OBJEKTORIENTIERTES PROGRAMMIEREN MIT C++, SOMMERSEMESTER 2018

Ihr Betreuer: Thomas Kammerer thomas.kammerer@th-nuernberg.de

UNSERE ENTWCKLUNGSUMGEBUNG

UNIT-TEST FRAMEWORK CATCH

Wir werden das Unit-Test-Framework Catch2 verwenden.

https://github.com/catchorg/Catch2

Abbildung 1: Logo catch2 - unit test framwork

Sie finden eine aktuelle Version des Single-Include-Headerfiles von Catch2 im Projektverzeichnis:

PROJECT_HOME\cpp_summer_course_2018\external\unit_test\catch_2

wobei PROJECT_HOME das von Ihnen gewählte Home-Verzeichnis ist.

Prüfen Sie, ob sie catch.hpp im angegebenen Verzeichnis finden.
Gehen Sie auf die Webseite von catch und schauen Sie sich die dazu das Tutorial und ggf. die Reference section an!

ANLEGEN EINES PROJEKTES MIT CODE::BLOCKS

LEERES PROJEKT ANLEGEN

Wir legen ein Projekt an, das eine **statische Library** erzeugt, die von einer **Unit-Testumgebung** aufgerufen und **automatisiert getestet** werden kann. Falls gewünscht, kann auch noch eine normale main() in das Projekt integriert werden. Die Verzeichnisstruktur in unserem Projektverzeichnis soll folgendermaßen aussehen:

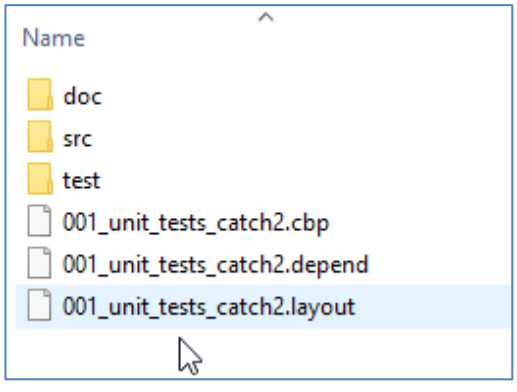

Abbildung 2: Verzeichnisstrukur

doc: Aufgabenbeschreibung und sonstige Dokumentation

src: die Source-Files (Header und Implementierung)

test: die Source-Files für die Tests (test_main.cpp, class_test.cpp)

***.cbp**: Code::Blocks Project file
***.depend**: Build-Dependencies
***layout**: Code::Blocks Layout File (merkt sich, welche Files offen sind usw.)

Legen Sie ein neues Projekt an: File/New/Project…

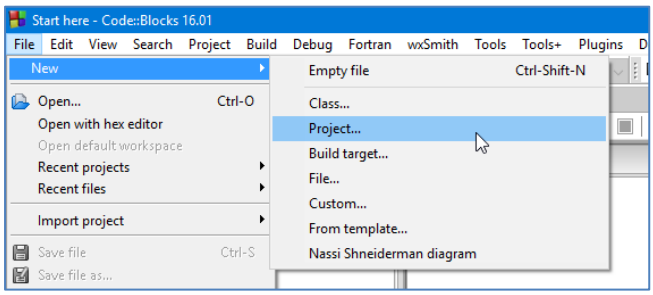

Abbildung 3: Code::Blocks: neues Projekt anlegen

Wählen Sie Empty project

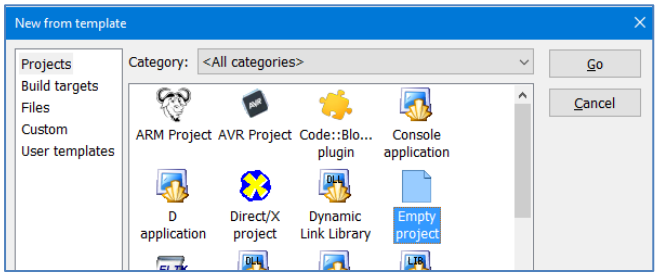

Abbildung 4: Leeres Projekt als Vorlage

Geben Sie jetzt „Project title“ und „Folder to create project in“ ein:

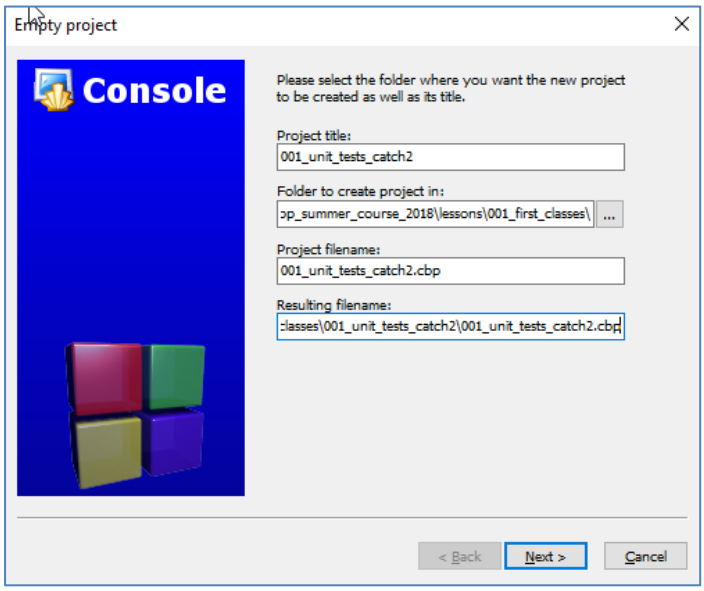

Abbildung 5: Projekt Titel und Ort angeben

Im letzten Schritt können Sie die zu erzeugenden Configurationen angeben und wo die Object- und Output-Files erzeugt werden sollen, hier können Sie einfach auf Finish.

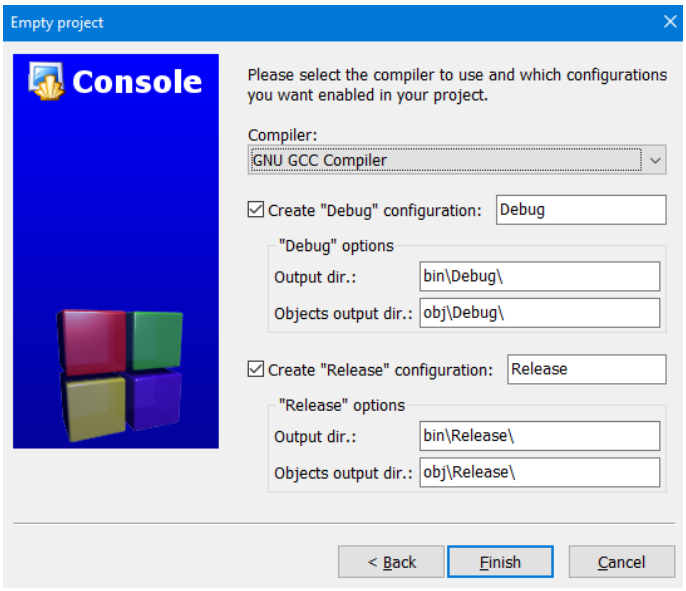

Abbildung 6: Configurationen und Output Directories

Gehen Sie jetzt in das Projektverzeichnis und legen die Unterordner /doc, /src und /test an.

