32_t>> RingBuffer1;

int main() {

   std::vector<int32_t , util::ring_allocator<int32_t>> RingBuffer2; //size defined in ring allocator

   std::vector<int32_t , util::ring_allocator_std<int32_t>> RingBuffer3; //size defined in ring allocator

   std::vector<int32_t , util::ring_allocator_std<int32_t>> *ptrRingBuffer4=new std::vector<int32_t , util::ring_allocator_std<int32_t>>(); //size defined in ring allocator

   return 0;
}
```

heap w/ "cage" thx to custom allocator

Ring buffer	Buffer located in memory segment
RingBuffer1	memory segment data
RingBuffer2	memory segment data
RingBuffer3	memory segment stack
RingBuffer4	memory segment heap

Remark:The source code of: “ring_allocator_std” can be found on fhnw gitlab of this part 1.

Custom memory allocator from day 3

The allocator from Ch. K is a ring allocator. → circular buffer

↳ good when using containers w/o fragmentation "memory pooling"

Task of a memory allocator:

1. Take the request to allocate memory ? ✓
2. Search the "heap" for a suitable place to allocate memory ? ~ no free list
3. On success return a point to the new region requested otherwise return a `nullptr` ? ✓
4. When a memory region is released (`delete` or `free`) then make the region available for new allocation. ? ✓

hard - ones

Conclusion: Christopher Kormanyos Ring Allocator

- Allocator replaces the standard library allocator
- Enables the use of the standard library classes while creating a memory pool in user defined RAM segment
- Safe use of classes that use dynamic memory allocation in the background
- Not efficient use of memory
- Has not a “free list”

→ Well suited for replacement of standard library allocator
→ It is a sort of memory pool, however not a pool as a heap replacement!

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

String class

String class

The C++ `string` class can grow and shrink dynamically, hence it is in HEAP.
The string class has no memory allocation in the interface.

The string class (`std::string`) is actually a `typedef` of a container, the `basic_string` container.
Consequently, this class also uses a memory allocation.
https://en.cppreference.com/w/cpp/string/basic_string
https://www.cplusplus.com/reference/string/string/

The solution is to take the basic class and assign a custom memory allocation to it.
https://www.enseignement.polytechnique.fr/informatique/INF478/docs/Cpp/en/cpp/string/basic_string.html

```
basic_string<CharT, std::char_traits<CharT>, std::allocator<CharT>>;
```

specifically:

```
std::basic_string<char,std::char_traits<char>,util::ring_allocator<char>>
MyStringObj;
```

String class

Consequently, the string class realized with `ring_allocator` and the `basic_string` template can be used on the MCU in a safe way:

```
std::basic_string<char,std::char_traits<char>,util::ring_allocator<char>> MyStringObj;

const char hell[]="Hello";
const char wel[]=" World";

