Object Oriented programming and software engineering – Lab 21

Adam Korytowski – 2025

1. Code testing

What is Testing?

Software testing is the process of checking that a program behaves as expected.
This includes verifying that:

- Functions return correct results
- Classes behave correctly in different scenarios
- Errors are handled gracefully
- Changes don't break existing features (regression)

In C++, we usually write **unit tests** — small pieces of code that test individual components (like a class or method) in isolation.

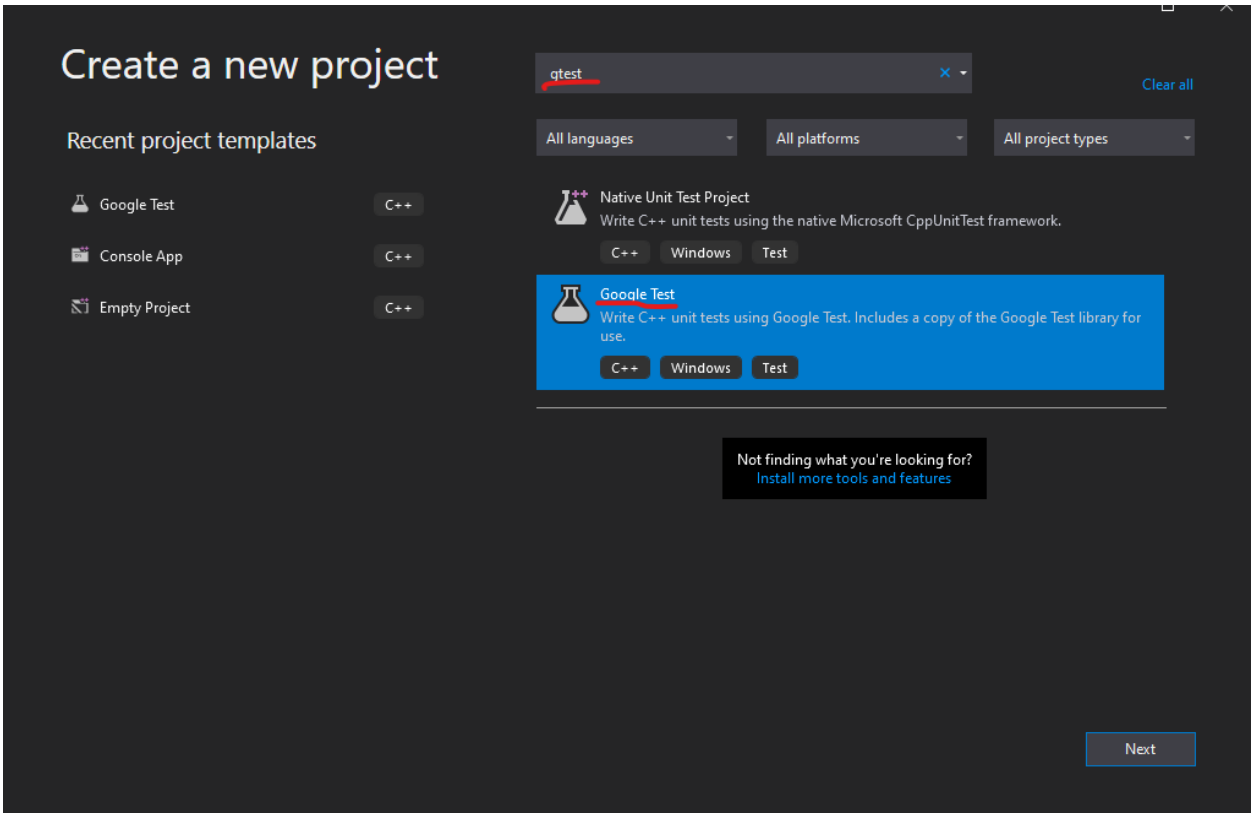
Why Do We Need Testing?

- Catch Bugs Early
  It's much easier to fix a bug when you discover it during development than after release. Tests help spot mistakes automatically.
- Verify Behavior
  Tests ensure your class does what it should — no more, no less. This is especially important in projects with inheritance or complex logic.
- Make Refactoring Safe
  Want to change some logic? If you already have tests, you'll know immediately if you broke something.
- Improve Design
  Writing tests often forces you to write cleaner, more modular code — because it's easier to test small, well-structured components.
- Confidence to Collaborate
  In team projects, tests give confidence that everyone's code works together — and future changes won't cause silent bugs.

