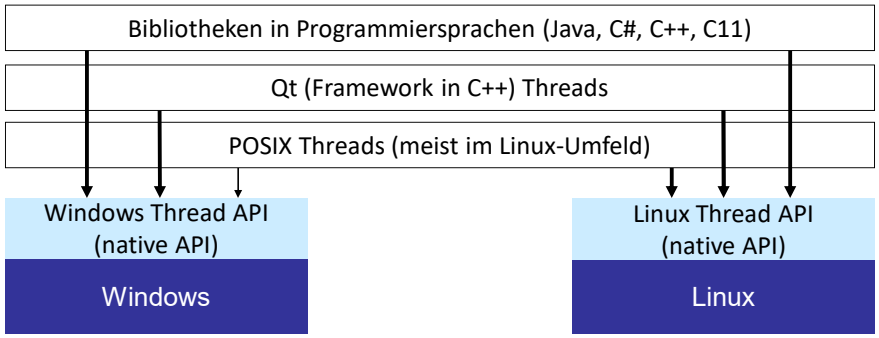
Kurseinheit 12:
Odds & Ends

1. Threads
2. Sicheres Programmieren
3. Testen mit GoogleTest

1. Threads

Wiederholung

Threads wurden bereits in Ing.-Inf. (Programmiersprache C) behandelt (Windows Thread API).



Meist wird nicht die native API der Betriebssysteme verwendet, sondern eine andere Bibliothek, welche intern dann auf die native API zugreift. In dieser LV soll **kurz** auf die **C++ Thread API** eingegangen werden. Diese steht seit C++11 zur Verfügung.

thread Klasse

Member types

id	Thread id (public member type)
native_handle_type	Native handle type (public member type)

Member functions

(constructor)	Construct thread (public member function)
(destructor)	Thread destructor (public member function)
operator=	Move-assign thread (public member function)
get_id	Get thread id (public member function)
joinable	Check if joinable (public member function)
join	Join thread (public member function)
detach	Detach thread (public member function)
swap	Swap threads (public member function)
native_handle	Get native handle (public member function)
hardware_concurrency [static]	Detect hardware concurrency (public static member function)

Native_handle_type würde unter Windows den nativen HANDLE (siehe Ing.-Inf.) zurückgeben.

Instanziierung eines Thread-Objektes

std::thread::thread C++11

```
default (1)          thread() noexcept;
initialization (2)   template <class Fn, class... Args>
                     explicit thread (Fn&& fn, Args&&... args);
copy [deleted] (3)   thread (const thread&) = delete;
move (4)             thread (thread&& x) noexcept;
```

Neben dem leeren Default-Konstruktor, dem Kopier- und Move-Konstruktor gibt es noch einen expliziten Konstruktor. Dieser erscheint auf den ersten Blick aufgrund der **variadic** Parametern (args) und der Templateparameter als sehr komplex. Praktische Beispiele folgen. Der Thread wird **automatisch** gestartet!

Es gibt **drei Möglichkeiten** wie ein Thread-Objekt mit diesem Konstruktor instanziiert wird:

- Mit einem Funktionszeiger (wie in Ing.-Inf.) + variadic Parametern
- Mit einem Objekt + variadic Parametern
- Mit einem Lambda + variadic Parametern

Möglichkeit 1: Instanziierung eines Thread-Objektes mit Funktionszeiger

```
#include <thread>
//…

void vIncrementGlobal(uint32_t u32Max);
```

Instanziierung eines Thread-Objektes (Thread startet automatisch)

```
 std::thread tObj1(&vIncrementGlobal, 1000000U);
```

Optional wie bei Arrays bei scanf

Implementierung der Funktion

```
void vIncrementGlobal(uint32_t u32Max)
{
   for (uint32_t u32C = 0U; u32C < u32Max; u32C++)
   {
      u32Global++;
   }
}
```