MyStringObj=hell;
MyStringObj.append(wel);
MyStringObj+=" wide";
```

uses

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Memory Pool

Definition of memory pool

“**Memory pools**, also called **fixed-size blocks allocation**, is the use of pools for memory management that allows dynamic memory allocation. Dynamic memory allocation can, and has been achieved through the use of techniques such as **malloc** and C++'s operator **new**; although established and reliable implementations, these **suffer from** (external) **fragmentation** because of **variable block sizes**, **it is not recommendable to use** them in a **real time system** due to performance. A more efficient solution is preallocating a number of memory blocks with **the same size called the memory pool**. The application can allocate, access, and free blocks represented by handles at run time.

Many real-time operating systems use memory pools, such as the Transaction Processing Facility.

Some systems, like the web server Nginx, use the term memory pool to refer to a group of variable-size allocations which can be later deallocated all at once. This is also known as a region; see region-based memory management.
”

https://en.wikipedia.org/wiki/Memory_pool

Approach can also be found in FreeRTOS for heap located OS elements!

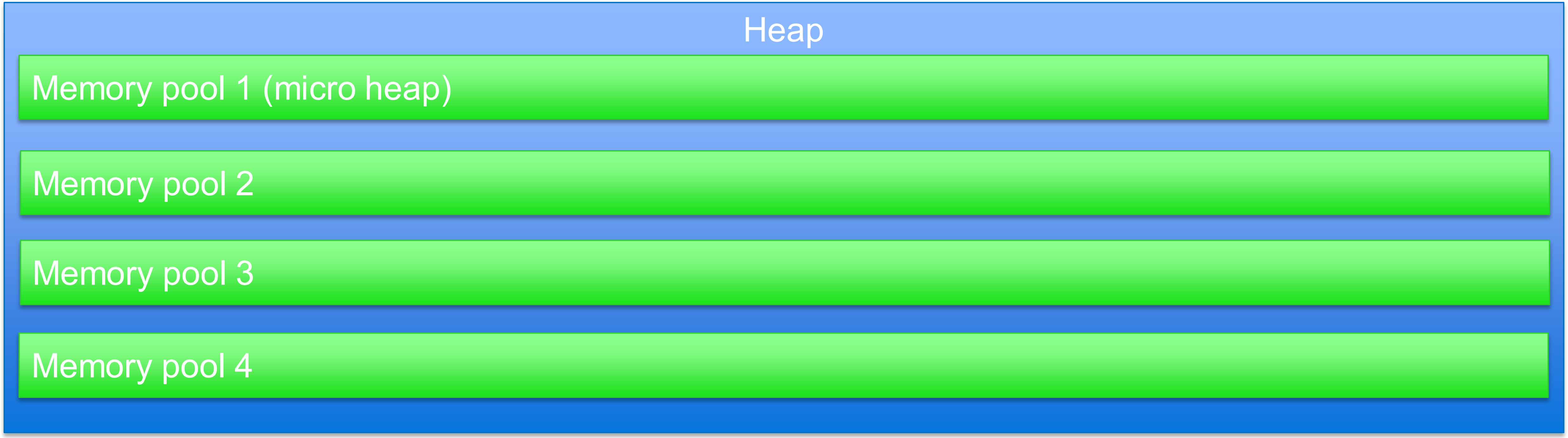
Memory pools are not only used on embedded systems also on:

- Machine learning: OneDNN,Tensorflow…
- Game Engine: Unity, Unreal Engine…
- Databases: Redis, Aerospike…
- …

Memory pools

Memory Pools should overcome the issue with performance and fragmentation.

- The idea is to leave the macro management of the heap to the standard dynamic memory allocation (new-delete, malloc-free) and allocate pools (green).
  Hint: The allocation of new memory segments in the heap needs a lot of time.
- Then do the micro management by a custom memory allocator (inside green).



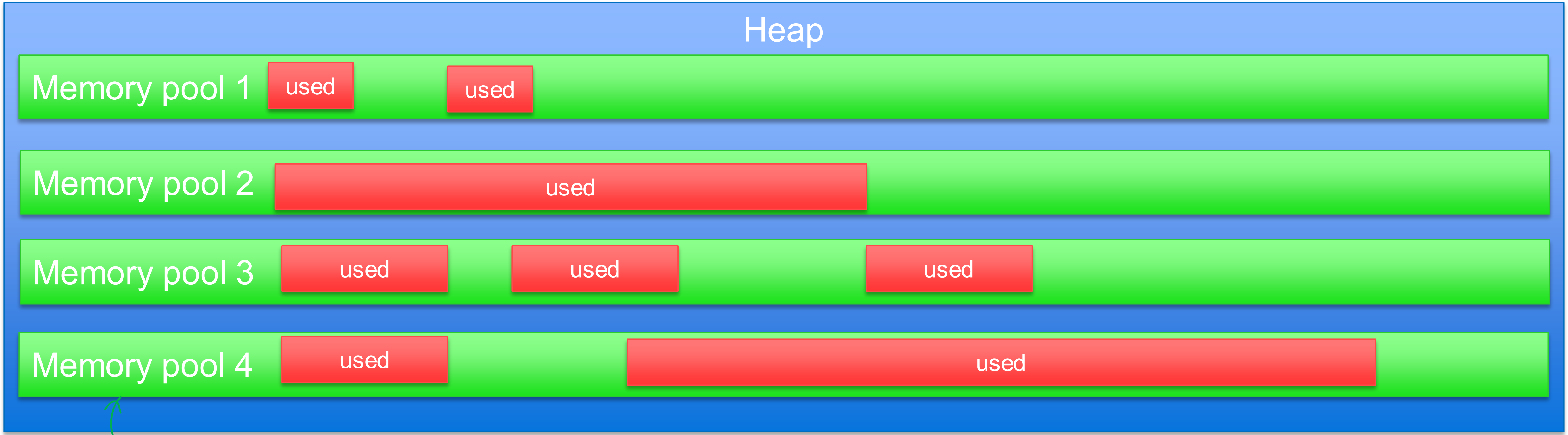
Memory pools

Reminder:
When to use heap:
for dynamic, time limited information (communication)

Do the micro management by a custom memory allocator.

Now a management of free Bytes inside the pool is needed.

Similar to the stdlib dynamic memory allocater with the free list (day3)

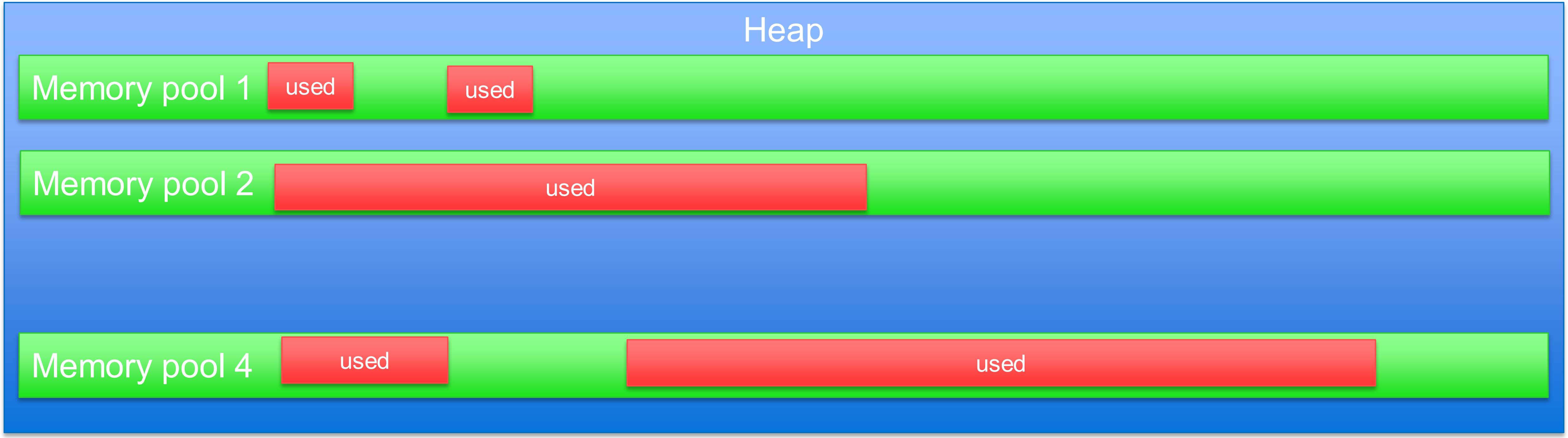


must be at least as big as biggest object

Memory pools

Removing pools is done by the standard dynamic memory allocator (delete Memorypool 3)

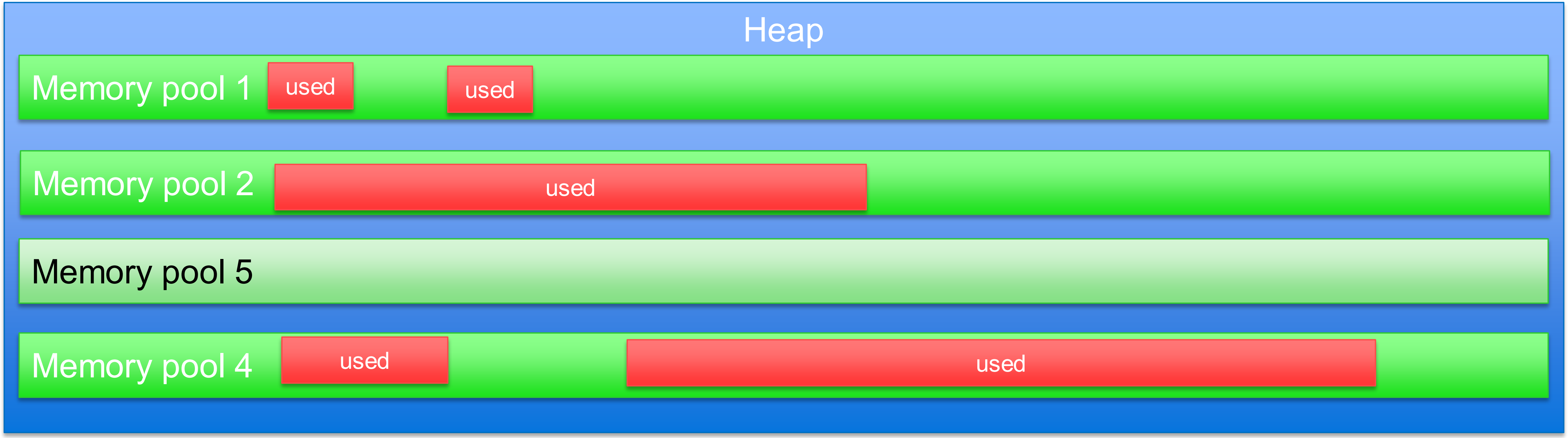
↳ memory pools themselves are dynamic! (created & deleted)



Memory pools

With big segments in the heap of **same size** it is very likely to find new allocation space.

new “Memory pool5”



no free list
but referencing
free spots

Memory pools: example implementation

MemPool Class Header-File

