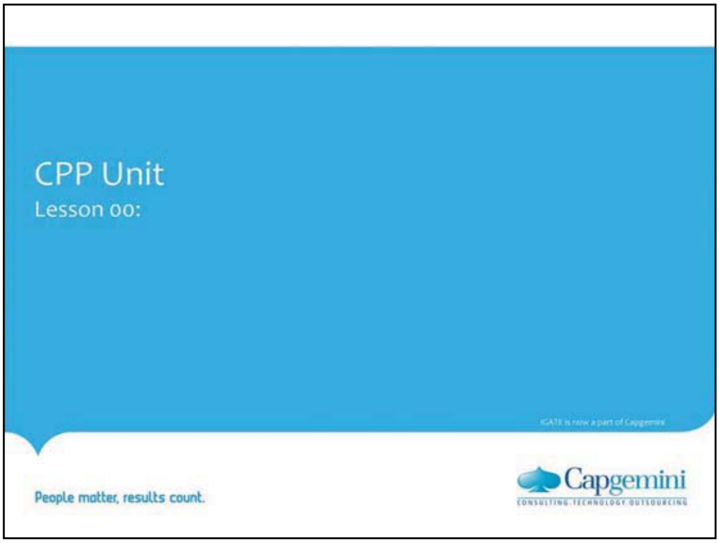


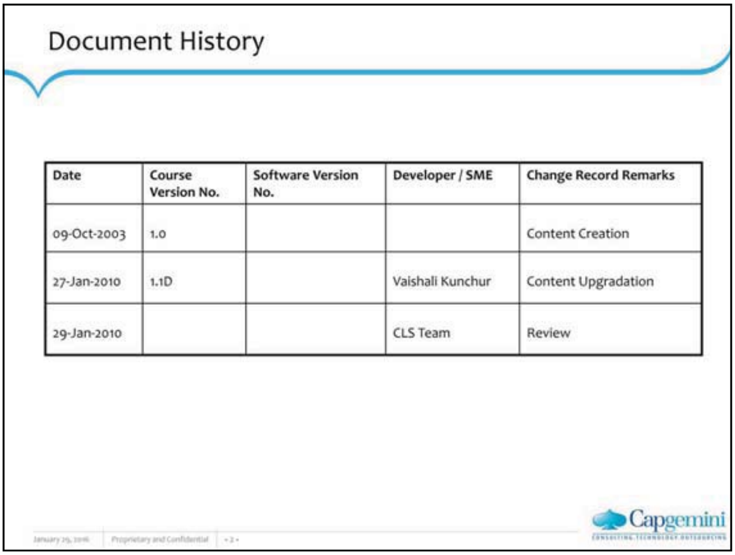
CPP Unit

Lesson 00:

IGATE is now a part of Capgemini

People matter, results count.

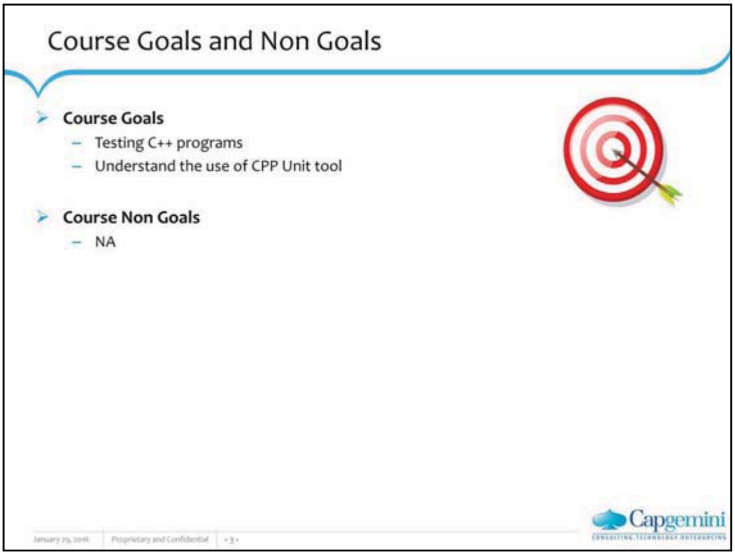
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
09-Oct-2003	1.0			Content Creation
27-Jan-2010	1.1D		Vaishali Kunchur	Content Upgradation
29-Jan-2010			CLS Team	Review

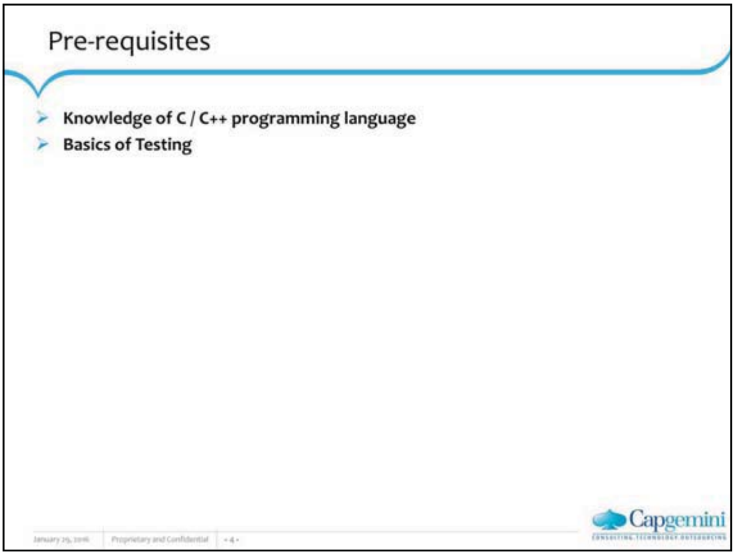
January 29, 2016 | Proprietary and Confidential | - 2 -



Course Goals and Non Goals

- **Course Goals**
  - Testing C++ programs
  - Understand the use of CPP Unit tool
- **Course Non Goals**
  - NA

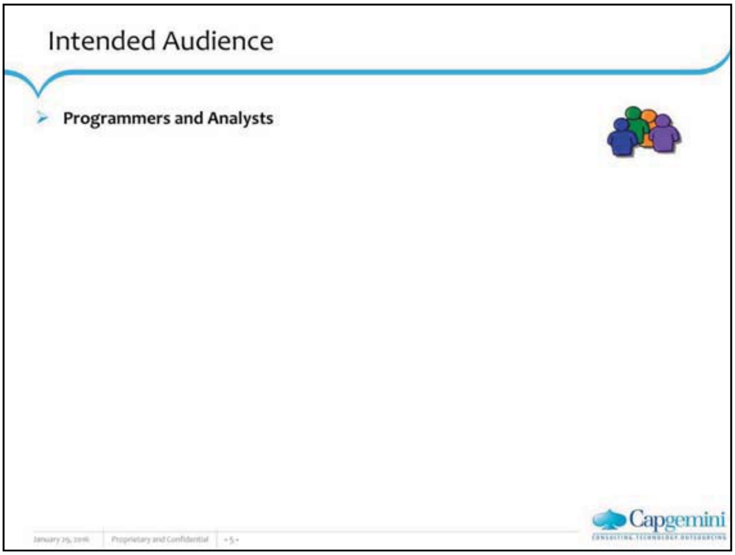
January 29, 2016 | Proprietary and Confidential | - 3 -



Pre-requisites

- Knowledge of C / C++ programming language
- Basics of Testing

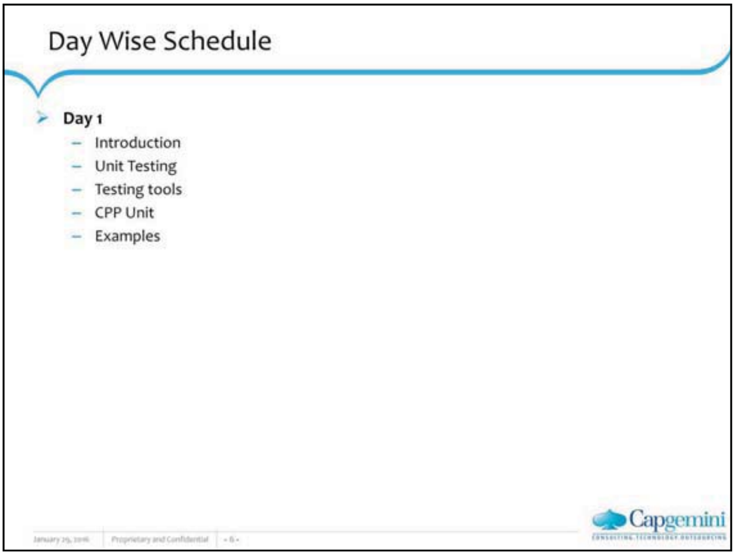
January 29, 2016 | Proprietary and Confidential | - 4 -



Intended Audience

- **Programmers and Analysts**

January 29, 2016 | Proprietary and Confidential | - 5 -



Day Wise Schedule

- **Day 1**
  - Introduction
  - Unit Testing
  - Testing tools
  - CPP Unit
  - Examples