Möglichkeit 2: Instanziierung eines Thread-Objektes mit einem Objekt

```
#include <thread>

class ThreadObj
{
public:
   void operator()(uint32_t u32Max)
   {
      for (uint32_t u32C = 0U; u32C < u32Max; u32C++)
      {
         u32Global++;
      }
   }
};
```

„Threadfunktion“ ist der ()-Operator

Instanziierung eines Thread-Objektes (Thread startet automatisch)

```
 std::thread tObj2(ThreadObj(), 1000000U);
```

Möglichkeit 3: Instanziierung eines Thread-Objektes mit Lambda-Funktion

```
#include <thread>

uint32_t u32Global = 0U;

int main(void)
{
   auto lamda_function = [](uint32_t u32Max)
   {
      for (uint32_t u32C = 0U; u32C < u32Max; u32C++)
      {
         u32Global++;
      }
   };

   Instanziierung eines Thread-Objektes (Thread startet automatisch)
   std::thread tObj3(lamda_function, 1000000U);

//..
}
```

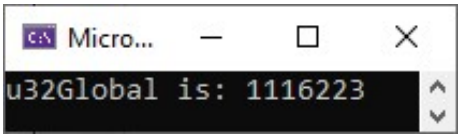

Synchronisation mit join

```
std::thread tObj1(&vIncrementGlobal, 1000000U);
std::thread tObj2(ThreadObj(), 1000000U);
std::thread tObj3(lamda_function, 1000000U);

tObj1.join();
tObj2.join();
tObj3.join();

std::cout << "u32Global is: " << u32Global << std::endl;
```

Debug

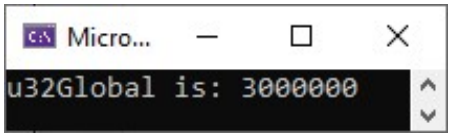


/Od

Gleichzeitiger Zugriff nicht verhindert!
`u32Global++;`

?

Release



/O2

Optimiert die Schleife weg!
`u32Global += 1000000U;`

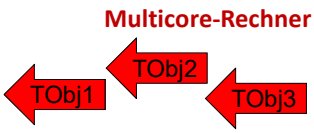
Herausforderung: Critical Sections

```
u32Global++;
```

Critical Section: Gleichzeitiger Zugriff auf eine gemeinsame Ressource (hier u32Global)

Generierter Assemblercode

```
mov   eax,dword ptr [u32Global (01823D0h)]
add   eax,1
mov   dword ptr [u32Global (01823D0h)],eax
```



Selbst bei einem **Singlecore-Rechner** gibt es auch Probleme (statistisch aber seltener):

TObj1

```
mov   eax,dword ptr [u32Global (01823D0h)]
add   eax,1
```

Scheduler schaltet um

TObj2

```
mov   eax,dword ptr [u32Global (01823D0h)]
add   eax,1
mov   dword ptr [u32Global (01823D0h)],eax
```

Dieser Inkrement geht verloren

TObj1

```
mov   dword ptr [u32Global (01823D0h)],eax
```

Scheduler schaltet um

Mutexe: Sperren des Mehrfachzugriffs - klassisch

```
std::mutex mutex_Counter;
```

Definition eines Mutexes

```
for (uint32_t u32C = 0U; u32C < u32Max; u32C++)
{
   mutex_Counter.lock();
   u32Global++;
   mutex_Counter.unlock();
}
```

Critical Section wird mit lock und unlock geschützt. Kein anderer Thread kann dann Section (Bereich) betreten.

Meist sind Critical Sections deutlich komplexer (Code). Hierin kann dann in Unterfunktionen verzweigt werden. Zudem können **Exceptions** auftreten. Der Block der Critical Sections wird dann verlassen, ein unlock geschieht oft nicht mehr (Programmierfehler). Weitere Threads können diesen Block nicht mehr betreten, ebenso könnte der Mutex noch an anderen Stellen verwendet werden und wäre dann für immer „locked“.

Die klassische Anwendung von lock/unlock bei Mutexen ist **bei komplexeren Programmen fehleranfälliger**, was gravierenden negativen Einfluss auf das Gesamtsystem hat.

