```
/////////////////////////////////////////////////////
Unit Testing II
______________________________________________________
maXbox Starter 79 -Unit Testing Routines with Asserts - Max Kleiner

"Love comes unseen; we only see it go."
    - Henry Austin Dobson


Use the Assert procedure to document and enforce the assumptions you must make when writing code. Assert is not a real procedure. The compiler handles Assert specially and compiles the filename and line number of the assertion to help you locate the problem should the assertion fail.

Assert is not a real procedure. The compiler handles Assert specially and compiles the filename and line number of the assertion to help you locate the problem should the assertion fail.
The syntax is like:

procedure Assert(Test: Boolean);
procedure Assert(Test: Boolean; const Message: string);

If you write a simple script program and distribute it to each computer, you can have the users start the tests on their own by running the script with a list of asserts.

  Assert(CopyFrom('a', 0) = 'a', 'CopyFrom');
  Assert(CopyFrom('a', -1) = 'a', 'CopyFrom');
  Assert(CopyFrom('', 1) = '', 'CopyFrom');
  Assert(CopyFrom('', -2) = '', 'CopyFrom');
  Assert(CopyFrom('1234567890', 8) = '890', 'CopyFrom');
  Assert(CopyFrom('1234567890', 11) = '', 'CopyFrom');
  Assert(CopyFrom('1234567890', 0) = '1234567890', 'CopyFrom');
  Assert(CopyFrom('1234567890', -2) = '1234567890', 'CopyFrom');

  Assert(not StrMatch('', '', 1), 'StrMatch');
  Assert(not StrMatch('', 'a', 1), 'StrMatch');
  Assert(not StrMatch('a', '', 1), 'StrMatch');
  Assert(not StrMatch('a', 'A', 1), 'StrMatch');
  Assert(StrMatch('A', 'A', 1), 'StrMatch');
  Assert(not StrMatch('abcdef', 'xx', 1), 'StrMatch');
  Assert(StrMatch('xbcdef', 'x', 1), 'StrMatch');
  Assert(StrMatch('abcdxxxxx', 'xxxxx', 5), 'StrMatch');
  Assert(StrMatch('abcdef', 'abcdef', 1), 'StrMatch');
  Assert(StrMatch('abcde', 'abcd', 1), 'StrMatch');
  Assert(StrMatch('abcde', 'abc', 1), 'StrMatch');
  Assert(StrMatch('abcde', 'ab', 1), 'StrMatch');
  Assert(StrMatch('abcde', 'a', 1), 'StrMatch');
  Assert(StrMatches('abcd', 'abcd', 1)=true, 'StrMatches');

Lets take the above single assert with

  Function StrMatches(const Substr, S: AnsiString; const Index: Int): Boolean;

As you can see the strings matches if equal otherwise we get an Exception:

  Assert(StrMatches('abcd', 'abcde', 1)=true, 'StrMatches');

  >>> Exception: StrMatches

If the test condition fails the SysUtils unit sets this variable to a procedure that raises the EAssertionFailed exception.
By the way dont comment an assert like that

  //Assert(StrMatchLeft('ABC1D', 'aBc1', False), 'StrMatchLeft');
  //Assert(StrMatchLeft('aBC1D', 'aBc1', True), 'StrMatchLeft');

You can also negate an assert as long as they deliver a boolean (logic) condition:

  Assert(not StrMatchLeft('AB1D', 'ABc1', False), 'StrMatchLeft');
  Assert(not StrMatchLeft('aBC1D', 'aBc1', True), 'StrMatchLeft');

Then you want to write more assert system information to a log file for analysing problems during installation, debuging, tests and deinstallation or app distribution like that:

10/01/2018 19:31:54 V:4.6.2.10 [max] MAXBOX8 A problem occurred in initializing MCI. [at:  3275216pgf; mem:1247492]
14/01/2018 17:15:18 V:4.7.2.30 [max] MAXBOX8 Out Of Range. [at:  2607048pgf; mem:1082444]
14/01/2018 17:15:21 V:4.7.2.40 [max] MAXBOX8 Out Of Range. [at:  2605716pgf; mem:1080012]
16/01/2018 09:18:00 V:4.7.5.20 [max] MAXBOX8 List index out of bounds (456). [at:  2913700pgf; mem:1157700]

