**Visual Leak Detector.**

Загрузить можно отсюда:

1. <https://visualstudiogallery.msdn.microsoft.com/7c40a5d8-dd35-4019-a2af-cb1403f5939c/>
2. <http://vld.codeplex.com/>

По существу, это обертка над C++ CRT.

* После загрузки установить (куда угодно, но лучше по тому пути, который прописан изначально).
* Скопировать VLD библиотеку “.lib” в Вашу папку с Visual Studio C++ (у меня это внутренняя папка студии VC) в папку lib.
* Скопировать VLD заголовочный файл “.h” из папки inclide в Вашу папку с Visual Studio C++ (у меня это внутренняя папка студии VC) в папку include.
* В файле программы перед главной точкой входа включить файл заголовка “vld.h”. Лучше, чтобы он был до остальных заголовочных файлов (но это не обязательно), но строго после “stdafx.h”.
* Запустить проект в режиме отладки

Отчет обо всех обнаруженных утечках памяти будет отображаться в окне вывода отладчика при выходе из вашей программы. Двойной щелчок по номеру строки файла с исходным текстом в отчете утечки памяти приведет к этому файлу и строке в окне редактора, что позволяет легко перемещаться по коду.

Примечание: При создании релиз-версии вашей программы VLD не будет включен в исполняемый файл. Так что его безопасно оставлять подключенным при выполнении релиз-сборки, и это не приведет к каким-либо ухудшениям рабочих характеристик или любой другой нежелательной ситуации.

**Создание детектора**

VLD создавалось как лучшая замена для детектора утечки памяти, встроенного в Visual C ++. Поэтому в VLD используется примерно тот же метод, что и во встроенном детекторе, а именно CRT Debug Heap с некоторыми добавлениями. Но этот новый детектор имеет значительные усовершенствования - в первую очередь, указывает полный путь, что полезно для обнаружения и устранения утечек.

Встроенный детектор

Встроенный детектор довольно прост на самом деле. Вместе с началом работы программы библиотека CRT запускает ветвь очистки. Если встроенный детектор включен, то он запускает проверку на утечку памяти как часть процедуры очистки. Проверка утечки памяти просто смотрит на отладочную ветвь: если какие-либо пользовательские блоки по-прежнему существуют в куче, то это должны быть утечки памяти. Отладочная версия malloc сохраняет файл и номер строки в заголовочный файл (то есть запоминается каждое выделение памяти). Когда встроенный детектор определяет утечку памяти, он просто смотрит внутрь заголовка, чтобы получить имя файл и номер строки. Затем он передает эту информацию в отладчик, где он отображается.

Обратите внимание, что встроенный детектор обнаруживает утечки, не делая мониторинг распределения или освобождения памяти. Он просто делает снимок кучи непосредственно перед тем, как процесс завершается и определяет, есть ли какие-либо утечки, на основе этого снимка. Снимок кучи только говорит нам, если есть утечки, но не говорит нам о том, как они просочились и где. Очевидно, чтобы определить это "как" мы также должны получить путь. Но для этого мы должны быть в состоянии контролировать все распределение на лету во время выполнения. Это главное, что отличает этот детектор от встроенного.

К счастью для нас, компания Microsoft создала простой способ контролировать каждое распределение, сделанное из кучи отладки: крючки распределения (allocation hooks(AH)). AH - просто функция обратного вызова, которая будет вызываться непосредственно перед тем как производится каждое распределение из кучи отладки. Microsoft представила функцию \_CrtSetAllocHook, которая регистрирует функцию AH в куче. Когда отладчик вызывает АН, одним из аргументов является идентификационный номер, который однозначно определяет каждое распределение - это в основном серийный номер для каждого выделенного блока памяти. Но места в заголовке блока памяти не хватает для того, чтобы записать любую информацию непосредственно в нем, поэтому мы можем использовать этот уникальный идентификационный номер в качестве ключа для отображения каждого блока в любых данных, которые мы хотим записать.