Mutexe: Sperren des Mehrfachzugriffs – std::lock_guard

```
std::mutex mutex_Counter;
```
Definition eines Mutexes

```
for (uint32_t u32C = 0U; u32C < u32Max; u32C++)
{
   std::lock_guard<std::mutex> LocalGuardObject(mutex_Counter);
   u32Global++;
}
```

Es wird ein Object der Templateklasse lock_guard instanziiert (LocalGuardObject). Templateparameter ist std::mutex. Im Konstruktor von lock_guard findet ein **lock** des Mutexes mutex_Counter statt. Im Destruktor findet das **unlock** statt.

Sobald LocalGuardObject den Focus verliert (bei jedem Schleifendurchlauf oder bei einer Exception), wird automatisch der Destruktor aufgerufen und der Mutex wieder freigegeben (unlock). Der Programmcode sieht deutlich aufgeräumter aus.

Das Prinzip wurde bereits mit den **Smart Pointern** eingeführt.

Ergebnis mit einem Mutex (std::lock_guard oder klassisch)

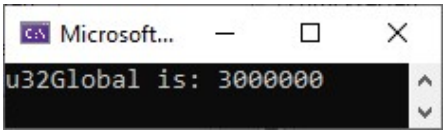
```
std::mutex mutex_Counter;
```

```
std::thread tObj1(&vIncrementGlobal, 1000000U);
std::thread tObj2(ThreadObj(), 1000000U);
std::thread tObj3(lamda_function, 1000000U);

tObj1.join();
tObj2.join();
tObj3.join();

std::cout << "u32Global is: " << u32Global << std::endl;
```

Drei Threadobjekte werden jeweils mit unterschiedlichen Möglichkeiten erzeugt und starten automatisch. Auf die Beendigung wird mittels join im Hauptthread gewartet.



/Od: Debug und Release

TINSTAAFL

"There is no such thing as a free lunch"

Acronym: **TINSTAAFL**

30er/40er Jahre in den USA
Gängige Praxis, dass Bars das Essen umsonst anboten, man musste nur entsprechend Alkohol konsumieren.

Gleiche Problematik bei der Softwareentwicklung:
- Der Preis für sicheres Programmieren (Abfrage der Rückgabewerte, Verwendung von Exceptions) geht zu Lasten der Performance.
- Der Preis für performantes Programmieren (kein Garbagge Collector in C++) geht zu Lasten der Zuverlässigkeit (Memory Leaks).

Exceptions

„Aufgeräumter" Sourcecode

```
void vCallerDiv(void)
{
   try
   {
      int32_t s32Res = s32Div(4, 0);
   }
   catch (std::exception ex)
   {
      std::cout << ex.what() << std::endl;
   }
}
```

Boilerplate Maschinencode

Exceptionhandling wird intern mit vielen if/else und Sprüngen nachgebildet.
- Sehr viel Code
- langsamer

C++ wird in folgenden Anwendungsdomänen hauptsächlich eingesetzt:

High-Performance-Anwendungen auf leistungsfähigen Rechnern (Bildverarbeitung, Facebook, CAD, …)
Hier werden Exceptions verwendet.

Embedded Anwendung auf Microcontrollern (Sensoren, Steuergeräte, …)
Hier werden Exceptions meist nicht angewendet (sind oft verboten).

assert-Makro

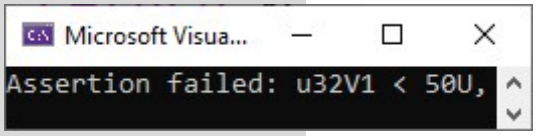
In C++ lassen sich Werte von Variablen einfach mit assert-Makro überprüfen.

```
#include <iostream>
#include <cassert>
uint32_t u32Add(uint32_t u32V1, uint32_t* pu32V2);

int main(void)
{
   uint32_t u32V2 = 42U;
   uint32_t* pu32V3 = nullptr;
   uint32_t u32Ret = u32Add(73U, &u32V2);
   std::cout << "u32Ret is " << u32Ret << std::endl;
   return 0;
}

uint32_t u32Add(uint32_t u32V1, uint32_t* pu32V2)
{
   assert(u32V1 < 50U);
   assert(pu32V2 != nullptr);
   return (u32V1 + *pu32V2);
}
```

