```
/*****************************************************************************/
/*!
\file   ObjectAllocator.cpp
\author Roland Shum
\par    email: roland.shum\@digipen.edu
\par    DigiPen login: roland.shum
\par    Course: CS280
\par    Assignment #1
\date   1/24/2020
\brief
  This is the interface for the Object Allocator and its accompanying classes.
  This includes OAConfig, MemBlockInfo, and ObjectAllocator.

  ObjectAllocator Functions
  - Constructor
  - Destructor
  - Allocate
  - Free
  - Debug functions:
    - DumpMemoryInUse
    - ValidePages
    - FreeEmptyPages
    - SetDebugState
    - GetFreeList
    - GetPageList
    - GetConfig
    - GetStats
```

```
28
29    OAConfig
30    - Constructor
31
32    MemBlockInfo
33    - Constructor
34  */
35  /**************************************************************************/
36  #include "ObjectAllocator.h"
37  #include <cstdint>   // size_t
38  #include <cstring>   // strlen, memset
39  #include <cstdlib>   // abs
40
41  #define PTR_SIZE sizeof(word_t)      //! Size of a pointer.
42  #define INCREMENT_PTR(ptr) (ptr + 1) //! Move the pointer by one
43
44  //! A word is a pointer
45  using word_t = intptr_t ;
46
47  /**************************************************************************/
48  /*!
49
50    Pass in a literal to convert it to a size_t type. Example: 2_z; Similar
51    to how 2.0f is a float.
52
53    \param n
54      A number to turn into size_t type in compile time.
55    \return
56      Returns n in size_t type.
57  */
58  /**************************************************************************/
59  constexpr size_t operator "" _z(unsigned long long n)
60  {
61      return static_cast<size_t>(n);
62  }
63
64  /**************************************************************************/
65  /*!
66
67    Given a size and an alignment, returns the size after it has been aligned.
68    Ex: Given a size of 7 and align of 8, returns 8. If size of 11, align of 8,
69    returns 16.
70
71    \param n

72      The size to align
73    \param align
74      The power of size to align to.
75    \return
```

```
    Returns n, expanded so that it aligns with align
*/
/*****************************************************************************/
inline size_t align(size_t n, size_t align)
{
    if (!align)
        return n;
    size_t remainder = n % align == 0 ? 0_z : 1_z;
    return align * ((n / align) + remainder);
}

/*****************************************************************************/
/*!

  Constructor for the mem block info.

  \param alloc_num_
    The allocation number/ID.
  \param label
    C string to identify the header / data block.
*/
/*****************************************************************************/
MemBlockInfo::MemBlockInfo(unsigned alloc_num_, const char* label_): in_use(true),
    label(nullptr), alloc_num(alloc_num_)
{
    if (label_)
    {
        try
        {
            // +1 to account for null.
            label = new char[strlen(label_) + 1];
        }
        catch (std::bad_alloc&)
        {
            throw OAException(OAException::E_NO_MEMORY, "Out of memory!");
        }
        strcpy(label, label_);
    }
}

/*****************************************************************************/
/*!

  Destructor for MemBlockInfo. Deletes the C-string array


*/
/*****************************************************************************/
MemBlockInfo::~MemBlockInfo()
```

```
{
    delete[] label;
}

/*****************************************************************************/
/*!

  Constructor for Object Allocator. Determines stats and allocates inital page
  of memory.

  /param ObjectSize
    The size of each object that the allocator allocates.
  /param config
    An instance of OAConfig that determines the configuration of the Object
    Allocator

*/
/*****************************************************************************/
ObjectAllocator::ObjectAllocator(size_t ObjectSize, const OAConfig &config) : configuration(config
{
    // TODO: Put this in constructor
    // First we have to calculate all the stats
    this->headerSize = align(PTR_SIZE + config.HBlockInfo_.size_ + config.PadBytes_, config.Alignm
    this->dataSize = align(ObjectSize + config.PadBytes_ * 2_z + config.HBlockInfo_.size_, config.
    this->stats.ObjectSize_ = ObjectSize;
    this->stats.PageSize_ = headerSize + dataSize * (config.ObjectsPerPage_ - 1) + ObjectSize + co
    this->totalDataSize = dataSize * (configuration.ObjectsPerPage_ - 1) + ObjectSize + config.Pad

    size_t interSize = ObjectSize + this->configuration.PadBytes_ * 2_z + static_cast<size_t>(this
    this->configuration.InterAlignSize_ = static_cast<unsigned>(align(interSize, this->configurati
    allocate_new_page_safe(this->PageList_);
}

/*****************************************************************************/
/*!