Jetzt müssen wir unsere Konfigurationen anpassen:

Gehen Sie im Projektbaum auf das gerade erzeugte Projekt, drücken die rechte Maustaste und wählen Properties!

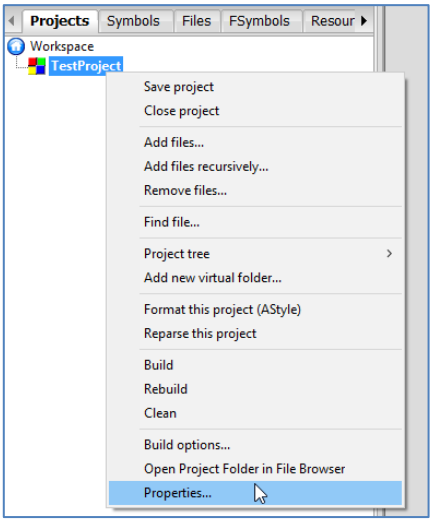

Abbildung 7: Projektbaum -> rechte Maustaste -> Properties

Wählen Sie hier den Tab-Reiter Build targets

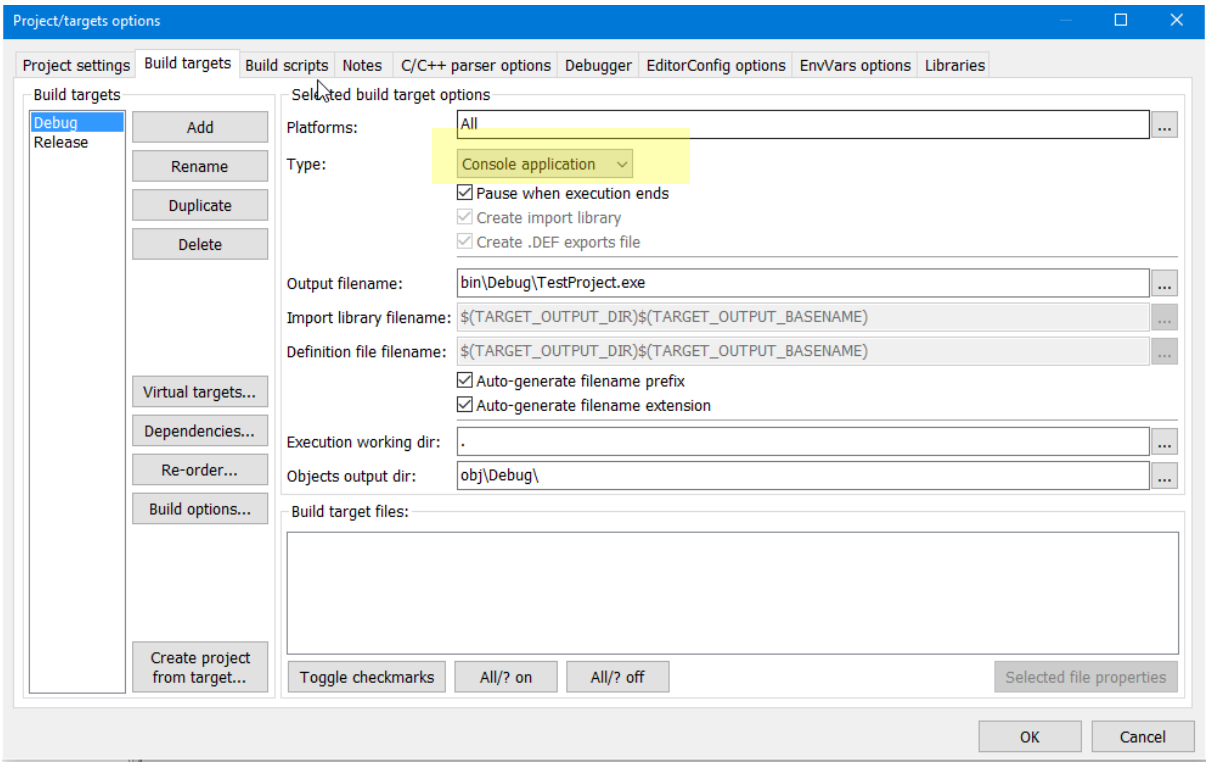

Abbildung 8: Build targets

*Ändern Sie für die Build Targets **Debug** und **Release** zu den Tpye zu **Static library.***