January 29, 2016 | Proprietary and Confidential | - 6 -

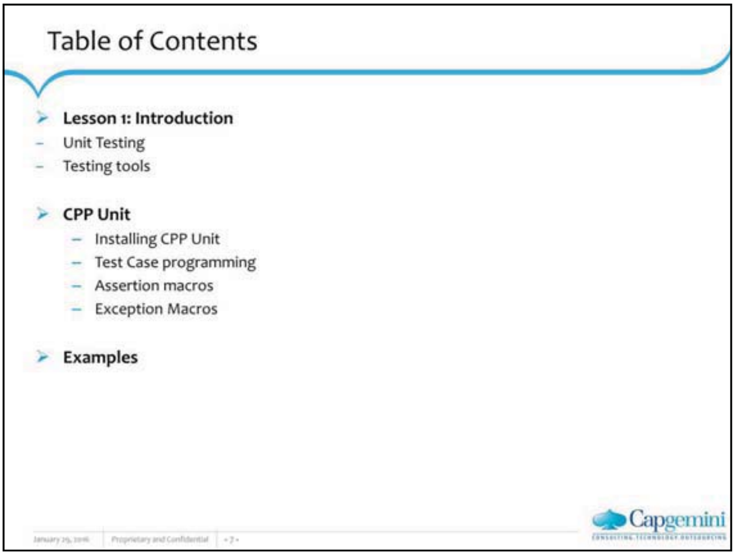


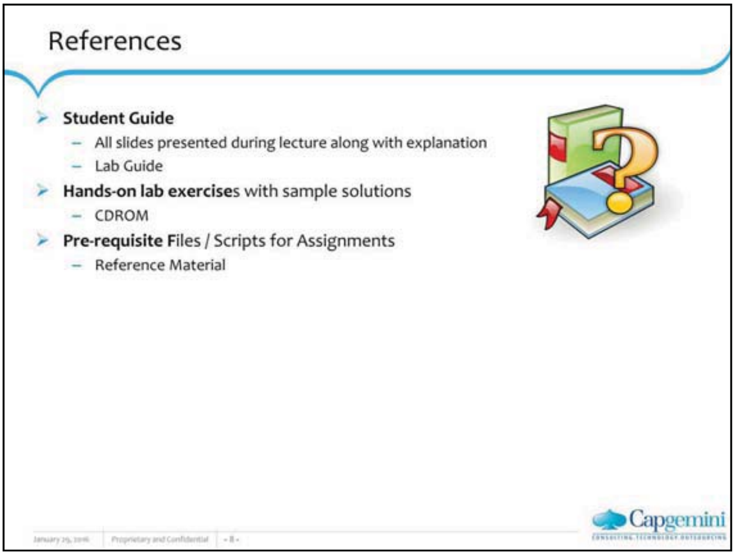
Table of Contents

- **Lesson 1: Introduction**
- Unit Testing
- Testing tools

- **CPP Unit**
  - Installing CPP Unit
  - Test Case programming
  - Assertion macros
  - Exception Macros

- **Examples**

January 29, 2016 | Proprietary and Confidential | - 7 -

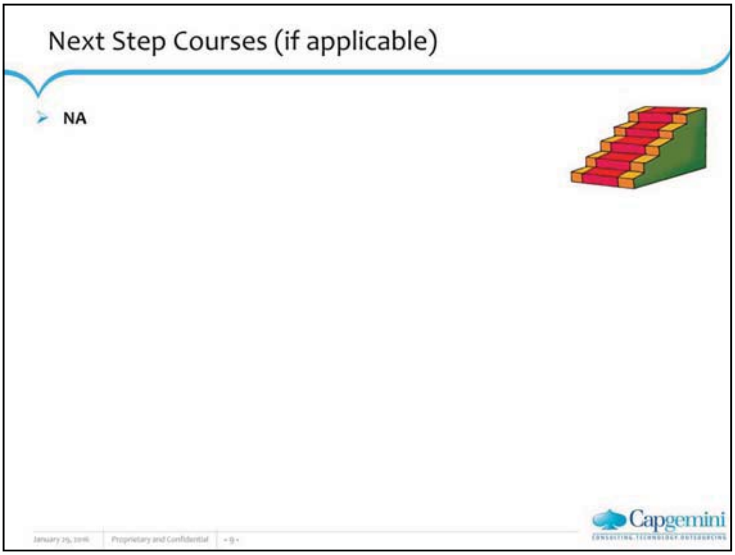


References

- **Student Guide**
  - All slides presented during lecture along with explanation
  - Lab Guide
- **Hands-on lab exercise**s with sample solutions
  - CDROM
- **Pre-requisite F**iles / Scripts for Assignments
  - Reference Material

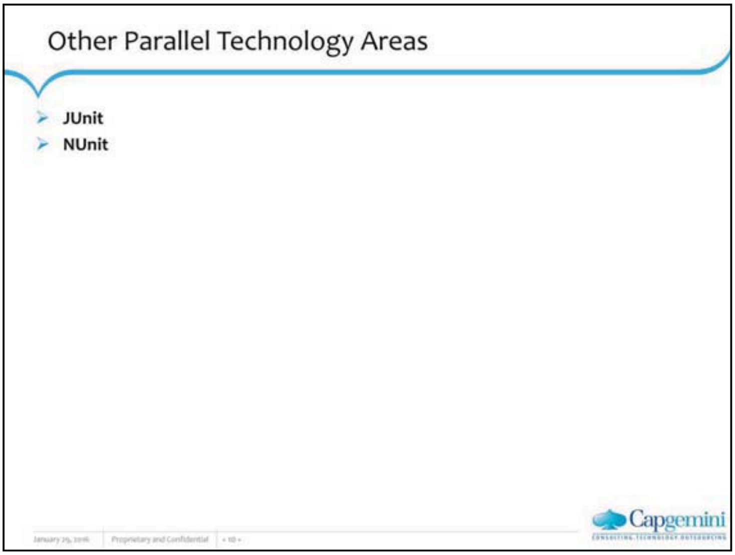
January 29, 2016 | Proprietary and Confidential | - 8 -

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



Next Step Courses (if applicable)

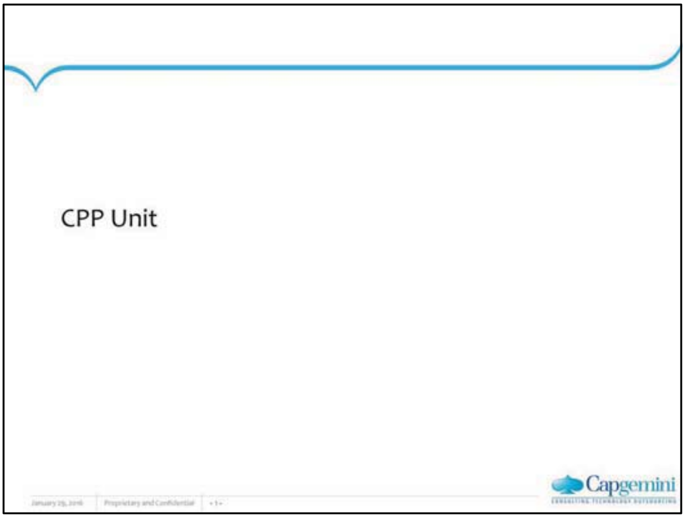
- **NA**



Other Parallel Technology Areas

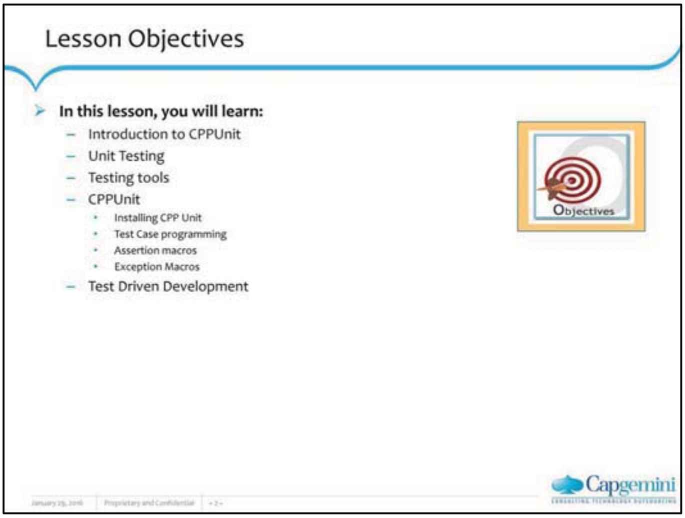
- **JUnit**
- **NUnit**

January 29, 2016 | Proprietary and Confidential | - 10 -



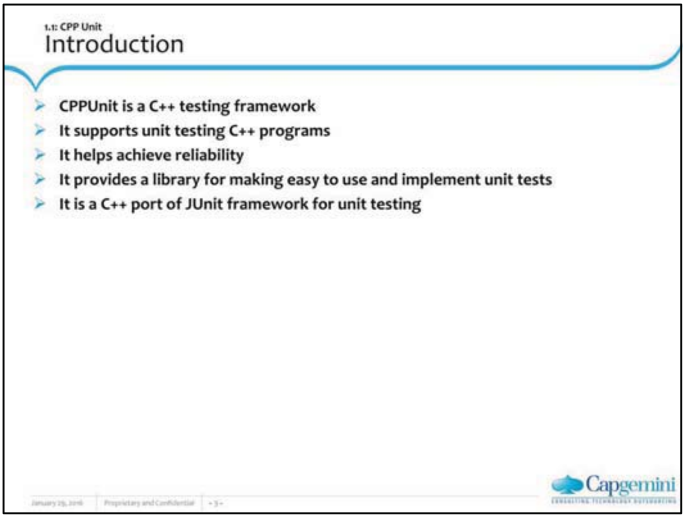
CPP Unit

January 29, 2016 | Proprietary and Confidential | - 1 -



Lesson Objectives

- **In this lesson, you will learn:**
  - Introduction to CPPUnit
  - Unit Testing
  - Testing tools
  - CPPUnit
    - Installing CPP Unit
    - Test Case programming
    - Assertion macros
    - Exception Macros
  - Test Driven Development



1.1: CPP Unit

Introduction

- CPPUnit is a C++ testing framework
- It supports unit testing C++ programs
- It helps achieve reliability
- It provides a library for making easy to use and implement unit tests
- It is a C++ port of JUnit framework for unit testing

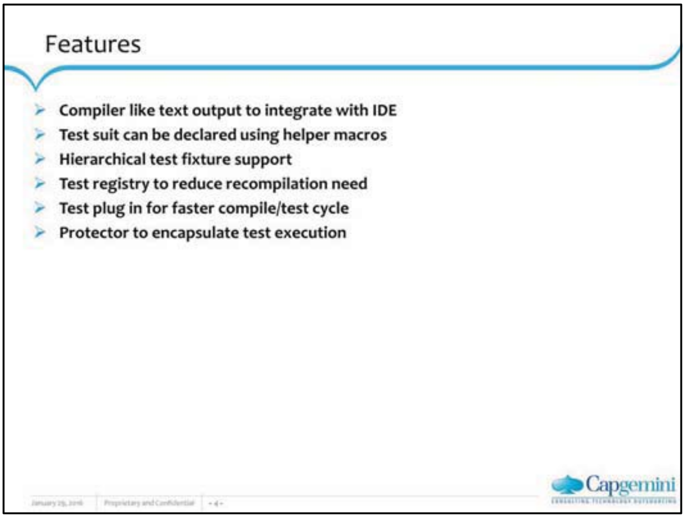
January 29, 2016 | Proprietary and Confidential | - 3 -

CPP Unit Testing

- **Introduction**

CppUnit is a testing framework developed by Erich Gamma and Kent Beck. It is ported by Michael Feathers. The purpose of CppUnit is to support developers in doing their unit testing of C++ programs. For C++ language projects, we can use CppUnit extensively for testing purpose. It will help us achieve reliability for our projects.

