TestDrive Tutorial

Questions : clonextop@gmail.com

본 튜토리얼은 TestDrive를 이용한 구현과정을 간략히 설명함으로써 그 동작방법을 이해하고 응용하는데 도움을 드리기 위해 작성되었습니다. 설명한 소스의 자료는 설치 폴더의 다음 경로에 존재합니다.
(설치경로)/example/simple/

1. 프로젝트 프로파일 생성

프로젝트 생성에는 별도의 GUI 환경을 제공하지 않고, TSL(TestDrive Scription Language)을 이용하여 Text 파일을 만들어 설계한다. 단 프로젝트는 profile의 확장자를 가져야 한다.

일정 폴더를 생성한 뒤, Simple.profile 이라는 텍스트 파일을 생성하고 아래와 같이 작성한다. 이처럼 여기서 TSL로 작성된 파일을 **프로파일**이라 정의하며, 특히 profile 확장자를 가는 프로파일을 **프로젝트 프로파일**이라 정의한다.

- Simple.profile
  system.title **"주 타이틀"**
  system.subtitle **"보조 타이틀"**

이를 저장하고 익스플로어 창에서 엔터를 입력하여 프로젝트 프로파일을 실행한다.

위 결과 화면과 같이 윈도우 타이틀이 TSL 명령에 따른 제목으로 변경되었다. 또한 profile의 실행 과정을 출력에서 그 과정을 적절히 알 수 있도록 표시된다. 그리고 현재 실행된 프로파일은 “홈(H)/프로파일 초기화” 버튼에 등록되어 언제라도 다시 재실행할 수 있다.

2. 서브 프로파일 생성

방금 작성한 Simple.profile을 아래의 굵은 글씨 내용을 추가하여 수정한다.

- Simple.profile

```
system.title        "주 타이틀"
system.subtitle     "보조 타이틀"

profile.tree{
    profile("Simple profile # 1", "test\\main.sp");
    profile("Simple profile # 2", "");
    profile("Simple profile # 3", "");
}

system.clear
```

이를 다시 실행한다.

몇 가지 차이점은 profile.tree 명령에 의해 프로파일 뷰에 몇가지 프로파일 리스트가 추가되었다. 그리고 출력에는 프로파일이 정상 동작하자 system.clear 명령으로 중간 과정들이 지워졌다. 프로파일 리스트 목록은 더블 클릭으로 프로젝트 프로파일과 동일한 문법으로 작성하여 지정된 프로파일을 실행할 수 있다.

서브 프로파일은 실행 방법이나 동작에서 프로젝트 프로파일과 차이점이 없으나 동일한 확장자로 지정하지 않는다. 그 이유는 익스플로어 상에서 엔터로 실행될 수 있는 프로파일을 제한하기 위해서이다.

이제 한번 더 Simple.profile을 아래의 굵은 글씨 내용으로 추가하여 수정한다

- Simple.profile
  system.title "주 타이틀"
  system.subtitle "보조 타이틀"

  profile.tree{
  profile("Simple profile # 1", "test\\main.sp");
  profile("Simple profile # 2", "");
  profile("Simple profile # 3", "");
  }

  system.clear

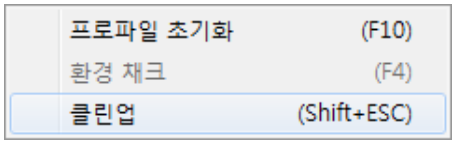
  profile.set.cleanup "cleanup.sp"

"Simple profile #1"을 더블 클릭하면 test\main.sp 위치의 파일이 실행된다.
추가된 Profile.set.cleanup은 "클린업" 버튼 메뉴를 활성화하고 눌렸을 때 실행할 프로파일을 지정한다.
그러나 두 프로파일 모두 현재는 존재하지 않으므로 지금부터 작성하도록 한다.

프로젝트 프로파일과 동일한 폴더 위치에 cleanup.sp 를 생성하고 아래와 같이 작성한다.

- cleanup.sp
  profile.tree{
  }

프로젝트 프로파일을 다시 실행하면 아래와 같이 클린업 메뉴가 활성화 된다.



이 버튼을 누르면 프로파일 메뉴가 모두 삭제된 것을 알 수 있으며, 프로파일 초기화 버튼을 누르면 다시 프로파일 리스트가 생성된다.
그 외의 버튼들도 아래와 같은 명령으로 특정 프로파일을 메뉴로 등록하여 버튼을 활성화하는 것이 가능하다.

profile.set.initialize : "프로파일 초기화" 등록
profile.set.check : "환경 채크" 등록
profile.set.cleanup : "클린업" 등록

다음은 test\main.sp 파일을 생성하기로 한다.