```
template<typename T, size_t NumCells>
class MemPool
{
public:
   MemPool();
   ~MemPool();
   void CreatePool( uint32_t num_cells );
   void DestroyPool();
   T* Allocate();                           <- within-pool
   void Deallocate( void* p );                 custom allocator

private:
   uint8_t* AddrFromIndex( uint32_t i ) const;
   uint32_t IndexFromAddr( const uint8_t* p ) const;

   const uint32_t cell_size_ = sizeof( T );
   uint32_t num_cells_ = NumCells;
   uint32_t num_free_cells_ = NumCells;
   uint32_t num_init_ = 0;
   uint8_t mempool_buffer[sizeof(T)*NumCells];//adaption
   uint8_t* mem_beg_ = mempool_buffer;
   uint8_t* next_ = mem_beg_;
};
```

MemPool Class Cpp-File

```
template<typename T, size_t NumCells>
MemPool<T, NumCells>::MemPool()    // ctor
{
      static_assert( sizeof( uint32_t ) <= sizeof( T ), "sizeof( T ) must be equal or greater than sizeof( uint32_t )" );
}

template<typename T, size_t NumCells>
 MemPool<T, NumCells>::~MemPool()   // dtor
{}

template<typename T, size_t NumCells>
T* MemPool<T, NumCells>::Allocate()
{
   if ( num_init_ < num_cells_ )
   {
      uint32_t* p = reinterpret_cast<uint32_t*>( AddrFromIndex( num_init_ ) );
      *p = ++num_init_;
   }
   T* res = nullptr;
   if ( num_free_cells_ > 0 )
   {
      res = reinterpret_cast<T*>( next_ );

      if ( --num_free_cells_ > 0 )
      {
         next_ = AddrFromIndex( *reinterpret_cast<uint32_t*>( next_ ) );
      }
      else
      {
         next_ = nullptr;
      }
   }
   return res;
}
```

```
template<typename T, size_t NumCells>
void MemPool<T, NumCells>::Deallocate( void* p )
{
   *static_cast<uint32_t*>( p ) = next_ == nullptr ? num_cells_ :
IndexFromAddr( next_ );
   next_ = static_cast<uint8_t*>( p );
   ++num_free_cells_;
}

template<typename T, size_t NumCells>
uint8_t* MemPool<T,NumCells>::AddrFromIndex( uint32_t i ) const
{
   return mem_beg_ + ( i * cell_size_ );
}

template<typename T, size_t NumCells>
uint32_t MemPool<T, NumCells>::IndexFromAddr( const uint8_t* p ) const
{
   return static_cast<uint32_t>( p - mem_beg_ ) / cell_size_;
}
```

Adapted: https://github.com/green-anger/MemoryPool/tree/master

Memory pools: example implementation

```
int main( int argc, char** argv ) {
   constexpr uint8_t NUMELEMENTS = 5;

   MemPool<uint32_t,NUMELEMENTS> mp;

   uint32_t *p0 = mp.Allocate();
   uint32_t *p1 = mp.Allocate();
   uint32_t *p2 = mp.Allocate();
   *p0 = 100;
   *p1 = 10;
   *p2 = 20;
   mp.Deallocate(p0);
   mp.Deallocate(p2);
   uint32_t *p3 = mp.Allocate();
   uint32_t *p4 = mp.Allocate();
    *p3 = 30;
    *p4 = 40;

   auto *ptr_mp_heap = new MemPool<uint32_t,NUMELEMENTS> ();
   uint32_t *p0_heap = ptr_mp_heap->Allocate();
   uint32_t *p1_heap = ptr_mp_heap->Allocate();
   *p0_heap = 523;
   *p1_heap = 623;

   ptr_mp_heap->Deallocate(p0_heap);
   ptr_mp_heap->Deallocate(p2_heap);
   uint32_t *p3_heap = ptr_mp_heap->Allocate();
   uint32_t *p4_heap = ptr_mp_heap->Allocate();
   uint32_t *p5_heap = ptr_mp_heap->Allocate();
   ptr_mp_heap->Deallocate(p1_heap);