  Destructor for Object Allocator. Removes pages.

*/
/*****************************************************************************/
ObjectAllocator::~ObjectAllocator()
{
    // We will need to call the destructor for every active object, and then
    // free the entire page

    // Walk the pages
    GenericObject* page = this->PageList_;
    while(page != nullptr)
    {
```

```
172         GenericObject* next = page->Next;
173         // Free external headers.
174         if(this->configuration.HBlockInfo_.type_ == OAConfig::hbExternal)
175         {
176             unsigned char* headerAddr = reinterpret_cast<unsigned char*>(page) + this->headerSize;
177             for(unsigned i = 0; i < configuration.ObjectsPerPage_; i++)
178             {
179                 freeHeader(reinterpret_cast<GenericObject*>(headerAddr), OAConfig::hbExternal, tru
180             }
181         }
182         delete[] reinterpret_cast<unsigned char*>(page);
183         page = next;
184     }
185 }
186 
187 /**************************************************************************/
188 /*!
189 
190   Given a label, allocates memory from the page and returns it to the user.
191   If UseCPPMemManager is true, bypasses OA and use new.
192 
193 
194   \param label
195     Label to pass to the external header to keep track of.
196 
197   \return
198     Pointer to block of memory to use.
199 
200   \exception OAException
201     Exception contains the type of exception it is. Possible types are
202     -OAException::E_NO_MEMORY -- signifies no more memory from OS
203     -OAException::E_NO_PAGES  -- There are no more memory in the pages to use,
204                                  and the configuration also does not allow more
205                                  memory.
206 */
207 /**************************************************************************/
208 void* ObjectAllocator::Allocate(const char* label)
209 {
210     //If we are not using the mem manager.
211     if (this->configuration.UseCPPMemManager_)
212     {
213         try
214         {
215             unsigned char* newObj = new unsigned char[this->stats.ObjectSize_];
216             incrementStats();
217             return newObj;
218         }
219         catch (std::bad_alloc&)
```

```
220         {
221             throw OAException(OAException::E_NO_MEMORY, "Out of memory!");
222         }
223     }
224 
225     // First we check if we can allocate from freelist.
226     if (nullptr == FreeList_)
227     {
228         // Nothing left in freelist, allocate new page.
229         allocate_new_page_safe(this->PageList_);
230     }
231     // Give them from the new page.
232     GenericObject* objectToGive = this->FreeList_;
233 
234     // Update the free list
235     this->FreeList_ = this->FreeList_->Next;
236 
237     // Update sig
238     if (this->configuration.DebugOn_)
239     {
240         memset(objectToGive, ALLOCATED_PATTERN, stats.ObjectSize_);
241     }
242     incrementStats();
243     // Update header
244     updateHeader(objectToGive, configuration.HBlockInfo_.type_, label);
245 
246 
247     // Return the object
248     return objectToGive;
249 }
250 
251 /*************************************************************************/
252 /*!
253 
254   Given a block of memory given by Allocate(), returns the memory back to the
255   Object Allocator.  If UseCPPMemManager is true, bypasses OA and use delete.
256   If debugOn is True, will perform checking to see if the block is corrupted
257   and within the same boundaries.
258 
259   \param Object
260     Pointer to object allocated by Allocate().
261   \exception OAException
262     Exception contains the type of exception it is. Possible types are
263     -OAException::E_CORRUPTED_BLOCK -- The memory block is corrupted.
264     -OAException::E_BAD_BOUNDARY    -- The Object given is not aligned properly.
265                                        Possibly wrong object.
266 */
267 /*************************************************************************/
```

```
268 void ObjectAllocator::Free(void* Object)
269 {
270     // Increment Deallocation count
271     ++this->stats.Deallocations_;
272 
273     //If we are not using the mem manager.
274     if (this->configuration.UseCPPMemManager_)
275     {
276         delete[] reinterpret_cast<unsigned char*>(Object);
277         return;
278     }
279 
280 
281     GenericObject* object = reinterpret_cast<GenericObject*>(Object);
282     // Check if pad bytes are OK AKA boundary check
283     // If there are no paddings...
