C++ Software Engineering

for engineers of other disciplines

Module 7

"Software Quality Assurance"

1st Lecture: Test

ALTEN

Autumn 2021

Gothenburg, Sweden

petter.lerenius@alten.se

rashid.zamani@alten.se

Code Quality

- Satisfying the indented requirements is indeed the least requirement for "*high-quality*" code.
- *High-quality* code is:
  - *efficient,*
  - *simple,*
  - *readable,*
  - and *robust.*
- Architectural and design decision would also affect the quality of the code.
- Code *Quality* could be checked at either source or binary.

"SQA encompasses the entire software development process, including requirements engineering, software design, coding, code reviews, source code control, software configuration management, testing, release management and software integration. It is organized into goals, commitments, abilities, activities, measurements, verification and validation." https://en.wikipedia.org/wiki/Software_quality_assurance

- Software is not only code, and many factors can influence its quality. For instance, any factors affecting programmers, like even the company culture, could affect quality of their work, and that is why SQA covers a very wide spectrum of everything which could have influence directly, or indirectly to the quality of the software.

Testing

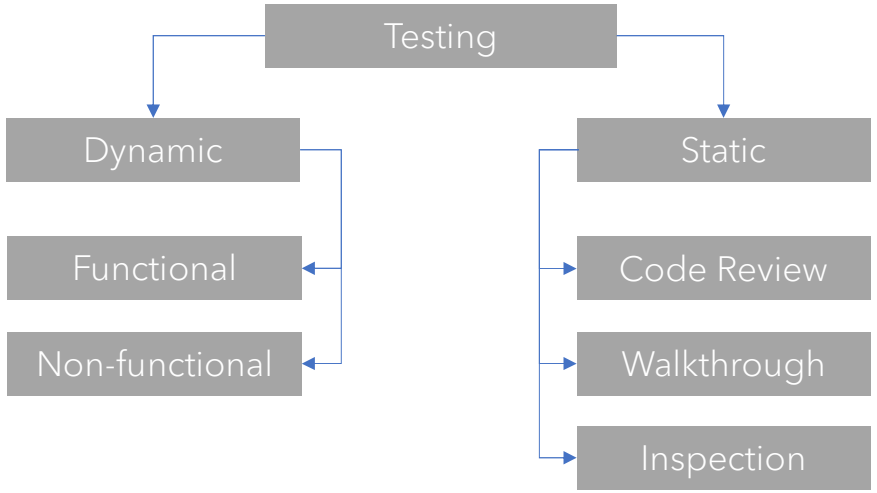
ALTEN

- Software testing intention is to either:
  - Crash the software i.e. find a bug
  - Verify its *functionality*
- Write code to test code -- software testing could include *running* parts or even the whole software
- There is infinite amount of tests that could target each project, only a certain *test sets*, sometime only on specific components makes it to the *test plan*.

"On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed. http://www-users.math.umn.edu/~arnold/disasters/ariane.html

Dynamic vs Static

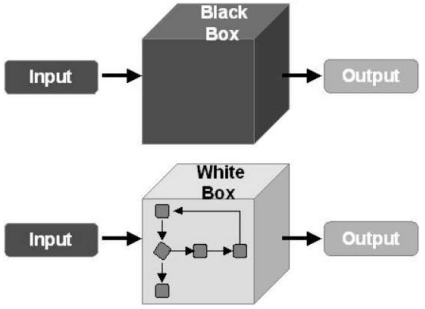
- Dynamic testing requires execution of the software.
- Static testing occurs on the source code.
- Static testing can only ***verify*** while dynamic testing can both ***verify*** & ***validate***:
  - Functional *verifies* whether the right product is made.
  - Non-functional *validates* if the product is the right one.



- Inspection covers a broad range of activities – code reviews and peer reviews are both inspections.
- Walkthroughs and inspections could target any artefacts not just source code.
- Verification is to figure whether the *correct* product is implemented, whereas validation's concerns is if the product is implemented *correctly*.

White vs Black

- *White-box* tests internal structure. And *Black-box* tests functionality, based on specifications and requirements
- Knowledge of source code is needed to perform white-box test.
- Both *can* happen at different *levels* – white-box is the preferred approach for *Unit testing* while *System* and *Acceptance test* are usually back-box.
- White-box testing can be measured in *code coverage.* Higher coverage *suggests* lower chances of bugs -- static testing is also white-box.
- Black-box is un-biased, and *usually* a functional test. There are tools to measure code coverage.
- Grey-box testing uses knowledge of internal structures to designate certain *paths* – may involve *reverse engineering*.