CppUnit is a member of the “Xunit family” test framework. For automatic testing the test output is in XML or in the text format and for supervised tests it is GUI based.



Features

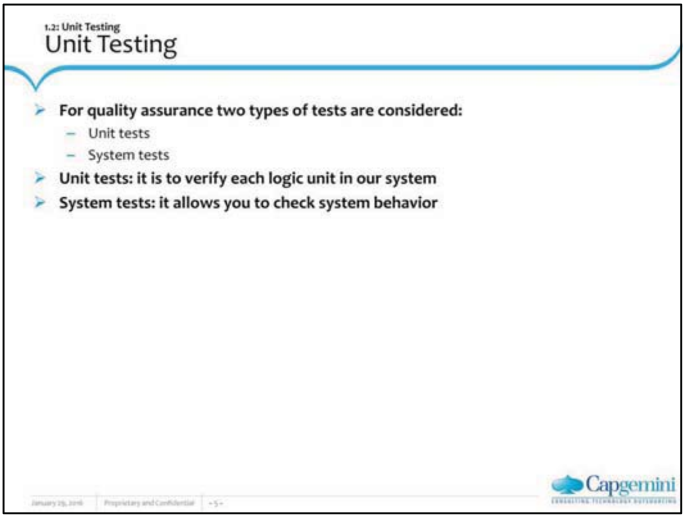
- Compiler like text output to integrate with IDE
- Test suit can be declared using helper macros
- Hierarchical test fixture support
- Test registry to reduce recompilation need
- Test plug in for faster compile/test cycle
- Protector to encapsulate test execution

January 29, 2016 | Proprietary and Confidential | - 4 -

Capgemini
CONSULTING. TECHNOLOGY. OUTSOURCING

CppUnit features:

- Compiler like text output to integrate with IDE
- Test suit can be declared using helper macros hence test programming becomes easy and fast.
- Hierarchical test fixture support – different test cases can be created and executed together or separately
- Test registry to reduce recompilation need
- Test plug in for faster compile/test cycle
- Protector to encapsulate test execution



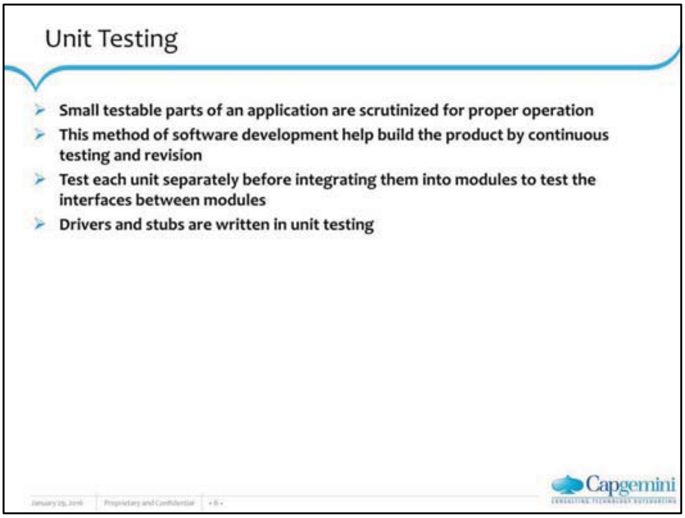
1.2: Unit Testing

Unit Testing

- For quality assurance two types of tests are considered:
  - Unit tests
  - System tests
- Unit tests: it is to verify each logic unit in our system
- System tests: it allows you to check system behavior

January 29, 2016 | Proprietary and Confidential | - 5 -

Within Quality assurance process, we consider two types of tests mainly:
1. Unit test: These are also called as acceptance tests. It is a set of verifications we can make to each logic unit in our system. With each test, we are testing its behavior, without keeping in mind all collaborations with other units.
2. System tests: This is also called as integration test. This test allows us to check systems behavior, emphasizing unit collaborations.



Unit Testing

- Small testable parts of an application are scrutinized for proper operation
- This method of software development help build the product by continuous testing and revision
- Test each unit separately before integrating them into modules to test the interfaces between modules
- Drivers and stubs are written in unit testing

January 29, 2016 | Proprietary and Confidential | - 6 -

Unit Testing:

It is a software development process in which the smallest testable parts (units) of an application are tested individually and independently for proper operations.

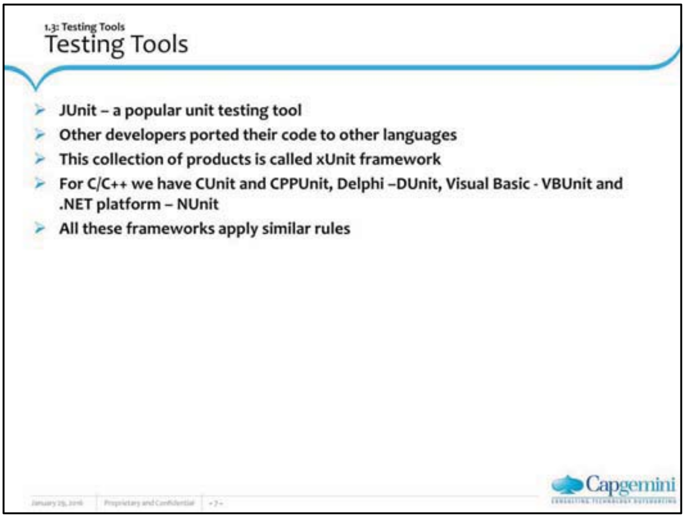
Generally Unit testing is automated by it can also be done manually.

This method of software development takes a meticulous approach to building a product by means of continual testing and revision.

Unit testing involves only those characteristics that are vital to the performance of the unit under test. This encourages developers to modify the source code without immediate concerns about how such changes might affect the functioning of other units or the program as a whole.

Once all the units in a program have been found to be working in the most efficient and error free manner possible, larger components of the program can be evaluated by means of integration testing.

The common approach to unit testing requires drivers and stubs to be written. The driver simulates a calling unit and the stub simulates a called unit.



1.3: Testing Tools

Testing Tools

- JUnit – a popular unit testing tool
- Other developers ported their code to other languages
- This collection of products is called xUnit framework
- For C/C++ we have CUnit and CPPUnit, Delphi –DUnit, Visual Basic - VBUnit and .NET platform – NUnit
- All these frameworks apply similar rules

January 29, 2016 | Proprietary and Confidential | - 7 -

Testing Tools:

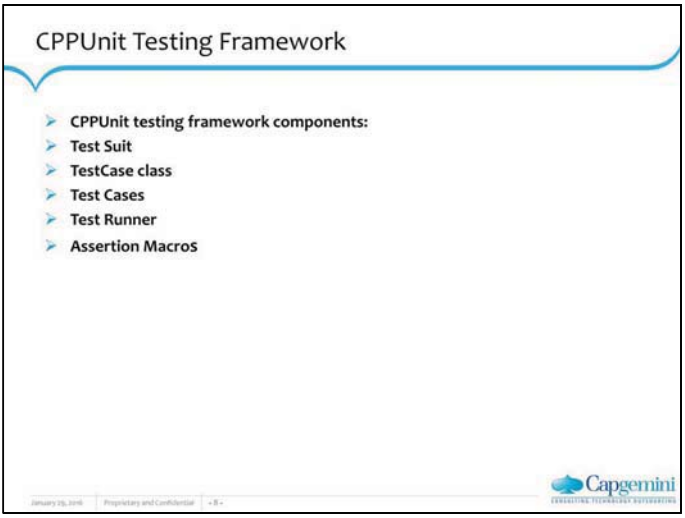
Kent Beck and Eric Gamma worte a set of Java classes to make unit testing automated. This was JUnit, which became very popular in testing world. Other developers ported their code to other languages, building a big collection of products called xUnit framework.

Some of them are:

C/C++	CUnit and CPPUnit
Delphi	DUnit
Visual Basic	VBUnit

.NET framework NUnit

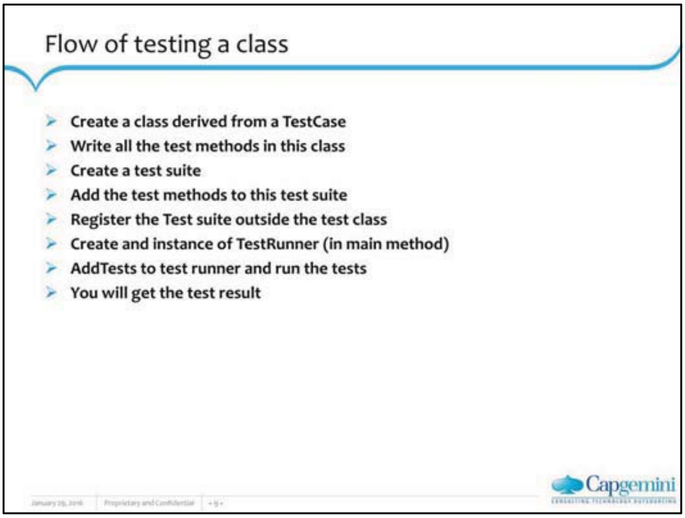
All these frame works apply similar rules, and you can use one if you are familiar with the other. There would be few language dependency exceptions.



CPPUnit Testing Framework

- CPPUnit testing framework components:
- Test Suit
- TestCase class
- Test Cases
- Test Runner
- Assertion Macros

CPPUnit Testing Framework:
CPPUnit testing framework is made up of following components :-
1. Test Suit: Test Suit is a collection of test cases to be executed together.
2. TestCase class is the class which supports creating test cases and the test suit for the class that is to be tested.
3. Test Cases: These are the test methods that will test the class/method.
4. Test Runner: this is an environment that will show the test results
5. Assertion macros: The assertion macros are the once which are used to check for assert conditions.

