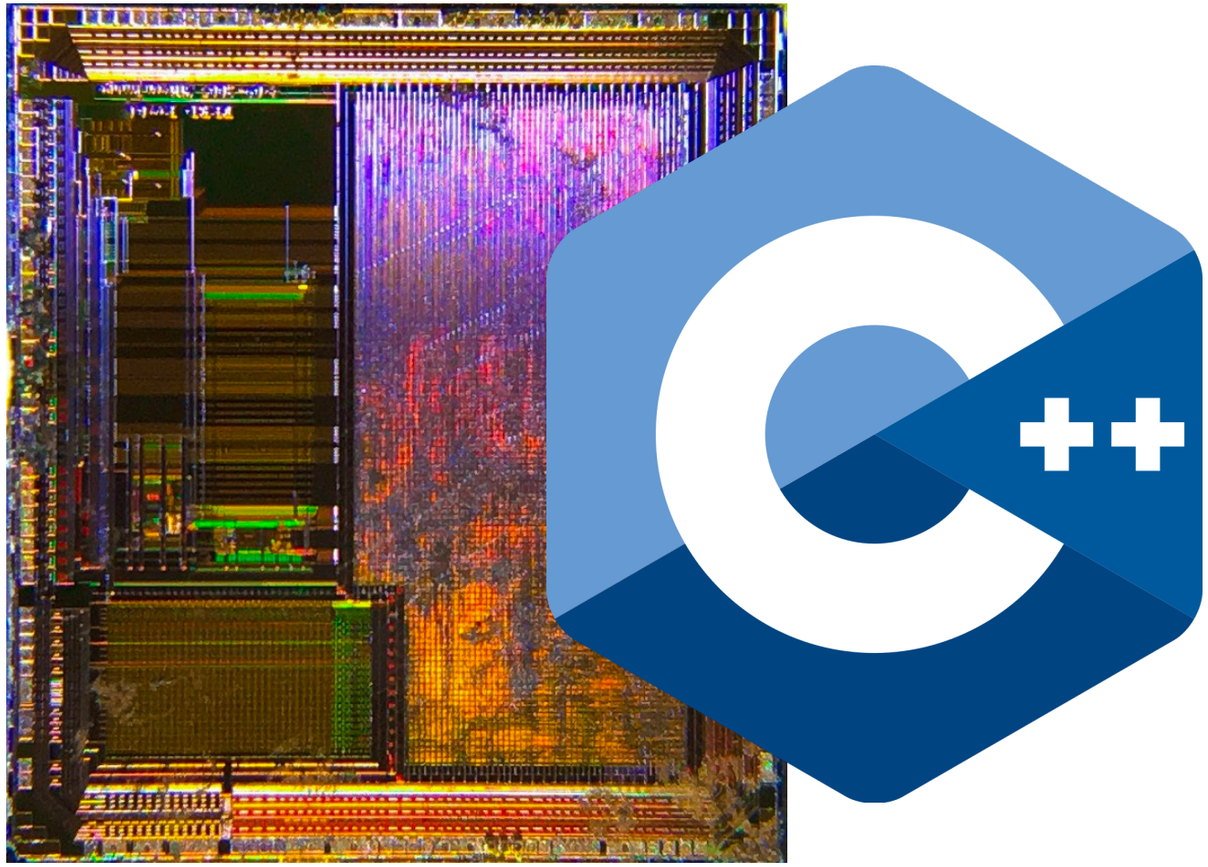
# Embedded Real-Time Software

object oriented programming on Microcontroller C++, containers and dynamic memory allocation



**3th October 2023 , Prof. Dominique-Stephan Kunz**

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Self study from last time

# 02_1_CPP_BlinkyDelayNotBlocking

Expand project: CPP_Blinky_02_01.

NoneBlockSystemTickDelay:
Use the SystemTick timer to implement a software delay that does not block. (see exercise 1_5)

STM32H7Led:
Use the HAL library to control the LEDs in STM32H7Led

Caution we need 2 constructors for STM32H7Led:

- One without initialization parameters.
- One with initialization parameters!

BlinkingLed class:

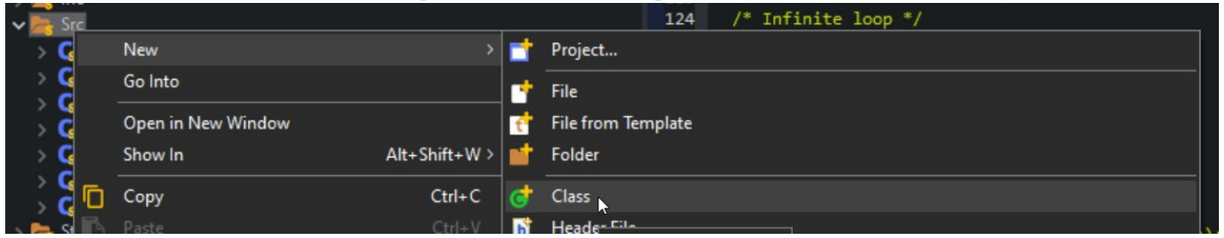
Initialised with a Led and realises blinking on every call by using the inherited logic of STM32H7Led and NoneBlockSystemTickDelay.

Main:

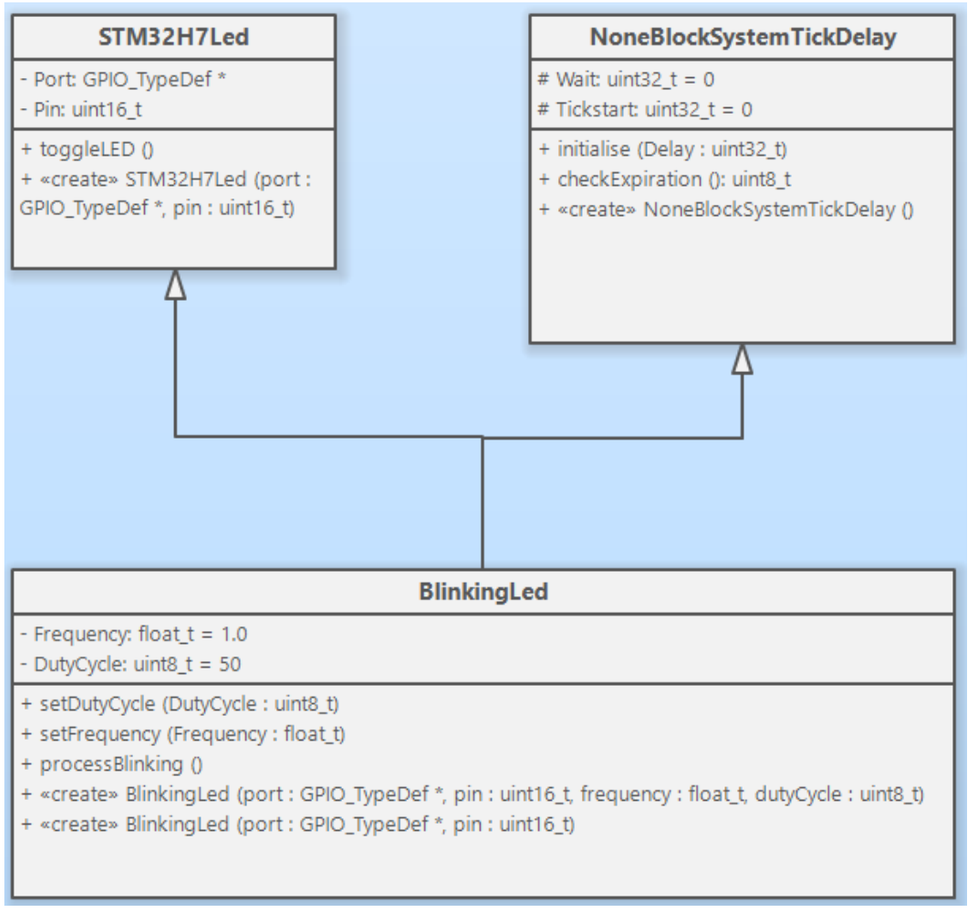
Instantiated the three LEDs with 50% duty cycles and the times: LED1 every 250ms.

LED2 every 500ms, LED3 every 1000ms.

Make use of the wizard for creating a class and setter/getters (setters for BlinkinLed)!




Setter and getters: select Header-File, ALT+SHIFT+S, then “Generate Getter Setters…”

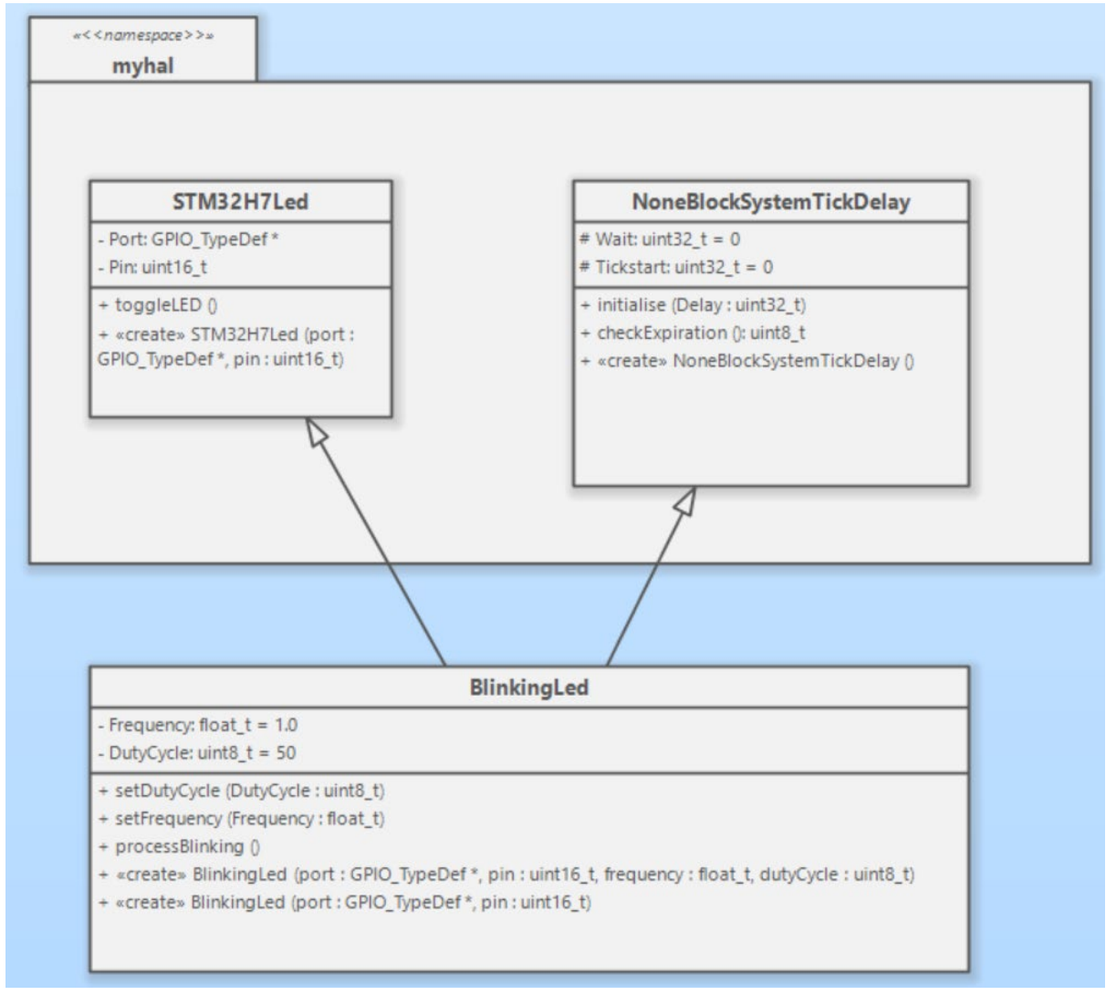


## 02_2_CPP_BlinkyNamespace

Expand project: CPP_Blinky_02_02.

Define a namespace: “myhal”

And include STM32H7Led and NoneBlockSystemTickDelay into this namespace.