{                                                                              }
{ Test cases                                                                   }
{                                                                              }
{$IFDEF DEBUG}
{$IFDEF LOG}
{$IFDEF TEST}
//{$ASSERTIONS ON}

Next step is to bundle asserts in a Test Procedure with sections like that:

procedure TestBitsflc;
```

```
begin
  Assert(SetBit32($100F, 5) = $102F,            'SetBit');
  Assert(ClearBit32($102F, 5) = $100F,          'ClearBit');
  Assert(ToggleBit32($102F, 5) = $100F,         'ToggleBit');
  Assert(ToggleBit32($100F, 5) = $102F,         'ToggleBit');
  Assert(IsBitSet32($102F, 5),                  'IsBitSet');
  Assert(not IsBitSet32($100F, 5),              'IsBitSet');
  Assert(IsHighBitSet32($80000000),             'IsHighBitSet');
  Assert(not IsHighBitSet32($00000001),         'IsHighBitSet');
  Assert(not IsHighBitSet32($7FFFFFFF),         'IsHighBitSet');

  Assert(SetBitScanForward32(0) = -1,           'SetBitScanForward');
  Assert(SetBitScanForward32($1020) = 5,        'SetBitScanForward');
  Assert(SetBitScanReverse32($1020) = 12,       'SetBitScanForward');
  Assert(SetBitScanForward321($1020, 6) = 12,    'SetBitScanForward');
  Assert(SetBitScanReverse321($1020, 11) = 5,    'SetBitScanForward');
  Assert(ClearBitScanForward32($FFFFFFFF) = -1, 'ClearBitScanForward');
  Assert(ClearBitScanForward32($1020) = 0,      'ClearBitScanForward');
  Assert(ClearBitScanReverse32($1020) = 31,     'ClearBitScanForward');
  Assert(ClearBitScanForward321($1020, 5) = 6,   'ClearBitScanForward');
  Assert(ClearBitScanReverse321($1020, 12) = 11, 'ClearBitScanForward');

  Assert(ReverseBits32($12345678) = $1E6A2C48,  'ReverseBits');
  Assert(ReverseBits32($1) = $80000000,         'ReverseBits');
  Assert(ReverseBits32($80000000) = $1,         'ReverseBits');
  Assert(SwapEndian32($12345678) = $78563412,   'SwapEndian');

  Assert(RotateLeftBits32(0, 1) = 0,          'RotateLeftBits32');
  Assert(RotateLeftBits32(1, 0) = 1,          'RotateLeftBits32');
  Assert(RotateLeftBits32(1, 1) = 2,          'RotateLeftBits32');
  Assert(RotateLeftBits32($80000000, 1) = 1,  'RotateLeftBits32');
  Assert(RotateLeftBits32($80000001, 1) = 3,  'RotateLeftBits32');
  Assert(RotateLeftBits32(1, 2) = 4,          'RotateLeftBits32');
  Assert(RotateLeftBits32(1, 31) = $80000000, 'RotateLeftBits32');
  Assert(RotateLeftBits32(5, 2) = 20,         'RotateLeftBits32');
  Assert(RotateRightBits32(0, 1) = 0,         'RotateRightBits32');
  Assert(RotateRightBits32(1, 0) = 1,         'RotateRightBits32');
  Assert(RotateRightBits32(1, 1) = $80000000, 'RotateRightBits32');
  Assert(RotateRightBits32(2, 1) = 1,         'RotateRightBits32');
  Assert(RotateRightBits32(4, 2) = 1,         'RotateRightBits32');

  Assert(LowBitMask32(10) = $3FF,               'LowBitMask');
  Assert(HighBitMask32(28) = $F0000000,         'HighBitMask');
  Assert(RangeBitMask32(2, 6) = $7C,            'RangeBitMask');

  Assert(SetBitRange32($101, 2, 6) = $17D,      'SetBitRange');
  Assert(ClearBitRange32($17D, 2, 6) = $101,    'ClearBitRange');
  Assert(ToggleBitRange32($17D, 2, 6) = $101,   'ToggleBitRange');
  Assert(IsBitRangeSet32($17D, 2, 6),           'IsBitRangeSet');
  Assert(not IsBitRangeSet32($101, 2, 6),       'IsBitRangeSet');
  Assert(not IsBitRangeClear32($17D, 2, 6),     'IsBitRangeClear');
  Assert(IsBitRangeClear32($101, 2, 6),         'IsBitRangeClear');
  Assert(IsBitRangeClear32($101, 2, 7),         'IsBitRangeClear');
end;
{$ENDIF}
{$ENDIF}
```

