new 오버로딩으로 메모리 누수 체크

```
#include "stdafx.h"

// 메모리 할당,해제 정보를 저장해주는 클래스

// 실제 라이브 프로젝트에 사용하기 위해선 몇 가지의 추가 개선이 필요하다.
// 1. 메모리 할당정보 저장을  배열에서  리스트 방식으로 (현재는 1000개 배열이라 제한이 있음)
// 2. 클래스에 대한 배열 생성시 4,8바이트를 더 확보하므로 내부에서 저장한 할당 포인터와
//    사용자에게 반환된 포인터는 다를 수 있음.  이에 대한 처리는 생략


class CMemoryHistory
{
public:
	enum
	{
		FILE_NAME_LEN = 128,
		ALLOC_INFO_MAX = 1000
	};
	// 메모리 할당 정보 하나
	struct stALLOC_INFO
	{
		void	*pAlloc;		// 할당받은 포인터
		int	iSize;			// 할당 사이즈
		char	szFile[FILE_NAME_LEN];	// 할당 파일 위치
		int	iLine;			// 파일 라인
		bool	bArray;			// 배열생성 여부
	};

private:
	// 메모리 할당 정보들 저장 배열.
	stALLOC_INFO	_allocInfos[ALLOC_INFO_MAX];
	char		_logFileName[64];

public:
	CMemoryHistory(char *szLogFile = "AllocInfo");
	~CMemoryHistory();

private:
	bool	SaveLogFile(void);
	bool	NewAlloc(void *pPtr, char *szFile, int iLine, int iSize, bool bArray = false);
	bool	Delete(void *pPtr, bool bArray = false);

	// new 연산자 오버로딩 함수에서는 본 클래스를 맘껏 접근 할 수 있다.
	friend void * operator new (size_t size, char *File, int Line);
	friend void * operator new[] (size_t size, char *File, int Line);
	friend void operator delete (void * p);
	friend void operator delete[] (void * p);
};
```



```
CMemoryHistory::CMemoryHistory(char *szLogFile)
{
	char szTime[17] = "";
	time_t timer;
	struct tm TM;

	memset(_allocInfos, 0, sizeof(stALLOC_INFO) * ALLOC_INFO_MAX);
	memset(_logFileName, 0, 64);

	time(&timer);
	localtime_s(&TM, &timer);

	// YYYYMMDD_hhmmss 형식의 날짜로 만든다.
	sprintf_s(szTime, 17, "%04d%02d%02d_%02d%02d%02d ",
			TM.tm_year + 1900,
			TM.tm_mon + 1,
			TM.tm_mday,
			TM.tm_hour,
			TM.tm_min,
			TM.tm_sec);

	strcat_s(_logFileName, 64, szLogFile);
	strcat_s(_logFileName, 64, szTime);
	strcat_s(_logFileName, 64, ".txt");
}
CMemoryHistory::~CMemoryHistory()
{
	SaveLogFile();
}
bool CMemoryHistory::SaveLogFile(void)
{
	//-----------------------------------------------------
	// 지금은 클래스 파괴자에서 일괄저장을 하고 있다.
	// 프로그램이 돌아가는 중간에 상황을 보고 싶거나, 모니터링이 필요한 경우가 있다면 기능추가 필요
	//-----------------------------------------------------
	FILE *pLogFile;
	errno_t err = fopen_s(&pLogFile, _logFileName, "a");
	if ( err != 0 ) return false;

	for ( int iCnt = 0; iCnt < ALLOC_INFO_MAX; iCnt++ )
	{
		if ( _allocInfos[iCnt].pAlloc != NULL )
		{
			// 저장포멧
			// 해제된 메모리 -		[메모리포인터] [사이즈] 파일위치 : 줄번호
			// 누수된 메모리 - LEAK !! [메모리포인터] [사이즈] 파일위치 : 줄번호
			fprintf(pLogFile, "LEAK !! ");
			fprintf(pLogFile, "[0x%p] [%7d] %s : %d \n",
					_allocInfos[iCnt].pAlloc,
					_allocInfos[iCnt].iSize,
					_allocInfos[iCnt].szFile,
					_allocInfos[iCnt].iLine);
		}
	}
	fclose(pLogFile);
	return true;
}
```

```
bool CMemoryHistory::NewAlloc(void *pPtr, char *szFile, int iLine, int iSize, bool bArray)
{
        for ( int iCnt = 0; iCnt < ALLOC_INFO_MAX; iCnt++ )
        {
                if ( _allocInfos[iCnt].pAlloc == NULL)
                {
                        _allocInfos[iCnt].bArray = bArray;
                        _allocInfos[iCnt].pAlloc = pPtr;
                        _allocInfos[iCnt].iLine = iLine;
                        _allocInfos[iCnt].iSize = iSize;
                       strcpy_s(_allocInfos[iCnt].szFile, FILE_NAME_LEN, szFile);
                       return true;
                }
        }
        return false;
}
```

```
bool CMemoryHistory::Delete(void *pPtr, bool bArray)
{
        FILE *pLogFile;
        for ( int iCnt = 0; iCnt < ALLOC_INFO_MAX; iCnt++ )
        {
                if ( _allocInfos[iCnt].pAlloc == pPtr )
                {
                        //-----------------------------------------------------
                        // 배열선언 후 일반삭제,  일반삭제 후 배열삭제시 오류!