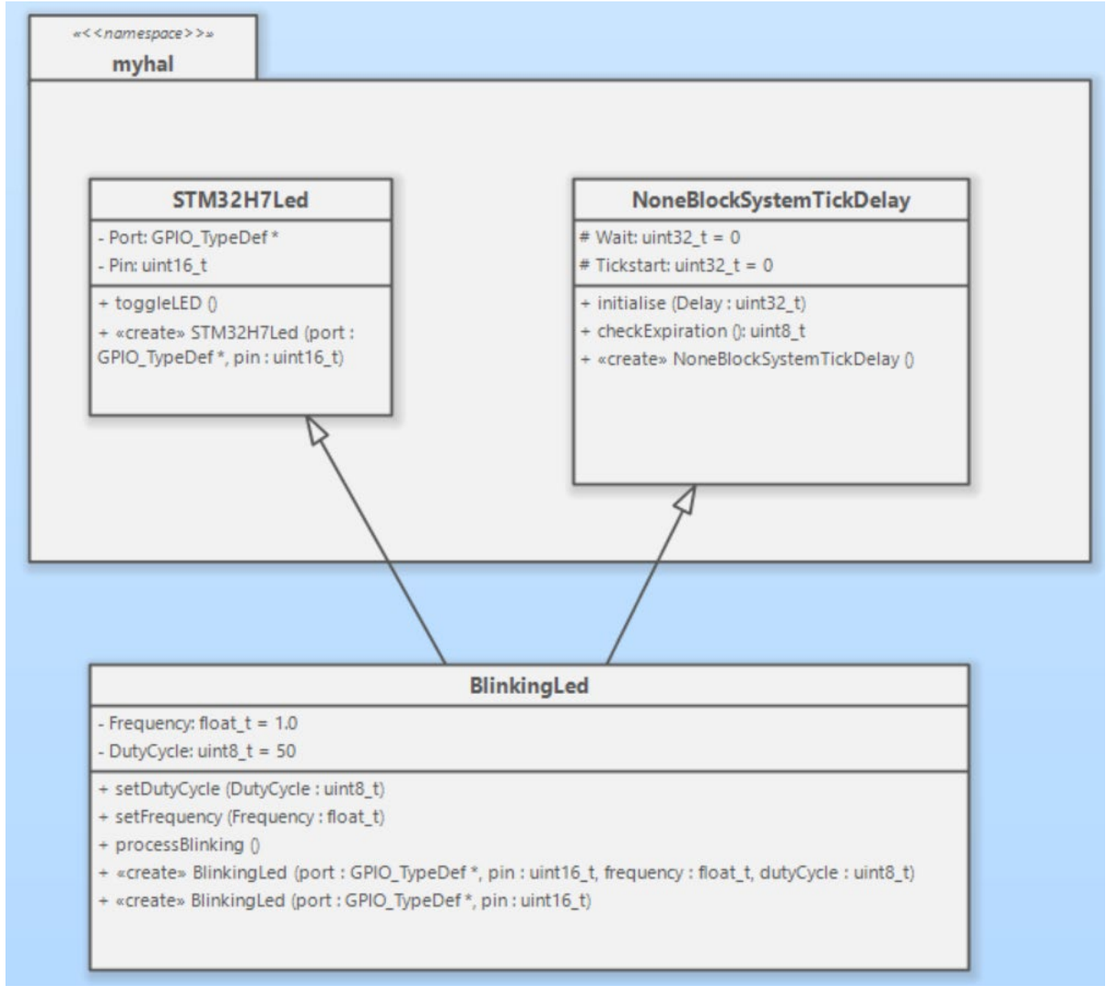


## 02_3_CPP_BlinkyConst

Expand project: CPP_Blinky_02_03.

Try to use:

`Const`, `consteval` and `constexpr` as often as possible on the methods and attributes!



## 02_4_CPP_Template

Create a new project: CPP_Template_02_04.

Create a Array of int_16 with 6 Elements defined randomly.

Create a template function which calculates the average value of an array.

Pass the array to the template function and store the average value in a variable.

## 02_5_CPP_Template

Extend the project: Template _02_04 to CPP_ Template _02_05

Use the sort algorithm of std and sort only the last 3 Elements in the array.

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

## Learning Objectives

- You will get known the data type and command `auto`. → obsolete!
- You will learn how to output data on the serial interface (VCP) in C and C++.
- You will see how to setup the MEMS sensor
- You will learn containers and their iterators.
- You will know the group of sequential containers and look at the representative array.
- You will learn how the memory segments are on the MCU (STM32).
- You will get known the function of the MMU in terms of memory leakage.
- You will see how memory is allocated dynamically in C and C++ and where the problems are without MMU.
- You will know the container vector and his memory behavior..
- You will learn how to write you own memory allocator in C++.

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Command / datatype: auto

## `auto`

The `auto` command has different functions.

Automatic data type

Binding of field(C arrays), a tuple and a structure to names.

Vefore C++11, `auto` was used as a specifier of memory classes.
Since C++11 this is obsolete and invalid!

## `auto` for datatypes

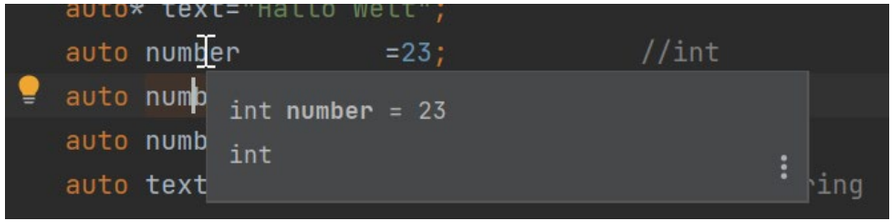
The command `auto` can also be used to let the compiler automatically determine the data type. This means that it can no longer be changed after compilation.

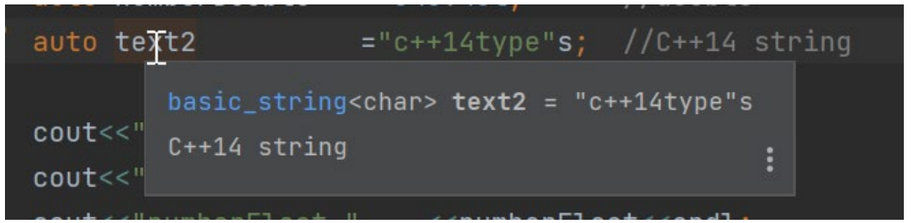
```
auto* text="Hallo Welt";
auto number       =23;          //int
auto numberFloat   = 34.4f;      //float
auto numberDouble  = 345.45l;    //double
auto text2        ="c++14type"s; //C++14 string

cout<<"text "<< text<<endl;
cout<<"number "       <<number<<endl;
cout<<"numberFloat "   <<numberFloat<<endl;
cout<<"numberDouble "  <<numberDouble<<endl;
cout<<"text2 "        <<text2<<endl;
```

Console output

```
text Hallo Welt
number 23
numberFloat 34.4
numberDouble 345.45
text2 c++14type
```

The tool tip provides information on which data type the compiler will assign to the variable.



## `auto` for binding of field(C arrays),

The `auto` command can be used to bind structures.
There are three types: binding a field (C array), a tuple and a structure.
That is, you link a name to an element. You can do this as a copy or as a reference.

```
struct teststruct{
   uint32_t element1;
   uint32_t element2;
} teststruktur;

teststruktur.element1 =1;
teststruktur.element2 =2;

auto [el1, el2] = teststruktur;
auto& [el1ref, el2ref] = teststruktur;

teststruktur.element1 =42;

cout<<(uint32_t)teststruktur.element1<<endl;
cout<<el1<<endl;
cout<<el1ref<<endl;
```

Output of the code on the console:

Binding as a copy is done with :

```
auto [el1, el2] = teststruktur;
```

the binding as reference with the &after the auto:

```
auto& [el1ref, el2ref] = teststruktur;
```

## Cases where `auto` is often used → main use case!

`auto` is used, when the declaration of a datatype/class is redundant:

Pointer:

```
int32_t variable =35;
auto * pointer=&variable;
```

Smart Pointer:

```
unique_ptr<uint32_t>  SmartPointerOnValue(new uint32_t (42));
auto SmartPointerOnValue2 = make_unique<uint32_t>(42);
```

↑ C++17

↳ only type datatype once

Iterators

```
for(auto i:RingBuffer)std::cout<<i <<" ";
std::cout<<std::endl;
```

# Structure of the lessons

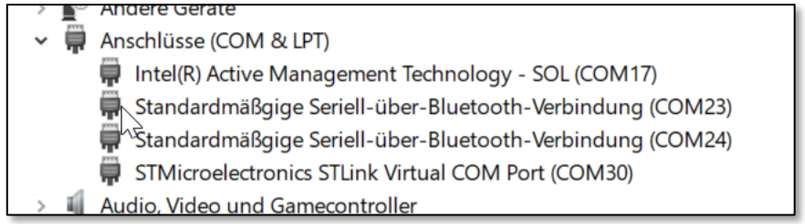
| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Output to console:
# Serial port on NucleoBoard:
# ST-Link: Virtual Communication Port (VCP)

## ST-Link VCP

The ST-Links have a virtual serial port.

This can be observed hen connecting to a PC an checking the drivers.




If the RX and TX Pins are connected to the MCU in the particular Board (Nucleo, Discovery) can only be found out by investigating the boards manual.

ON our Board nucleo STM32H745 it is the case…

UART3 is connected to ST-Link.

## ST-Link VCP as printf solution

In tutorials you can find mostly the approach to replace the printf sub routines through a put or write routine.

I would recommend not to replace the printf methods since it is used for other applications!

```
MX_USART3_UART_Init();

HAL_UART_Transmit(&huart3, "VCP\n",4, 0xFFFF);
```

Then you can use your favourite Console-Terminal: Putty, Tera Term…

```
static void MX_USART3_UART_Init(void)
{

  /* USER CODE BEGIN USART3_Init 0 */

  /* USER CODE END USART3_Init 0 */

  /* USER CODE BEGIN USART3_Init 1 */

  /* USER CODE END USART3_Init 1 */
  huart3.Instance = USART3;
  huart3.Init.BaudRate = 115200;
  huart3.Init.WordLength = UART_WORDLENGTH_8B;
  huart3.Init.StopBits = UART_STOPBITS_1;
  huart3.Init.Parity = UART_PARITY_NONE;
  huart3.Init.Mode = UART_MODE_TX_RX;
  huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart3.Init.OverSampling = UART_OVERSAMPLING_16;
  huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart3.Init.ClockPrescaler = UART_PRESCALER_DIV1;
  huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart3) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetTxFifoThreshold(&huart3, UART_TXFIFO_THRESH
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetRxFifoThreshold(&huart3, UART_RXFIFO_THRESH
  {
    Error_Handler();
  }
  if (HAL_UARTEx_DisableFifoMode(&huart3) != HAL_OK)
  {
    Error_Handler();
  }
```

## Datatypes on the console

C -func

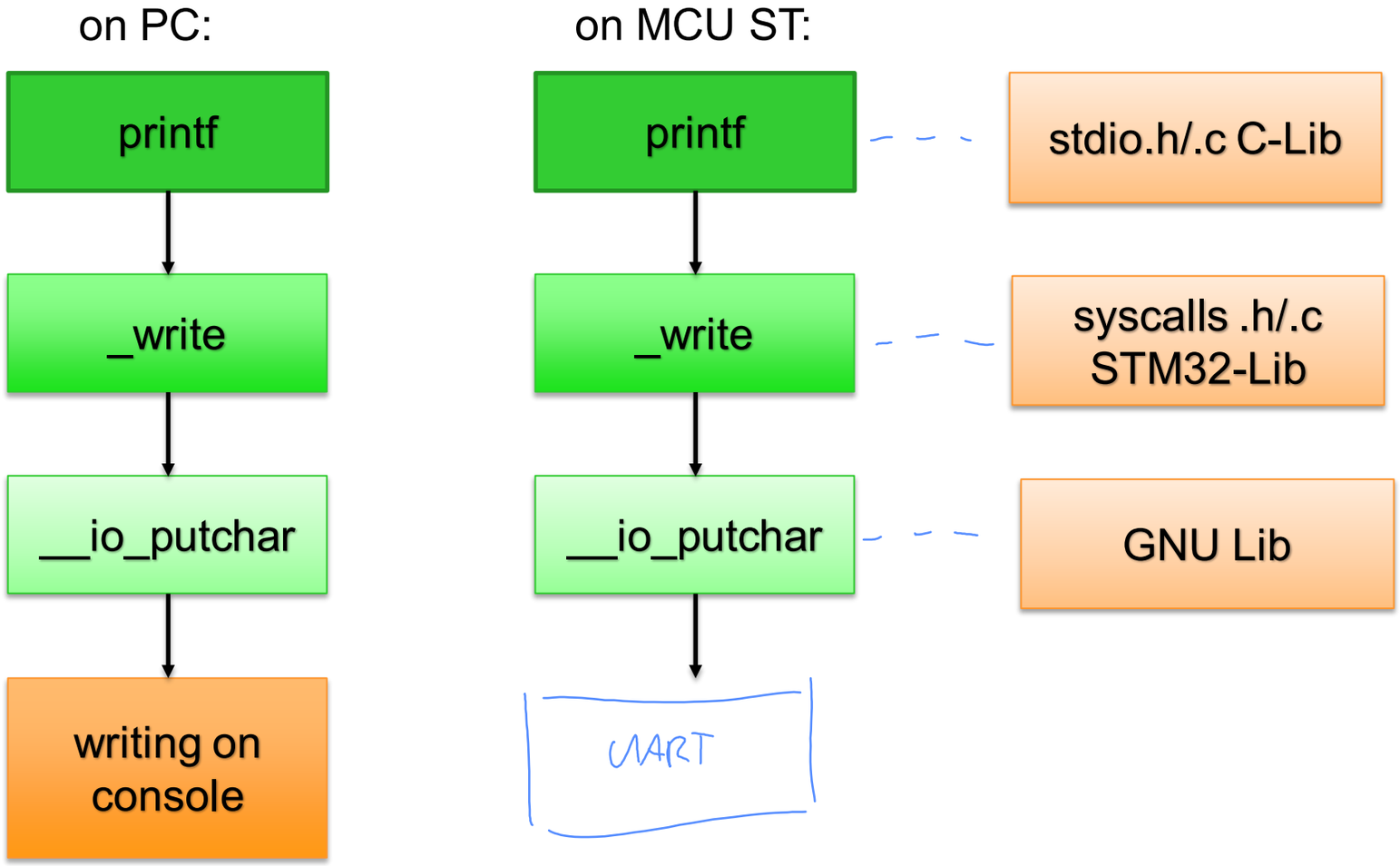
On the PC, we can use printf to do output to console...