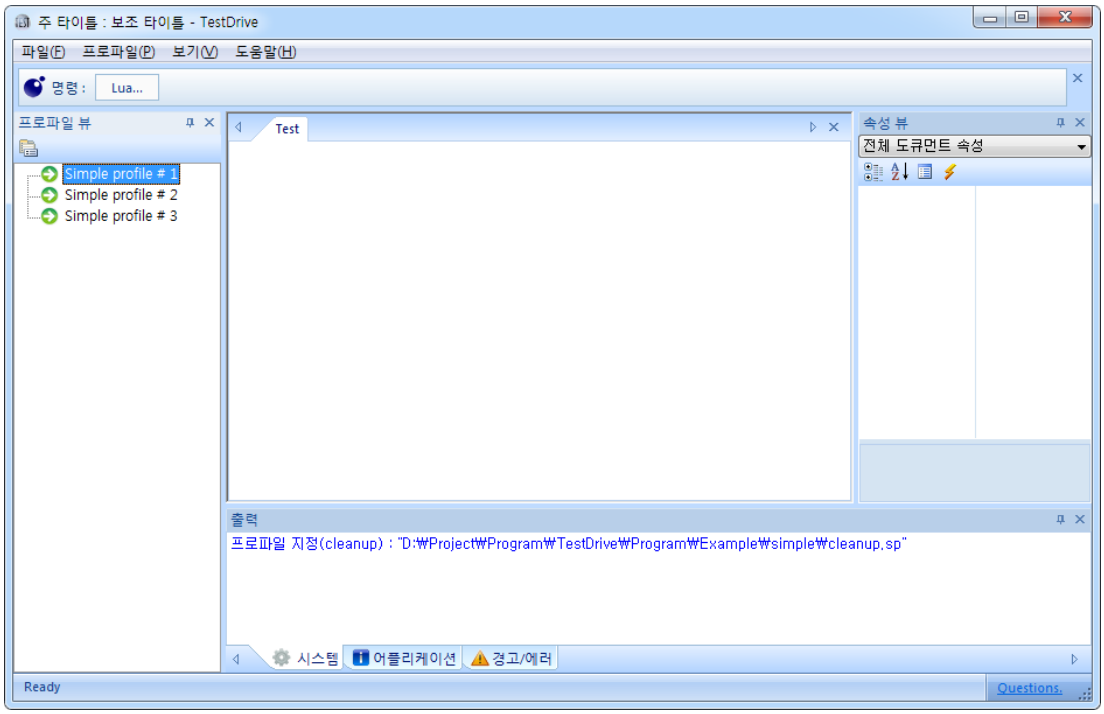
profile.tree 문법에서 “profile("Simple profile # 1", "test\\main.sp");” 명령에 의해 등록되었는데, 실행될 프로파일의 위치는 현재 작성하는 프로파일의 위치에서 상대적인 주소로 지정해야 한다. 절대주소로 지정하여도 관계없으나 프로젝트가 다른 PC로 옮기거나 하여 폴더 위치가 바뀌어도 정상적으로 실행되려면 파일 위치가 상대적이어야 한다. 또한 폴더 구분자인 ‘\’는 C문법과 같이 ‘\\’로 표시되어야 하는데, 이는 ‘\t’와 같은 특수 문자 표시에도 사용되기 때문에 이를 구별하기 위함이다.

우선 프로젝트 프로파일 위치에서 test 폴더를 생성하고 그곳에 아래와 같이 main.sp를 작성한다.

- main.sp

```
system.document.add "Test"{
    USE_EDIT_MODE
}
```

프로파일을 초기화 하고 “Simple profile #1”을 더블클릭하여 test/main.sp를 실행하면 아래와 같이 “Test”라는 화면을 볼 수 있다.



여기서 Test 화면과 그 영역을 **도큐먼트**라 정의한다. 이 도큐먼트에는 각종 GUI 환경과 실제 구현을 적용할 수 있다. 이를 **도큐먼트 레벨 구현**이라 정의하며, C/C++로 작성된 DLL 파일에 의해 동작되며, GUI 환경은 TSL 문법에 의해 구축된다.

프로파일, 도큐먼트 구현과 같은 의미 부여가 생소하게 느껴질 수 있으나 이는 TestDrive의 구조를 이해하는데 필수적이다.

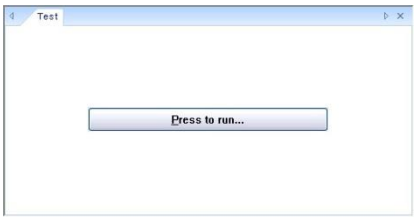
3. 도큐먼트 GUI 구현

앞 단락에서 작성한 main.sp를 아래와 같이 수정한다.

- main.sp

```
system.document.add "Test"{
        USE_EDIT_MODE
        button("run", 0,0, 300, 30){
                SetText("&Press to run...");
        }
}
```

여기서 한가지 재미있는 것은 텍스트 에디터에서 위의 내용을 추가하여 저장하는 순간 TestDrive는 'USE_EDIT_MODE'로 인해 main.sp가 다시 실행하여 아래와 같이 도큐먼트 영역이 업데이트 되는 것을 알 수 있다.