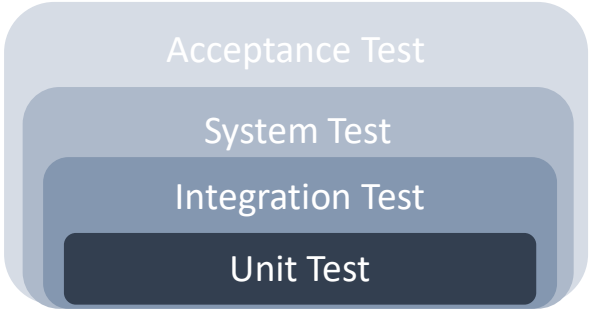


http://www.softwaretestinggenius.com/photos/wbtut1.JPG

- In black-box testing *knowledge* of the source code is not considered, as opposed to white-box testing. Imagine testing a division function -- a black-box testing approach would then be to test division over zero, since it's the riskiest, or *randomly* choose an input. While a white-box test could deal with the knowledge of the algorithm and types being used and tries to verify and validate those, hence inputs are *chosen.*
- *Code coverage* is a software metric and is reported based on *function, statement,* and *decision coverage.* 100% coverage ensures all paths been tested at least once, ensuring functionality to some degree, yet not sufficient.

Testing Level

- Testing, conceptually, can occur on four levels:
  - ***Unit Testing***: testing the smallest working *units,*
  - ***Integration Testing***: testing the *interfaces* between *components,*
  - ***System Testing***: testing the whole *integrated* system, and
  - ***Acceptance Test***: testing the system for the stakeholders



"Broadly speaking, there are at least three levels of testing: unit testing, integration testing, and system testing. However, a fourth level, acceptance testing, may be included by developers. This may be in the form of operational acceptance testing or be simple end-user (beta) testing, testing to ensure the software meets functional expectations. Based on the ISTQB Certified Test Foundation Level syllabus, test levels includes those four levels, and the fourth level is named acceptance testing. https://en.wikipedia.org/wiki/Software_testing#Testing_levels

- *Acceptance test* is usually a test performed to certify the system is satisfying the requirements -- this is a very common test which occurs upon delivery of a system.

Unit Test

- *Unit* is the smallest *testable* part of a software, which could be tested in ***isolation***.
- Unit testing can start early in the project.
- In C++, units are methods – but method could be in `class`es.
- There are unit testing frameworks for different programming languages, some measure code coverage.
- Unit testing is most of the time an automated process.

> *"Finite state machine coverage […] works on the behavior of the design. In this coverage method, you need to look for how many time-specific states are visited, transited. It also checks how many sequences are included in a finite state machine."* https://www.guru99.com/code-coverage.html

```
class Foo {
  public:
    Foo(){}
    void showMe() {
      this->decrease();
      std::cout << this->calc() << std::endl;
    }
  protected:
    void decrease() {this->bar--;}
  private:
    int calc() { return 10/this->bar;}
    int bar = 2020;
};
```

- *Mock objects* are *fake* objects which a unit requires to run, but their creation is out of that unit's scope. For instance, variables or object a method takes as input. *Mocks* are created with sole purpose of testing the unit, not testing the object!

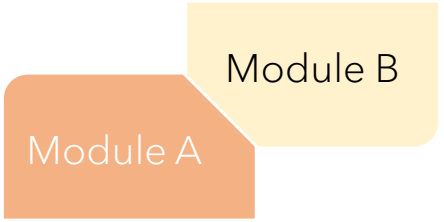
Unit Test – Pros & Cons

Advantages	Disadvantages
Finds bug in early development stages which is *cheaper*	Will not catch all errors (e.g. system level)
Helps structuring the code better	There are problems which are hard to be tested in nature (e.g. concurrent or non-deterministic)
Could help tracing the faults	Unit test involves writing code which could be buggy itself!
Provides documentation for a unit, exposing the critical characteristic of the unit	Writing useful unit test is very hard

Integration Test

ALTEN

- *Integrating* a group of testing them together.
- Integration testing evaluate each modules compliance with the *functional* requirements i.e. testing their interface.
- There are four approaches:
  - Big-Bang
  - Top-down
  - Bottom-up
  - Sandwich



Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing. https://en.wikipedia.org/wiki/Integration_testing

- *Stubs* and *drivers* are *fake* module simulating a component yet-to-be-implemented. *Test Drivers* are usually more complex than *Stubs*.