What does prinft do again???

C++ variant? → cout

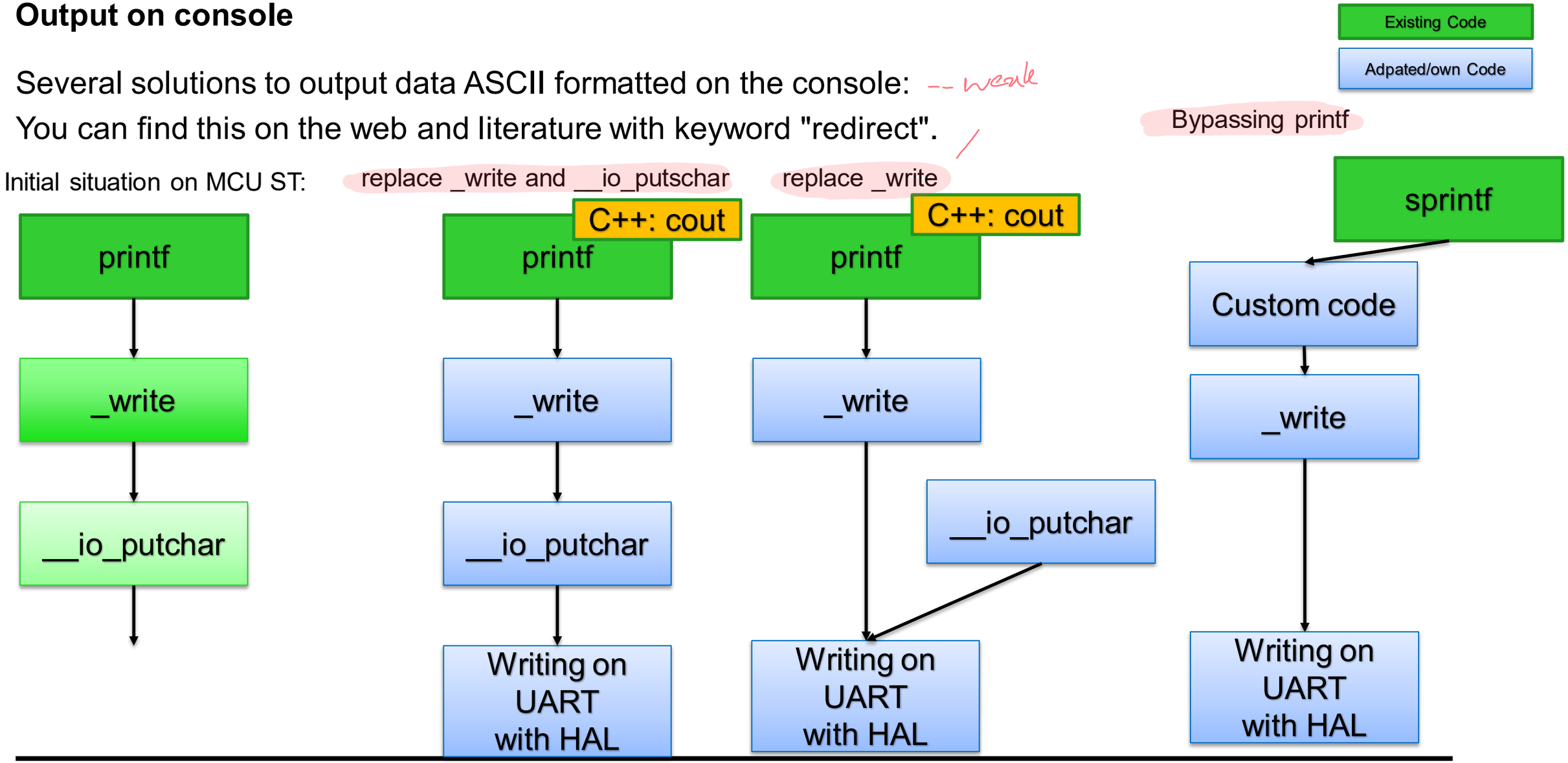
## Output on console

printf uses several functions until the output takes place on the console....



## Output on console

Several solutions to output data ASCII formatted on the console: -- weak
You can find this on the web and literature with keyword "redirect".



## Cout and printf

The C++ Standard Library (<iostream>) is designed to be flexible and efficient, and it typically manages memory efficiently for standard use cases. For simple text and number output, like printing integers, floating-point numbers, or strings, std::cout is a suitable choice, and it doesn't imply a direct use of the heap.

On the other hand there is **no warranty** that the **heap** is **not used by printf** or **cout**!

To make sure this does not happen:

1. Minimize Heap (turn it off)
2. Use custom implementation of printf. For instance:
   https://github.com/mpaland/printf
   https://os.mbed.com/blog/entry/Reducing-memory-usage-with-a-custom-prin/
   (`https://github.com/janongboom/mbed-coremark-lm32-printf`)
   https://mcuoneclipse.com/2014/08/17/xformat-a-lightweight-printf-and-sprintf-alternative/
   (https://github.com/ErichStyger/McuOnEclipse_PEx)

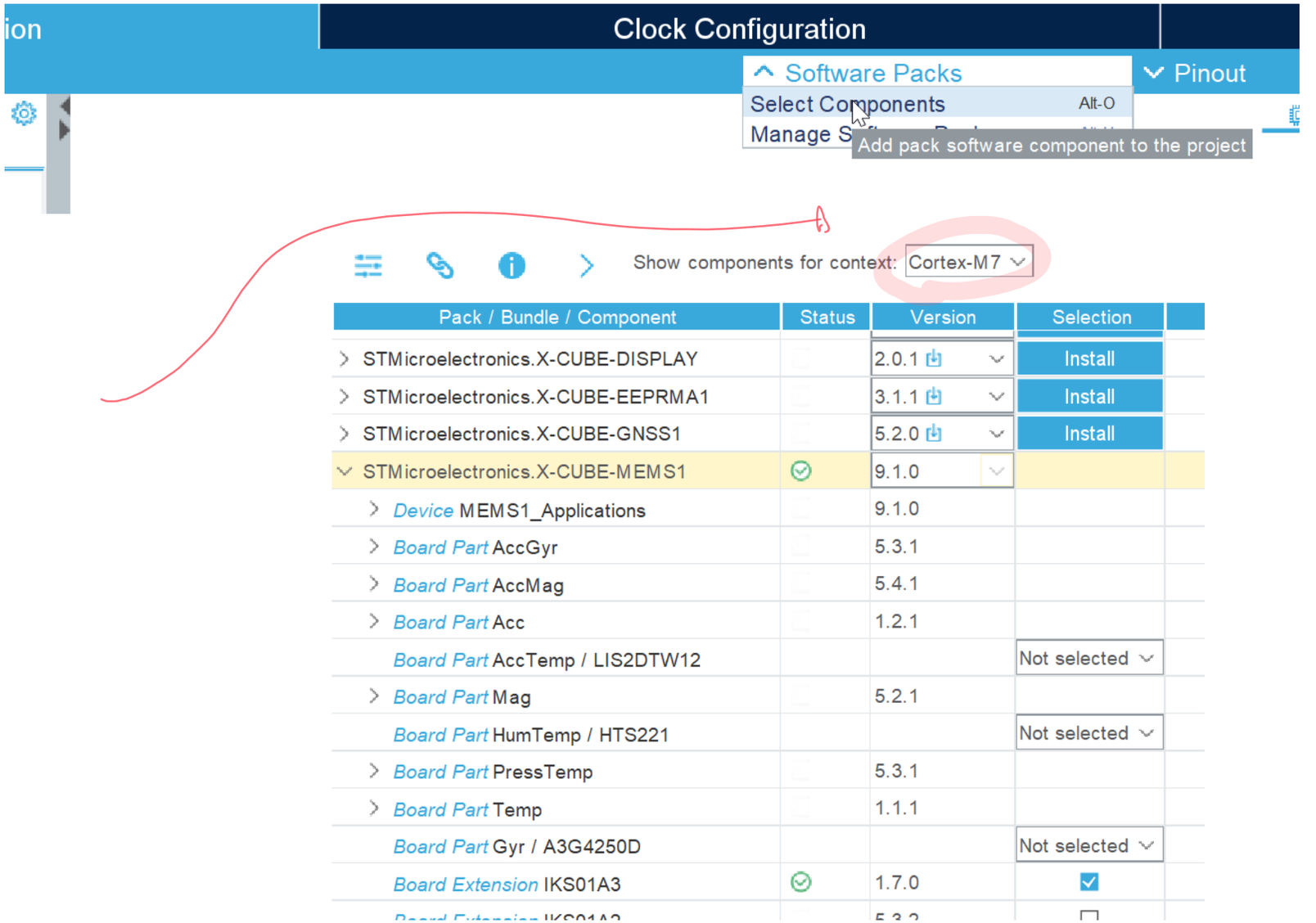
# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# MEMS Software Package für IKS01A3

# New project

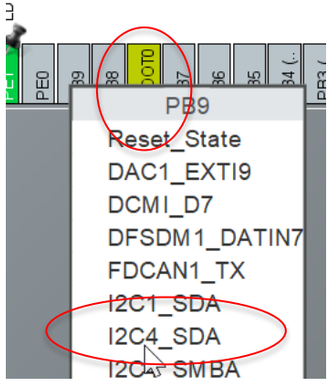
→Configurator

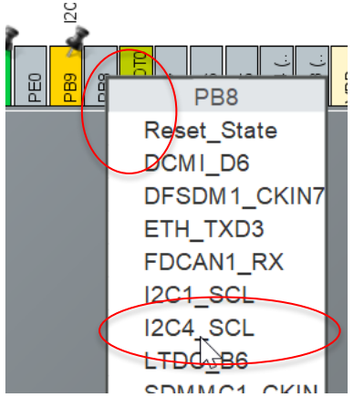


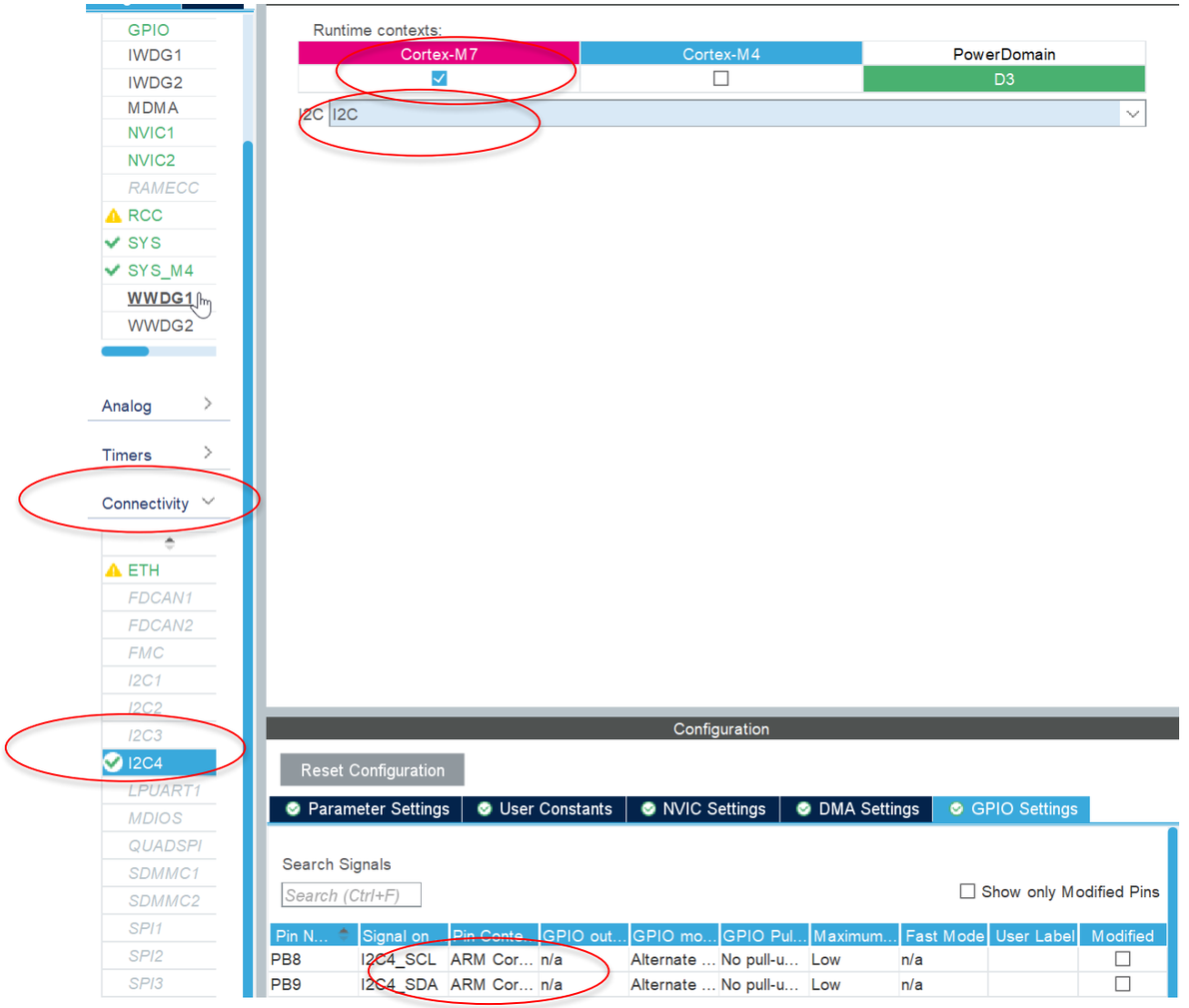

| Pack / Bundle / Component | Status | Version | Selection |
|---|---|---|---|
| STMicroelectronics.X-CUBE-DISPLAY | | 2.0.1 | Install |
| STMicroelectronics.X-CUBE-EEPRMA1 | | 3.1.1 | Install |
| STMicroelectronics.X-CUBE-GNSS1 | | 5.2.0 | Install |
| STMicroelectronics.X-CUBE-MEMS1 | ✓ | 9.1.0 | |
| *Device* MEMS1_Applications | | 9.1.0 | |
| *Board Part* AccGyr | | 5.3.1 | |
| *Board Part* AccMag | | 5.4.1 | |
| *Board Part* Acc | | 1.2.1 | |
| *Board Part* AccTemp / LIS2DTW12 | | | Not selected |
| *Board Part* Mag | | 5.2.1 | |
| *Board Part* HumTemp / HTS221 | | | Not selected |
| *Board Part* PressTemp | | 5.3.1 | |
| *Board Part* Temp | | 1.1.1 | |
| *Board Part* Gyr / A3G4250D | | | Not selected |
| *Board Extension* IKS01A3 | ✓ | 1.7.0 | ☑ |

can't be changed later !!!