이것은 TSL을 텍스트 에디터로 작성하면서 GUI 구성을 실시간으로 확인하는데 도움을 준다. 'USE_EDIT_MODE'는 GUI 작성이 완료되면 제거해도 된다.

내친 김에 아래처럼 GUI 객체 하나 더 추가해 본다.

- main.sp

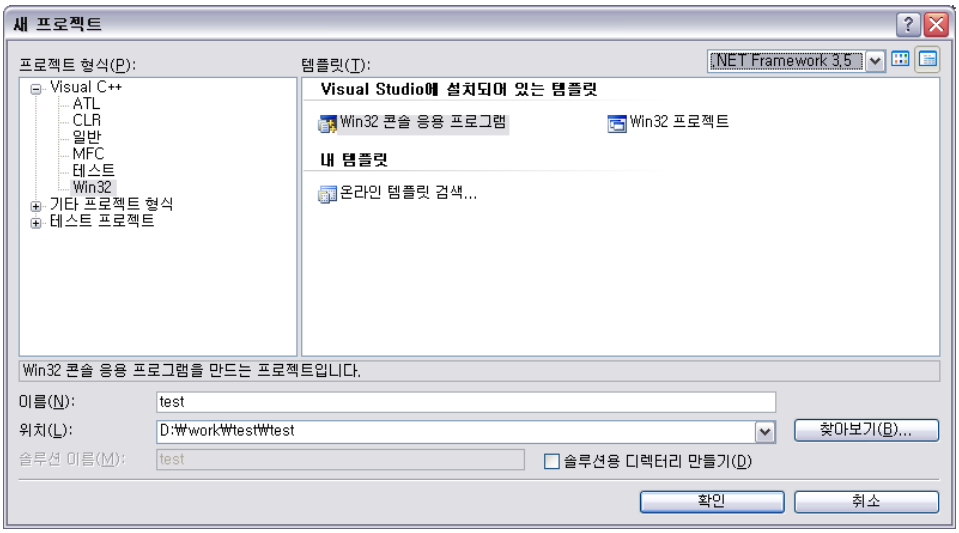
```
system.document.add "Test"{
        USE_EDIT_MODE
        button("run", 0,0, 300, 30){
                SetText("&Press to run...");
        }
        screen("ABC", 0, 40, 500, 300){
                Create(640,480, RGBA_8888);
        }
}
```

아래는 그 실행화면이다.

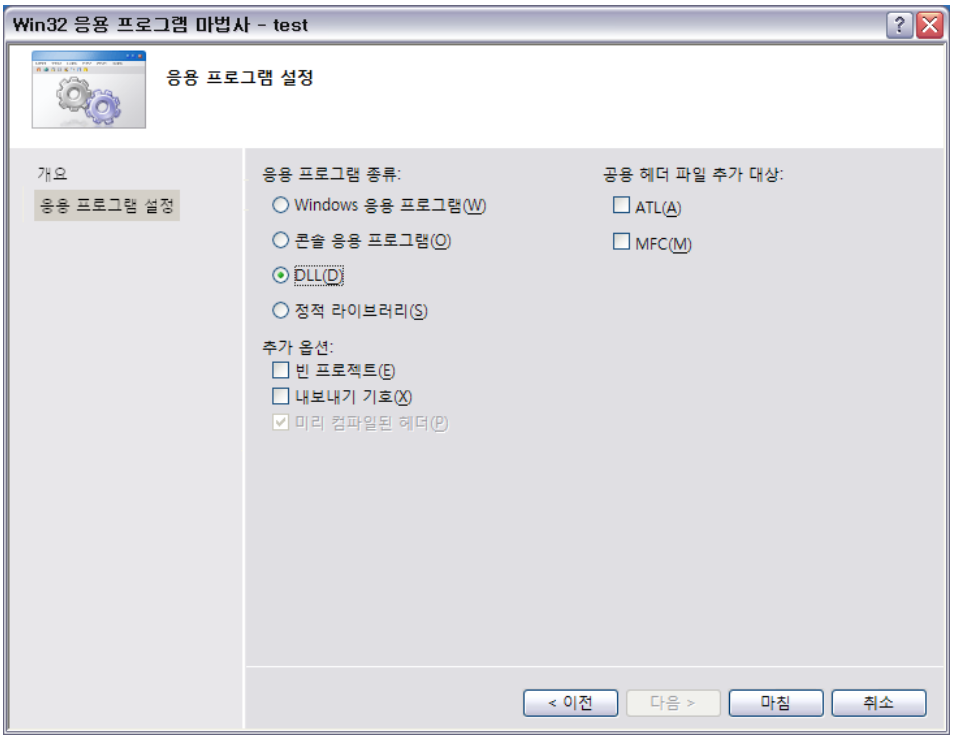
위의 TSL 문법에 의해 도큐먼트에 버튼과 스크린 버퍼영역을 생성했으나, 현재는 눌러도 아무런 반응이 없다. 그것은 **도큐먼트 레벨 구현**이 현재 존재하지 않기 때문이다.

