BCB에서 UnitTest 이용하기(2)

*** UnitTest 적용**

UnitTest를 적용해서 어떻게 개발을 하는지 살펴봅시다. UnitTest를 적용한다는 것은 TDD(Test Driven Development)[1]방법론을 적용하기 위해 구현된 framework를 따른다는 말이 됩니다. 실제로 TDD를 적용해서 개발하는 방법을 설명하기에는 지면상으로는 어려움이 많습니다. 생각의 흐름은 계속 이어지고, 코딩과 잦은 리펙토링(refactoring)[2]을 반복할 것이고 무엇보다 기존의 개발 방법과는 다르다는 것입니다.. 옆에 앉아서 중얼대며 개발하고 있는 모습을 보는 것도 좋은 방법입니다. 주위에 혹시 TDD로 개발하고 계시는 분이 있으시면, 꼭 부탁을 해서 옆에 지켜보시기 바랍니다. 짝 프로그래밍(Pair Programming)[3]을 한다면 TDD를 배우기 가장 좋은 조건입니다. 물론 문서를 찾아보는 것만으로도 충분히 습득할 수 있습니다. 어쨌든 이 문서의 목적은 TDD를 설명하는 것이 아니라, BCB에서 UnitTest를 이용하는 것이기 때문에 자세한 설명은 미루겠습니다.

* 예제 프로젝트

TDD에만 집중할 수 있도록 되도록 간단한 예제를 만들어 보겠습니다. 원본은 C++ Builder Journal의 2002년 4월 내용 중 “Using Windows file mapping for inter-process communications by Mark G. Wiseman” 입니다. 본격적으로 진행하기에 앞서서 file mapping에 관한 설명이 있을 것입니다. 물론 이미 이 기사 내용에 대해서 아시는 분들은 **‘Server 데모 제작’** 으로 건너가셔도 됩니다.

집중할 수 있도록 간단하게 한다고 하지 않았느냐? 라고 물으실 분이 있을 것 같습니다. 간단하게 한다고 한 의미는 디자인 패턴, 리팩토링, STL, Template 등에 대한 설명을 자제하겠다는 것(사용하지 않겠다는 것은 아닙니다)입니다. 이에 대한 주제는 추후 진행할 기회가 있었으면 하는 개인적인 바람도 있습니다. 그리고 너무 간단한 예제면 같이 진행해봐도 맛이 없잖습니까? 맛을 더하기 위해 추후에 여러분이 해결해야 할 부분에 대해서도 언급이 있습니다.

---

[1] TDD(Test Driven Development) : http://xper.org/wiki/xp/TestDrivenDevelopment 참조

[2] 리펙토링(refactoring) : http://xper.org/wiki/xp/ReFactoring 참조

[3] 짝 프로그래밍(Pair Programming) : http://xper.org/wiki/xp/PairProgramming, http://www.pairprogramming.com/ 참조

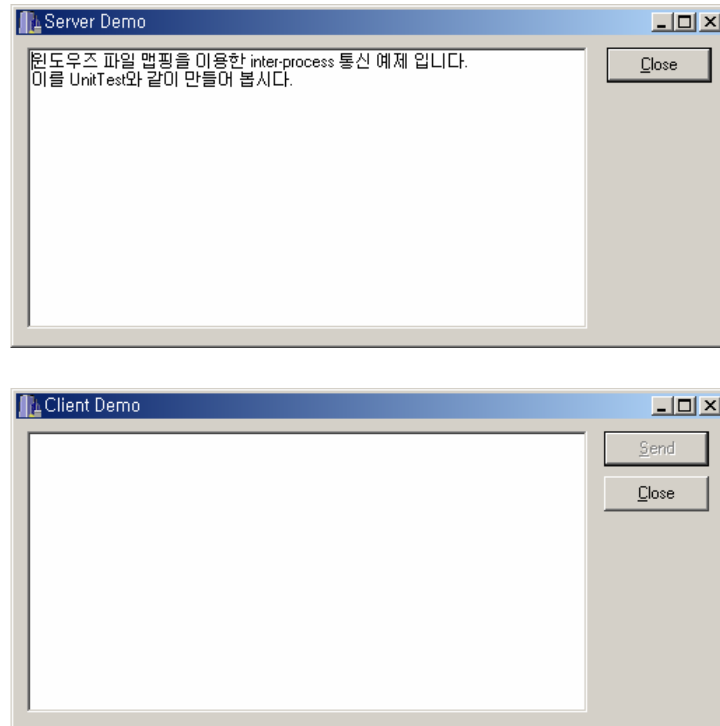

Fig. 1 예제 "Using Windows file mapping for inter-process communications"의 실행창

Fig.1 에서 보여지듯 Server와 Client 프로그램을 만들어서 inter-process 통신을 하는 데모입니다. 윈도우 운영체제를 사용하는 동일한 시스템에서 독립적인 프로세스 간의 데이터 공유를 Memory-Mapped File 방식으로 구현을 했습니다. 기사 내용에는 이러한 inter-process 통신을 위해 다른 방법을 언급하고 있습니다. 하지만 DDE(Dynamic Data Exchange)는 오래된 방법이고 사용하기도 어렵고, COM (Component Object Model)은 DDE 보다는 근래 방식이지만 사용하기가 더 어렵다 라고 합니다. 또한 File을 사용할 경우는 File Access에 많은 비용이 소요 될 겁니다. 그래서 고전전인 방법으로 파일을 사용해서 최대한 간단한 방법으로 inter-process를 구현하되, 성능 향상을 위해서 메모리에서 파일을 조작하는 방법을 사용하게 된 것입니다. 데모의 사용법은 아래와 같습니다.