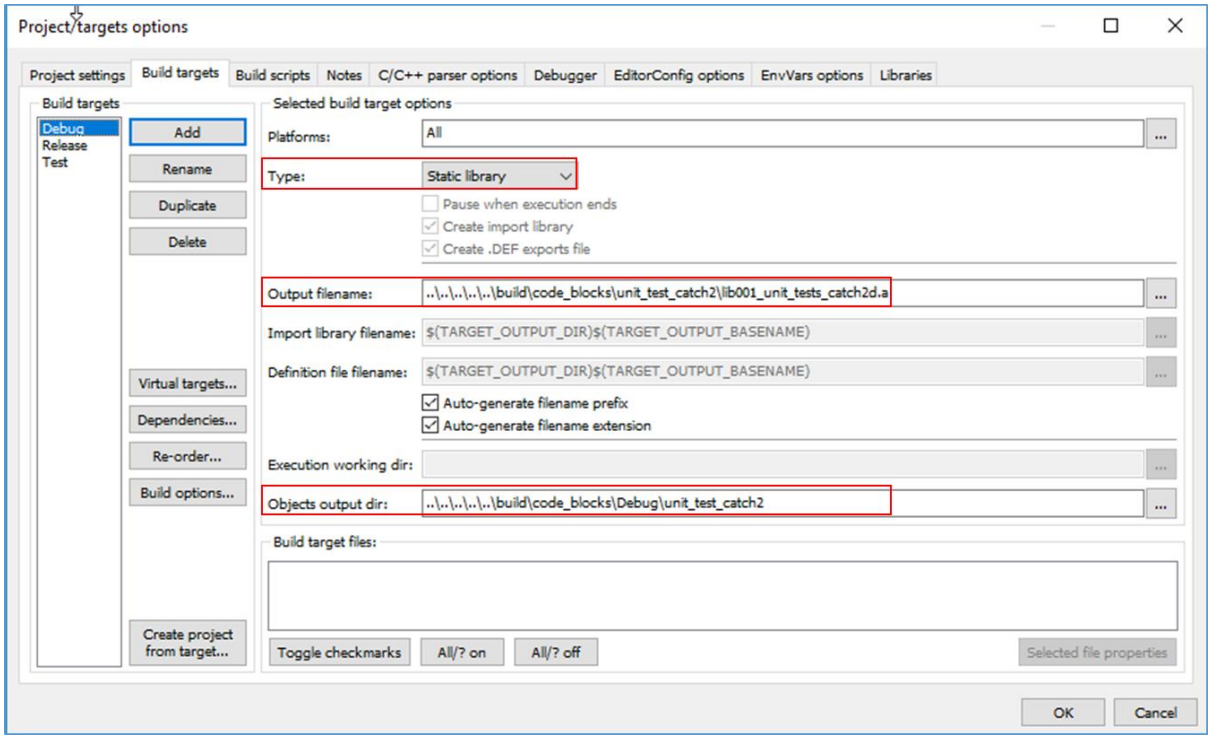

Abbildung 9: Geänderte Target Types und Output Directories für Debug-Configuration

*Ändern Sie für die Build Targets **Debug** und **Release** Output filename in ein Verzeichnis wo Sie ihre Build-Ergebnisse (die Libraries und Executables) erzeugt haben möchten.*

*Ändern Sie für die Build Targets **Debug** und **Release** Objects output dir in ein Verzeichnis, wo die Object-Files erzeugt werden sollen.*

Anmerkung: Es empfiehlt sich, die Build-Ergebnisse nicht in das Source-Verzeichnis zu legen, sondern an einen zentralen Ort außerhalb Ihrer Quelldateien. Das ist unter anderen deshalb vorteilhaft, weil die Build-Ergebnisse nicht in ein Code-Verwaltungssystem aufgenommen werden sollten.

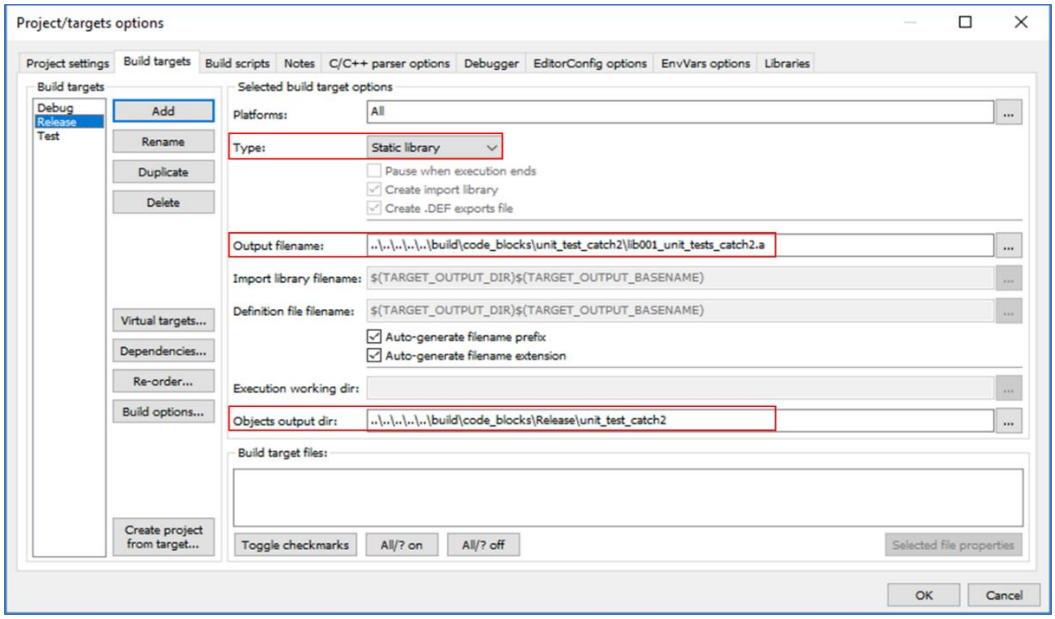

Abbildung 10: Geänderte Target Types und Output Directories für Release-Configuration

*Legen Sie ein mit dem Button Add neues Buid Target **Test** an!*

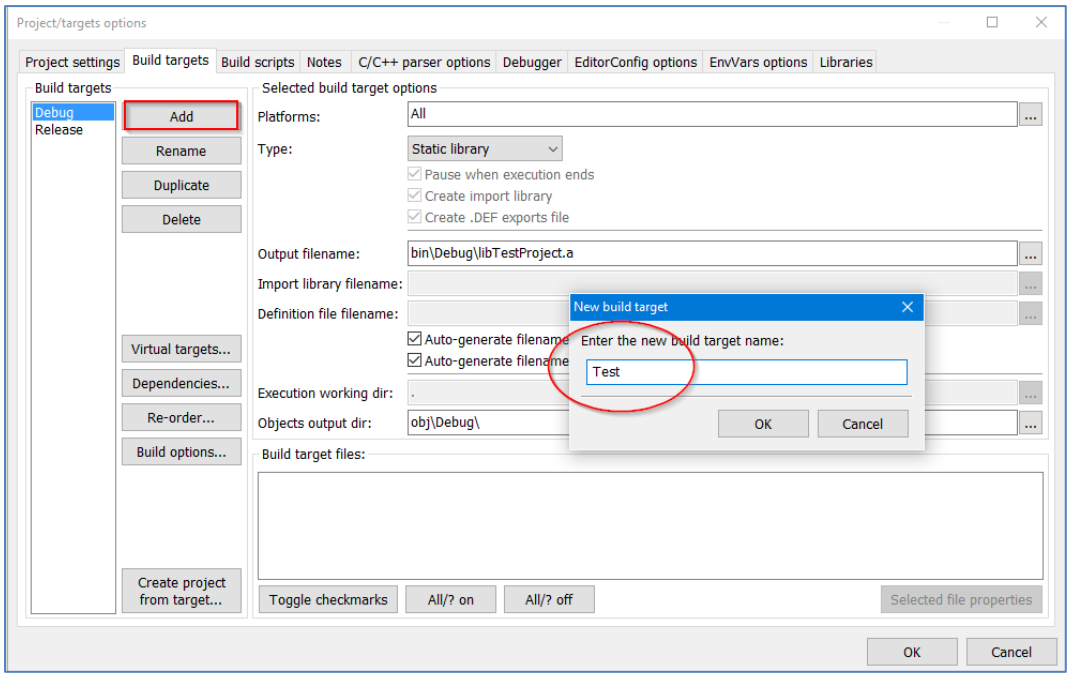

Abbildung 11: Anlelgen eines neuen Build Targets Test

Ändern Sie den Type von Target Test in Console application, markieren Sie „Pause when execution ends“ und geben Sie die richtigen Output Verzeichnisse an!