4. 도큐먼트 레벨 구현

설계한 GUI에 대한 반응을 위한 레벨 구현을 하기 위해 다음과 같이 VisualStudio 프로젝트를 생성하도록 한다.

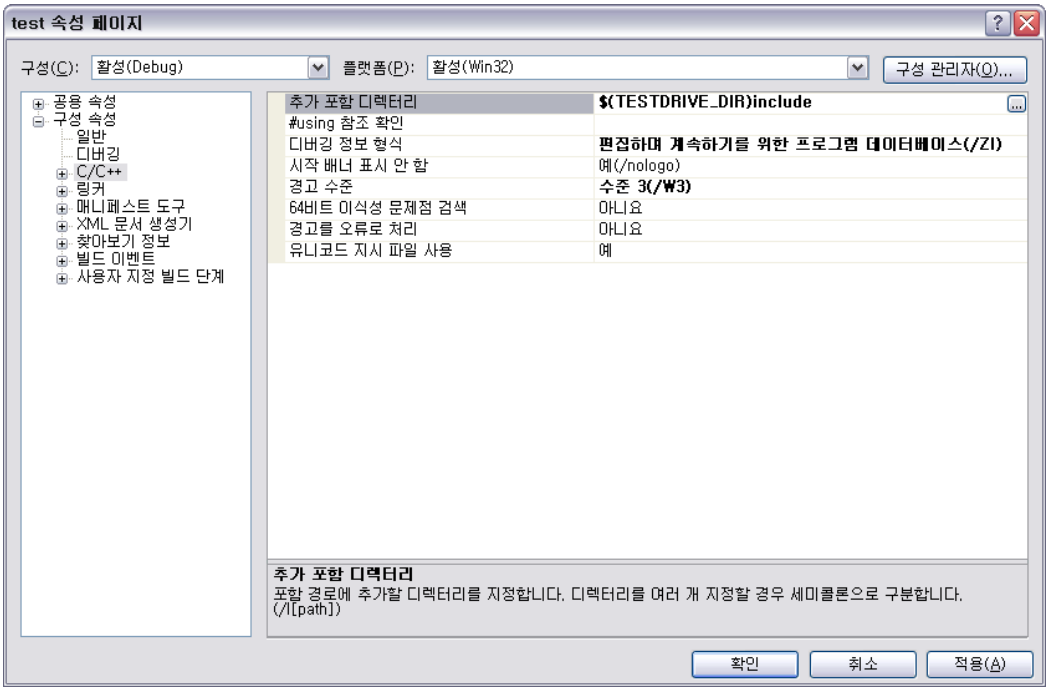


DLL 을 작성하기 위해서는 Win32 프로젝트로 작성해야 하며 '확인'버튼을 누른 후 아래와 같이 응용 프로그램 설정에서 DLL 프로젝트로 생성해야 한다.



프로젝트 생성을 마치면 컴파일 하면 “F7”을 누르게 되면 정상적으로 debug 폴더에 DLL이 생성되는 것을 볼 수 있다.

이 DLL에 TestDrive가 제공하는 TestDrive.h를 포함시킬 수 있어야 하는데 다음과 같은 과정을 거친다.



솔루션탐색기에서 굵은글씨 “**test**” 에 오른쪽 버튼을 눌러 “속성(R)” 클릭하여 위와 같이 추가 포함 디렉터리에 “$(TESTDRIVE_DIR)include”를 입력한다.

그리고 컴파일이 정상적으로 수행된 결과 dll 파일이 하위 폴더에 자동 복사되도록 빌드 이벤트의 ‘빌드 후 이벤트’의 명령 줄을 다음과 같이 지정한다.

명령줄 : copy $(TargetPath) ..\

확인을 누르고, stdafx.h 파일의 마지막에 아래와 같은 내용을 추가함으로써 기본적인 프로젝트 준비를 마친다.

- stdafx.h (마지막 줄에 첨부…)

```
// TODO: 프로그램에필요한추가헤더는여기에서참조합니다.
#include "TestDrive.h"
```

TestDrive를 정상적으로 설치를 하였다면, 빌드를 하였을 때 정상적으로 컴파일이 완료될 것이고 실패하였을 경우 TestDrive를 설치하고 재부팅을 하여 다시 확인한다. 이 과정이 완료되지 않으면 다음 과정을 진행할 수 없다.

DLL이 생성되었으니 main.sp를 아래와 같이 수정한다.