Integration Test -- Approaches

Approach	Advantages	Disadvantages
Big-bang	Saves time and costs, feasible for small projects.	All modules should be implemented, and there is no priority between modules.
Bottom-up	No stubs is required, and can happen in parallel for different sub-modules	Most of sub-modules should be ready. Drivers might be needed, and it could be complex for systems with many submodules
Top-down	No need for drivers, confirmation on high level interfaces (global) early in the project.	Needs many stubs, with inadequate focus on lower level components.
Sandwich	Useful for large projects, overcoming hurdles of both Top-down and Bottom-Up	Could be complex for systems with high interdependency and expensive

System Testing vs Acceptance Testing

ALTEN

System Test	Acceptance Test
Test against *specified* requirements	Test against *user* requirements
Performed by developers, users not involved.	Performed by customer, manager, or set of test targets, users are involved.
Happens after implementation finishes prior to delivery (acceptance test)	Happens after the system test, and during delivery
The intention is to find bugs (negative)	The intention is to test *functionality* (positive)
Dummy input values	Actual *real-time* input values
Covers both functional & non-functional	Covers only functional
Is a combination of integration testing & system testing	Is a combination of *alfa* & *beta* testing

Test Case

- Specification of inputs, execution condition, steps required and expected outcome of a single test.
- *Positive* test cases are those testing the behavior on normal inputs.
- *Negative* test cases are those testing the behavior against invalid inputs.
- Collection of test cases is called a *test suite.*

"Information that may be included [in a testcase are]:

•Test Case ID - This field uniquely identifies a test case.
•Test case Description/Summary - This field describes the test case objective.
•Test steps - In this field, the exact steps are mentioned for performing the test case.
•Pre-requisites - This field specifies the conditions or steps that must be followed before the test steps executions.
•Depth
•Test category
•Author- Name of the Tester.
•Automation - Whether this test case is automated or not.
•pass/fail
•Remarks

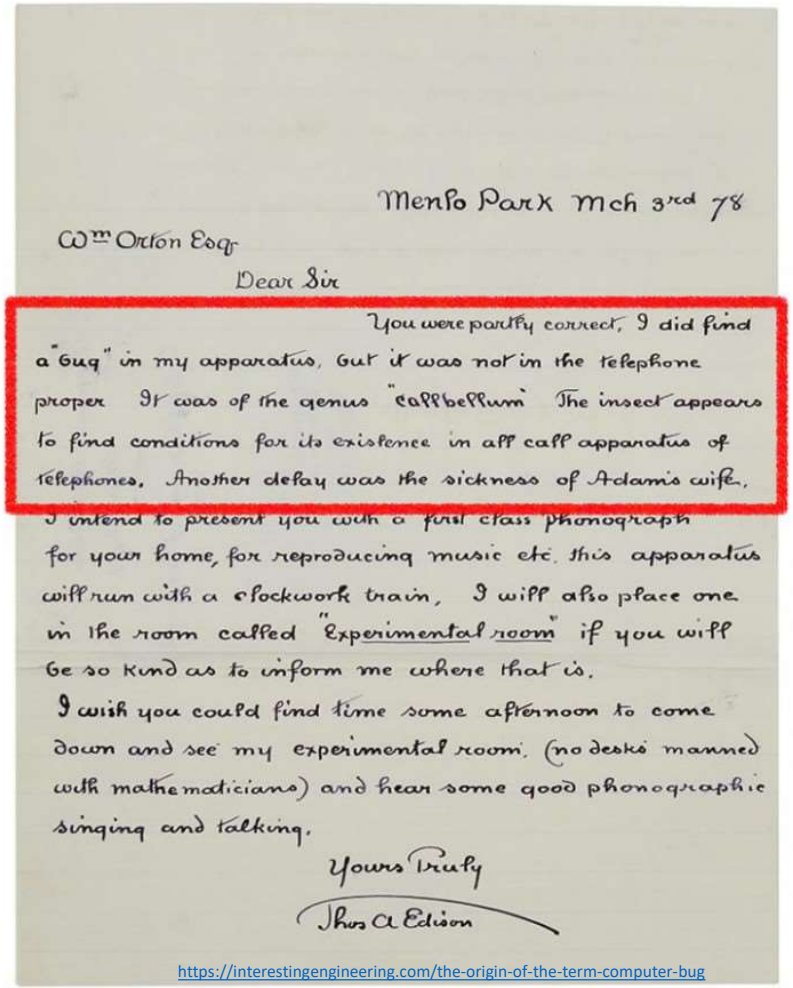
A written test case should also contain a place for the actual result." https://en.wikipedia.org/wiki/Test_case#Typical_written_test_case_format