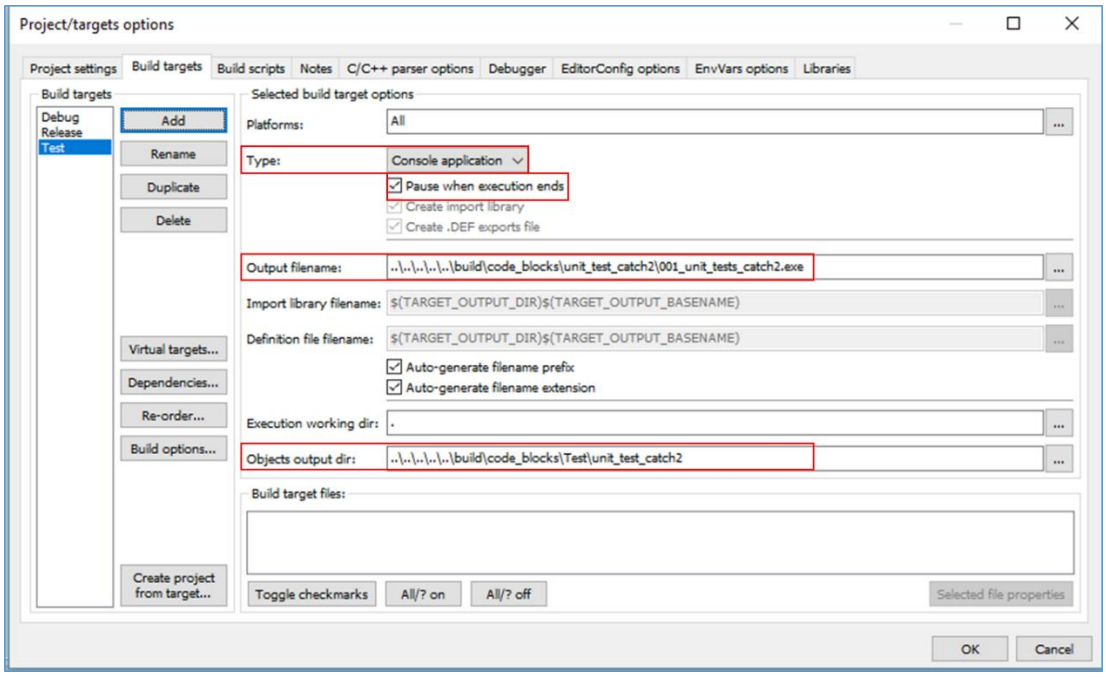

Abbildung 12: Einstellungen für Target Test

*Legen Sie nun ein virtuelles Target **All** an!*

1. *Drücken Sie Virtual targets…*
2. *Im Dialog Add*
3. *All eingeben*
4. *Haken Sie alle Build Targets an (Debug, Release und Test)*

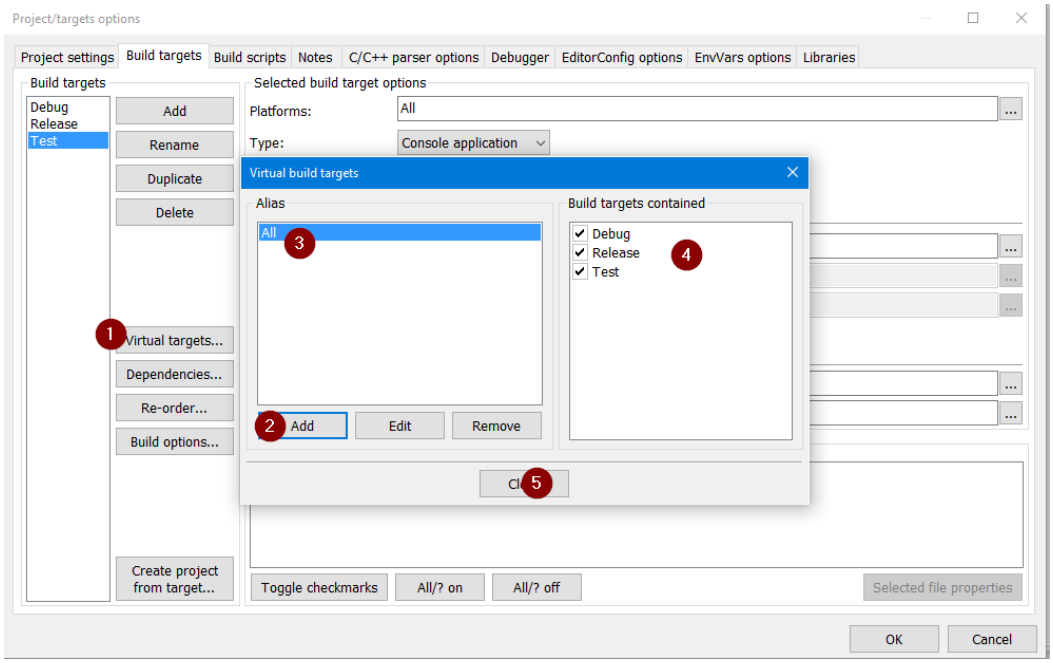

Abbildung 13: virtuelles Build Target All anlegen

Jetzt muss das main-file für den Unit-Test angelegt werden:

Wählen Sie File/New/Empty file!

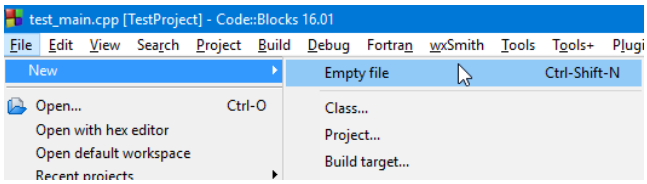

Abbildung 14: neues leeres File anlegen

Gehen Sie in das Unterverzeichnis test (falls nicht da jetzt anlegen!) und speichern sie die Datei als test_main.cpp ab.