- main.sp

```
system.document.add "Test"{
        USE_EDIT_MODE
        button("run", 0,0, 300, 30){
                SetText("&Press to run...");
        }
        screen("ABC", 0, 40, 500, 300){
                Create(640,480, RGBA_8888);
        }
        SetProgram("test.dll");
}
```

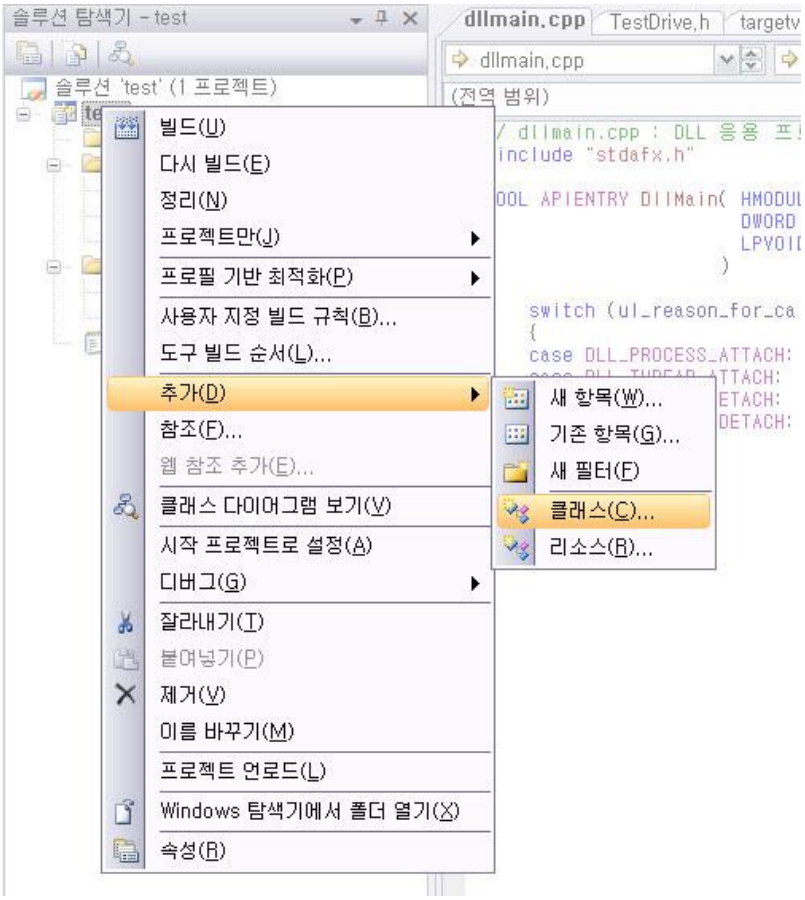
그러나 저장하고 메시지를 확인하면 다음과 같은 에러를 발생한다.

이것은 DLL에 아직 우리가 구현 내용을 만들지 않았기 때문이다.
TestDrive는 DLL을 참조할 때 RegisterDocument 함수를 호출하여 현재 **도큐먼트 레벨 구현 인터페이스**를 요청한다.
TestDrive.h에는 아래와 같이 RegisterDocument의 함수가 정의되어 있다.

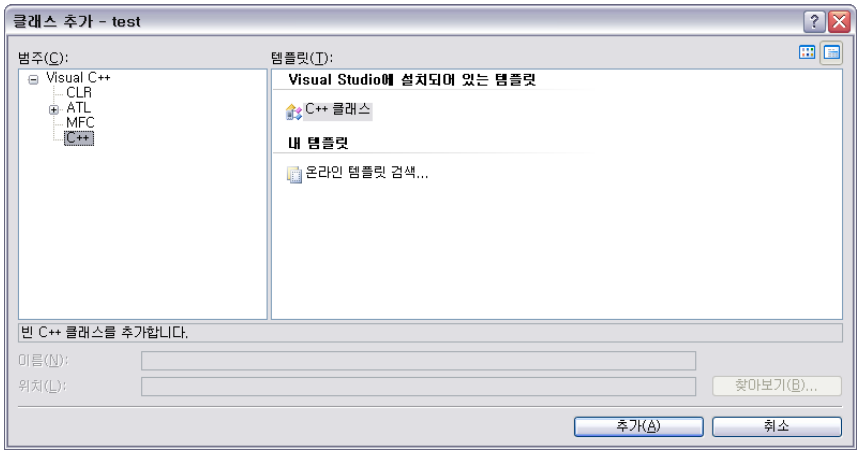
```
__declspec(dllexport)  ITDImplDocument*  __cdecl
RegisterDocument(ITDDocument* pDoc);
```

여기서 pDoc은 도큐먼트 환경을 제어할 수 있는 인터페이스이며, 해당 도큐먼트는 ITDImplDocument 클래스를 상속받아 생성되어야 하고 pDoc 인터페이스를 통해 GUI 객체를 제어할 수 있다.
C++에 대한 개념이 미흡하더라도 튜토리얼을 에서 쉽게 설명하고 있으니 끝까지 진행하도록 한다.