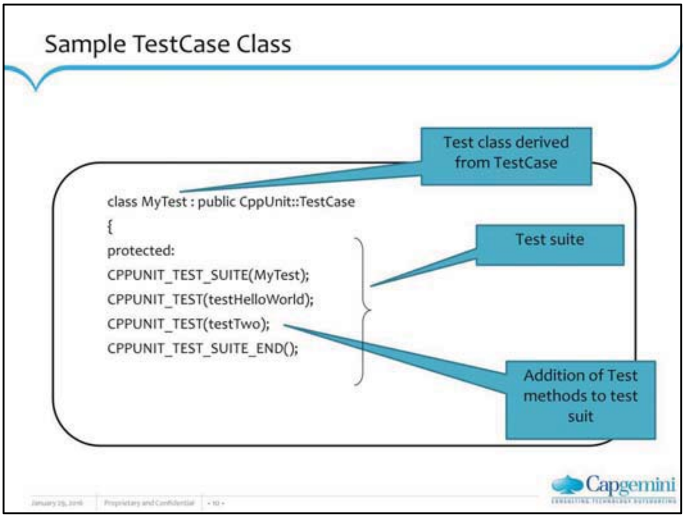


Flow of testing a class

- Create a class derived from a TestCase
- Write all the test methods in this class
- Create a test suite
- Add the test methods to this test suite
- Register the Test suite outside the test class
- Create and instance of TestRunner (in main method)
- AddTests to test runner and then run the tests
- You will get the test result

January 29, 2016 | Proprietary and Confidential | - 9 -

Flow of testing a class
For testing any class first we have to create a class derived from a class TestCase. This class will have all the test cases that we want to create for testing different methods with the class to be tested.
Some of the test methods will have to be executed together, for this we have to create a test suite. Use CPPUNIT_TEST_SUITE(<class name having test methods>) macro to create the Test suite. This macro take the parameter as the name of the class that contains test methods. Each test is added to the test suite with the macro CPPUNIT_TEST(<test method name>) macro which takes the parameter as the test method name.
After creating the test suite, define all the test methods in the Test class.
Register the Test suite outside the test class using macro:
CPPUNIT_TEST_SUITE_REGISTRATION(<class name having test suite>);
Create main method and instantiate TestRunner class to run the tests.
Add the test suite to the TestRunner and then call run method.



Sample TestCase Class

```
class MyTest : public CppUnit::TestCase
{
protected:
CPPUNIT_TEST_SUITE(MyTest);
CPPUNIT_TEST(testHelloWorld);
CPPUNIT_TEST(testTwo);
CPPUNIT_TEST_SUITE_END();
```

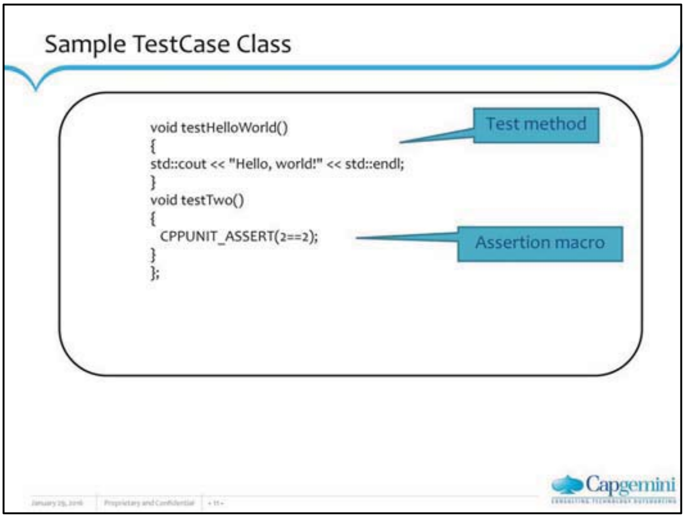
Test class derived from TestCase

Test suite

Addition of Test methods to test suit

January 29, 2016 | Proprietary and Confidential | - 10 -

CPP Framework provides us with following:
CppUnit::TestCase class

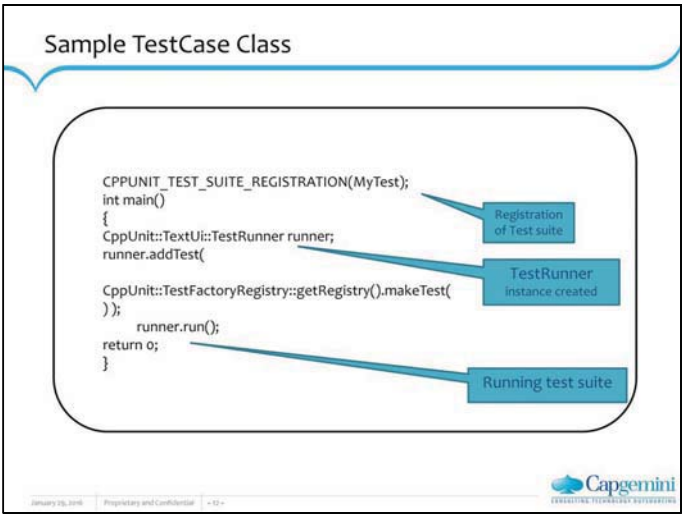


Sample TestCase Class

```
void testHelloWorld()
{
std::cout << "Hello, world!" << std::endl;
}
void testTwo()
{
  CPPUNIT_ASSERT(2==2);
}
};
```

Test method

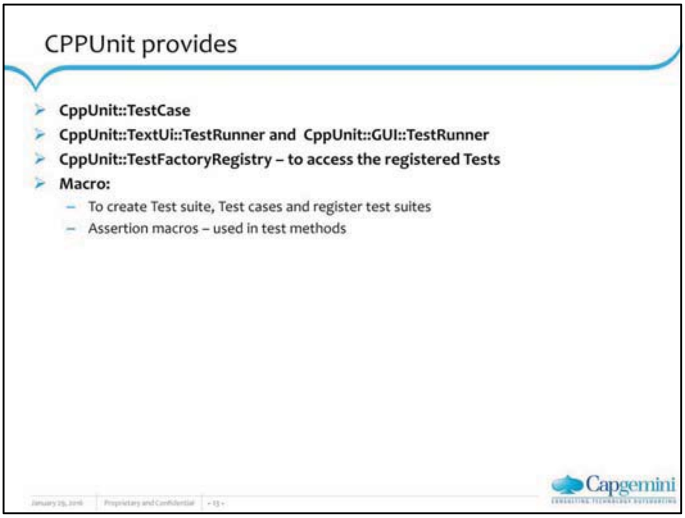
Assertion macro



Sample TestCase Class

```
CPPUNIT_TEST_SUITE_REGISTRATION(MyTest);
int main()
{
CppUnit::TextUi::TestRunner runner;
runner.addTest(

CppUnit::TestFactoryRegistry::getRegistry().makeTest(
) );
      runner.run();
return 0;
}
```

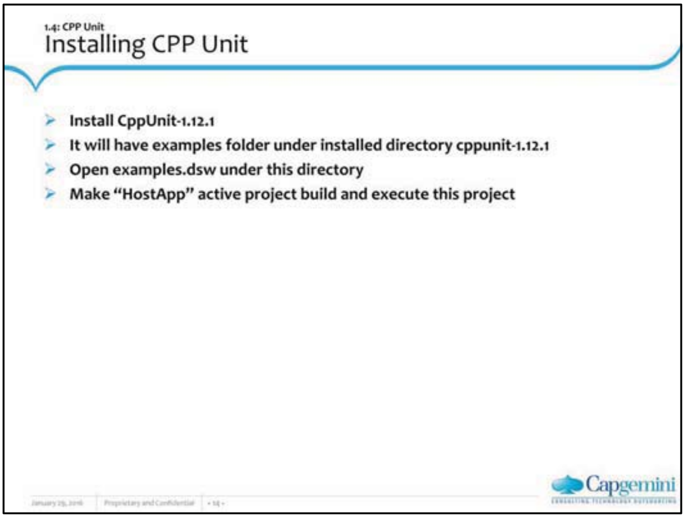
January 29, 2016 | Proprietary and Confidential | - 12 -



CPPUnit provides

- **CppUnit::TestCase**
- **CppUnit::TextUi::TestRunner and CppUnit::GUI::TestRunner**
- **CppUnit::TestFactoryRegistry – to access the registered Tests**
- **Macro:**
  - To create Test suite, Test cases and register test suites
  - Assertion macros – used in test methods

January 19, 2016 | Proprietary and Confidential | - 13 -



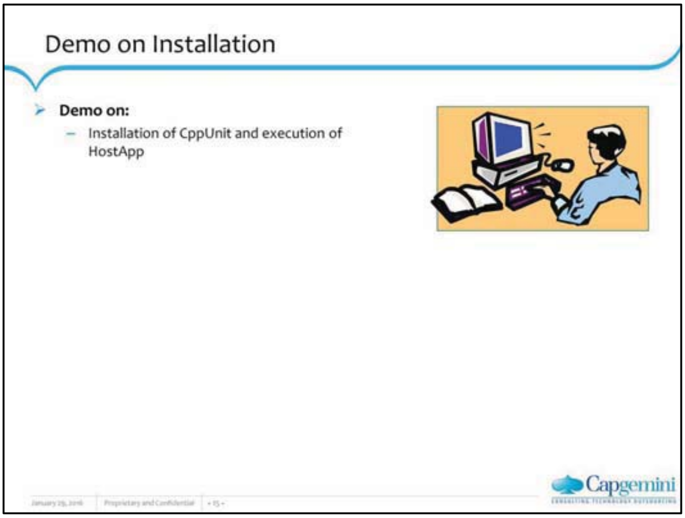
1.4: CPP Unit

Installing CPP Unit

- Install CppUnit-1.12.1
- It will have examples folder under installed directory cppunit-1.12.1
- Open examples.dsw under this directory
- Make "HostApp" active project build and execute this project

January 25, 2016 | Proprietary and Confidential | - 14 -

All steps and information about building libraries can be found in *INSTALL-WIN32.txt* file, inside CPPUnit distribution. Once all binaries are built, you can write your own Test Suites.

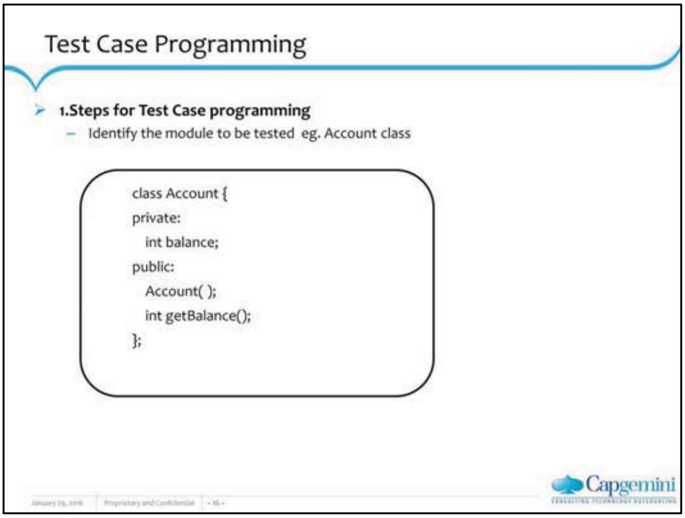


Demo on Installation

- **Demo on:**
  - Installation of CppUnit and execution of HostApp

January 29, 2016 | Proprietary and Confidential | - 15 -

Add the notes here.

