Speicher-Debugging
mit DUMA

Hayati Aygün

DUMA (Detect Unitended Memory Access) ist eine C/C++ Bibliothek, die Ihnen das Aufspüren von Speicher Fehlern erleichtert. Die Bibliothek wurde von Bruce Perens unter dem Namen Electric-Fence 1987-1999 für verschiedene Linux/Unix Varianten unter GPL Lizenz entwickelt.

Der Autor dieses Artikels hat die Bibliothek, aufgrund der freien Lizenz und mangelnder freier Alternativen unter Windows aufgegriffen und auf Windows ab 2000/XP portiert.

Motivation

Viele Fehler werden dadurch verursacht, dass „ungewollte" Werte gelesen oder Werte an „falsche" Adressen geschrieben werden. Dies passiert, weil der Programmierer beim Zugriff auf Array Variablen nicht die gegebene Vorsicht walten lässt. Meist überschreitet die Index Variable ungewollt die Grenzen des Arrays. Leider bleiben viele Überschreitungen oft unentdeckt.

Ursache – Wirkung

Es gibt die folgenden Möglichkeiten, wie sich diese Speicher bzw. Array-Überschreitungen auswirken können.

Bei Berechnungen wird nicht das gewünschte berechnet, z.B. ein Statiker berechnet mit einer Tabellenkalkulation die Tragfähigkeit eines geplanten Gebäudes. Hierbei greift nun seine Formel auf falsche Felder zu und das Ergebnis ist um Faktoren höher als in Wirklichkeit. Beim ersten Schneefall stürzt das Gebäude aber zusammen!

Im Speicher befindlicher Programmcode wird modifiziert. Dies hat nicht notwendigerweise sofort Auswirkungen. Soll dann später der modifizierte Programmcode ausgeführt werden, so kommt es zu einem Absturz des Programmes. Der Entwickler des Programmes kann nun stundenlang erfolglos an der Absturzstelle nach dem Fehler suchen. Er gibt nach einigen Tagen auf und das Programm bleibt instabil.

Man sieht, dass die Auswirkungen in obigen Beispielen nichts mit der eigentlichen Fehlerursache zu tun haben. Die Wirkung ist scheinbar „zufällig".

Wunschvorstellung

Obwohl es zunächst paradox klingt ist es vorteilhaft wenn Programme bei solchen

Über den Autor

Hayati Aygün ist Entwickler der Bibliothek, Diplom-Informatiker und als System-Ingenieur im Bereich digitaler Signalverarbeitung tätig.

Kontakt: *h_ayguen@web.de*

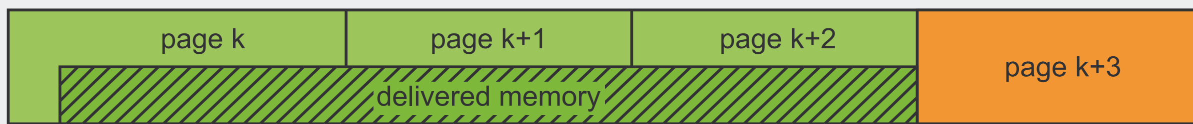

Abbildung 1. *Überschreitungsschutz*

Abbildung 2. *Unterschreitungsschutz*

Fehlern sofort abstürzen. Im ersten Beispiel hätte dies eine Katastrophe verhindert. Im zweiten Beispiel hätte der Entwickler des Programmes viel Zeit beim Finden des Fehlers sparen können. In jedem Falle wären Berechnungsergebnisse zuverlässiger und Programme von vornherein stabiler.

Mit dem sofortigem Absturz besteht nun auch die Möglichkeit im Debugger sofort an die fehlerverursachende Programmzeile zu springen.