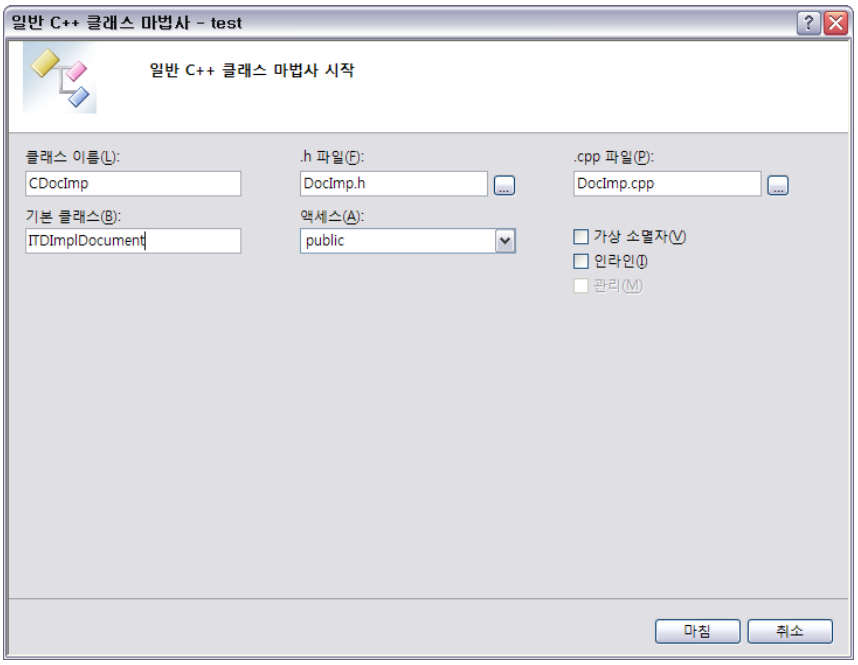
우선 도큐먼트 레벨 구현 인터페이스를 제공하기 위한 클래스를 생성해 본다.



클래스 생성 방법은 위와 같이 솔루션 탐색기의 'test'의 오른쪽 버튼 메뉴에 의해 생성할 수 있다.



추가할 때에는 일반 C++로 생성하고 아래와 같이 클래스 설정 후 마침을 클릭한다.



클래스 이름 : CDocImp
기본 클래스 : ITDImplDocument

우선 구현이 정상적으로 TestDrive에 적용되었는지 확인하기 위해 생성자와 소멸자가 호출될 때 메시지를 출력창에 표시하도록 해 보자.

먼저 DocImp.h 파일의 CDocImp 클래스를 아래와 같이 수정한다.

- DocImp.h

```
class CDocImp : public ITDImplDocument
{
public:
    CDocImp(ITDDocument* pDoc);
    ~ CDocImp(void);

    ITDDocument* m_pDoc;
};
```

DocImp.cpp 파일의 생성자와 소멸자를 아래와 같이 수정한다.

- DocImp.cpp

```
CDocImp::CDocImp(ITDDocument* pDoc){
    m_pDoc = pDoc;
    m_pDoc->GetSystem()->LogOut(_T("CDocImp이생성되었습니다."));
}

CDocImp::~CDocImp(void){
    m_pDoc-> GetSystem()->LogOut(_T("CDocImp이해제되었습니다."));
}
```

dllmain.cpp 파일에 다음과 같은 내용을 추가하여 TestDrive가 CDocImp

인터페이스를 도큐먼트 레벨 구현으로 사용할 수 있도록 한다.

- dllmain.cpp (마지막 줄에 추가...)

```
#include "DocImp.h"
ITDImplDocument* __cdecl RegisterDocument(ITDDocument* pDoc){
    return new CDocImp(pDoc);
}
```

이렇게 수정을 완료한 후 컴파일(F7)하여 DLL 파일을 생성한다.
이제 main.sp 가 실행하면 "Test" 도큐먼트가 생성되어 생성자의 메시지를 출력하고 X 버튼을 눌러 도큐먼트를 닫으면 소멸자의 메시지가 호출됨을 알 수 있다.

이제 버튼을 눌렀을 때 메시지 출력을 하는 예를 보인다.
TestDrive의 모든 GUI 객체는 문자열 이름으로 구별할 수 있으며, 각각 ID 번호를 부여할 수 있다.
버튼의 문자열 이름은 앞의 main.sp에서 "button("run", 0,0, 300, 30)" 명령에 의해 "run" 이라는 이름을 가진다. 그리고 모든 GUI 객체의 ID는 DWORD(-1)=0xFFFFFFFF로 초기화 되어 있다.
테스트 드라이브는 객체를 클릭할 때 구현 인터페이스의 OnButtonClick 멤버함수를 호출한다. 단 객체의 ID가 -1 일 때에는 호출되지 않으므로 버튼의 ID 부여와 OnButtonClick 멤버함수 구현이 모두 이루어져야 한다.
먼저 버튼에 ID를 부여하기 위해 생성자를 아래와 같이 수정한다.

- DocImp.cpp (CDocImp::CDocImp 수정...)

```
CDocImp::CDocImp(ITDDocument* pDoc){
    m_pDoc = pDoc;
    m_pDoc->GetSystem()->LogOut(_T("CDocImp이생성되었습니다."));
    m_pDoc->GetButton(_T("run"))->SetManager(this, 0); // ID 0번 부여
}
```

위의 SetManager 를 통해 버튼의 메니지먼트를 this(CDocImp) 클레스에서 가능하지만 버튼의 구현이 분리되어 있기 때문에 다음과 같이 수정되어야 한다.