284 
285     if (this->configuration.DebugOn_)
286     {
287         check_boundary_full(reinterpret_cast<unsigned char*>(Object));
288         {
289             if (!isPaddingCorrect(toLeftPad(object), this->configuration.PadBytes_))
290             {
291                 throw OAException(OAException::E_CORRUPTED_BLOCK, "Bad boundary.");
292             }
293             if (!isPaddingCorrect(toRightPad(object), this->configuration.PadBytes_))
294             {
295                 throw OAException(OAException::E_CORRUPTED_BLOCK, "Bad boundary.");
296             }
297         }
298     }
299 
300     freeHeader(object, this->configuration.HBlockInfo_.type_);
301 
302     if (this->configuration.DebugOn_)
303     {
304         memset(object, FREED_PATTERN, stats.ObjectSize_);
305     }
306     object->Next = nullptr;
307 
308 
309 
310 
311     put_on_freelist(object);
312     // Update stats
313     --this->stats.ObjectsInUse_;
314 }
315 
```

```
316 /**************************************************************************/
317 /*!
318
319   Debug Function: Walks through the pages in the OA and calls the given
320   function if the blocks of data are active
321
322   \parem fn
323     Function to call for each active object
324   \return
325     Number of active objects
326 */
327 /**************************************************************************/
328 unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
329 {
330     // Walk through each page, and dump it.
331     if (!PageList_)
332     {
333         // empty list
334         return 0 ;
335     }
336     else
337     {
338         unsigned memInUse = 0;
339         // walk to the end, while calling the fn for each page.
340         GenericObject* last = PageList_;
341         while (last)
342         {
343             unsigned char* block = reinterpret_cast<unsigned char*>(last);
344             // Move to the first data block.
345             block += this->headerSize;
346
347                         // For each data
348                         for (unsigned i = 0; i < configuration.ObjectsPerPage_; ++i)
349                         {
350                                 GenericObject* objectData = reinterpret_cast<GenericObject*>(block
351
352                 if (isObjectAllocated(objectData))
353                 {
354                                         fn(objectData, stats.ObjectSize_);
355                                         ++memInUse;
356                 }
357                                 // Check if mem is in use. By using slower checkData function
358                                 //if (checkData(objectData, ALLOCATED_PATTERN))
359                                 //{
360                                 //      fn(objectData, stats.ObjectSize_);
361                                 //      ++memInUse;
362                                 //}
363                         }
```

```
364
365             last = last->Next;
366         }
367         return memInUse;
368     }
369 }
370
371 /**************************************************************************/
372 /*!
373
374   Debug Function: Walks through each page and verifies if each data block is
375   corrupted or not.
376
377   \param fn
378     Function to call once a data block is identified to be corrupted.
379   \return
380     Number of corrupted data blocks.
381 */
382 /**************************************************************************/
383 unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
384 {
385     if (!configuration.DebugOn_ || !configuration.PadBytes_)
386         return 0;
387     unsigned corruptedBlocks = 0;
388
389     //You need to walk each of the pages in the page list checking
390     // the pad bytes of each block (free or not).
391     GenericObject* last = PageList_;
392     while (last)
393     {
394         unsigned char* block = reinterpret_cast<unsigned char*>(last);
395         // Move to the first data block.
396         block += this->headerSize;
397
398         // For each data
399         for(unsigned i = 0; i < configuration.ObjectsPerPage_; ++i)
400         {
401             GenericObject* objectData = reinterpret_cast<GenericObject*>(block + i * dataSize);
402             // Validate the left and right pad
403             if(!isPaddingCorrect(toLeftPad(objectData), configuration.PadBytes_) ||
404                 !isPaddingCorrect(toRightPad(objectData), configuration.PadBytes_))
405             {
406                 fn(objectData, stats.ObjectSize_);
407                 ++corruptedBlocks;
408                 continue;
409             }
410         }
411         last = last->Next;
```

```
412     }
413
414     return corruptedBlocks;
415 }
416
417 /**************************************************************************/
418 /*!
419
420   Debug Function: Walks through each page and checks if the page is empty.
421   If it is, returns the memory of the page back to the OS.
422
423   \return
424     The number of pages freed.
425 */
426 /**************************************************************************/
427 unsigned ObjectAllocator::FreeEmptyPages()
428 {
429     if (this->PageList_ == nullptr)
430         return 0;
431     // Return value
432     unsigned emptyPages = 0;
433
434     // Store head node
435     GenericObject* temp = this->PageList_, * prev = nullptr;
436
437     // If head node itself holds the key or multiple occurrences of key
438     while (temp != nullptr && isPageEmpty(temp))
439     {
440         this->PageList_ = temp->Next;   // Changed head
441         freePage(temp);
442         temp = this->PageList_;         // Change Temp
443         emptyPages++;
444     }
445
446     // Delete occurrences other than head
447     while (temp != nullptr)
448     {
449         // Search for the key to be deleted, keep track of the
450         // previous node as we need to change 'prev->next'
451         while (temp != nullptr && !isPageEmpty(temp))
452         {
453             prev = temp;
454             temp = temp->Next;
455         }
456
457         // If key was not present in linked list
458         if (temp == nullptr) return emptyPages;
459
```

```
460         // Unlink the node from linked list
461         prev->Next = temp->Next;
462         freePage(temp);
463 
464         //Update Temp for next iteration of outer loop
465         temp = prev->Next;
466         emptyPages++;
467     }
468     return emptyPages;
469 }
470 
471 /**************************************************************************/
472 /*!