Sobald ein assert failed, bricht Programm ab.



assert-Makro: Unterschiede Debug/Release

```
#include <cassert>
```

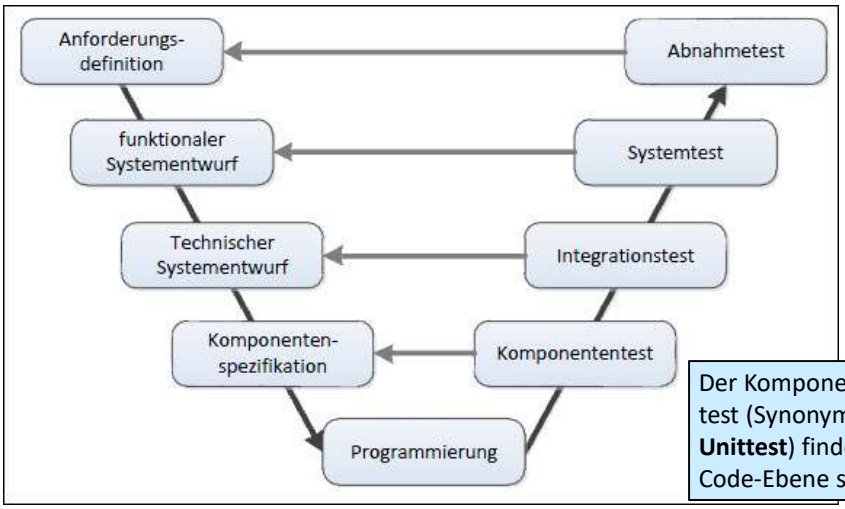
Cpp-Version von assert.h (namespace std)

```
#ifdef NDEBUG
    #define assert(expression) ((void)0)
#else
    _ACRTIMP void __cdecl _wassert(
        _In_z_ wchar_t const* _Message,
        _In_z_ wchar_t const* _File,
        _In_   unsigned       _Line
        );
#define assert(expression) (void)(                       \
    (!!(expression)) ||                                  \
    (_wassert(_CRT_WIDE(#expression), _CRT_WIDE(__FILE__),\
    (unsigned)(__LINE__)), 0)                            \
                                )
#endif
```

Das Makro wirkt sich nur unter Debug aus. In Release wird durch den Präprozessor assert(expression) zu ((void)0), der Compiler erzeugt daraus keinen Maschinencode.

Testen im V-Modell

Bei der Systementwicklung wird oft klassisch nach dem sogenannten V-Modell vorgegangen (siehe spätere Lehrveranstaltungen im Bereich Software Engineering).



Der Komponententest (Synonym ist **Unittest**) findet auf Code-Ebene statt.

https://www.evas.de/leistung/von-der-idee-zur-software/v-modell/

Softwarewerkzeuge für Unittests

Für (fast) jede Programmiersprache (und auch oft für eine Domäne) gibt es kommerzielle und frei verfügbare Softwarewerkzeuge für Unittests.

frei verfügbar			**kommerziell**
CUnit	CppUnit	JUnit	QA Systems Cantata
NUnit	uCUnit	Python Unittest	Parasoft
PHPUnit	DUnit	QUnit	

Oft als xUnit bezeichnet (x ist die Sprache)

Im Rahmen dieser LV soll ein modernes und frei verfügbares Softwarewerkzeug für C/C++ vorgestellt werden: **GoogleTest**

Nutzung von GoogleTest

GoogleTest kann wie folgt in MS Visual Studio verwendet werden:

Der allgemeine Ansatz (für alle IDE):
Sourcen etc. können unter https://github.com/google/googletest/tree/main/googletest heruntergeladen und mit cmake gebaut werden (->lib)

NuGet Package Manager:
In MSVS kann über den NuGet Package Manager (Tools -> NuGet Package Manager) die kompilierte Bibliothek geladen werden.

```
#include "gtest/gtest.h"
int main(int argc, char **argv)
{
   ::testing::InitGoogleTest(&argc, argv);
   RUN_ALL_TESTS();
   return 0;
}
```

Main bei den ersten beiden Ansätzen!

