# leakHunter

Technical Concept Paper


Reto Carrara
Gyrenstrasse 17
CH-8967 Widen
+4179/328'17'05
www.carrara.ch
info@carrara.ch

# Inhalt

# Revision

| Datum | Version | Autor | Änderungen |
|---|---|---|---|
| 17.01.02 | 1.0 | rc | Basisversion |
| 15.02.02 | 1.1 | rc | Layout |
| 15.12.03 | 1.2 | rc | Fehler in LeakHunter5::HeapMgr::free() korrigiert |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Einführung

Vermutlich jeder C++ Programmierer kennt sie: die Memory Leaks. Auf die Frage, ob ein Programm noch Leaks hat antwortet der Programmierer grundsätzlich mit: „Ich bin mir nicht sicher, wahrscheinlich schon...!“.

Auf dem Markt existieren verschiedene Tools mit verschiedenen Ansätzen in den unterschiedlichsten Preislagen. Das vorliegende Paper zeigt, wie Memory-Leaks mit ein paar wenigen Zeilen Code höchsteffizient detektiert und für immer ausgemerzt werden können.

Dazu werden nur Standard-Sprachkomponenten der Sprache C++ verwendet, weshalb dieses System auch für jeden Compiler portierbar ist.

# Feststellen der Anzahl Leaks

In einer einfachsten Version wollen wir uns nur damit beschäftigen, die Anzahl der aufgetretenen Memory Leaks festzustellen.

Da es in C++ möglich ist, die Operatoren new und delete zu überladen, ist dies eine einfache Aufgabe.

Interessant ist folgender Umstand: sind in einer Basisklasse die Operatoren new und delete deklariert, so werden diese auch für alle Subklassen übernommen. Aus diesem Grund wird es sehr einfach möglich, das Allozieren von Speicher auf dem Heap durch eine Klasse zu kontrollieren.

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize);
    void operator delete(void* i_paObject);
};
```

Soll das Kreieren und Destruieren von Instanzen einer Klasse nun überwacht werden, so muss diese Klasse lediglich von HeapControl abgeleitet werden.

```
class A : public HeapControl
{
  int m_nInt;
};

class B : public A
{
  float m_nFloat;
};

class C : public HeapControl
{
  double m_nDouble;
};
```

Um Speicher tatsächlich auch zu allozieren, werden in diesem Beispiel die C-Funktionen malloc und free verwendet.

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize)
    {
      void* paObject = malloc(i_nSize);
      if (paObject)
      {
        ++sm_nCounter;
      }

      return paObject;
```

```
    }

    void operator delete(void* i_paObject)
    {
      if (i_paObject)
      {
        free(i_paObject);
        --sm_nCounter;
      }
    }

    static int sm_nCounter;

};
```

Die Zähler kann nun bei jedem Allozieren inkrementiert und entsprechend beim Freigeben wieder dekrementiert werden. Da der Zähler als Klassen-Variable[1] deklariert ist, wird er von allen Objekten verwendet, die von HeapControl abgeleitet sind..

Ist ein Programm zu Ende, zeigt der Wert des Counters an, wie viele Objekte noch nicht mit delete gelöscht wurden.

```
void Test()
{
  A* paA = new A();
  A* paB = new B();
  C* paC = new C();

  delete paC;

  cout << "LeakHunter: " << HeapControl::sm_nCounter
       << " leaks detected!" << endl;

}
```

Noch ein Wort zu Null-Pointers bei new und delete. Wird von new null zurückgeliefert, so wird der Konstruktor-Code vom Compiler sinnvollerweise nicht ausgeführt. Das bedeutet für uns, dass wir den Zählerwert auch nicht inkrementieren dürfen.

Wird delete auf einem null Pointer aufgerufen, so ist dies eine legale Operation. Der Operator ist selber dafür zuständig, abzuchecken, ob der Pointer null ist. Das Aufrufen des Operators delete auf einem Null-Pointer hat ebenfalls keine Wirkung. Deshalb ist auch hier ein Dekrementieren nicht erlaubt.

Viele Programmierer checken den Pointer vor dem delete Aufruf auf null. Dies generiert nur Laufzeit und ist nicht nötig.

