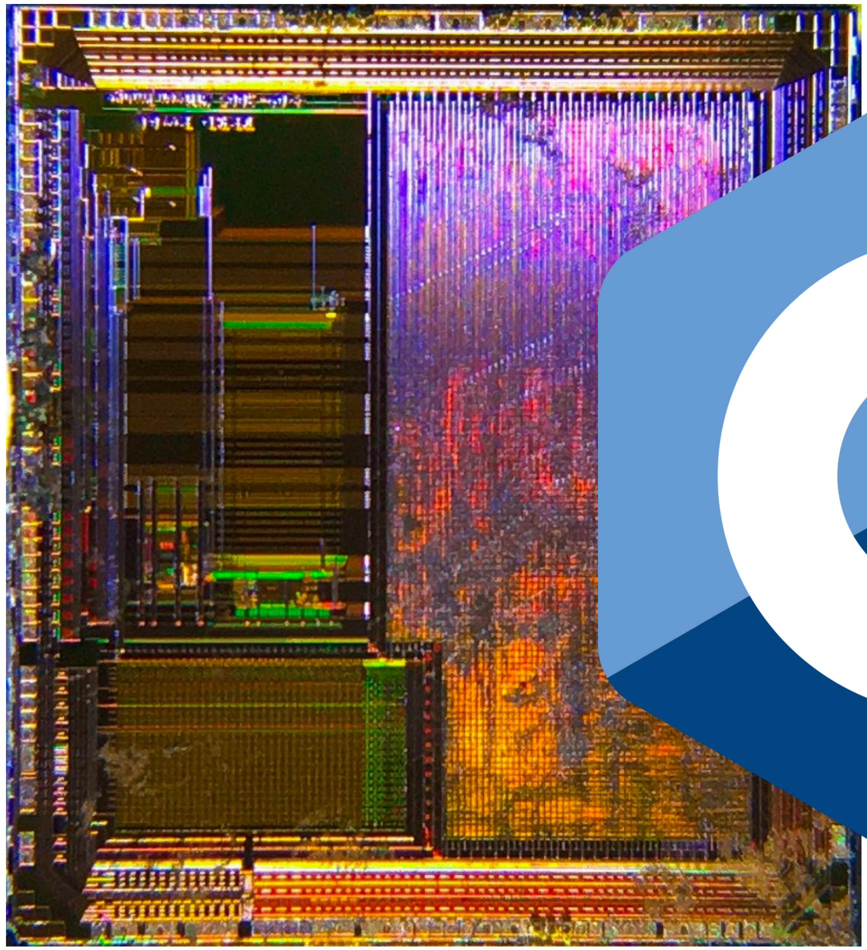
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

MASTER OF SCIENCE
IN ENGINEERING

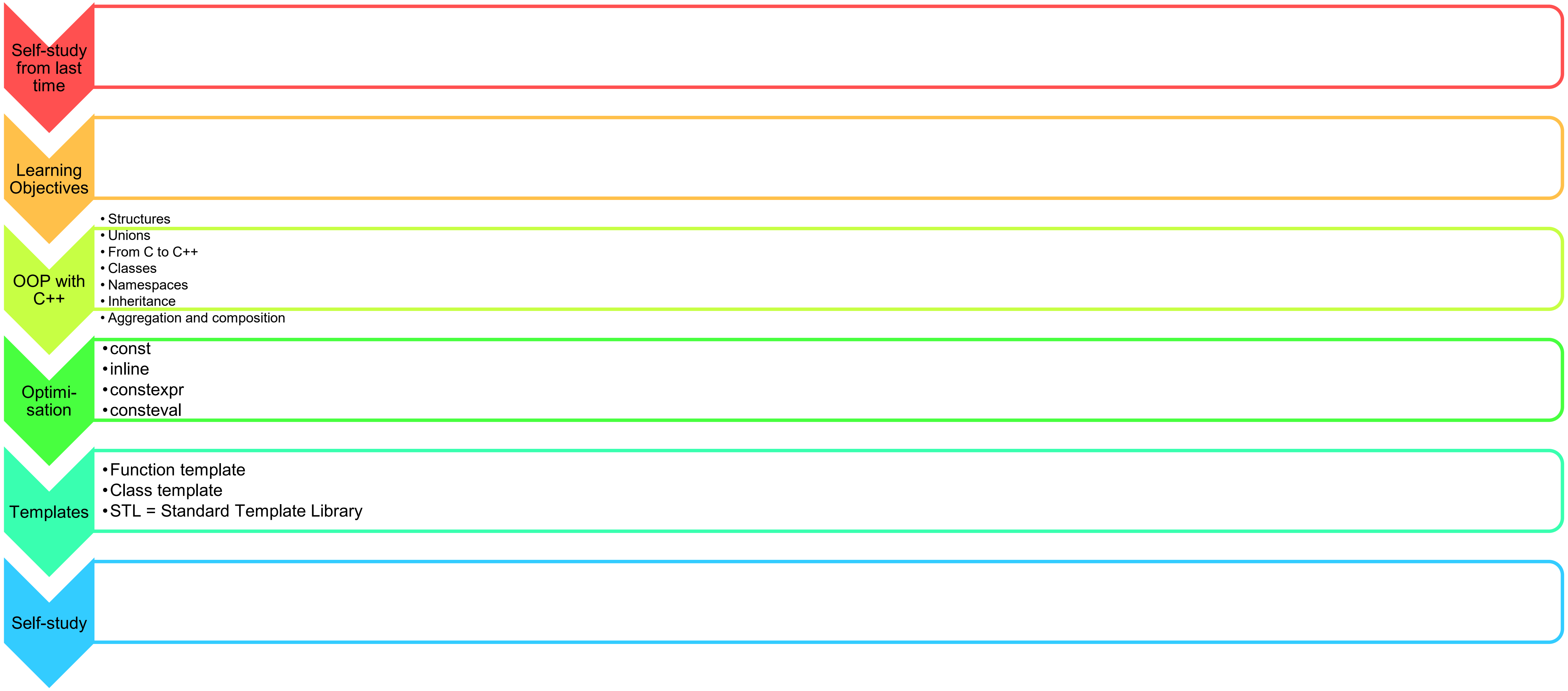
Embedded Real-Time Software

object oriented programming on Microcontroller from C to C++



26th September 2023 , Prof. Dominique-Stephan Kunz

Structure of the lessons



- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

- const
- inline
- constexpr
- consteval

- Function template
- Class template
- STL = Standard Template Library

Structure of the lessons

Self study from last time

Learning Objectives

OOP with C++

- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

Optimisation

- const
- inline
- constexpr
- consteval

Templates

- Function template
- Class template
- STL = Standard Template Library

Self study

Self study from last time

Installation of STM32CubeIDE

Any Questions?

01_1_Blinky

Create a new project: C_Blinky.

Use the HAL library to control the LED2 from the NucleoBoard every 500ms.

Note:

Search in HAL functions: Toggle and Delay

Press CTRL SPACE for autocompletion

01_2_BlinkyDelayNotBlocking

Expand project: C_Blinky.

Use the SystemTick timer to implement a software delay that does not block.

Note: HAL Delay

01_3_CPP_ Blinky

Create a new project: CPP_Blinky.

Use the HAL library to control the LED2 from the NucleoBoard every 500ms.
Implement this function in the C++ main.

Note:
Search in HAL functions: Toggle and Delay
Press CTRL & SPACE for autocompletion

01_4_CPP_EnumState

Create a new project: CPP_EnumState.

Implement two state machines by implementing it in the C++ part with two enum classes.

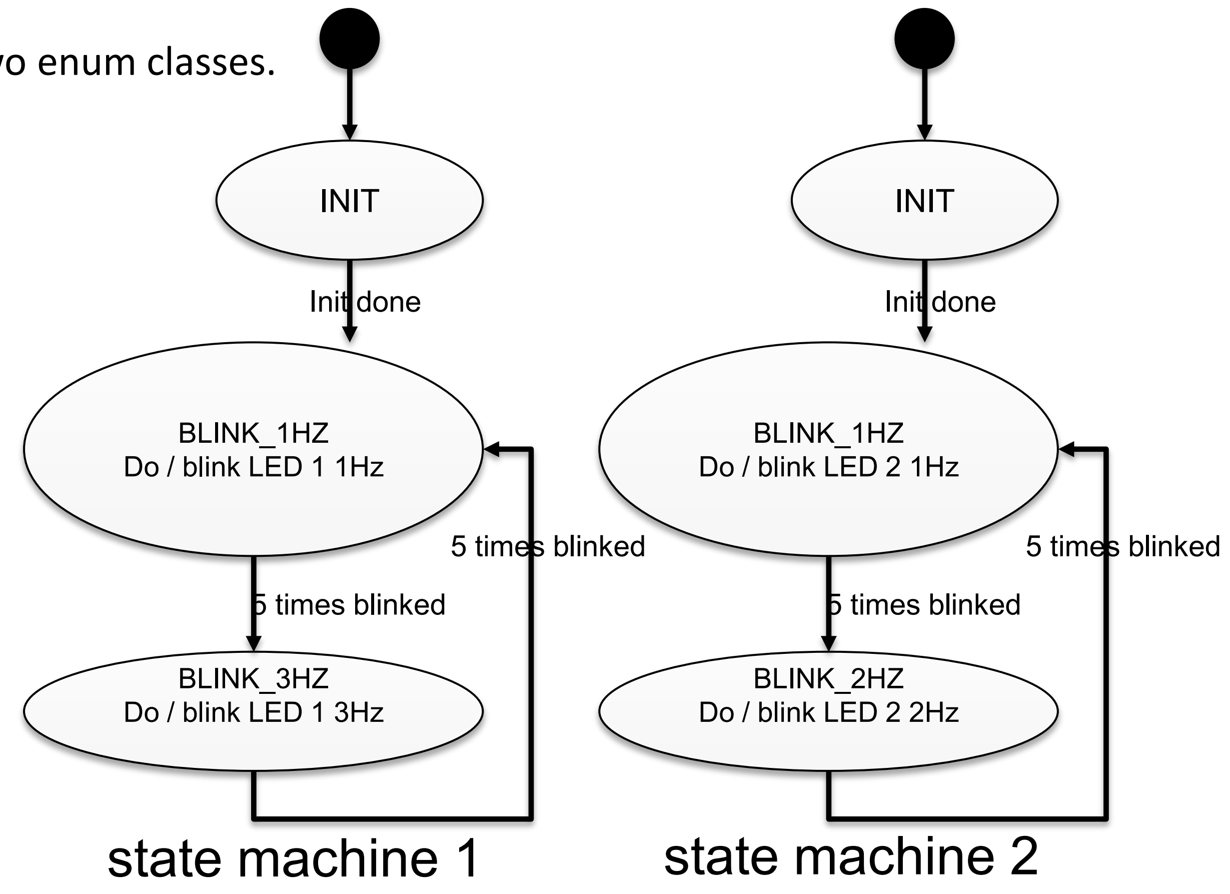
Following states should be implemented:

State machine 1 for LED 1:

- INIT (initializes the hardware)
- BLINK_1HZ (led the LED blink with 50% duty cycle at 1 Hz)
- BLINK_3HZ (led the LED blink with 50% duty cycle at 3 Hz)

State machine 2 for LED 2:

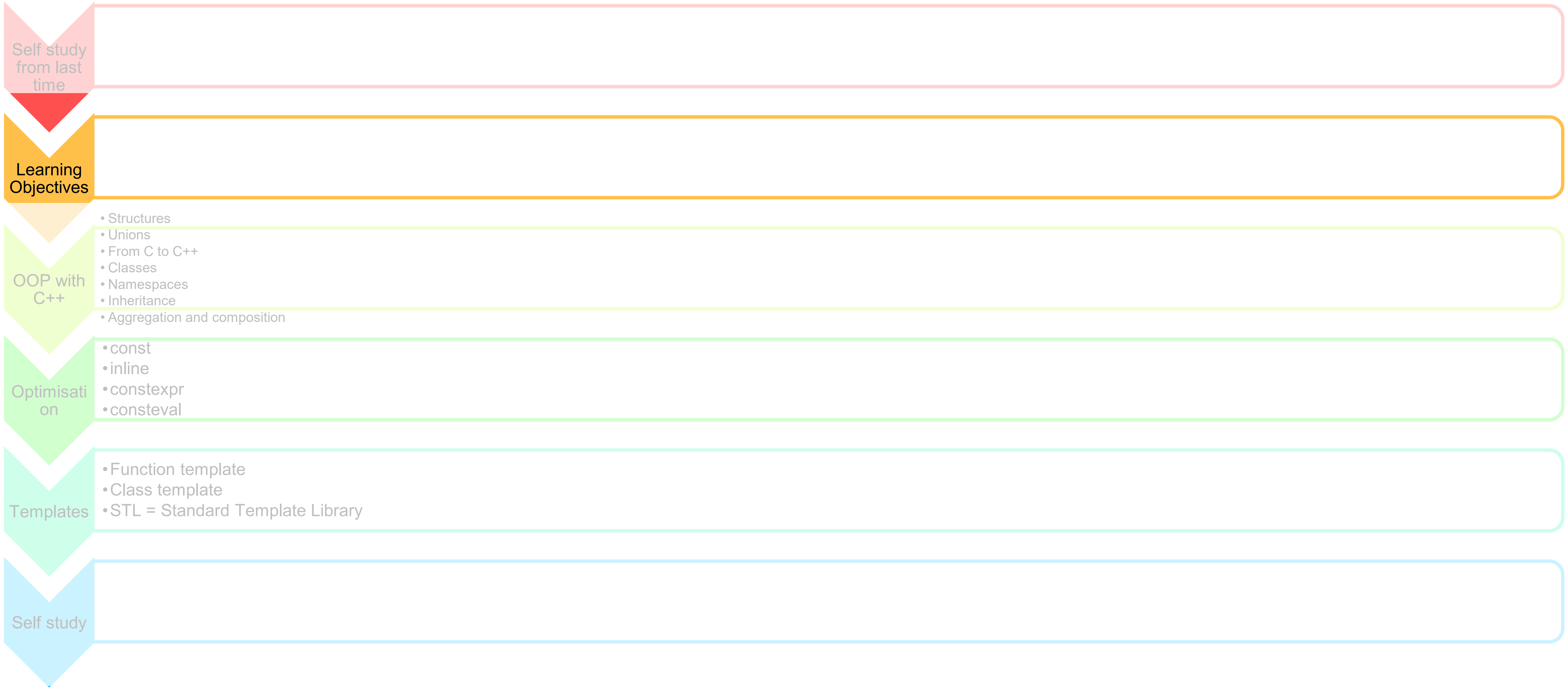
- INIT (initializes the hardware)
- BLINK_1HZ (led the LED blink with 50% duty cycle at 1 Hz)
- BLINK_2HZ (led the LED blink with 50% duty cycle at 2 Hz)



After 5 times blinking the LED the state is switched to BLINK_2HZ/BLINK3HZ and after 5 times back to BLINK_1HZ. Implement the delay non blocking.

