LECTURE 6

Coding practices

When writing code, one aims to guarantee that

- the implemented features behave as specified (in some specification),
- incorrect behaviour is detectable, its source is identifiable,
- bugs introduced in future changes to the code base will get detected automatically.

Practices:

- Using assertions to check pre-conditions, post-conditions, and other conditions.
- Using exceptions to detect and handle unexpected behaviour.
- Implement, run and regularly re-run tests.

Assertions

A function may:

- act on its arguments (if there is any),
- act on some environment (member function on its class's state, external objects, read/write files, etc.),
- return a value.

Specification determines:

- **Precondition**: condition to hold true right before the function is executed (conditions on input arguments, on the environment, etc.)
- **Postcondition**: condition to hold true right after the function execution is completed (conditions on the post-state of the arguments, the environment, returned value).

Design by contract:

- Preconditions are to be guaranteed by user.
- Postconditions are to be guaranteed by the implementation.

Assertions - an example

`Lecture10/Examples.hpp` defines alternative names to the `assert` function in order to emphasise the purpose (these are just messages to the compiler):

```
1 #include <cassert> // required for assert
2 #define PRECONDITION assert
3 #define POSTCONDITION assert
4 #define ASSERT assert
```

`Lecture10/Examples.cpp` checks if a vector is large enough when the second entry is about to be accessed:

```
1 double thirdOfSecondElement(const dVector & vec){
2   PRECONDITION(vec.size()>=2); // assertion checked in debug mode
3   return vec[1]/3.0;
4 }
```

When the assertions fails, the code execution in debug mode stops with a message that specifies which assertions failed.

```
Assertion failed: (vec.size()>=2),
function thirdOfSecondElement,
file Examples.cpp, line 8.
```

Note: In release mode the assertions are ignored, hence assertions cannot be overused.

Exception

What if error happens?

- Design by contract: if preconditions does not hold, it is the user's fault.
- The other runtime errors are not all the developer's fault. Examples?

The concept of exceptions in C++ is about separating **error detection** and **error handling**.

- The implementation is responsible for detecting the error, and informing the user about it.
- The user has the option to decide how to handle the error: let it fail, or continue execution on an alternative route.

References:

- Stroustrup - *The c++ programming language*

Exception - throwing one

Example in `Lecture10/Examples.cpp`:

```
1  double thirdOfSecondElementWithBoundCheck(const dVector & vec){
2    if(vec.size()<2)
3      throw string("Input vector has less than two entries; at least two are
           required.");
4
5    return vec[1]/3.0;
6  }
```

- In this particular case, if the input argument is invalid, the function throws a `std::string`
- Users can use built in types (e.g. `invalid_argument`) or define new exception classes for particular errors. For more advanced exception types, see the references on the previous slide.
- Some argue, that this particular failure is not exceptional, and no exception should be thrown.
- Some argue, that a clear error message is essential to deal with system failures, and throwing an exception is the right thing to do here.
- Midway: assume correct usage within the interface boundary (non-exported features).
- Note: conditions are checked in both release and debug modes, hence overusing exceptions may have an impact on the performance.

Exception - catching one

Using the function that may throw an exception:

```
1  // try block, attempted execution
2  try{
3    cout << "Result of thirdOfSecondElementWithBoundCheck(vec2): "
4     << thirdOfSecondElementWithBoundCheck(vec2) << endl;
5  }
6  // catch block handles potential errors
7  catch(const string msg){
8    cout << msg << endl;
9  }
```

- The call of the function is wrapped into a `try` block.
- If the function fails, all local variables in the `try` block are destroyed. The code will not crash if the thrown exception is caught by a `catch` block.
- The `catch` blocks catch exceptions by type.
- If more than one type of exception can be thrown from the `try` block, a `catch` block for each type is to be implemented.
- The handling action is typically specific to the type of the exception.
- `catch(...)` catches everything. But it may hide errors that should be fixed.
- If an exception is not caught, the execution will crash. The failure is identifiable.