Gegenmittel

Mit folgenden Mitteln kann man Fehler dieser Art einfacher aufdecken:

Verwendung von Hochsprachen: Die meisten Hochsprachen sind sicherer in der Nutzung von Arrays und Zeichenketten. Hochsprachen haben prinzipiell die Möglichkeit beim Interpretieren solche Fehler sofort zu melden. Allerdings geht die Nutzung von Hochsprachen immer mit einem höherem Resourcenverbauch von Prozessor und Speicher einher. Aus diesem Grund stellt dies in vielen Fällen keine Alternative dar.

Für das Visual Studio unter Windows gibt es das kommerzielle Programm BoundsChecker von Compuware. Für effektiven Schutz kann BoundsChecker sich in den Kompilierungsprozess einhängen und flexibel den Einbau von zusätzlichem Prüfcode veranlassen. Natürlich wird hierdurch die Ausführungsgeschwindigkeit des erzeugten Programmes beeinflusst.

Unter Linux gibt es das mächtige „valgrind" Werkzeug. Dieses führt das zu testende Programm in einer virtuellen Umgebung (Prozessor- und Speicheremulation) aus. Hierdurch können auch andere Fehlerarten entdeckt werden. Auch hier wird durch die emulierte Umgebung die Ausführungsgeschwindigkeit erheblich ausgebremst.

Nutzung der DUMA Bibliothek. Diese nutzt die Speicherverwaltungseinheit (MMU) und deren Schutzkonzepte, die moderne Prozessoren mitbringen. Dieses Werkzeug ist auch Thema dieses Artikels.

Konzept der DUMA Bibliothek

Ohne ein effektives Speicherschutzkonzept konnte „früher" ein fehlerhaftes Programm den gesamten Rechner in einen undefinierten Zustand versetzen. Durch das gleichzeitige Ausführen mehrerer Programme im Multitasking traten gehäuft Abstürze auf. Diese beeinflussten oft auch das Betriebssystem oder die anderen laufenden Programme. So wurden die Speicherverwaltungseinheiten inkl. Speicherschutzkonzepten zur besseren Unterstützung des Multitasking in die Prozessoren eingebaut. Aktuelle Betriebssysteme wie Linux und Windows NT/2000/XP nutzen diese Konzepte um fehlerhafte Programme vor dem System oder anderen Programmen zu schützen.

Bei intel wurde ab dem 386 Prozessor die lineare Adressierung und das Konzept der Speicherkachel (*Memory paging*) eingeführt. Hier kann das Betriebssystem jedem Prozess einen „eigenen" isolierten Speicherbereich zuweisen. Des weiteren kann für jede Speicherkachel (Grösse 4 kB) der Zugriff verwaltet werden:

- kein Zugriff,
- nur lesender Zugriff,
- schreibender (und lesender) Zugriff.

Andere Prozessoren haben ähnliche Fähigkeiten entwickelt. Bei Verletzung der Zugriffsrechte löst der Prozessor eine Ausnahmene aus. Ohne spezielle Umgebung stürzt das Programm also ab! Die Möglichkeit den Zugriffsschutz auf einzelne Speicherkacheln zuzuweisen steht unter Linux/Unix und Windows auch für Anwendungsprogramme zur Verfügung.

Funktionsweise

DUMA „überlädt" die Funktionsaufrufe zur Speicherallokation und Freigabe und reserviert den angeforderten Speicher so, dass das Ende (bzw. der Anfang) auf die Grenze einer Speicherkachel fällt. Direkt an dieser Grenze wird eine weitere

Listing 1. *Beispiel 1*

```
#include <malloc.h>
void main(){
   int *pi =
     (int*)malloc(10*sizeof(int));
   int i;
   for(i=0; i<11; ++i)
      pi[i] = i;
}
```

Speicherkachel reserviert. Allerdings wird jeglicher Zugriff auf diese weitere Kachel geschützt – siehe Abbildung 1. Dadurch, dass der Prozessor dieses Schutzkonzept direkt unterstützt, wird die Performanz des Programmes nur sehr gering beeinflusst. Nachteilig ist, dass nur der dynamisch reservierte Speicher auf diesem Wege geschützt werden kann.