A tester is then able to run a bunch of tests:

```
  setBitmaskTable;
  TestBitsflc;
```

```
15 CLF_Fundamentals Testroutines 47520
------------------------------------
   1   TestMathClass;
   2   TestStatisticClass;
   3   TestBitClass;
   4   TestCharset;
   5   TestTimerClass
   6   TestRationalClass
   7   TestComplexClass
   8   TestMatrixClass;
   9   TestStringBuilderClass
   10  TestASCII;
   11  TestASCIIRoutines;
   12  TestPatternmatcher;
   13  TestUnicodeChar;
   14  unit uPSI_AfUtils;
   15  unit uPSI_PsAPI;
```

Another way is to prevent call errors as a mistaken precondition of false assumption in
a procedure you designed. This pre- and postcondition can handle a lot of errors.
An example should make this clear.
A TStack object has a method called Pop to remove the topmost data object from the stack.

If the stack is empty, I count calling Pop as a programming mistake: you

really should check for the stack being empty in your program prior to calling Pop. Of course Pop could have an if statement within it

that did this check for you, but in the *majority* of cases the stack

wont be empty when Pop is called and in the *majority* of cases when

you use Pop, youll have some kind of loop in your program which is

continually checking whether the stack is empty or not anyway. In my

mind having a check for an empty stack within Pop is safe but slow.

So, instead, Pop has a call to an Assert procedure at the start

(activated by the DEBUG compiler define) that checks to see whether

the stack is empty. Here is the code for Pop:

```
      function TStack.Pop : pointer;
        var
          Node : PNode;
        begin
          {$IFDEF DEBUG}

          Assert(not IsEmpty, ascEmptyPop);

          {$ENDIF}

          Node := Head^.Link;

          Head^.Link := Node^.Link;

          Pop := Node^.Data;

          acDisposeNode(Node);
        end;
```

As you see, if DEBUG is set the Assert procedure checks whether the

stack is empty first, if not it executes the code that pops the data

object off the stack. If the stack is empty an EEZAssertionError

exception is raised (the constant ascEmptyPop is a string code for a

string-table resource). If DEBUG is not set the code runs at full speed.

So log the steps and compare test procedures before installation: The location of the update can be a local, UNC or network path to compare it.
When you need Admin Rights you can try this:

```
   ExecuteShell('cmd','/c runas "/user:Administrator" '+
                                             ExePath+'maXbox4.exe')
   or C:> net user Administrator /active:yes
```

After you have finishing and writing the script, the next and final step is select "Go Compile" in maXbox. What this does is create a complete, ready-to-run Setup program based on your script. By default, this is created in a directory named Exepath under the directory or UNC path containing the script or what destination you need.

```
function GetInstallScript(const S_API, pData: string): string;
var ts: TStrings;
begin
   with TIdHTTP.create(self) do begin
      try
        ts:= TStringList.Create
        ts.Add('install='+HTTPEncode(pData));
        result:= Post(S_API,ts);
      finally
        ts.Free;
        Free;
      end;
   end
end;
```

The big step comes with unit tests with setup and teardown. Generic "Assert This" Assertion Procedure means that most generic assertion program simply says "assert this" and passes a Boolean expression. It is used by all the other assertion routines, which constructa Boolean expression from their specific values and logic.
Unit testing is a way of testing the smallest piece of code referred to as a unit that can be logically isolated in a system. It is mainly focused on the functional correctness of standalone modules.