473 
474   Given a page, frees it from memory by returning it to the OS. It first
475   removes all the datablocks from the freelist, and decrements the stat counter.
476 
477 */
478 /**************************************************************************/
479 void ObjectAllocator::freePage(GenericObject* temp)
480 {
481     removePageObjs_from_freelist(temp);
482     delete[] reinterpret_cast<unsigned char*>(temp);
483     this->stats.PagesInUse_--;
484 
485 }
486 
487 /**************************************************************************/
488 /*!
489 
490   Does this OA implement extra credit? (Hint: Yes)
491 
492   \return
493     Returns true if implemented. False otherwise.
494 */
495 /**************************************************************************/
496 bool ObjectAllocator::ImplementedExtraCredit()
497 {
498     return true;
499 }
500 
501 /**************************************************************************/
502 /*!
503 
504   Given a debug state, sets the current OA to that debug state.
505 
506   \param State
507     True to turn on Debug, False to turn off.
```

```
508 */
509 /**************************************************************************/
510 void ObjectAllocator::SetDebugState(bool State)
511 {
512     this->configuration.DebugOn_ = State;
513 }
514 
515 /**************************************************************************/
516 /*!
517 
518   Debug Function: Returns the free list, a linked list determining what blocks
519   of memory are free.
520 
521   \return
522     The free list.
523 */
524 /**************************************************************************/
525 const void* ObjectAllocator::GetFreeList() const
526 {
527     return FreeList_;
528 }
529 
530 /**************************************************************************/
531 /*!
532 
533   Debug Function: Returns the page list, a linked list that chains all the pages
534   together.
535 
536   \return
537     The page list.
538 */
539 /**************************************************************************/
540 const void* ObjectAllocator::GetPageList() const
541 {
542     return PageList_;
543 }
544 
545 /**************************************************************************/
546 /*!
547 
548   Debug Function: Returns a copy of the current configuration
549 
550   \return
551     The a copy of the current configuration
552 */
553 /**************************************************************************/
554 OAConfig ObjectAllocator::GetConfig() const
555 {
```

```
    return this->configuration;
}

/**************************************************************************/
/*!

  Debug Function: Returns a copy of the current stats

  \return
    The a copy of the current stats
*/
/**************************************************************************/
OAStats ObjectAllocator::GetStats() const
{
    return this->stats;
}

/**************************************************************************/
/*!

  Allocates a new page after checks. Checks if we have hit the max number of
  pages we are allowed to allocate.

  \param LPageList
    A reference to the head of the linked list of the pages.
  \exceptions OAException
    OAException::OA_EXCEPTION::E_NO_PAGES  - Ran of out pages to use.
    OAException::OA_EXCEPTION::E_NO_MEMORY - Signifies no more memory from OS
*/
/**************************************************************************/
void ObjectAllocator::allocate_new_page_safe(GenericObject *&LPageList)
{
    // If we have hit the max amount of pages...
    if (stats.PagesInUse_ == configuration.MaxPages_)
    {
        // Max pages have been reached.
        throw OAException(OAException::OA_EXCEPTION::E_NO_PAGES, "Out of pages!");
    }
    // If we can still allocate pages.
    else
    {
        // DEBUG TODO REMOVE LATER
        if (stats.PagesInUse_ >= configuration.MaxPages_)
            throw std::exception();

        // Allocate a new page.
        GenericObject* newPage = allocate_new_page(this->stats.PageSize_);

        if (this->configuration.DebugOn_)
```

```
604         {
605             memset(newPage, ALIGN_PATTERN, this->stats.PageSize_);
606         }
607 
608         // Link it up to the page list.
609         InsertHead(LPageList, newPage);
610 
611 
612         // Putting objects on free list
613         unsigned char* PageStartAddress = reinterpret_cast<unsigned char*>(newPage);
614         unsigned char* DataStartAddress = PageStartAddress + this->headerSize;
615 
616 
617         // For each start of the data...
618         for (; static_cast<unsigned>(abs(static_cast<int>(DataStartAddress - PageStartAddress))) <
619          DataStartAddress += this->dataSize)
620         {
621             // We intepret it as a pointer.