Unit tests

Unit tests:

- Unit testing frameworks: Boost, Google Test, CppUnit, etc.
- Purpose of the tests: run functions with various values of input arguments and check if the output and/or the behaviour is expected.
- The unit tests are derived from the function specification.
- Unit tests typically cannot cover possible input values (or all possible states of the environment). **Values to test**: values around the boundary of the domain, values inside the domain.
- **No guarantees**: even if all tests pass, defects might be present.
- Failure indicates a defect in the implementation of the function and/or in the implementation of the failing test.

When to run the tests?

- **Immediately on a dummy implementation**. Of course tests should fail.
- **Repeatedly during the implementation**, until all tests pass.
- **Whenever the code-base (or any settings, platform, etc.) is modified**, in order to make sure the changes do not break the existing functions.

Unit tests - example 1

```
1  // adds two unsigned ints without using +,-,* or /
2  unsigned int add(unsigned int a , unsigned int b){
3     // 0 ^ 0 = 0, 1 ^ 0 = 1
4     // 0 ^ 1 = 1, 1 ^ 1 = 0
5    unsigned int c = a ^ b;
6    unsigned int d = (a & b) << 1; // carry when 1 + 1
7    if (d == 0)
8      return c;
9    else
10     return add(c,d);
11 }
12
13 unsigned int numberOfOnes(unsigned int a){
14   unsigned int res = a & 1;
15   while(a != 0){
16     a >>= 1;
17     res += a & 1;
18   }
19   return res;
20 }
```

- bit operations: and &, or |, xor ^,
- but manipulation: bit shift <<, <<=, >>, >>=

Unit tests - example 1

```
1  #include "gtest/gtest.h"
2  #include "bitManipulationExamples.hpp"
3
4  TEST(AdditionTest, ZeroZero){
5    EXPECT_EQ(0,add(0,0)); // expected vs returned
6  }
7
8  TEST(AdditionTest, ZeroOther){
9    for(unsigned int i(0); i<1000; i++)
10     EXPECT_EQ(i,add(0,i));
11 }
12
13 TEST(AdditionTest, OtherOther){
14   for(unsigned int i(1); i<200; i++)
15     for(unsigned int j(1); j<200; j++)
16       EXPECT_EQ(i+j, add(i,j));
17 }
```

Google Unit Testing Framework
https://code.google.com/p/googletest/wiki/Primer

- The first argument arguments of TEST is a name of a set of tests, the second argument is the name of the particular test.
- Tests may contain one or more assertions. EXPECT_EQ has two arguments; the second is typically a value returned by the function being tested, the first one is the value that is expected by the call.

Unit tests - example 1

```
1  TEST(NumberOfOnes, Zero){
2    EXPECT_EQ(0,numberOfOnes(0));
3  }
4
5  TEST(NumberOfOnes, One){
6    EXPECT_EQ(1,numberOfOnes(1));
7  }
8
9  TEST(NumberOfOnes, PowersOfTwo) {
10   for(unsigned int i(1); i<31; i++)
11     EXPECT_EQ(1, numberOfOnes(1<<i));
12 }
13
14 TEST(NumberOfOnes, PowersOfTwoMinusOne) {
15   for(unsigned int i(1); i<31; i++)
16     EXPECT_EQ(i, numberOfOnes((1<<i)-1));
17 }
18
19 TEST(NumberOfOnes, General) {
20   EXPECT_EQ(2, numberOfOnes(3));
21   EXPECT_EQ(2, numberOfOnes(5));
22   EXPECT_EQ(2, numberOfOnes(6));
23   EXPECT_EQ(2, numberOfOnes(12));
24 }
```