2. Create google test project or integrate it to your existing project.

For Visual Studio:

Create new Google Test project:



If it's not found, then:

- Download Google Test

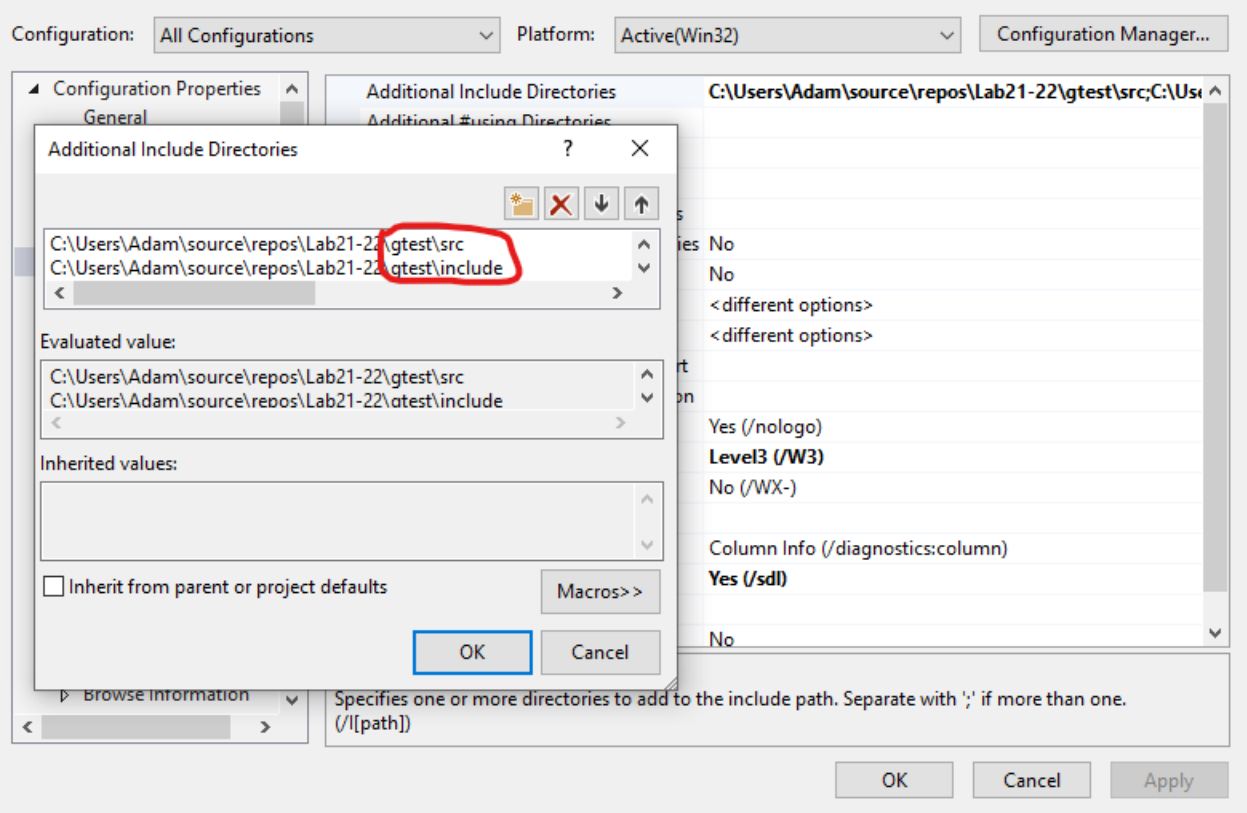
- Download https://github.com/google/googletest
- Inside it, you'll find the folders:
    - googletest/include/
    - googletest/src/
    - And the file: gtest-all.cc in googletest/src/

- add a directory called gtest in a folder with your project (should be next to the folder with a folder with your source files).