   p0_heap = p1_heap = p2_heap = p3_heap = p4_heap = p5_heap = nullptr;
  // ptr_mp_heap->DestroyPool();
   delete ptr_mp_heap;
return 0;
}
```

here, MemPool manager on stack

Memory pools: solutions on the internet

You will a lot of memory pool implementations on the internet.

Before you use one, you should study it thoroughly.
Most make abbreviations and therefore contain limitations:
- Some use the heap in the background
- Some can only be used on the memory segment data

used in P04_02

Investigate this tutorials:
http://www.mario-konrad.ch/blog/programming/cpp-memory_pool.html
https://github.com/green-anger/MemoryPool/tree/master

fast free spot localization

Study the famous TLSF: two-level segregated fit allocator:
https://github.com/rmind/tlsf; https://ieeexplore.ieee.org/document/1311009; https://os.inf.tu-dresden.de/Studium/ReadingGroupArchive/slides/2007/20070606-engel--tlsf.pdf; http://www.gii.upv.es/tlsf/files/ecrts04_tlsf.pdf

Conclusion: Memory pool

- Replaces new and delete for standard classes (not standard library allocator)
- Not efficient use of memory but speed!
- Many available implementations with different approaches and algorithms how to search for empty locations inside the memory pool.

→ Well suited for use in real time and constrained applications

Hint: memory pools can also be used in a memory segment data or stack.

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Container

Container

There are 4 groups of containers:

Sequential containers

Elements are arranged linearly
Members: array, vector, deque, list, forward_list

Green = No dynamic memory allocation used!

Associative ordered containers

Elements can be accessed with a key. The list is always **sorted**.

Members: set, map, multi_set, and multi_map

Associative unordered containers

Like ordered containers but **without** automatic **sorting**.
Members: unordered_set, unordered_ map, unordered_ multi_set, and unordered_ multi_map

Container Adapter

Similar to standard containers but with reduced functionality fit for their purpose.
Members: stack, queue, and priority_queue

Overview associative containers

Overview associative containers

Overview

Container	Beschreibung
set	Sorted and duplicate-free set of elements
map	Single keys refer to a target value
multiset	Sorted set of elements that occur more than once.
multimap	Sorted keys can occur several times each.

search for key, get element

↓

eg FSM

transition lists

map<Key, Type>
mulitmap<Key, Type>

set<Key>
mulitset<Key

Similarities and differences

Eigenschaft	set	map	multiset	multimap
Unique key	Yes	-	Yes	-
Key to target values	-	Yes	-	Yes
Key type	<	<	<	<
Dynamic size	Yes	Yes	Yes	Yes
Overhead per Element	Yes	Yes	Yes	Yes
Memory layout	tree	tree	tree	tree
Insert, delete*	guaranteed	guaranteed	guaranteed	guaranteed
search	guaranteed	guaranteed	guaranteed	guaranteed
sort	always	always	always	always
Iterators	random	random	random	random
Algorithms	all	all	all	all

Source: C++, Umfassendes Profiwissen zu Modern C++

Iterators especially for associative containers

Each container has 2 different iterator types:

- iterator = allows dereferencing and changing the elements in the container.
- const_interator = allows dereferencing and prohibits changing the elements.

Methods:

begin() end() backwards: rbegin() rend()

Returns the reference of the iterators from the first or last element.

size() empty() max_size()

Returns the number of elements

resize() reserve() clear()

Resizes the container (except array)

operator[]at(), dat()

Read and write to any element in the container: container[index] or container.at(index)

Insert() erase() extract()

Insert, delete or cut elements.

assign() swap() merge()

Reinitialise container, swap two elements, merge two elements into one

push_front() push_back() emplace_front() emplace_back() emplace

Insert elements at the end or beginning of sequence-based elements.

find() count() contains() lower_bound() upper_bound() equal_range()

Find specific elements in an associative container or count how many times it occurs.

NOTE: std::string fulfils all requirements for a container except that the data type is fixed.

Container: maps

Container: map

A map is an associative container (a superimposed template class).

It has a strong similaritiy to TreeMap with interface NaviagableMap in Java.

This allrounder grows automatically. You can insert anywhere. The elements are read sequentially forwards, backwards or randomly by numerical index. It keeps its elements as a tree structure.

The elements are pairs of <const key, target>

What implies this definition?

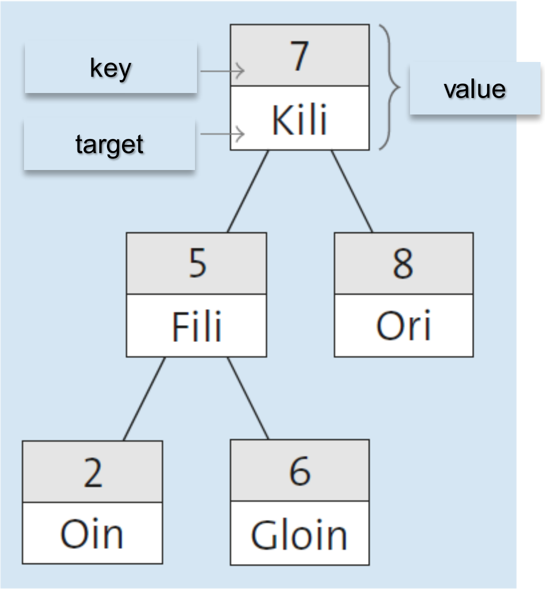
You have to include:

#include <map>

to make use of this container.

```
std::map<int16_t, std::string> numStringMap;

numStringMap.insert(std::make_pair(1,„FIRST"));
```



Container: set

Container: set

A `set` is an associative container (a superimposed template class).

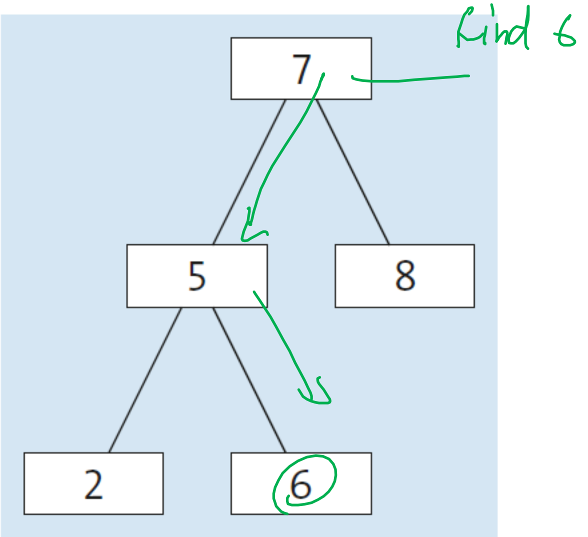
Has strong similarities to `TreeMap` with interface `NavigableMap` in Java.

This all-rounder grows automatically. Although you can insert anywhere.

The elements are read sequentially forwards, backwards or randomly by number index. It holds its elements as a tree structure.

The individual elements in the `set` are single elements as with vector, <key>.

search tree



With the container, you must always include the corresponding header file!

#include <set>

Example:

```
std::set<int16_t> numSet;

numSet.insert( 1);
```

Container operations

Container: Initialisations

There are several ways to initialise a container:

1. Individually with `insert` (`pushback` with `vector`)
   ```
   container1.insert = ‘A’;
   ```

2. Constructor with initialisation list
   ```
   container<type> container1 = {'A','B','C','D','E','F’};
   ```
   or with maps
   ```
   container<type> container1 = {{'A',23},{'B',44}};
   ```
   Note it works with or without =.

3. With `insert` and an initialisation list
   ```
   container1.insert({'A','B','C','D','E','F'});
   ```

Container: copy

There are several ways to copy a container:

1. Iteration
   `FOR loop`
2. Assignment (=)
   `container2 = container1;`
3. As constructor pass element
   `container<type> container2(container1);`
4. Using the STL `copy` function
   `copy(container1.begin(), container1.end(), back_inserter(container2));`
5. With the container method `assign`
   `container2.assign(container1.begin(), container1.end());`

With these methods you can also insert and append.

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

RAII: <u>R</u>esource <u>Ac</u>quisition <u>i</u>s <u>I</u>nitialisation

RAII: Resource Acquisition is Initialisation

This comes from a technical problem and leads to a programming principle.

Request resources (objects, variables in memory, etc.) and always initialise and if it fails, clean it up. This is a very relevant topic in C++.

RAII

Other description and view:

Scope-Bound Resource Management (SBRM).

This sums it up very well. It is mainly about managing dynamically allocated resources safely.

There are two main issues why RAII what makes it so important:

- Constructors that dynamically create resources.
- Throwing exceptions and/or cleaning up

RAII: Problem Example 1: dynamic memory allocation

Dynamic memory allocation is always performed in the constructor.

A constructor must always leave valid resources.

If the constructor is terminated because of an error, the destructor is not called because the program assumes that no objects were created!

If there is a resource acquisition error in the constructor, the constructor must also it clean up.

Another aspect is that the destructor can remove the object by itself, with all existing resources and shares. (Stack Based Resource Management)

RAII: Problem Example 1: dynamic memory allocation

```
class Buffer{
private:
   uint32_t * PointFirstBuffer;
   uint32_t * PointSecondBuffer;
public:
   Buffer(){ // ctor
      PointFirstBuffer = new uint32_t [500]();
      PointSecondBuffer = new uint32_t [500]();
   };
};
```

What to do?

if 2nd array doesn't allocate correctly
ctor fails but 1st buffer stays
↓
memory leakage

RAII: Problem Example 1: dynamic memory allocation

```
class Buffer{
public:
   uint32_t * PointFirstBuffer;
   uint32_t * PointSecondBuffer;
   Buffer(){
      PointFirstBuffer = new uint32_t [500](); //if fails, will throw an allocation error
      PointSecondBuffer = new(std::nothrow) uint32_t [500]();
      if(PointSecondBuffer == NULL) delete[] PointFirstBuffer;
   };
   ~Buffer(){
      delete[] PointFirstBuffer;
      delete[] PointSecondBuffer;
   };
};
```

Other solutions?

Solve RAII automatically

Solve RAII automatically

In C++, many innovations go in the direction of solving the RAII problem automatically.

Examples:

-> only ptr pointing to this object

- Pointers: unique_ptr, shared_ptr, weak_ptr instead of pointers (auto_ptr since C++ 11 deprecated).
- Threads: std::jthread instead of std::thread
- Mutex: lock_guard or unique_lock instead of mutex::lock

Dynamic memory allocation in C++ (new, smart pointer, auto make_)

Dynamic memory allocation in C++:

new

In C we have seen that methods are necessary for dynamic reservation in HEAP.
In C++, the same functions as from C can be used, but there are more comfortable C++ solutions.
As with Java, C++ has the `new` operator.
This creates an object dynamically in the HEAP and must be removed again with `delete`.

The new operator always returns a pointer to the newly created object.
Not only objects of classes can be created dynamically, but all data types!

```
uint8_t *Einbyte = new uint8_t;
*Einbyte=5;
std::cout << "Adresse Einbyte " << (uint32_t)Einbyte << endl;
std::cout << "Wert Einbyte " << (uint32_t)*Einbyte<<endl;
delete Einbyte;
```

Initialization with a value can take place in the definition in a subsequent bracket:

```
uint8_t *Einbyte = new uint8_t(5);
```

Dynamic Memory Allocation in C++: Exception

new

If not enough memory is available, an "exception" of the type: `std::bad_alloc` is thrown.

However, catching this "exception" is not always reliable.

Alternatively, `new(notthrow)` can be used to specify that no "exception" should be thrown, but that a NULL should be assigned to the pointer if there is not enough memory.

```
uint8_t *Einbyte = new(nothrow) uint8_t;
if (Einbyte==NULL) cout << "Memory allocation failed\n";
else
{
*Einbyte=5;
std::cout << "Adresse Einbyte " << (uint32_t)Einbyte << endl;
std::cout << "Wert Einbyte " << (uint32_t)*Einbyte<<endl;
delete Einbyte;
}
```

Dynamic memory allocation in C++: with intelligent pointers

Smart pointer | C++11

Smart pointers ensure that objects, to which they point, are deleted after leaving the scope! They belong to a class and **are not standard data types**.

The smart pointer: `auto_ptr` was the first and has been replaced by `unique_ptr` with C++11!

Do not use `auto_ptr` any more.

The intelligent pointers include: `unique_ptr`, `shared_ptr` and `weak_ptr`.

These intelligent pointers belong to the std namespace.

To use them, the file `memory` must be included.

```
#include <memory>
```

std:unique_ptr <Datatype> ObjectName (new Datatype (Initialisation));

```
unique_ptr<uint32_t> SmartPointerOnValue(new uint32_t (42));
*SmartPointerOnValue=34;
```

The use is like a pointer...But the pointing to other objects is different from standard pointers!

These pointers are classes and consequently also contain methods for operations.

Dynamic memory allocation in C++: Fields with smart pointers, unique_ptr

Smart pointer

C++11

Fields can also be created with the unique_ptr.

