
*** 수정 내용**

BCB에서 UnitTest 이용하기(2)에서 작성된 소스 중에 kTMemoryMappedFile 클래스 destructor를 다음과 같이 수정합니다.

```
/*** kTMemoryMappedFile.h **********************/
kTMemoryMappedFile::~kTMemoryMappedFile()
{
  if (memmap != 0)
    UnmapViewOfFile(memmap);

  if(fmHandle != 0)
    CloseHandle(fmHandle);
}

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

  startTesting();
}
```

BCB에서 UnitTest 이용하기(3)

BCB에서 UnitTest 이용하기가 마지막에 이르렀습니다. 앞서 작성되었던 Server 데모 프로그램과 연동해서 실행되는 Client 데모 프로그램 제작을 끝으로 마무리 하게 될 것입니다. 이 단계는 Server 데모 프로그램 작성과 거의 동일하기 때문에 소스만 올리고 여러분께 맡기려고 했으나, 마무리가 덜된 느낌을 받을 것 같아서 한번 더 반복하기로 했습니다. 따라서 이미 UnitTest 이용하기에 익숙해지신 분들은 소스만 추가하여, 전체 데모를 테스트 해보셔도 좋은 방법이 되겠습니다. 익숙하지 않으신 분들은 연습 겸 한번 더 따라 가보는 것을 권

장합니다.

*** Client 데모 제작**

Client 데모를 위해 아래와 같은 작업을 합시다.

[ToDo List]

**Client 데모용 프로젝트 생성*

**프로젝트에 UnitTest Framework 추가 하기*

** GUI 작성*

** 생성된 메모리 파일에 쓰기*

Client 데모용 프로젝트 생성

비어있는 프로젝트를 생성합니다.

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

프로젝트에 Unit Test 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.

경로 설정까지 완료가 되었다면, Run(F9)를 해봅니다. 어떤가요? 이제 UnitTest Framework을 추가하는 것이 낯설지가 않나요? 잠자기 전에 현관문이 잘 잠겼는지 확인하는 기분이 든다면 TDD에 대한 이해를 깊이 하신 것입니다. ToDo 리스트에서 ***~~'프로젝트에 UnitTest Framework 추가 하기'~~*** 처럼 완료처리 하시기 바랍니다.

GUI 작성

GUI 작성을 해봅시다. Server 데모 프로그램과 동일합니다. 다만 Send 용 버튼이 하나 추가되었을 뿐이랍니다.

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

생성된 메모리 파일에 쓰기

Server 데모 프로그램을 실행을 하면, 메모리에 파일을 생성할 것입니다. 그리고 생성된 파일을 1초 주기로 내용이 있는지 체크를 하고 있을 겁니다. 작성하고 있는 Client 데모 프로그램은 Server 데모 프로그램이 생성한 파일에 데이터를 작성하게 됩니다. 이를 위하여 확보되어 있는 kTMemoryMappedFile 클래스를 활용합시다.

작업을 시작하기 전에 ToDo 리스트를 점검해 봅시다. 생성된 메모리 파일에 쓰기 항목을 세분화 합니다. 먼저 메모리 파일을 읽어야 겠습니다. 즉 읽기 위해 포인트를 Open 해야 한다는 겁니다. 그리고 쓰면 되겠지요. 다음과 같이 추가하고 몇 가지 사전 준비를 해봅시다.

[ToDo List]

~~**Client 데모용 프로젝트 생성*~~

~~**프로젝트에 UnitTest Framework 추가 하기*~~

~~** GUI 작성*~~

** 생성된 메모리 파일에 쓰기*

** 메모리 파일 Open 루틴 작성*

** 메모리 파일 Write, Read 루틴 작성*

kTMemoryMappedFile 클래스를 프로젝트에 추가하기

1. Project >> Add to Project 를 선택합니다.
2. 만들어 놓은 kMemoryMappedFileTest.cpp, kMemoryMappedFile.cpp 를 추가합니다.

이제 kTMemoryMappedFile 클래스와 테스트 클래스가 추가가 되었으니 테스트 실행을 위해 ClientForm을 수정합니다. 한가지 언급을 하자면, 이미 완성되어 있는 클래스의 경우 테스트가 있다고 하더라도 굳이 테스트 클래스까지 추가할 필요는 없습니다. 다만 수정할 필요가 있을 때는 꼭 테스트 클래스를 추가하여서 테스트를 하면서 수정을 합니다. 이를 위해서 생성한 테스트 클래스를 삭제하지 않고 클래스와 붙여서 보관하는 요령이 필요할 것 입니다.