622             GenericObject* dataAddress = reinterpret_cast<GenericObject*>(DataStartAddress);
623 
624             // Add the pointer to the free list.
625             put_on_freelist(dataAddress);
626 
627             if (this->configuration.DebugOn_)
628             {
629                 // Update padding sig
630                 memset(reinterpret_cast<unsigned char*>(dataAddress) + PTR_SIZE, UNALLOCATED_PATTE
631                 memset(toLeftPad(dataAddress), PAD_PATTERN, this->configuration.PadBytes_);
632                 memset(toRightPad(dataAddress), PAD_PATTERN, this->configuration.PadBytes_);
633             }
634             memset(toHeader(dataAddress), 0, configuration.HBlockInfo_.size_);
635         }
636     }
637 }
638 
639 /*************************************************************************/
640 /*!
641 
642   Allocates a page according to the configuration given. No checks are done in
643   regards to pages.
644 
645   \param pageSize
646     The total size of the page
647   \exceptions OAException
648 
649     OAException::OA_EXCEPTION::E_NO_MEMORY - Signifies no more memory from OS
650   \return A pointer to the newly allocated page of memory from the OS
651 */
652 /*************************************************************************/
```

```
652 GenericObject* ObjectAllocator::allocate_new_page(size_t pageSize)
653 {
654     try
655     {
656         GenericObject* newObj = reinterpret_cast<GenericObject*>(new unsigned char[pageSize]());
657         ++this->stats.PagesInUse_;
658         return newObj;
659     }
660     catch (std::bad_alloc& exception)
661     {
662         throw OAException(OAException::OA_EXCEPTION::E_NO_MEMORY, "OA out of mem!");
663     }
664 }
665 
666 /*****************************************************************************/
667 /*!
668 
669   Given an object, places it at the front of the freelist.
670 
671   \param Object
672     Object to place on Free List
673 
674 */
675 /*****************************************************************************/
676 void ObjectAllocator::put_on_freelist(GenericObject* Object)
677 {
678     GenericObject* temp = this->FreeList_;
679     this->FreeList_ = Object;
680     Object->Next = temp;
681 
682     this->stats.FreeObjects_++;
683 }
684 
685 /*****************************************************************************/
686 /*!
687 
688   Given a page, removes all the data blocks in that page from the free list.
689 
690   \param pageAddr
691     Pointer to the page that the user wants to remove all blocks from free list
692     from.
693 
694 */
695 /*****************************************************************************/

696 void ObjectAllocator::removePageObjs_from_freelist(GenericObject* pageAddr)
697 {
698     // Store head node
699     GenericObject* temp = this->FreeList_, * prev = nullptr;
```

```
700
701     // If head node itself holds the key or multiple occurrences of key
702     while (temp != nullptr && isInPage(pageAddr, reinterpret_cast<unsigned char*>(temp)))
703     {
704         this->FreeList_ = temp->Next;   // Changed head
705         temp = this->FreeList_;         // Change Temp
706         this->stats.FreeObjects_--;
707     }
708
709     // Delete occurrences other than head
710     while (temp != NULL)
711     {
712         // Search for the key to be deleted, keep track of the
713         // previous node as we need to change 'prev->next'
714         while (temp != nullptr && !isInPage(pageAddr, reinterpret_cast<unsigned char*>(temp)))
715         {
716             prev = temp;
717             temp = temp->Next;
718         }
719
720         // If key was not present in linked list
721         if (temp == nullptr) return;
722
723         // Unlink the node from linked list
724         prev->Next = temp->Next;
725
726         this->stats.FreeObjects_--;
727         //Update Temp for next iteration of outer loop
728         temp = prev->Next;
729     }
730 }
731
732 /*************************************************************************/
733 /*!
734
735   Increment the stats when an Object is allocted.
736
737 */
738 /*************************************************************************/
739 void ObjectAllocator::incrementStats()
740 {
741     // Update stats
742     ++this->stats.ObjectsInUse_;
743     if (this->stats.ObjectsInUse_ > this->stats.MostObjects_)
744         this->stats.MostObjects_ = this->stats.ObjectsInUse_;
745     --this->stats.FreeObjects_;
746     ++this->stats.Allocations_;
747 }
```

```
748
749 /*****************************************************************************/
750 /*!
