有效的C++内存泄露检测方法

邵叶秦
（南通大学 现代教育技术中心，江苏 南通 226001）

摘要：提出了一个有效的C++内存泄漏检测方法。方法在分析内存泄漏的基础上，通过重新实现动态内存分配和释放函数，记录内存分配的确切位置并跟踪动态内存的使用情况，在程序结束时，方法利用跟踪结果检测和定位内存泄漏。最后，通过在Windows和Linux平台上的实验验证了本文方法的有效性。

关键词：内存泄漏 动态内存 泄漏检测
中图分类号：TP314　　文献标识码：A　　文章编号：1009-3044(2007)14-30455-02

Effective Method for Memory leak Detection in C++

SHAO Ye-qin
(Modern Education Technology Center, Nantong university, Nantong 226001, China)

Abstract: The paper presents an effective method for the detection of memory leak. Based on the analysis of memory leak, the method overloads the allocation function and overrides the free function to store the allocation spot and trace the dynamic memory allocation. At the conclusion of the program, the method detects and locates the memory leak depending on the tracing result. Finally, an experiment on the Windows and Linux platforms comes up and proves the effectiveness of the proposed method.

Key words: memory leak; dynamic memory; leak detection

内存管理是计算机科学中的一个重要的领域，而内存泄漏的处理是内存管理中的一个重要研究课题。内存泄露是指由动态分配的内存在使用结束后，由于某种原因没有被及时释放，使得这些动态内存不能再被程序使用，导致可用内存数量减少，甚至耗尽。内存泄漏直接影响程序的稳定性和可用性。

为了解决内存泄漏问题，业界出现了不少相关的技术。一些高级语言通过垃圾收集(Garbage Collection)机制跟踪内存的使用情况，并自动回收不再使用的内存。这减轻程序员了程序员的负担，但垃圾收集不能彻底解决内存泄露问题，而且目前C++不支持这一机制。也有一些现成的内存泄露检测工具，它们虽然检测的范围较广，能检测其它的资源泄露，但它们的运行花费较大，而且不能处理各种内存泄漏问题。为此，本文给出了一种有效的C++内存泄露检测方法，不仅能处理常见的内存泄漏问题而且具有平台独立性。

1 内存泄漏的种类

内存泄漏是由于动态分配的内存没有及时、正确的释放而引起的。按发生的原因，内存泄漏主要可以分成以下几类：

(1)显式内存泄露

动态分配的内存在运行过程中没有调用相关的函数进行释放，但指向该内存的指针却被指向其它内存或被销毁，使得这块内存丢失。程序逻辑复杂、发生异常或程序员的疏忽等都会导致显式内存泄漏。

(2)隐式内存泄露

在程序退出后已经分配内存都会被释放，但在程序运行过程中，内存使用会不断增加，甚至被耗尽。程序设计不良，在运行过程中动态分配的内存没有被及时释放，直到程序退出时才统一回收，就会导致隐式内存泄露。

2 本文的内存泄露检测方法

为了有效地检测内存泄露，本文方法监视动态内存的分配和释放。方法通过改写负责内存分配和释放的new和delete操作符来监控内存使用情况。当new被调用时，在一个保存内存使用情况的列表中新增一项，并记录新分配内存的信息；当delete被调用时，在列表中删除对应的记录项。在程序结束时，列表中剩下的就是泄漏的内存，通过打印相关信息就可以检测和定位内存泄露。

2.1 方法的数据结构

为了记录内存的使用情况，本文需要一个全局列表。考虑到程序的通用性和方法实现的效率，本文没有采用现成的容器模板，而是直接实现了一个使用链地址法解决冲突的哈希表类，并说明了一个全局的哈希表对象HashMem，用指向动态内存的指针作为哈希表的键，记录各块内存的使用信息，如下图。