A unit can be almost anything you want it to be - a specific piece of functionality, a program, or a particular method within the application:

```
type
  THugeCardinal_TestCase =  TTestCase;
  var
    Fbig1234: THugeCardinal;
    Fbig2313: THugeCardinal;
    Fbig3547: THugeCardinal;
    //TVerifyResult
    Temp1, Temp2, Temp3, Temp4: THugeCardinal;
    Temp2000_1: THugeCardinal;
    Temp2000_2: THugeCardinal;
    T3, F100: THugeCardinal;
    TmpStream: TMemoryStream;

    procedure THugeCardinal_TestCaseSetUp; //override;
    procedure THugeCardinal_TestCaseTearDown; //override;

  //published
       //procedure Test_CreateZero;
       procedure Test_CreateRandom;
       procedure Test_CreateSmall;
       procedure Test_Clone;
       procedure Test_Assign;
       procedure Test_Zeroise;
       procedure Test_CompareSmall;
       procedure Test_Compare;
       procedure Test_AssignSmall;
       procedure Test_BitLength;
       procedure Test_MaxBits;
       procedure Test_Add;
       procedure Test_Increment;
       procedure Test_Subtract;
       procedure Test_MulPower2;
       procedure Test_MulSmall;
       procedure Test_Multiply;
       procedure Test_Modulo;
       procedure Test_AddMod;
       procedure Test_MultiplyMod;
       procedure Test_isOdd;
       procedure Test_CreateFromStreamIn;
       procedure Test_CloneSized;
       procedure Test_Resize;
       procedure Test_AssignFromStreamIn;
       procedure Test_Swap;
       procedure Test_ExtactSmall;
       procedure Test_StreamOut;
       procedure Test_PowerMod;
       procedure Test_SmallExponent_PowerMod;

   procedure InitUnit_HugeCardinalTestCases;
   begin
   //TestFramework.RegisterTest( THugeCardinal_TestCase.Suite)
      THugeCardinal_TestCaseSetUp;
   end;

   procedure DoneUnit_HugeCardinalTestCases;
   begin
     THugeCardinal_TestCaseTearDown
   end;
```

Conclusion:
The proper way to use Assert is to specify conditions that must be true in order for your code to work correctly.

```
  Assert(StrMatches('abcd', 'abcde', 1)=true, 'StrMatches');
```

All programmers make assumptions about internal state of an object or function, the value or validity of a subroutine's arguments, or the value returned from a function. A good way to think about assertions is that they check for programmer errors, not user errors!

7 Steps for maintainable code:
• Maintain separation of concerns (avoid unnecessary dependencies)
• Fully qualified unit names to be used: Winapi.Windows not Windows
• Code format to be consistent with LIB source
• Do not put application-specific implementations in general code libraries
• Carefully consider modification to common code – the way to proceed
• No hints (instant code review fail) and No warnings
• Keep code small – avoid long methods and should be broken down

Ref:
http://www.softwareschule.ch/download/maxbox_starter36.pdf
https://www.oreilly.com/library/view/delphi-in-a/1565926595/re18.html
http://www.softwareschule.ch/examples/unittests.txt
script: 919_uLockBox_HugeCardinalTestCases.pas

Doc:
https://maxbox4.wordpress.com
>>> https://basta.net/speaker/max-kleiner/
>>> https://entwickler-konferenz.de/speaker/max-kleiner/