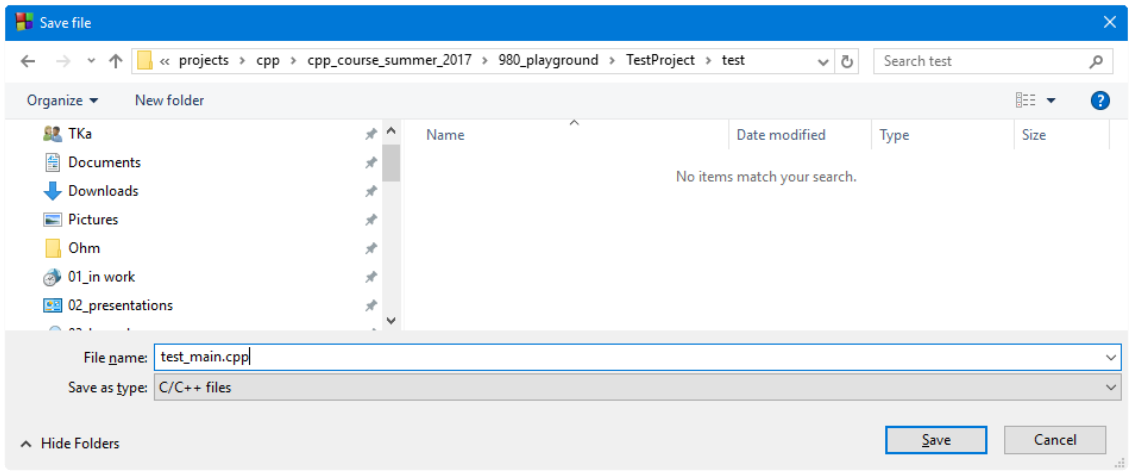

Abbildung 15: Unter ./test/test_main.cpp anlegen

Wählen Sie das File nur für das Target Test aus!

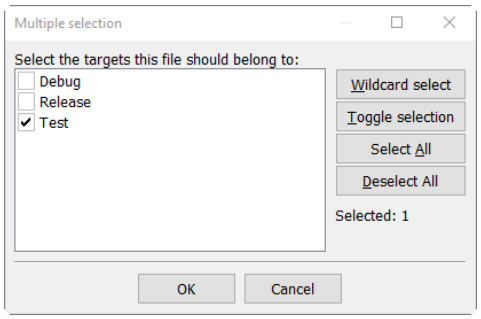

Abbildung 16: Auswahl für welche Build Targets test_main gebaut wird

Schreiben Sie in das test_main.cpp folgende Zeilen:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

test_main.cpp | test_eventtarget.cpp

```
1  #define CATCH_CONFIG_MAIN
2  #include "catch.hpp"
3
```

Abbildung 17: test_main.cpp

Konfigurieren Sie Code::Blocks so, dass für jedes Projekt catch.hpp gefunden wird:

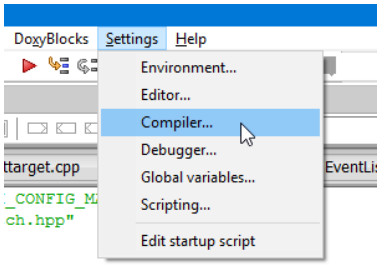

Abbildung 18: Code::Blocks Global Compiler Settings

Überprüfen Sie die Compiler settings:

- -std=c++14 (C++ 14 Standard verwenden)
  (wenn ihr Compiler das nicht kann, dann –std=c++11 setzen!)
- -Wall (alle Compiler Warnings)

Search directories/compiler zu
<Project Workspace>/cpp_course_summer_2017_students/998_sofware/catch/single_include
setzen!

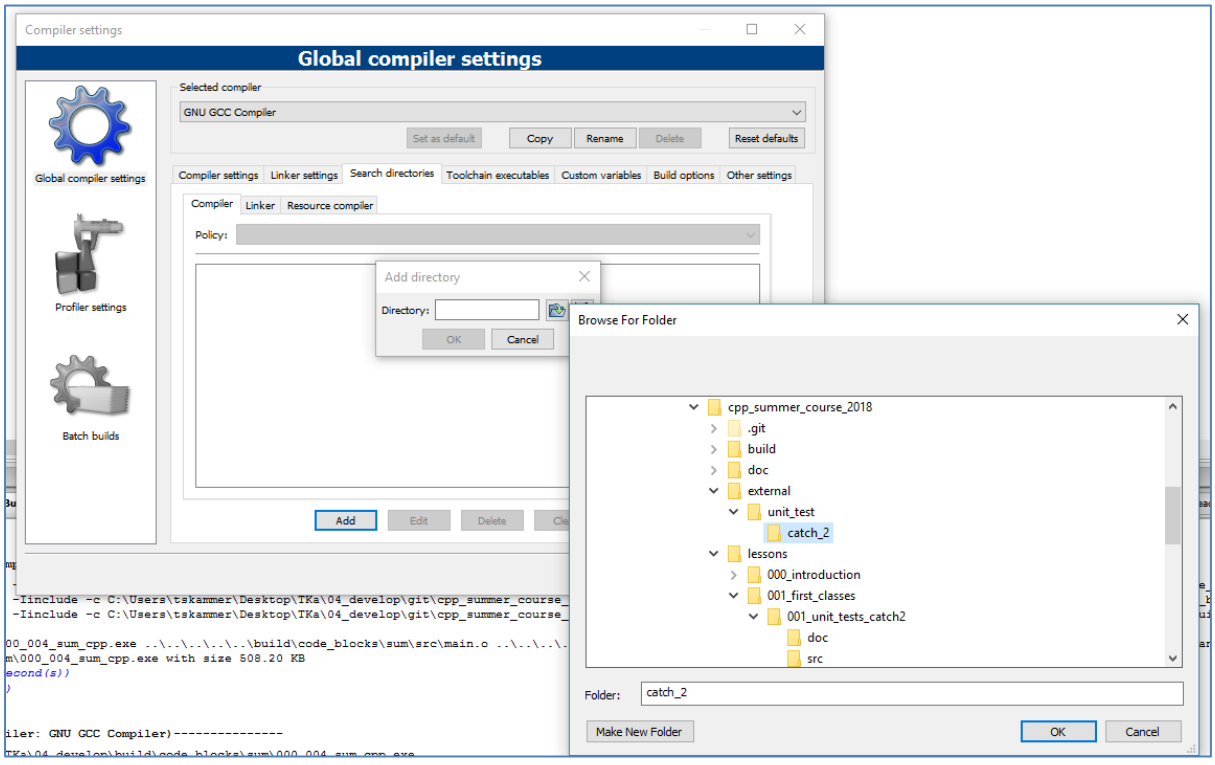

Abbildung 19: Global Settings: Search directories

Legen Sie Ihre erste Klasse an:

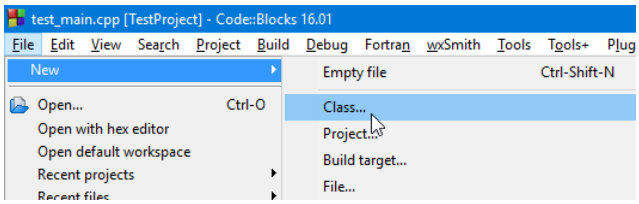

Abbildung 20: neue Klasse anlegen

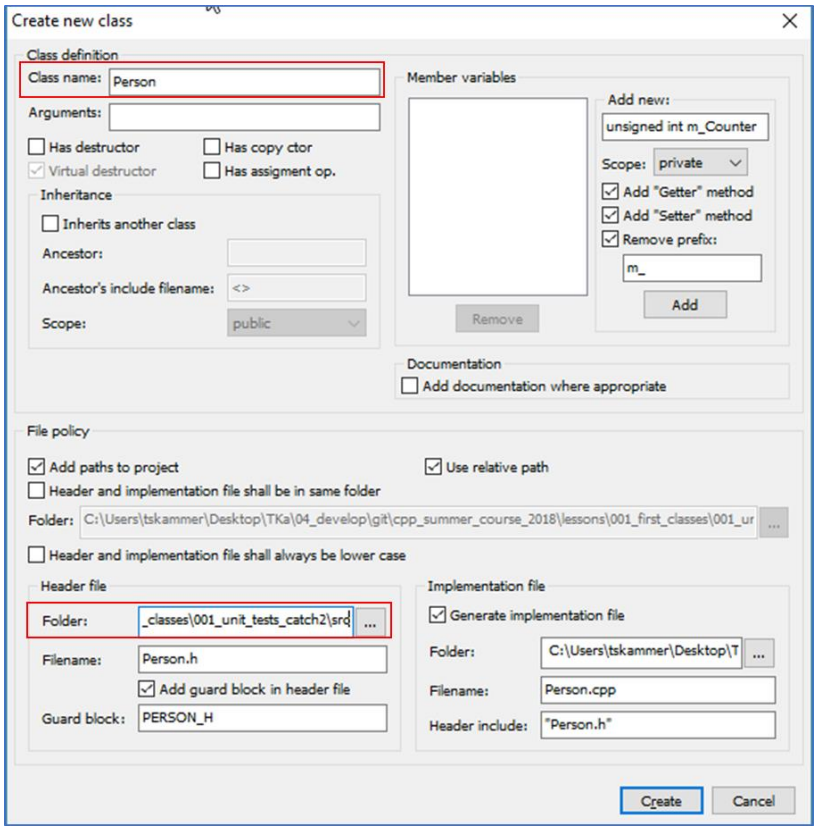

Abbildung 21: Neue Klasse (hier Person) anlgegen

Wählen Sie hier als Build-Target Debug und Release!

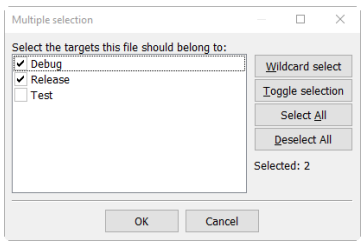

Abbildung 22: die Klasse nur für Debug und Release Target auswählen

Jetzt können wir das Debug-Target übersetzen.

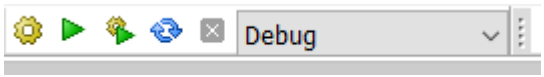

Abbildung 23: Debug Target auswählen und übersetzen

Jetzt binden wir noch erzeugte Debug Library in das Test-Target ein:

- *Öffnen Sie die Build-Options*
- *wählen Sie das Test Target aus*
- *Fügen sie mit add die erzeugte Library unter bin\Debug\libTestProject.a hinzu.*

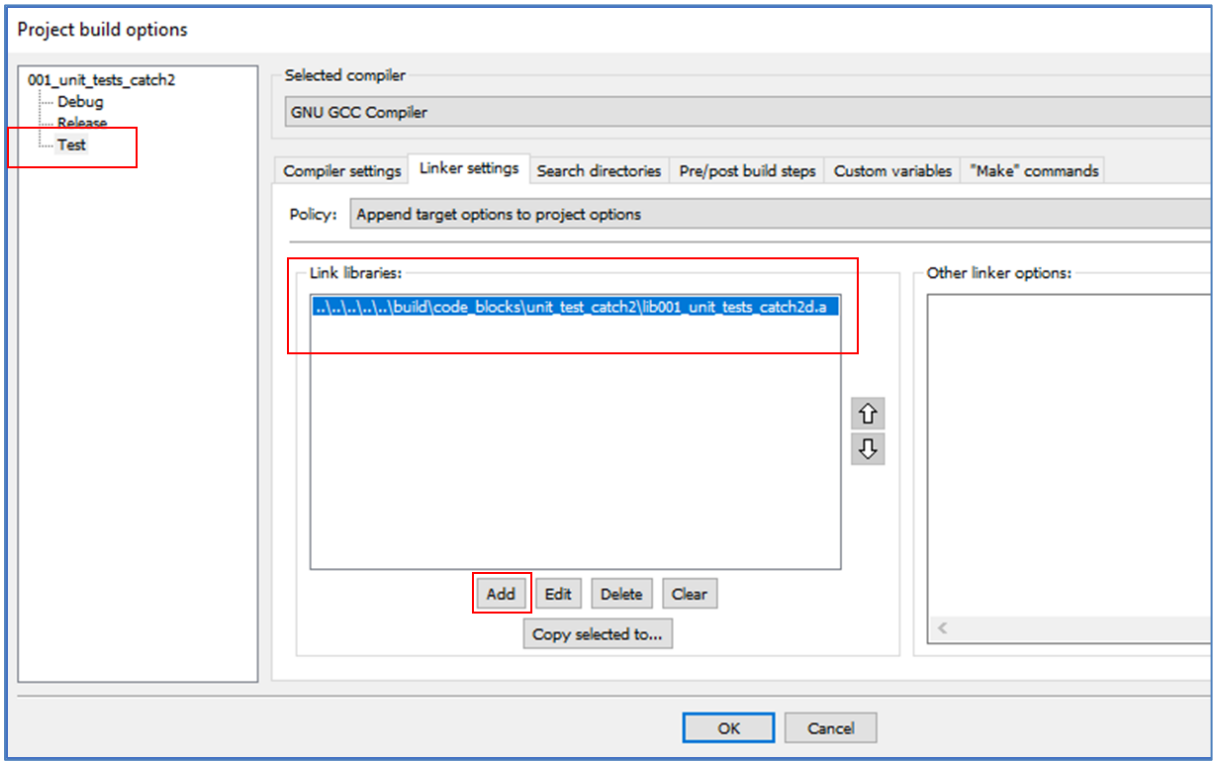

Abbildung 24: Library ins Test-Target einbinden

OPTIONAL: nur wenn wir wirklich Debuggen wollen!

Um auch im Test Debuggen zu können müssen wir noch Debug-Symbole erzeugen:

Hierzu unter Compiler settings den Compilerschalter aktivieren. (ACHTUNG: Da hier die Übersetzungzeit sehr groß wird, schalten wir diesen Compilerschalter nur ein, wenn auch wir Debuggen wollen!)