1. Server.exe 실행
2. Client.exe 실행
3. Clinet 실행 폼에서 메모장에 타이핑한 후 Send를 선택
4. Server 에서 데이터를 가져오는지 확인

*** File Mapping**

윈도우에서 프로세스[4]나 어플리케이션[5]이 사용할 메모리 할당은 Win API인 GlobalAlloc()과 LocalAlloc() 만으로 이루어집니다. C++ 에서는 new operator를 사용함으로써 이를 사용할 수 있습니다.

또한 Win API에서 file mapping을 지원합니다. File mapping을 사용함으로써, 프로세스의 주소 영역 내의 memory block을 파일처럼 사용할 수 있습니다. 즉, 기존의 파일 조작하던 방법과 동일하게 read, write 등의 기능을 사용할 수 있다는 겁니다. 이러한 방법의 장점은 개발자는 file 조작에만 익숙해지는 것에만 신경을 쓰면 되고, file을 disk에 위치를 시킬 것인지, 메모리에서 사용할지는 필요에 따라 선택하더라도 비용이 거의 들지 않을 겁니다.

어쨌든 File mapping을 메모리에서 다루는 기법을 Memory-Mapped File 이라고 합니다. File mapping 생성을 위해 Win API 함수인 CreateFileMapping() 을 사용해야 합니다. 또한 생성된 파일 접근을 위하여 OpenFileMapping(), MapViewOfFile(),UnmapViewOfFile() 도 사용합니다. 자세한 것은 Win API 레퍼런스를 참조하시기 바랍니다.

```
HANDLE CreateFileMapping(
  HANDLE                  hFile,
  LPSECURITY_ATTRIBUTES   lpFileMappingAttributes,
  DWORD                   flProtect,
  DWORD                   dwMaximumSizeHigh,
  DWORD                   dwMaximumSizeLow,
  LPCTSTR                 lpName
);
```

마지막으로 진행할 예제 프로젝트의 문제점을 말씀 드리겠습니다. Client가 file의 내용을 업데이트 하고 Server에서 업데이트된 내용을 가져오는 일련의 과정에서 동기화 문제로 인해 Server 쪽에서 데이터를 잃을 수 있습니다. 즉, Server가 미처 읽어 오기 전에 Client가 다시 file에 작성을 하면 이전 데이터는 잃어버린다는 겁니다. 이에 대한 해결책은 몇가지가 있겠으나, 여기서는 다루지 않습니다. 기사의 내용에도 다음에 의논한다고 했는데, 저도 마찬가지입니다. 이것은 여러분의 몫으로 남겨 드리겠습니다.

[4] Process, Thread, Task 등 구분이 모호합니다만, 실행영역에 있는 단일 Thread라는 의미로 사용했습니다.

[5] 실행 entry point를 가진 process와 이 process가 control 할 수 있는 process 그룹이라는 의미로 사용했습니다.

*** Server 데모 제작**

Server 데모를 위해 아래와 같은 작업을 합시다.

> **[ToDo List]**
> ** Server 데모용 프로젝트 생성*
> **프로젝트에 UnitTest Framework 추가 하기*
> ** GUI 작성*
> ** 메모리에 파일 생성*
> ** 타이머를 이용한 파일에서 데이터 읽기*

Server 데모용 프로젝트 생성

비어있는 프로젝트를 생성합니다.

1. BCB6를 실행하시고, File >> New >> Application 을 선택합니다.
2. 프로젝트가 생성이 되면 저장을 합니다. Unit 이름은 'ServerMain' 으로 합니다.
3. 프로젝트 명은 "ServerApp"라고 해둡니다.
4. 이제 Run(F9)을 해봅니다. 무사히 실행이 된다면 ToDo 리스트에서 ~~***'Server 데모용 프로젝트 생성'***~~ 처럼 완료처리 합니다.

프로젝트에 UnitTest Framework 추가 하기

프로젝트에 UnitTest에 필요한 파일을 추가합니다.

1. Project >> Add to Project 를 선택합니다.
2. %CppUnit17BCB30Pro 설치폴더%/ E:\DEV_DATA2\CppUnit17BCB30Pro\borland\TestRunner\ 를 선택하고
3. GUITestResult.cpp ITestRunner.cpp ProgressBar.cpp TestRunner.cpp TestRunnerUI.cpp TestRunnerUnitFor.cpp TreeTestUnitForm.cpp 를 선택하고 추가합니다.
4. %CppUnit17BCB30Pro 설치폴더%/ E:\DEV_DATA2\CppUnit17BCB30Pro\borland\culib 를 선택하고
5. culib.lib을 추가합니다.

프로젝트의 Include path 설정을 합니다.