테스트 실행 코드 작성

1. ClientMain.h 에서 "kMemoryMappedFile.h" 를 사용한다고 선언해줍니다.
2. FormCreate 이벤트 메쏘드를 생성합니다.
3. startTesting 메쏘드를 작성합니다.
4. kMemoryMappedFile.cpp 에서 FormCreate 이벤트 메쏘드를 작성합니다.
4. startTesting 메쏘드도 작성합니다.

```
//ClientMain.h
#ifndef ClientMainH
#define ClientMainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "kMemoryMappedFile.h"
//---------------------------------------------------------------------------
class TClientForm : public TForm
{
__published:  // IDE-managed Components
  TMemo *Memo1;
  TButton *CloseBtn;
  TButton *SendBtn;
  void __fastcall FormCreate(TObject *Sender);
private:      // User declarations
public:              // User declarations
  __fastcall TClientForm(TComponent* Owner);
private:
  void __fastcall startTesting();
};
extern PACKAGE TClientForm *ClientForm;
#endif

//ClientMain.cpp
#include "ClientMain.h"
```

```
#include "ITestRunner.h"
#include "kMemoryMappedFileTest.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TClientForm *ClientForm;
//---------------------------------------------------------------------
__fastcall TClientForm::TClientForm(TComponent* Owner)
  : TForm(Owner)
{
}
void __fastcall TClientForm::FormCreate(TObject *Sender)
{
  mmf      = new kTMemoryMappedFile("DemoMainFileMap");
  mmfhdl    = mmf->openFileMapping();
  memmap    = mmf->mapViewOfFile();

  startTesting();
}
```

Run을 선택하고 Test가 제대로 실행되는지 확인해 봅시다. Server 데모 프로그램 작성 시에 사용했던 Test가 그대로 등록되어서 실행되는 것을 확인할 수 있을 겁니다. 클래스 생성 시 테스트와 같이 작성해 두면 두고두고 사용할 수 있습니다.

본격적으로 기능 추가를 해봅시다. 먼저 생성된 파일을 읽어 봅시다.

메모리 파일 Open 루틴 작성

메모리 파일을 Open 하는 기능을 kMemoryMappedFile 클래스에서 아직 가지고 있지 않습니다. 이를 위해 openFileMapping 메쏘드를 작성해 봅시다.

1. kMemoryMappedFile.h 에서 openFileMapping 메쏘드를 추가 선언합니다.

```
class kTMemoryMappedFile
{
public:
```

```
…(중략)
  HANDLE       openFileMapping();
};
```

2. kMemoryMappedFile.cpp 에서 openFileMapping 메쏘드 본문을 구현합니다.

```
HANDLE kTMemoryMappedFile::openFileMapping()
{
  fmHandle = OpenFileMapping(FILE_MAP_ALL_ACCESS,
                             false, fileName.c_str());
  return fmHandle;
}
```

3. kMemoryMappedFileTest.h 에서 testopenFileMapping 메쏘드를 추가 선언합니다.

```
class kTMemoryMappedFileTest : public TestCase
{
...(중략)
protected:
   void         testopenFileMapping();
};
```

4. kMemoryMappedFileTest.cpp 에서 testopenFileMapping 메쏘드 본문을 구현합니다.

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

  assert( tmphdl != 0 );
  assert( tmpopenhdl != 0 );

  delete mmf;
}
```

5. kMemoryMappedFileTest.cpp 에서 test suite에 testopenFileMapping 를 추가합니다.

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

Run을 해서 테스트 해봅니다. 잘 되나요? 그럼 테스트 코드를 kMemoryMappedFile 클래스에 적용하고, ToDo 리스트에서 ***'메모리 파일 Open 루틴 작성'*** 를 완료처리 하시기 바랍니다. 이제 메모리 파일 Write, Read 루틴을 작성 해봅시다.

```
HANDLE kTMemoryMappedFile::openFileMapping()
{
  fmHandle = OpenFileMapping(FILE_MAP_ALL_ACCESS,
                          false, fileName.c_str());
  if (fmHandle == 0){
    throw "Unable to open File Mapping.";
  }
  return fmHandle;
}
```

메모리 파일 Write, Read 루틴 작성

메모리 파일에 Write 루틴을 작성해 봅시다.

1. kMemoryMappedFile.h 에서 write 메쏘드를 추가 선언합니다.

```
class kTMemoryMappedFile
{
public:
…(중략)
```

```
    void        write(std::string src);
};
```

2. kMemoryMappedFile.cpp 에서 write 메쏘드 본문을 구현합니다.