Use the HAL library to control the LED1 and LED 2 from the NucleoBoard.

Structure of the lessons



- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

- const
- inline
- constexpr
- consteval

- Function template
- Class template
- STL = Standard Template Library

Learning Objectives

- You will make the transition from structures to classes.

- You will cover the principle of classes.
- You will cover namespaces.
- You will cover and apply class inheritance.
- You will cover aggregation and composition.

- You will cover C++ specifiers that help make C++ code more efficient and secure.

- You will learn what templates are and that they are a special technology of C++.

Structure of the lessons

Self study from last time
Learning Objectives
OOP with C++
Optimasation
Templates
Self study

- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

- const
- inline
- constexpr
- consteval

- Function template
- Class template
- STL = Standard Template Library

From C to C++

C++ is object oriented

C procedural

In C we know struct, which are an approach to C++.

Therefore, for repetition struct

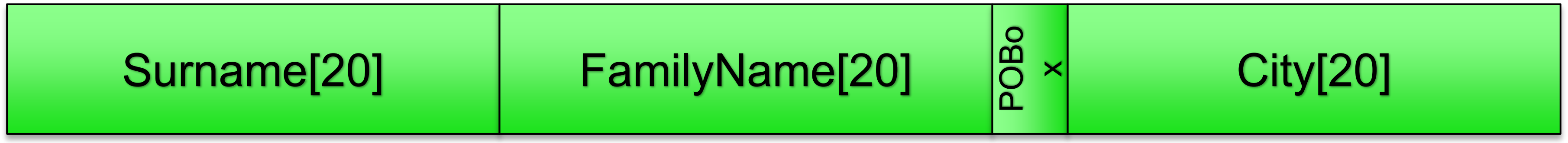
Structs

Previously, fields were used to group together the same data types.
uint32_t field[3];
With structures one can take different data types together.
It is the preliminary stage to the classes.
Since classes: Methods (functions) and attributes (variables) together to a "structure".

If one would like to summarize data logically e.g.:

```
char Surname[20];
char FamilyName[20];
uint64_t POBox;
char City[20];
```

In memory this is stored as one packet and padded to architecture width....



Structs: access on elements

```
struct adrestag_t{
char Surname[20];
char FamilyName[20];
uint64_t POBox;
char City[20];
} address1;
struct adrestag_t address2;
```

The elements are accessed as in JAVA on object elements:
VariableName(ObjectName) . (element)

```
address2.POBox=5560;
```

However, the access to the element to which the pointer points is done differently:

```
struct adrestag *PointerOnStrucutre;
PointerOnStrucutre=&address2;
*PointerOnStrucutre->POBox=5560;
```

The elements are addressed via the arrow operator ->.

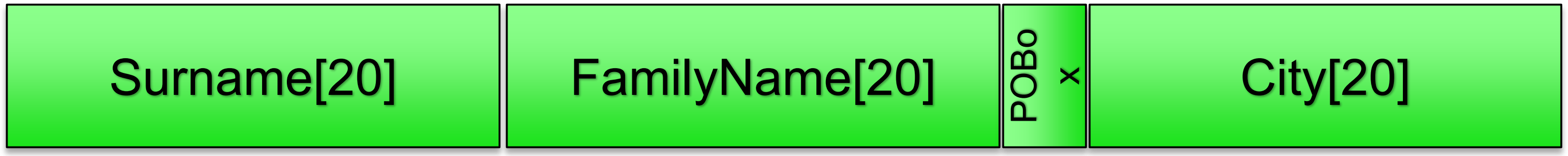
Unions

Unions

Another way to structure data are unions (also called variants). Apart from a different keyword, there are no syntactic differences between unions and structures. The difference is in the way the storage space of the data is handled.

Here is an example of how a structure uses memory:

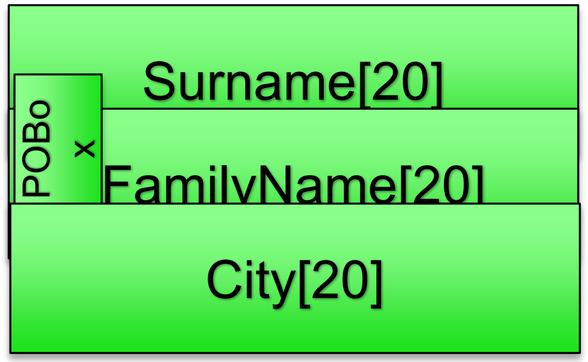
```
struct a_t{char Surname[20];
char FamilyName[20];
uint64_t POBox;
char City[20];}adress
```



size: 68 Byte

In a union, the elements are placed on top of each other and the largest element defines the memory requirement.

```
union a_t{char Surname[20];
char FamilyName[20];
uint64_t POBox;
char City[20];}adress
```



size: 20 Byte

Unions and structs combined
Importand for embedded Systems!

This application is very common and is a very elegant solution.

Example:

Definition of structures and union:

```
struct BIT32_T {
   unsigned int Bit0:1;
   unsigned int Bit1:1;
   unsigned int Bit2:1;
   unsigned int Bit3:1;
   unsigned int Bit4:1;
   unsigned int Bit5:1;
   unsigned int Bit6:1;
   unsigned int Bit7:1;
   unsigned int Bit8:1;
   unsigned int Bit9:1;
   unsigned int Bit10:1;
   unsigned int Bit11:1;
   unsigned int Bit12:1;
   unsigned int Bit13:1;
   unsigned int Bit14:1;
   unsigned int Bit15:1;
   unsigned int Bit16:1;
   unsigned int Bit17:1;
   unsigned int Bit18:1;
   unsigned int Bit19:1;
   unsigned int Bit20:1;
   unsigned int Bit21:1;
   unsigned int Bit22:1;
   unsigned int Bit23:1;
   unsigned int Bit24:1;
   unsigned int Bit25:1;
   unsigned int Bit26:1;
   unsigned int Bit27:1;
   unsigned int Bit28:1;
   unsigned int Bit29:1;
   unsigned int Bit30:1;
   unsigned int Bit31:1;
} ;
```

```
struct BYTE32_T {
   uint8_t Byte0;
   uint8_t Byte1;
   uint8_t Byte2;
   uint8_t Byte3;
} ;
struct WORD2_T {
   uint16_t Word0;
   uint16_t Word1;
} ;
```

```
union BITBYTEWORD_DWORD_t{
   struct BIT32_T  BitStructure;
   struct BYTE32_T ByteStructure;
   struct WORD2_T WordStructure;
   uint32_t DoubleWord;
};
```

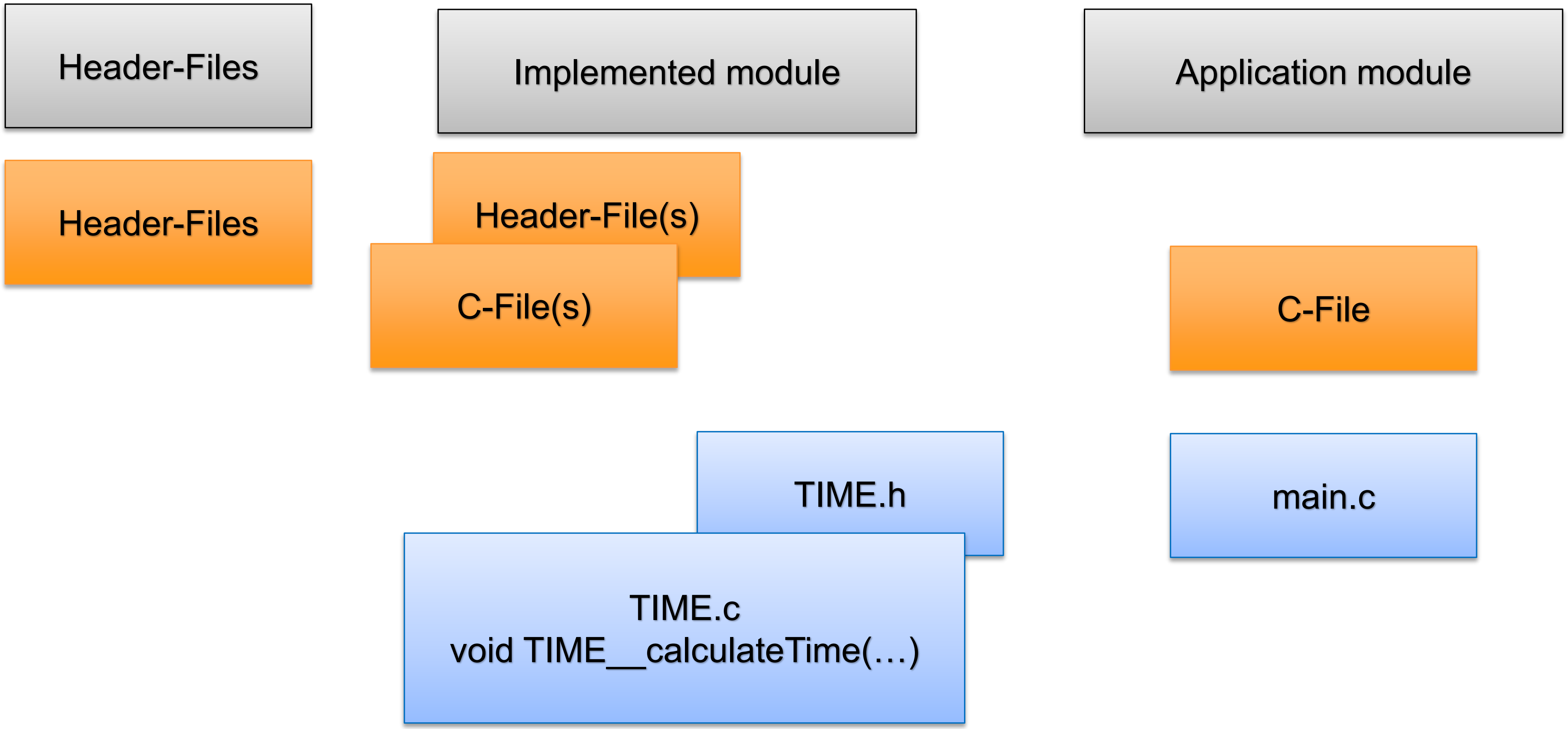
Instantiation and application.

```
union BITBYTEWORD_DWORD_t test;
  test.DoubleWord=0;
  test.BitStructure.Bit7=1;
```

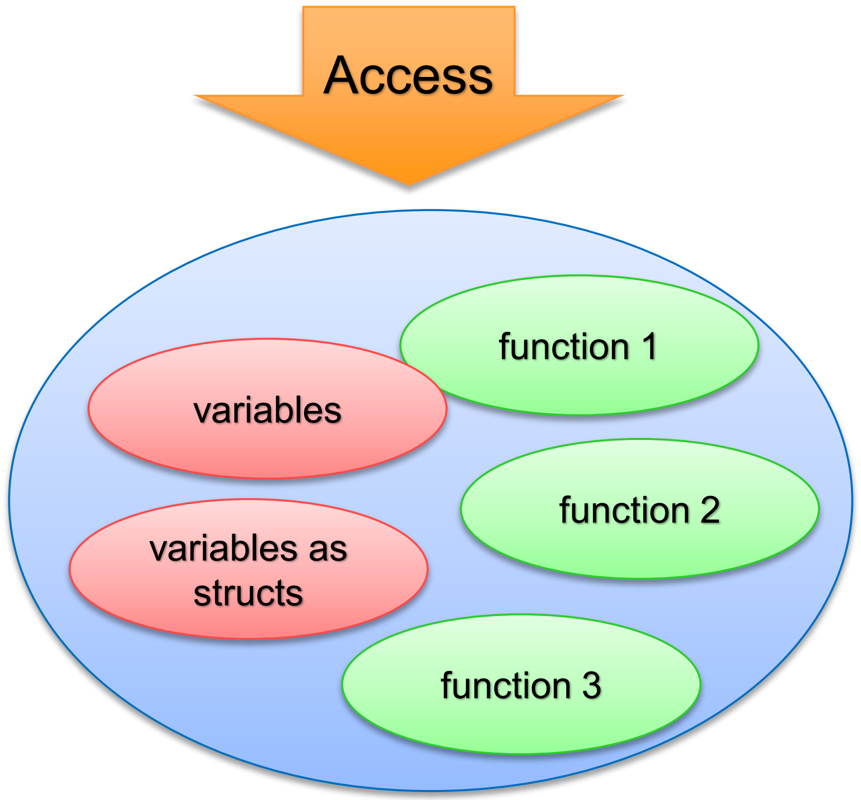
Result on the console:

From C to C++

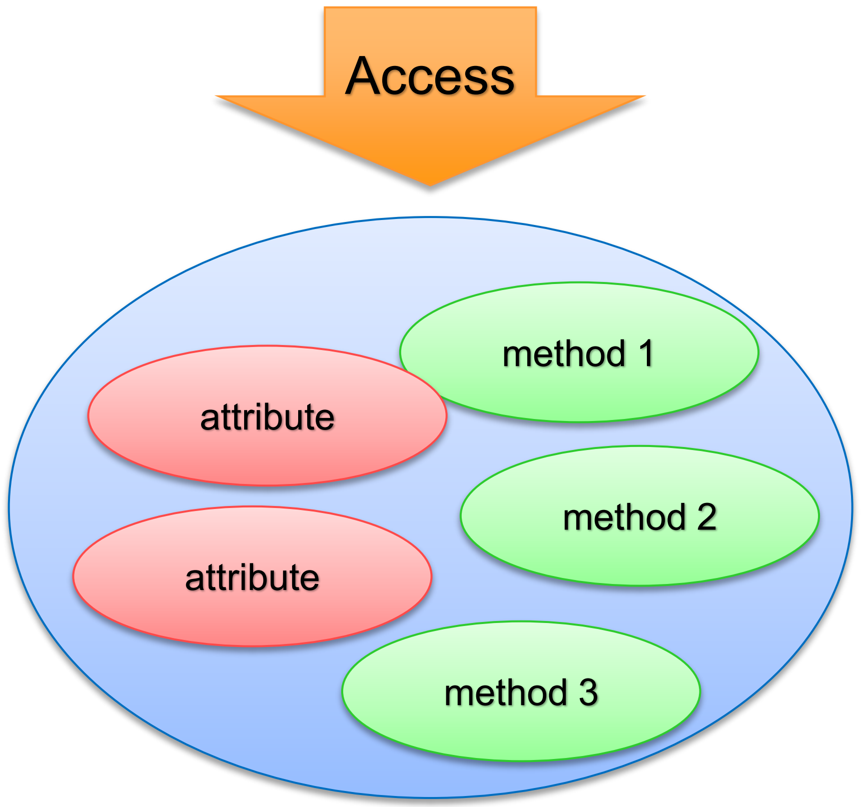
Modules in C


Header-Files
Implemented module
Application module
Header-Files
Header-File(s)
C-File(s)
C-File
TIME.h
main.c
TIME.c
void TIME__calculateTime(…)

From modules to structures to classes



module in C (*.h and *.c)



class in C++ (*.h and *.cpp)

Mentally, a module often represents a logical object!!