Теперь, когда у нас есть способ получать уведомления каждый раз, когда блок будет выделен, а также способ однозначно идентифицировать каждое распределение, все, что осталось сделать, это делать запись в стеке вызовов каждый раз, когда происходит распределение. Опять же, Microsoft предоставила нам инструмент: на этот раз это функция, которую мы можем назвать walk to stack. Это функция StackWalk64. Она является частью отладки библиотеки DBGHELP.DLL. Каждый раз, когда вызывается StackWalk64, она возвращает структуру STACKFRAME64, которая может быть повторно использована в качестве входных данных для следующего вызова StackWalk64. Можно неоднократно использовать этот путь, пока не будет достигнут конец стека.

Теперь у нас есть начало детектора утечки памяти. Мы можем контролировать каждое распределение, и для каждого распределения можем получить и записать путь в стеке. Единственная проблема, которая остается, это гарантировать, чтобы функция АН регистрировала выделение памяти, как только программа начинает выполняться. Это может быть очень просто решено путем создания глобального экземпляра объекта класса ++ C. Конструктор \_CrtSetAllocHook будет работать как программа инициализации. Но как мы можем гарантировать, что наш конструктор будет вызван в первую очередь, и что функция АН будет установлена ​​до того, как будут построены другие глобальные объекты? К сожалению, нет абсолютных гарантий. Но мы можем подойти очень близко: использовать конкретную директиву препроцессора, которая явно говорит компилятору, чтобы наша глобальная переменная строилась как можно скорее: #pragma init\_seg (компилятор). Объекты в этом сегменте должны быть построены первыми и только потом объекты из сегмента "библиотеки" и объекты из сегмента "пользователь". Сегмент "пользователь" по умолчанию используется сегмент для глобальных объектов. Вообще говоря, никакие нормальные пользовательские объекты никогда не должны быть помещены в сегмент "компилятор", так что это обеспечивает довольно сильную уверенность, что наш глобальный объект будет построен перед любыми пользовательскими объектами.

Обнаружение утечек памяти

Поскольку глобальные объекты уничтожаются в порядке, обратном порядку их построения, наш глобальный объект будет уничтожен после любых пользовательских объектов. Затем мы можем исследовать кучу, так же, как это делает встроенный детектор. Если мы найдем блок в куче, который не был освобожден, то это утечка, и мы можем посмотреть его стек вызовов с использованием уникального идентификационного номера, записанного с помощью нашей функции АН.

Все, что мы имеем в стеке вызовов, представляет собой множество программных адресов в шестнадцатеричной форме. Для того, чтобы эти адреса были нам понятны, нужно перевести их на язык, понятный человеку: имя файлы и номера строк (и имена функций тоже). Опять же, Microsoft помогает нам с инструментами: из API мы можем использовать обработчик символов, чтобы получить имена файлов, номера строк и имена функций( SymGetLineFromAddr64 преобразует адреса в исходные имена файлов и номера строк, а SymFromAddr преобразует адреса в имена символов).

Полезная вроде как статься, мало ли на будущее.

<https://habrahabr.ru/company/sebbia/blog/243537/>

**Debug CRT С++**

Для использования Debug CRT надо подключить соответствующий header и включить использование Debug Heap Alloc Map. Делается это всего несколькими строками кода:

Подключение Debug CRT

#ifdef \_DEBUG  
#include <crtdbg.h>  
#define \_CRTDBG\_MAP\_ALLOC  
#endif

После этих действий при выделении памяти через new и malloc() данные будут оборачиваться в специальную структуру \_CrtMemBlockHeader. С помощью этой обертки мы сможем узнать имя файла и строку, в которой резервировалась ликнутая память, ее объем и сами данные. Все записи объединены в двусвязный список, поэтому по нему можно легко пробежаться и найти проблемные участки.

Структура \_CrtMemBlockHeader

typedef struct \_CrtMemBlockHeader  
{  
struct \_CrtMemBlockHeader \* pBlockHeaderNext;  
struct \_CrtMemBlockHeader \* pBlockHeaderPrev;  
char\* szFileName;  
int nLine;  
size\_t nDataSize;  
int nBlackUse;  
long lRequest;  
unsigned char gap[nNoMansLandSize];  
unsigned char data[nDataSize];  
unsigned char anotherGap[nNoMansLandSize];

} \_CrtMemBlockHeader;

Чтобы пройтись по этому списку, нужно воспользоваться функцией \_CrtDumpMemoryLeaks(). Она не принимает никаких параметров, а просто выводит список утекших блоков памяти. Но, к сожалению, она ничего не говорит нам о файле и строке, в которых выделялась память. Результат работы этой функции выглядит примерно так:

Вывод \_CrtDumpMemoryLeaks()

Detected memory leaks!  
Dumping objects ->  
{163} normal block at 0x00128788, 4 bytes long.  
Data: < > 00 00 00 00  
{162} normal block at 0x00128748, 4 bytes long.  
Data: < > 00 00 00 00  
Object dump complete.