```
void kTMemoryMappedFile::write(std::string src)
{
  if (fmHandle != 0){
    lstrcpy(memmap, static_cast<const char *>(src.c_str()));
  }
}
```

3. kMemoryMappedFileTest.h 에서 testwrite 메쏘드를 추가 선언합니다.

```
class kTMemoryMappedFileTest : public TestCase
{
...(중략)
protected:
    void testwrite();
};
```

4. kMemoryMappedFileTest.cpp 에서 testwrite 메쏘드 본문을 구현합니다.

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

 mmf->mapViewOfFile();
 mmf->write("is correct?");

  delete mmf;
}
```

5. kMemoryMappedFileTest.cpp 에서 test suite에 testwrite 를 추가합니다.

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

테스트 루틴까지 작성했는데 뭔가 이상한 느낌이 있으실 겁니다. 테스트 루틴에서 테스트 하는 기능이 빠져 있으니 허전할 수 밖에 없습니다. "`mmf->write("is correct?");`" 처럼 작성을 했으니 제대로 되었는지 확인을 해야하는 데 읽어올 수 있는 기능이 없습니다. Read 메쏘드를 작성을 하고 같이 테스트를 해야겠군요. 메모리 파일에 Read 루틴을 작성해 봅시다.

1. kMemoryMappedFile.h 에서 read 메쏘드를 추가 선언합니다.

```
class kTMemoryMappedFile
{
public:
…(중략)
    std::string   read();
};
```

2. kMemoryMappedFile.cpp 에서 read 메쏘드 본문을 구현합니다.

```
std::string kTMemoryMappedFile::read()
{
  if (fmHandle == 0)
    return 0;
  std::string returnString(memmap);
  return returnString;
}
```

3. kMemoryMappedFileTest.cpp 에서 write 메쏘드를 업데이트 합니다.

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

  mmf->mapViewOfFile();
  mmf->write("is correct?");
  std::string tmpread(mmf->read());
  assert( tmpread == static_cast<std::string>("is correct?") );

  delete mmf;
}
```

여기까지 작성을 했다면 Run을 하고 테스트를 수행합니다. 테스트가 성공한다면, 원하던 기능을 다 수행한 것입니다. 작성된 kMemoryMappedFile 클래스를 이용해서 Client 데모 프로그램을 작성하는 것만이 남았습니다. 마무리 하기 전에 모니터에 있는 모자를 사용해 봅시다. 구현만 하다 보니 지저분한 부분이 있을 수 있으니까요. 모자를 쓰고 보니 눈에 몇 가지가 있습니다. 이전에도 지적을 했지만, 테스트 루틴에서 보면 kMemoryMappedFile 객체를 계속 생성했다가 지우는 반복 루틴이 들어 있습니다. 깔끔하신 분들이 보면 굉장히 싫어하는 지저분한 코드입니다. 하지만 이것은 그대로 두겠습니다. 제가 깔끔한 성격도 아니고, 테스트 특성상 공용으로 사용되는 것이 많기 때문에 별도 처리를 하는 것이 관리하기가 편하다고 생각되어서 입니다.

또 한가지는 testwrite 작성 시에 read 기능이 필요해서 testwrite 구현을 미완성인체 두고 read 기능을 구현하러 갔다 온 적이 있다는 것을 기억하실 겁니다. 이 과정에서 기능은 잘 수행 되었지만, testwrite에는 결과적으로 write와 read 의 테스트를 겸하고 있습니다. 그래서 이 부분을 RENAME 을 시켜 봅니다. RENAME은 개발자가 일상적으로 사용하고 있습니다. 클래스명을 바꾼다든가, 메쏘드나 변수명을 바꾸는 것은 모두 포함 됩니다. Refactoring[1] 을 위해 RENAME을 한다는 것과 그냥 rename을 하는 것은 조금 차이가 있습니다. RENAME 이라는 패턴을 규정해 놓지 않았더라면, 개발자 사이에서 의사소통 시에 약간의 문제가 있을 수 있습니다. 또한 이를 툴에서 지원하기 위해서는 어느 정도 일관화된 유형이 필요했습니다. 따라서 이를 지원하는 툴에서 RENAME 기능을 선택하

[1] 도서 “Refactoring: Improving the Design of Existing Code by Martin Fowler 에서 정의한 패턴 유형을 의미합니다.

면 동일한 유형, 예를 들어 클래스 명을 RENAME 하면 클래스명을 모두 찾아 바꿔 줄 것입니다. 물론 수동으로도 얼마든지 가능합니다. 지금 그렇게 할 것 입니다.

1. kMemoryMappedFile.h 에서 testwrite 메쏘드를 testwriteopen 로 바꿉니다.