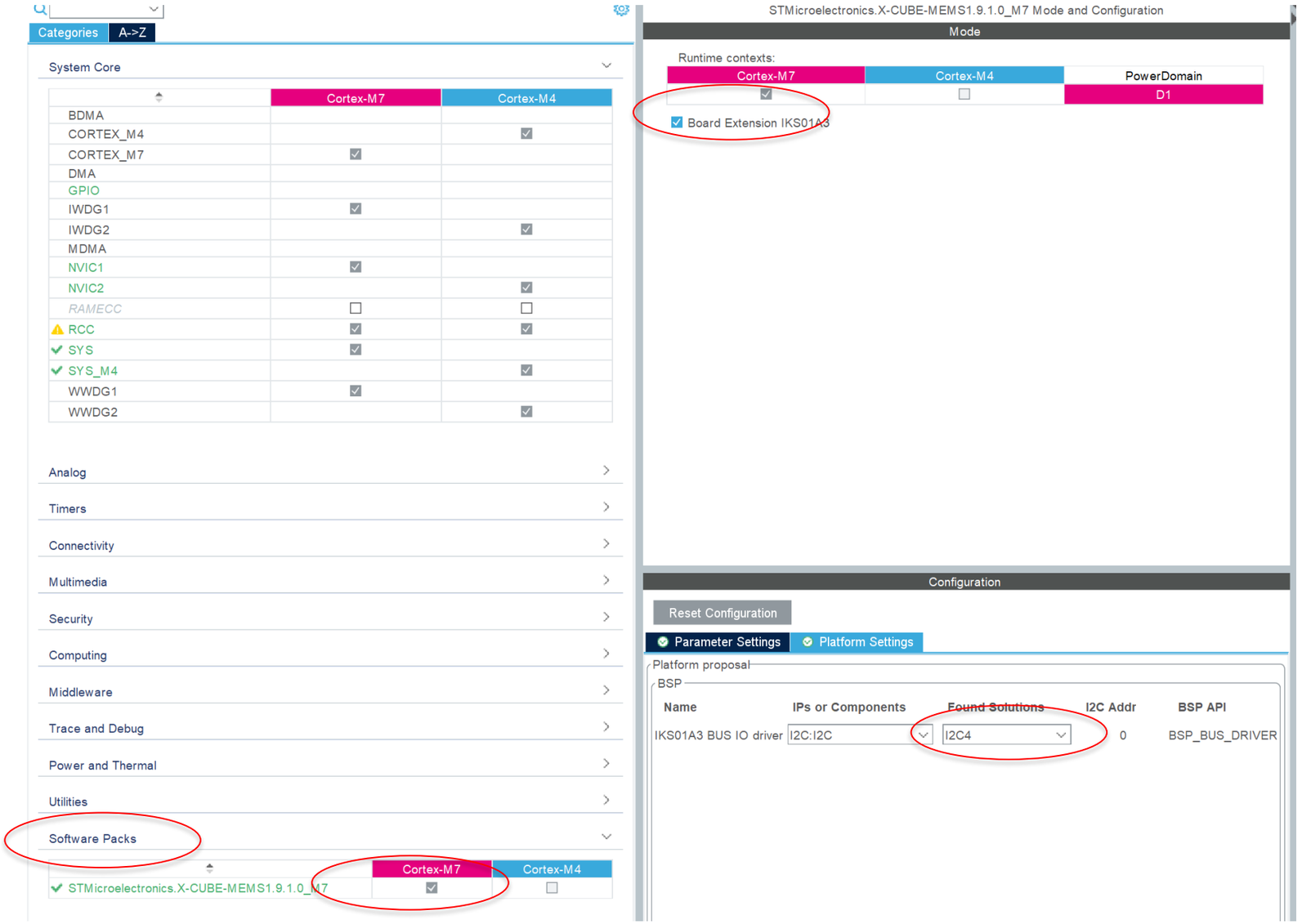
# Setup I2C


PB9
Reset_State
DAC1_EXTI9
DCMI_D7
DFSDM1_DATIN7
FDCAN1_TX
I2C1_SDA
I2C4_SDA
I2C4_SMBA


I2C
PB8
Reset_State
DCMI_D6
DFSDM1_CKIN7
ETH_TXD3
FDCAN1_RX
I2C1_SCL
I2C4_SCL
LTDC_B6


GPIO
IWDG1
IWDG2
MDMA
NVIC1
NVIC2
RAMECC
RCC
SYS
SYS_M4
WWDG1
WWDG2
Analog
Timers
Connectivity
ETH
FDCAN1
FDCAN2
FMC
I2C1
I2C2
I2C3
I2C4
LPUART1
MDIOS
QUADSPI
SDMMC1
SDMMC2
SPI1
SPI2
SPI3
Runtime contexts:
Cortex-M7
Cortex-M4
PowerDomain
D3
I2C
I2C
Configuration
Reset Configuration
Parameter Settings
User Constants
NVIC Settings
DMA Settings
GPIO Settings
Search Signals
Search (Ctrl+F)
Show only Modified Pins
Pin N...
Signal on
Pin Conte...
GPIO out...
GPIO mo...
GPIO Pul...
Maximum...
Fast Mode
User Label
Modified
PB8 I2C4_SCL ARM Cor... n/a Alternate ... No pull-u... Low n/a
PB9 I2C4_SDA ARM Cor... n/a Alternate ... No pull-u... Low n/a

# If STM32H7 is used



Documentation for the software package can be found in the repository of ST

....Users\USERNAME\STM32Cube\Repository\Packs\STMicroelectronics\X-CUBE-MEMS1\9.1.0\Documentation

or similar

The relevant functions are in:

iks01a3_rnv_sensor.c

IKS01A3_ENV_SENSOR_Init

IKS01A3_ENV_SENSOR_GetValue

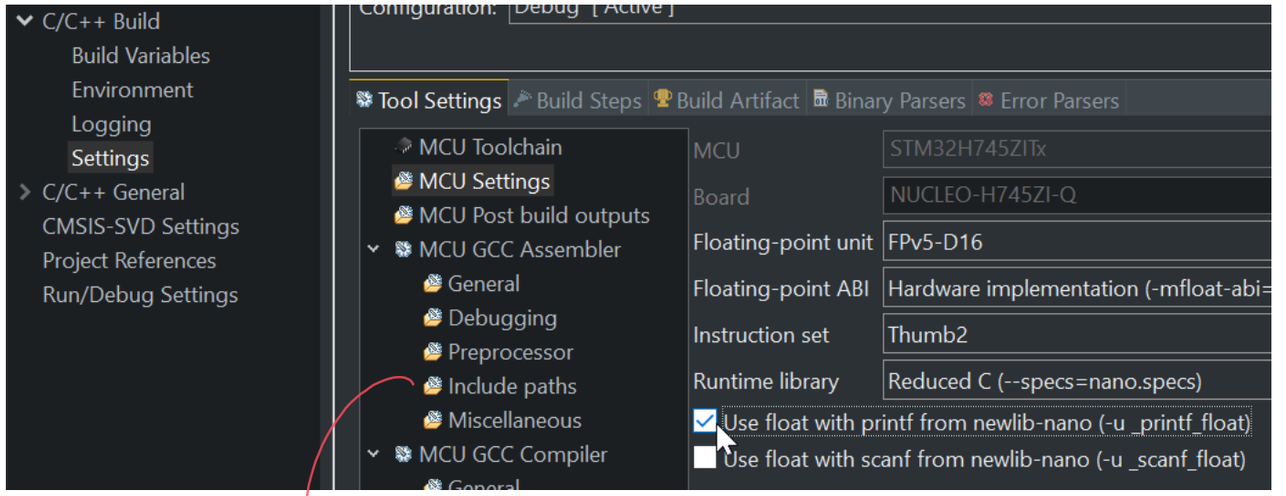
iks01a3_motion_sensors.c

IKS01A3_MOTION_SENSOR_Init

IKS01A3_MOTION_SENSOR_GetAxes

# To be able to use floats with printf or cout

The float formatting support is not enabled, check your MCU Settings from "Project Properties > C/C++ Build > Settings > Tool Settings", or add manually "-u _printf_float" in linker flags.




files from prev. slide in here?

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| **Container** | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Container

## Container


Containers or better the container classes are **data**-containers in C++, with a large extent at methods and automatisms, these count to the STL.
These containers can hold different data types and objects, for this the **container is** implemented as **a template class**.

Standardized operations and functions:

Search, sort, memory management, delete duplicates.

cont.size()

For this, the containers have **iterators**(cursors).

# Container

almost all use heap!

There are 4 groups of containers:

**Sequential containers**

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

**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

**Green = No dynamic memory allocation used!**

# Iterators

## Iterators

Iterators are a kind of "pointer", but for more complex elements.

The term originates from mathematics, from the mathematical method iteration.

An iterator is therefore a navigation aid mostly in containers, since one does not know what elements are in containers.

The syntax of the standard iterators was based on the pointer arithmetic of C.
The operators * and **->** are used to reference the elements.
Other operators like **++** are used to navigate through the elements.

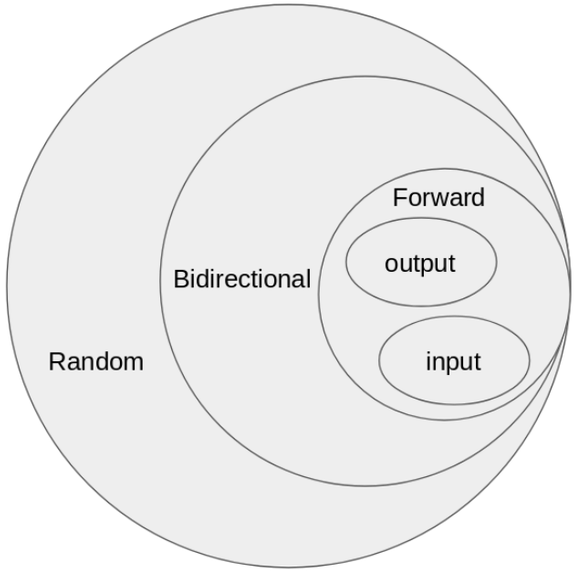
See also:

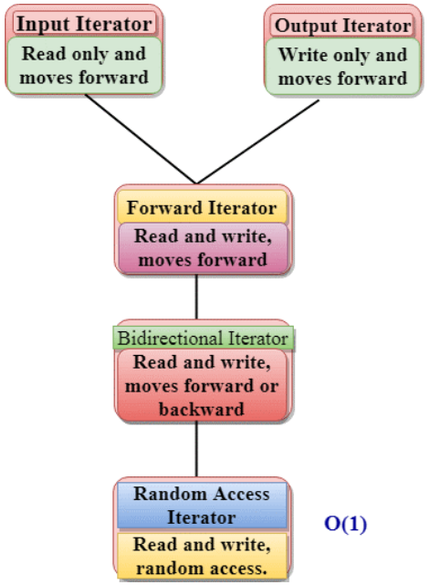
https://en.wikipedia.org/wiki/Iterator

## Container: Iterators

There are differnt iterator types:

- forward iterators (Input und Output)
- bidirectional iterators
- random access iterators




You can think of the iterators as a collection of overloaded operators.
http://www.cplusplus.com/reference/iterator/

Each of the standard container classes has iterator types.

## Container: Iterators

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.

For example, in containers there are 2 variants of iterator each for `begin` and `end`:
`begin`() and `cbegin`(), and `end`(), `cend`().

↳ Readonly

`begin` points to the first element, whereas `end` points to the last element that is no longer present!
Check whether the container is empty or not before addressing iterators.

```
if(!LocalVector2.empty())
   for (auto iterator = LocalVector2.cbegin(); iterator < LocalVector2.cend(); iterator++) {
      std::cout << „Values: " << *iterator<< std::endl;
   }
else std::cout << „no Values" << std::endl;
```

Using an iterator that is empty leads to?