In Abblildung 1 und 2 sieht man, dass immer nur ein Ende des ausgeliefertten Speichers durch den Prozessor effektiv geschützt werden kann. Es entsteht ein zusätzlicher Speicherverbrauch von bis zu zwei Speicherkacheln je Allokation; bei einem x86-PC entspricht dies 2 x 4 kB.

Einsatz

Für diese Bestriebssysteme muss die Bibliothek zunächst heruntergeladen, entpackt, konfiguriert und kompiliert werden. Die Konfiguration erfolgt durch Bearbeitung des Makefile. Kompilation erfolgt mit *gmake*. Es gibt Pläne (für die Zukunft) diesen Vorgang mit autoconf/automake durchzuführen. Im Makefile sind auch Beispiele (dumatest und tstheap), wie Programme gegen DUMA gelinkt werden. Wichtig ist hier die Link-Reihenfolge: die Objektdateien / Bibliotheken, die DUMA nutzen sollen, müssen beim Linken vor der DUMA Bibliothek stehen.

Für Linux kann auch der `LD_PRELOAD` Mechanismus genutzt werden; siehe Skript duma.sh. Dies ist insbesondere dann interessant, wenn das Programm nicht im Quelltext vorliegt aber dessen Stabilität geprüft werden soll. So kann mit diesem Mechanismus dem Lieferanten des Programmes nachgewiesen werden, dass das Programm fehlerbehaftet ist. Es sei hier aber erwähnt, dass die DUMA Bibliothek weitere Funktionen mitbringt, die mit dem `LD_PRELOAD` Mechanismus nicht funktionieren.

Listing 2. *Beispiel 2*

```
#include <malloc.h>
void main(){
   int *pi =
     (int*)malloc(10*sizeof(int));
   int i;
   for(i=0; i<11; ++i)
      pi[i] = i;
}
```

Listing 3. *Beispiel 3*

```
#include <malloc.h>
#include <duma.h>
void main(){
   int *pi =
     (int*)malloc(10*sizeof(int));
   int i;
   for(i=0; i<11; ++i)
      pi[i] = i;
}
```

Visual C++ unter Windows

Für den Einsatz unter Visual C++ 6.0 sind einige Voraussetzungen zu erfüllen:

- Die DUMA-Bibliothek (*.dsp*/*.vcproj* Datei) muss in den Arbeitsbereich aufgenommen werden. Anschliessend wird noch die Abhängigkeit von dem Startprojekt auf die DUMA gesetzt.
- Unter *Projekt–>Einstellungen–>Reiter C/C++–>Allgemein–>Debug-Info* sollte nicht '*Programmdatenbank für „Bearbeiten & Fortfahren"*' eingestellt sein.
- Unter *Projekt–>Einstellungen–>Reiter C/C++–>Kategorie Code Generation–>Laufzeit-Bibliothek* sollte keine DLL Variante eingestellt sein.
- Unter *Projekt–>Einstellungen–>Reiter Linker–>Kategorie Allgemein* sollte '*Inkrementelles Binden*' ausgeschaltet sein.
- Das Feld unter *Projekt–>Einstellungen–>Reiter Linker–>Kategorie Allgemein–>Projekt Optionen* muss um die Option `/FORCE:MULTIPLE` ergänzt werden.
- Das Feld unter *Projekt–>Einstellungen–>Reiter C/C++–>Kategorie Präprozessor–>Zusätzliche Include-Verzeichnisse* muss um den Pfad der DUMA Header Dateien ergänzt werden.

Beim Einstellen der oben genannten Schritte sollten möglichst alle Projekte ausgewählt sein. Bei der Linker-Einstellung ist nur das Startprojekt wählbar. Die Linker-Einstellungen können entfallen, wenn die Präprozessor Option `DUMA_NO_GLOBAL_MALLOC_FREE` gesetzt wird. Nicht zu vergessen ist auch, dass die Einstellungen für *Debug / Release / Alle Konfigurationen* zu tätigen sind. Mit analogen Einstellungen ist auch die Visual C++ .NET 2003 Version zur Zusammenarbeit zu bewegen.