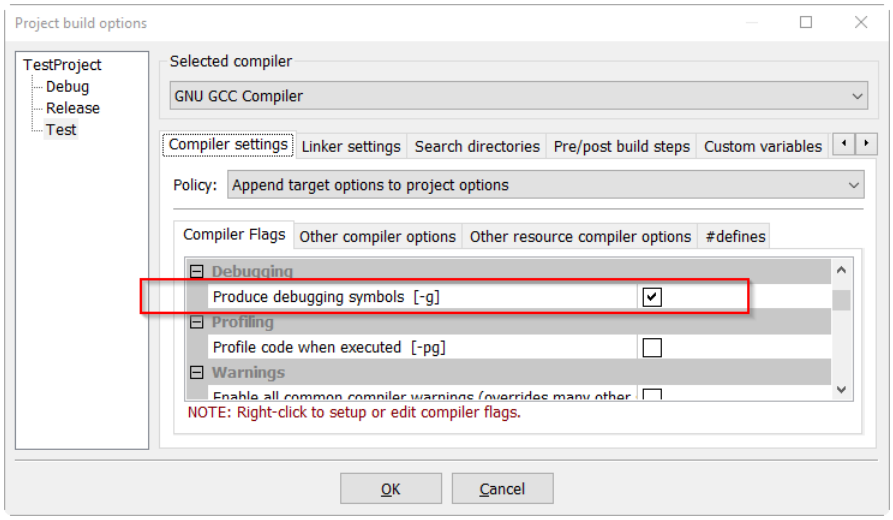

Abbildung 25: Compilerschalter Debugging Symbole erzeugen

Als letztes müssen wir dem Testtarget noch den Pfad zu den include-Files bekannt machen:

Im Tabreiter für das Test Target die Search directories einstellen:

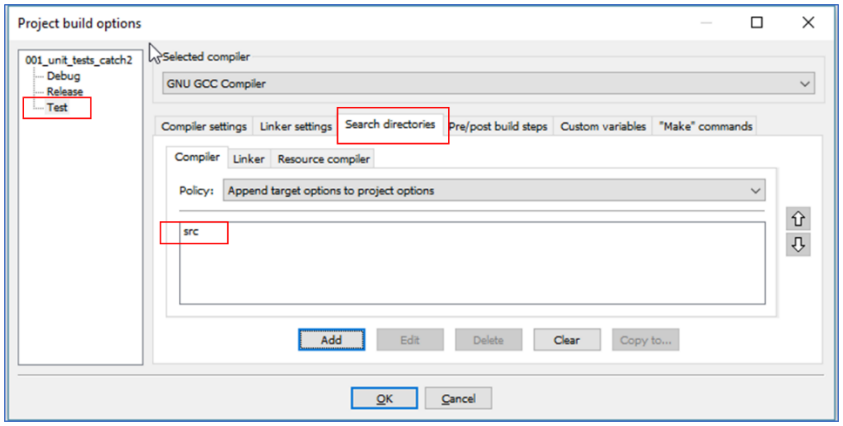

Abbildung 26: Suchpfad für Include-Files setzen

Jetzt das virtuelle Target All auswählen und übersetzen!

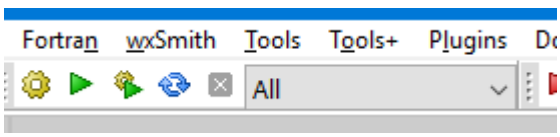

Abbildung 27: Target All auswählen

Schließlich können wir das Projekt starten!

Da wir noch keine Tests geschrieben haben, werden auch keine ausgeführt.

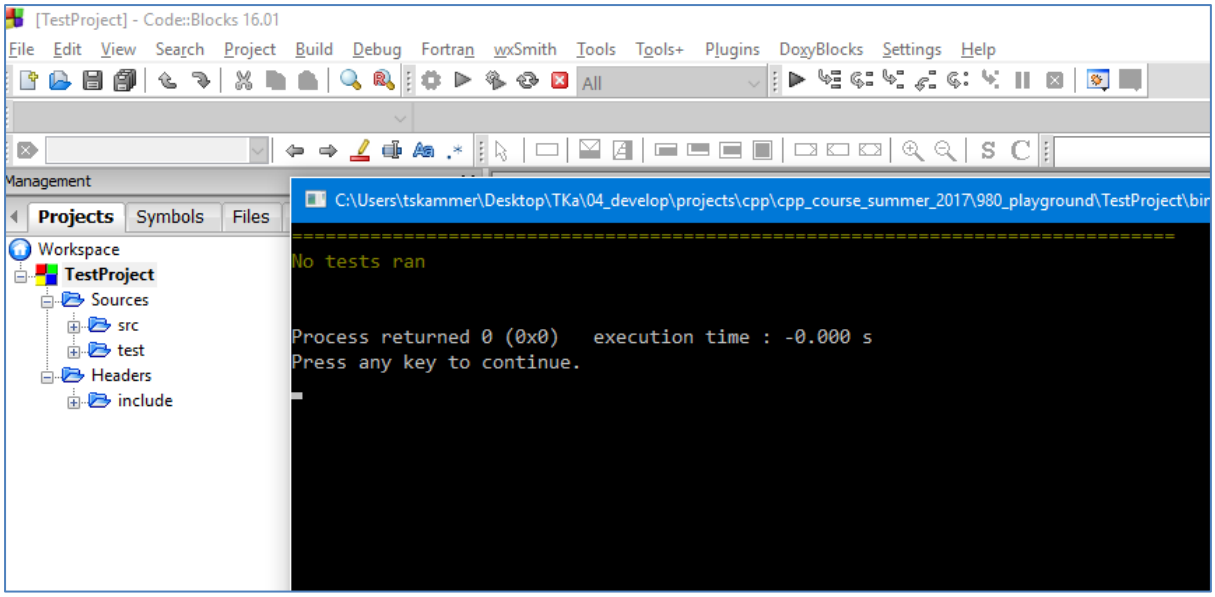

Abbildung 28: Unit Tests werden noch nicht ausgeführt.

Um einen Test ausführen zu können erweitern wir die Klasse Person:

Person.h

```
#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person {
  public:
    Person( const std::string& first_name, const std::string& sure_name );

    const std::string& get_first_name() const;
    const std::string& get_sure_name() const;

  private:
    std::string first_name_;
    std::string sure_name_;
};

#endif // PERSON_H
```

Person.cpp

```
#include "Person.h"

Person::Person( const std::string& first_name, const std::string& sure_name )
    : first_name_ ( first_name )
    , sure_name_ ( sure_name ) {
}
```

```
const std::string& Person::get_first_name() const {
    return first_name_;
}

const std::string& Person::get_sure_name() const {
    return sure_name_;
}
```

Jetzt legen wir ein neues File test_person.cpp unter /test an.

