文章编号:1671 - 251X(2009)07 - 0019 - 03

一种有效的 C + + 内存泄漏自检测方法 *

孙凌宇[1], 冷 明[1,2], 夏洁武[1]

(1. 井冈山大学计算机科学系,江西 吉安 343009; 2. 上海大学计算机工程与科学学院,上海 200072)

摘要:针对内存的动态分配和释放特点,文章提出了一种有效的 C + + 内存泄漏自检测方法,给出了该方法的对象行为结构模型以及内存动态分配和释放的具体实现。基于顺序存储数据结构的实验及分析表明,该方法不仅能正确地动态分配和释放类对象和数组块内存,还能监视所运行的程序中是否存在内存泄漏。该方法已经成功应用于算法设计和系统实现中,具有简单、速度快的优点。

关键词:动态内存; C + + 语言; 内存泄漏; 分配; 释放; 自检测

中图分类号:TP312 **文献标识码**:A

0 引言

C + + 动态内存使用技术是 C + + 程序设计员长期探讨的问题。动态内存若使用不当,容易造成内存泄漏(memory leak)。所谓的内存泄漏是指程序在申请获得并使用完动态内存块后,没有释放所申请的动态内存就将保存动态内存地址的变量用于其它用途,使得这些动态内存不能再被程序使用,也无法被操作系统回收[1~3]。常说的内存泄漏一般是指堆内存的泄漏。堆内存是程序从堆中动态分配的、任意大小的存储区,使用完后必须由程序释放。应用程序一般使用 malloc 或 new 函数从堆中分配到 1 块内存,使用完后,必须调用 free 或 delete 函数释放该内存块,否则该内存就不能被再次使用,即造成内存泄漏。

针对内存的动态分配和释放,本文给出了一种 C + + 内存泄漏自检测方法,通过一定的手段检测所运行的程序中是否存在内存泄漏,以便及时发现并纠正错误。该方法已经成功应用于算法设计[4]和系统实现[5]。本文仅基于简单的顺序存储数据结构模型[6~7]阐述该方法的实现步骤。

1 C + + 内存泄漏自检测方法

图 1 为简单的顺序存储数据结构模型的对象类图。

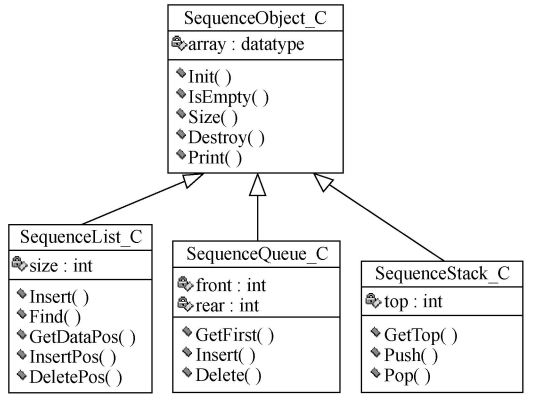

图 1 简单的顺序存储数据结构模型的对象类图

从图 1 可看出,该模型的顺序存储线性表、队列、堆栈类都是基类 SequenceObject_C 的子类。SequenceObject_C 中的指针 array 指向一块地址连续存储区域,使得其逻辑相邻的结点一定在物理上相邻。在该模型中,不仅需要为 3 个子类的对象进行内存分配,还需要为子类成员数组指针 array 指向的连续存储区域分配内存。以下程序的第二行到第七行定义了 6 种存储空间类型。

```
typedef enum E_MType{
    SequenceList_MT,
    SequenceListArray_MT,
    SequenceQueue_MT,
    SequenceQueueArray_MT,
    SequenceStack_MT,
    SequenceStackArray_MT,
}E_ MType;
```

1.1 自检测方法的对象行为型结构

C + + 内存泄漏自检测方法采用如图 2 所示的对象行为型结构。其中,MemoryMonitor_C类负责收集内存分配和释放信息,给出内存使用信息报告,

收稿日期:2009 - 03 - 04

*基金项目:科技部国际合作项目(CB7 - 2 - 01)

作者简介:孙凌宇(1976 -),女,硕士,副教授,现主要从事算法分析与设计、面向对象技术等方面的教学与研究工作。E-mail: lzylmsly @gmail.com