1. Project >> Options >> Directories/Conditionals Tab에서 Include path의 '…' 선택
2. CppUnit17BCB30Pro\borland\TestRunner 의 경로를 찾아서 Add 합니다.
3. \CppUnit17BCB30Pro\test\framework 의 경로를 찾아서 Add 합니다.

프로젝트의 Library path 설정을 합니다.

1. Project >> Options >> Directories/Conditionals Tab에서 Library path의 '…' 선택
2. CppUnit17BCB30Pro\borland\TestRunner 의 경로를 찾아서 Add 합니다.

이제 Run(F9)를 해봅니다. 무사히 실행이 된다면, 드디어 BCB 프로젝트에 UnitTest Framework를 추가한 것입니다. 이 정도면 자축해도 됩니다. 이런 일에 스스로 기뻐하지 않으면 일이 힘들어집니다. 누구는 춤을 추라고도 하는데, 어떤 방법으로든 기뻐하시길 바랍니다(이런 기쁨을 누가 알아주겠습니까? 관리자가? PM이?). ToDo 리스트에서 ***~~'프로젝트에 UnitTest Framework 추가 하기'~~*** 처럼 완료처리를 잊지 마시길 바랍니다.

GUI 작성

이제 BCB Edit 환경으로 돌아가서 익숙해져 있는 작업을 해봅시다. Fig.1 을 참조해서 배치 하시면 어려움이 없을 겁니다.

1. TForm1의 name을 Form1 → ServerForm 으로 수정합니다.
2. TServerForm의 Width, Hight를 각각 450, 250 정도로 합니다.
3. TMemo 추가합니다. name은 그대로 두고 Width, Hight를 각각 350, 210 정도로 합니다.
4. TButton 추가합니다. Name은 Button1 → CloseBtn 으로 수정합니다.
5. 적절하게 배치가 끝났다면, Run(F9)를 해봅니다. 무사히 실행 된다면, ToDo 리스트에서 ***~~'GUI 작성'~~*** 처럼 완료처리를 하시기 바랍니다.

메모리에 파일 생성

메모리에 파일 생성을 해봅니다. 이를 위해 File mapping을 위한 Class를 만들면 유용할 것 같습니다. 그렇다면? 예. 당연히 ToDo 리스트에 추가를 합니다.

[ToDo List]

** 메모리에 파일 생성*

** File mapping을 위한 kTMemoryMappedFileTest 클래스 생성*

** 타이머를 이용한 파일에서 데이터 읽기*

~~** Server 데모용 프로젝트 생성*~~

~~**프로젝트에 UnitTest Framework 추가 하기*~~

~~** GUI 작성*~~

File mapping을 위한 kTMemoryMappedFileTest 클래스 생성

kTMemoryMappedFileTest Class를 생성하고, TestCase의 상속을 받습니다. 명명은 알려진 표기법에 따라 다음과 같이 했습니다. 단 접두어로 k를 붙였습니다. 이러면 충돌 일어날 가능성이 좀더 줄어 들겠지요.

파일명 : kLikeThis
클래스명 : kTLikeThis
메쏘드명/맴버명 : likeThis
테스트 클래스명 : kTLikeThisTest
테스트 메쏘드명/맴버명 : testlikeThis

1. File >> New >> Unit 을 선택해서 Unit1.cpp를 생성합니다.
2. Unit1.cpp를 kMemoryMappedFileTest.cpp 로 저장합니다. 이런 방식으로 Unit을 생성하게 되면, BCB에서 .cpp와 .h 를 동시에 생성시켜주고, 다른 이름 저장 등의 기능에서 편리합니다.
3. kMemoryMappedFileTest.h 파일을 BCB에서 다음과 같이 편집합니다. 헤드 파일 추가에 주의하셨으면 합니다.

```
#ifndef kMemoryMappedFileTestH
#define kMemoryMappedFileTestH
#include "TestCase.h"
#include "TestSuite.h"
#include "TestCaller.h"

class kTMemoryMappedFileTest : public TestCase
{
public:
              kTMemoryMappedFileTest(std::string name);
  virtual      ~kTMemoryMappedFileTest ();
  void         setUp();
  void         tearDown();
protected:
};
#endif
```

4. 이제 kMemoryMappedFileTest.cpp 파일을 BCB에서 다음과 같이 편집합니다.

```
#pragma hdrstop
#include "kMemoryMappedFileTest.h"
#pragma package(smart_init)

kTMemoryMappedFileTest::kTMemoryMappedFileTest
(std::string name) : TestCase (name)
{
}
kTMemoryMappedFileTest::~kTMemoryMappedFileTest()
{
}
void kTMemoryMappedFileTest::setUp ()
{
}
void kTMemoryMappedFileTest::tearDown ()
{
}
```