Schalter / Optionen

Die DUMA Bibliothek bietet verschiedene Präprozessor Schalter, mit denen der Funktionsumfang konfiguriert werden kann. Diese Präprozessor Optionen werden unter Linux/Unix im Makefile eingestellt. Beim Visual C++ werden diese Optionen unter *Projekt–>Einstellungen–>Reiter C/C++–>Kategorie Präprozessor –>Präprozessor-Definitionen* angegeben. Die wichtigsten Schalter lauten wie folgt:

- `DUMA_EXPLICIT_INIT` – Dieser Schalter lässt Teile der Initialisierung explizit vom Anwender aufrufen. Die Funktion `duma_init()` muss von der `main()`-Funktion aufgerufen werden. Dies lässt die Verwendung von DUMA in Umgebungen zu, wo Systembibliotheken Speicherallokationen aufrufen und diese in DUMA landen. Oft ist dies im Zusammenspiel mit der pthread-Bibliothek Ursache für endlose Rekursionen. Zusätzlich werden Speicherallokationen, die vor dem `duma_init()`-Aufruf erfolgen, zu den Systembibliotheken gerechnet und nicht mehr in der Speicherleck-Suche berücksichtigt. Somit werden die Fehler von einigen Systembibliotheken ausgeblendet. Aber Achtung: Möglicherweise werden auch Speicherlecks von globalen C++ Objekten verdeckt, da der Konstruktor dieser Objekte ebenfalls vor dem Aufruf der `main()`-Funktion ausgeführt wird!
- `DUMA_NO_GLOBAL_MALLOC_FREE` – Dieser Schalter verhindert den Einsprung von Bibliotheken in DUMA Funktionen. Somit wird der Einsatz von DUMA auf Quelltext beschränkt, der explizit die Header Datei *duma.h* bzw. *dumapp.h* einbindet. Dies ist praktisch, wenn man vor nervenden Fehlermel-

Listing 4. *Beispiel 4*

```
#include <malloc.h>
#include <new>
#include <duma.h>
#include <dumapp.h>

void main()
{
   int *pi =
     (int*)malloc(10*sizeof(int));
   int i;
   for(i=0; i<10; ++i)
      pi[i] = i;
   delete pi;
}
```

Listing 5. *Beispiel 5*

```
#include <malloc.h>
#include <new>
#include <duma.h>
#include <dumapp.h>

void main()
{
   int *pi = new int[10];
   int i;
   for(i=0; i<10; ++i)
      pi[i] = i;
   delete []pi;
}
```

dungen in Bibliotheken von Dritt-Herstellern verschont werden möchte.

- `DUMA_NO_THREAD_SAFETY` – Dieser Schalter deaktiviert die Verwendung von Semaphoren zum Schutz der DUMA internen Funktionen. Wird dieser Schalter gesetzt, so darf kein Multi-Threading aktiviert sein.
- `DUMA_NO_CPP_SUPPORT` – Dieser Schalter deaktiviert die globalen C++ Operatoren `new`, `new[]`, `delete` und `delete[]`. Zur Nutzung der genannten C++ Operatoren muss die Datei *dumapp.h* eingebunden werden.
- `DUMA_NO_LEAKDETECTION` – Dieser Schalter schaltet die Detektion von Speicherlecks ab. Bleibt die Leck-Detektion eingeschaltet, so wird bei Programmende geprüft ob alle Allokationen auch wieder freigegeben wurden. Auch Systembibliotheken geben nicht immer alle allokierterten Speicherbereiche wieder frei. Mit dem Schalter `DUMA_NO_GLOBAL_MALLOC_FREE` kann in diesem Falle ausgeholfen werden. Nicht erfolgte Freigaben werden bei Programmende mit Fehlermeldungen und einem Absturz quittiert.