Ist Datentyp von argv geändert?

Über ein vorhandenes Projekttemplate: Siehe folgende Folie! Obiges main schon enthalten!

Verwendung von GoogleTest in MSVS – Vorhandenes Projekttemplate

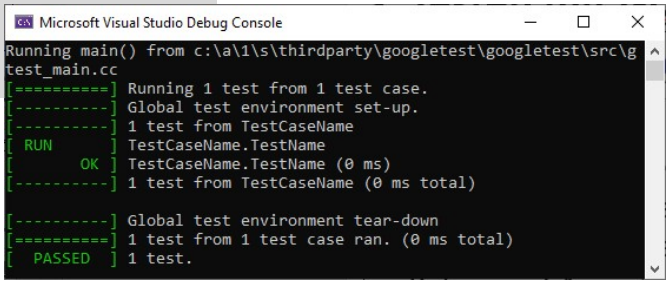
- Add new project
- Templates: C++ - Test -> Google Test -> Next
- Project Name eingeben -> Create
- Consume Google Test as: Dynamic Library, C++ runtime libraries: Link dynamically -> OK
- Release, x64

```
#include "pch.h"

TEST(TestCaseName, TestName)
{
  EXPECT_EQ(1, 1);
  EXPECT_TRUE(true);
}
```

pch: Precompiled Headers -> hier sind **alle h-Dateien zu inkludieren**
main wird intern schon angelegt.

Nach dem Ausführen erscheint das Ergebnis des Tests mit zwei (Dummy-)Checks, die als Macros implementiert sind.



Wichtige Checks in GoogleTest

Es gibt zwei Arten von Checks: **ASSERT_** bricht gesamten Testlauf ab, während **EXPECT_** im Fehlerfall den gesamten Testablauf fortführt.

binär

```
EXPECT_TRUE(condition)

EXPECT_FALSE(condition)

EXPECT_EQ(val1, val2)
EXPECT_NE(val1, val2)

EXPECT_LT(val1, val2)
EXPECT_LE(val1, val2)
EXPECT_GT(val1, val2)
EXPECT_GE(val1, val2)
```

Condition kann beliebige C/C++ Bedingung sein.

C-String

```
EXPECT_STREQ(str1, str2)
EXPECT_STRNE(str1, str2)

EXPECT_STRCASEEQ(str1, str2)
EXPECT_STRCASENQ(str1, str2)

EXPECT_EQ(str, nullptr)
EXPECT_NQ(str, nullptr)
```

String Objects

```
EXPECT_EQ(str1, str2)
EXPECT_NE(str1, str2)
```


Wichtige Checks in GoogleTest

Fließkomma

Rundungsungenauigkeiten!!!

```
EXPECT_FLOAT_EQ(val1, val2)
EXPECT_DOUBLE_EQ(val1, val2)
```

Ungenauigkeit von 4 ULPs bei beiden Vergleichen. ULP: **U**nit in the **L**ast **P**lace

```
EXPECT_NEAR(val1, val2, abs_error)
```

Windows-Unterstützung

```
EXPECT_HRESULT_SUCCEEDED(expression)
EXPECT_HRESULT_FAILED(expression)
```

Exceptions

```
EXPECT_THROW(statement, exception_type)
EXPECT_ANY_THROW(statement)
EXPECT_NO_THROW(statement)
```

Death Assertions

```
EXPECT_DEATH(statement, matcher)
EXPECT_DEATH_IF_SUPPORTED(statement, matcher)
```

Process wird abgebrochen mit einem String als matcher.