[1] Der als statc deklarierte Counter

# Feststellen der Anzahl Leaks improved

Der versierte OO-Programmierer wird sich wohl gleich an der als public deklarierten Zähler-Variablen gestört haben. In einem weiteren Schritt soll diese Unschönheit[2] beseitigt werden. Gleichzeit führen wird einen HeapMgr ein, welcher sowohl für das Allozieren und Freigeben von Speicher, sowie das Zählen von allozierten und freigegebenen Speicherblöcken zuständig ist.

Der HeapMgr ist Singleton, so dass er einfach referenzierbar ist.

```
class HeapMgr
{
  public:

    static HeapMgr& instance()
    {
      if (!ms_paInstance)
      {
        ms_paInstance = new HeapMgr();
      }
      return *ms_paInstance;
    }

    void* alloc(size_t i_nSize)
    {
      void* paObject = malloc(i_nSize);
      if (paObject)
      {
        ++m_nCounter;
      }

      return paObject;
    }

    void free(void* i_paObject)
    {
      if (i_paObject)
      {
        ::free(i_paObject);
        --m_nCounter;
      }
    }

    void printStatus()
    {
      cout << "LeakHunter: " << m_nCounter << " leaks detected!" << endl;
    }

  private:
    HeapMgr()
    : m_nCounter (0)
    {}

    int m_nCounter;
```

[2] Verletzung des Information-Hiding-Konzeptes

```
    static HeapMgr* ms_paInstance;
};
```

Die HeapControl-Klasse wird nun dementsprechend vereinfacht:

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize)
    {
      return HeapMgr::instance().alloc(i_nSize);
    }

    void operator delete(void* i_paObject)
    {
      HeapMgr::instance().free(i_paObject);
    }
};
```

Durch den obigen Redesign wird es uns möglich, einen für alle Objekte gemeinsamen Status zu verwalten. Ferner könnte auch die Strategie, wie Speicher angefordert wird, noch einmal in einer eigenen Klasse modelliert werden. Im vorliegenden Fall wird der Speicher immer mit malloc reserviert und mit free freigegeben. Programmierer von Embedded-Systems wissen aber, wie wichtig es sein kann, einen eigenen, deterministischen MemoryMgr einzusetzen. Über ein entsprechendes Interface könnte der HeapMgr eine adäquate Strategie[3] aggregieren.

[3] Siehe auch Strategy Pattern in Design Patterns von E. Gamma et al.

# Wo wird Speicher alloziert

Dem Operator new wird standardmässig die Anzahl zu Allozierende Bytes als Parameter übergeben. Nun ist es aber auch möglich, beliebig viele benutzerdefinierte Parameter zu übergeben[4].

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize, const char* i_pcFile, int i_nLine);
    void operator delete(void* i_paObject);
};
```

Der Aufruf gestaltet sich dann aber eher unnatürlich, was die Syntax betrifft. Gegeben sei die Klasse A, die im folgenden instanziert werden soll.

```
class A : public HeapControl
{
  int m_nInt;
};

// Instantiate an object of class A
A* paA = new ("Hello World", 1000) A();
```

Die Parameterliste wird also zwischen dem Keyword new und dem Klassennamen übergeben.

Um festzustellen, wo Speicher alloziert wurde, muss dem Operator new lediglich der Dateinamen und die Zeilennummer mitgegeben werden. Glücklicherweise stellt uns der Präprozessor die vordefiniertern Makros __FILE__ und __LINE__ zur Verfügung, welche genau diese Informationen beinhalten.

| Makro | Type | Beschreibung |
|---|---|---|
| __FILE__ | const char* | Der Name der Datei, in welcher die __FILE__ Anweisung steht |
| __LINE__ | int | Die Zeilennummer, in welcher die __LINE__ Anweisung steht |

Somit können wir schreiben:

```
// Instantiate an object and pass location information
A* paA = new (__FILE__, __LINE__) A(....);
B* paB = new (__FILE__, __LINE__) B(....);
C* paC = new (__FILE__, __LINE__) C(....);
```

Interessant dabei ist, dass die Anweisung `new (__FILE__, __LINE__)` immer gleich ist. Es liegt also nahe, für diesen Teil ein Makro zu definieren.

```
#define TEST new (__FILE__,__LINE__)
// Instantiate an object and pass location information
A* paA = TEST A(....);
```

[4] Achtung: dies ist nur bei new, nicht aber bei delete möglich.