- DocImp.h (CDocImp에 추가...)

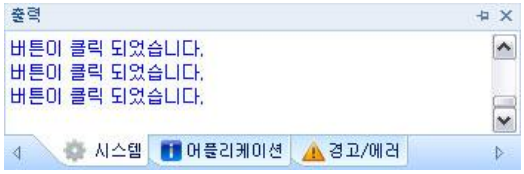
```
class CDocImp : public ITDImplDocument,
                public ITDButtonManager
{
public:
```

```
        ……(생략)
        STDMETHOD_(void, OnButtonClick)(DWORD dwID);
    };
```

- DocImp.cpp (구현 내용 추가…)

```
    void CDocImp::OnButtonClick(DWORD dwID){
        m_pDoc->GetSystem()->LogOut(_T("버튼이 클릭 되었습니다."));
    }
```

위와 같이 수정하면 버튼을 누를 때 마다 아래와 같은 메시지가 출력됨을 알 수 있다.
이 내용은 “run” 이라는 버튼을 눌렸을 때 메니지먼트로 지정된 클래스의 OnButtonClick을 호출한다. 이 때 dwID 인수는 SetManager를 통해 설정한 0 이 입력된다. 이 말은 여러 버튼 객체가 동일한 메니지먼트 클래스를 공유할 수 있다는 뜻이 된다.



다음은 버튼이 눌렸을 때 스크린 버퍼에 무언가 출력하는 예제를 구현한다.
위 스크린 GUI 객체는 main.sp에서 “screen("ABC", 0, 40, 500, 300)” 명령으로 생성함에 따라 이름은 “ABC”로 부여 받았으며, “Create(640,480, RGBA_8888);”에 따라 32 비트 RGBA 색상의 640x480 해상도를 가지는 버퍼이다.

따라서 이 객체의 인터페이스를 생성자에서 “ABC”로 검색해 멤버변수로 저장하고 이 인터페이스를 활용해 OnButtonClick 함수에서 버퍼를 그리는 과정을 보인다.

우선 인터페이스를 담을 멤버변수를 추가한다.

- DocImp.h (CDocImp에 추가…)

```
    class CDocImp : public ITDImplDocument,
                    public ITDButtonManager
    {
    public:
        ……(생략)
        ITDBuffer*    m_pBuffer;
    };
```

그리고 생성자에 버퍼 인터페이스를 검색하여 m_pBuffer에 넣는다.

- DocImp.cpp (CDocImp::CDocImp에 추가…)

```
CDocImp::CDocImp(ITDDocument* pDoc){
    ……(생략)
    m_pBuffer     = m_pDoc->GetBuffer(_T ("ABC"));
}
```

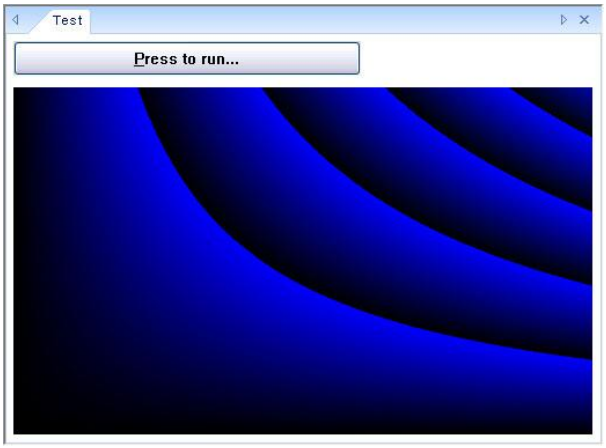
그리고 OnClick에 원하는 버퍼조작을 구현한다. 여기서는 간단한 예제를 보이기 위해 간단한 방법으로 연산을 수행한다.

- DocImp.cpp (구현 내용 추가…)

```
void CDocImp::OnButtonClick(DWORD dwID){
    ……(생략)
    DWORD* pB = (DWORD*)m_pBuffer-> GetPointer(); // 버퍼 포인터 얻기

    for(int y=0;y<480;y++){
        for(int x=0;x<640;x++){
            pB[x + y * 640] = x*y;
        }
    }
    m_pBuffer->Present();       // 그린 결과를 화면에 바로 보인다.
}
```

이제 버튼을 누르면 아래와 같이 출력됨을 알 수 있다.

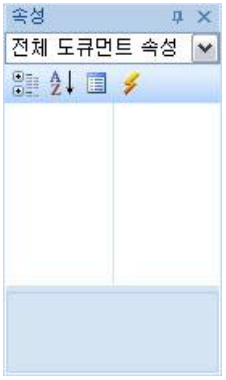


주의할 것은 버퍼는 윈도우의 DIB 형식을 따른다. 따라서 메모리상 첫 번째 위치 색상은 왼쪽 맨 아래가 된다. 따라서 메모리 포인터를 가지고 직접 접근하려면 이에 유의하고 그렇지 않으려면 HDC 인터페이스나 별도로 제공하는 함수를 사용한다.

