DCL Runtime Debugging Support

2005.06.08

김대중<daejung@sysdeveloper.net>
http://www.sysdeveloper.net/daejung

요약

버그 없는 소프트웨어를 개발하는 것은 결코 쉬운 일이 아니다. 대부분의 상업적 개발 도구들이 소스 프로그램을 추적할 수 있도록 하는 디버깅 환경을 제공하고 있고 이러한 것들은 단위 함수들을 디버깅 하는데 있어서 매우 유용한건 사실이다. 그러나, 시스템 전반에 걸친 문맥(context)오류를 발견하기 위해서는 별로 도움이 되지 못한다. 특히, 메모리 누출과 같은 버그들은 기존의 소프트웨어 개발 도구를 사용하여 발견해 내는 것은 거의 불가능 하다고 할 수 있다. 따라서, 이러한 것들을 위해서는 프로그램 작성 단계에서부터 버그의 원인을 알아내도록 하는 대책이 필요하다.

이러한 사유로 DCL은 소스 코드 차원에서의 디버깅을 위한 코드 삽입을 자동화 하도록 하여 소프트웨어의 개발 단계 및 테스트 단계에서 필요한 정보를 생성하도록 하는 런타임 환경을 제공하고 있다. 특히, 동적 메모리 관련 디버깅 환경 부분은 프로세스 뿐만 아니라 다중 스레드 어플리케이션에서 각각의 스레드별 추적도 가능 하도록 하고 있다.

본 문서는 런타임 디버깅 환경을 위한 DCL의 ASSERT, TRACE, 그리고 동적 메모리 할당과 관련한 참조 매뉴얼을 제공한다.

- 목 차 -

1. 디폴트 컴파일러 매크로

1.1 _DEBUG

이 매크로는 Microsoft Visual C++에서 디버그 버전으로 컴파일 할 때 사용된다. DCL에서 이 매크로가 설정하려면 “CRTD Debug” 설정을 사용하여 MSVCRTD.DLL을 사용하도록 하여야 한다. 이것의 사용 목적은 DCL 자체의 디버깅을 위해 사용되어 진다.

DCL을 사용하여 프로그램을 작성하는 대부분의 경우에는 이 매크로 사용하지 않도록 하여야 한다.

1.2 NDEBUG

표준 C 의 런타임 라이브러리인 LIBC에서 사용하는 매크로로 표준 assert와 관련이 있다. LIBC에서 제공하는 assert의 기본 동작은 abort 하기 때문에 DCL을 사용하여 개발할 때에는 이것을 사용하지 않도록 하여야 한다. 컴파일 타임에 NDEBUG가 정의되면 표준 assert 매크로는 컴파일 타임에 무시된다.

2. 디버그 버전 라이브러리 컴파일 매크로

DCL에 포함된 디버깅코드는 원칙적으로 C/C++의 매크로 기능을 사용한다. 따라서 릴리즈 버전으로 컴파일이 된 코드에는 대부분의 디버깅 코드가 포함되지 않는다. 디버그 버전 라이브러리를 위한 매크로의 기본 설정은 dcl/_Config.h에 포함되어 있다.

2.1 __DCL_DEBUG

라이브러리를 디버깅 버전으로 컴파일 하기 위해서는 dcl/_Config.h에 포함된 __DCL_DEBUG가 정의(define)되어야 한다.

2.2 __DCL_HAVE_ALLOC_DEBUG

동적메모리 할당 관련 디버깅을 활성화 한다. 이 매크로가 정의되면 new, new[], delete, delete[], malloc, calloc, realloc, free에 대하여 메모리 할당과 사용에 대하여 런타임 디버깅 정보를 유지한다.

2.3 __DCL_HAVE_STRING_ALLOC_DEBUG

DCL의 String 클래스는 문자열을 저장하기 위하여 new[] 연산자를 사용한 동적인 메모리 할당을 한다. 이것은 String 클래스의 버그를 발견하기 위함으로 디폴트 설정은 비활성화 되어 있다.

2.4 __DCL_HAVE_THROW_LOCATION

예외가 발생하여 Exception 객체를 던질(throw) 때 예외가 발생한 소스 파일 및 라인 정보를 포함하도록 한다. __DCL_DEBUG가 정의되어 있으면 이것의 디폴트 설정은 활성화 된다.