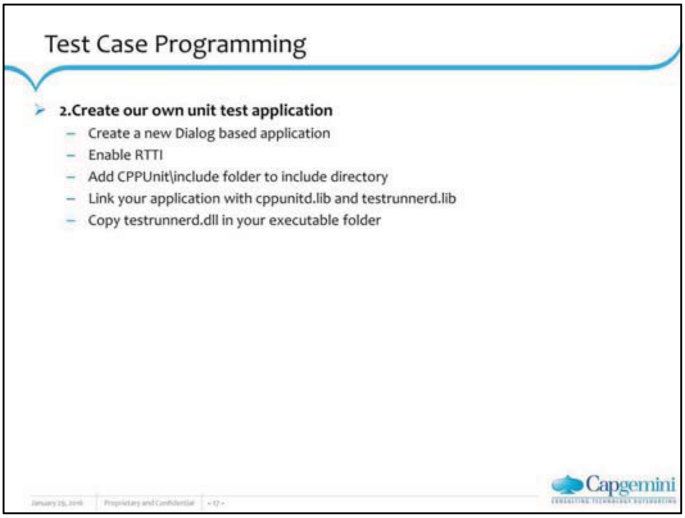


Test Case Programming

- **1.Steps for Test Case programming**
  - Identify the module to be tested eg. Account class

```
class Account {
private:
  int balance;
public:
  Account( );
  int getBalance();
};
```

This class Account has a constructor and a getBalance() method.
Important thing is that we must be sure this class is doing all the things it must do, here creating the Account object and getting balance.
To verify this we are going to create a new Test Suit with two test cases: one for constructor and another for getBalance()

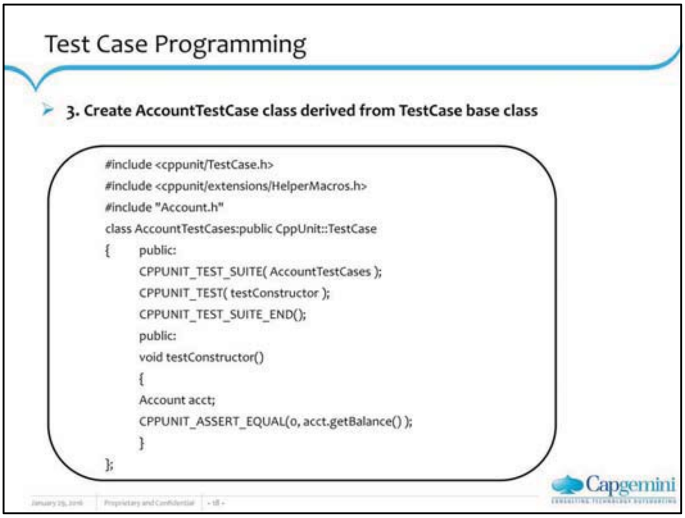


Test Case Programming

- **2.Create our own unit test application**
  - Create a new Dialog based application
  - Enable RTTI
  - Add CPPUnit\include folder to include directory
  - Link your application with cppunitd.lib and testrunnerd.lib
  - Copy testrunnerd.dll in your executable folder

January 29, 2016 | Proprietary and Confidential | - 17 -

Please refer to the Lab guide for detailed setting to be done in visual studio project.



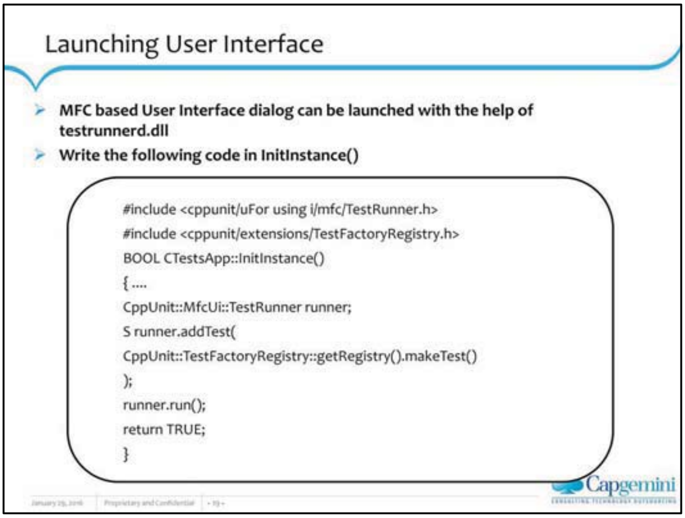
Test Case Programming

- 3. Create AccountTestCase class derived from TestCase base class

```
#include <cppunit/TestCase.h>
#include <cppunit/extensions/HelperMacros.h>
#include "Account.h"
class AccountTestCases:public CppUnit::TestCase
{       public:
        CPPUNIT_TEST_SUITE( AccountTestCases );
        CPPUNIT_TEST( testConstructor );
        CPPUNIT_TEST_SUITE_END();
        public:
        void testConstructor()
        {
        Account acct;
        CPPUNIT_ASSERT_EQUAL(0, acct.getBalance() );
        }
};
```

January 29, 2016 | Proprietary and Confidential | - 18 -

TestCase.h helps us derive our class from TestCase
HelperMacro.h gives us the definitions of macors like: CPPUNIT_TEST_SUITE, CPPUNIT_TEST, CPPUNIT_ASSERT_EQUAL etc.

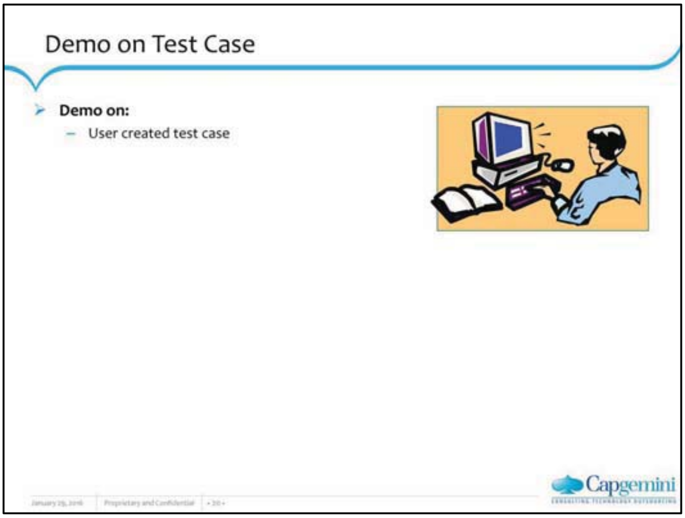


Launching User Interface

- MFC based User Interface dialog can be launched with the help of testrunnerd.dll
- Write the following code in InitInstance()

```
#include <cppunit/uFor using i/mfc/TestRunner.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
BOOL CTestsApp::InitInstance()
{ ....
CppUnit::MfcUi::TestRunner runner;
S runner.addTest(
CppUnit::TestFactoryRegistry::getRegistry().makeTest()
);
runner.run();
return TRUE;
}
```

January 29, 2016 | Proprietary and Confidential | - 19 -

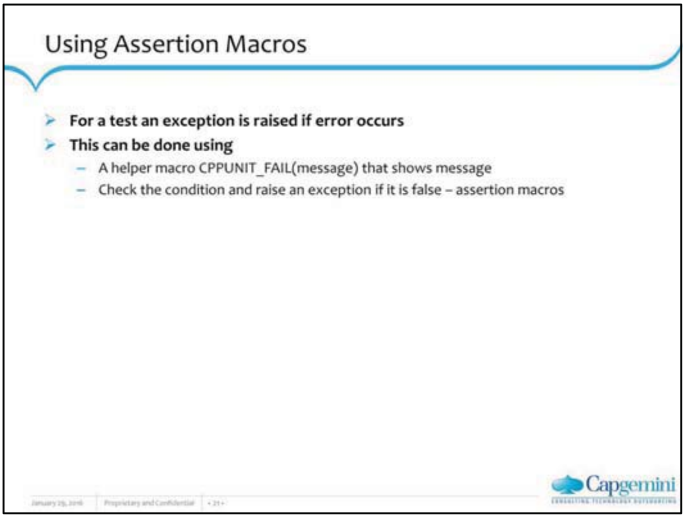


Demo on Test Case

- **Demo on:**
  - User created test case

January 29, 2016 | Proprietary and Confidential | - 20 -

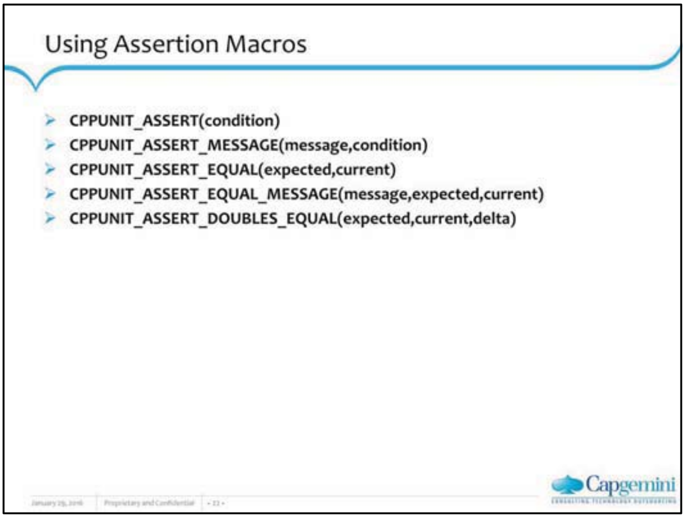
Add the notes here.