kMemoryMappedFile Class를 생성했으니, 사용을 해야겠지요. 더미 테스트를 추가해 봅시다. kMemoryMappedFileTest 클래스에서 아래와 같이 Test Case를 작성합니다. TestCase가 되기를 원하는 Method를 생성하고 명명 시에 접두어로 test를 붙여줘야 합니다. 그리고 Test* 를 리턴하는 suite() 메쏘드를 작성하고 이미 생성한 TestCase를 addTest 메쏘드를 이용해서 인자로 넘겨주면 됩니다.

```
/*** kTMemoryMappedFileTest.h *********************/
class kTMemoryMappedFileTest : public TestCase
{
public:
             kTMemoryMappedFileTest(std::string name);
  virtual      ~kTMemoryMappedFileTest ();
  void         setUp();
  void         tearDown();
  static Test *suite();          //TestSuite
protected:
  void         testDummy();      //TestCase
};

/*** kTMemoryMappedFileTest.cpp *********************/
void kTMemoryMappedFileTest::testDummy()
{
  assertEquals(1,1);                    // 1 == 1 인지 테스트
}

Test *kTMemoryMappedFileTest::suite ()
{
  TestSuite *suite = new TestSuite ("kMemoryMappedFileTest");
  suite->addTest(new TestCaller<kTMemoryMappedFileTest> (
                  "testDummy",
                  &kTMemoryMappedFileTest::testDummy));
  return suite;
}
```

ServerMain.cpp로 돌아가서 프로그램 실행 시에 테스트를 실행하도록 설정합니다. 폼이

생성될 때 Test를 추가하면 됩니다. 추가하는 방법은 ITestRunner 의 객체를 생성하고 addTest() 메쏘드에 인자로 TestCase 객체의 suite()를 넘겨주면 됩니다.

```
/*** ServerMain.cpp *********************/
#include <vcl.h>
#pragma hdrstop
#include "ServerMain.h"
#include "ITestRunner.h"
#include "kMemoryMappedFileTest.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TServerForm *ServerForm;

_fastcall TServerForm::TServerForm(TComponent* Owner)
  : TForm(Owner)
{
}
//폼 생성시에 UnitTest 실행
void __fastcall TServerForm::FormCreate(TObject *Sender)
{
  ITestRunner runner;
  runner.addTest( kTMemoryMappedFileTest::suite() );
  runner.run();
}
```

여기까지 되었다면 Run을 해봅시다. 단 Run 전에 프로젝트 저장 폴더로 가셔서 빌드된 ServerApp.exe를 바탕화면이나 쉽게 접근할 수 있는 곳에 바로가기 아이콘을 만들어 둡니다. 아시겠지만, 실패하게 되면 BCB에서 자체 메시지를 띄우니까 불편하지만 따로 실행을 해야 됩니다. 추후에 BCB가 업그레이드가 되면서 UnitTest 가 툴에 통합되었으면 하는 바람이 이런 점 때문이기도 합니다.

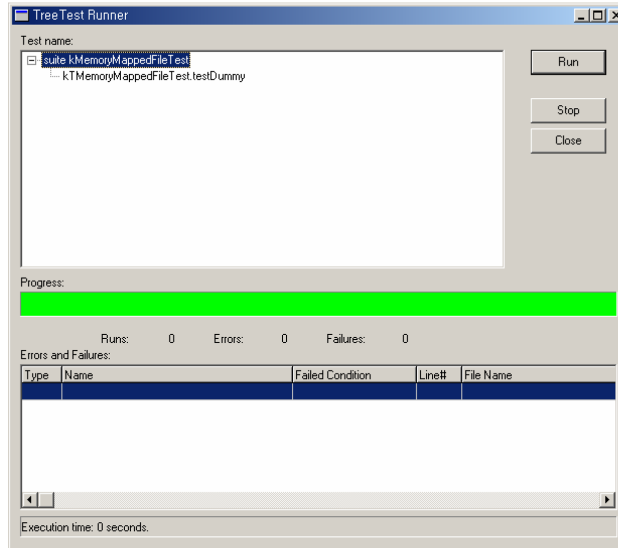

Fig. 2 Dummy Test Case 추가 후 실행

Test Suite 를 확장해보면 testDummy 가 추가 되어 있는 것을 볼 수 있습니다. 테스트는 성공하겠지요. 테스트를 실패하게 하려면 assert(1 == 2); 으로 해서 테스트 해보시면 됩니다. Failed Condition을 보시면 1 == 2 라고 표시될 것입니다.

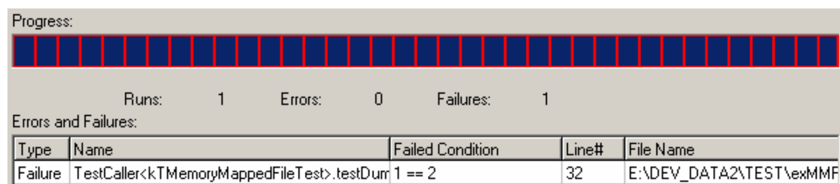

Fig. 3 Dummy Test Case 실패

.ToDo 리스트에서 '~~**File mapping을 위한 kTMemoryMappedFileTest 클래스 생성**~~' 처럼 완료처리를 하시기 바랍니다. 하지만 kTMemoryMappedFileTest를 다 작성한 것은 아니지요? 그러니 ToDo 리스트 중 '메모리에 파일 생성' 아래에 다음을 추가하기 바랍니다. **'메모리 파일 생성 루틴 작성'**.