Statt **EXPECT_** kann überall auch **ASSERT_** (Gescheiterter Check -> Abbruch) verwendet werden.

Weitere siehe: https://google.github.io/googletest/reference/assertions.html

Begriffe

GoogleTest lehnt sich sehr stark an den Begriffen des International Software Testing Qualifications Board (ISTQB – www.istqb.org) an.
Testfälle (umgangssprachlich „Tests“) sind sogenannten Testsuiten (logisch zusammengehörende Testfälle) zugeordnet. Innerhalb eines Testfalls können sich mehrere Checks befinden. Leider vergibt MS beim Anlegen von test.cpp etwas unpassende Namen.

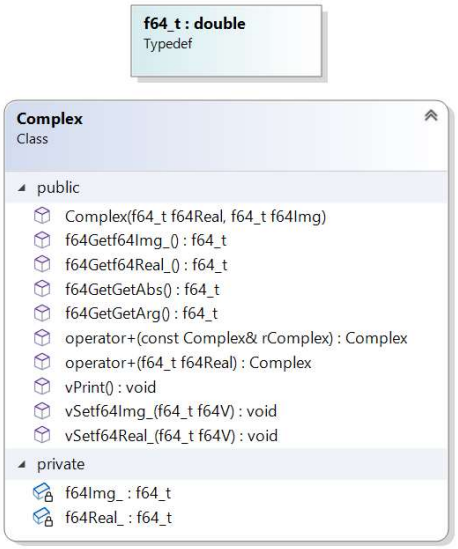
Testsuite **Testfall**

```
#include "pch.h"

TEST(TestCaseName, TestName)
{
  EXPECT_EQ(1, 1);
  EXPECT_TRUE(true);
}
```

Ein Testfall besteht aus:
- Eindeutiger ID (gerne natürlich als Schlüssel „FT_FS_R_42“)
- Vorbedingung (EXPECT_) - Precondition
- Input
- Expected Output (EXPECT_)
- Nachbedingung (EXPECT_) - Postcondition

Einfaches Beispiel Klasse Complex



Klasse aus den ersten Kurseinheiten.
Test über die public-Methoden möglich.

Schwer testbar mit GoogleTest sind Methoden wie vPrint. Output auf der Console!

Ebenso sind private und protected-Methoden von außerhalb nicht aufrufbar.

Manchmal kann es daher notwendig sein, spezielle Methoden zum Testen zur Verfügung zu stellen. Ggf. könnte man dies in einer Kindklasse zur Verfügung stellen und über diese dann testen (greifen dann auf protected Methoden der Elternklasse zu).

DfT: Design for Test

Anpassungen die für das Testen notwendig sind.