并统计内存泄漏情况;Allocator_C 类作为内存分配器,采用内存池技术,加速内存的分配和释放操作;MemoryContoller_C 类作为内存分配控制器,提供统一的内存分配和释放的接口。

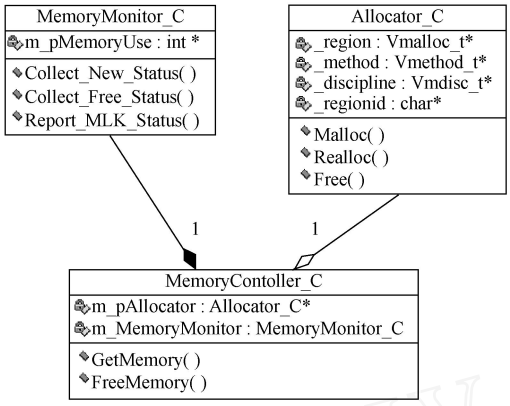

图 2 C++内存泄漏自检测方法的对象行为型结构图

在该行为型结构中,内存分配控制器为单件模式,其源代码如下:

```
MemoryContoller_C * GetGMContoller ()
{
    static MemoryContoller_C g_MemoryContoller;
    return &g_MemoryContoller;
}
class MemoryContoller_C
{
    friend MemoryContoller_C * GetGMContoller();
    protected: MemoryContoller_C();
  ……
}
```

上述代码的第一行到第五行提供一个访问内存分配控制器的全局访问点 GetGMContoller 函数,其中第三行定义 g_MemoryContoller 为静态局部变量,保证系统的内存控制分配器仅有 1 个实例;第八行将 GetGMContoller 全局函数声明为 MemoryContoller_C 类的友元函数;第九行将 MemoryContoller_C 类的构造函数声明为保护类型。

1.2 内存动态分配操作的实现

通常情况下,C++语言中使用 new 操作符分配内存。在本文的自检测方法中,通过重载 new 操作符来监测跟踪所有的内存分配操作,进行内存泄漏的自检测,具体代码如下:

```
void * operator new(size_t size, MType eOper)
{
    MemoryContoller_C * pMMgr = GetGMContoller ();
    void *pvTemp = pMMgr->GetMemory(size, eOper);
    return pvTemp;
}
void free_memory(size_t size,void * block,MType eOper)
{
    MemoryContoller_C * pMMgr = GetGMContoller ();
    pMMgr->FreeMemory(size,block, eOper);
}
```

上述代码的第一行到第六行给出了全局 operator new 的重载代码。1.3 节中代码的第五行为指针 array 动态分配 datatype[100]数组块内存,第十六行为指针 pStack 动态分配 SequenceStack_C 类对象内存。

1.3 内存动态释放操作的实现

以下为调用内存分配和释放操作的源代码:

```
class SequenceStack_C: public Object_C
{
 public:
 SequenceStack_C(){
  this->array = new(SequenceStackArray_MT) datatype[100]; };
 virtual void Destroy()
 {
  free_memory(100,this->array,SequenceStackArray_MT);
  int nSize = this->Size();this->~SequenceStack_C();
  free_memory(nSize,this,SequenceStack_MT);
 };
  ……
}
void main()
{
 SequenceStack_C * pStack;
 pStack = new (SequenceStack_MT) SequenceStack_C();
 pStack->Destroy();
}
```

针对内存的动态释放,1.2 节中代码的第七行到第十一行定义了 free_memory 函数进行内存释放操作。该内存泄漏自检测方法根据类对象内存和数组块内存这 2 种不同类型,采取不同方法删除 new 操作符分配的内存,以便监测跟踪所有的内存删除操作:(1) 针对数组块内存,上述代码的第七行直接调用 free_memory 函数释放数组块内存;(2) 针对类对象内存,为每种类提供 Destroy 自定义函数释放类对象内存,如上述代码中的 Destroy 函数,第八行调用析构函数,第九行调用 free_memory 函数释放类对象内存。

1.4 内存分配控制器的实现

如 1.2 节中的代码所示,系统的内存动态分配和释放操作都通过唯一的内存分配控制器实例进行。内存分配控制器的 GetMemory 接口通过调用

Allocator_C 类的 Malloc 函数实现内存的动态分配,通过调用 MemoryMonitor_C 类的 Collect_New_Status 函数实现分配操作的监视。内存分配控制器的 FreeMemory 接口,通过调用 Allocator_C 类的 Free 函数实现内存的动态释放,通过调用 MemoryMonitor_C 类的 Collect_ Free _Status 函数实现释放操作的监视。