Using Assertion Macros

- **For a test an exception is raised if error occurs**
- **This can be done using**
  - A helper macro CPPUNIT_FAIL(message) that shows message
  - Check the condition and raise an exception if it is false – assertion macros

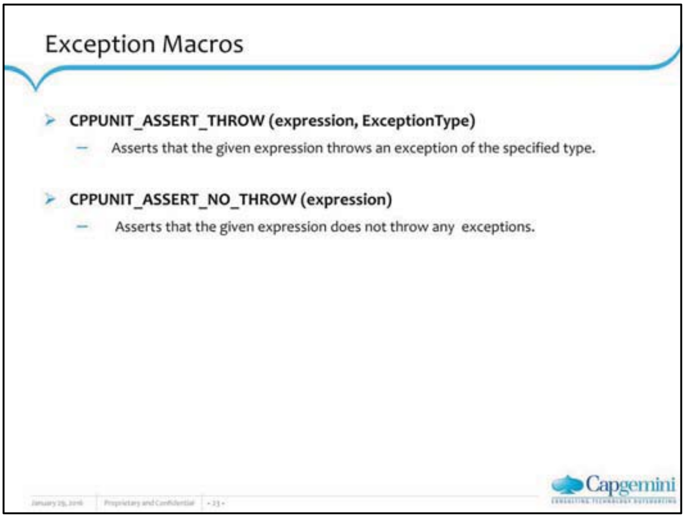
January 29, 2016 | Proprietary and Confidential | - 21 -



Using Assertion Macros

- CPPUNIT_ASSERT(condition)
- CPPUNIT_ASSERT_MESSAGE(message,condition)
- CPPUNIT_ASSERT_EQUAL(expected,current)
- CPPUNIT_ASSERT_EQUAL_MESSAGE(message,expected,current)
- CPPUNIT_ASSERT_DOUBLES_EQUAL(expected,current,delta)

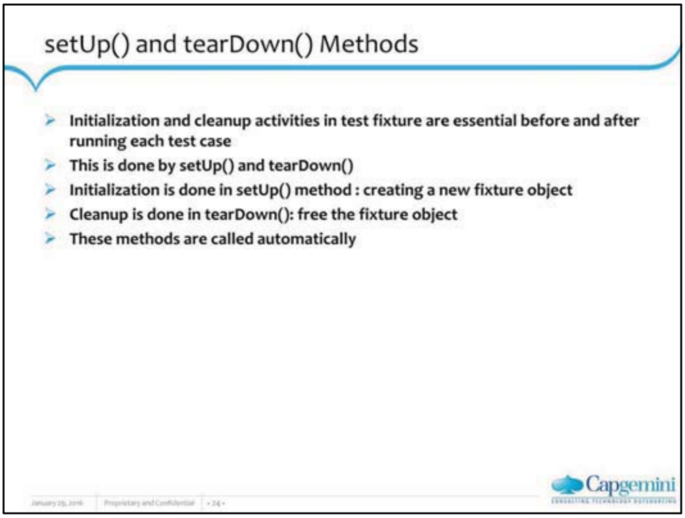
1. CPPUNIT_ASSERT(condition): checks condition and throws an exception if it's false.
2. CPPUNIT_ASSERT_MESSAGE(message, condition): checks condition and throws an exception and showing specified message if it is false.
3. CPPUNIT_ASSERT_EQUAL(expected,current): checks if expected is the same as current, and raises exception showing expected and current values.
4. CPPUNIT_ASSERT_EQUAL_MESSAGE(message,expected,current): checks if expected is the same as actual, and raises exception showing expected and current values, and specified message.
5. CPPUNIT_ASSERT_DOUBLES_EQUAL(expected,current,delta): checks if expected and current difference is smaller than delta. If it fails, expected and current values are shown.



Exception Macros

- **CPPUNIT_ASSERT_THROW (expression, ExceptionType)**
  - Asserts that the given expression throws an exception of the specified type.
- **CPPUNIT_ASSERT_NO_THROW (expression)**
  - Asserts that the given expression does not throw any exceptions.

January 29, 2016 | Proprietary and Confidential | - 23 -



setUp() and tearDown() Methods

- Initialization and cleanup activities in test fixture are essential before and after running each test case
- This is done by setUp() and tearDown()
- Initialization is done in setUp() method : creating a new fixture object
- Cleanup is done in tearDown(): free the fixture object
- These methods are called automatically

January 29, 2016 | Proprietary and Confidential | - 24 -

setUp() and tearDown() methods

Our class can override setUp() and tearDown() methods of TestCase class.

These methods are called automatically. The execution is setUp() is called before each test starts and tearDown() is called after the test case ends.

Example:

```
CPPUNIT_TEST_SUITE_REGISTRATION(DiskDataTestCase);
void DiskDataTestCase::setUp()
{
                    fixture = new DiskData();
}
void DiskDataTestCase::tearDown()
{
                    delete fixture;
                    fixture = NULL;
}
```

The calling sequence will be:

Call the setUp() method; create our fixture object.

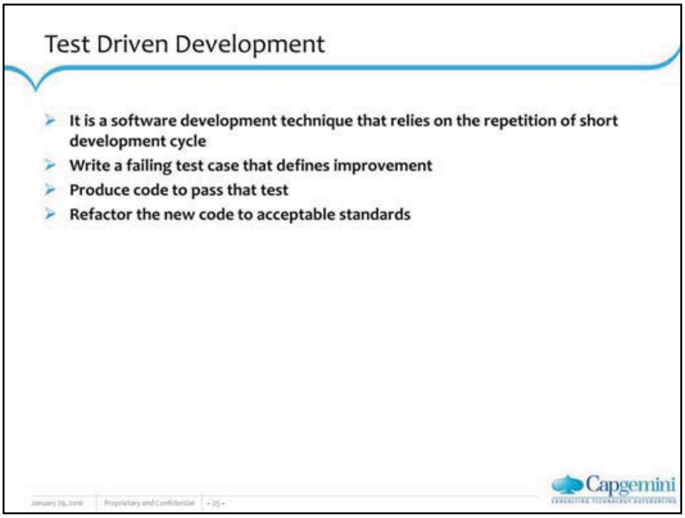
Call the first test case method.

Call the tearDown() method; free the fixture object.

Call the setUp() method; create our fixture object.

Call the second test case method.

Call the tearDown() method; free the fixture object

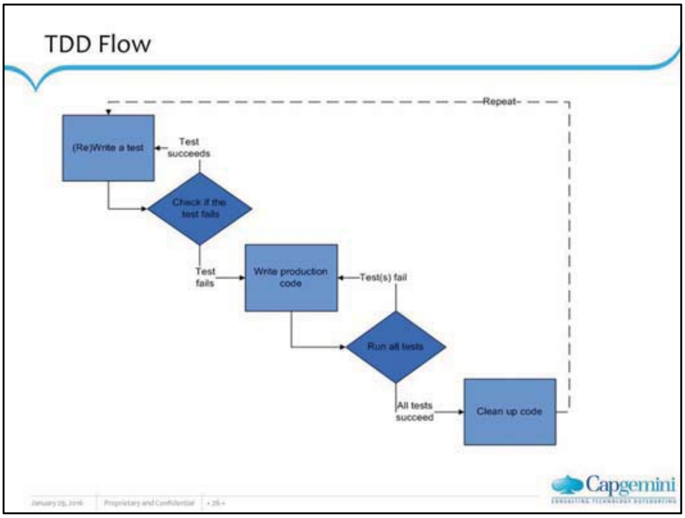


Test Driven Development

- It is a software development technique that relies on the repetition of short development cycle
- Write a failing test case that defines improvement
- Produce code to pass that test
- Refactor the new code to acceptable standards

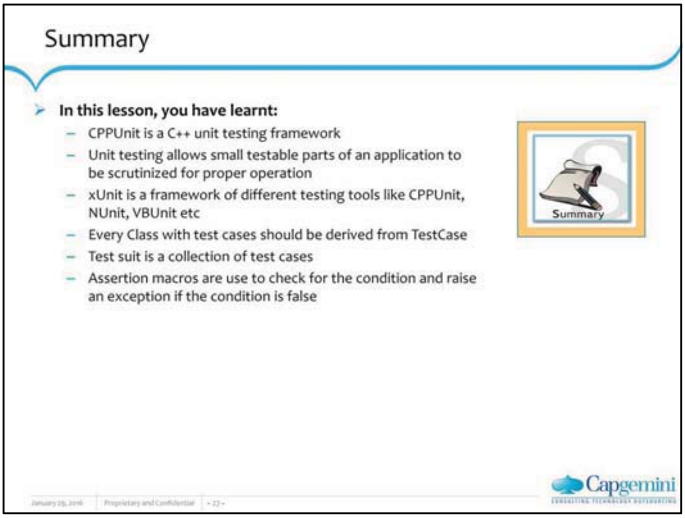
January 25, 2016 | Proprietary and Confidential | - 25 -

Test driven development is a technique that relies on the repetition of a short development cycle. Developer first writes a failing automated test case that defines a desired improvement. Code is then written to pass this test. This refactors the new code to acceptable standards. TDD hence encourages simple design and builds confidence. Test driven evelopment is related to the test first programming concept.



TDD Flow

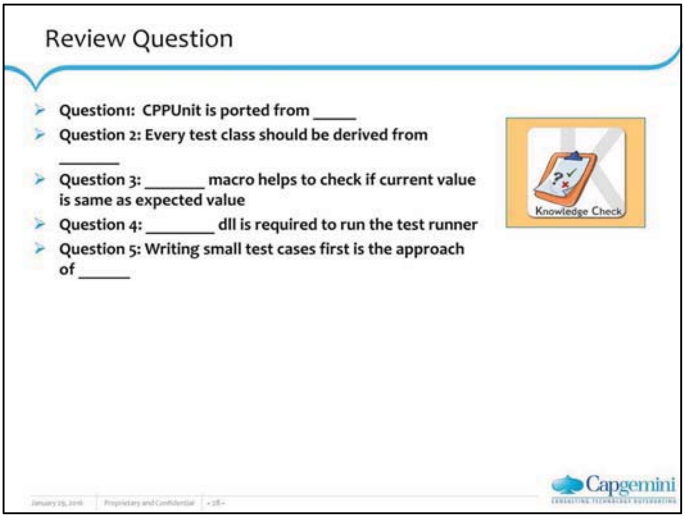
Test driven development requires developers to create automated tests that define code requirements before writing the code. The tests contains assertions that are either true or false. Passing the test confirms correct behavior as developer evolve and refactor the code.
In such scenario testing framework like CPPUnit is mode usefull.



Summary

- **In this lesson, you have learnt:**
  - CPPUnit is a C++ unit testing framework
  - Unit testing allows small testable parts of an application to be scrutinized for proper operation
  - xUnit is a framework of different testing tools like CPPUnit, NUnit, VBUnit etc
  - Every Class with test cases should be derived from TestCase
  - Test suit is a collection of test cases
  - Assertion macros are use to check for the condition and raise an exception if the condition is false

Add the notes here.