2.5 __THIS_FILE__

TRACE, ASSERT, __DCL_THROW, malloc, calloc, realoc, free, new, new[] 등의 위치를 위한 소스 파일의 이름을 지정한다. 이것을 지정하지 않으면 기본적으로 __FILE__일 매크로로 대체된다.

3. 디버그 라이브러리의 메시지 출력

3.1 DCLInitialize

```
DCLCAPI bool DCLInitialize(
                    void*,          // reserved      must NULL
                    int nFD         // for Debug Report -1(no debug) or valid file descriptor
                    );
```

DCL을 사용하기 위하여 반드시 이 함수를 호출 하여야 한다. 디버그 버전에서 nFD가 -1 이상의 유효한 파일 기술자(descriptor)이면, 내부적으로 XFileOutputStream(Mutexed FileOutputStream) 객체를 생성하여 그 포인터를 사용하여 DCLDebugSetGlobalReport를 호출한다.

DCLInitialize는 내부적으로 함수호출 카운터를 유지하기 때문에 2번 이상의 호출에는 아무런 행위를 하지않고 true를 리턴한다.

3.2 DCLCleanup

```
DCLCAPI void DCLCleanup();
```

DCL을 사용하면 내부적으로 시스템 자원을 할당한다. 이 함수는 DCL이 사용한 시스템 자원을 해제한다. 디버그 버전에서는 DCLInitialize에서 할당한 XFileOutputStream이 사용한 자원을 해제한다.

라이브러리가 __DCL_HAVE_ALLOC_DEBUG가 활성화 있고 DCLDebugSetGlobalReport에 의하여 설정된 스트림이 있으면 DCLDebugDumpGlobalMemoryLeak을 호출하여 메모리 누출을 보고한다.

3.3 DCLDebugSetGlobalReport

```
DCLCAPI __DCL_NAMESPACE OutputStream* DCLDebugSetGlobalReport(
                    __DCL_NAMESPACE OutputStream* pNewOutput
                    );
```

프로세스 전역에 걸쳐 __DCL_TRACE, __DCL_ASSERT의 출력 스트림을 설정한다. 리턴값은 이전의 스트림 객체의 포인터로 NULL 일 수 있다. 다중 스레드 어플리케이션의 경우 OutputStream의 write 멤버는 뮤텍스에 의해 동기 되어져 있어야 한다.

3.4 DCLDebugSetThreadReport

```
DCLCAPI __DCL_NAMESPACE OutputStream* DCLDebugSetThreadReport(
                        unsigned long uThreadId,
                        __DCL_NAMESPACE OutputStream* pNewOutput
                        );
```

스레드 uThreadId의 디버깅 출력을 설정한다. 이 함수를 사용하여 특정 스레드에 디버깅 출력이 설정되면 __DCL_TRACE, __DCL_ASSERT는 이 출력 스트림에 메시지를 출력한다. 스레드를 위한 OutputStream 객체는 특정 스레드에 의해서만 사용되기 때문에 뮤텍스(mutex)에 의한 동기화가 필요 없다.

3.5 OutputStream

디버깅 출력은 스트림 객체의 write, print, printf 등의 멤버함수를 통해 이루어 진다. 다중 스레드 어플리케이션의 경우에는 뮤텍스에 의해 출력이 동기화 될 필요가 있다.

OutputStream 클래스의 다중 스레드 안전 버전은 'X' 접두사를 갖는다. 대표적인 클래스는 XFileOutputStream, XByteBufferOutputStream과 같은 것이 있다.

4. ASSERTION

ASSERTION는 해당 위치에서 주어진 조건을 만족하지 못하면 조건식, 소스파일, 라인을 출력하고 프로그램의 실행을 중단(abort)하는 강력한 디버깅 도구이다.

DCL에 삽입된 ASSERTION의 동작은 C-런타임의 assert와 동일한 동작 외에 AssertException*를 throw 하게 할 수 있다. 이는 데몬 프로세스나 CGI 프로그램과 같이 프로그램의 실행이 사용자와 직접적인 상호작용이 불가능 할 경우 유용하다.

4.1 ASSERT

__DCL_ASSERT(expr), ASSERT(expr)
__DCL_ASSERT_EX(expr, msg), ASSERT(expr, msg)

expr이 실패(false)하면 ASSERTION동작을 한다. __DCL_DEBUG가 정의되어 있을 때에만 의미가 있으며 릴리즈 버전에서 expr은 그 문장 자체가 무시된다. __DCL_ASSERT_EX는 expr과 함께 msg를 출력한다.