Types of Test

- **Smoke Test:** preliminary test addressing basic questions – does the application even run? Runs quicker and provides faster feedback. Determines whether it is *sane* to continue with *more extensive* test suits.
- **Regression Test:** Re-running previously passed test suites to ensure a modification has not broken anything. Modifications include any changes from configuration and hardware to bug fixes and performance tunings.
- **Alpha Test:** simulated or actual operational testing of the whole system by a *selected* group of users prior to going *Beta.*
- **Beta Test:** Targets wider range of user and could be viewed as *external Acceptance Test.* Feedback is collected from the users.
- **Software Performance Test:** testing the performance of the system under different circumstances including *load* and *stress* testing. For real-time systems, there are test to measure the delay of the system.
- **Destructive Test:** tries to tear down the software and crashes software – guarantees robustness as software is test against invalid input.

Debugging

- Bugs are faults in the program which causes it to behave unexpectedly.
- Debugging is the procedure of resolving the bugs.
- *Debugger* is a tool used to *execute* program *step by step,* providing developer with a *real-time* inside look into software.
- Compiler can insert *special header* in the machine code for debuggers.
- Bugs which are not reproducible are very bad!
- **gdb** from GNU project is the main debugger used for C and C++.



Menlo Park Mch 3rd 78

Wm Orton Esqr

Dear Sir

You were partly correct, I did find a "bug" in my apparatus, but it was not in the telephone proper It was of the genus "cappberrum" The insect appears to find conditions for its existence in all capp apparatus of telephones. Another delay was the sickness of Adams wife. I intend to present you with a first class phonograph for your home, for reproducing music etc. this apparatus will run with a clockwork train, I will also place one in the room called "Experimental room" if you will be so kind as to inform me where that is.

I wish you could find time some afternoon to come down and see my experimental room. (no desks manned with mathematicians) and hear some good phonographic singing and talking.

Yours Truly

Thos A Edison

https://interestingengineering.com/the-origin-of-the-term-computer-bug

assert

- Checks whether a condition holds or not (equals to 0).
- If the condition doesn't hold, information regarding where in the source code the *assertion* has failed is printed out and the program execution is *aborted.*
- Usually would go to *release builds* as the extra check hampers the performance.
- Gives developers the better possibility of bug tracing.

Defined in header <cassert>

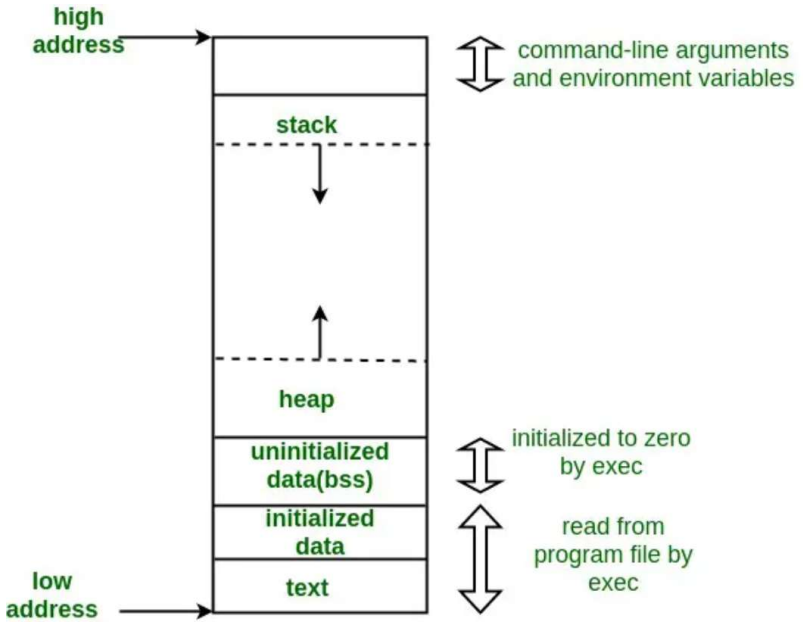
```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /*implementation defined*/
#endif
```