```
class kTMemoryMappedFileTest : public TestCase
{
...(중략)
protected:
    void testwriteopen();
};
```

2. kMemoryMappedFile.cpp 에서 testwrite 메쏘드를 testwriteopen 로 바꿉니다.

```
void kTMemoryMappedFileTest::testwriteopen()
```

3. Search → Find in Files 에서 Text to Find 에 'testwrite'를 입력하고 확인을 합니다.

4. 검색 결과창을 검토해서 testwrite라고 되어 있는 부분을 찾아가서 testwriteopen으로 교체합니다.

다 되셨으면 Run 하고 테스트를 수행합니다. 이제 Client 데모 프로그램을 마무리 해 봅시다.

ClientForm 작성

다음과 같은 순서로 합니다.

1. ClientMain.cpp 에서 FormCreate 이벤트를 수정하여 kTMemoryMappedFile 생성하는 루틴을 추가합니다.

```
void __fastcall TClientForm::FormCreate(TObject *Sender)
{
  mmf      = new kTMemoryMappedFile("DemoMainFileMap");
  mmfhdl   = mmf->openFileMapping();
```

```
  memmap    = mmf->mapViewOfFile();

  startTesting();
}
```

2. FormClose 이벤트도 작성합니다.

```
void __fastcall TClientForm::FormClose(TObject *Sender,
     TCloseAction &Action)
{
  delete mmf;
}
```

3. SendBtn 이벤트를 작성합니다.

```
void __fastcall TClientForm::SendBtnClick(TObject *Sender)
{
  mmf->write(Memo1->Text.c_str());
  Memo1->Clear();
}
```

저번 회에 작성한 Server 데모 프로그램을 띄워놓고, Run을 실행하여 테스트를 수행합니다. 테스트가 통과 되었으면, 테스트 폼은 닫고 Client 데모 프로그램에서 메모장에서 입력을 한 후에 Send 를 선택합니다. 어떤가요? Server 데모 프로그램에서 잘 받아 온다면 모든 프로그램 작성을 완료 한 것 입니다.

*** BCB에서 UnitTest 이용하기(3) 정리하는 말**

총 3회에 걸쳐서 BCB에서 UnitTest 이용하기를 적어봤습니다. 지금 당장은 적용해보시지 않더라도 시도 해볼 시기가 꼭 올 거라고 생각됩니다. BCB로 시도해볼 수 있는 영역은 많습니다. C++을 사용하기 때문입니다. C++ 은 1998년에 표준 스팩이 재정되었기 때문에 역사가 그리 긴 편은 아닙니다. 즉, 발전하고 있고 C++ 의 능력이 다 드러나지는 않았다는 겁니다.

그리고 C++ 언어의 영역을 넘어서는 시도도 많이 되고 있습니다. 똑 같은 찰흙이라도 초등학생이 주물럭거려서 만든 작품과 조각의 대가가 만지작거려 만든 작품은 분명히

다를 겁니다. 찰흙과 찰흙은 만지는 방법은 정해져 있더라도, 머리 속에는 다른 이미지를 가지고 있는 것입니다.

이는 개발 영역에도 적용됩니다. 어떤 언어를 사용하던지 환경이 제공하는 것은 기본이고, 어떻게 개발하느냐는 개발자에게 달려있습니다. 찰흙을 누구보다 빨리 만진다고 좋은 작품이 되는 것도 아니고, 실수 없이 만진다고 멋진 결과물이 나오는 것은 아닐 겁니다. 어떻게 생각하고 어떻게 표현할 것이냐가 중요합니다.

BCB 라는 멋진 도구를 가지고 무엇을 만들어 볼지 생각하며 시간을 투자해보는 것도 좋은 일일 것 같습니다.

아이러니하게도 개발환경이 갖추어지고, 안정적이 될수록 개발 자체에 대한 흥미는 잃게 됩니다. 원하던 기술을 얻기까지는 안절부절하고 갈망을 하며 매달리지만, 확보하고 나면 언제 그랬냐는 듯 시큰둥해집니다. 그럼에도 불구하고 여전히 기술확보에 매달립니다. 이런 삶을 지탱하기 위해서는 나름의 개발 철학이 필요할 것입니다.

by 김성진.kark kark@borlandforum.com ****/