버퍼에는 다양한 활용법(다른 버퍼와 컨트롤 연동, 동영상 제작, 파일 저장/로딩 등)을 제공하므로 상세한 내용은 TestDrive.h의 선언을 살펴본다.

5. 도큐먼트 속성 제어

도큐먼트가 사용자에 의해 실행되고 관리되려면 단순히 명령뿐 아니라 변수들을 상세하게 제어할 수 있어야 한다. 이를 위해 TestDrive는 아래와 같은 속성창이 존재하고 도큐먼트 레벨 구현에서 다양한 형식의 변수들을 등록할 수 있다.



이를 위해 다음과 같이 버퍼 연산하는 내용의 일부를 수정한다.

- DocImp.cpp (OnButtonClick 구현 내용 수정...)

```
void CDocImp::OnButtonClick(DOWRD dwID){
    ……(생략)
    for(int y=0;y<480;y++){
        for(int x=0;x<640;x++){
            pB[x + y * 640] = x*y*m_Const;
        }
    }
    ……
}
```

여기서 m_Const 라는 변수를 선언하고, 사용자가 제어가능 하도록 도큐먼트 속성에 추가하려고 한다.

먼저 다음과 같이 m_Const 변수를 만든다.

- DocImp.h (CDocImp에 추가...)

```
class CDocImp : public ITDImplDocument,
                public ITDButtonManager
{
public:
    ……(생략)
    int    m_Const;
};
```

그리고 생성자에서 이 변수를 속성목록에 추가한다.

- DocImp.cpp (CDocImp::CDocImp 수정...)

```
CDocImp::CDocImp(ITDDocument* pDoc){
```

```
        ……(생략)
        m_Const = 1;
        pDoc->AddPropertyData(PROPERTY_TYPE_INT, 1, _T("상수"),
                              (DWORD_PTR)&m_Const, _T("상수 값을
                변경합니다."));
    }
```

이제 “Test” 도큐먼트에 대한 변경 가능한 속성이 추가 되었음을 알 수 있다.

이 값을 변경하고 “run” 버튼 객체를 클릭하여 보면 아래와 같이 변경된 속성이 적용된 결과를 확인할 수 있다.

m_Const = 1 m_Const = 5 m_Const = 20

사용자에 의해 속성이 변경 가능하지만 입력 값을 필요에 따라 제한을 둘 필요가 있다. 이 때에는 OnPropertUpdate 함수를 구현함으로써 가능하다. TestDrive는 속성의 변경이 있을 때 이 함수를 호출하여 FALSE를 리턴하면 TestDrive가 자동으로 업데이트하고 TRUE를 입력하면 도큐먼트 레벨에서 처리됨으로 인식하고 TestDrive는 아무런 작업도 수행하지 않는다.

예로 m_Const 입력을 1~20 사이로 제한하도록 한다.

먼저 함수 원형을 추가한다. 이 함수들은 ITDImplDocument에서 상속받은 것이므로 TestDrive.h를 참조하면 이외에 어떤 것들이 있는지 살펴볼 수 있다.

- DocImp.h (CDocImp에 추가…)

```
class CDocImp : public ITDImplDocument,
                public ITDButtonManager
{
public:
    ……(생략)
```

```
    STDMETHOD_(BOOL, OnPropertyUpdate)(ITDPropertyData* pProperty);
};
```

OnPropertyUpdate를 구현한다.

- DocImp.cpp (구현 내용 추가...)

```
BOOL CDocImp:: OnPropertyUpdate(ITDPropertyData* pProperty){
    // 생성자에서 속성 ID를 1로 지정했었다.
    if(pProperty->GetID()!=1) return FALSE;
    int data = *(int*)(pProperty->GetData());  // 새로운 값을 입력
받는다.
    // 값을 1~20으로 제한
    if(data < 1) data =1;
    else if(data >20) data = 20;
    *(int*)(pProperty-> GetData()) = data;
    // 수정된 값을 업데이트 한다
    pProperty->UpdateData(FALSE);
    return TRUE;
}
```

이제 속성을 1~20 범위 이외의 값을 입력하면 자동으로 범위 안으로 변경된다.

그 외 AddPropertData 함수로 속성을 등록할 때 반환하는 ITDPropertyData 인터페이스를 활용하면 옵션을 부여하거나 비활성화 하는 등의 여러 작업이 가능하다.

모든 구조들, 그리고 이 구조들을 연결시키는 법칙들은
수학적으로 가장 단순한 개념들을 구한 다음
그것들을 연결시키는 원리에 따라 구해질 수 있다.
(All these constructions and the laws connecting them
can be arrived at by the principle of looking for the mathematically
simplest concepts and the link between them.)

알베르트 아인슈타인 (Albert Einstein, 1879.3.14 ~ 1955.4.18)