Вот оно как: в Microsoft Visual C++ 6.0 в файле crtdbg.h имело место переопределение функции new, которое должно было точно показать файл и строку, в котором происходило выделение памяти. Но оно не давало желаемого результата — FILE:LINE всегда разворачивались в crtdbg.h file line 512. В следующих версиях Microsoft вообще убрали эту фичу, и весь груз ответственности лег на программистов. Сделать нормальный вывод можно с помощью следующего переопределения:

Переопределение new

#define new new( \_NORMAL\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_)

Эту строчку желательно вынести в какой-нибудь общий заголовочный файл и подключать его после crtdbg.h. Теперь перед нами стоит задача записать все это в какой-нибудь лог или хотя бы выводить в консоль. Для перенаправления вывода нам потребуются две функции: \_CrtSetReportMode и \_CrtSetReportFile. Вторым параметром \_CrtSetReportFile может быть хендл нашего лог-файла или флаг вывода в stdout.

Перенаправление вывода

\_CrtSetReportMode( \_CRT\_WARN, \_CRTDBG\_MODE\_FILE );  
// выводим все в stdout  
\_CrtSetReportFile( \_CRT\_WARN, \_CRTDBG\_FILE\_STDOUT );

У этого метода есть еще одна проблемка — он выводит информацию о памяти, которая не утекла, а просто не успела вернуться. Это, например, может быть какая-нибудь глобальная переменная или объект. Нам нужно как-то удалить эти куски памяти из вывода \_CrtDumpMemoryLeaks(). Делается это следующим образом:

Ограничение зоны действия \_CrtDumpMemoryLeaks()

int \_tmain(int argc, \_TCHAR\* argv[])  
{  
\_CrtMemState \_ms;  
\_CrtMemCheckpoint(&\_ms);  
// some logic goes here...  
\_CrtMemDumpAllObjectsSince(&\_ms);  
return 0;  
}

Мы записываем начальное состояние памяти в специальную структуру с помощью функции \_CrtMemCheckpoint(), а в конце, используя \_CrtMemDumpAllObjectsSince(), выводим все, что утекло после того, как мы сделали слепок памяти.

Вот так вот, с помощью нехитрых функций Debug CRT, мы можем достаточно эффективно бороться с мемори ликами в нашей программе. Конечно, это не заменит серьезных библиотек по отлову утечек, но вполне подойдет для небольших проектов.

**Еще один вариант:**

// test.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include "windows.h"

#ifdef \_DEBUG

#include <crtdbg.h>

#define \_CRTDBG\_MAP\_ALLOC // enable generation of debug heap alloc map

#define new new( \_NORMAL\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_) // redefine "new" to get file names in output

#endif

class T // simple test class to demonstrate global initialization memory leak

{

int\* m;

public:

T() { m = new int(); };

~T() { delete m; };

};

T b;

int \_tmain(int argc, \_TCHAR\* argv[])

{

#ifdef \_DEBUG

\_CrtSetReportMode( \_CRT\_WARN, \_CRTDBG\_MODE\_FILE ); // enable file output

\_CrtSetReportFile( \_CRT\_WARN, \_CRTDBG\_FILE\_STDOUT ); // set file to stdout

\_CrtMemState \_ms;

\_CrtMemCheckpoint(&\_ms); // now forget about objects created before

#endif

for (int i = 0; i < 24; i++) new int(); // create test stuff

#ifdef \_DEBUG

\_CrtMemDumpAllObjectsSince(&\_ms); // dump leaks

#endif

return 0;

}

И инструкция по ссылке (там все хорошо описано и у меня все работало)

https://habrahabr.ru/post/82514/