

附录 B 内存登录例程

本附录中的代码实现了第 3 章中讨论的内存登录例程的一个简单链表版本。这个代码有意作了简化使之便于理解,但这并不意味着它不可以用在那些大量地使用内存管理程序的应用之中。但在你花时间重写代码使其使用 AVL 树、B 树或其它可以提供快速查找的数据结构之前,试一下这个代码验证它对于实际应用是否太慢了。你也许会发现这个代码很合用,特别是在没有分配许多全局共享的存储模块之时,更是如此。

该文件中给出的实现是很直观的:每当分配一个内存块时,该例程就额外地分配一小块内存以存放 blockinfo(块信息)结构,块信息中有登录信息(定义见下文)。当一个新的 blockinfo 结构创建时,就填充登录信息并置于链表结构的头部。该链表没有特意的顺序。再次说明,该实现是精选的,因为它既简单又容易理解。

block.h:

```
# ifdef DEBUG
/* -----
 * blockinfo 是个数据结构。它记录一个已分配内存块的存储登录信息。
 * 每个已分配的内存块在内存登录中都有一个相应的 blockinfo 结构
 */
typedef struct BLOCKINFO
{
    struct BLOCKINFO * pbiNext;
    byte* pb;                /* 存储块的开始位置 */
    size_t size;             /* 存储块的长度 */
    flag fReferenced;        /* 曾经引用过吗? */
}blockinfo;                /* 命名: bi、*pbi */

flag fCreateBlockInfo(byte* pbNew, size_t sizeNew);
void FreeBlockInfo(byte* pbToFree);
void UpdateBlockInfo(byte* pbOld, byte* pbNew, size_t sizeNew);
size_t sizeofBlock(byte* pb);

void ClearMemoryRefs(void);
void NoteMemoryRef(void* pv);
void CheckMemoryRefs(void);
flag fValidPointer(void* pv, size_t size);
```

```
#endif
```

block.c:

```
#ifdef DEBUG
```

```
/* -----  
 * 该文件中的函数必须要对指针进行比较，而 ANSI 标准不能确保该操作是  
 * 可移植的。  
 *  
 * 下面的宏将该文件所需的指针比较独立出来。该实现采用了总能进行直接  
 * 比较的“直截了当”的指针，下面的定义对某些通用 80x86 内存模型不适用。  
 */
```

```
#define fPtrLess(pLeft,pRight)      ((pLeft) < (pRight))  
#define fPtrGrtr(pLeft,pRight)     ((pLeft) < (pRight))  
#define fPtrEqual(pLeft, pRight)    ((pLeft) == (pRight))  
#define fPtrLEssEq(pLeft, pRight)   ((pLeft) <= (pRight))  
#define fPtrGrtrEq(pLeft, pRight)   ((pLeft) >= (pRight))
```

```
/* ----- */  
/*          * * * * 私有数据/函数 * * * *          */  
/* ----- */
```

```
/* -----  
 * pbiHead 指向内存管理程序调试的单向链接列表。  
 */
```

```
static blockinfo* pbiHead = NULL;
```

```
/* -----  
 * pbiGetBlockInfo(pb)  
 *  
 * pbiGetBlockInfo 查询内存登录找到 pb 所指的存储块，并返回指向内  
 * 存登录中相应 blockinfo 结构的指针。注意：pb 必须指向一个已分配的  
 * 存储块，否则将得到一个断言失败；该函数或者引发断言或者成功，它从  
 * 不返回错误。  
 *  
 * blockinfo * pbi;  
 * .....
```

```

* pbi = pbiGetBlockInfo(pb);
* // pbi->pb 指向 pb 所指存储块的开始位置
* // pbi->size 是 pb 所指存储块的大小
*/

static blockinfo* pbiGetBlockInfo(byte* pb)
{
    blockinfo* pbi;
    for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
    {
        byte* pbStart = pbi->pb;          /* 为了可读性 */
        byte* pbEnd = pbi->pb + pbi->size - 1;
        if( fPtrGrtrEq( pb, pbStart ) && fPtrLessEq( pb, pbEnd ) )
            break;
    }
    /* 没能找到指针？它是 (a) 垃圾？(b) 指向一个已经释放了的存储块？
    * 或 (c) 指向一个在由 fResizeMemory 重置大小时而移动了的存储块？
    */
    ASSERT( pbi != NULL );
    return( pbi );
}

/* ----- */
/*          * * * * 公共函数 * * * *          */
/* ----- */

/* ----- */
* fCreateBlockInfo(pbNew, sizeNew)
*
* 该函数为由 pbNew : sizeNew 定义的存储块建立一个登录项。如果成功地
* 建立了登录信息则该函数返回 TRUE ，否则返回 FALSE 。
*
*
*          if( fCreateBlockInfo( pbNew, sizeNew ) )
*              成功 —— 该内存登录具有 pbNew : sizeNew 项
*          else
*              失败 —— 由于没有该项则应释放 pbNew
*/

```

```

flag fCreateBlockInfo( byte* pbNew, size_t sizeNew )
{
    blockinfo* pbi;
    ASSERT( pbNew != NULL && sizeNew != 0 );
    pbi = ( blockinfo* )malloc( sizeof( blockinfo ) );
    if( pbi != NULL )
    {
        pbi->pb = pbNew;
        pbi->size = sizeNew;
        pbi->pbiNext = pbiHead;
        pbiHead = pbi ;
    }
    return(flag)( pbi != NULL );
}