ASSERT는 __DCL_ASSERT와 동일하다.

4.2 VERIFY

__DCL_VERIFY(expr), VERIFY(expr)

ASSERT와 동일한 동작을 하며 __DCL_DEBUG가 정의되어 있지 않으면 expr은 평가되지 않고 그대로 실행된다.

ASSERT와 VERIFY의 차이점은 릴리즈 버전에서 VERIFY의 expr은 정상적인 문장의 역할을 수행하고 ASSERT의 expr은 제거된다는 것이다.

VERIFY는 __DCL_VERIFY와 동일하다.

4.3 DCLAssert

```
DCLCAPI void DCLAssert(
                const char* pszExpr,
                const char* pszMessage,
                const char* pszFileName,
                int nLine
                ) __DCL_THROW;
```

__DCL_ASSERT(expr)와 __DCL_ASSERT_EX(expr, msg)가 실패하면 호출되는 함수이다. 입력 expr과 msg의 내용을 출력하고 abort하거나 throw한다.

4.4 DCLAssertSetAction

DCLAssertAction DCLAssertSetAction(DCLAssertAction assertAction)

DCLAssert의 동작을 결정한다. assertAction은 DCL_ASSERT_ABORT와 DCL_ASSERT_THROW이어야 하며 이 함수를 호출하기 전의 디폴트 값은 DCL_ASSERT_ABORT이다. 함수 호출 후 리턴값은 이전에 설정된 값이다.

DCL_ASSERT_ABORT는 __DCL_ASSERT(expr)의 expr을 출력하고 abort 되며, DCL_ASSERT_THROW는 AssertException* 를 throw한다.

5. TRACE

TRACE는 프로그램의 실행 중에 유용한 정보를 출력할 수 있도록 하는 매크로이다. TRACE는 디버그 버전에서만 컴파일 된다.

TRACE의 출력은 프로세스에 할당된 파일과 쓰레드에 할당된 파일이 있을 수 있으며, 쓰레드에 할당된 파일이 있으면 이곳에 기록을 하고 그렇지 않으면 프로세스에 할당된 파일에 기록한다. 어떠한 파일

도 할당되어 있지 않으면 TRACE의 출력은 무시된다.

5.1 TRACE의 종류

__DCL_TRACE0(psz)
__DCL_TRACE1(fmt, arg1)
__DCL_TRACE2(fmt, arg1, arg2)
__DCL_TRACE3(fmt, arg1, arg2, arg3)
__DCL_TRACE4(fmt, arg1, arg2, arg3, arg4)

TRACE0(psz)
TRACE1(fmt, arg1)
TRACE2(fmt, arg1, arg2)
TRACE3(fmt, arg1, arg2, arg3)
TRACE4(fmt, arg1, arg2, arg3, arg4)

__DCL_TRACE의 사용은 printf 함수를 사용하는 것과 동일하다.

fmt는 반드시 상수문자열 이어야 한다.

const char* pszFormat = “%d”;
TRACE1(pszFormat, 10);

과 같은 형태로 사용할 수 없으며

TRACE1(“%d₩n”, 10);

과 같이 사용하여야 한다. 이는 TRACE 매크로가 출력 포맷을 변경하여 소스파일의 파일이름과 라인번호를 추가하기 때문이다.

TRACE는 __DCL_TRACE와 동일하다. 라이브러리 내부에서는 __DCL_TRACE가 사용되고 있으며 이것은 미래의 다른 라이브러리와의 충돌을 방지하기 위함이다.

5.2 DCLTrace

void DCLTrace(pszFormat, ...)

TRACE 관련 매크로에서 사용하는 함수로 사용방법은 printf와 동일하다. DCLDebugSetGlobalReport나 DCLDebugSetThreadReport에 의해 지정된 스트림에 출력한다.

6. 동적 메모리 디버그

DCL의 동적 메모리 디버깅이 활성화 되려면 __DCL_HAVE_ALLOC_DEBUG가 활성화 되어야 한다. 이 매크로가 정의되면 __DCL_DEBUG는 반드시 정의되어야 한다. __DCL_DEBUG가 정의되어 있어도 __DCL_HAVE_ALLOC_DEBUG가 정의되지 않으면 메모리 관련 디버깅 루틴들은 활성화 되지 않는다. DCL의 동적 메모리 할당관련 디버깅은 메모리 블록의 종류와 스레드 정보를 유지한다.