From modules to structures to classes

Glossar	C	C++ (Generally object oriented)
Data	variables	attribute
Bundlet commands	functions	methods
Data without being created in the working memory (not instantiated).	module	class
Data created in the working memory(instantiated).	Instantiated variables Will be created only once otherwise a new code must be created (variable or module copied and renamed).	object Can be created multiple times with new name.

Here is one reason why C++ could be more useful than C…

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**.

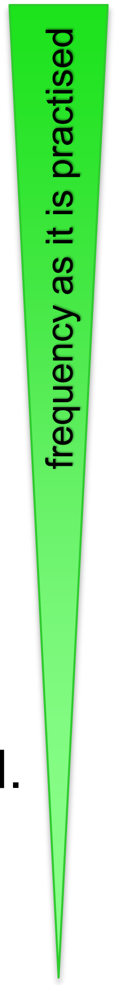
Day 3 and 4

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)



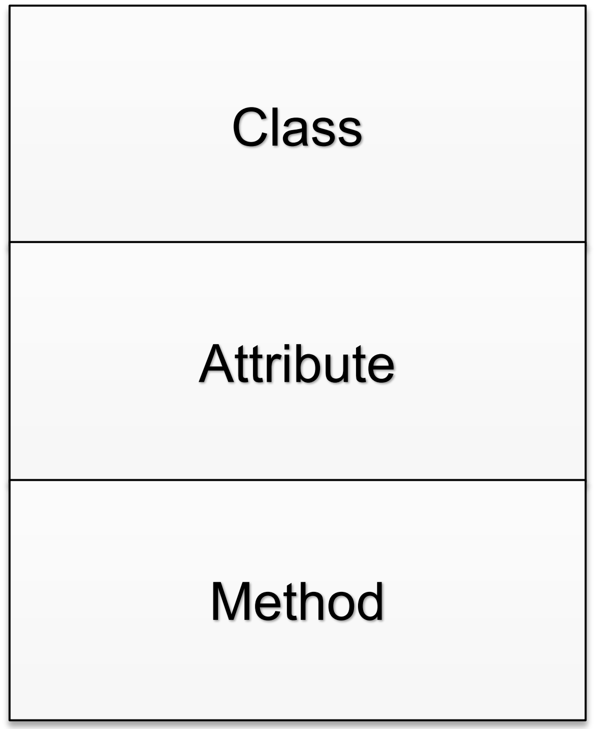
Classes

classes

A class consists of 3 elements:

- Class name
- Attributes
- Methods

This also corresponds to the UML notation



Properties of attributes and methods

Attributes and methods can be assigned properties. These define how these are visible to the outside and how they are accessible.

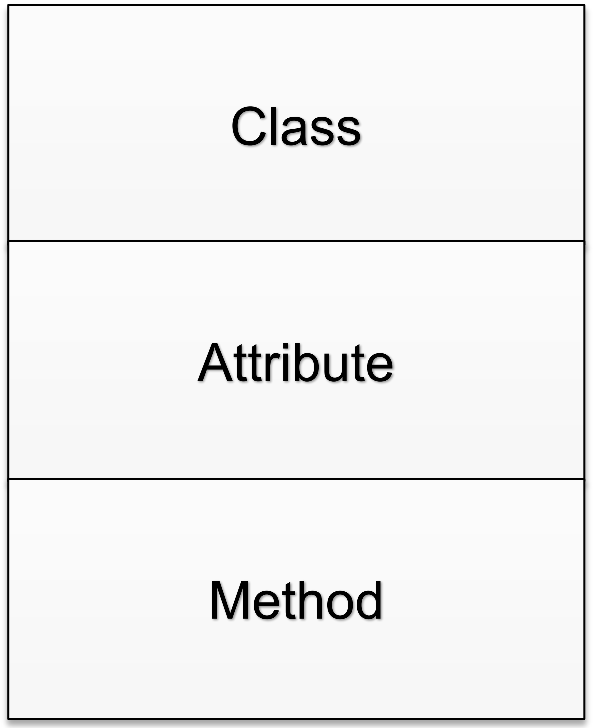
Furthermore, it is defined how access is possible with inheritance.

Public: externally visible, access possible

Private: not visible from outside and not accessible

Protected: like private, but special behavior when inherited.

Is there an analogy to C?



C++ class: code example

What do we see here?

Does anything stand out?

What is strange in contrast to C?

(Difference to Java?)

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class rectangle
  {
  private:
      double Area;
      double length;
      double width;

      void resetAreaToZero (void);
  protected:
      void setArea (double Area);

  public:
      double getArea (void );
      void calcArea (double length, double width);
      void calcArea (void);
};


#endif //RECTANGLE_H
```

```
#include "rectangle.h"

double rectangle::getArea (void)
{
   return(Area);
}

void rectangle::setArea (double Area)
{
   rectangle::Area = Area;
}

void rectangle::resetAreaToZero (void)
{
   Area = 0;
}

void rectangle::calcArea (double length, double width)
{
   Area= width * length;
}

void rectangle::calcArea (void)
{
Area=width*length;
}
```

Classes: Java vs. C++; Header and CPP file

In Java, the interface is given by the class definition.

In C++ the class definition and the implementations can be done separately.

Normally it is done separately.

```
//Example…
//Header file Part
class Geometry {
private:
   const double PI;
   std::string VIN;
   double Area;
public:
   Geometry (std::string id);
   void calculateArea (double Radius);
   double getArea();
}; // SEMICOLON!

//CPP file Part
   const double PI = 3.141;

   Geometry::Geometry (std::string id) {
      VIN = id;
      Area = 0;
   }
   void Geometry::calculateArea (double Radius) {
      Area = Radius^2*PI;
   }
   double Geometry::getArea() {
      return Area;
   }
```

Typical software principles in C)

Best practice

- No global variables
  - Getter and setter functions... getXYZ setXYZ

C++: Getter and setter methods

Module files are named as a bundle

C++: module/class name

You can plan in C the projects/modules as it would be objects.
Classes do not exist.

C++ class: Code Example

class

object

Header-File

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class rectangle
  {
  private:
     double Area;
     double length;
     double width;

     void resetAreaToZero (void);
  protected:
     void setArea (double Area);

  public:
     double getArea (void );
     void calcArea (double length, double width);
     void calcArea (void);
};


#endif //RECTANGLE_H
```

CPP-File

```
#include "rectangle.h"

double rectangle::getArea (void)
{
   return(Area);
}

void rectangle::setArea (double Area)
{
   rectangle::Area = Area;
}

void rectangle::resetAreaToZero (void)
{
   Area = 0;
}

void rectangle::calcArea (double length, double width)
{
   Area= width * length;
}

void rectangle::calcArea (void)
{
Area=width*length;
}
```

main-File

```
#include "rectangle.h"

int main() {

   double area;

   rectangle ObjRectangle;

   ObjRectangle.calcArea(5.4,43);
   area = ObjRectangle.getArea();

   return 0;
}
```

Constructor, destructor and scope

What are constructor and destructor?

What do they do?

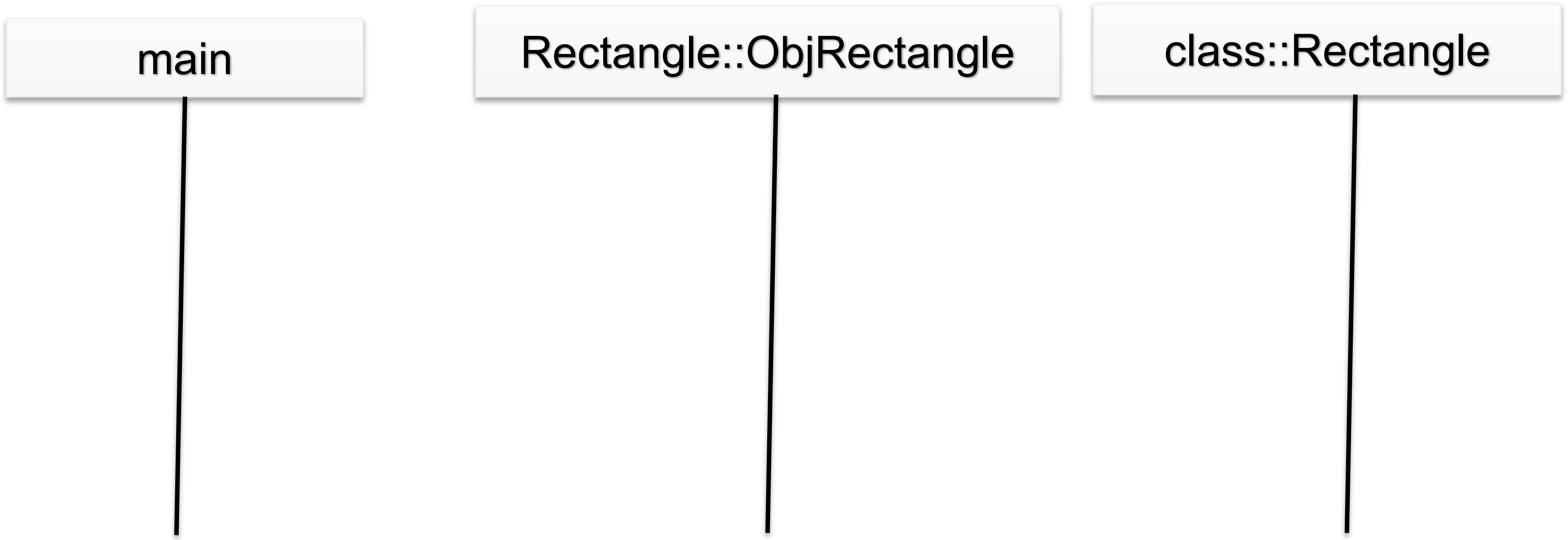
When are they active?

Where in the memory is the object intatiated ?

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class rectangle
  {
  private:
    double Area;
    double length;
    double width;

    void resetAreaToZero (void);
  protected:
    void setArea (double Area);

  public:
    double getArea (void );
    void calcArea (double length, double width);
    void calcArea (void);
};


#endif //RECTANGLE_H
```

main-Datei

```
#include "rectangle.h"

int main() {

   double area;

   rectangle ObjRectangle;

   ObjRectangle.calcArea(5.4,43);
   area = ObjRectangle.getArea();

   return 0;
}
```



Namespaces

Namespaces

Java uses the package mechanism to reduce the potential for naming conflicts.

All classes must belong to one package. Two classes with the same name can coexist as long as the two classes belong to different packages.

C++ namespaces are roughly equivalent to Java packages and serve the same purpose. To create a namespace and include some elements in the namespace, use a `namespace` declaration.

In namespaces the names may occur only once otherwise there are name conflicts.

The base of C++ is the `std` library which uses the namespace `std`.

The namespace resolver (**scope resolution operator**) `::` is used to access elements:

```
std::cout
```

Namespaces in use

On one hand you have to include the files where the code you want to use is. On the other hand, you have to use the namespace to apply the elements. There are three ways to do this:

Directly with resolution operator ::

```
std::string text ="Hello";

std::cout<<"text "<< text<<std::endl;
```

good practice

Importing an element from the namespace with using and direct access to the elements .

```
using std::string;
string text ="Hello";
```

namespace applied to whole file

Importing the whole or subset of the namespace with using and direct access to the elements.

```
using namespace std; // imports all elements of the namespace std
cout<<"text "<< text<<endl;
```

The scope of using can be defined or restricted with {}. The namespace is only valid inside the parenthesis.

```
int functiona() {
   using namespace std; }
```

Create a namespace

Namespaces are created with `namepace`.

These can also contain other namespaces.

```
namespace geometry{
   class myclass{
      private
      uint32_t one;
   public:
      void calcualtesomething(void ){};
      namespace circle{
         std::string circlename = "criclename";
      }
   };
}
```

The content can then only be used via the namespace:

```
int main() {

   myclass Geo; //ERROR
   geometry::myclass Geo;
   std::cout<<geometry::circle::circlename;
```

good practice

Anonymous or unnamed namespaces

You can also create **anonymous** or **unnamed namespaces** by **not specifying** a **name**.
This has almost the same effect as defining everything `static`.
All identifiers become file locally.