# Contianer & Iterators: Similarities

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.

Examples methods:

- `begin() end()` and `cbegin() cend()` respectively.
  Returns the reference of the iterators from the first or last element. The c variants are `const_interator`, i.e. read-only.
- `size() empty()`
  Returns the number of elements
- `resize() reserve() clear()`
  Enlarges 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() pop_front() pop_back()`
  Insert elements at the end or beginning of sequence-based elements.
- `find() count() contains()`
  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.
Consequently the same methods also apply to string!

# Overview: sequential containers

# Overview: sequential containers

## Overview

| Container | Description |
|---|---|
| array | Fixed size, comparable with C-Array |
| vector | Allrounder, comparable with new Elementtyp[size], but dynamically size adaption: insert and remove at the end very efficient |
| deque | Insert and remove begin and end very efficient |
| list | Very efficient inserting everywhere, no random iterator, forward and backward iterator supported |
| forward_list | Very efficient inserting everywhere, small memory overhead, no random iterator, no size(), forward and backward iterator supported |

## Similarities and differences

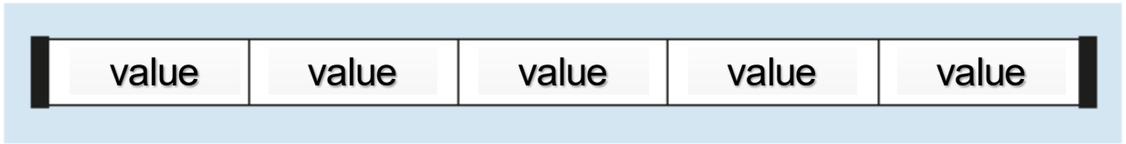
| Eigenschaft | array | vector | deque | list | forward_list |
|---|---|---|---|---|---|
| Dynamic size | - | Yes | Yes | Yes | Yes |
| Keytype | size_t | size_t | size_t | - | - |
| Forward iterator ++ | Yes | Yes | Yes | Yes | Yes |
| Backward iterator -- | Yes | Yes | Yes | Yes | - |
| Overhead per Element | none | special | Very little | Yes | little |
| Efficient inserting | none | end | begin end | everywhere | special |
| Insert location | - | everywhere | everywhere | everywhere | everywhere |
| Splicen | - | - | - | Yes | Yes |
| Memory layout | Stack | one peace | open | fragm. | fragm. |
| Iterators | together | together | free | Bidirect. | forward |
| Algorithms | all | all | all | special | special |

# Container: array

## Container array → No heap usage, since fixed size

An `array` is a sequential container (a superimposed template class).

A fixed number of elements are stored here.
The addition and removal is only possible by overwriting.
The elements lie directly next to each other in the memory.
This makes the `array` suitable for both: a few huge and many tiny element types,
because it has almost no memory overhead.




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

```
#include <array>
```

# Container array: code example

## Code:

```
std::array<int16_t,10> RingBuffer{0,0,0,0,0,0,0,0,0,0};

std::cout<<RingBuffer.size();
std::cout<<std::endl;

int16_t i=0;
for (auto iterator = RingBuffer.begin();iterator<RingBuffer.end(); iterator++) {
   *iterator=i;
   i++;
}
//or
for(auto i:RingBuffer)std::cout<<i <<" ";
std::cout<<std::endl;

std::rotate(RingBuffer.begin(),RingBuffer.begin()+3,RingBuffer.end());
//RingBuffer.at(RingBuffer.end())=6;
for(auto i:RingBuffer)std::cout<<i <<" ";
std::cout<<std::endl;

RingBuffer[5]=20;
for(auto i:RingBuffer)std::cout<<i <<" ";
std::cout<<std::endl;

std::cout<<RingBuffer.size();
std::cout<<std::endl;
//fills array with 2
RingBuffer.fill(2);
for(auto i:RingBuffer)std::cout<<i <<" ";
std::cout<<std::endl;
```

## Console output:

```
10
0 1 2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 0 1 2
3 4 5 6 7 20 9 0 1 2
10
2 2 2 2 2 2 2 2 2 2
```

# DYNAMIC MEMORY ALLOCATION

Up to here no dynamic memory allocation is used or has to be used!

For microcotnrollers dynamic memory allocation is a challenge!

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| **Strategies for using C++ on microcontroller** | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Strategies for using C++ on microcontroller

## 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 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. Consequently, it represents a “circular buffer”. Almost like an MMU. (C++ only)

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| **Memory segments in the working memory** | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Memory segments in the working memory

# Memory segments

**.data**
**Initialised** global variables or static variables. Assigned at the start of the program. The values are read from the program memory for initialisation.

**.bss**
**Uninitialised global variables** or **static variables**. The memory is pre-initialised with 0.

**Heap**
**Dynamically created variables**

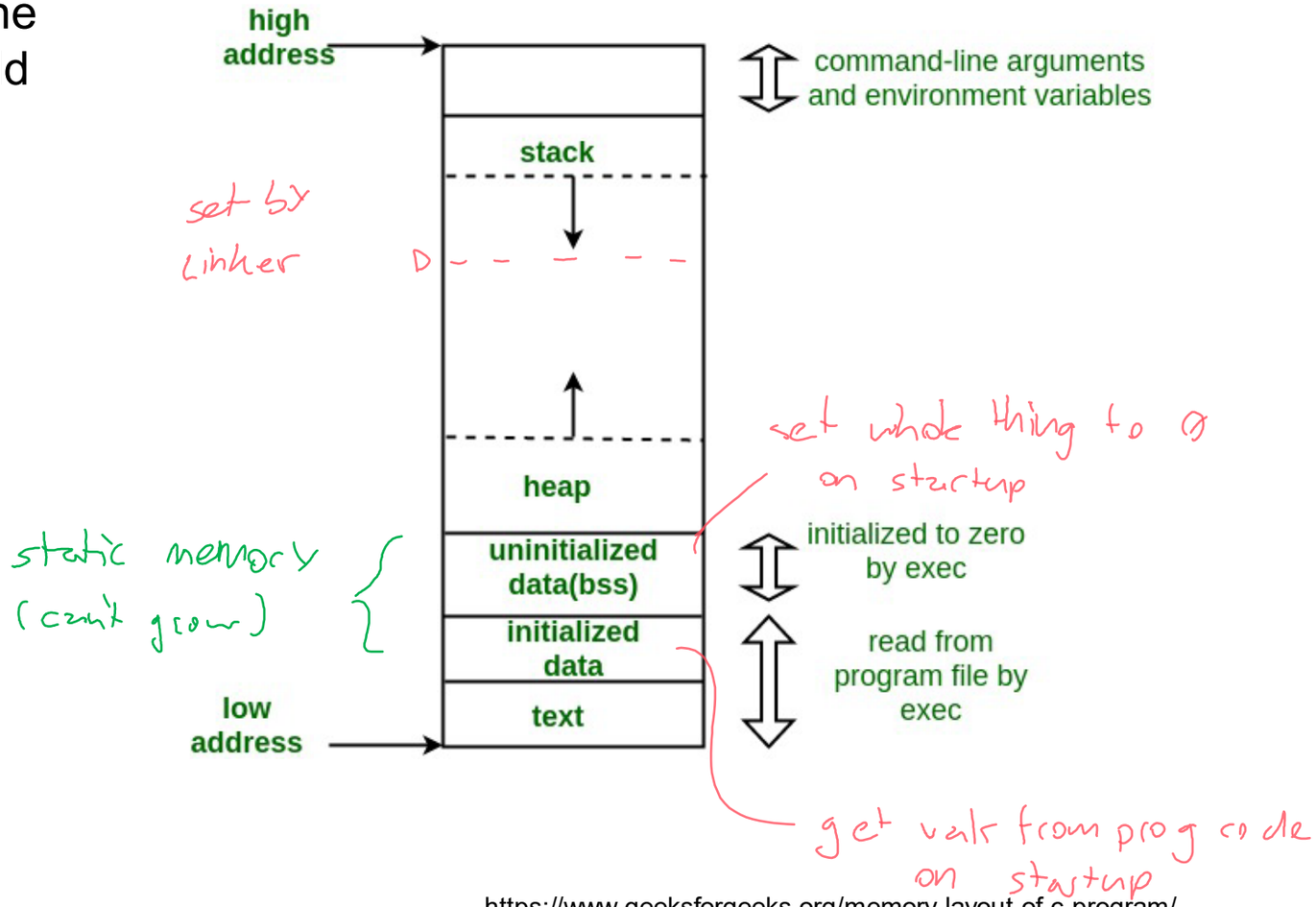
**Stack**
**Local variables**, **grows** on **entering functions** and **shrinks** on **exiting functions**. → GC "garbage collector"

## Memory map of STM32X

For each project the linker/locater generates the code and map according to the linker script: *.ld

debug*.map

- Stack
- Heap
- Data bss
- Data
- text




https://www.geeksforgeeks.org/memory-layout-of-c-program/

# Memory map of heterogeneous controllers

On heterogeneous controllers: for each project, the Linker/Locater generates one map

One map per CPU.

Debugmap

```
.bss            0x0000000024000490     0x1ef4 load address 0x0000000008035634
                0x0000000024000490                _sbss = .
                0x0000000024000490                __bss_start__ = _sbss
 *(COMMON)
                0x0000000024002384                . = ALIGN (0x4)
                0x0000000024002384                _ebss = .
                0x0000000024002384                __bss_end__ = _ebss

._user_heap_stack
                0x0000000024002384      0x604 load address 0x0000000008035634
                0x0000000024002388                . = ALIGN (0x8)
 *fill*         0x0000000024002384        0x4
                [!provide]                        PROVIDE (end = .)
                0x0000000024002388                PROVIDE (_end = .)
                0x0000000024002588                . = (. + _Min_Heap_Size)
 *fill*         0x0000000024002388      0x200
                0x0000000024002988                . = (. + _Min_Stack_Size)
 *fill*         0x0000000024002588      0x400
                0x0000000024002988                . = ALIGN (0x8)
```

**Table 6. Internal memory summary of the STM**

| Memory type | Memory region | Address start | Size |
|---|---|---|---|
| Flash memory | FLASH-1 | 0x0800 0000 | 1 Mbyte(1) |
| | FLASH-2(2) | 0x0810 0000 | 1 Mbyte(1) |
| RAM | DTCM-RAM | 0x2000 0000 | 128 Kbytes |
| | ITCM-RAM | 0x0000 0000 | 64 Kbytes |
| | AXI SRAM | 0x2400 0000 | 512 Kbytes(3) |
| | SRAM1 | 0x3000 0000 | 128 Kbytes(4) |
| | SRAM2 | 0x3002 0000 | 128 Kbytes(4) |
| | SRAM3(5) | 0x3004 0000 | 32 Kbytes |
| | SRAM4 | 0x3800 0000 | 64 Kbytes |
| | Backup SRAM | 0x3880 0000 | 4 Kbytes |

AN4891 STM32H7, Seite 23

# Dynamic memory allocation in C and C++

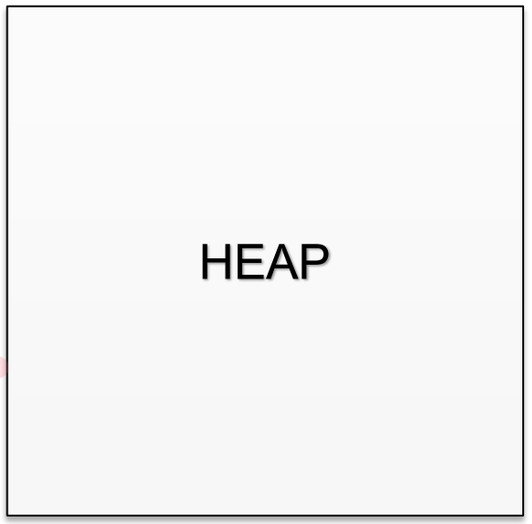
## Dynamic memory allocation

Dynamic memory allocation is when more memory is added to the program while it is running. The reasons why this is done can be different:

- New objects or memory elements are needed at runtime due to user behavior.
- The stack is too small to hold more objects/variables.

- Local variables
- Linear
- Fast
- Fixed size according to compilation
- Allocation and deallocation by compiler


- Global variables, objects in C++
- Hierarchical
- Can fragment
- Allocation by code
- Deallocation through code



# Dynamic memory allocation

| Function / method | Function / method declaration | C | | C++ | |
|---|---|---|---|---|---|
| Creates a memory block in the HEAP memory allocation | `void *malloc(size_t size);` | C | <stdlib.h> | C++ | **#include <cstdlib>** std::malloc |
| Creates a memory block in HEAP initialises the content with 0. | `void *calloc(size_t num, size_t size );` | C | <stdlib.h> | C++ | #include <cstdlib> std::calloc |
| Adjusts the size of the memory block in HEAP, which must first be created with malloc or calloc. | `void *realloc(void *ptr, size_t new_size );` | C | <stdlib.h> | C++ | #include <cstdlib> std::realloc |
| Aligns the existing memory block to the given address. | `void *aligned_alloc(size_t alignment, size_t size );` | C11 | <stdlib.h> | C++17 | #include <cstdlib> std::calloc |
| Releases memory block in HEAP, deletes it. | `void free( void* ptr );` | C | | C++ | #include <cstdlib> std::free |
| Create object in HEAP | `new` | | | C++ | |
| Delete object in HEAP | `delete` | | | C++ | |

## Dynamic memory allocation: malloc in C

The function malloc is passed the memory size you want to create in bytes.

```
void *malloc(size_t size);
```

You get back a void pointer.