                        //-----------------------------------------------------
                        if ( _allocInfos[iCnt].bArray != bArray )
                        {
                                errno_t err = fopen_s(&pLogFile, _logFileName, "a");
                                if ( err == 0 )
                                {
                                        fprintf(pLogFile, "ARRAY !! [0x%x] [%7d] Wn",
                                                _allocInfos[iCnt].pAlloc, _allocInfos[iCnt].iSize);
                                        fclose(pLogFile);
                                }
                                return false;
                        }
                        _allocInfos[iCnt].pAlloc = NULL;
                        return true;
                }
        }

        errno_t err = fopen_s(&pLogFile, _logFileName, "a");
        if ( err != 0 ) return false;

        fprintf(pLogFile, "NOALLOC  [0x%x] Wn", pPtr);
        fclose(pLogFile);

        return false;
}
```



```
CMemoryHistory g_MemHistory;

// 이 객체를 MemoryHistory.cpp 내에 전역 변수로 선언한다
// CMemoryHistory 객체는 외부 cpp 에서 임의로 선언해서 사용하는 목적이 아니며
// 전역객체로서 프로세스 가동시 생성자, 종료시 소멸자의 호출로 초기화,자동저장이 목적이다


// new 연산자를 오버로딩 한다.
// 어떤 클래스던 받아주기 위해 전역함수로 new 를 오버로딩 한다.

// new 연산자의 메모리 할당 > 클래스 생성자 호출 (+ 가상함수 테이블 셋팅)을 해준다.
// new 연산자 오버로딩 시 우리는 우리가 원하는 사이즈로 메모리 할당만을 하고
// 클래스 생성과 관련된 작업은 컴파일러가 해주게 된다.
//

/*  우리가 쓸건 이 기본형이 아니다.
void * operator new (size_t size)
{
        void *p = malloc(size);
        // void *p = new char[size];
        // 이 내부에서 또 new 로 할당도 가능하다.  여기서 호출하는 new 는 오버로딩 된 new 가 아니다
        // 하지만 일부 컴파일러 에서 delete 오버로딩의 재귀호출 문제가 발생한다. 이는 뒤에서 설명
        cout << "메모리 할당" << size << endl;
        return p;
        // 메모리 할당 후 리턴을 해주면 컴파일러는 이 메모리에 클래스 셋팅을 해준다.
}
*/

// 다음과 같은 new 오버로딩이 가능하다. new 에 인자를 붙여 넣도록. 우린 이걸 사용
//
// 사용은 new("ddd", 10) int;  처럼 사용한다.
// 단 이 경우는 delete 도 같은 형식으로 해주어야 경고가 없어진다.
void * operator new (size_t size, char *File, int Line)
{
        void *p = malloc(size);
        g_MemHistory.NewAlloc(p, File, Line, size);
        return p; // 메모리 할당 후 리턴을 해주면 컴파일러는 생성자 호출 등을 해준다.
}

void * operator new[] (size_t size, char *File, int Line)
{
        void *p = malloc(size);
        g_MemHistory.NewAlloc(p, File, Line, size, true);
        return p;
}


// 위와 같이 추가인자가 존재하는 new 를 만들 경우
// 쌍의 delete 를 만들어야 경고가 없다 이는 예외처리용 이며 호출용이 아님
void operator delete (void * p, char *File, int Line) { }
void operator delete[] (void * p, char *File, int Line) { }
```

```
void operator delete (void * p)
{
        if ( g_MemHistory.Delete(p) )
                free(p);
}
void operator delete[] (void * p)
{
        if ( g_MemHistory.Delete(p, true) )
                free(p);
}

// delete 오버로딩의 경우 delete 에서 delete  ,  delete[] 에서 delete[] 는 문제가 없으나
// delete 에서 delete[] 를 하는 경우 오버로딩 된 delete[] 가 호출 된다. (이는 컴파일러에 따라 다를 수 있다)
// new delete 오버로딩 내부에선 malloc free 를 쓰자.

// #define new(X)         new(__FILE__, __LINE__) X;
// 이렇게 디파인을 걸면 new(char[100])  스타일.

#define new      new(__FILE__, __LINE__)
// 이렇게 디파인을 걸면 new char[100] 스타일. << 기존 new 와 동일.

//------------------------------------------------
// 테스트용 클래스
//------------------------------------------------
class Base
{
public:

        Base() { cout << "ABC 생성자" << endl; }
        int _a;
        int _b;
        int _c;
};

class A : public Base
{
public :

        A() { cout << "A 생성자" << endl; }
};
```



```
int _tmain(int argc, _TCHAR* argv[])
{
        // 앞으로 나오는 모든 new 는 디파인 된 new 이다.
        // 기본 연산자 보다 define 이 우선됨.
        A *p1 = new A[10];

        // new 를 디파인 매크로로 하지 않았다면
        // p1 = new(__FILE__, __LINE__) A[10];  처럼 써야한다.

        A *p2 = new A;
        A *p3 = new A[100];

        // p3 의 경우 클래스 배열을 생성 하였으므로,
        // 실제 필요 메모리보다 4바이트 (8바이트) 정도 큰 메모리가 할당된 후
        // 4바이트 뒤의 포인터에 p3 로 리턴 됨.

        // 그러므로 메모리 히스토리 내부에 저장된 할당 포인터와 우리가 받은 p3 포인터 주소가 다를 수 있음
        // 이는 delete p3;  의 일반 해제 실수를 제대로 감지하지 못할 수 있음.

        // placement new~  메모리 할당은 하지 않고, 그 공간을 셋팅 (생성자 호출) 만 함.
        // p = new(p) A;
        // 이건 그냥 알아두어야 할 new 문법.

        delete p1;
        delete p2;
        delete[] p3;

        return 0;
}
```

프로그램이 종료되고 나면 메모리 할당내역, 해제내역 및 정보가 파일로 저장된다.