ToDo 리스트는 프로젝트 진행 중에 지속적으로 업데이트 되어야 합니다. 프로젝트 시작 시에 할 일을 다 알면 좋겠지만, 현실적으로 불가능하고 없던 일이 생기기도 합니다. ToDo 리스트의 지속적인 관리는 지극히 단순한 일이지만 중요한 일이기도 합니다.

'메모리 파일 생성 루틴 작성'에 들어가기에 앞서서 잠시 모니터 위에 있는 모자를 머리에 올려 놓으시기 바랍니다. Refactor라는 이름의 모자 말입니다. 코딩 시에는 이 모자를 벗어서 모니터에 두고, Refactring 시에는 모자를 쓰셨으면 합니다(모자가 실제 존재하

는 것이면 더할 나위 없죠). 모자를 쓰지 않은 상태라면, 기능 구현에만 신경을 쓰기 바랍니다. 뒤를 생각해서 코드를 더 만들어 두지 마시고, 당장 쓰일 코드만 작성하길 권합니다. 모자를 쓰신 상태라면, 더 이상의 기능 구현은 하지 마시고, 이미 구현된 코드를 검토하고 업데이트하는 작업을 하셨으면 합니다. 익숙해지면 분 단위, 초 단위로 모자를 벗었다가 썼다가 할 것입니다만 아쉽게도 BCB 환경에서는 조금 더딜 수 밖에 없습니다. 하지만 충분한 속도이니까 너무 걱정 하지는 마시기 바랍니다. 자! 모자를 쓰시고 아래의 소스를 봅시다.

```
/*** ServerMain.cpp *********************/
void __fastcall TServerForm::FormCreate(TObject *Sender)
{
  ITestRunner runner;
  runner.addTest( kTMemoryMappedFileTest::suite() );
  runner.run();
}
```

FormCreate 이벤트 메쏘드에서 Test Suite를 추가하는 기능의 루틴이 있습니다. 주석이 없는 것이 좋은 코드입니다. 주석이 필요 없을 정도로 간략하다면 말이지요. Test suite 추가하는 기능을 ExtractMethod(메쏘드 추출) 하겠습니다. startTesting 메쏘드를 생성하고 기능 루틴을 옮깁니다.

```
/*** ServerMain.h *********************/
class TServerForm : public TForm
{
...(중략)
private:
  void __fastcall startTesting();

};

/*** ServerMain.cpp *********************/
void __fastcall TServerForm::FormCreate(TObject *Sender)
{
  startTesting();
}
void __fastcall TServerForm::startTesting()
```

```
{
  ITestRunner runner;
  runner.addTest( kTMemoryMappedFileTest::suite() );
  runner.run();
}
```

이제 바탕화면(그렇다고 가정하겠습니다)에 만들어 놓은 바로가기 아이콘을 선택해서 실행해 봅니다. 실행이 되나요? 그럼 Test Run을 해보시길 바랍니다. 정상적인 경우라면 assert(1 == 2); 를 수정을 안해놓았으니 Fail이 날 겁니다. 당장 assert(1 == 1); 로 업데이트하고 리빌드 합니다. 그리고 다시 바로가기 아이콘으로 실행해서 Test Run을 합니다. ExtractMethod 를 행하기 전과 동일하게 됨을 볼 수 있을 겁니다. 이제 여러분도 점점 TDD에 익숙해져 가는군요.

메모리 파일 생성 루틴 작성

실제 Memory-Madded File 의 클래스를 만들어 봅시다. 방법은 kMemoryMappedFileTest 만들 때와 거의 동일합니다만, 문서의 일관적인 스타일을 위해서 다시 한번 적습니다.

1. File >> New >> Unit 을 선택해서 Unit1.cpp를 생성합니다.
2. Unit1.cpp를 kMemoryMappedFile.cpp 로 저장합니다.
3. kMemoryMappedFile.h 파일을 BCB에서 다음과 같이 편집합니다.

```
/*** kMemoryMappedFile.h **********************/
#ifndef kMemoryMappedFileH
#define kMemoryMappedFileH

class kTMemoryMappedFile
{
public:
              kTMemoryMappedFile();
              ~kTMemoryMappedFile();
  void        setUp();
  void        tearDown();
  HANDLE      createFileMapping();
};
```

```
#endif

/*** kMemoryMappedFile.cpp *********************/
#pragma hdrstop
#include "kMemoryMappedFile.h"
#pragma package(smart_init)

kTMemoryMappedFile::kTMemoryMappedFile()
{
}
kTMemoryMappedFile::~kTMemoryMappedFile()
{
}
void kTMemoryMappedFile::setUp ()
{
}
void kTMemoryMappedFile::tearDown ()
{
}
```

createFileMapping() 메쏘드를 만들어 봅시다.

```
/*** kMemoryMappedFile.cpp *********************/
#include <Classes.hpp>
class kTMemoryMappedFile
{
... (중략)
  HANDLE         createFileMapping();
private:
  HANDLE         fmHandle;
};

/*** kMemoryMappedFile.cpp *********************/
kTMemoryMappedFile::kTMemoryMappedFile()
:fmHandle(0)
{
```

```
}... (중략)
HANDLE kTMemoryMappedFile::createFileMapping()
{
  fmHandle = CreateFileMapping( (HANDLE)0xFFFFFFFF, 0,
                                PAGE_READWRITE, 0, 1024,
                                "DemoFileMap"          );
  return fmHandle;
}
```