6.1 Functions and Operators

동적 메모리 할당 관련 디버깅은 malloc, free, realloc, calloc과 new, new[], delete, delete[] 연산자에 대해서 행해지며 이들 함수와 연산자의 추적은 __DCL_HAVE_ALLOC_DEBUG가 활성화 되어 있으면 자동으로 이루어 진다.

동적 메모리를 조작하는데 있어서 범하기 쉬운 실수의 첫번째는 할당하지 않는 메모리나 혹은 해제한 메모리를 해제 하려고 시도하는 경우이고 두 번째는 매치되지 않는 연산자의 사용이다. 가령 new[] 연산자에 의하여 할당된 객체는 반드시 delete[] 연산자에 의해 파괴 되어야 한다. delete와 delete[] 연산자는 내부적으로 동일한 free 함수를 사용하지만 delete[] 연산자는 배열을 이루는 모든 객체의 파괴자(destructor)를 개별적으로 호출하는 차이점이 있다.

DCL은 내부적으로 이러한 실수들에 대한 정보를 보고한다.

6.2 DCLDebugGetLastAllocPosition

```
DCLCAPI const void* DCLDebugGetLastAllocPosition(
                        unsigned long uThreadId
                        );
```

uThreadId가 할당한 메모리 블록의 마지막 위치를 되돌린다. uThreadId가 할당한 메모리가 없으면 NULL을 되돌린다. 이 함수는 DCLDebugDumpThradMemoryLeak을 사용하여 스레드가 할당한 메모리 블록의 리스트를 출력하는데 있어서 범위를 지정하기위한 목적이 있다.

6.3 DCLDebugDumpThreadMemoryLeak

```
DCLCAPI int DCLDebugDumpThreadMemoryLeak(
                        unsigned long    uThreadId,
                        const void*      pvStartPosition,
                        DCLAllocLeakDumpLevel level
                        __DCL_NAMESPACE OutputStream* pOut
                        );
```

uThreadId가 할당한 pvStartPosition에서 최근까지의 메모리 누출을 스트림에 출력한다. 성공하면 메

모리 누출 항목의 개수를 되돌리며 그 값은 0 보다 크거나 같다. 실패 하면 –1을 되돌리고 이것은 출력을 위한 버퍼할당 오류를 의미한다.

pvStartPosition은 NULL일 수 있으며 이 경우 uThreadId가 할당한 전체 메모리 블록의 리스트를 보고한다.

level은 다음 값 중 하나이다.

- DCL_ALLOC_DUMP_DEFAULT : DCL 외부에서 할당된 동적 메모리 정보와 DCL 내부에서 생성되었지만 파괴되지 않은 Exception 객체를 보고한다.
- DCL_ALLOC_DUMP_INTERNAL : DCL 내부를 포함한 모든 해제되지 않은 메모리 블록의 정보를 보고한다.

6.4 DCLDebugDumpGlobalMemoryLeak

```
DCLCAPI int DCLDebugDumpGlobalMemoryLeak(
                        DCLAllocLeakDumpLevel level
                        __DCL_NAMESPACE OutputStream* pOut
                        );
```

프로세스가 할당한 전체 메모리 누출을 보고한다. 성공하면 메모리 누출 항목의 개수를 되돌리며 그 값은 0 보다 크거나 같다. 실패 하면 –1을 되돌리고 이것은 출력을 위한 버퍼할당 오류를 의미한다.

7. 예외(exception) 위치 보고

7.1 __DCL_THROW

DCL은 예외가 발생했을 때 Exception 클래스로부터 파생된 객체의 포인터를 던진다(throw). 따라서 이들 예외객체를 잡기(catch) 위해서는 적어도 다음과 같은 문장 이어야 한다.

```
try
{
        AnyException* e = new AnyException;
        __DCL_THROW(e);
}
catch(Exception* e)
{
        e->destroy();
}
```

여기에서 __DCL_THROW 매크로는 __DCL_HAVE_THROW_LOCATION가 활성화 되었을 경우 예외를 던지기(throw) 전에 예외객체 *e 에 예외가 던져지는 소스파일의 이름과 라인번호를 설정한다. 이러한 방법은 예외의 메시지로부터 예외가 발생한 원인을 쉽게 발견하도록 하는데 도움을 준다.