When creating the memory, it is advisable to multiply the number of variables by the data type size.

```
PointerOnMemorySegment = (uint32_t*) malloc(1024*sizeof (uint32_t));
```

You get back a pointer that you should cast back to the pointer data type.

Each memory allocation must be released manually by the code, before the end of the program!

This is done with the function free and the pointer to the memory area.

```
free(PointerOnMemorySegment);
```

## Dynamic memory allocation: malloc in C++

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!

C++

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

C++

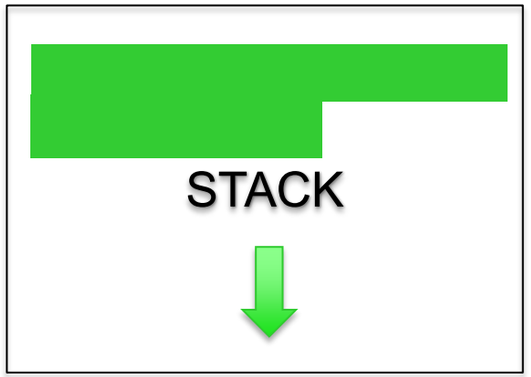
Initialisation with a value can take place in the definition in a subsequent bracket:

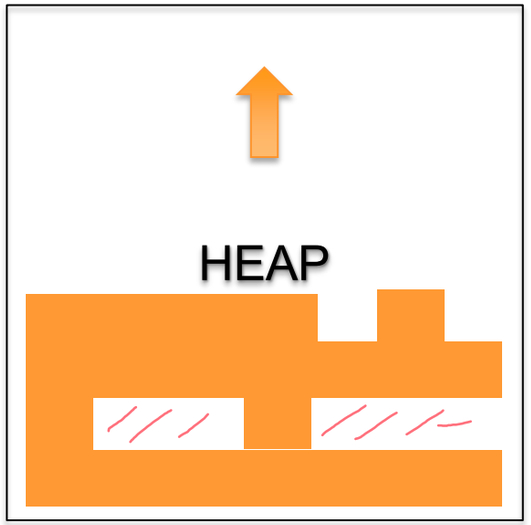
```
uint8_t * OneByte = new uint8_t(5);
```

# Dynamic memory allocation: the issues

The following situations can appear:

- Stack overflow
- Heap Overflow
- Heap memory leak → more allocated than deleted
- Heap fragmentation




fragmentation

Example: project map-file CM7:

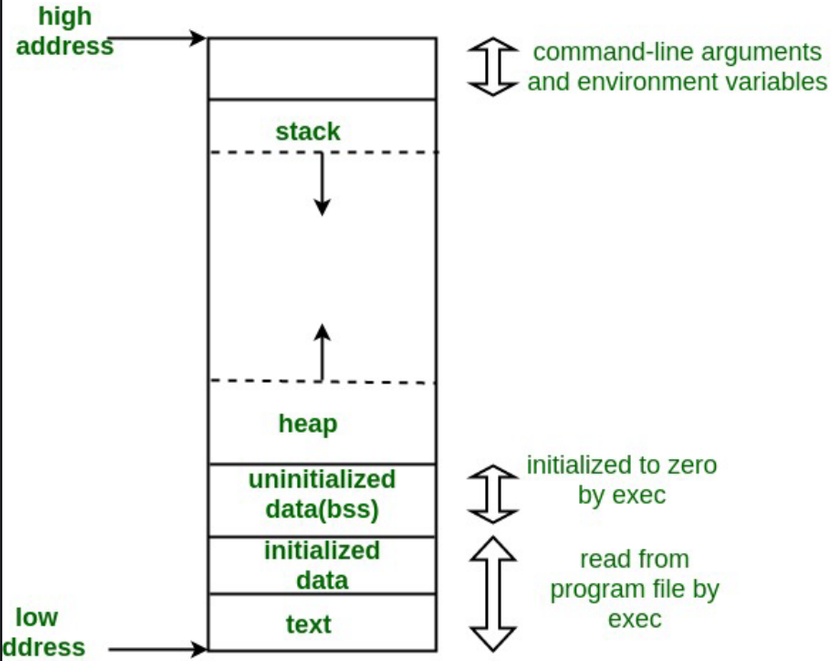
```
user_heap_stack
                0x0000000024002384      0x604 load address 0x0000000008035634
                0x0000000024002388                . = ALIGN (0x8)
 *fill*         0x0000000024002384        0x4
                [!provide]                        PROVIDE (end = .)
                0x0000000024002388                PROVIDE (_end = .)
                0x0000000024002588                . = (. + _Min_Heap_Size)
 *fill*         0x0000000024002388      0x200
                0x0000000024002988                . = (. + _Min_Stack_Size)
 *fill*         0x0000000024002588      0x400
                0x0000000024002988                . = ALIGN (0x8)
```

## Memory allocation in C++

In Summary we have 3 memory segments which can be used to instantiate an object/variable:

1. «RAM segment» alias «data segment» alias «static RAM»
2. Stack
3. Heap

```
modul2 RAMSegment_Obj;

int main() {

  modul2 Stackt_Obj;

  modul2 *Heap_Obj = new modul2;

   std::cout << "static RAM: " << RAMSegment_Obj.getAttribute() << std::endl;
   std::cout << "Stack RAM: " << Stackt_Obj.getAttribute() << std::endl;
   std::cout << "HEAP RAM: " << Heap_Obj->getAttribute() << std::endl;
   delete Heap_Obj;
   return 0;
}
```




This statement is true, as long as no other memory allocation is used in the class!

PC - CPUs

MCU don't have MMU!

Memory Management Unit

## How does it look a CPU(s) .... with MMU

A Memory Management Unit (MMU) is located between the CPU and the physical memory.

The MMU has several functions.

The most important is translating from: physical to logical address of the memory.

Consequently, the logical addresses are virtual addresses (virtual memory).

The memory is segmented into frames.

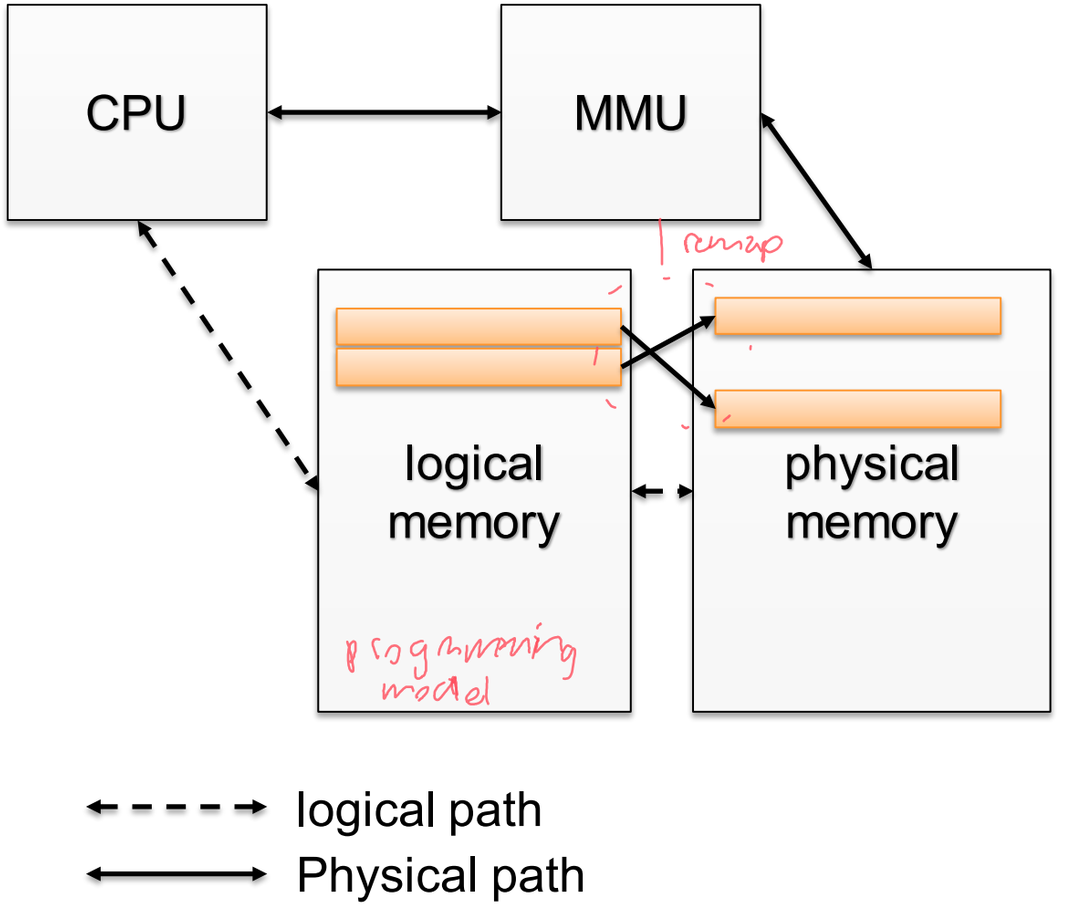
The frames are presented to the CPU as contiguous, however they cannot be contiguous in the physical memory.

During memory allocation and deallocation, the memory is allocated or released on a frame-by-frame basis.

The frame size is always a multiple of 2^X.

For PC 32Bit architecture typically 4kBytes

Consequently: no fragmentation, as MMU solves this.




More information: Betriebssysteme kompakt, Ch. Baun 2019 Springer S. 98
https://en.wikipedia.org/wiki/Memory_management_unit
https://blogs.sw.siemens.com/embedded-software/2009/09/28/mmu-and-heap-contiguity/

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| **Dynamic memory allocation in C and C++ in detail** | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Dynamic memory allocation in C and C++ in detail

## Dynamic Memory Allocation in C++, Tasks

Allocation in C++ is done by using new.

Depending on the Compiler and the implementation of it, it will use malloc (C-Variant) or another implementation of allocating memory.

→ New uses malloc over a wrapper or an own implementation.

Task of a dynamic memory allocator:

1. Take the request to allocate memory
2. Search the heap for a suitable place to allocate memory
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.

Step 1 and 3 are straight forward.

Let us examine Step 2 and 4.

## Dynamic Memory Allocation in C++: Memory-List

Step 2 and 4 are depending on the implementation of the compiler!

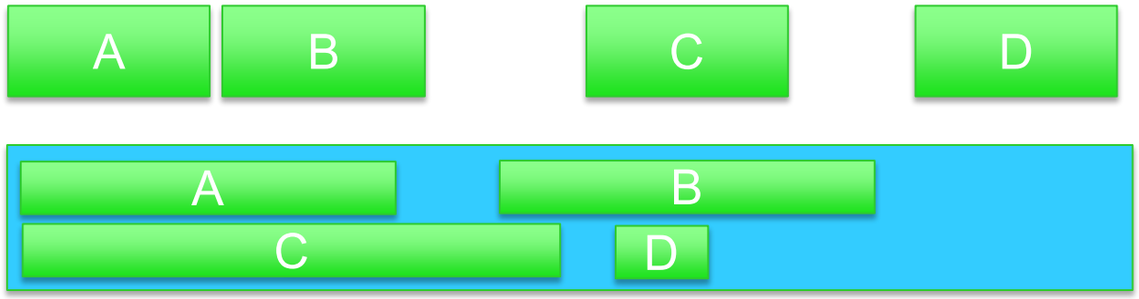
Here is a general explanation how it is often done.

Task 2: Search the heap for a suitable place to allocate memory

This is the task of sometimes called: dynamic memory allocator manager (DMAM)

The manager keeps the information with Bytes are used and which are free in the heap.

This bookkeeping is often done with a linked list the "**Free list**". Since a linked list can be dynamically be enhanced or reduced.




```
// List of free memory blocks
struct FreeMemoryBlock {
   void* startAddress;
   size_t size;
};

std::list<FreeMemoryBlock> freeMemoryBlocks;
```

## Dynamic Memory Allocation in C++: Search algorithms

There are different strategies for finding available chunk of memory, for a given request using free list.
The simple and standard algorithms are:

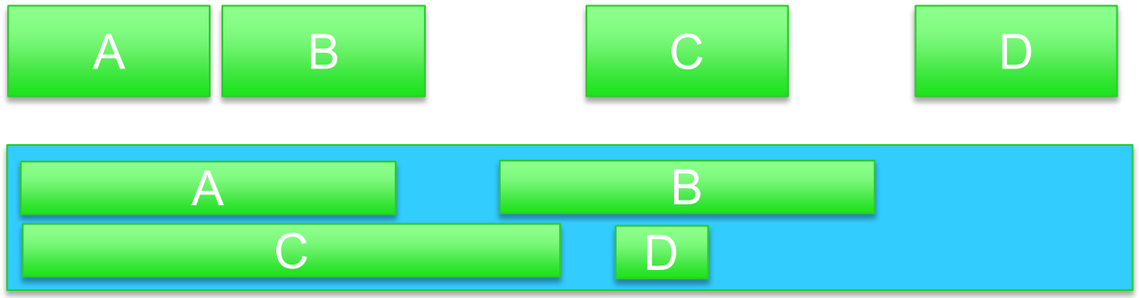
1. **First Fit:** use the first block in the list which fits → might use large gap for small data element

2. **Best Fit:** use the block in the list whose size is closest to the requested size.