```
namespace {
   class myclass{
      private
      uint32_t one;
   public:
      void calcualtesomething(void ){};
      namespace circle{
         std::string circlename = "criclename";
      }
   };
}
```

Useful for defining a scope for mutex to delete themselves when leaving scope

I other words: it is useful when you want to make variable declarations invisible to code in other files (i.e. give them internal linkage) without having to create a named namespace. All code in the same file can see the identifiers in an unnamed namespace but the identifiers, along with the namespace itself, are not visible outside that file—or more precisely outside the translation unit.

Conclusion of namespaces

Namespaces can be bound several classes and modules into one namespace!

Namespaces help:

- Improving program organisation
- Improving code readability -> dependencies

Hint: Namespaces can be used in a C++ project where primarily C Technologies are used!

Namespaces & classes

resolution operator not needed inside class methods, since already in its namespace

A class is actually also a “namespace”!

(**scope resolution operator**) `::`

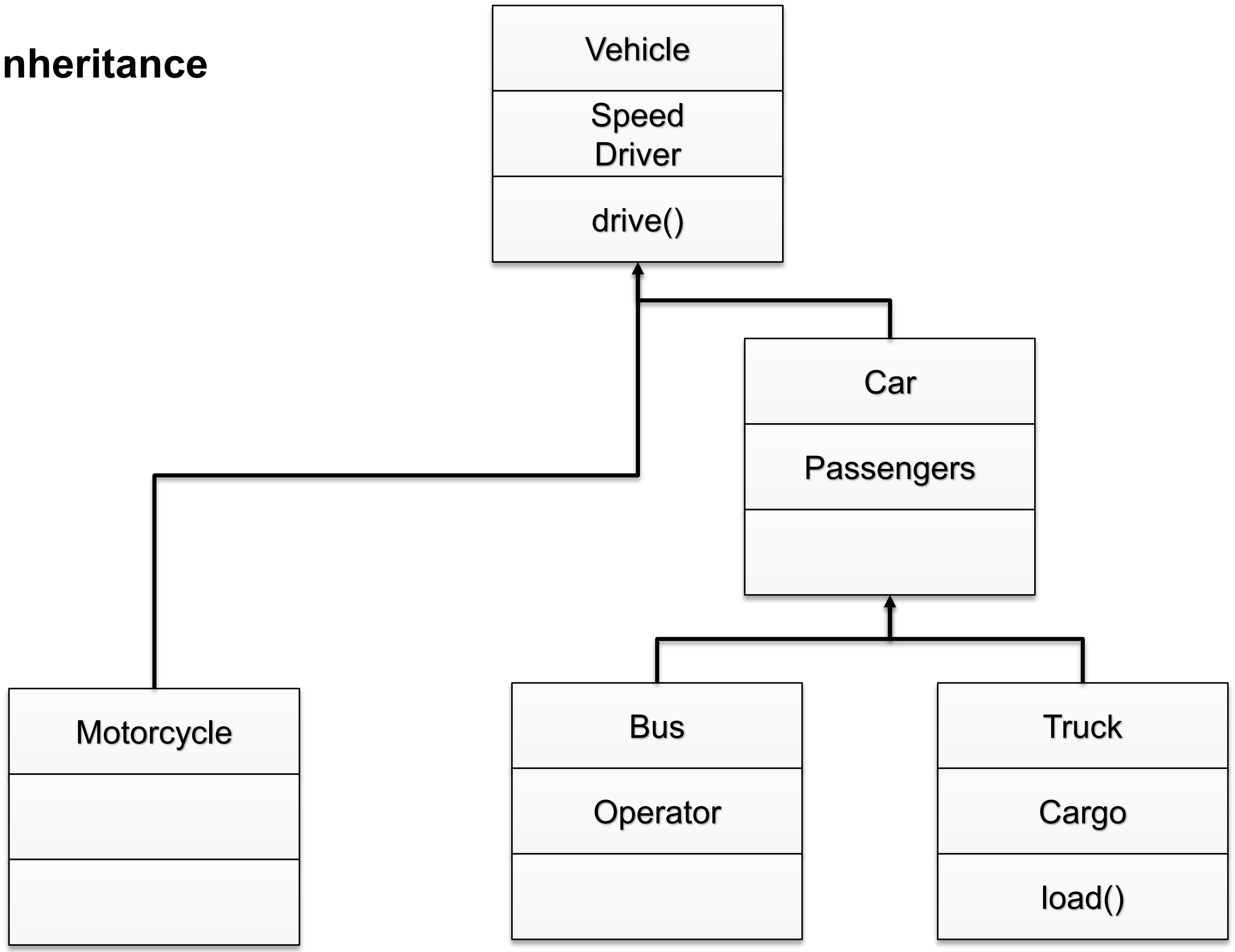
```
BlinkingLed::BlinkingLed(GPIO_TypeDef *Port,uint16_t Pin) {
BlinkingLed::Port=Port;
BlinkingLed::Pin=Pin;
}
```

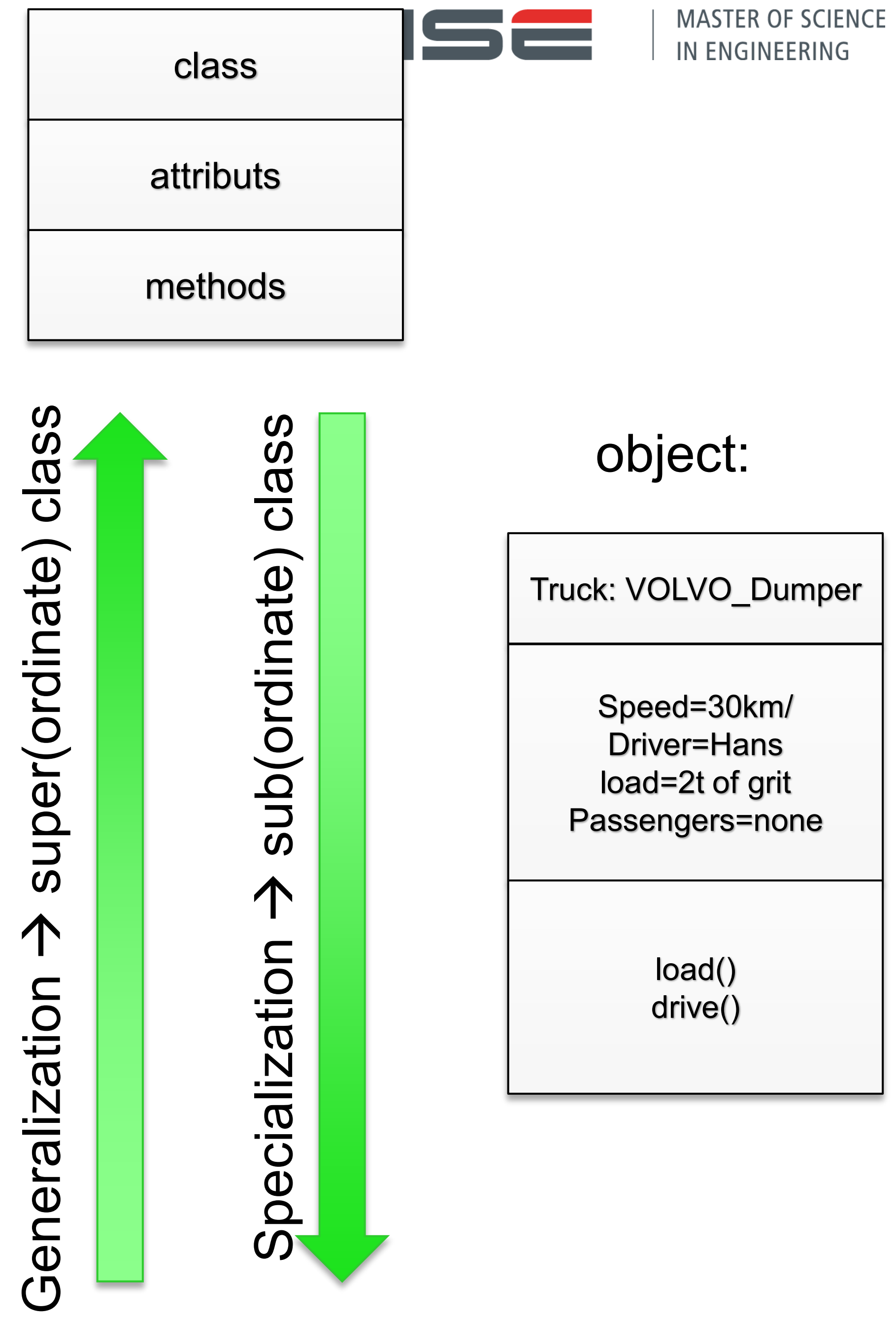
What does this do?

How does the class looks like? (atrributes, method)

Inheritance

Inheritance





Classes: Java versus C++

C++ allows inheritance of multiple classes, while Java only allows inheritance of a single class.
C++ e.g.:

```
class oev{ //class oev
};

class car{ //class car
 };

class suv : public car{ //class SUV inherits from car
};

class oevsuv : public suv, oev{ // class SUV inherits from car  and oev

};
```

C++ classes: Inheritance

- `private` -members (attributes and methods)
  - Access only possible from within the class.
- `protected` member
  - Access possible from within the class
  - Access possible from within a derived class
- `public` member
  - Access possible from within the class
  - Access possible from inside a class
  - access possible from outside a class

This applies when inherited with `public`!

```
class suv : public car{ //class SUV inherits from car
};
```

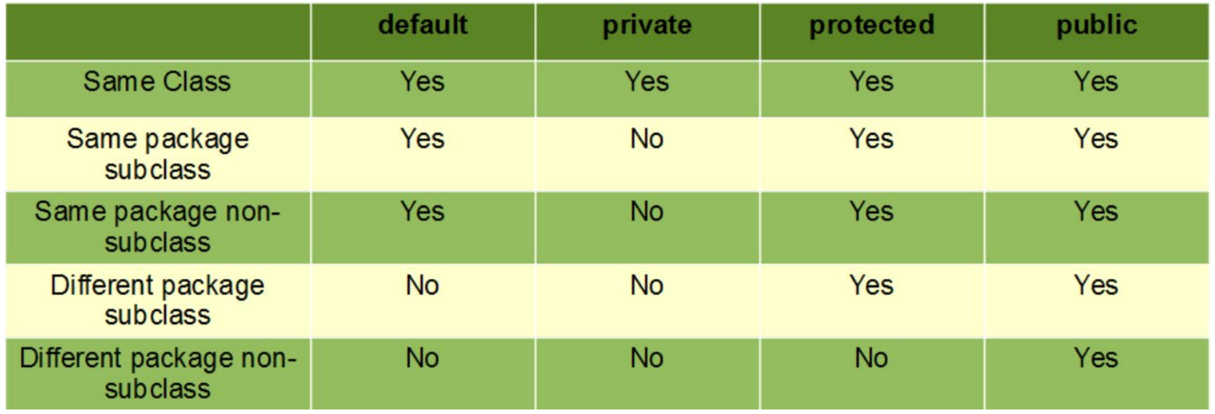
C++ classes: Inheritance, public protected and private

Inheritance can override access rights in C++.

```
class suv: public car{ //class SUV inherits from car
};

class luxury: protected car{ // class SUV inherits from car
};