WinAPI CreateFileMapping()에 대해서는 BCB에서 도움말을 참조하셨으면 합니다. 다만, 몇가지만 설명을 드리겠습니다. hFile 인자에는 File Handle이 위치해야 합니다. 0xFFFFFFFF으로 표기하면 Window Page File 즉, 메모리를 사용하겠다는 의미가 됩니다. 이 외의 경우는 CreateFile()해서 얻은 Handle을 인자로 넘겨줘야 합니다. 그리고 lpName 인자에는 file mapping의 이름이 위치합니다. 이 정보를 다른 프로그램이 알아야 접근 가능해 집니다.

이 코드가 정상적으로 실행될지 자신이 없으시다면, Test Case를 작성합니다. CreateFileMapping()에서 실패한다면 NULL을 리턴할 것입니다.

1. kMemoryMappedFileTest.h 을 편집합니다.
   - Class Method를 먼저 선언하기 위해 testcreateFileMapping()를 추가 합니다.
2. kMemoryMappedFileTest.cpp 을 편집합니다.
   - kMemoryMappedFile.h을 include 시킵니다.
   - testcreateFileMapping() 을 작성합니다.
   - suite() 에 testcreateFileMapping()를 추가 시킵니다.

```
/*** kMemoryMappedFileTest.h *********************/
class kTMemoryMappedFileTest : public TestCase
{
...(중략)
protected:
  void         testDummy();
  void         testcreateFileMapping();
};

/*** kMemoryMappedFileTest.cpp *********************/
#include "kMemoryMappedFile.h"
```

```
...(중략)
void kTMemoryMappedFileTest::testcreateFileMapping()
{
  kTMemoryMappedFile* mmf = new kTMemoryMappedFile();
  HANDLE tmphdl = mmf->createFileMapping();

  assert( tmphdl != 0 );  // Test : createFileMapping Error

  delete mmf;
}
Test *kTMemoryMappedFileTest::suite ()
{
  TestSuite *suite = new TestSuite ("kMemoryMappedFileTest");

  suite->addTest(new TestCaller<kTMemoryMappedFileTest> (
                 "testDummy",
                 &kTMemoryMappedFileTest::testDummy));
  suite->addTest(new TestCaller<kTMemoryMappedFileTest> (
             "testcreateFileMapping",
              &kTMemoryMappedFileTest::testcreateFileMapping));

  return suite;
}
```

바로가기 아이콘으로 테스트를 해봅니다. Test Run을 하면 어떻게 될지 저로서는 알 수가 없습니다만 모두 통과 되었다고 생각하겠습니다. “assert(tmphdl != 0);” 에서 CreateFileMapping이 실패했다면, tmphdl은 0 값을 가지고 있을 것입니다. assert 결과 0 이 아니니 정상 호출 되었다는 의미가 됩니다. 이렇게 계속 테스트를 누적 시켜 나가야 합니다. 지금 테스트 통과 되었다고 testcreateFileMapping Test Case를 삭제해버리면, 개발 중 시스템 이상으로 인해 CreateFileMapping 이 실패했을 때, 이를 찾기 위해 한참을 고생할 수도 있습니다. 테스트를 간직하고 있다면 매번 검사해 주는 안전장치가 됩니다.

테스트가 되었으니, 모자를 바꿔 쓰고 추가된 기능의 루틴을 잘 살펴 봅니다. 어떤가요? 고쳤으면 하는 부분이 눈에 보이나요? 저는 kTMemoryMappedFile 기본 생성자가 마음에 걸리는 군요. CreateFileMapping API 함수에서 고유한 lpName을 인자로 넘기게 되어 있습니다. 이를 생성시에 파일명을 주게 하고 싶습니다. 타 프로세스에서 파일에 접근하기 위해서는 고유한 파일명을 알아야 하니까 이만한 가치는 있습니다.

```
kTMemoryMappedFile* mmf = new kTMemoryMappedFile("파일명");
```

모자를 쓰셨나요? 그럼 바꿔봅시다.

1. kMemoryMappedFile.h에서 kTMemoryMappedFile 클래스의 생성자 선언을 수정합니다.

```
#include <string>
...(중략)
public:
  kTMemoryMappedFile(std::string fname);              //modify
private:
  std::string       fileName;                          //add
```

2. kMemoryMappedFile.cpp에서 kTMemoryMappedFile 클래스의 생성자 메쏘드를 수정합니다.

```
kTMemoryMappedFile::kTMemoryMappedFile(std::string fname)
:fmHandle(0), fileName(fname)
{
}
```

3. kMemoryMappedFile.cpp에서 createFileMapping 메쏘드를 수정합니다.

```
HANDLE kTMemoryMappedFile::createFileMapping()
{
  fmHandle = CreateFileMapping( (HANDLE)0xFFFFFFFF, 0,
                                         PAGE_READWRITE, 0, 1024,
                                         fileName.c_str()         );
  return fmHandle;
}
```

기본 생성자를 두고 인자만 추가한 생성자를 만들 수도 있지만, kTMemoryMappedFile 객체 생성 시에 고유한 파일명을 꼭 넣도록 하기 위해서 기본 생성자를 놔두지 않았습니다. 또 createFileMapping() 메쏘드에서도 인자로 파일명을 받는 형태로 오버로딩할 수 있지만, kTMemoryMappedFile은 파일을 하나씩 다루는 클래스로 설계가 될 것이기 때문

에 생성 시 외에는 임의로 파일명을 주지 못하게 한 것입니다. 이제 테스트 쪽을 수정합니다.