```
unique_ptr<uint32_t[]> SmartPointerOnArray(new uint32_t [256]);
SmartPointerOnArray[33]=34;
```

Accessed can be as with standard pointers.

However, there are also other possibilities:

`std::vector` und `std::array`.

Dynamic memory allocation in C++: with intelligent pointers

Smart pointer C++11

The most commonly used are: `unique_ptr` and `shared_ptr`.

As the name already says: `unique_ptr` there is only one pointer pointing to the object.

It cannot be copied (otherwise there would be several pointing to the object) but can be moved (e.g. as a return value).

The `shared_ptr` is used to share the pointer. It can be copied, then two pointers point to the same value. Only when the last pointer is removed, the object is deleted from the memory. The size consists of two pointers: one for the object and one for the common control block that contains the number of references.

The `weak_ptr` is a special case of a smart pointer for use in conjunction with `shared_ptr`. A `weak_ptr` provides access to an object that is owned by one or more `shared_ptr` instances but does not participate in reference counting. Used when you want to observe an object but do not require it to stay alive. Required in some cases to resolve circular references between `shared_ptr` instances.

Dynamic memory allocation in C++: intelligent pointers with auto and make

Smart pointer | C++11

When defining an intelligent pointer, it is noticeable that the data type must be specified twice.

```
std:unique_ptr <Datatype> ObjectName (new Datatype (Initialisation));
```

Since C++14 and C++17 there is a solution to this problem: make_unique<> and make_shared<> in connection with auto.

```
auto ObjectName = make_unique <Datatype> (Initialisation);
```

```
std::unique_ptr<uint32_t> SmartPointerOnValue(new uint32_t (42));    C++11
auto SmartPointerOnValue2 = std::make_unique<uint32_t>(42);          C++17

*SmartPointerOnValue=34;
*SmartPointerOnValue2=1562;

std::unique_ptr<uint32_t[]> SmartPointerOnArray(new uint32_t [256]);

SmartPointerOnArray[33]=34;
```

This does not seem to work for fields....

Smart Pointer use on mem Pool

Smart pointer

C++11

auto ObjectName = make_unique <Datatype> (Initialisation);

```
int main( int argc, char** argv ) {

    constexpr uint8_t NUMELEMENTS = 5;


    std::unique_ptr<MemPool<uint32_t,NUMELEMENTS>>  SmartPointerOnValue(new MemPool<uint32_t,NUMELEMENTS> ());

    uint32_t *p0_heap_smart = SmartPointerOnValue->Allocate();
    *p0_heap_smart=50;


    auto SmartPointerOnValue2  = std::make_unique <MemPool<uint32_t,NUMELEMENTS>> ();

    uint32_t *p0_heap_smart_make = SmartPointerOnValue2->Allocate();
    *p0_heap_smart_make = 100;

    SmartPointerOnValue2->Deallocate(p0_heap_smart_make);

    return 0;
}
```

// MemPool->Allocate() Method returns pointer to allocated buffer

<-- new: on heap

<-- make_unique creates object on heap and returns unique pt

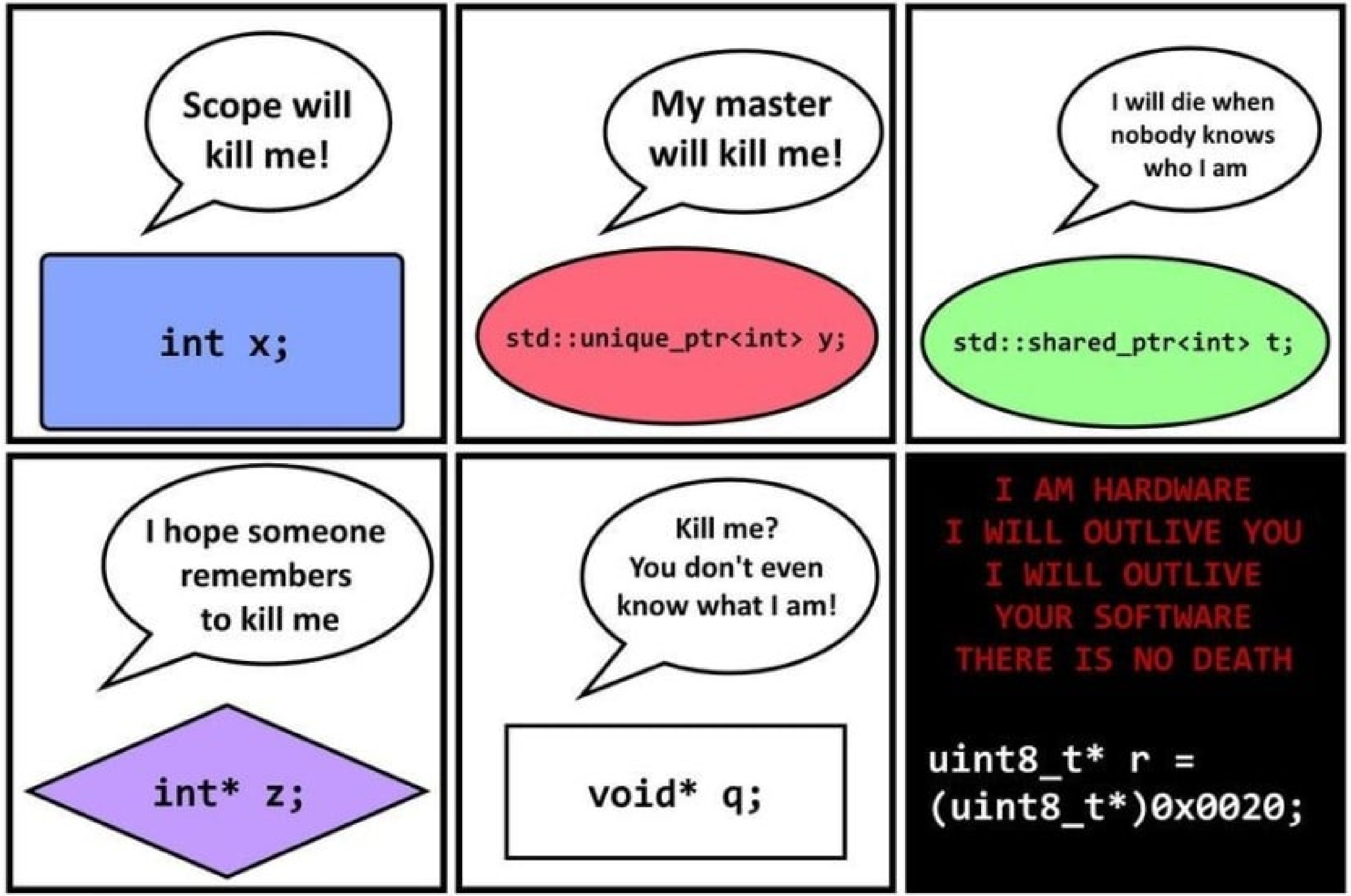
Conclusion

!

Smart Pointer are useful for object which are located on the heap.

→ For the use of dynamic allocated memory pools on the heap the smart pointers are useful on embedded systems.

Death and Memory (C++ Stories)



2017 Ólafur Waage (@olafurw)
with thanks to Frank A. Krueger (@praeclarum)

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Placement new

Palcement new

In C++, "placement new" is a specialized version of `new` that allows you to construct an object at a specific memory location you specify, rather than on the heap or the stack as usual.

Even when the term `new` is used, there is **no dynamic memory allocator involved**!

management lists not affected!

`new`, in this version, will not throw a memory exception!

Consequently, there is no need for `delete`, however the **destructor must be called manually**!