3. **Use the most/least recently free-d block** (LIFO, FIFO).

RT perspective:

Time to find spot not predictable!



## Dynamic Memory Allocation in C++: free concepts

There are different strategies for free memory and to update the “free list”:

1. Add to the front of the list. (When used with first-fit, this results in LIFO-like behavior).
2. Add to the back of the list. (FIFO)
3. Insert it into the list so that the list is always ordered by address. This is most common, as the structure of the list mirrors the structure of the memory layout.

## Dynamic Memory Allocation: fragmentation

Fragmentation refers to the tendency of the heap to go from one large available block at the beginning to many small available blocks. It may be possible that, while the total amount of memory is larger than the amount requested, no single block is big enough to satisfy it.

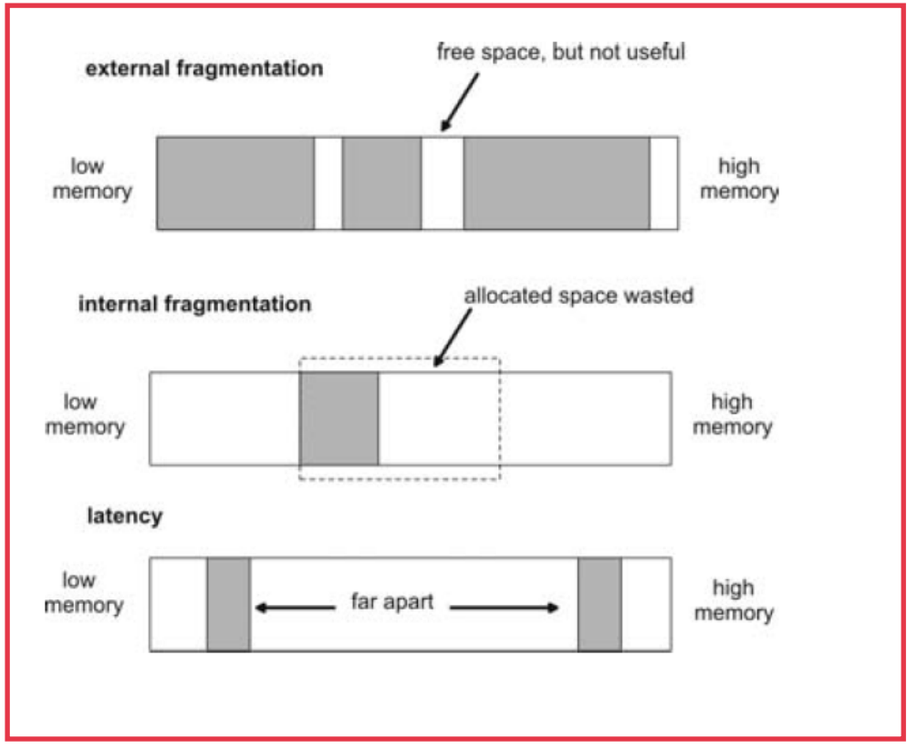
If there are theoretically more bytes available than can be allocated, then this is called **external fragmentation**.

A memory allocator can nothing do against external fragmentation, since moving blocks would mean to redirect all the according pointer.

On the other side **internal fragmentation** occurs when an allocator reserves more space per-block than is actually requested. → why would this happen?

can't really manage this

Far placement can lead to higher latency.



# Dynamic Memory Allocation in C++: alternative Memory-List

Instead of circular list based allocator two alternate approaches can be used:

**1. Slab allocator (GNU libc) :**

- Divide blocks into small and large. Only block > threshold size managed using free list. Else, allocate power of two and use “bitmap” for each range of blocks of same size, for bookkeeping.
- Fast for small blocks, slower for large blocks. Minimal space overhead. No external fragmentation (as it’s a predefined block for small block allocations, but has wasted space)

**2. Buddy system (Linux kernel)**

- Similar to slab allocator but only allocate blocks in sizes that are power of 2. Keep separate free lists for 16 byte, 32 byte and 64 byte blocks etc.
- If no free block of size n available, find block of size 2n and split it into two blocks of size n.

## Conclusion

How dynamic memory allocation is implemented in detail is compiler specific.

The available memory is tracked by the dynamic memory allocator manager.

There are different algorithms to search and free memory.

→
Real time applications should not use the heap memory.
Since allocating, freeing and accessing the memory will lead to none deterministic behaviour!

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| **Container: vector** | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Overview: sequential containers

# Overview: sequential containers

## Overview

| Container | Description |
|---|---|
| array | Fixed size, comparable with C-Array |
| vector | Allrounder, comparable with new Elementtyp[size], but dynamically size adaption: insert and remove at the end very efficient |
| deque | Insert and remove begin and end very efficient |
| list | Very efficient inserting everywhere, no random iterator, forward and backward iterator supported |
| forward_list | Very efficient inserting everywhere, small memory overhead, no random iterator, no size(), forward and backward iterator supported |

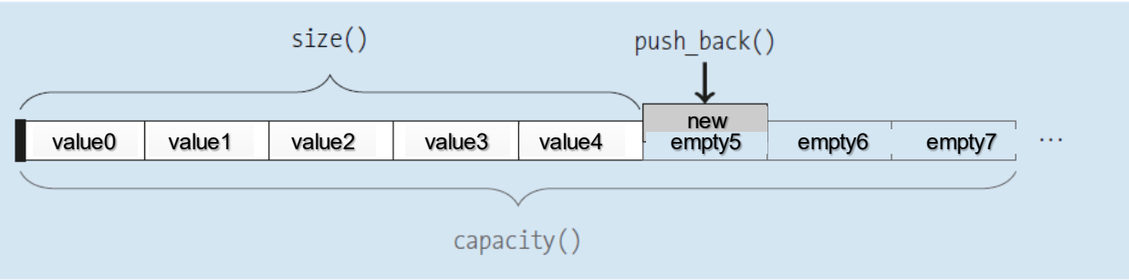
## Similarities and differences

| Eigenschaft | array | vector | deque | list | forward_list |
|---|---|---|---|---|---|
| Dynamic Size | - | Yes | Yes | Yes | Yes |
| Keytype | size_t | size_t | size_t | - | - |
| Forward iterator | Yes | Yes | Yes | Yes | Yes |
| Backward iterator | Yes | Yes | Yes | Yes | - |
| Overhead per Element | none | special | Very little | Yes | little |
| Efficient inserting | none | end | begin end | everywhere | special |
| Insert location | - | everywhere | everywhere | everywhere | everywhere |
| Splicen | - | - | - | Yes | Yes |
| Memory layout | Stack | one peace | open | fragm. | fragm. |
| Iterators | together | together | free | Bidirect. | forward |
| Algorithms | all | all | all | special | special |

# Container: vector

## Container: vector

A vector is a sequential container (a superimposed template class).
Has strong similarities to array lists in Java.
This all-rounder automatically grows. You can insert anywhere, however only efficiently
at the back end. The elements are read forwards in sequence,
backwards or randomly by number index. The vector also keeps its elements
directly next to each other in memory, which makes it practical for a few huge and many tiny elements.
Optimally used, it has almost no memory overhead, in the non-optimal case still acceptable as it does not fragment the memory.




With the container, you always have to include the corresponding header file!

```
#include <vector>
```

extendible → HEAP!

## Container: vector

```
std::vector<uint16_t> LocalVector2;
std::vector<std::string> LocalStringVector;
std::vector<ClassA> LocalClassVector;

LocalVector.push_back(34);
```

```
auto LocalVector3 = new std::vector<uint16_t>;
```

The iterator is used for example in FOR loops.

```
for (auto iterator = LocalVector.cbegin(); iterator < LocalVector.cend(); iterator++) {
   std::cout << „values: " << *iterator<< std::endl;
}
for (int i = 0; i < LocalVector.size(); i++) {                    //Alternative, but without Iterator object
   std::cout << „values: " << LocalVector[i]<< std::endl;
}
for (auto & it : LocalVector) {                                   //Alternative, range based
   std::cout << „values: " << it<< std::endl;
}
```

## Container: vector and HEAP

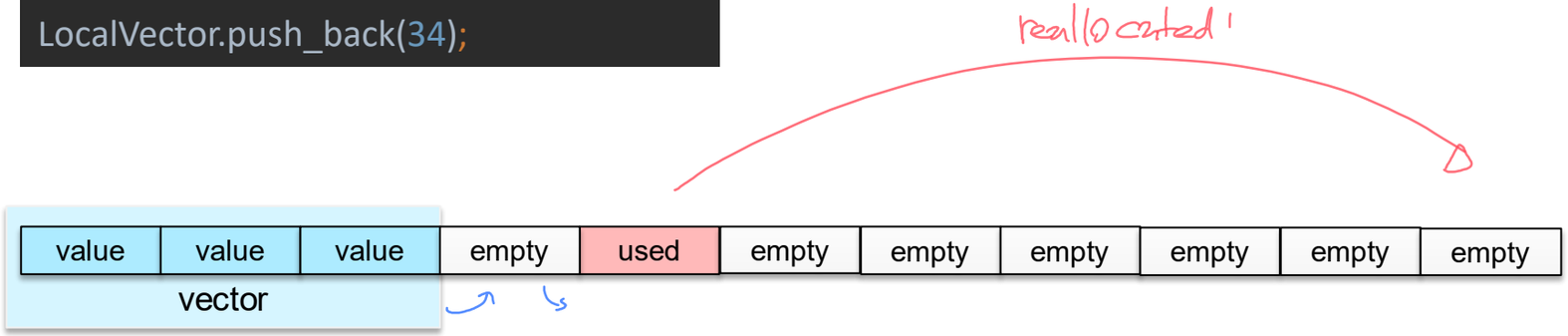
Whether you create a vector with new or without makes only a small difference.

It only affects the initial overhead of a vector object. The content of the vector is always created in the HEAP as long as the standard memory allocator is used.

This is because at compile time it is not known how many objects will exist.

```
std::vector<uint16_t> LocalVector2;
std::vector<std::string> LocalStringVector;
std::vector<ClassA> LocalClassVector;

LocalVector.push_back(34);
```

```
auto LocalVector3 = new std::vector<uint16_t>;
```



# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| **Custom memory Allocator** | |
| Self study | |

# Custom memory Allocator

## Custom memory Allocator

C and C++ delivers high freedom how the developer can control the language elements.

The memory allocator can be programmed by the developer selve. However, this will primarly lead to not using the heap.

Many classes (for instance of the stdlib) have superimposed methods where you can specifiy your own memory allocators.

# Ring Allocator according to CH Kormanyos

Doesn't use heap but data

Ch Book

Header:

```
class ring_allocator_base
{
public:
  typedef std::size_t size_type;

protected:
  ring_allocator_base() { }

  // The ring_allocator's buffer size.
  static constexpr size_type buffer_size = 64U;

  // The ring_allocator's memory allocation.
  static void* do_allocate(const size_type);
};
```

Note: for the source code on GitHub:
16 bytes is the default size.

C++:

```
void*
ring_allocator_base::do_allocate(const size_type size)
{
  // Define a static buffer and memory pointer.
  static std::uint8_t  buffer[buffer_size];
  static std::uint8_t* get_ptr = buffer;

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

  // Increment the pointer for the next allocation.
  get_ptr += size;

  // Does this allocation overflow the top
  // of the buffer?
  const bool is_wrap =
    (get_ptr >= (buffer + buffer_size));

  if(is_wrap)
  {
    // Here, the allocation overflows the top
    // of the buffer. Reset the allocated pointer
    // to the bottom of the buffer and increment
    // the next get-pointer accordingly.
    p       = buffer;
    get_ptr = buffer + size;
  }

  return static_cast<void*>(p);
}
```

# Ring Allocator according to CH Kormanyos

The base class is "wrapped" with a template class.

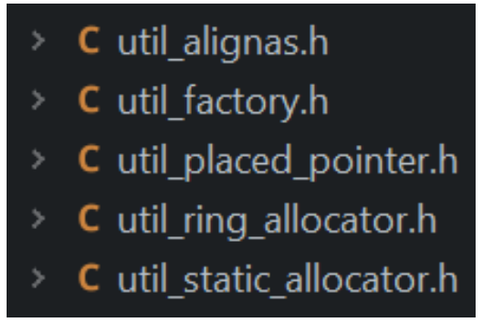
This is how `ring_allocator` is used.

As template class

```
template<typename T>
class ring_allocator : public ring_allocator_base
{
public:
  // ...

  size_type max_size() const noexcept
  {
    // The max. size is based on the buffer size.
    return buffer_size / sizeof(value_type);
  }

  pointer allocate(size_type count,
    ring_allocator<void>::const_pointer = nullptr)
  {
    // Use the base class ring allocation mechanism.
    void* p = do_allocate(count * sizeof(value_type));

    return static_cast<pointer>(p);
  }
```

## Ring Allocator according to CH Kormanyos

We need the following files:




**Important**: the class you instantiate must have a **memory allocator interface**.
Here with the example of a vector container:

```
std::vector<int32_t , util::ring_allocator<uint16_t>>  BufferYAxis2;
```

Or with typedefs according to examples (see also in the book):

```
typedef util::ring_allocator<uint16_t> alloc_typeint32_t;
typedef std::vector<int32_t , alloc_typeint32_t> Average_type;

Average_type BufferYAxis;
```

## Solution in C++ if there is too little in ring buffer segment.

The implementation always uses the memory from the ring buffer.