class smart: private car{ // class SUV inherits from car
};
```

Access right in the base class / Type of inheritance	public	protected	private
Public inheritance	public	protected	blocked
protected inheritance	protected	protected	blocked
private inheritance	private	private	blocked

In contrast, Java works like public inheritance.



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

https://www.geeksforgeeks.org/access-modifiers-java/

02_1_CPP_BlinkyDelayNotBlocking

Expand project: CPP_Blinky.

Use the SystemTick timer to implement a software delay that does not block.

Use the HAL library to control the LED1 every 250ms.

Use the HAL library to control the LED2 every 500ms.

Use the HAL library to control the LED3 every 1000ms.

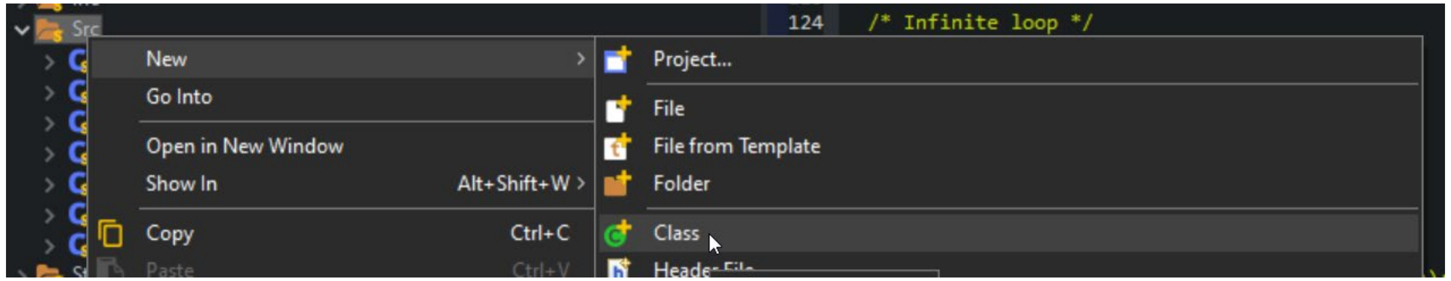
Implement this function in the C++ main.

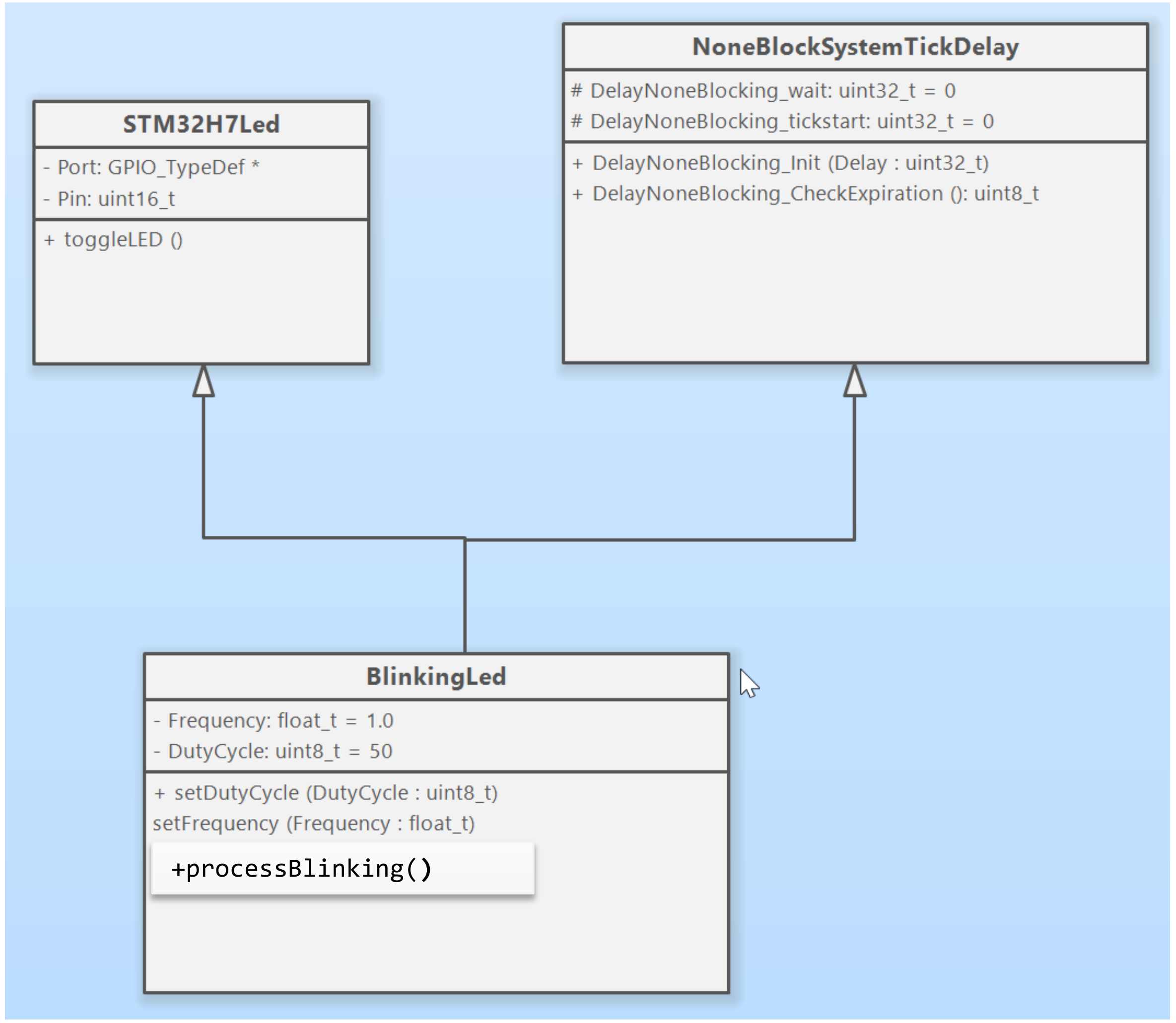
Caution we need 2 constructors:

1 standard

1 with initialization parameters!

Make use of the wizard for creating a class and setter/getters!




STM32H7Led
- Port: GPIO_TypeDef *
- Pin: uint16_t
+ toggleLED ()
NoneBlockSystemTickDelay
# DelayNoneBlocking_wait: uint32_t = 0
# DelayNoneBlocking_tickstart: uint32_t = 0
+ DelayNoneBlocking_Init (Delay : uint32_t)
+ DelayNoneBlocking_CheckExpiration (): uint8_t
BlinkingLed
- Frequency: float_t = 1.0
- DutyCycle: uint8_t = 50
+ setDutyCycle (DutyCycle : uint8_t)
setFrequency (Frequency : float_t)
+processBlinking()

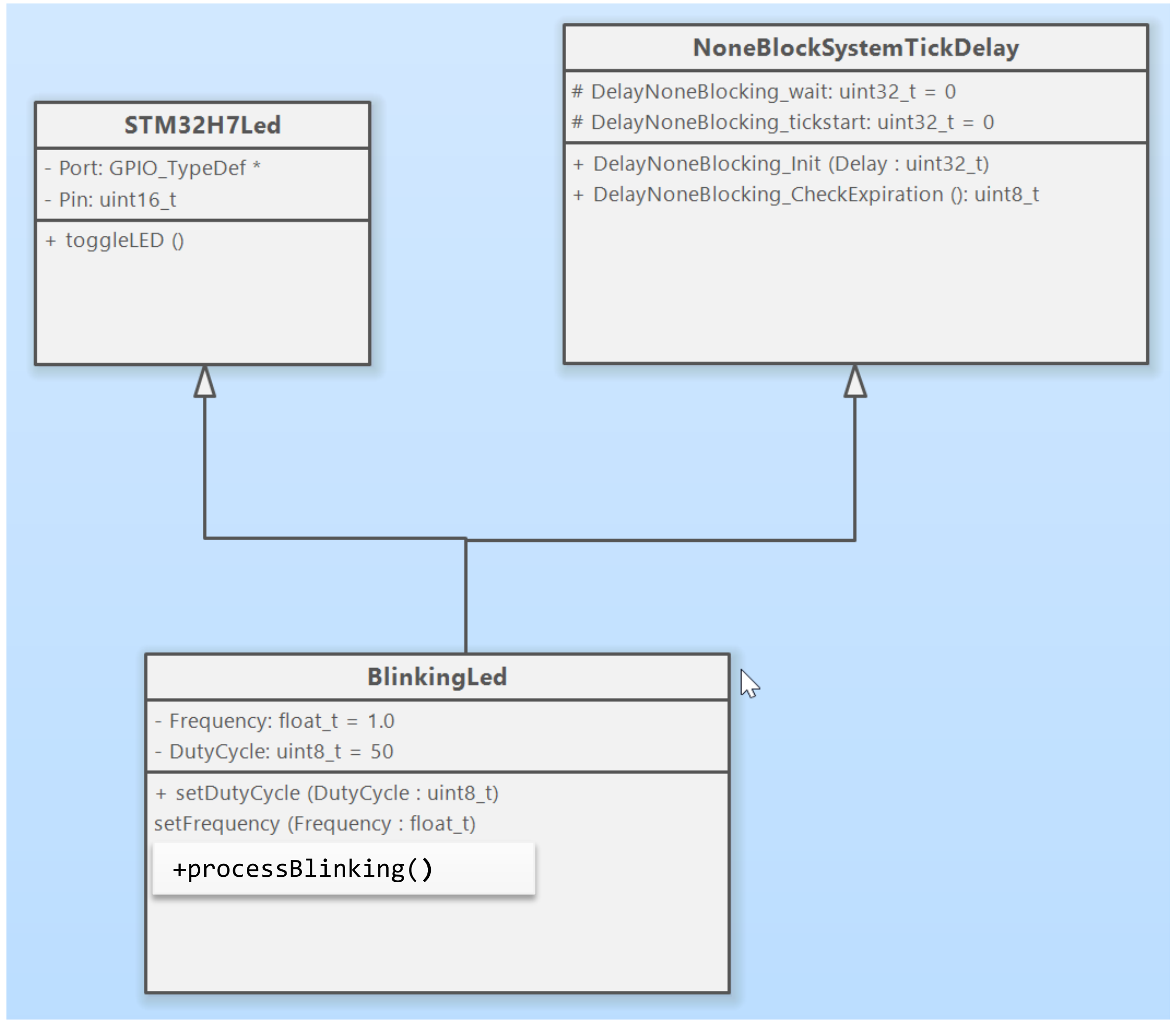
Setter and getters: select Header-File, **ALT+SHIFT+S+R**, then “Generate Getter Setters…”

02_2_CPP_BlinkyNamespace

Expand project: CPP_Blinky 02_02.

Define a namespace: “myhal”

And include STM32H7Led and NoneBlockSystemTickDelay into this namespace.

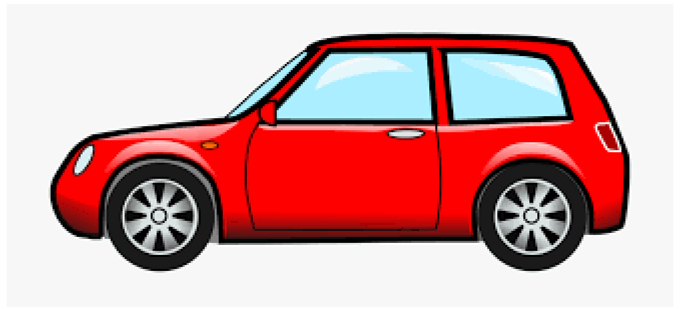

STM32H7Led
- Port: GPIO_TypeDef *
- Pin: uint16_t
+ toggleLED ()
NoneBlockSystemTickDelay
# DelayNoneBlocking_wait: uint32_t = 0
# DelayNoneBlocking_tickstart: uint32_t = 0
+ DelayNoneBlocking_Init (Delay : uint32_t)
+ DelayNoneBlocking_CheckExpiration (): uint8_t
BlinkingLed
- Frequency: float_t = 1.0
- DutyCycle: uint8_t = 50
+ setDutyCycle (DutyCycle : uint8_t)
setFrequency (Frequency : float_t)
+processBlinking()

Aggregation and composition

Aggregation or composition

Aggregation or composition arises through decomposition

Parts (components, part) are subordinated to the whole (aggregate)



POWER
Navigation

Bilder: http://clipart-library.com/

Aggregation or composition

“True aggregation” or **composition**

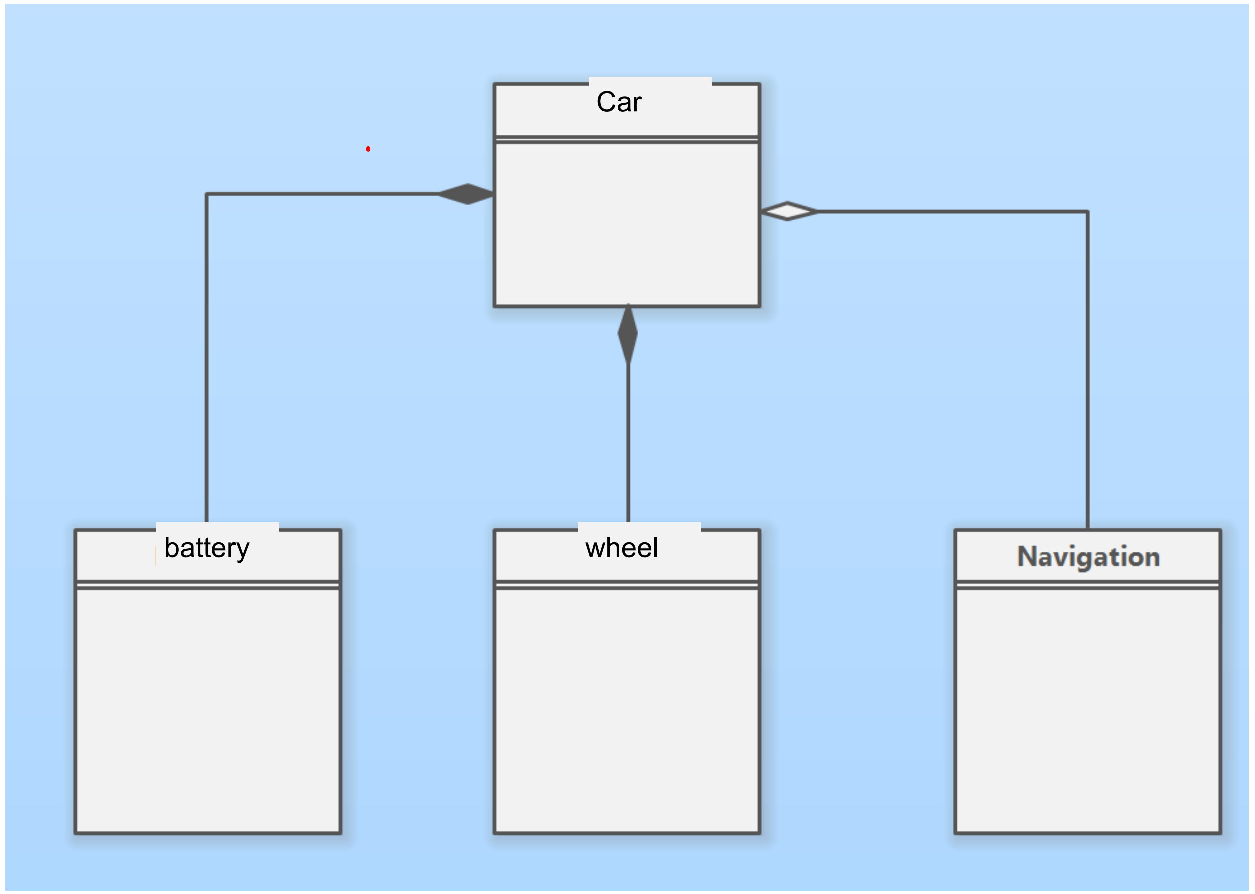
Is represented by the **filled (black) diamond**.

In a true aggregation, the aggregate (Car) can exist only if all its components also exist.

Aggregation

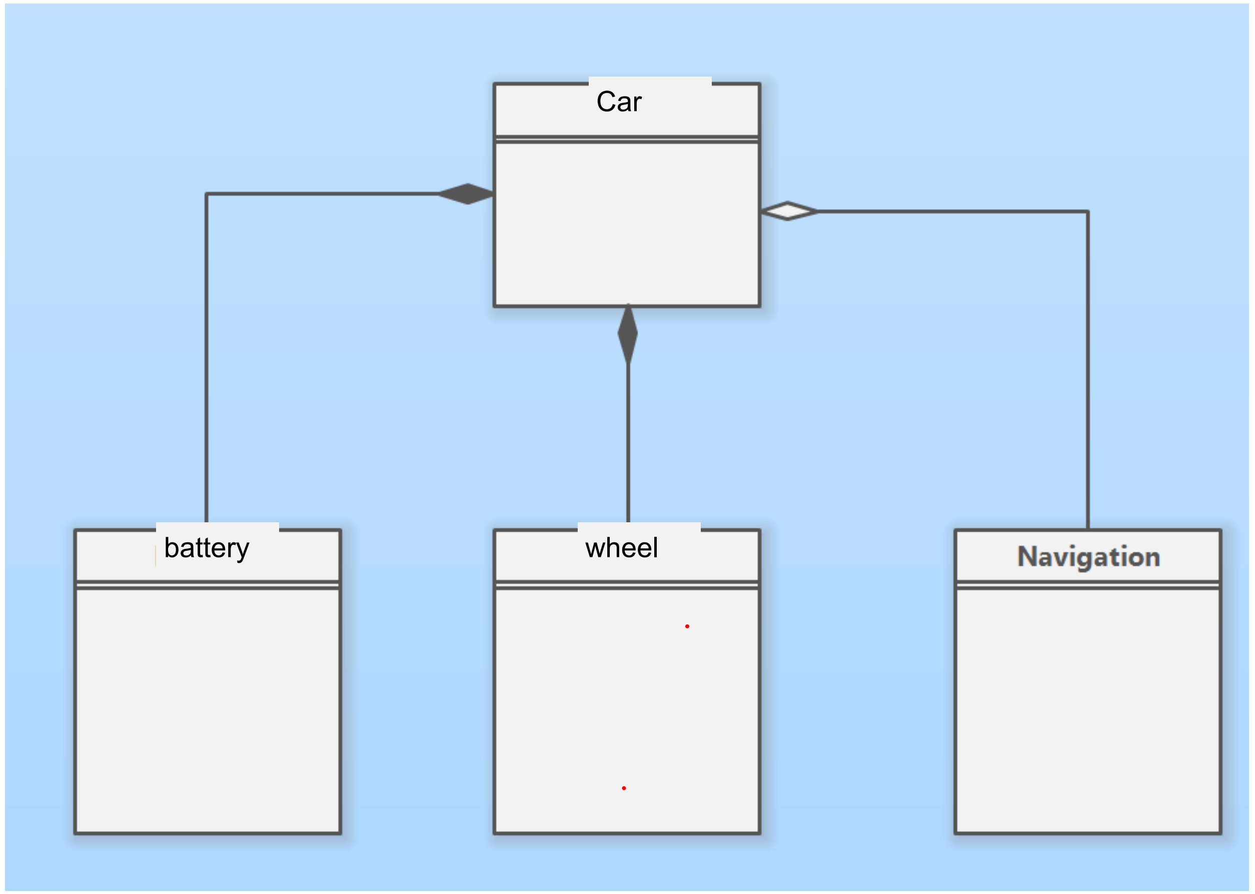
Is represented by the **open (white) diamond**.

The aggregate does not have to contain the named component.

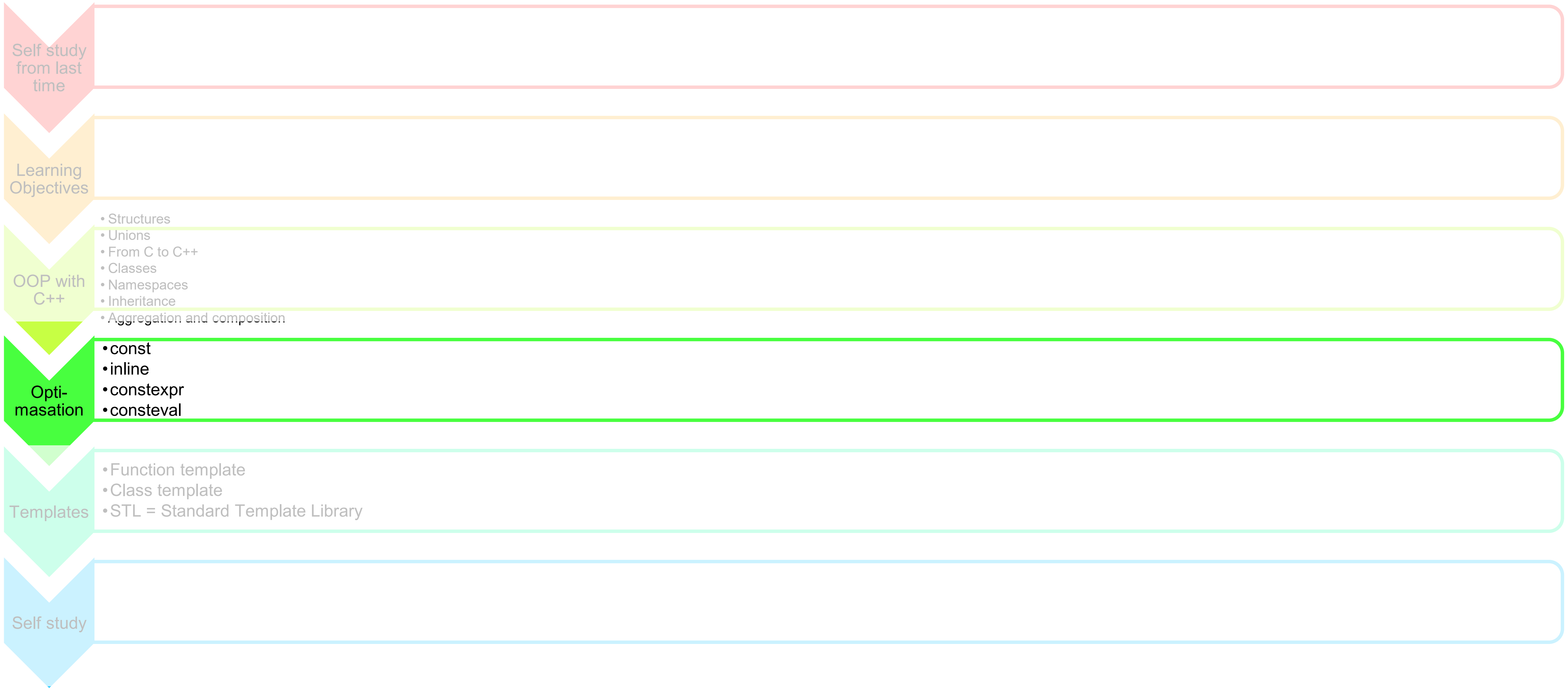


Code example of composition and aggregation