1. kMemoryMappedFileTest.cpp에서 testcreateFileMapping 메쏘드를 수정합니다.

```
void kTMemoryMappedFileTest::testcreateFileMapping()
{
  kTMemoryMappedFile* mmf = new kTMemoryMappedFile("DemoFileMap");
  HANDLE tmphdl = mmf->createFileMapping();

  assert( tmphdl != 0 );
  assert( mmf->getFileName() ==
                           static_cast<std::string>("DemoFileMap") );
  delete mmf;
}
```

테스트를 위한 객체 생성 시 에 고유한 파일명을 인자로 넘겨주고, 생성 시에 의도되로 초기화 되었는지 확인 합니다. 테스트가 제대로 되었다면, 모자를 모니터 위에 올려두고 테스트 코드를 다음과 같이 구현 코드에 적용합니다.

```
HANDLE kTMemoryMappedFile::createFileMapping()
{
  HANDLE fmHandle = CreateFileMapping( (HANDLE)0xFFFFFFFF, 0,
                                     PAGE_READWRITE, 0, 1024,
                                     fileName.c_str()         );
  if (fmHandle != 0){
    throw("Unable to create File Mapping.");
  }
  return fmHandle;
}
```

클래스 종료 시에 해제할 것이 뭔가도 꼼꼼히 살펴야 겠습니다.

```
kTMemoryMappedFile::~kTMemoryMappedFile()
{
  if(fmHandle != 0)
```

```
    CloseHandle(fmHandle);
}
```

빌드로 실수를 하지 않았는지 확인을 합니다. 빌드가 되었다면, 코드 변경 내용 중에 컴파일러가 찾아내지 못한 문제가 있는지 테스트해 봅니다.

메모리 파일 접근 루틴 작성

ToDo 리스트 중 ~~'메모리 파일 생성 루틴 작성'~~ 완료 처리하시고, **'메모리 파일 접근 루틴 작성'** 을 추가 합시다. File 생성 후에 접근 목적의 Win API인 MapViewOfFile(), UnmapViewOfFile()을 랩핑(wrapping)할 것 입니다.

[ToDo List]

* *메모리에 파일 생성*

~~** File mapping을 위한 kTMemoryMappedFileTest 클래스 생성*~~

~~** 메모리 파일 생성 루틴 작성*~~

* *메모리 파일 접근 루틴 작성*

* *타이머를 이용한 파일에서 데이터 읽기*

~~** Server 데모용 프로젝트 생성*~~

~~**프로젝트에 UnitTest Framework 추가 하기*~~

~~** GUI 작성*~~

1. kMemoryMappedFile.cpp 파일을 BCB에서 다음과 같이 편집합니다.

```
/*** kMemoryMappedFile.h **********************/
class kTMemoryMappedFile
{
public:
...(중략)
  std::string   mapViewOfFile();
private:
...(중략)
```

```
  char*        memmap;
};

/*** kMemoryMappedFile.cpp *********************/
std::string kTMemoryMappedFile::mapViewOfFile()
{
  memmap = static_cast<char *>( MapViewOfFile(fmHandle,
                                    FILE_MAP_ALL_ACCESS, 0, 0, 0));
}
```

인자를 추가하여 완전한 랩핑을 할 수 도 있겠지만 지금 당장 할 필요는 없을 겁니다. 지금 필요하지 않는 코드를 미리미리 만들어 둘 정도의 여유는 없습니다. 눈 앞에 있는 것만 확실하게 테스트 하면서 넘어간다면, 언제든지 필요 시 추가 또는 변경 할 수 있을 겁니다. 혹시나 해서 만든 코드는 디버깅에 어려움만 더할 뿐이랍니다. 이것만 기억합시다. "당장 사용할 코드만 작성하고, 이 코드를 수정할 시에는 꼭 모자를 쓴다".

2. 테스트를 작성합니다.

```
/*** kMemoryMappedFileTest.h *********************/
class kTMemoryMappedFileTest : public TestCase
{
...(중략)
protected:
  void        testmapViewOfFile();
};

/*** kMemoryMappedFileTest.cpp *********************/
...(중략)
void kTMemoryMappedFileTest::testmapViewOfFile()
{
  kTMemoryMappedFile* mmf = new kTMemoryMappedFile("DemoFileMap");
  HANDLE tmphdl = mmf->createFileMapping();
  char *tmpmap  = mmf->mapViewOfFile();

  assert( tmpmap != 0 );
  delete mmf;
```

```
}
Test *kTMemoryMappedFileTest::suite ()
{
...(중략)
   suite->addTest(new TestCaller<kTMemoryMappedFileTest>(
                  "testmapViewOfFile",
                  &kTMemoryMappedFileTest::testmapViewOfFile));
   return suite;
}
```