Nun drängt es sich förmlich auf, das Makro anstelle von Test new zu nennen. Durch diesen Trick wird es möglich, die herkömmliche Syntax weiterhin zu verwenden.

```
#define new new (__FILE__,__LINE__)
// Instantiate an object and pass location information
A* paA = new A(....);
B* paB = new B(....);
C* paC = new C(....);
```

Diese Methode ist sehr bequem, da sich der Programmierer keine Gedanken darüber machen muss, wie und ob die Location-Info übertragen wird.

Hier sei aber ein Wort der Warnung angebracht: wird nämlich das Keyword new durch new (__FILE__,__LINE__) ersetzt, so muss garantiert werden, dass jede Klasse über einen Operator new mit entsprechender Signatur verfügt. Es mag sein, dass das bei der eigenen Software noch garantiert werden kann. Aber spätestens dann, wenn Drittprodukte[5] eingesetzt werden, fangen die Probleme an.

Es wird deshalb vorgeschlagen, ein eigenes Keyword zu wählen.

```
#define B_NEW new (__FILE__,__LINE__)
// Instantiate an object and pass location information
A* paA = B_NEW A(....);
B* paB = B_NEW B(....);
C* paC = B_NEW C(....);
```

Somit wird für alle Objekte der eigenen Software-Library der neue Operator B_NEW, für alle zugekauften Produkte weiterhin new verwendet. Dies ist zwar nicht so elegant, in grossen Systemen aber der einzig gangbare Weg.

Man kann sich leicht vorstellen, dass die gezeigte Variante sowohl einen gewissen Laufzeitoverhead und einen beträchtlichen Speicherbedarf nach sich zieht.

Laufzeitoverhead deshalb, weil die zwei zusätzlichen Parameter übergeben werden müssen. Der Speicheraufwand kommt daher, weil der Präprozessor für jede __FILE__ Anweisung eine neue Zeichenkette generiert und diese ins Code-Segment ablegt.

Um dies im Release-Mode zu unterbinden, genügt eine einfache Ergänzung. Wie bei jedem anderen Operator, erlaubt auch new das Überladen von Funktionen. Das heisst, dass mehrere Funktionen denselben Namen haben können, wenn sie sich nur in der Parameterliste unterscheiden.

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize);
    void* operator new(size_t i_nSize, const char* i_pcFile, int i_nLine);
    void operator delete(void* i_paObject);
};
```

[5] MS MFC verwendet ein ähnliches Konzept, um MemoryLeaks aufzuspüren. Das Keyword new wird auch dort umdefiniert. Wird das in einer eigenen SW-Library auch nochmals getan, führt dies unweigerlich zu Problemen.

Wird mehr als ein Operator new mit unterschiedlichen Parameter deklariert, so können auch beide verwendet werden.

Die Idee ist nun, den Operator B_NEW abhängig vom Compile-Mode (Debug, Release) zu definieren.

```
#ifdef _DEBUG
  #define B_NEW new (__FILE__,__LINE__)
#else
  #define B_NEW new
#endif
```

Kompiliert der Code nun im Release-Mode, so wird bereits schon vom Präprozessor der Standard-Operator new eingesetzt. Der Compiler sucht sich nun den entsprechendem Operator aus. Aufgrund der Parameterliste wird die Version operator new(size_t) ausgewählt, womit sämtlicher Overhead eliminiert werden kann.

# Wo wurde der Speicher alloziert, der freigegeben wird

Nur zu wissen, wo der Speicher alloziert wird, ist wohl eher von akademischem Interesse. Natürlich interessiert es vielmehr, wo der Speicher alloziert wurde, wenn er wieder freigeben wird. So lassen sich zumindest Rückschlüsse daraus ziehen, welche Speicherblöcke nur alloziert, nicht aber freigeben wurden.

Um die Information, wo der Speicher alloziert wurde, auch im Operator delete zur Verfügung zu haben, muss diese während der Lebzeit des Objektes irgendwo zwischengespeichert werden.

Wo sollen aber nun diese 8 Bytes (const char* und int) abgelegt werden. Die offensichtliche Antwort ist wohl als Member-Variable in der Klasse HeapControl:

```
class HeapControl
{
  public:
    void* operator new(size_t i_nSize);
    void* operator new(size_t i_nSize, const char* i_pcFile, int i_nLine);
    void operator delete(void* i_paObject);

    const char* m_pcFile,
    int m_nLine;
};
```

Das primäre Ziel des LeakHunters ist es aber, die bestehende Software so wenig wie möglich zu beeinflussen Würde ein Anwender den Operator sizeof auf einem Derivat von HeapControl anwenden, so wäre das Ergebnis unterschiedlich, je nachdem ob die Leakverfolgung ein- (Debug-Mode) oder ausgeschaltet (Release-Mode) ist.

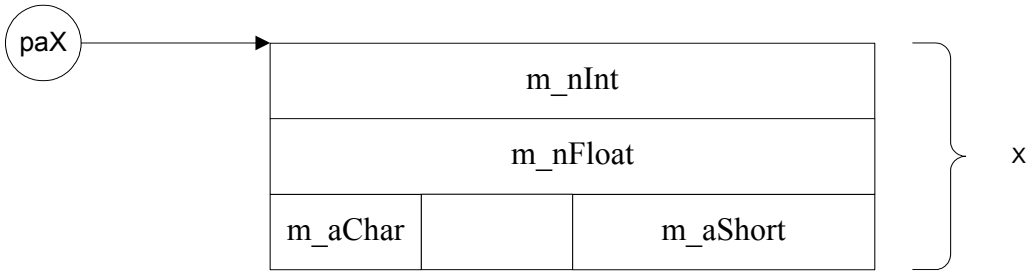
Das Ziel ist es demnach, etwas wie Membervariablen zu deklarieren, ohne dabei jedoch die Klassendeklaration zu verändern. Dies mag seltsam klingen, ist aber möglich.

Führen wir uns einmal die Funktionalität des HeapMgr's vor Augen. Dieser hat nicht vielmehr zu tun, als die geforderte Anzahl Bytes zu reservieren.

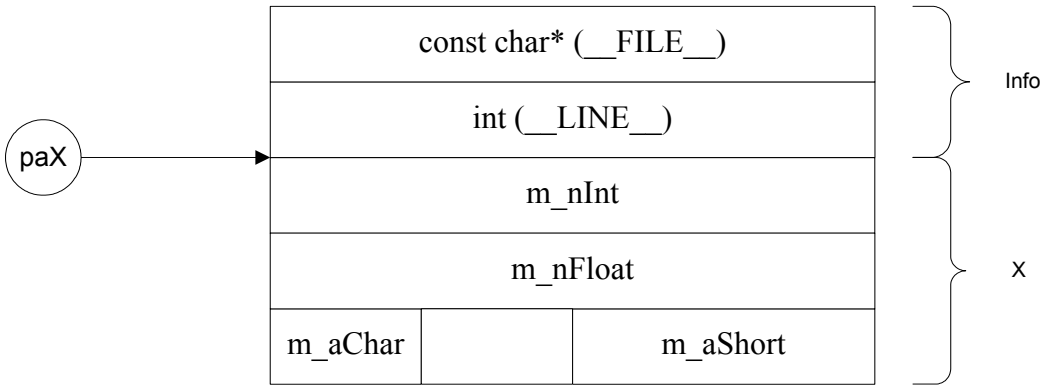
z.B. für eine Klasse X:

```
class X : public HeapControl
{
  int m_nInt;
  float m_nFloat;
  char m_aChar
  short m_aShort;
};
```

X* paX= B_NEW X();




Es ist nun ein leichtes für uns, jeweils 8 Bytes mehr zu reservieren, und zwar am Anfang des geforderten Blocks.




Der Wert des Pointers, welcher der operator new dem User liefert, zeigt selbstverständlich immer noch auf das Objekt der Klasse X. Dass vor diesem Speicherbereich nochmals 8 Bytes reserviert werden, ist sowohl für den Anwender als auch für den Compiler komplett transparent!

Damit die __FILE__ und __LINE__ Information einfach zu bearbeiten ist, definieren wird eine Klasse Info, welche über den entsprechenden Speicherbereich gelegt werden kann.