751
752   Given an object and the current header type configuration, frees the header.
753   If debug is set to true, this function will check for multiple frees
754
755
756   \param Object
757     Object to place on Free List
758   \param headerType
759     Type of header.
760   \param ignoreThrow
761     If true, this function will not throw.
762   \exception OAException
763     OAException::E_MULTIPLE_FREE -- If given block is already freed.
764
765 */
766 /*****************************************************************************/
767 void ObjectAllocator::freeHeader(GenericObject* Object, OAConfig::HBLOCK_TYPE headerType,
768   bool ignoreThrow)
769 {
770     unsigned char* headerAddr = toHeader(Object);
771     switch (headerType)
772     {
773     case OAConfig::hbNone:
774     {
775         // We check if it has been freed by checking the last byte of the object and comparing
776         // to 0xCC
777         if (this->configuration.DebugOn_ && !ignoreThrow)
778         {
779             unsigned char* lastChar = reinterpret_cast<unsigned char*>(Object) + stats.ObjectSize_
780             if (*lastChar == ObjectAllocator::FREED_PATTERN)
781                 throw OAException(OAException::E_MULTIPLE_FREE, "Multiple free!");
782         }
783         break;
784     }
785     case OAConfig::hbBasic:
786     {
787         // Check if the bit is already free
788         if (this->configuration.DebugOn_ && !ignoreThrow)
789         {
790             if (0 == *(headerAddr + sizeof(unsigned)))
791                 throw OAException(OAException::E_MULTIPLE_FREE, "Multiple free!");

792         }
793         // Reset the basic header
794         memset(headerAddr, 0, OAConfig::BASIC_HEADER_SIZE);
795         break;
```

```
796         }
797         case OAConfig::hbExtended:
798         {
799             if (this->configuration.DebugOn_ && !ignoreThrow)
800             {
801                 if (0 == *(headerAddr + sizeof(unsigned) + this->configuration.HBlockInfo_.additional_
802                     + sizeof(unsigned short)))
803                     throw OAException(OAException::E_MULTIPLE_FREE, "Multiple free!");
804             }
805             // Reset the basic header part of the extended to 0
806             memset(headerAddr + this->configuration.HBlockInfo_.additional_ + sizeof(unsigned short),
807             break;
808         }
809         case OAConfig::hbExternal:
810         {
811             // Free the external values
812 
813             MemBlockInfo** info = reinterpret_cast<MemBlockInfo**>(headerAddr);
814             if(nullptr == *info     && this->configuration.DebugOn_ && !ignoreThrow)
815                 throw OAException(OAException::E_MULTIPLE_FREE, "Multiple free!");
816             delete *info;
817             *info = nullptr;
818         }
819         default:
820             break;
821         }
822     }
823 
824     /*****************************************************************************/
825     /*!
826 
827       Given a pointer to a data block, builds the basic header for that data block.
828       A basic header block consists of 5 bytes. 4 for allocation number, and one flag
829       to determine if the block is allocated or not.
830       \param addr
831         Pointer to data block to build header for
832     */
833     /*****************************************************************************/
834     void ObjectAllocator::buildBasicHeader(GenericObject* addr)
835     {
836         unsigned char* headerAddr = toHeader(addr);
837         unsigned* allocationNumber = reinterpret_cast<unsigned*>(headerAddr);
838         *allocationNumber = this->stats.Allocations_;
839         // Now set the allocation flag
840         unsigned char* flag = reinterpret_cast<unsigned char*>(INCREMENT_PTR(allocationNumber));
841         *flag = true;
842     }
843 
```

```
844 /*****************************************************************************/
845 /*!
846
847   Given a pointer to a data block, builds the external header for that data block.
848   The external header is a MemBlockInfo
849
850   \param Object
851     Pointer to data block to build header for
852   \param label
853     The label for the external header to hold
854   \exceptions OAException
855     OAException::E_NO_MEMORY -- Thrown if OS is out of memory.
856 */
857 /*****************************************************************************/
858 void ObjectAllocator::buildExternalHeader(GenericObject* Object, const char* label)
859 {
860     unsigned char* headerAddr = toHeader(Object);
861     MemBlockInfo** memPtr = reinterpret_cast<MemBlockInfo**>(headerAddr);
862     try
863     {
864         *memPtr = new MemBlockInfo(stats.Allocations_, label);
865     }
866     catch (std::bad_alloc&)
867     {
868         throw OAException(OAException::E_NO_MEMORY, "No memory");
869     }
870 }
871
872 /*****************************************************************************/
873 /*!
874
875   Given a pointer to a data block, builds the extended header for that data block.
876   The extended header adds on to the basic header two more things, a counter and
877   a user specified field.
878
879   \param addr
880     Pointer to data block to build header for
881 */
882 /*****************************************************************************/
883 void ObjectAllocator::buildExtendedHeader(GenericObject* Object)
884 {
885     unsigned char* headerAddr = toHeader(Object);
886     // Set the 2 byte use-counter, 5 for 5 bytes of user defined stuff.