Umgebungsvariablen

Mit Umgebungsvariablen können einige Einstellungen der Bibliothek noch nach der Kompilierung parametriert werden. Die wichtigsten Variablen lauten:

- `DUMA_ALIGNMENT` – Mit dieser Variable wird die Ausrichtungsgrösse der zurückzuliefernden Speicheradressen vorgegeben. Standard ist `sizeof(int) = 4`. Für spezielle Anwendungen können grössere Werte wie z.B. 16 erforderlich sein. Der Wert sollte nicht zu gross gesetzt werden, da sonst der effektive Überschreitungsschutz der DUMA Bibliothek ausgehebelt wird!
- `DUMA_PROTECT_BELOW` – Mit einem Wert ungleich 0 wird festgelegt, dass das untere Ende des zurückgelieferten Speichers geschützt werden soll. Diese Variable sollte öfters umgestellt werden um möglichst viele Fehler aufzudecken.
- `DUMA_ALLOW_MALLOC_0` – Mit einem Wert ungleich 0 werden Allokationen der Grösse 0 nicht mehr als Fehler gewertet. Obwohl der C Standard dies offiziell erlaubt, deutet diese Grösse oft auf Fehler hin!
- `DUMA_FILL` – Diese Variable legt den Wert fest, mit dem der Speicher initialisiert wird. Auch der Wert dieser Variablen sollte ab und zu umgestellt werden, damit Initialisierungsfehler aufgedeckt werden. Oft hat das Programm „Glück" und die System-Initialisierung produziert keine Fehler.

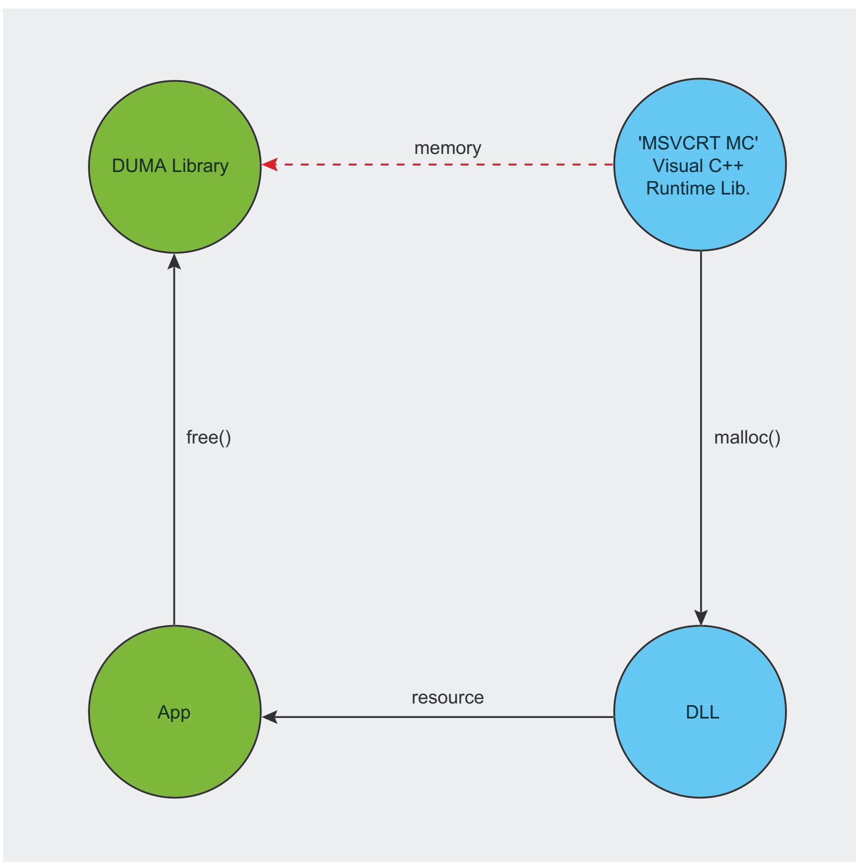

Abbildung 3. *Konflikt-Beispiel*

Listing 6. *Beispiel 6*

```
void main()
{
   int *pi = new int[10];
   int i;
   for(i=0; i<10; ++i)
      pi[i] = i;
   delete []pi;
}
```

Bei Änderung von Umgebungsvariablen muss die Visual-C++ Entwicklungsumgebung geschlossen und neu geöffnet werden, damit die Änderungen sich auswirken.