```
class Info
{
  public:
    const char* m_pcFile;
    int         m_nLine;
};
```

Die alloc-Methode des HeapMgr's wird so angepasst, dass zusätzlicher Speicher für die Info-Struktur zur Verfügung steht und die entsprechenden Daten abgefüllt werden.

Die Info-Struktur wird dabei als private Inner-Class modelliert, da sie ausserhalb des HeapMgr's keine Verwendung findet. Zu beachten ist auch, dass zum Pointer auf das jeweilige Objekt im return-Statement der Offsetwert sizeof(Info) hinzuaddiert wird.

```
class HeapMgr
{
  private:

    class Info
```

```
    {
      public:
        const char* m_pcFile;
        int m_nLine;
    };

  public:

    ....

    void* alloc(size_t i_nSize, const char* i_pcFile, int i_nLine)
    {
      void* paObject = malloc(i_nSize+sizeof(Info));
      if (paObject)
      {
        ++m_nCounter;
        Info* paInfo = (Info*)paObject;
        paInfo->m_pcFile = i_pcFile;
        paInfo->m_nLine  = i_nLine;
        cout << "LeakHunter: " << i_nSize << " bytes allocated @ "
             << paInfo->m_pcFile
             << ":" << paInfo->m_nLine << endl;
      }
      return (char*)paObject+sizeof(Info);
    }
    ....
};
```

Analog dazu wird die free-Methode codiert. Da jedem Objekt, das kreiert wurde, der Info-Block am Anfang des Speichersegmentes zugefügt wurde, ist dieser garantierterweise auch noch vorhanden, wenn das Objekt wieder gelöscht wird. Vom Wert des Objekt-Pointers, welcher der Methode free als Parameter übergeben wird, muss also lediglich wieder der Offsetwert sizeof(Info) subtrahiert werden, um an die gewünschten Daten zu kommen.

```
class HeapMgr
{
  public:

    ....

    void free(void* i_paObject)
    {
      if (i_paObject)
      {
        Info* paInfo = (Info*)((char*)i_paObject-sizeof(Info));
        cout << "LeakHunter: " << "Object freed @ " << paInfo->m_pcFile
             << ":" << paInfo->m_nLine << endl;
        ::free(paInfo);
        --m_nCounter;
      }
    }
    ....
};
```

Die gezeigte Version des LeakHunters's ist in der Lage, sowohl während dem Kreieren als auch während dem Destruieren von Objekten den genauen Ort der Speicher-Allokation zu bestimmen.

Dies hilft uns insofern weiter, als dass es möglich wird, zu jedem Anfordern von Speicher das entsprechende Freigeben zu suchen.