```

```

/* -----
 * FreeBlockInfo( pbToFree )
 *
 * 该函数清除由 pbToFree 所指存储块的登录项。pbToFree 必须指向一
 * 个已分配存储块的开始位置，否则将得到一个断言失败。
 */

```

```

void FreeBlockInfo( byte* pbToFree )
{
    blocinfo *pbi, *pbiPrev;
    for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
    {
        if( fPtrEqual( pbi->pb, pbToFree ) )
        {
            if( pbiPrev == NULL )
                pbiHead = pbi->pbiNext;
            else
                pbiPrev->pbiNext = pbi->pbiNext;
            break;
        }
        pbiPrev = pbi;
    }
}

```

```

    }

    /* 如果是 pbi 是 NULL 则 pbToFree 无效 */
    ASSERT( pbi != NULL );
    /* 在释放之前破坏*pbi 的内容 */
    memset(pbi, bGarbage, sizeof(blockinfo));
    free(pbi);
}

/* -----
 * UpdateBlockInfo ( pbOld , pbNew , sizeNew )
 *
 * UpdateBlockInfo 查出 pbOld 所指存储块的登录信息，然后该函数修
 * 改登录信息已反映该存储块现在所处的新位置 (pbNew) 和新的字节长
 * 度 (sizeNew)。pbOld 必须指向一个已分配存储块的开始位置，否则
 * 将得到一个断言失败。
 */
void UpdateBlockInfo( byte* pbOld, byte* pbNew, size_t sizeNew )
{
    blockinfo* pbi;
    ASSERT( pbNew != NULL && sizeNew != 0 );
    pbi = pbiGetBlockInfo( pbOld );
    ASSERT( pbOld == pbi->pb );      /* 必须指向一个存储块的开始位置 */
    pbi->pb = pbNew;
    pbi->size = sizeNew;
}

/* -----
 * sizeofBlock (pb )
 * sizeofBlock 返回 pb 所指存储块的大小。pb 必须指向一个已分配存储块
 * 的开始位置，则将得到一个断言失败。
 */
size_t sizeofBlock( byte* pb )
{
    blockinfo* pbi;
    pbi = pbiGetBlockInfo(pb);
    ASSERT(pb == pbi->pb );          /* 必须指向存储块的开始位置 */

```

```

        return( pbi->size );
    }

/* ----- */
/*      下面例程用来寻找丢失的存储块和悬挂的指针。有关这些例程的      */
/*      说明参见第三章                                          */
/* ----- */

/* -----
* ClearMemoryRefs( void )
*
* ClearMemoryRefs 将内存登录中所有存储块标志为未引用。
*/
void ClearMemoryRefs( void )
{
    blockinfo* pbi;
    for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
        pbi->fReferenced = FALSE;
}

/* -----
* NoteMemoryRef( pv )
*
* NoteMemoryRefs 将 pv 所指的存储块标志为已引用。注意：pv 不必指向一
* 个存储块的开始位置；它可以指向一个已分配存储块的任何位置。
*/

void NoteMemoryRef ( void * pv )
(
    blockinfo* pbi;
    pbi = pbiGetBlockInfo( (byte*)pv );
    pbi->fReferenced = TRUE;
}

/* -----
* CheckMemoryRefs( void )

```

```

* CheckMemoryRefs 扫描内存登录以寻找未通过调用 NoteMemoryRef 进行标志
* 的存储块。如果该函数发现了一个未被标志的存储块，它就引发断言。
*/
void CheckMemoryRefs( void )
{
    blockinfo * pbi ;
    for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
    {
        /* 简单检查存储块的完整性。如果引发该断言，就说明管理 blockinfo
        * 的调试代码有某些错误，或者说明紊乱的内存已经破坏了数据结构。
        * 无论哪种情况，都存在错误。
        */
        ASSERT ( pbi->pb != NULL && pbi->size != 0 );
        /* 检查丢失/遗漏的存储空间。如果引发了该断言，就说明 app 或者丢
        * 失了该存储块的轨道或者没有用 NoteMemoryRef 解释所有的全局指针。
        */
        ASSERT( pbi->fReferenced );
    }
}

/* -----
* fValidPointer( pv, size )
*
* fValidPointer 验证 pv 指向一个已分配的存储块并且从 pv 所指处到
* 块的结尾至少有“size”个已分配字节。如果有任一个条件没有满足，
* fValidPointer 将引发断言；该函数将从不返回 FALSE，fValidPointer
* 之所以返回一个（总为 TRUE）标记是为了允许在断言宏内调用该函
* 数。当这不是最有效的方法时，不采用#ifdef DEBUG 或者不引入其它象
* 断言的宏，而单纯地使用断言来处理调试/交付版本控制。
*
* ASSERT( fValidPointer( pb, size ) );
*/
flag fValidPointer( void* pv, size_t size )
{
    blockinfo* pbi;
    byte* pb = ( byte* )pv;
    pbi = pbiGetBlockInfo( pb );
    /* 使 pv 有效 */

```

```
    ASSERT( pv != NULL && size != 0 );  
    /* pv 是有效的，但 size 呢？（如果 pb + size 上溢出了该存储块，则  
     * size 是无效的）  
     */  
    ASSERT( fPtrLessEq( pb + size, pbi->pb + pbi->size ) );  
    return( TRUE );  
}  
#endif
```