Review Question

- Question1: CPPUnit is ported from _____
- Question 2: Every test class should be derived from _______
- Question 3: _______ macro helps to check if current value is same as expected value
- Question 4: ________ dll is required to run the test runner
- Question 5: Writing small test cases first is the approach of ______

Knowledge Check

January 25, 2016 | Proprietary and Confidential | - 28 -

Add the notes here.

CPPUnit
Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
05-Feb-2009	0.1D	Vaishali Kunchur	Content Creation
09-Feb-2009		CLS team	Review

Lab 1. Table of Contents

Lab 2. Getting Started

Overview

This lab book is a guided tour for learning DBMS SQL. It comprises ‘To Do’ assignments. Follow the steps provided and work out the ‘To Do’ assignments.

Setup Checklist

Here is what is expected on your machine in order for the lab to work

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP
- Connectivity to a Linux /Unix Server which has got gcc and g++ compiler installed in it

Instructions

- Create a directory by your name in the home directory. For each lab exercise create a directory as lab <lab number>

Learning More

- Linux application development – Michael K. Johnson
- Managing Projects with GNU make, 3rd Edition - Robert Mecklenburg
- http://www.network-theory.co.uk/docs/gccintro/gccintro_4.html

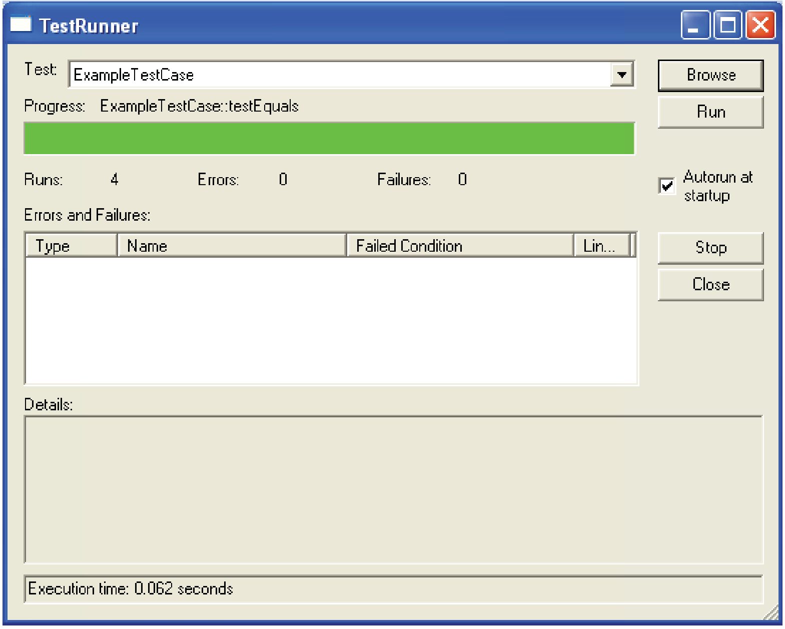
Lab 3. Installing CPPunit

Goals	• Learn and Understand the process of : • Installing CPPUnit
Time	180 minutes

1.1 Installation of CPPUnit

Step 1: Unzip cppunit-1.12.1 zip file.
Open cppunit-1.12.1\examples\examples.dsw
Click Yes if we get format conversion message box.

Step 2: Make HostApp as active project and build it.
Execute HostApp and you will see the Test Runner as shown below



TestRunner

Test: ExampleTestCase

Progress: ExampleTestCase::testEquals

Runs: 4 Errors: 0 Failures: 0

☑ Autorun at startup

Errors and Failures:

Type	Name	Failed Condition	Lin...

Details:

Execution time: 0.062 seconds

Step 3: Click on Browse to select the test suit/test cases and click select. And RUN.

Step 4: If all these steps are successful test runs then CppUnit is installed successfully.

1.2 : Create our own test cases:

<u>Follow the following steps to create a project with CPPUnit support:</u>

Step 1: Steps for Setting a Project to Use CPPUnit: (In Visual Studio 2008)

a. One time setting:

Go to Tools -> Options -> Projects and Solutions -> VC++ Directories
Select Source Files from "Show directories for:" Select "Include files
Add the directory Path for CPPUnit1.12.1\INCLUDE
Same step should be repeated for "Source files" and "Library files" under "Show directory for:"

Eg:D:\CPPUNIT\CPPUNIT-1.12.0\INCLUDE

Step 2: Every Project for which you need to use CPPUnit, follow the below steps:

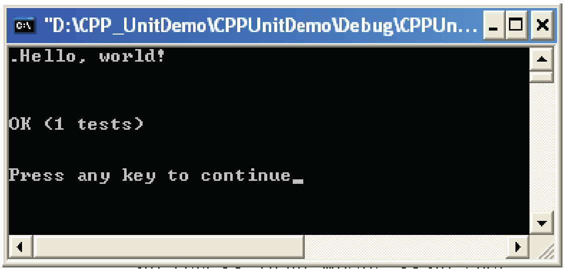
i. Create a new "Win32 Console Based" Project with name CPPUnitDemo
ii. Create a New C++ source file with the name "HelloCPPUnitTest"
iii. Right click on Project (HelloCPPUnitTest) -> select "Properties
iv. In Microsoft Foundation Classes: select "Use MFC in Shared DLL"
v. Go to Configuration Properties-> C/C++ -> Language
vi. Enable Run-Time Type Information (RTTI) should have value "Yes"
vii. Select Linker -> General and set "Additional Library Directories" as "Debug/cppunitd.lib Debug/testrunnerd.lib"
viii. Select Input in Linker and set "Additional Dependencies" as "Debug/cppunitd.lib Debug/testrunnerd.lib" and "Ignore Specific Library" as "/NODEFAULTLIB:library".
ix. Copy cppunitd.lib, testrunnerd.lib, cppunitd_dll.dll, testrunnerd.dll files from CppUnit installable into Debug directory of CPPUnitDemo project.

Enter following code to HelloCPPUnitTest.cpp:

```
#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/TestCase.h>

#include <string>
class MyTest : public CppUnit::TestCase
{

protected:

CPPUNIT_TEST_SUITE(MyTest);
CPPUNIT_TEST(testHelloWorld);
CPPUNIT_TEST(testTwo);
CPPUNIT_TEST_SUITE_END();
        void testHelloWorld()
        {
        std::cout << "Hello, world!" << std::endl;
        }
        void testTwo()
```

```
            {
              CPPUNIT_ASSERT(2==2);
            }
};
CPPUNIT_TEST_SUITE_REGISTRATION(MyTest);
int main()
{
            CppUnit::TextUi::TestRunner runner;
            runner.addTest(
            CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
            runner.run();
            return 0;
}
```

Step 3: Build and execute the project to get following output.



```
.Hello, world!


OK (1 tests)


Press any key to continue_
```

Lab 4. Test a class using CPPunit

Goals	• Learn and Understand the process of : • Installing CPPUnit
Time	60 minutes

2.1 Create a Complex class

Write a class Complex, having two operators: == and + that are overloaded for appropriate functionality. We will create the test cases to verify the functionality of these two overloaded operators of Complex class.

Step1: Create a New Project "TestComplex", which is a Win32 Console based application

Step 2: Add a new file "Complex.h" with following code

```
class Complex {
 friend bool operator ==(const Complex& a, const Complex& b);
 double real, imaginary;
public:
 Complex(){}
 Complex( double r, double i = 0 ) : real(r), imaginary(i)
 {
 }
};
```

Step 3: Add a new file "Complex.cpp" with following code

```
#include "Complex.h"

bool operator ==( const Complex &a, const Complex &b )
{
 return a.real == b.real  &&  a.imaginary == b.imaginary;
}
```

Step 4: Add following code to "TestComplex.cpp"

```
#include <string>
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/TestCase.h>
#include <cppunit/TestCaller.h>
#include "Complex.h"


class ComplexNumberTest : public CppUnit::TestCase {
public:
 ComplexNumberTest(){}
 void runTest() {
  CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
```

```
    CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
  }
};
int main()
{
   CppUnit::TextUi::TestRunner runner;
   runner.addTest(new CppUnit::TestCaller<ComplexNumberTest>
("runTest",&ComplexNumberTest::runTest));
   return runner.run()?1:0;
}
```

Step 5: Compile and execute the project to get following output on Console:

```
OK (1 tests)
```

Step 6: Now change the code:

```
CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
```

To

```
CPPUNIT_ASSERT( (Complex (1, 1) == Complex (2, 2)) );
```

Compile and execute the project and check the output on Console.

2.2 Using setup() and teardown()

Step 1: Now make following changes to the code of "ComplexNumberTest.cpp":
include file cppunit/TestFixture.h

Step 2: Extend ComplexNumberTest class from public CppUnit::TestFixture

Step 3: Add following private members:

```
Complex *m_10_1, *m_1_1, *m_11_2;
```

Step 4: Add these methods setup() for initialization and tearDown():

```
void setUp()
{
   m_10_1 = new Complex(10,1);
   m_1_1 = new Complex(1,1);
   m_11_2 = new Complex(11,2);
}
void tearDown()
{
   delete m_10_1;
   delete m_1_1;
   delete m_11_2;
}
```

Step 5: Add following tests for equality and addition:

```
void testEquality()
{
```

```
CPPUNIT_ASSERT(*m_10_1 == *m_10_1);
CPPUNIT_ASSERT(*m_1_1 == *m_1_1);
CPPUNIT_ASSERT(*m_11_2 == *m_11_2);
}
void testAddition()
{   CPPUNIT_ASSERT(*m_10_1 + *m_1_1 == *m_11_2);
}
```

Step 6: Add following code to “Complex.h”:

```
friend Complex operator +(const Complex& a, const Complex& b);
```

Step 7: Add following code to “Complex.cpp”:

```
#include "Complex.h"

Complex operator +( const Complex &a, const Complex &b )
{
 return Complex (a.real + b.real ,  a.imaginary + b.imaginary);
}
```

Step 8: “TestComplex.cpp” using helper macros

```
#include <cppunit/extensions/HelperMacros.h>
#include <string>
#include <cppunit/ui/text/TestRunner.h>
#include "Complex.h"


class ComplexNumberTest : public CPPUNIT_NS::TestFixture {
private:
Complex *m_10_1, *m_1_1, *m_11_2;
protected:
CPPUNIT_TEST_SUITE(ComplexNumberTest);
CPPUNIT_TEST(testEquality);
CPPUNIT_TEST(testAddition);
CPPUNIT_TEST_SUITE_END();
public:
ComplexNumberTest(){}

 void setUp()
 {
 m_10_1 = new Complex(10,1);
 m_1_1 = new Complex(1,1);
 m_11_2 = new Complex(11,2);
 }
 void tearDown()
 {
 delete m_10_1;
 delete m_1_1;
 delete m_11_2;
 }
 void runTest() {
  CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
  CPPUNIT_ASSERT( (Complex (1, 1) == Complex (2, 2)) );
 }
 void testEquality()
 {
  CPPUNIT_ASSERT(*m_10_1 == *m_10_1);
  CPPUNIT_ASSERT(*m_1_1 == *m_1_1);
  CPPUNIT_ASSERT(*m_11_2 == *m_11_2);
 }
 void testAddition()
 {
   CPPUNIT_ASSERT(*m_10_1 + *m_1_1 == *m_11_2);
 }
};
CPPUNIT_TEST_SUITE_REGISTRATION(ComplexNumberTest);
int main()
{
  CppUnit::TextUi::TestRunner runner;
  runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
  return runner.run()?0:1;
}
```

Lab 5. GUI application using CPPunit

Write a class DiskData, having two responsibilities: load and store data inside a file. We will create the test cases to verify the functionality of these two responsibilities of DiskData class.