```
LeakHunter: 4 bytes allocated @ LeakHunter.cpp:374
LeakHunter: 8 bytes allocated @ LeakHunter.cpp:375
LeakHunter: 8 bytes allocated @ LeakHunter.cpp:376
LeakHunter: Object freed @ LeakHunter.cpp:376
LeakHunter: 2 leaks detected!
```

Dieses einfache Beispiel zeigt, dass wir sowohl auf Zeile 374 als auch auf Zeile 375 ein Memory-Leak haben, da dieser Speicher nicht mehr freigegeben wurde.

Diese Methode mag ja ein gangbarer Weg für Schulbeispiele sein. Stellt man sich aber vor, obige Arbeit für eine Maschinensteuerung zu machen, die nur alle 5 Wochen ein Memory-Leak produziert, kommt man zum Schluss, dass die gezeigte Lösung im professionellen Umfeld noch immer unbrauchbar ist.

# Automatisiertes Detektieren von Memory-Leaks

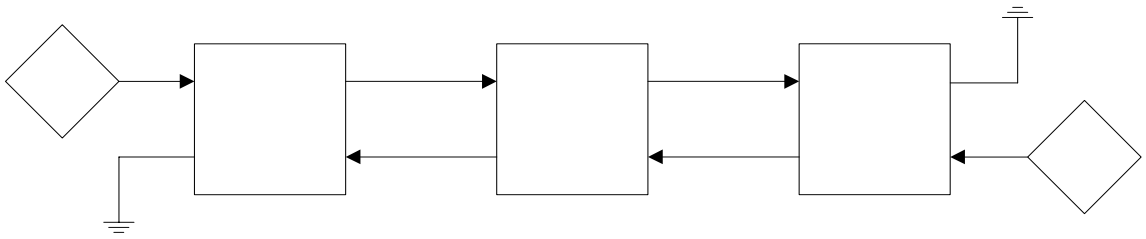
Um Memory-Leaks komplett automatisiert detektieren zu können, ist beim Shutdown einer Software eigentlich nur eine Information wichtig: Wo steht das new Statement, welches schlussendlich ein Memory-Leak verursacht hat.

Um diese Aussage machen zu können ist es wichtig, dass unser System ständig darüber Bescheid weiss, ob ein allozierter Block zurückgegeben wurde oder nicht. Um dieses Wissen zu erlangen, muss ständig Buch über alle momentan allozierten Blöcke geführt werden.

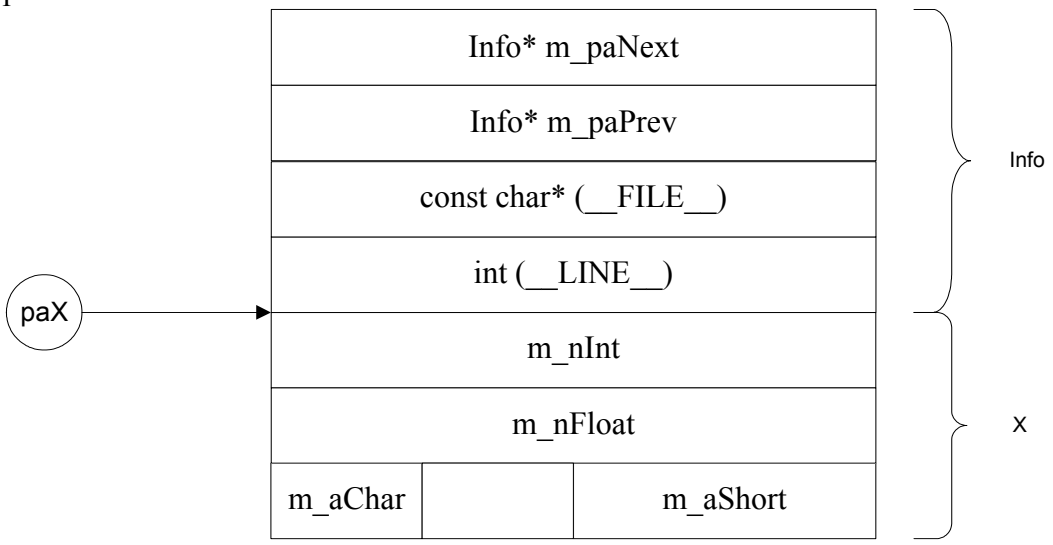
Die Aufgabe, die sich nun bietet, ist dieser Anforderung mit einer möglichst hoch-performanten Lösung nachzukommen.

Dabei soll sowohl das Einfügen in die Liste (operator new) als auch das entfernen (operator delete) in Ordnungsrelation O(1) geschehen. Die einzige Datenstrukur, die das erlaubt ist eine Doubly-Linked-List, kurz DList.

Um ein Element in eine DList einfügen zu können, muss es sowohl über einen Next-, als auch über einen Previous-Pointer verfügen, um das nächste, respektive das vorhergehende Element referenzieren zu können. Das Einfügen beziehungsweise Entfernen von Elementen aus einer DList soll hier nicht genauer beschrieben werden; es lässt sich in jedem Schulbuch nachlesen.



Es liegt Nahe, die zwei Pointer zum Verlinken der Elemente ebenfalls in der Info-Struktur zu platzieren:



Jedes Element, welches alloziert wird, muss nun gleichzeitig auch in die DList eingetragen werden. Analog dazu wird jedes Element, welches wieder freigegeben wird aus der DList entfernt. Alle Blöcke, welche sich am Ende des Programmes noch in der Liste befinden, sind Memory-Leaks.

Nebst dem Wissen, wie viele Leaks vorhanden sind, lässt sich genau feststellen, wo diese aufgetreten sind. Nachfolgend noch die Modifikationen am HeapMgr.

Da der HeapMgr auch als DList fungiert, braucht er sowohl einen Start- als auch einen End-Pointer, um die Verwaltung der Elemente wahrnehmen zu können.