https://en.cppreference.com/w/cpp/error/assert

```
#include <cassert>
class Foo {
  public:
    Foo(){}
    void showMe() {
      this->decrease();
      std::cout << this->calc() << std::endl;
    }
  protected:
    void decrease() {this->bar--;}
  private:
    int calc() {
        assert(this->bar);
        return 10/this->bar;
      }
    int bar = 2020;
};
```

C++ Worst Bugs?

- C++ code can make memory faulty, even crashing the whole system.
- There are specific tools which look into memory allocation and de-allocation through dynamic and static testing.
- Users with malicious intentions will try to exploit memory bugs.
- Certain security techniques exist to ensure memory safety:
  - Stack canaries
  - Non-executable memory space
  - Address space layout randomization



https://microcontrollerslab.com/difference-between-stack-and-heap/

DEMO!

gdb

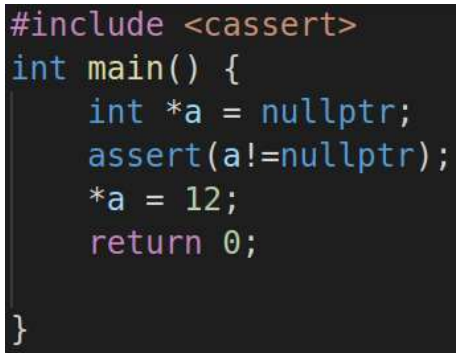
ALTEN

gdb.cpp

```
   7
   8          int *fun1(const size_t &_s) {
   9              return new int[_s];
   10         }
   11
  >12         void fun2(int *_d, const size_t &_s) {
   13             for (int i = _s - 1; i >= 0; i--) {
   14                 *(_d+i) = i+_s;
   15             }
   16         }
   17
B+ 18         int main() {
   19             fun2(fun1(fun0()),fun0());
   20         }
```

```
native process 34615 In: fun2
0x00005555555552f4 in main () at gdb.cpp:19
fun2 (_d=0x555555555265 <fun1(unsigned long const&)+64>, _s=@0x555555555390: 10181608798458613747) at gdb.cpp:12
(gdb) print d[1]
cannot subscript requested type
(gdb) print _d[1]
$5 = 1213594142
(gdb) p _d[1]
$6 = 1213594142
```

Assert



```cpp
#include <cassert>
int main() {
    int *a = nullptr;
    assert(a!=nullptr);
    *a = 12;
    return 0;

}
```

```
mrz@vbubu:~/cc/CXX_Course_Demo/Day9/1_assert$ g++ -DNDEBUG -O3 art.cpp
mrz@vbubu:~/cc/CXX_Course_Demo/Day9/1_assert$ ./a.out
Segmentation fault (core dumped)
mrz@vbubu:~/cc/CXX_Course_Demo/Day9/1_assert$ g++ art.cpp
mrz@vbubu:~/cc/CXX_Course_Demo/Day9/1_assert$ ./a.out
a.out: art.cpp:7: int main(): Assertion `a!=nullptr' failed.
Aborted (core dumped)
```

Test


```cpp
int main () {
    test0();
    test1();
    test2();
    test3();
    return 0;
}
```

```cpp
void findAnElement(int *p, size_t s, size_t e) {
    bool b = false;
    for (size_t i = 0; i < s; i++)
        if (*(p+i) == e) {
            std::cout << "FOUND" << std::endl;
            b = true;
            break;
        }
    if(b);else std::cout << "NOT FOUND" << std::endl;
}
```

```cpp
void test0() {
    int a[45] = {0};
    findAnElement(a,45,0);
}

void test1() {
    int a[45]={0};
    findAnElement(a,45,1);
}

void test2() {
    int *a;
    findAnElement(a,45,0);
}

void test3() {
    int *a = new int [3];
    findAnElement(a,-2,11);
}
```