887     unsigned short* counter = reinterpret_cast<unsigned short*>(headerAddr + this->configuration.H
888     ++(*counter);
889
890     unsigned* allocationNumber = reinterpret_cast<unsigned*>(INCREMENT_PTR(counter));
891     *allocationNumber = this->stats.Allocations_;
```

```
892     // Now set the allocation flag
893     unsigned char* flag = reinterpret_cast<unsigned char*>(INCREMENT_PTR(allocationNumber));
894     *flag = true;
895 }
896 
897 /*************************************************************************/
898 /*!
899 
900   A slower but more throrough check to see if the given object is in a bad
901   boundary. Used to check address from Free. If the object is in a bad boundary,
902   an exception is thrown.
903 
904   \param addr
905     The address to check if it is 1) within the page, and 2) a correct address.
906   \exception OAException
907     OAException::E_BAD_BOUNDARY -- Object is in a bad bounday.
908 */
909 /*************************************************************************/
910 void ObjectAllocator::check_boundary_full(unsigned char* addr) const
911 {
912         // Find the page the object rests in.
913         GenericObject* pageList = this->PageList_;
914         // While loop stops when addr resides within the page.
915         while(!isInPage(pageList, addr))
916         {
917             pageList = pageList->Next;
918             // If its not in our pages, its not our memory.
919             if(!pageList)
920             {
921                 throw OAException(OAException::E_BAD_BOUNDARY, "Bad boundary.");
922             }
923         }
924         // We have found that is is in our pages. Check the boundary using %
925         unsigned char* pageStart = reinterpret_cast<unsigned char*>(pageList);
926 
927         // Check if we are intruding on header.
928         if(static_cast<unsigned>(addr - pageStart) < this->headerSize)
929             throw OAException(OAException::E_BAD_BOUNDARY, "Bad boundary.");
930 
931         pageStart += this->headerSize;
932         long distance = addr - pageStart;
933         if(static_cast<size_t>(distance) % this->dataSize != 0)
934             throw OAException(OAException::E_BAD_BOUNDARY, "Bad boundary.");
935 
936 }
937 
938 /*************************************************************************/
939 /*!
```

```
940
941   A thorough check to see if the padding surrounding a block is correct.
942
943   \param paddingAddr
944     The address to the start of a padding.
945   \size
946     The size of the padding.
947   \return
948     True if padding is not corrupted, false if not.
949 */
950 /**************************************************************************/
951 bool ObjectAllocator::isPaddingCorrect(unsigned char* paddingAddr, size_t size) const
952 {
953     for(size_t i = 0; i < size; ++i)
954     {
955         if (*(paddingAddr + i) != ObjectAllocator::PAD_PATTERN)
956             return false;
957     }
958     return true;
959 }
960
961 /**************************************************************************/
962 /*!
963
964   Given a data block and a pattern, checks if the pattern exists on the data
965   block.
966
967   \param objectData
968     Address to the data block to check.
969   \param pattern
970     Pattern to check the data block for.
971   \return true if the pattern is in the data, false if not.
972 */
973 /**************************************************************************/
974 bool ObjectAllocator::checkData(GenericObject* objectdata, const unsigned char pattern) const
975 {
976     unsigned char* data = reinterpret_cast<unsigned char*>(objectdata);
977     for(size_t i = 0; i < stats.ObjectSize_; ++i)
978     {
979         if (data[i] == pattern)
980             return true;
981     }
982     return false;
983 }


984
985 /**************************************************************************/
986 /*!
987
```

```
  Given a page address and any address, checks if the address is in the page.

  \param pageAddr - Pointer to page
  \param addr - Address to check if it is within page.
  \return true if in page, false if not.
*/
/**************************************************************************/
bool ObjectAllocator::isInPage(GenericObject* pageAddr, unsigned char* addr) const
{
    return (addr >= reinterpret_cast<unsigned char*>(pageAddr) &&
        addr < reinterpret_cast<unsigned char*>(pageAddr) + stats.PageSize_);
}

/**************************************************************************/
/*!

  Given a page, checks if the page is empty by walking through the freelist
  and checking if there are $(ObjectsPerPage_) free items in a page.

  \param page
    Page to check if it is empty
  \return
    True if page is empty, else false
*/
/**************************************************************************/
bool ObjectAllocator::isPageEmpty(GenericObject* page) const
{
    // Walk though the linked list.