```
class MyClass {
public:
   MyClass(int value) : data(value) {}
   void doSomething() {
      // ... Perform some action ...
   }
private:
   int data;
};

int main() {
   // Define a memory buffer for MyObject
   char memoryBuffer[sizeof(MyClass)];
   // Use placement new to construct an object within the memory buffer
   MyClass* obj = new (memoryBuffer) MyClass(42);
   // Now, you can use obj as a regular object
   obj->doSomething();
   // Remember to explicitly call the destructor for manual cleanup
   obj->~MyObject();
   return 0;
}
```

Use Cases:
- Place an object on hardware registers

Not that useful on MCU

Garbage Collector in C++

Garbage Collector (GC) in C++ (and C)

We know Garbage Collector from other high level programming languages.

There are implementations in C++ available. However, they have not made it to the standard and not in to recognised public library nor frameworks.

One well known GC is the one of Boehm-Demers-Weiser

https://en.wikipedia.org/wiki/Boehm_garbage_collector
https://www.hboehm.info/gc/

Most of the GC cope with the issues:

Dangling pointers

Doubled unallocating memory

→ What smartpointers and other modern classes do by leaving the scope!

there ar GC's but usually "decentralized" approach

Only a few GC cover the topic of defragmenting the heap.

→ What are the risks and side effects of defragmenting?

Custom literals

Custom literals

Constants are often used in programs.

These can be solved with `#define` and `const` in the program.
But what if you use values with unit references?

This is where user-defined literals come into play.

The use of this technique should also be advantageous for security-relevant developments.
See: 06-Literatur/IX_201610_CPP_ProgrammierungFuerEmbedded_RGrimm.pdf

Custom literals

Custom defined literals are calculated at compile time.

A string literal is a composition of:
Built-in-literal+_+suffix.
Typically:

Built-in-literal	corresponds	to value
Suffix	corresponds	to unit

One defines a base unit and converts all units with operator methods (functions) into the base unit.

One creates these operator functions as follows:

Data type `operator""` _Suffix	(data type variable name)	{conversion function with `return`}
(return data type,	(function interface)	{function body}

Custom literals

Example Code:

```
#include <iostream>

long double operator"" _kg (long double x) {return x*1;}
long double operator"" _t (long double x) {return x*1'000;}
long double operator"" _g (long double x) {return x/1'000;}
long double operator"" _mg (long double x) {return x/1'000'000;}

int main() {
   std::cout << „base unit is kg:" << std::endl;
   std::cout << "1 kg: " << 1.0_kg<< std::endl;
   std::cout << "1 t: " << 1.0_t<< std::endl;
   std::cout << "1 g: " << 1.0_g<< std::endl;
   std::cout << "1 mg: " << 1.0_mg<< std::endl;
return 0;
}
```

CLI output

```
Basiseinheit ist kg:
1 kg: 1
1 t: 1000
1 g: 0.001
1 mg: 1e-06
```

Custom literals

Example Code:

```
#include <iostream>

long double operator"" _kg (long double x) {return x*1;}
long double operator"" _t (long double x) {return x*1'000;}
long double operator"" _g (long double x) {return x/1'000;}
long double operator"" _mg (long double x) {return x/1'000'000;}

int main() {
std::cout << std::endl;
   std::cout << "1.234_kg+2345.6_g:  " <<(1.234_kg+2345.6_g)<< std::endl;

   std::cout << std::endl;
   std::cout << "1.234_kg + 4 * 2345.6_g:  " <<(1.234_kg+4*2345.6_g)<< std::endl;