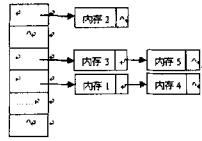

图1　记录内存使用信息的哈希表

其中的每一块内存按下图结构保存信息。

图2　本文方法分配的内存结构

其中，MIB是结构MemInfoBlock的缩写，记录了动态内存相关的信息。

```
Struct MemInfoBlock
{
 Void *key;                    // hash 表的键
 Char file[MAX_FILE_NAME];     // 分配发生的文件名
 Int  line;                    // 分配发生的行号
 Size_t size;                  // 用户要求分配的内存大小
 MemInfoBlock * next;          // 解决 hash 冲突的链表指针
};
```

Ptr0和Ptr是两个指针。Ptr0指向本文方法实际分配的内存空间。Ptr是指向用户需要的内存空间。在new函数结束时，本文方法只返回用户需要的那部分内存。

为了检测隐式内存泄露，本文需要统计程序运行过程中的动态内存使用情况，所以还需要一个全局的哈希表HashStat。它以动态内存分配函数所在的位置为哈希表的键，以该处请求的内存总量为内容，直接统计各处调用的内存使用量。每次分配内存时，新分配的内存大小累加到现有的内存使用量中；每次释放内存时，释放的内存大小从现有的内存使用量中减去。通过创建一个单独的进程Pm，本文方法周期性遍历HashStat并将信息输出到指定的内存跟踪文件，以便分析。

收稿日期：2007-06-03
作者简介：邵叶秦(1979-)，男，浙江海宁人，江苏南通大学教师，硕士，研究方向：程序设计、嵌入式软件开发

2.2 方法的实现

C++语言通过 Operator new 和 operator delete，Operator new [] 和 operator delete[]来完成动态内存的分配和释放。为了监视动态内存的使用情况，本文方法重载了 new 操作符，但 delete 操作符不能被重载，所以直接覆盖 delete 原来的实现。new 操作符的重载形式如下：

void* operator new(size_t nSize, const char* pszFileName, int nLineNo)

void* operator new[](size_t nSize, const char* pszFileName, int nLineNo)

由于内存泄漏比较隐蔽，一般在发生之初没有异常的表现，因此内存泄漏的定位比较困难。为了准确的找到发生泄漏的位置，本文方法在重载的 new 操作符时，设置了 pszFileName 和 nLineNo 参数来接收动态分配函数所在的源文件名和行号，并将其保存在内存信息块中。

为了得到上述的源文件名和行号，本文方法自定义了下面两个宏，并通过编译，把原始的 new 操作符转化为重新实现的 new，同时利用两个预定义宏 __FILE__ 和 __LINE__，获得动态分配函数所在的源文件名和行号，并作为参数传入重载的 new。

```
#define new DIAGNOSE_NEW
#define DIAGNOSE_NEW new(__FILE__, __LINE__)
```

经过重新实现，new 除了分配需要的内存外，还要填充相关的信息并把内存使用情况添加到 HashMem 和 HashStat 中，以表示新的内存被分配。Delete 除了释放相应得内存外，还要从 HashStat 减去相应的内存空间，从 HashMem 中删除对应的项，以表示相应的内存被释放。它们的具体实现如下：

new 操作符的实现如下：

(1)计算实际需要的内存大小，其中包括 MemInfoBlock；

(2)分配所需的内存空间；

(3)按照图 2，在 MemInfoBlock 中记录动态内存的相关信息；

(4) 以动态内存的指针为键把新分配的内存保存在哈希表 HashMem；

(5)以动态分配所在的文件名和行号为键，将本次分配的内存大小记入哈希表 HashStat 中；

(6)返回用户所需内存的指针 ptr。

delete 操作符的实现如下：

(1)根据指针参数 ptr 计算 ptr0；

(2)根据 ptr0 在哈希表 HashMem 中定位要释放的动态内存；

(3)如果没有找到，记录一个错误信息并返回；

(4)把要释放的内存大小从 HashStat 中去除；

(5)从 HashMem 中删除要释放的内存；

(6)释放相应的内存。

在程序运行过程中，进程 Pm 周期性的将表中的内存使用情况输出到特定的内存跟踪文件中，用于分析内存泄漏。

代码执行完成时，程序中要释放的内存都已释放，HashStat 表中只留下泄漏的内存。本文方法通过遍历 HashStat 表把这些内存的信息输出到内存跟踪文件，以便用户发现内存泄漏。