Unit tests - example 2

Testing `thirdOfSecondElement...`

```
#include "gtest/gtest.h"
#include "Examples.hpp"
#include <string>

TEST(thirdOfSecondElement, badInput){
  dVector vec(1,0.0);
  EXPECT_DEATH(thirdOfSecondElement(vec),"Assertion failed");
}

TEST(thirdOfSecondElement, validInput1){
  dVector vec(2,1.0);
  EXPECT_DOUBLE_EQ(1.0/3.0, thirdOfSecondElement(vec));
}

TEST(thirdOfSecondElement, validInput2){
  dVector vec(200,1.0);
  EXPECT_DOUBLE_EQ(1.0/3.0, thirdOfSecondElement(vec));
}
```

- `EXPECT_DEATH` checks if the execution crashes.
- `EXPECT_DOUBLE_EQ` checks if the `double`'s are *almost* equal accounting for discrepancies due to round off errors.

Unit tests - example 2

Testing `thirdOfSecondElement...`

```
TEST(thirdOfSecondElementWithBoundCheck, badInput){
  dVector vec(1,0.0);
  EXPECT_THROW(thirdOfSecondElementWithBoundCheck(vec),std::string);
}

TEST(thirdOfSecondElementWithBoundCheck, validInput1){
  dVector vec(2,1.0);
  EXPECT_NEAR(1.0/3.0, thirdOfSecondElementWithBoundCheck(vec),1.0e-15);
}

TEST(thirdOfSecondElementWithBoundCheck, validInput2){
  dVector vec(200,1.0);
  EXPECT_NEAR(1.0/3.0, thirdOfSecondElementWithBoundCheck(vec),1.0e-15);
}
```

- `EXPECT_THROW` checks if exception of a certain type is thrown.
- `EXPECT_NEAR` checks if the two values are within error bounds; more flexible than `EXPECT_DOUBLE_EQ`. When would you use it?

Unit tests

Running unit tests requires the following main:

```
#include "BitManipTests.hpp"
#include "ExampleTests.hpp"
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

- Running `main` without arguments invokes all the tests.
- Subsets can be invoked via command line arguments:

```
--gtest_filter=NumberOfOnes*
```

Useful feature when applying the test-driven-development approach on a relatively small part of a bigger project.

- Note: the `gtest` is a static library, required to be built only once.
- When implementing new features and running the unit test on them, only the new project is required to be built.

Unit tests - running them

Command line argument: `--gtest_filter=NumberOfOnes*`:

```
Note: Google Test filter = NumberOfOnes*
[==========] Running 5 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 5 tests from NumberOfOnes
[ RUN      ] NumberOfOnes.Zero
[       OK ] NumberOfOnes.Zero (0 ms)
[ RUN      ] NumberOfOnes.One
[       OK ] NumberOfOnes.One (0 ms)
[ RUN      ] NumberOfOnes.PowersOfTwo
[       OK ] NumberOfOnes.PowersOfTwo (0 ms)
[ RUN      ] NumberOfOnes.PowersOfTwoMinusOne
[       OK ] NumberOfOnes.PowersOfTwoMinusOne (0 ms)
[ RUN      ] NumberOfOnes.General
[       OK ] NumberOfOnes.General (0 ms)
[----------] 5 tests from NumberOfOnes (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 5 tests.
```

Unit tests - further features and references

- Assertions:
  `https://code.google.com/p/googletest/wiki/Primer#Assertions`
- More assertions:
  `https://code.google.com/p/googletest/wiki/AdvancedGuide#More_Assertions`
- Global setup and teardown (setting up testing objects and share them among different tests):
  `https://code.google.com/p/googletest/wiki/AdvancedGuide#Global_Set-Up_and_Tear-Down`
- Further features and references:
  `https://code.google.com/p/googletest/wiki/AdvancedGuide`

Summary

design-by-contract	`assert`
precondition	postcondition
exception	`throw`, `try`, `catch`
test-driven development	`gtest`
`EXPECT_EQ`	`EXPECT_NEAR`
`EXPECT_DOUBLE_EQ`	`EXPECT_THROW`