```cpp
class Battery{
public:
   uint32_t StateOfCharge;
};

class Wheel{
public:
   uint32_t Radius;
};

class NavigationSystem{
public:
   uint32_t Manufacturer;
};

class Pkw{
public:
   NavigationSystem GarminX10;
   Wheel GoodyearFL, GoodyearFR , GoodyearRL , GoodyearRR;
   Battery Varta70Ah;
};

int main() {
   Pkw VW_Golf;

   VW_Golf.Varta70Ah.StateOfCharge=50;
}
```



Structure of the lessons



- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

- const
- inline
- constexpr
- consteval

- Function template
- Class template
- STL = Standard Template Library

const

const: a case like in Java with final

The specifier const is used in two ways:
For values or for methods.
For values it behaves like in Java with final.

```
const double PI =3.141;
```

Whereby whether one:

Writes means the same.

```
const int a=2323;
int const b=234423;
```

const variables, can be initialized at **compile time or at runtime**. After that they are write protected!

const: more cases

```
int const * PointOnInt;
int * const PointOnInt2;
int const * const PointOnIn3;
```

```
const double getPI(void) {
   double PI=3.14;
   return (PI);
}

const double setGetPI(const double *Pi) {
   double PI=*Pi;
   return (PI);
}
```

```
const std::string& printID(void) {
return ("const string");
}
```

const: more cases

With methods the usage says that the method does not change any attributes.

This does not exist in Java!

```
class C {
private:
   double value=35.43;
public:
   double getVAlue(void) const{
      return(value);
   }
};
```

```
std::cout << CObj.getVAlue()<< std::endl;
```

inline

Inline

The specifier `inline` can be prefixed to functions.

This **suggests** the compiler to copy in the code where the function is called.
Just as if one would make a macro with `define`.

This is **only a suggestion**.

Compiler inlines also without keyword if:

- If the function is a function template.
- At maximum compiler level for speed.

```
inline double calculateArea(double Radius){
   double localArea;
   localArea = Radius * Radius * 3.14159265359;
   return (localArea);
}
```

constexpr

C++11 and following

constexpr

As with `inline` before a method, constexpr ensures that the function is used directly where the method is called.
However, this is **not a suggestion but a must**. The compiler will throw an error if this is not possible.

```
constexpr double calculateArea(double Radius){
   double localArea;
   localArea = Radius * Radius * 3.14159265359;
   return (localArea);
}
```

A `constexpr` specifier causes an **inline** if used in a function or static data element declaration (since C++17).

At **compile time** functions declared with `constexpr` are evaluated and processed. However, this does not necessarily lead to a costant value, it can lead to an inline function!

C++11 and following

constexpr

As with `const` before a variable, constexpr ensures that the variable is evaluated at compile time. However, this is **not a suggestion but a must**. The compiler will throw an error if this is not possible.

```
#include <iostream>
#include <cstdint>

constexpr int my_variable = 10;

int main() {
     int a = my_variable;
     int b = my_variable;
     int c = my_variable;
   std::cout << "a: " <<a << std::endl;
   std::cout << "b: " <<b << std::endl;
   std::cout << "c: " <<c << std::endl;
   return 0;
}
```

A `constexpr` specifier causes a **constant** when used in an object declaration or a non-static member function (up to C++14). This is called: constant folding.

C++11 and following

Best practice

constexpr

Same example but with more robust and portable measures.

```
#include <iostream>
#include <cstdint>

constexpr std::int32_t my_variable = INT32_C(10);

int main() {
     int a = my_variable;
     int b = my_variable;
     int c = my_variable;
   std::cout << "a: " <<a << std::endl;
   std::cout << "b: " <<b << std::endl;
   std::cout << "c: " <<c << std::endl;
   return 0;
}
```

A `constexpr` specifier causes a **constant** when used in an object declaration or a non-static member function (up to C++14). This is called: constant folding.

consteval

consteval

C++20

With `consteval` the highest form for the least time required to execute a program has been added in C++20.

This specifier causes the function to become an **immediate** function.

One **may not apply** this to **destructors**, **allocation** or **deallocation** functions!

```
consteval double calculateArea(double Radius){
   double localArea;
   localArea = Radius * Radius * 3.14159265359;
   return (localArea);
}
```

But this is only possible if constants are used.

```
std::cout << calculateArea(5.5)<< std::endl;
```

At **compile time** functions declared with `consteval` **must** be evaluated and processed otherwise an error is thrown!

Consequently, the code can not allocate memory on Heap.

Overview const’s mechanism

use for function arguments

Overview

Feature	define	const	inline	constexpr	consteval
Can be applied to	As **Values**, as **Macros**	**Variables**, (usage for more but different use cases)	**Functions**	**Variables/ attributes** and **functions/ methods**	**Functions/ methods only**
Must be evaluated at compile time	Yes	No	Yes	Variables: Yes Functions: No	Yes
Can perform operations that cannot be evaluated at compile time	No	No	No	Yes	No
Inserts	Value or Code	Reference to Value	Code	Value `foo(1,2)` Code, `foo(x,y) -> inline`	Value
Type safety	No	Yes	Yes	Yes	Yes
	C & C++	C & C++	C & C++	C++ (11, 20)	C++ (11, 20)

Conclusion of optimisation on constant expressions

`const` and its derivatives (`consteval constexp` etc.) should be used as often as possible!

The rule should be a mutable Reference, variable, access etc. is a special situation!

In general: `mutable: able to be changed after initial assignment`

1. `consteval`
2. `constexp`
3. `const`
4. None of these

To speed up methods consider

1. `consteval`
2. `constexp`
3. `inline`

Leads to more robust, efficient and reliably Code.

Hint: This can be used in a C++ project where primarily C Technologies are used!

Structure of the lessons

Self study from last time	
Learning Objectives	
OOP with C++	• Structures • Unions • From C to C++ • Classes • Namespaces • Inheritance • Aggregation and composition
Optimasati on	• const • inline • constexpr • consteval
Templates	• Function template • Class template • STL = Standard Template Library
Self study	

Templates

Template

Template is an important base technology behind C++.

There are function and class templates.

In Java, this corresponds to the generic method and generic class.

Problem situation:

If one would like to write an algorithm, then the data type is mostly unknown when implementing.
E.g. the maximum function.
This leads to the fact that the functions must be implemented per data type.

```
int max_int(int value_a, int value_b)
{
   if(value_a>value_b)return(value_a);
   else return (value_b);
}
double max_double(double value_a, double value_b)
{
   if(value_a>value_b)return(value_a);
   else return (value_b);
}
```

The function body is always the same only the data type varies.
Solution: Implementation as **function template**

Function template

Function template

The identifier template introduces the template followed by <> brackets.

In these comes **typename** or **class,** these have the same meaning.

After typename comes the character or string which represents the data type in the code.

template <class type identifier> or template <typename type identifier>.

```
template <typename VARIABLEDATATYPE>
VARIABLEDATATYPE max(VARIABLEDATATYPE value_a, VARIABLEDATATYPE value_b)
{
   if(value_a>value_b)return(value_a);
   else return (value_b);
};
```

VARIABLEDATATYPE used for both input arguments and return type of function max()

When compiling, the compiler generically creates the necessary function implementations in machine code.

Application:

```
double a=3.2, b=33.2;
int32_t c=342,d=432;
std::cout<<"max ist " << max(a,b)<<std::endl;
std::cout<<"max ist " << max(c,d)<<std::endl;
```

A function template is a function that is defined for abstract data types as generation parameters. With each function call, the function matching the data type is generated based on the current arguments.

How man functions are in the program code?

Class template

Class template

Like function templates, you can program class templates.

```
template <class T>
class GenericDataTypeClass{
private:
   T localValue;
public:
   T getValue(void){
      return(localValue);
   }
   void setValue(T value)
   {
      localValue = value;
   }
};
```

like Python decorator

generic attribute

generic return type

generic method argument

In a class template the data types are defined as generation parameters. When defining instances, concrete data types are specified so that the compiler generates objects of the corresponding type according to the specifications of the template.