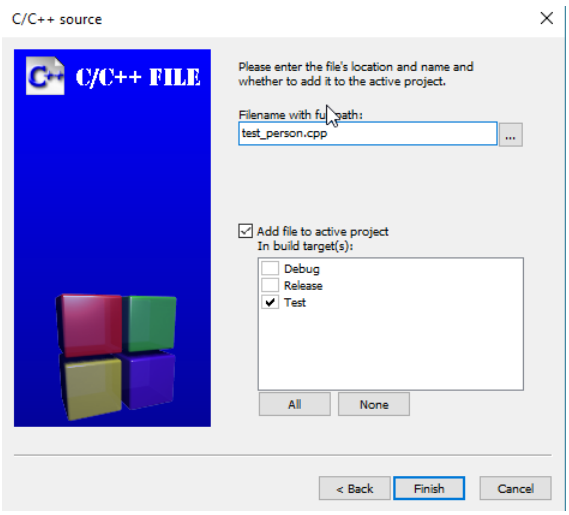

Abbildung 29: test_person.cpp anlegen

Test_person.cpp wird nur im Test-Target benötigt!

test_person.cpp:

```
#include "catch.hpp"
#include "Person.h"

TEST_CASE( "test construction", "[Person]" ) {
    Person person( "Vorname", "Nachname" );
    REQUIRE( person.get_first_name() == "Vorname" );
    REQUIRE( person.get_sure_name() == "Nachname" );
}
```

TEST_CASE(…):	legt einen Testfall an.
Person person(…) :	Hier legen wir eine Instanz der Klasse Person an
REQUIRE(…):	Hier überprüfen wir, ob Vorname und Nachname im Konstruktor richtig gesetzt wurden.

Übersetzen und ausführen!

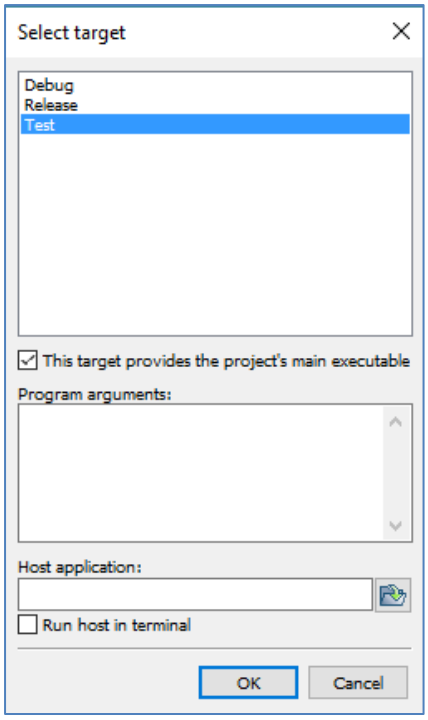

Abbildung 30: Test ist unser Ausführbares target!

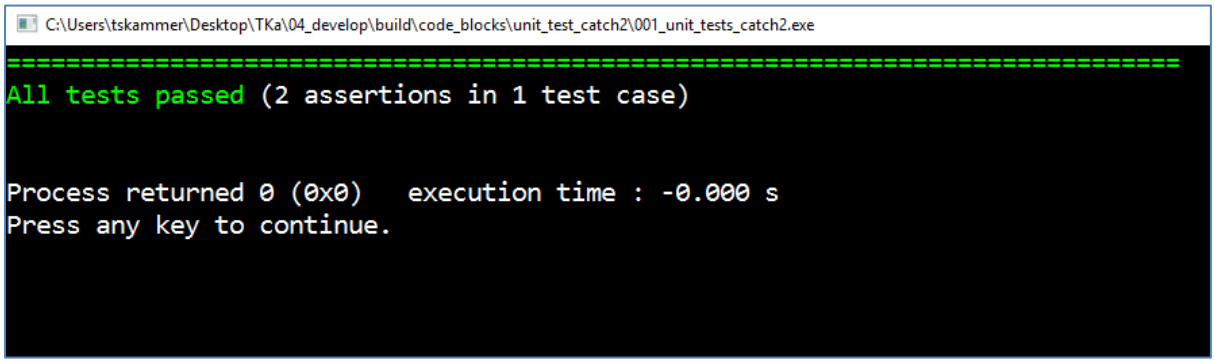

Abbildung 31: Ein test case, 2 Assertions!

AUFGABE BIS ZUM NÄCHSTEN MAL: SCHREIBEN SIE EINE NEUE KLASSE EMPLOYEE!

Employee soll einen Angestellten unserer Firma darstellen. Jeder Employee hat folgende Eigenschaften (Attribute):

- Vorname (last_name),
- Nachname (first_name),
- eindeutige Nummer (id),
- Gehalt (salary)

Jeder Employee hat einen Status, der Anzeigt, ob er angestellt ist oder nicht.

Employee soll folgende öffentlichen Methoden bekommen:

- `Einen Default-Konstruktor, der Vorname und Nachname setzt`
- `void promote( int raise_amount )`
- `void demote( int demerit_amount )`
- `void hire()`
- `void fire()`
- `void display() - outputs employee info to console`
- `getter und setter - Methoden für Name, Vorname, Nummer, Gehalt`
- `bool is_hired()`

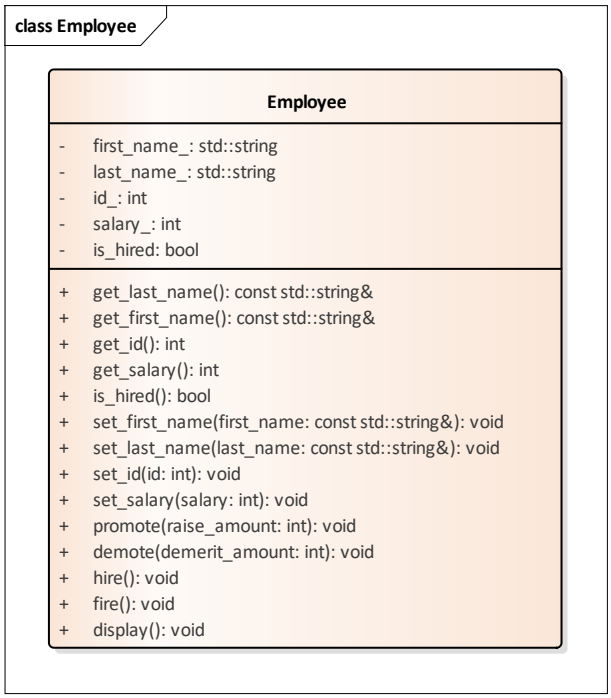

Abbildung 32: Employee in UML

Schreiben Sie die Klasse getrennt in Header- und cpp-File

Testen Sie die Klasse mit catch2!