If the buffer is too small, the program counter will end in the _exit function!

```
59 void _exit (int status)
60 {
61     _kill(status, -1);
62     while (1) {}        /* Make sure we hang here */
63 }
```

This function is located in syscalls.c.

# Example of effficiency from STL to C++ or even C code

## STL-Bibliothek:

```
// chapter06_17-001_use_the_stl.cpp

#include <numeric>

std::uint8_t checksum(const std::uint8_t* p,
                      const std::uint_fast8_t len)
{
  // Use the STL's version of accumulate.
  return std::accumulate(p,
                         p + len,
                         std::uint_fast8_t(0U));
}
```

## Reverse Engineered STL:

```
// chapter06_17-002_use_the_stl.cpp

std::uint8_t checksum(const std::uint8_t* p,
                      const std::uint_fast8_t len)
{
  // Compare with a manually written accumulate.
  std::uint_fast8_t   sum = UINT8_C(0);
  const std::uint8_t* end = p + len;

  while(p != end)
  {
    sum += *p;
    ++p;
  };

  return sum;
}
```

## Entwickler-Code

```
// chapter06_10-002_checksum_uint_fast8_t.cpp

std::uint8_t checksum(const std::uint8_t* p,
                      const std::uint_fast8_t len)
{
  std::uint_fast8_t sum = UINT8_C(0);

  for(std::uint_fast8_t i = UINT8_C(0); i < len; i++)
  {
    sum += *p;
    ++p;
  };

  return sum;
}
```

The STL was more efficient than developer code.
Reverse engineered (trial and error) code.
Note: Use of const
libraries can be more efficient than own implementation.

Mehr Infos: Real Time C++, Ch. Kormanyos 2021 Springer S. 154, 168

## What we have learned

- We have covered the data type and command `auto`.

- We have discussed how to output data on the serial interface in C and C++.

- We have discussed how to setup the MEMS sensor

- We have covered containers and their iterators.
- We got to know the group of sequential containers and looked at the representative array.

- We have discussed how the memory segments are on the MCU (STM32).
- We looked at the function of the MMU in terms of memory leakage.
- We discussed how memory is allocated dynamically in C and C++ and where the problems are without MMU.

- We have covered the container vector and his memory behavior.

- We have discussed the solution of how to write you own memory allocator in C++.

# Structure of the lessons

| | |
|---|---|
| Self study from last time | |
| Learning Objectives | |
| Command / datatype: auto | |
| console printing | • Output to console:<br>• Serial port on NucleoBoard:<br>• ST-Link: Virtual Communication Port (VCP) |
| IKS01A3 Software | • MEMS Software Package für IKS01A3 |
| Container | • Container<br>• Iterators<br>• Overview: sequential containers<br>• Container: array |
| Strategies for using C++ on microcontroller | |
| Memory segments in the working memory | • Dynamic memory allocation in C and C++ |
| Dynamic memory allocation in C and C++ in detail | |
| Container: vector | • Overview: sequential containers<br>• Container: vector |
| Custom memory Allocator | |
| Self study | |

# Self study

## 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;

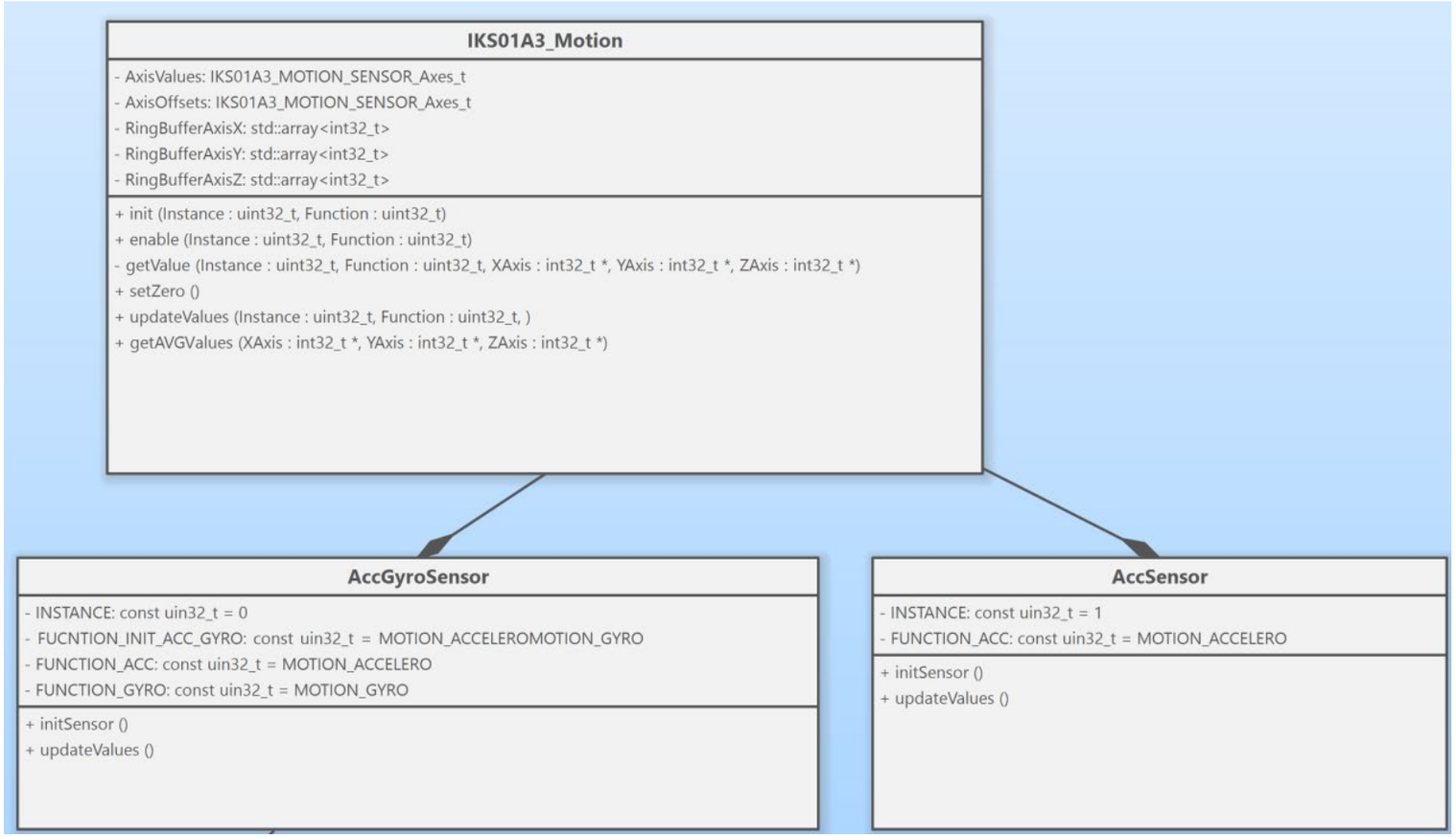
Print the value on the console.

## 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

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.

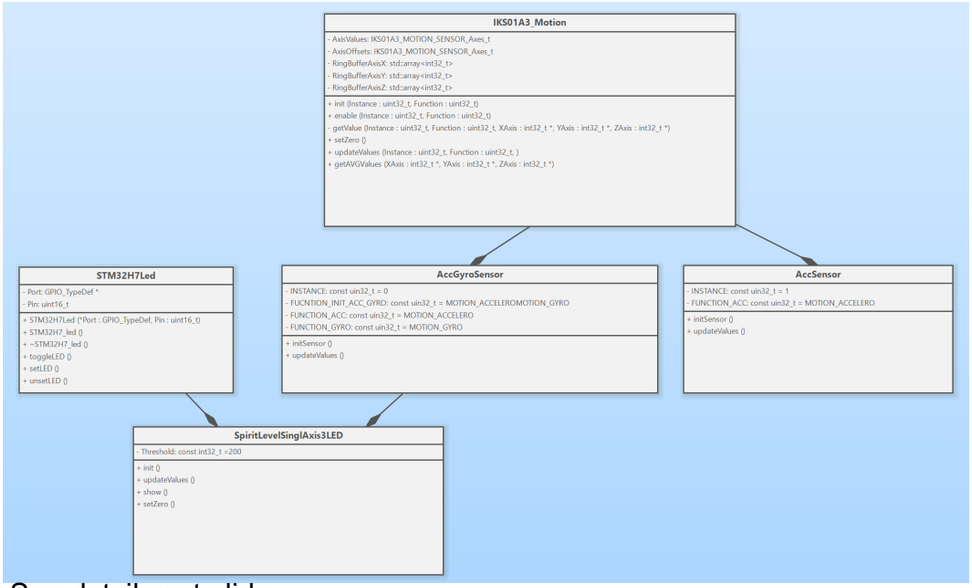
We use a custom allocator from Mr. Kormanyo's author of Real-Time C++.
We take the mean value of the accelerometer of the axis: Y of 10 values.

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 .

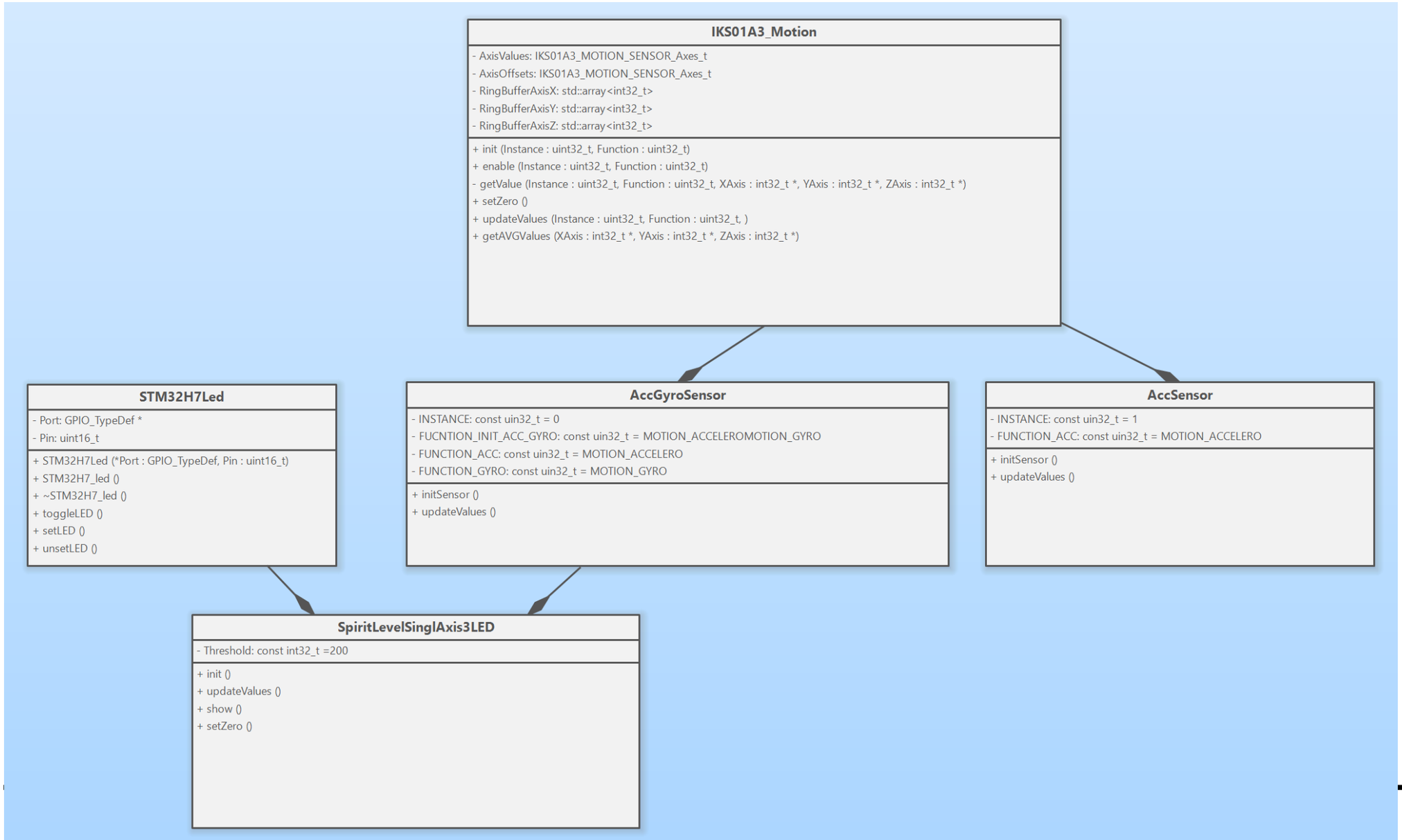
Output the value of the Y-axis on the console. Solution:

- Project:
  - CPP_SpiritLevel_Sol
- Git branch:
  - CPP_ SpiritLevel _03_04_Sol




See detail next slide

# 03_4_CPP_ SpiritLevel



# Thank you for your attention and cooperation