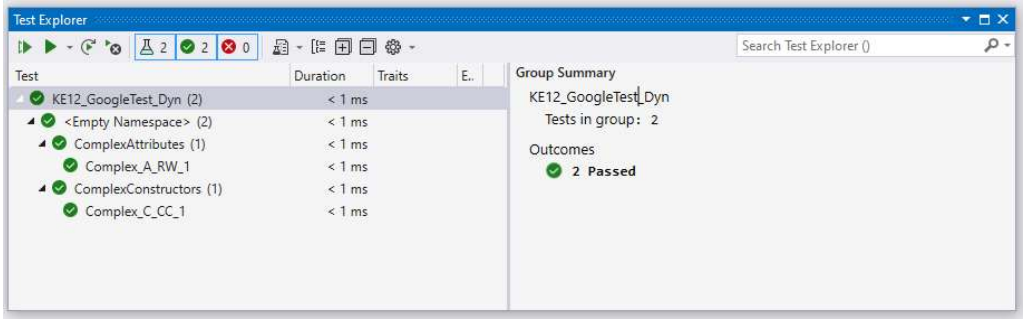
Einfaches Beispiel Klasse Complex

```
TEST(ComplexAttributes, Complex_A_RW_1)
{
  Complex C1(1., 1.);
  //Precondition
  EXPECT_DOUBLE_EQ(1.0, C1.f64Getf64Real_());
  EXPECT_DOUBLE_EQ(1.0,
  //Input
  C1.vSetf64Real_(42.0)
  //Output
  EXPECT_DOUBLE_EQ(42.0
  //Postcondition
  EXPECT_DOUBLE_EQ(1.0,
}
```

```
TEST(ComplexConstructors, Complex_C_CC_1)
{
   Complex C1(-47.11, -42.001);
   //Precondition
   EXPECT_DOUBLE_EQ(-47.11, C1.f64Getf64Real_());
   EXPECT_DOUBLE_EQ(-42.001, C1.f64Getf64Img_());
   //Input
   Complex C2(C1);
   //Output
   EXPECT_DOUBLE_EQ(-47.11, C2.f64Getf64Real_());
   EXPECT_DOUBLE_EQ(-42.001, C2.f64Getf64Img_());
   //Postcondition
   EXPECT_DOUBLE_EQ(-47.11, C1.f64Getf64Real_());
   EXPECT_DOUBLE_EQ(-42.001, C1.f64Getf64Img_());
}
```


MS Test Explorer

Microsoft Visual Studio enthält einen Test Explorer. Mit diesem Tool können die Tests aber auch gestartet und die Testergebnisse visualisiert werden.



Hier zeigt sich auch, wie wichtig die Strukturierung in Testsuiten und Testfällen ist. Im vorliegenden Beispiel gab es zwei Testsuiten mit nur einem Testfall.
Ein Testfall ist nur dann erfolgreich, wenn alle Checks (Expects) erfolgreich sind.

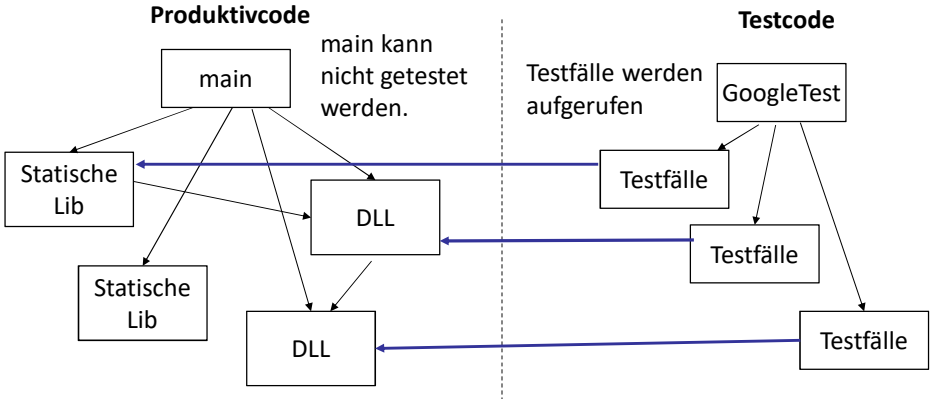
3. Testen mit GoogleTest

No Go

Im vorliegenden Beispiel wurde ein prinzipieller Fehler gemacht:

Produktiv- und Testcode dürfen sich **niemals** in einem Programm befinden.

Das zu testende Programm (Produktivcode) muss in C++ modular mit statischen und dynamischen Libs aufgebaut werden!



ENDE

C++

Mit C++ haben Sie nun eine der anspruchsvollsten Programmiersprachen gelernt!

In Ihrem späteren Berufsleben werden Sie im Bereich:

- Embedded Systems
- High Performance Anwendungen

mit C++ zu tun haben.

Fortsetzung folgt?
Zumindest im INFM ist eine Fortsetzung geplant. In EIM, MMR, CME und MTM wäre dies sicherlich auch notwendig.

https://www.amazon.com/Programming-T-Shirt-Funny-Coding-Tees/dp/B07KPF5493