```
class HeapMgr
{
    ....
  private:
    ....

    Info* m_paFront;
    Info* m_paBack;
};
```

Ebenfalls entsprechend angepasst wird die private Info Klasse:

```
class Info
{
  public:
    const char* m_pcFile;
    int         m_nLine;
    Info*       m_paNext;
    Info*       m_paPrev;
};
```

Das Allozieren und Freigeben von Speicherblöcken wird nun von der Algorithmik der DList dominiert. Wichtig beim Einfügen und Entfernen von Blöcken in eine Doubly-Linked-List ist, dass unterschieden werden muss, ob sich bereits Elemente in der Liste befinden oder nicht.

```
class HeapMgr
{
  public:

    ....

    void* alloc(size_t i_nSize, const char* i_pcFile, int i_nLine)
    {
      void* paObject = malloc(i_nSize+sizeof(Info));
      if (paObject)
      {
        Info* paInfo = (Info*)paObject;
        paInfo->m_pcFile = i_pcFile;
        paInfo->m_nLine  = i_nLine;

        if (m_paFront)
        {
          paInfo->m_paPrev = m_paBack;
          paInfo->m_paNext = 0;
```

```
          m_paBack = paInfo;
          paInfo->m_paPrev->m_paNext = paInfo;
        }
        else
        {
          m_paFront = paInfo;
          m_paBack  = paInfo;
          paInfo->m_paNext = 0;
          paInfo->m_paPrev = 0;
        }

      }
      return (char*)paObject+sizeof(Info);
    }

    void free(void* i_paObject)
    {
      if (i_paObject)
      {
        Info* paInfo = (Info*)((char*)i_paObject-sizeof(Info));
        if (paInfo->m_paPrev)
        {
          paInfo->m_paPrev->m_paNext=paInfo->m_paNext;
        }
        else
        {
          m_paFront = paInfo->m_paNext;
        }

        if (paInfo->m_paNext)
        {
          paInfo->m_paNext->m_paPrev=paInfo->m_paPrev;
        }
        else
        {
          m_paBack = paInfo->m_paPrev;
        }        paInfo->m_paNext = 0;
        paInfo->m_paPrev = 0;
        ::free(paInfo);
      }
    }

    ....
};
```

Die Ausgabe der vorhandenen Memory-Leaks ist grundsätzlich ein Traversieren der Elemente in der Doubly-Linked-List:

```
class HeapMgr
{
  public:

    ....

    void printStatus()
    {
      if (m_paFront)
```

```
      {
        int nCounter=0;
        cout << "Reporting: " << endl;
        Info* paInfo = m_paFront;
        while (paInfo)
        {
          cout << "LeakHunter: " << "Object lost @ " << paInfo->m_pcFile
               << ":" << paInfo->m_nLine << endl;
          paInfo = paInfo->m_paNext;
          ++nCounter;
        }
        cout << "LeakHunter: " << nCounter << " leaks detected!" << endl;
      }
    }
    ....
};
```

Als Ausgabe werden nur noch die tatsächlich vorhandenen Leaks erwartet:

```
Reporting:
LeakHunter: Object lost @ LeakHunter.cpp:535
LeakHunter: Object lost @ LeakHunter.cpp:536
LeakHunter: 2 leaks detected!
```

# Schlussüberlegungen

Mit der gezeigten Version des LeakHunters lässt sich sehr gut arbeiten. Kontroll-Mechanismen in der gezeigten Art wurden von uns bereits in vielen Software-Systemen integriert.

Jedes Programm, welches nicht eine direkte und ständige menschliche Interaktion zu erwarten hat und über lange Zeit laufen muss, sollte mit einem Mechanismus zur Prüfung von Memory-Leaks ausgestattet sein.

Es ist einfach vorstellbar, dass Memory-Leaks erst in einem gewissen Zustand des Programmes auftreten. Aus Erfahrung lässt sich sagen, dass dies meist die Extrem-Situationen sind, in denen Meldungs-Queues überlaufen oder Timeouts generiert werden. Gerade aber diese Zustände sind es, die sich im Labor-Umfeld kaum oder nur sehr schwer testen lassen (und auch nicht reproduzierbar sind). Es ist daher sehr viel wert, die Information über eventuelle Memory-Leaks immer in ein Trace-File zu schreiben.

Wichtig zu bemerken an dieser Stelle ist natürlich, dass Memory-Leaks nur festgestellt werden können, wenn für eine Software ein wohldefinierter Shutdown-Zustand existiert, in welchem die Auswertung vorgenommen werden kann. Es ist per Definition nicht möglich, in einer laufenden Software auf Leaks zu prüfen.

Gerade in der Programmierung von Embedded-Systems ist ein Shutdown aber nicht immer vorgesehen, da das Ausschalten der Maschine meist auch der Steuerung den Strom abdreht. Trotzdem sei an dieser Stelle bemerkt, dass das kontrollierte Herunterfahren von Software sehr viel über deren Qualität aussagt.

Hat ein Programmierer bereits schon einmal eine korrekte Shutdown Sequenz für ein Multi-Threading-System mit bidirektionaler Interaktion zwischen den einzelnen Entitäten entwickelt, so weiss er, wovon hier die Rede ist. Ein Shutdown-Prozess zu codieren ist ungleich komplizierter als der Startup!

Noch schwieriger ist es, Grundlagen-Software, also ein Framework zu bauen, dass die Anforderung für einen kontrollierten Shutdown-Prozess erfüllt.

Doch nur in diesem Shutdown-Zustand kann - wie schon erwähnt - auf Memory-Leaks getestet werden. Aus diesem Grund ist diese Funktionalität in jeder Qualitäts-Software (Flugzeug, Bahn, Maschinen) ein Muss.

Auf die Frage, ob ein Programm noch Memory-Leaks aufweist kann der Entwickler dann mit gutem Gewissen folgendermassen antworten: „Nein!“

# Literatur

**Gamma, Erich et al.: Design Patterns. Addison Wesley, 1995. ISBN 0-201-63361-2.**
Ein Standardwerk, das in keiner Bibliothek fehlen sollte. Vor allem als Nachschlagewerk und Lebensgefährte gedacht. Das beste Buch auf seinem Gebiet! Sehr empfehlenswert für jedermann.

**Stroustrup, Bjarne: The C++ Programming Language, Third Edition. Addison Wesley, 1997. ISBN 0-201-88954-4.**
Die beste C++ Referenz. Achtung: Nur die dritte Edition kaufen. Exzellente Einführung in STL. Sehr empfehlenswert für jedermann.

**Lippmann, Stanley B., Lajoie, Josée: C++ Primer, Third Edition. Addison Wesley, 1998. ISBN 0-201-82470-1.**
Ebenfalls ein sehr gutes C++ Buch. Einfacher zu verstehen als das Buch Stroustrup. Auch hier: Nur die dritte Edition kaufen. Gute STL Referenz

**Lippmann, Stanley B: Inside the C++ Object Model. Addison Wesley, 1996. ISBN 0-201-83454-5.**
Ein geniales und einfach geschriebenes Buch, für alle die wissen wollen, wie C++ intern funktioniert. Vom "virtual call" bis hin zu "multiple inheritance RTTI".

**Coplien, James O.: Advanced C++ Programming Styles and Idioms. . Addison Wesley, 1992. ISBN 0-201-54855-0.**
Ein Buch mit weiterführenden Konzepten. Zum Teil hochtechnische und sehr schwierig zu verstehende Lösungsansätze. Die Perle unter den C++ Büchern.

**Meyers, Scott: Effective C++, Second Edition. Addison Wesley, 1998. ISBN 0-201-92488-9.**
**Meyers, Scott: More Effective C++. Addison Wesley, 1998. ISBN 0-201-63371-X.**
Zwei Bücher, die sich mit dem korrekten Schreiben von C++ Programmen befassen. So zum Beispiel, wie man Assignment Operators, Copy Constructors und deren mehr korrekt implementiert. Sehr empfehlenswert für jedermann.

**Stroustrup, Bjarne: The Annotated C++ Reference Manual. Addison Wesley, 1995. ISBN 0-201-51459-1.**
Die C++ Referenz für Compilerbauer. Geht sehr tief und ist mit vielen Bemerkungen versehen.