2　内存泄漏自检测方法的实验

内存自检测方法评估实验的源代码如下:

```
void main()
{
  Object_C *pObject[100];
  for(int i = 0;i < 100;i + +)
  { int type = rand() %3;
    if(0 = = type)
    pObject[i] = (SequenceList_C *) new
    (SequenceList_MT) SequenceList_C();
    else if(1 = = type)
    pObject[i] = (SequenceStack_C *) new
    (SequenceStack_MT) SequenceStack_C();
    else if(2 = = type)
    pObject[i] = (SequenceQueue_C *) new (SequenceQueue_
                MT) SequenceQueue_C();
  }
  for(i = 0;i < 100;i + +)
  { int isDestroy = rand() %2;
    if(TRUE = = isDestroy) pObject[i] - > Destroy();
  }
}
```

如上述代码所示,针对顺序存储数据结构模型的 6 种内存空间分配类型进行随机内存动态分配及释放操作实验。其中,第五行的随机数决定了动态分配的对象类型,第十四行的随机数决定了是否进行动态释放操作。内存泄漏自检测方法的实验结果如表 1 所示。

表 1　内存泄漏自检测方法的实验结果表

存储空间类型		分配	释放
SequenceList_MT	类对象	31	15
SequenceListArray_MT	数组块	31	15
SequenceQueue_MT	类对象	32	19
SequenceQueueArray_MT	数组块	32	19
SequenceStack_MT	类对象	37	17
SequenceStackArray_MT	数组块	37	17

1.3 节中代码的第五行是类在构造函数中为指针 array 动态分配数组块内存,表明类对象内存和数组块内存的分配操作是同步的;第七行和第九行表明类在 Destroy 函数中调用 free_memory 函数释放数组块内存和类对象内存,表明类对象内存和数组块内存的释放操作是同步的。例如线性表的类对象和数据块内存分配次数都是 31 次,释放次数都是 15 次。经统计,类对象内存泄漏 49 次,数据块内存同步泄漏 49 次,符合本节代码中第十六行的 100 次循环内存随机泄漏实验。

3　结语

本文提出了一种有效的 C + + 内存泄漏自检测方法,给出了其对象行为结构模型以及内存动态分配和释放的具体实现。基于顺序存储数据结构模型的实验及分析表明,该方法不仅能正确地为类对象和数组块内存进行动态分配和释放,还能监视运行程序中是否存在内存泄漏。该方法采用内存池技术的 Allocator_C 类作为内存分配器,减少了申请释放存储空间时的时间消耗,高效地实现了存储管理器的分配和回收操作。该内存泄漏自检测方法已经成功应用于算法设计和系统实现,具有简单、速度快的优点。今后将针对动态内存分配器 Allocator_C 类,在尽量减少空间的浪费及申请释放存储空间时的时间消耗方面展开进一步的研究。

参考文献:

[1] 周　超,林邓伟. Linux 下 C 语言程序内存泄漏的研究[J]. 工矿自动化, 2008(4):137～139.

[2] 王　勇,曾庆凯. 一种内存错误的动态检测方法[J]. 计算机应用研究, 2008,25(5):1 550～1 552.

[3] 程振林,方金云,唐志敏. C + + 编码中减少内存缺陷的方法和工具[J]. 计算机工程, 2007, 33(4):40～44.

[4] 孙凌宇,彭宣戈,冷　明. 一种 ISPD98 电路网表到图的转换算法[J]. 井冈山学院学报:自然科学版, 2008, 29(4): 19～21.

[5] 孙凌宇,彭宣戈,冷　明,等. 一种基于微 GIS 的租赁管理信息系统[J]. 微计算机信息,2007, 28(23): 184～186.

[6] 殷人昆,陶勇雷,谢若阳,等. 数据结构(用面向对象方法与 C + + 描述)[M]. 北京:清华大学出版社,1999.

[7] 陈慧南. 数据结构 ——使用 C + + 语言描述[M]. 2 版. 北京:人民邮电出版社,2008.