   return 0;
}
```

CLI output

```
1.234_kg+2345.6_g:  3.5796

1.234_kg + 4 * 2345.6_g:  10.6164
```

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Closing C++ on MCU

C++ for embedded

Bjarne Stroustrup is a professor of computer science at Texas A&M University. He is best known for developing the C++ programming language, which he is still involved in standardising today.[2] He is currently a visiting professor at Columbia University and works at Morgan Stanley.[3][4]

Source: https://www.morganstanley.com/profiles/bjarne-stroustrup-managing-director-technology

Source: https://de.wikipedia.org/wiki/Bjarne_Stroustrup

„Improve performance and ability to work directly with hardware —
make C++ even better for embedded systems programming and highperformance computation."

Quelle: https://www.stroustrup.com/C++11FAQ.html#specific-aims

Or Rust?

“If you’re building something embedded, it's easier to be safe with Rust, though you will probably need to rely heavily on unsafe code, since the performance requirements are more intense. Rust can also readily interact with the C APIs which means that it can readily use any existing C code for embedded development”
9.7.2021 Ben Fenwick

Source: https://devetry.com/blog/c-v-rust-speed-safety-community-comparison/

No problem für MCU

Watch out on MCU

Caution on MCU

C++ for microcontrollers?

We have got to know some of C++.

Much of it is said to be well suited for microcontrollers, even without MMU.

Discussion:

What is missing....?

What are advantages / disadvantages?

Your opinion for embedded application?

C++ without OOP?

Discussion on/with Paddlet

https://padlet.com/dominiquekunz1/68osqg8kljhrb33u

OOP
- polymorphism

Stdlib container:array

enum class

Templates

STL-Algorithms

Container

Class: String

Dynamic memory allocation

Custom literals

Console

Strategies to use C++ (also C) for MCUs without MMU

There are recognized strategies how to use C++ on microcontrollers without having a memory management unit:

Day 1 and 2

1. **Do not use heap.** Use everything of the language except elements that use the heap!
2. **Only allocate elements in the heap, do not release it**.

Day 3 and 4

3. **Use stack or “static memory” for elements that usually use heap.**
4. **Free heap completely, at time X.**

Day 3 and 4

5. **Segment heap into smaller "heaps" that shrink and grow in the defined frame/segment.** Frame size is fixed and defined at compile time. These segments can be released and allocated.
   → memory pools

frequency as it is practised

C++ can be used on MCU

C++ can be very efficient on MCU.

C++ can be blended with C

→ Suitable for realtime applications

What we have learned

- We discussed the solution of the custom ring allocator useable for every memory segment.
- We discussed how to create a string class object using the user defined memory allocator.
- We have discussed what a memory pool is and how it is used.
- We have discussed the associative containers.
- We have covered the RAII problem and discussed the countermeasures.
- We discussed briefly placement new and garbage collectors in C++.
- We covered how to create user defined literals and use them for constants with units.

Structure of the lessons

Self study from last time	
Learning Objectives	
Custom memory allocator Part 2	
String class	
Memory Pool	
Container	• Overview associative containers • Container: maps • Container: set • Container operations
RAII	• RAII: Resource Acquisition is Initialisation • Solve RAII automatically • Dynamic memory allocation in C++ (new, smart pointer, auto make_)
Other C++ topics you can come across in C++ for embedded	• Placement new • Garbage Collector in C++ • Custom literals
Closing C++ on MCU	
Self study	

Self study

Task 04_1_CPP_SpitirLevel

Extend the project: CPP_SpiritLevel_03_04.

Add a container of type `map` with 3 Elements: Min, Max and AVG.
Store the minimum, maximum and average of acc axis Y in the container in the elements: Min, Max and AVG.
Use the ring-memory allocator for the map.

Write on the console the min, the max and the average values.

Create an object of a `string class` with ring-memory allocator and store the message to be send to console in it.

Optional:

- Make use of smartpointers

Task 04_2_CPP_Blinky_MemoryPool on heap with smartpointer

Extend the project: CPP_Blinky_02_03.

We want to use a smartpointer pointing on a memory pool containing 3 `BlinkingLed` Objects. → the pool is running on heap.

The rest of the task remains unchanged.

Steps:

Initialise a memory pool to containing 4 objects of the class `BlinkingLed` by using a smartpointer.

Allocate 4 Leds 0 to 4. 1 to 3 are the physical LEDs.

Extend the constructor of `BlinkingLed` with an empty default constructor.

Add a method `setPinPort` to the class `BlinkingLed` to set the pin and port if the object is created with the empty constructor.

Set the Frequency, pin and port to Led 1 to 3 accordingly to exercise 02_03.

Invoke processBlinking for each LED.

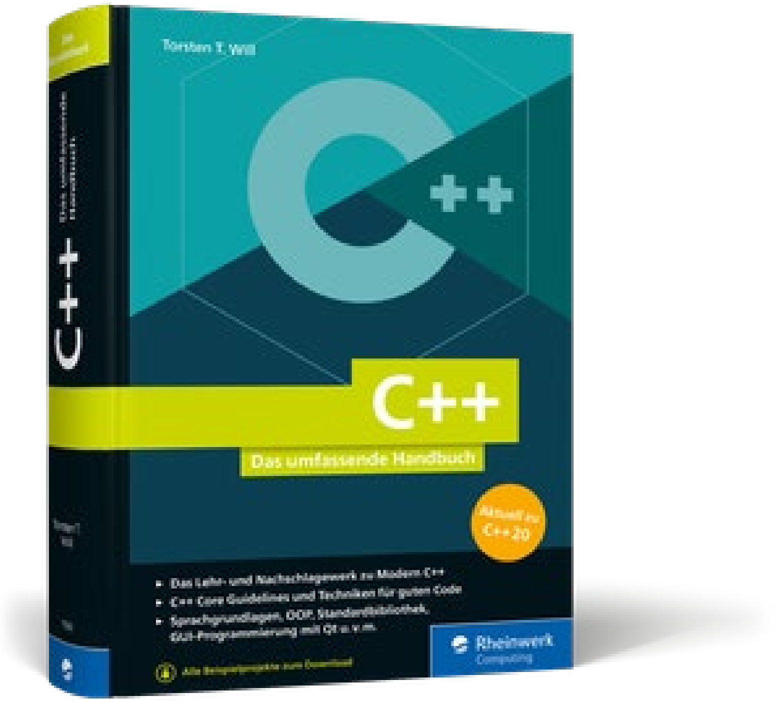
Thank you for your attention and cooperation

Appendix: optional Literature

C von A bis Z

kostenlos

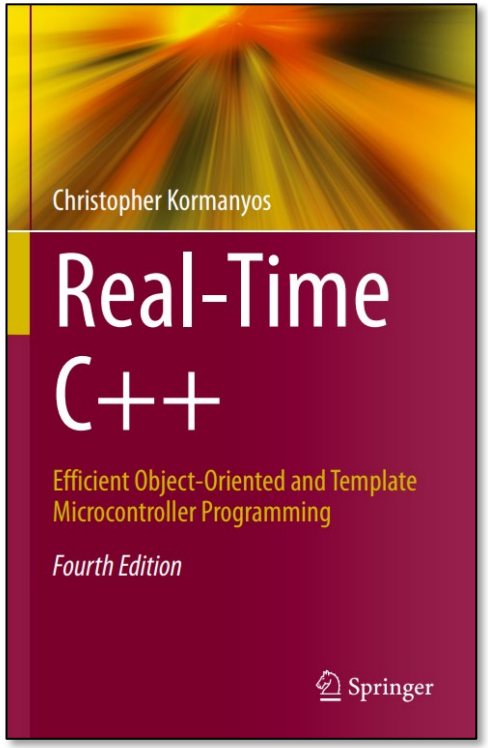
https://openbook.rheinwerk-verlag.de/c_von_a_bis_z/



C++

Umfassendes Profiwissen zu Modern C++

https://www.rheinwerk-verlag.de/linux-das-umfassende-handbuch/?v=3855&GPP=openbook



Real-Time C++

Efficient Object-Oriented and Template Microcontroller Programming

https://link.springer.com/book/10.1007/978-3-662-62996-3