Step1: Create a New Project "TestDemo", which is a dialog based application.

Step2: Right click on TestDemo project in Solution Explorer -> Properties
Select Configuration Properties -> C/C++ ->Language. Ensure "Enable Run-Time Type Info" is "Yes" and click OK

Step 3: Select Tools -> Options -> Projects and Solutions -> VC++ Directories
Select "Include files" under Show directories for.
Select New Line option and browse for *cppunit-1.12.1\include.* Click OK. Again Click OK.

Step 4: Build the project

Step 5: Copy testrunner.lib and cppunitd_dll.lib file in Debug folder

Step 6: Go to project Properties. Select Configuration Properties -> Linker -> Input -> Additional Dependencies -> Enter "Debug\cppunitd_dll.lib" and "Debug\testrunnerd.lib" and click OK

Step 7: Add New C++ class DiskData and enter following code in DiskData.h

```
typedef struct _DATA
{
   int  number;
   char string[256];
} DATA, *LPDATA;
class DiskData
{
public:
   DiskData();
   ~DiskData();
   LPDATA getData();
   void setData(LPDATA value);
   bool load(char *filename);
   bool store(char *filename);
private:
```

```
         DATA m_data;
        };
```

Step 8: Add another class called DiskDataTestCase and add following code to DiskDataTestCase.cpp

```
#if !defined(DISKDATA_TESTCASE_H_INCLUDED)
#define DISKDATA_TESTCASE_H_INCLUDED

#if _MSC_VER > 1000
#pragma once
#endif   // _MSC_VER > 1000
#include <cppunit/TestCase.h>
#include <cppunit/extensions/HelperMacros.h>
#include "DiskData.h"

class DiskDataTestCase : public CppUnit::TestCase
{
 CPPUNIT_TEST_SUITE(DiskDataTestCase);
   CPPUNIT_TEST(loadTest);
   CPPUNIT_TEST(storeTest);
 CPPUNIT_TEST_SUITE_END();
public:
  void setUp();
  void tearDown();
protected:
  void loadTest();
  void storeTest();
private:
  DiskData *fixture;
};
#endif
```

Step 9: Add following code to DiskDataTestCase.cpp

```
 #include "DiskDataTestCase.h"
```

```
CPPUNIT_TEST_SUITE_REGISTRATION(DiskDataTestCase);
void DiskDataTestCase::setUp()
{
  fixture = new DiskData();
}
void DiskDataTestCase::tearDown()
{
  delete fixture;
  fixture = NULL;
}
void DiskDataTestCase::loadTest()
{
  // our load test logic
}
void DiskDataTestCase::storeTest()
{
  // our store test logic
}
```

Step 10: Copy testrunner.dll and cppunitd_dll.dll file in TestDemo\TestDemo folder

Step 11: Add following code to InitInstance() method

```
#include <cppunit/ui/mfc/TestRunner.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
BOOL CTestDemoDlg::InitInstance()
{
  ....
  // declare a test runner, fill it with our registered tests,
  // and run them
  CppUnit::MfcUi::TestRunner runner;
  runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().
          makeTest() );
  runner.run();
```

```
      return TRUE;
    }
```

Step 12: Build and execute TestRunner will be displayed. Browse the tests from Test Hierarchy and Run

Step 13: Add following code to loadTest() to test the working of load() method.
// These are correct values stored in an auxiliary file

```
    #define AUX_FILENAME   "ok_data.dat"
    #define FILE_NUMBER    19
    #define FILE_STRING    "this is correct text stored in auxiliary
                          file"
    void DiskDataTestCase::loadTest()
    {
      // convert from relative to absolute path
      TCHAR    absoluteFilename[MAX_PATH];
      DWORD    size = MAX_PATH;
      strcpy(absoluteFilename, AUX_FILENAME);
      CPPUNIT_ASSERT( RelativeToAbsolutePath(absoluteFilename,                    &size) );
      // executes action
      CPPUNIT_ASSERT( fixture->load(absoluteFilename) );
      // ...and check results with assertions
      LPDATA    loadedData = fixture->getData();
      CPPUNIT_ASSERT(loadedData != NULL);
      CPPUNIT_ASSERT_EQUAL(FILE_NUMBER, loadedData->number);
      CPPUNIT_ASSERT( 0 == strcmp(FILE_STRING,
          fixture->getData()->string) );
    }
```

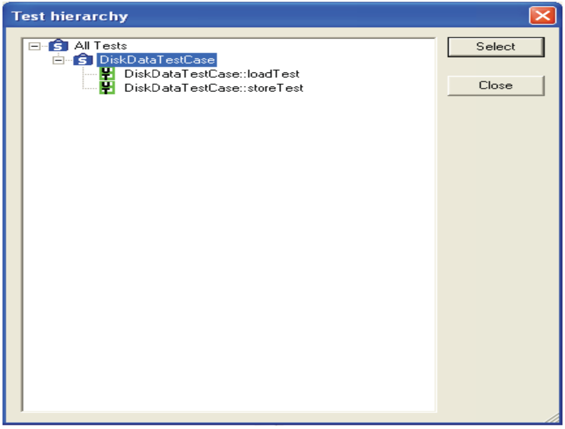
Step 14: Add following code to storeTest() to test the working of store() method.

```
   void DiskDataTestCase::storeTest()
   {
     DATA    d;
     DWORD    tmpSize, auxSize;
     BYTE    *tmpBuff, *auxBuff;
```

```
TCHAR   absoluteFilename[MAX_PATH];
DWORD   size = MAX_PATH;
// configure structure with known data
d.number = FILE_NUMBER;
strcpy(d.string, FILE_STRING);
// convert from relative to absolute path
strcpy(absoluteFilename, AUX_FILENAME);
CPPUNIT_ASSERT( RelativeToAbsolutePath(absoluteFilename,
                &size) );
// execute action
fixture->setData(&d);
CPPUNIT_ASSERT( fixture->store("data.tmp") );
// Read both files contents and check results
// ReadAllFileInMemory is an auxiliary function that allocates
// a buffer and saves all file content inside it. Caller should
// release the buffer.
// Check demo project for details
tmpSize = ReadAllFileInMemory("data.tmp", tmpBuff);
auxSize = ReadAllFileInMemory(absoluteFilename, auxBuff);
// files must exist
CPPUNIT_ASSERT_MESSAGE("New file doesn't exist?", tmpSize > 0);
CPPUNIT_ASSERT_MESSAGE("Aux file doesn't exist?", auxSize > 0);
// sizes must be valid
CPPUNIT_ASSERT(tmpSize != 0xFFFFFFFF);
CPPUNIT_ASSERT(auxSize != 0xFFFFFFFF);
// buffers must be valid
CPPUNIT_ASSERT(tmpBuff != NULL);
CPPUNIT_ASSERT(auxBuff != NULL);
// both files' sizes must be the same as DATA's size
CPPUNIT_ASSERT_EQUAL((DWORD) sizeof(DATA), tmpSize);
CPPUNIT_ASSERT_EQUAL(auxSize, tmpSize);
// both files' content must be the same
```

```
        CPPUNIT_ASSERT( 0 == memcmp(tmpBuff, auxBuff, sizeof(DATA)) );
        delete [] tmpBuff;
        delete [] auxBuff;
        ::DeleteFile("data.tmp");
    }
```

Step 15: Build and execute TestRunner will be displayed. Browse the tests from Test Hierarchy and Run Select all/some tests and click Run.



1.2: <TO DO>

Step 1: Create a C++ class myStack as given below:

```
Stack.h
class myStack
{
private:
        bool isEmpty;
        int valOnStack;
public :
        myStack(){isEmpty=true;};
        bool IsEmpty();
        void Push(int val);
        void Pop();
};
Stack.cpp
#include "Stack.h"

bool myStack::IsEmpty()
{
```

```
            return isEmpty;
}

void myStack::Push(int val)
{
             isEmpty = false;
}

void myStack::Pop()
{
            isEmpty= true;
}
```

Step 2: Create a Test suit for class myStack to test for all the methods in myStack class.

Step 3: Run the tests.

```
class Rect
{
   int len,bth;
public:
   Rect(int l,int b)
   {
      len=l;
      bth=b;
   }
   int area()
   {
      return len*bth;
   }

};
// ClassTest.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/TestCase.h>
#include "MyClass.h"
#include <conio.h>
class MyClassTest:public CppUnit::TestCase
{
public:
   Rect *r1;
   MyClassTest(){
```

```
   r1=new Rect(10,5);
  }

protected:
  CPPUNIT_TEST_SUITE(MyClassTest);
  CPPUNIT_TEST(testArea);
  CPPUNIT_TEST_SUITE_END();
  void testArea()
  {
    //r1=new Rect(10,15);
    CPPUNIT_ASSERT(r1->area()==150);
  }
};
CPPUNIT_TEST_SUITE_REGISTRATION(MyClassTest);
int main()
{
  CppUnit::TextUi::TestRunner runner;
  runner.addTest(CppUnit::TestFactoryRegistry::getRegistry().makeTest());
  runner.run();
  getch();
        return 0;
}
```