Einbindung und Konflikte

Durch Einbinden von Header-Dateien werden Funktionen von DUMA eingeschaltet bzw. erweitert.

Folgende Zeilen binden die Deklarationen für die zu ersetzenden C Speicherfunktionen ein. Zuvor muss *malloc.h* bzw. *stdlib.h* eingebunden werden:

```
#include <malloc.h>   /* or stdlib.h */
#include <duma.h>   /* for C */
```

Zur Ersetzung der C++ Operatoren zur Speicherverwaltung muss folgendes eingebunden werden:

```
#include <new>
#include <dumapp.h> // for C++
```

Bei Konflikten mit Präprozessor-Makros von DUMA mit anderen Deklarationen / Definitionen kann folgende Zeile verwendet werden:

```
#include <noduma.h>
```

um die Präprozessor-Makros von DUMA aufzuheben. Nach dem „Konflikt"-Bereich kann wieder *duma.h* oder *dumapp.h* eingebunden werden – auch mitten im Quelltext. Beispielsweise taucht in Header-Dateien eines kommerziellen Herstellers in einer C++ Klasse die Methode `free()` auf. Mit diesen Einbindungen um die include Anweisung des komerziellen Headers kann eine Fehlermeldung / Warnung des Compilers vermieden werden.

Mit der Einbindung von *duma.h* oder *dumapp.h* Präprozessor-Makros zur Weitergabe von Dateinamen und Zeilennummer des Aufrufers definiert. Dies ermöglicht bei der Ausgabe eines Speicherlecks die Angabe der Zeile, in der der Speicher allokiert wurde. Erst dieser Hinweis ermöglicht die Beseitigung des Speicherlecks.

Beispiele

Beispiel 1

Wird das Projekt zu Beispiel 1 wie beschrieben erstellt, so wird das Programm nach Start im Debugger mit einer Ausnahme (siehe Abbildung 3) beendet.

Nach Bestätigung der Ausnahme befindet man sich im Debugger auf Zeile 8. In der Überwachung des Debuggers kann der Wert des Schleifenzählers `i = 10` eingesehen werden. Für `pi[i]` wird „`CXX0030: Fehler: Ausdruck kann nicht ausgewertet werden`" in der Überwachung angezeigt. Dies deutet darauf hin, dass der Inhalt von `pi[i]` in einem geschütztem Bereich liegt.
Als Ursache lässt sich Zeile 7 ausmachen. Korrekt müsste es

```
for(i=0; i<10; ++i)
```

heissen.

Beispiel 2

Nach Korrektur von Beispiel 1 erhalten wir Listing 2.
Kurz nach Start erhalten wir folgende Fehlermeldung auf der Konsole:

```
DUMA: ptr=0x340fd8 size=40
   alloced from UNKNOWN (use #include
   "duma.h")(0) not freed
DUMA Aborting: DUMA_delFrame():
Found non free'd pointers
```

Die Fehlermeldung berichtet ein Speicherleck – allerdings ohne Angabe der Zeilennummer im Quelltext. In einem grösseren Programm, welches aus vielen Dateien und mehreren Speicherallokationen besteht, wäre der Fehler so nicht zu finden. Damit uns DUMA unterstützt, müssen wir die Header-Datei einbinden.