2.3 方法的结果分析

经过检测，本文方法最终产生一个内存跟踪文件(如图 3)，它保存着在程序运行过程中通过周期性的遍历 HashStat 表而输出的内存使用情况。

图中每一项输出表示某个文件的某一行当前分配了多少内存。通过分析所有的输出，用户可以总体上了解内存的动态变化情况，从而发现隐式内存泄漏(如果某个位置分配的内存持续不断增加，就可能存在隐式内存泄漏)。通过分析最后一次输出，用户则可以发现显式内存泄漏(如果最后一次输出使用量为 0，表明没有显示内存泄漏)。

```
内存跟踪文件的内容如下：
第一次遍历输出：
  1.1 A.cpp [14]------512 allocated
  1.2 B.cpp [34]------32 allocated

第二次遍历输出：
  2.1 A.cpp [14]------678 allocated
  2.2 B.cpp [34]------16 allocated
  ……
程序结束时最后一次遍历输出：
  n.1 A.cpp [14]------128 allocated
  n.2 B.cpp [34]------4 allocated
```

图 3 内存跟踪文件内容举例

3 实验

为了验证本文方法的有效性，我们用标准的 C++实现了本文的内存泄漏检测方法，并在 Window 和 Linux 平台上，利用下面两种情况分别对两类内存泄露进行实验。

(1)动态分配一个 char 数组，但不释放空间；

(2)建立一个大小为 256 的指针数组，通过一个循环每隔 5 秒钟动态分配一个 8192 个字节的空间并把首地址记录在指针数组中。在数组满后，释放指针数组中记录的所有内存。

由于篇幅原因，这里仅以检测第二类内存泄露为例。

代码如下

```
// Test2.cpp
  Void * pList[256];
  For (int i=0; i<256; i++)
{ pList[i] = new char[8192];
//Do something
sleep(5000); }
For (int i=0; i<256; i++)
{ delete [] pList[i]; }
```

内存跟踪文件的输出结果如下：

```
1 Test2.cpp[4]------8192 allocated
2 Test2.cpp[4]------16384 allocated
3 Test2.cpp[4]------24576 allocated
……
256 Test2.cpp[4]------2097152 allocated
257 Test2.cpp[4]------0 allocated
```

从输出的结果可知，在程序运行过程中，Test2.cpp 文件第 4 行代码分配的内存一直在不断增加。但在程序结束时，所有的内存全部被释放(只有 0 字节被分配)，所以这段代码存在隐式内存泄漏。对于其它的泄露类型，通过分析最后一次遍历输出，本文方法也都能正确的检测和定位内存泄露。

4 结论

内存泄露一直是 C++程序员面临的一个难点问题，本文针对内存泄露的实际情况提出了一种有效的检测方法。结果表明，本文方法能有效的检测常见内存泄漏并准确定位泄漏的位置。

参考文献：

[1]Steve Maguire. Writing Solid Code[M]. Microsoft Press, 1993.

[2]Bjame Stroustrup. The Design and Evolution of C++[M]. Addision-Welsley, 1994.

[3]N Nethercote, J Seward. Valgrind: a program supervison framework[C] In: O Sokolsky, M Viswanathan eds. Electronic Notes in Theoretical Computer Science, Elsevier, 2003, 89

[4]Scot Meyers 著，侯捷，译.More Efective C++[M].中国电力出版社，2003.

(上接第 425 页)

各种电路，对电路进行仿真和测试。只有熟悉它，不断分析和总结，才能应用自如。

参考文献：

[1] 阎石. 数字电子技术基础(第五版)[M]. 北京：高等教育出版社，2006.

[2] 周凯一，郝文俊，RWB 虚拟电子实验室——Multisim 7 & Ultiboard 7 电子电路设计与应用 [M]. 北京：电子工业出版社，2006.

[3] 孙晓艳，黄萍. 基于 Multisim 的电子电路课堂教学[J]. 微电子技术，2006，24:142-144.

[4] 陈志贵，郭隐彪. Multisim 在数字电子设计中的应用技巧[J]. 机电产品开发与创新，2007，20(2):148-149.