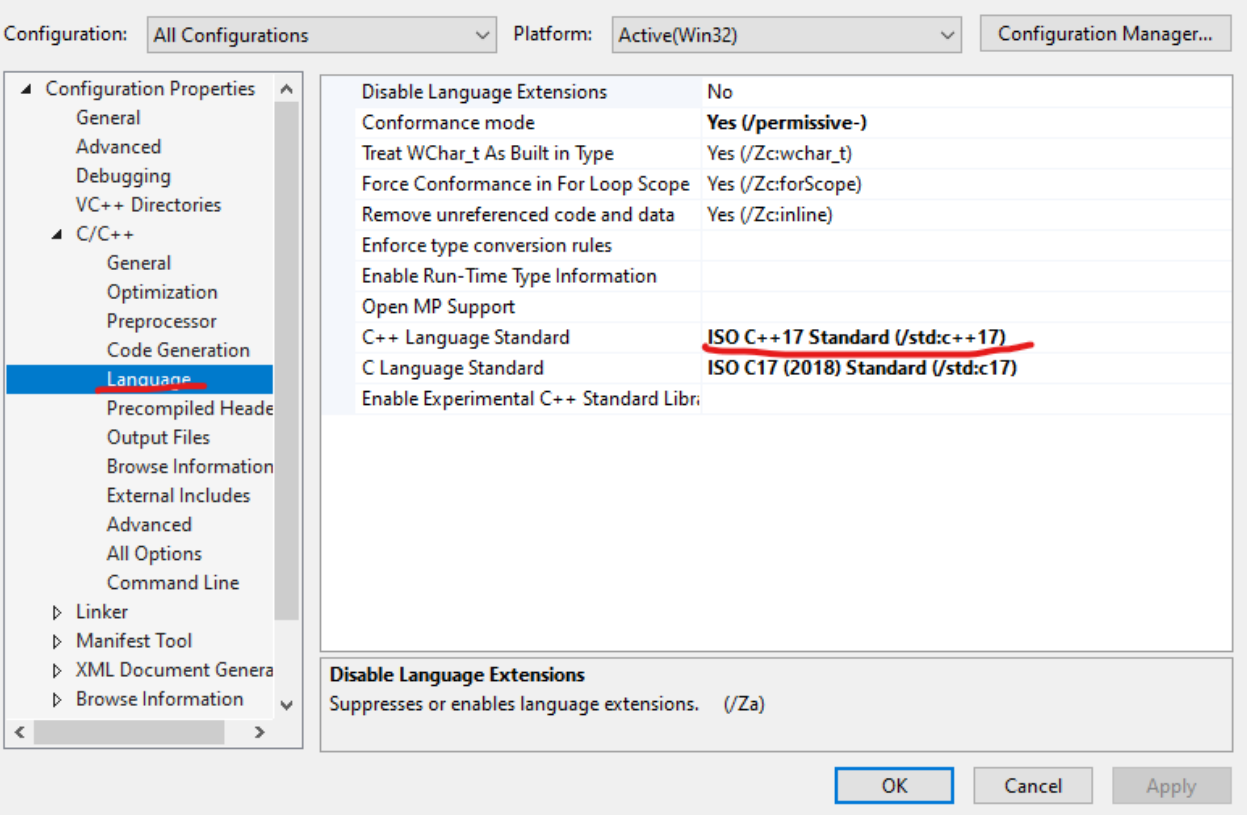
- grab src and include folders and put them inside of your gtest folder

- add the *include* and *src* folders as working directories in your project:

- right click your project -> properties -> C/C++ -> Additional Include Directories, then:



- move gtest-all.cc file outside of src folder

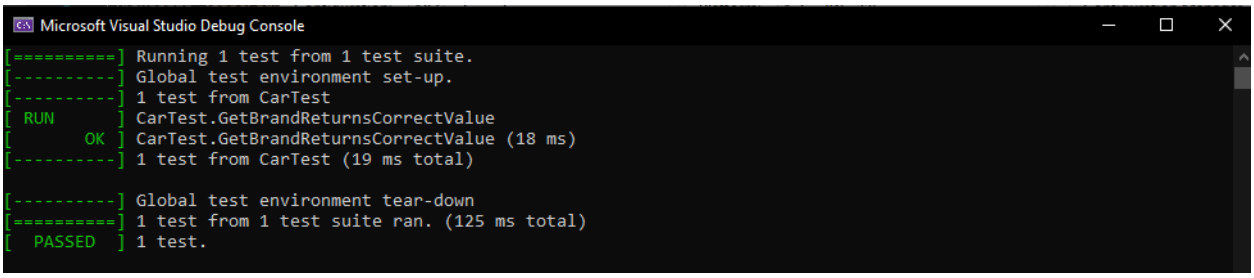


- create a test file, e. g.:

```
al-inl.h   CarTest.cpp   Lab21-22.cpp
#include <iostream>
#include "Lab21-22.cpp"
#include "../include/gtest/gtest.h"

TEST(CarTest, GetBrandReturnsCorrectValue)
{
    Car* h = new Honda();
    Car* o = new Opel();

    EXPECT_EQ(h->getBrand(), "Honda");
    EXPECT_EQ(o->getBrand(), "Opel");

    delete h;
    delete o;
}

int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- add gtest-all.cc to your project: right click folder with your source files -> add existing item -> choose the gtest-all.cc file.

- Compile the project. You should see output similar to this:



Dev C++ instruction:

- Go to the official repo: https://github.com/google/googletest
- Download the ZIP or clone the repo:

From the downloaded files, copy these folders into your Dev-C++ project directory:

  - googletest/include/ (contains headers)
  - googletest/src/ (contains source files like gtest-all.cc)

- Create a New Console Project

1. Open Dev-C++ → File → New → Project → Console Application → C++
2. Name the project (e.g., TestCar)
3. Save it to the same folder where you placed the Google Test files.

- Add Google Test to Your Project

1. In Dev-C++, go to Project → Project Options → Parameters
2. Under the Compiler tab, add:

-std=c++17

(Google Test requires at least C++17)

3. Under the Linker tab, add:

-lpthread

4. In the project tree, right-click on Sources → Add a new source file:
    - Name it e.g. gtest_main.cpp
    - Paste the following:

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

5. Add another source file for your actual test cases, e.g. CarTests.cpp and paste test code there.
6. Add gtest-all.cc to your project:
    - Right-click → Add to project → navigate to googletest/src/gtest-all.cc

---

- Include Paths Setup

Go to Tools → Compiler Options → Directories → C++ Includes

Add the relative path to the googletest/include directory, for example:

.\googletest\include

Also, if you get errors about internal includes, you may also need to add:

.\googletest

(So both the include and root folder are available to resolve internal headers)

- Build & Run
- Click Compile & Run (F11)
- If everything is configured properly, your tests will run in the console output.

3. Tasks:

Use your existing code with Base and Derived classes (2 pts each):

- Write a test that checks whether an object of a subclass is correctly initialized. Verify that the constructor assigns the expected values to the relevant attributes.

```
class Car
{
public:
    std::string brand;
    int year;

    Car(const std::string& brand, int year) : brand(brand), year(year) {}
    virtual ~Car() = default;
};

class Honda : public Car
{
public:
    Honda(int year) : Car("Honda", year) {}
};
```

```
TEST(ConstructorTest, HondaInitialization)
{
    Car* car = new Honda(2020);

    EXPECT_EQ(car->brand, "Honda");
    EXPECT_EQ(car->year, 2020);

    delete car;
}
```

- Write a test that verifies whether a virtual method behaves correctly when overridden in derived classes. Call the method via a base-class pointer to ensure polymorphism works properly. Check that each subclass returns the expected result.

Base class:

```
virtual std::string honk() const
{
    return "Generic honk";
}
```

Derived classes:

```
std::string honk() const override
{
    return "Honda horn!";
}
```

Test file:

```
TEST(PolymorphismTest, HondaHonk)
{
    Car* car = new Honda(2025);
    EXPECT_EQ(car->honk(), "Honda horn!");
    delete car;
}

TEST(PolymorphismTest, OpelHonk)
{
    Car* car = new Opel(2025);
    EXPECT_EQ(car->honk(), "Opel horn!");
    delete car;
}
```

- Write a test that ensures constructors handle invalid input safely. If incorrect values are passed (like a negative year), the constructor should throw an exception or prevent object creation. Use assertions to verify that invalid objects cannot be created silently.

Base:

```
Car(int year)
{
    if (year < 1886)
    {
        throw std::invalid_argument("Invalid year");
    }
    this->year = year;
}
```

Test file:

```
TEST(ValidationTest, RejectsInvalidYear)
{
    EXPECT_THROW({
        Car * car = new Car(1700);
        delete car;
        }, std::invalid_argument);
}
```