```
GenericDataTypeClass<double> DTClass;
DTClass.setValue(6.3);
std::cout<<"value " << DTClass.getValue() <<std::endl;
```

The type must be specified with the instantiation!

Templates with more than one generic data type

Several data types can be declared as generic.

For example, if you deliberately want to support different data types in a function or class.
Note: `typename` and `class` mean the same here...

```
template <typename T, class S>
T maximumHeterogene(T value_a, S value_b)
{
   if(value_a> (T) value_b)return(value_a);
   else return ((T) value_b);
};
```

```
double a=3.2, b=33.2;
int32_t c=342,d=432;

std::cout<<"max ist " << maximumHeterogene(a,c)<<std::endl;
```

Approach 1: class

```cpp
class led
{
public:
   // Use convenient class-specific typedefs.
   using port_type = std::uint8_t;
   using bval_type = std::uint8_t;
   // The led class constructor.

   explicit led(const port_type p, const bval_type b)
         : port(p),
          bval(b)
   {
      // Set the port pin value to low.
      *reinterpret_cast<volatile bval_type*>(port) &= static_cast<bval_type>(~bval);

      // Set the port pin direction to output.

      // Note that the address of the port direction
      // register is one less than the address
      // of the port value register.
      const auto pdir = static_cast<port_type>(port - 1U);

      *reinterpret_cast<volatile bval_type*>(pdir) |= bval;
   }

   auto toggle() const -> void
   {
      // Toggle the LED via direct memory access.
      *reinterpret_cast<volatile bval_type*>(port) ^= bval;
   }

private:
   // Private member variables of the class.
   const port_type port;
   const bval_type bval;
};
```

class defintion

```cpp
namespace
{
   // Create led_b5 on portb.5.
   auto led_b5 = led  {mcal::reg::portb, mcal::reg::bval5 };
}
```

initialisation

```cpp
namespace mcal
{
  namespace reg
  {
    constexpr std::uint8_t portb = 0x25U;

    constexpr std::uint8_t bval0 = 0x01U;
    constexpr std::uint8_t bval1 = 0x01U << 1U;
    constexpr std::uint8_t bval2 = 0x01U << 2U;
    constexpr std::uint8_t bval3 = 0x01U << 3U;
    constexpr std::uint8_t bval4 = 0x01U << 4U;
    constexpr std::uint8_t bval5 = 0x01U << 5U;
    constexpr std::uint8_t bval6 = 0x01U << 6U;
    constexpr std::uint8_t bval7 = 0x01U << 7U;
  }
}
```

port defintion

Approach 2: Template class

two generic attributes

```cpp
template<typename port_type,
       typename bval_type,
       const port_type port,
       const bval_type bval>
class led_template
{
public:
   led_template()
   {
      // Set the port pin to low.
      *reinterpret_cast<volatile bval_type*>(port) &= static_cast<bval_type>(~bval);

      // Set the port pin to output.
      *reinterpret_cast<volatile bval_type*>(port - 1U) |= bval;
   }

   void toggle(void) const
   {
      // Toggle the LED.
      *reinterpret_cast<volatile bval_type*>(port) ^= bval;
   }
};
```

Template class defintion

```cpp
int main()
{
   // Toggle led_b5 forever.
   for(;;)
   {
      led_b5.toggle();
   }
}
```

```cpp
namespace
{
  // Create led_b5 at port B, bit position 0.
  const led_template<std::uint8_t, std::uint8_t, mcal::reg::portb, mcal::reg::bval5>  led_b5;
}
```

initialisation

Templates vs Non-template implementation

Example of led.cpp driver by C. Kormanyos

Class version	Code size `main()` [byte]	RAM size `led_b5` [byte]	Runtime `for(;;)`-loop [μs]
Non-template	36	2	0.44
Template	16	0	0.31

```
template<typename port_type,
        typename bval_type,
        const port_type port,
        const bval_type bval>
class led_template
{
public:
   led_template()
   {
      // Set the port pin to low.
      *reinterpret_cast<volatile bval_type*>(port) &= static_cast<bval_type>(~bval);

      // Set the port pin to output.
      *reinterpret_cast<volatile bval_type*>(port - 1U) |= bval;
   }

   void toggle(void) const
   {
      // Toggle the LED.
      *reinterpret_cast<volatile bval_type*>(port) ^= bval;
   }
};
```

```
class led
{
public:
   // Use convenient class-specific typedefs.
   using port_type = std::uint8_t;
   using bval_type = std::uint8_t;
   // The led class constructor.

   explicit led(const port_type p, const bval_type b)
         : port(p),
           bval(b)
   {
      // Set the port pin value to low.
      *reinterpret_cast<volatile bval_type*>(port) &= static_cast<bval_type>(~bval);

      // Set the port pin direction to output.

      // Note that the address of the port direction
      // register is one less than the address
      // of the port value register.
      const auto pdir = static_cast<port_type>(port - 1U);

      *reinterpret_cast<volatile bval_type*>(pdir) |= bval;
   }

   auto toggle() const -> void
   {
      // Toggle the LED via direct memory access.
      *reinterpret_cast<volatile bval_type*>(port) ^= bval;
   }

private:
   // Private member variables of the class.
   const port_type port;
   const bval_type bval;
};
```

Conclusion of templates

- Use of Templates can reduce code size → execution time = efficiency
- Use of Templates increase reusability

Hint: This can be used in a C++ project where primarily C Technologies are used!

STL = Standard Template Library

STL = Standard Template Library

An essential feature of C++ is the STL, which has been developed since the beginning of C++ and has been extended with each standard renewal.

An important basis of the library are the templates.

The library essentially comprises 4 element types:

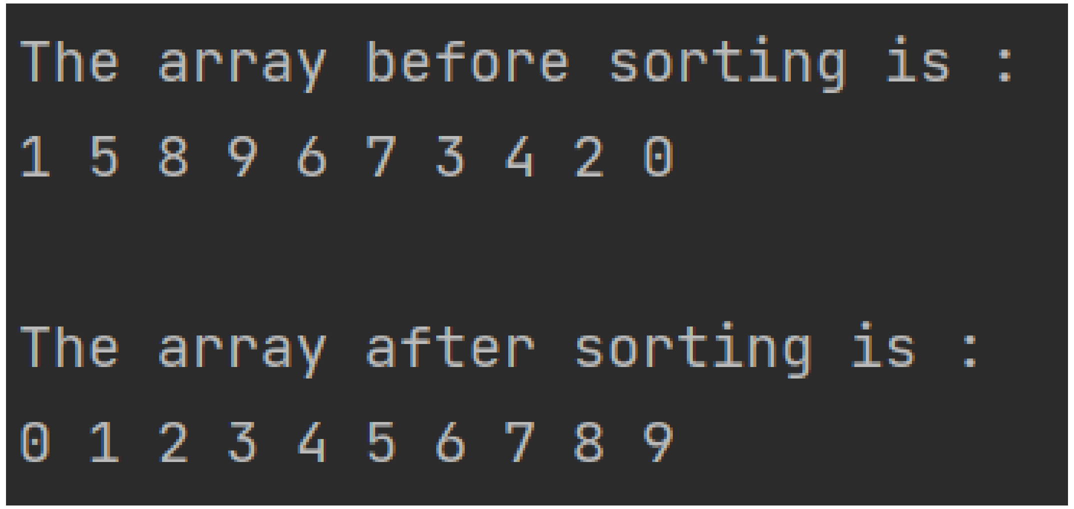
- Algorithms `#include <algorithm>`
- Container
- Functions
- Iterators

Typically, you always have to **include** the header file of the library functions you want to use.

STL: sort

Sort functions are among the algorithms which can be found in the STL.
The `sort` is a basic algorithm:
https://www.cplusplus.com/reference/algorithm/sort/
Here `algorithm` must be included!

```
int arrayA[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
// size of the array
int asize = sizeof(arrayA) / sizeof(arrayA[0]);
cout << "The array before sorting is : \n";
// print the array
show(arrayA, asize);
// sort from STL , sorts the array
sort(arrayA, arrayA + asize);
cout << "\n\nThe array after sorting is :\n";
// print the array after sorting
show(arrayA, asize);
```


```
The array before sorting is :
1 5 8 9 6 7 3 4 2 0

The array after sorting is :
0 1 2 3 4 5 6 7 8 9
```

Hint: show is a self implemented print (cout) function

What we have learned

- We made the transition from structures to classes.

- We covered the principle of classes.
- We treated namespaces.
- We covered class inheritance.
- We treated aggregation and composition.

- We covered C++ specifiers that help make C++ code more efficient and secure.

- We learned what templates are and why they are a special technology of C++.

Structure of the lessons

Self study from last time

Learning Objectives

OOP with C++
- Structures
- Unions
- From C to C++
- Classes
- Namespaces
- Inheritance
- Aggregation and composition

Optimasation
- const
- inline
- constexpr
- consteval

Templates
- Function template
- Class template
- STL = Standard Template Library

Self study

Self study

02_1_CPP_BlinkyDelayNotBlocking

Expand project: CPP_Blinky_02_01 or setup a new project.

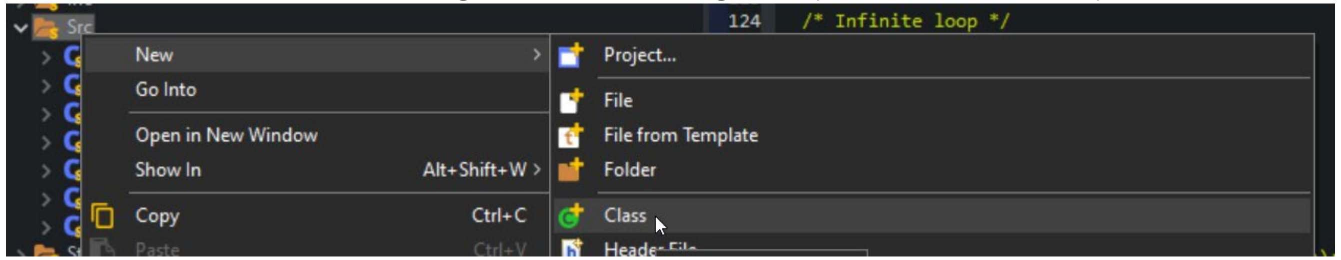
NoneBlockSystemTickDelay:
Use the SystemTick timer to implement a software delay that does not block. (see exercise 1_5)

STM32H7Led:
Use the HAL library to control the LEDs in STM32H7Led
Caution we need 2 constructors for STM32H7Led:

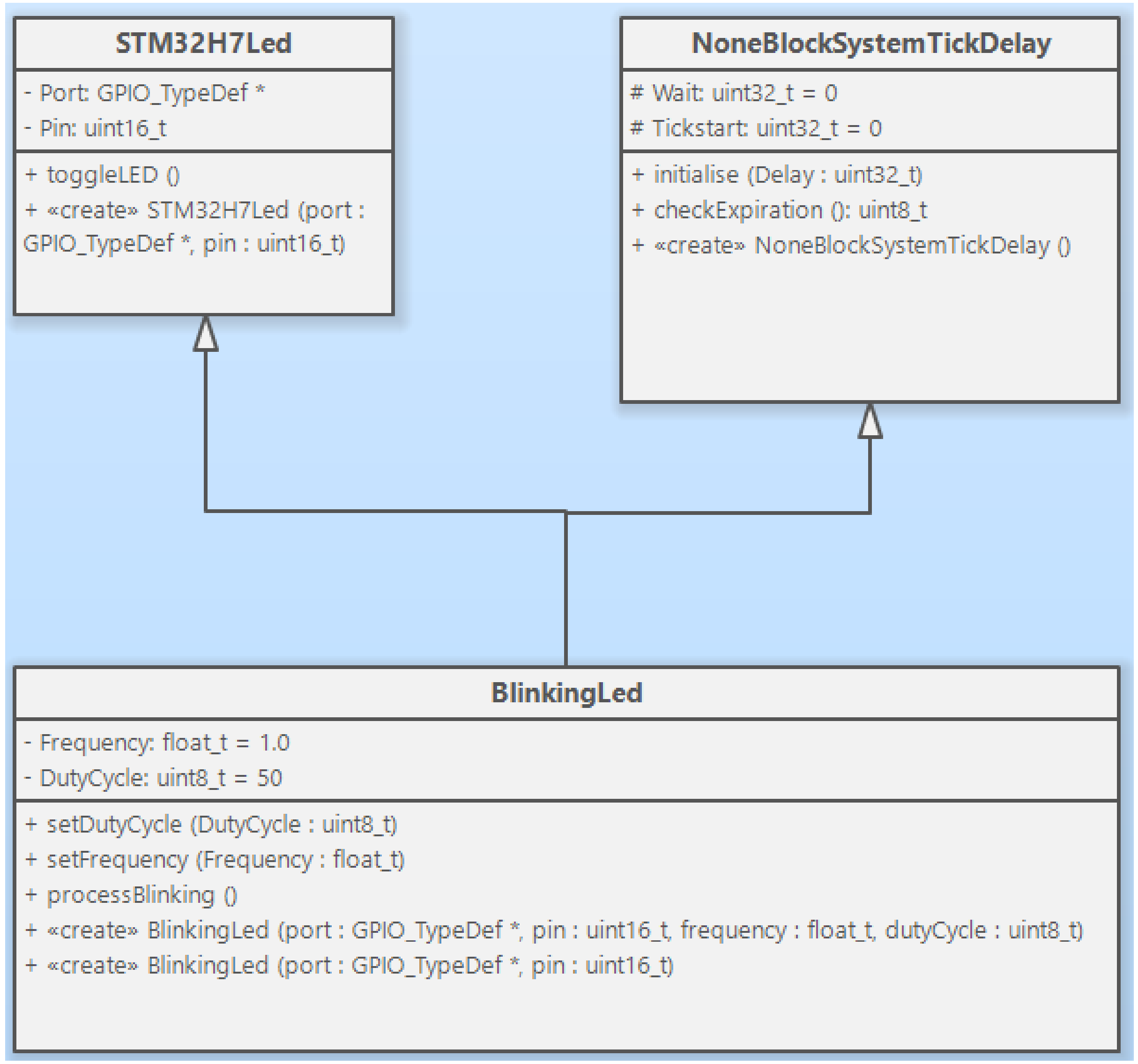
- One without initialization parameters.
- One with initialization parameters!

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…”



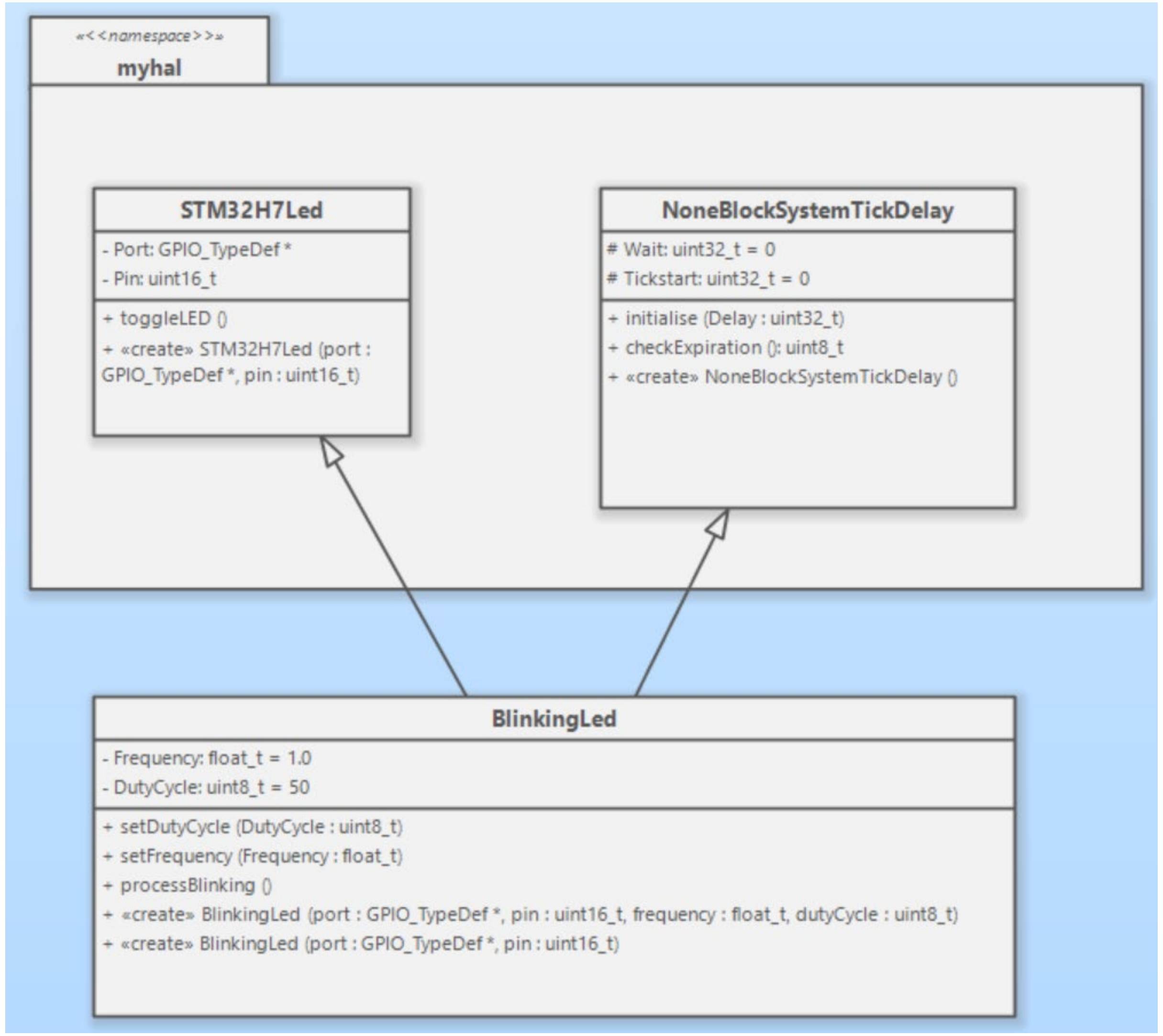
```
STM32H7Led
- Port: GPIO_TypeDef *
- Pin: uint16_t
+ toggleLED ()
+ «create» STM32H7Led (port : GPIO_TypeDef *, pin : uint16_t)