테스트를 해봅시다. 통과가 된다면, fail이 발생하게도 해봅니다. 코드를 자세히 보면 테스트 코드에 중복된 코드가 있을 겁니다. 이것을 수정할 수도 있겠지만, 일단 넘어갑시다. 이 항목의 마지막으로 테스트를 소스에 적용 시킵니다. Memory Map을 초기화하는 함수도 같이 작성합니다.

```
/*** kMemoryMappedFile.cpp *********************/
char* kTMemoryMappedFile::mapViewOfFile()
{
  memmap = static_cast<char *>( MapViewOfFile(fmHandle,
                                    FILE_MAP_ALL_ACCESS, 0, 0, 0));
  if (memmap == 0){
    throw "Error! invalid file";
  }
  else{
    initMemoryMap();
  }
  return memmap;
}
void kTMemoryMappedFile::initMemoryMap()
{
  *memmap = 0;    //initialize
}
```

이 항목의 마지막으로 초기화가 제대로 되었는지 확인하는 테스트를 추가합니다.

```
void kTMemoryMappedFileTest::testmapViewOfFile()
```

```
{
 ...(중략)
  assert( *tmpmap == 0 );

  delete mmf;
}
```

타이머를 이용한 파일에서 테이터 일기

ToDo 리스트 중 ‘~~메모리 파일 접근 루틴 작성~~’ 을 완료 처리하시기 바랍니다. ToDo 리스트를 보니 하나가 남아있군요. 마저 처리합시다.

[ToDo List]

** 타이머를 이용한 파일에서 데이터 읽기*

~~** Server 데모용 프로젝트 생성*~~

~~**프로젝트에 UnitTest Framework 추가 하기*~~

~~** GUI 작성*~~

~~** 메모리에 파일 생성*~~

  ~~** File mapping을 위한 kTMemoryMappedFileTest 클래스 생성*~~

  ~~** 메모리 파일 생성 루틴 작성*~~

~~** 메모리 파일 접근 루틴 작성*~~

타이머를 이용해서 1초마다 파일을 읽어서 데이터를 메모장에 뿌리는 루틴입니다. 앞서서 얘기했지만, 권장할 만한 방법은 아닙니다. 하지만 이 문서의 특성상 충분하리라 봅니다.

1. TTimer를 메인폼에 추가 합니다.
2. TServerForm 소스를 다음과 같이 편집합니다.

```
/*** ServerMain.h ***********************/
#include "kMemoryMappedFile.h"
```

```
class TServerForm : public TForm
{
...(중략)
private:
  kTMemoryMappedFile* mmf;
  HANDLE mmfhdl;
  char *memmap;
};

/*** ServerMain.cpp *********************/
void __fastcall TServerForm::FormCreate(TObject *Sender)
{
  mmf            = new kTMemoryMappedFile("DemoMainFileMap");
  mmfhdl         = mmf->createFileMapping();
  memmap         = mmf->mapViewOfFile();

  startTesting();
}
void __fastcall TServerForm::Timer1Timer(TObject *Sender)
{
  if (*memmap != 0){
    Memo1->Lines->Text = memmap;
    *memmap = 0;
  }
}
void __fastcall TServerForm::FormClose(TObject *Sender,
      TCloseAction &Action)
{
  delete mmf;
}
void __fastcall TServerForm::CloseBtnClick(TObject *Sender)
{
  Close();
}
```

빌드하고 테스트 해봅니다. 추가적으로 테스트를 해보기를 원하는 부분이 있다면 추가

하셔도 됩니다. 문제가 없다면 ToDo 리스트 중 ~~'타이머를 이용한 파일에서 데이터 읽기'~~를 완료 처리하시기 바랍니다.

[ToDo List]

~~** Server 데모용 프로젝트 생성*~~

~~**프로젝트에 UnitTest Framework 추가 하기*~~

~~** GUI 작성*~~

~~** 메모리에 파일 생성*~~

~~** File mapping을 위한 kTMemoryMappedFileTest 클래스 생성*~~

~~** 메모리 파일 생성 루틴 작성*~~

~~** 메모리 파일 접근 루틴 작성*~~

~~** 타이머를 이용한 파일에서 데이터 읽기*~~

Server 프로그램의 ToDo 리스트를 모두 끝냈군요. 문서상으로 따라해보기가 만만치 않을 것 같다는 것이 마음에 걸립니다. 다음에 Clinet 프로그램을 같이 작성하도록 하겠습니다.

*** BCB에서 UnitTest 이용하기(2) 정리하는 말**

예제 프로젝트 중에 Server 프로그램을 UnitTest와 작성해 보았고, 결과적으로 kTMemoryMappedFile 클래스를 얻을 수 있었습니다. 이 클래스는 Client 프로그램 작성시에도 사용되어 질 것입니다. 물론 필요한 기능은 추가되거나 변경되겠지요. 이제 TDD에 조금 익숙해졌나요?

/**** 이 문서의 저작권은 볼랜드포럼에 있습니다.
by 김성진.kark **kark@borlandforum.com** ****/