Beispiel 3

Wir erhalten nun Listing 3. Man beachte die Reihenfolge der `#include` Anweisungen!
Diesmal erhalten wir folgende Fehlermeldung:

```
DUMA: ptr=0x340fd8 size=40
   alloced from F:\_wincvs\duma\
   example3.cpp(6) not freed
DUMA Aborting:
DUMA_delFrame(): Found non free'd
                    pointers
```

Man sieht genauen Dateinamen und Zeilennummer des nicht freigegebenen Speichers. Sind mehrere Speicherbereiche nicht freigegeben, so werden alle einzeln gelistet.

Beispiel 4

Erweiterung von Beispiel 3 um eine Speicherfreigabe ergibt Listing 4. Nun lautet die Fehlermeldung auf der Konsole:

```
DUMA Aborting: Free mismatch:
allocator 'malloc()' used  at UNKNOWN
   (use #include "duma.h")(0) but
   deallocator 'delete (element)'
   called at UNKNOWN (use #include
   "dumapp.h")(0)!
```

Die Fehlermeldung besagt, dass wir bestimmte Allokationsmethoden nicht mit anderen Freigabemethoden vermischen dürfen. DUMA unterscheidet auch zwischen den Skalar und Vektor Varianten von `new`, `new[]` und `delete`, `delete[]`. Ausserdem können wir in den Debugger springen und auf dem Stack die `delete` aufrufende Zeile sehen. Somit können wir die Zeile 12 korrigieren.

Beispiel 5

Korrektur von Beispiel 4 ergibt Listing 4. Dieses Programm wird nun fehlerfrei ausgeführt und beendet.

Beispiel 6

Beispiel 5 kann für schreibfaule Programmierer noch weiter „optimiert" werden. Gerade für grosse Projekte, insbesondere bei gesetzter `DUMA_NO_GLOBAL_MALLOC_FREE` Option, mit vielen Dateien kann die manuelle Einbindung von `#include` Direktiven in den Quelltext eingespart werden. Für diesen Zweck gibt es beim gcc-Compiler die Option „`-include`". Beim Visual C++ heisst die Option „`/FI`".

Für die Nutzung dieser Option muss das Feld *Projekt Optionen* unter *Projekt–> Einstellungen–>Reiter C/C++–>Kategorie Allgemein* z.B. um `/FI"malloc.h" /FI"new" /FI"duma.h" /FI"dumapp.h"` ergänzt werden. Diese Änderung muss für Debug und Release Ihrer Projekte (nicht für die DUMA-Bibliothek) wiederholt werden. Der Quelltext kann ohne Einschränkung der Funktionalität zu Listing 6 reduziert werden.

Die abgedruckten Listings müssen nicht abgetippt werden. Die DUMA Archive (.zip /.tar .gz) ab Version 2.4.17 enthalten alle Listings inkl. den Visual C++ Projektdateien.

Ungelöste Probleme

DLL-Bibliotheken von Dritt-Herstellern werden fertig kompiliert bereitgestellt. In diesen Bibliotheken sind oft noch Verweise auf die Speicherallokations- und Freigabefunktionen der Standardbibliothek enthalten. Werden nun Speicherressourcen zwischen fremd-kompilierter DLL und eigener Anwendung ausgetauscht, so entstehen Konflikte. Beispiel zu Abbildung 4:

- Speicher wird in der DLL (über die Standardbibliothek) allokiert.
- Dieser Speicher bzw. die Ressource wird an die Anwendung übergeben.
- Der Speicher wird später von der Anwendung (über DUMA) freigegeben.

Im letzten Schritt entsteht der Konflikt. DUMA kann mit dem Speicher der Standardbibliothek nicht umgehen! In umgekehrter Richtung besteht dasselbe Problem. Bei Bibliotheken, die konsequent Freigabefunktionen für ihre Ressourcen bereitstellen, besteht dieses Problem nicht.

C++ Konformität

Die C++ Operatoren `new` und `new[]` der DUMA sind bisher nicht standard-konform implementiert. Dies stellt kein unlösbares Problem dar. Dem Nutzer der DUMA-Bibliothek sollte allerdings klar sein, dass aus diesem Grunde kein `new_handler` aufgerufen wird. ■