NoneBlockSystemTickDelay
# Wait: uint32_t = 0
# Tickstart: uint32_t = 0
+ initialise (Delay : uint32_t)
+ checkExpiration (): uint8_t
+ «create» NoneBlockSystemTickDelay ()

BlinkingLed
- Frequency: float_t = 1.0
- DutyCycle: uint8_t = 50
+ setDutyCycle (DutyCycle : uint8_t)
+ setFrequency (Frequency : float_t)
+ processBlinking ()
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t, frequency : float_t, dutyCycle : uint8_t)
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t)
```

02_2_CPP_BlinkyNamespace

Expand project: CPP_Blinky_02_02.

Define a namespace: “myhal”

And include STM32H7Led and NoneBlockSystemTickDelay into this namespace.

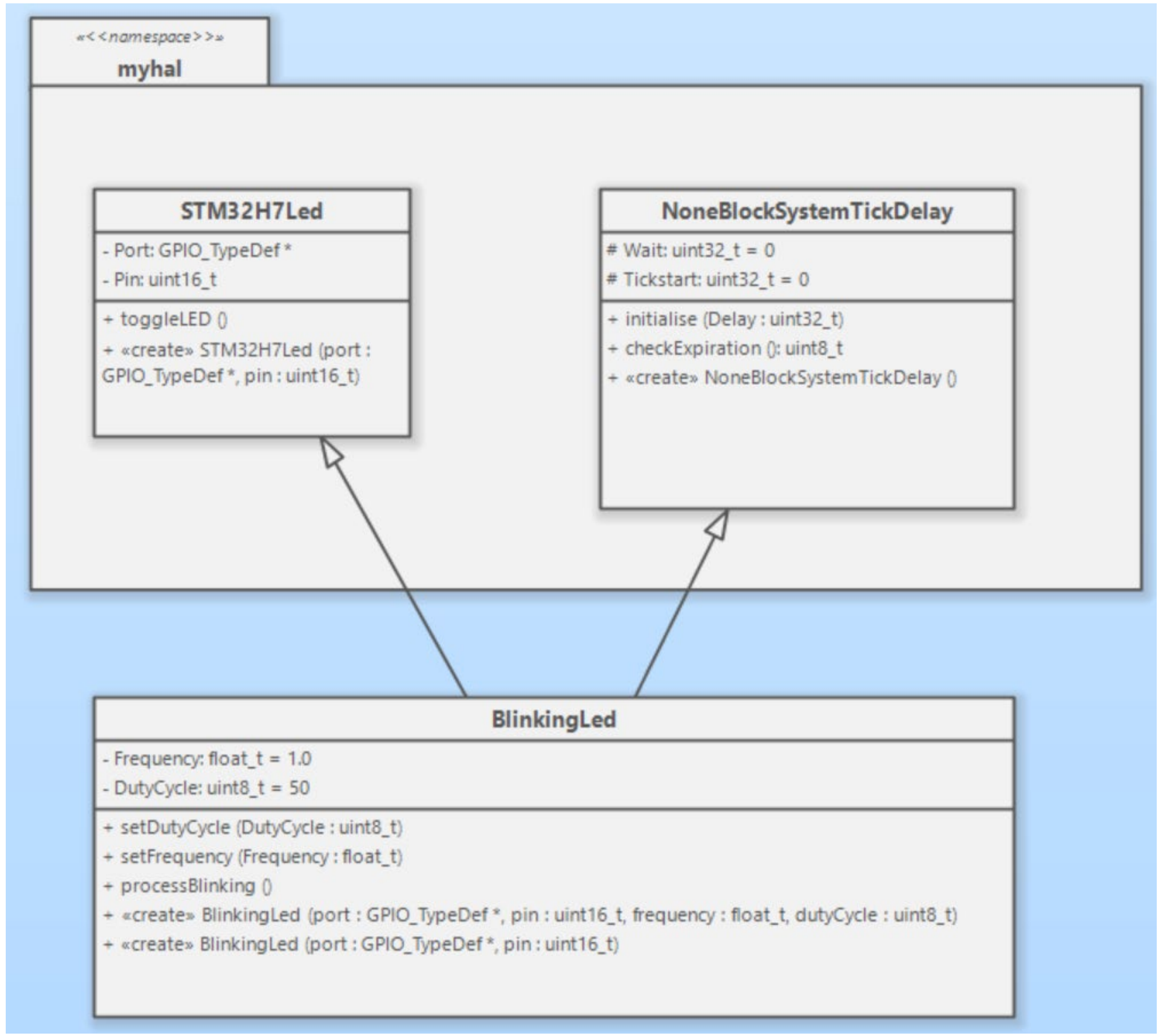

«<<namespace>>»
myhal
STM32H7Led
- Port: GPIO_TypeDef *
- Pin: uint16_t
+ toggleLED ()
+ «create» STM32H7Led (port : GPIO_TypeDef *, pin : uint16_t)
NoneBlockSystemTickDelay
# Wait: uint32_t = 0
# Tickstart: uint32_t = 0
+ initialise (Delay : uint32_t)
+ checkExpiration (): uint8_t
+ «create» NoneBlockSystemTickDelay ()
BlinkingLed
- Frequency: float_t = 1.0
- DutyCycle: uint8_t = 50
+ setDutyCycle (DutyCycle : uint8_t)
+ setFrequency (Frequency : float_t)
+ processBlinking ()
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t, frequency : float_t, dutyCycle : uint8_t)
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t)

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!


«<<namespace>>»
myhal
STM32H7Led
- Port: GPIO_TypeDef *
- Pin: uint16_t
+ toggleLED ()
+ «create» STM32H7Led (port : GPIO_TypeDef *, pin : uint16_t)
NoneBlockSystemTickDelay
# Wait: uint32_t = 0
# Tickstart: uint32_t = 0
+ initialise (Delay : uint32_t)
+ checkExpiration (): uint8_t
+ «create» NoneBlockSystemTickDelay ()
BlinkingLed
- Frequency: float_t = 1.0
- DutyCycle: uint8_t = 50
+ setDutyCycle (DutyCycle : uint8_t)
+ setFrequency (Frequency : float_t)
+ processBlinking ()
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t, frequency : float_t, dutyCycle : uint8_t)
+ «create» BlinkingLed (port : GPIO_TypeDef *, pin : uint16_t)

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: CPP_Template_02_05.

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

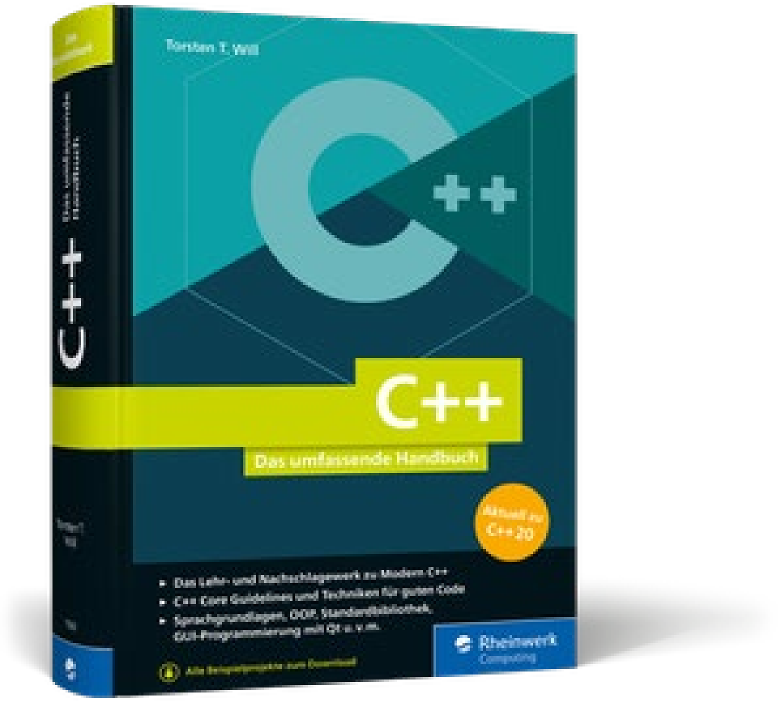
Topics for the next time

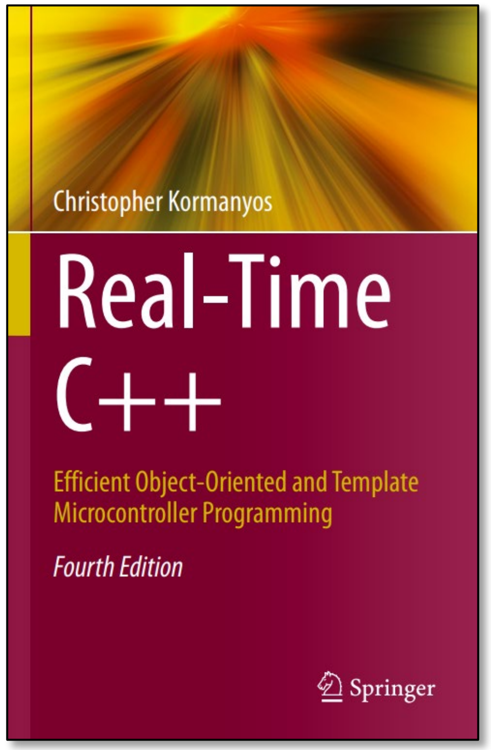
- Datatype auto
- Containers
- **Dynamic memory allocation part 1**
- IKS01A3 Sensorboard
- **Cout on Microcontroller**

Thank you for your attention and cooperation

Part 1 Reference

Appendix: optional Literature




Christopher Kormanyos
Real-Time C++
Efficient Object-Oriented and Template Microcontroller Programming
Fourth Edition
Springer

C von A bis Z

Online version for free

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

PDF version with academic account for free

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