    GenericObject* freeList = this->FreeList_;
    unsigned freeInPage = 0;
    while (freeList)
    {
        if (isInPage(page, reinterpret_cast<unsigned char*>(freeList)))
        {
            if (++freeInPage >= configuration.ObjectsPerPage_)
                return true;
        }
        freeList = freeList->Next;
    }
    return false;
}

/**************************************************************************/
/*!


  Given an object, checks if it is allocated. If configuration has a header, it
  would do a header check. Else it would check if the data is in freelist.

```

```
1036    \param object
1037      The object to check if it is allocated
1038    \return
1039      True if allocated, false if not.
1040  */
1041  /**************************************************************************/
1042  bool ObjectAllocator::isObjectAllocated(GenericObject* object) const
1043  {
1044      switch (this->configuration.HBlockInfo_.type_)
1045      {
1046      case OAConfig::HBLOCK_TYPE::hbNone:
1047      {
1048         // Checks if it is in free list.
1049        GenericObject* freelist = this->FreeList_;
1050        while (freelist != nullptr)
1051        {
1052          if (freelist == object)
1053            return true;
1054          freelist = freelist->Next;
1055        }
1056        return false;
1057      }
1058      case OAConfig::HBLOCK_TYPE::hbBasic:
1059      case OAConfig::HBLOCK_TYPE::hbExtended:
1060      {
1061          // Checks the flag bit.
1062          unsigned char* flagByte = reinterpret_cast<unsigned char*>(object) - configuration.PadByte
1063          return *flagByte;
1064      }
1065      case OAConfig::HBLOCK_TYPE::hbExternal:
1066      {
1067          // If we are a pointer, we are allocated. IF not, we aren't.
1068          unsigned char* header = toHeader(object);
1069          return *header;
1070      }
1071      default:
1072          return false;
1073          break;
1074      }
1075  }
1076
1077  /**************************************************************************/
1078  /*!
1079
1080    Given a pointer to an object and its header type, creates the headers for
1081    that object data.
1082
1083    \param object
```

```
1084      The pointer to the object to create a header for.
1085    \param label
1086      Only used for external labels. Labels the header.
1087    \param headerType
1088      The type of header to create
1089
1090  */
1091  /**************************************************************************/
1092  void ObjectAllocator::updateHeader(GenericObject* Object, OAConfig::HBLOCK_TYPE headerType, const
1093  {
1094      switch (headerType)
1095      {
1096      case OAConfig::hbBasic:
1097      {
1098          buildBasicHeader(Object);
1099          break;
1100      }
1101      case OAConfig::hbExtended:
1102      {
1103          // one unsigned for allocation number, and one flag to determine on or off.
1104          buildExtendedHeader(Object);
1105          break;
1106      }
1107      case OAConfig::hbExternal:
1108          {
1109          buildExternalHeader(Object, label);
1110          }
1111      default:
1112          break;
1113      }
1114  }
1115
1116  /**************************************************************************/
1117  /*!
1118
1119    Given an pointer to a data block, returns the header.
1120
1121    \param obj
1122      Pointer to data block to return header for.
1123    \return
1124      Pointer to the header for the obj passed in.
1125  */
1126  /**************************************************************************/
1127   unsigned char* ObjectAllocator::toHeader(GenericObject* obj) const

1128  {
1129      return reinterpret_cast<unsigned char*>(obj) - this->configuration.PadBytes_ - this->configura
1130  }
1131
```

```
 /**************************************************************************/
/*!

  Given an pointer to a data block, returns the left padding address.

  \param obj
    Pointer to data block to return left padding
  \return
    Pointer to the header for the obj passed in.
*/
/**************************************************************************/
unsigned char* ObjectAllocator::toLeftPad(GenericObject* obj) const
{
    return reinterpret_cast<unsigned char*>(obj) - this->configuration.PadBytes_;
}

/**************************************************************************/
/*!

  Given an pointer to a data block, returns right padding

  \param obj
    Pointer to data block to return right padding
  \return
    Pointer to the header for the obj passed in.
*/
/**************************************************************************/
unsigned char* ObjectAllocator::toRightPad(GenericObject* obj) const
{
    return reinterpret_cast<unsigned char*>(obj) + this->stats.ObjectSize_;
}

/**************************************************************************/
/*!

  Given a head node and a node, inserts the node before the head node in the
  linked list.

  \param head
    Head of the linked list to insert the new node in.
  \param node
    Node to insert before the head of the linkedl list.

*/

/**************************************************************************/
void ObjectAllocator::InsertHead(GenericObject*& head, GenericObject* node)
{
    node->Next = head;
```

```
1180        head = node;
1181    }
1182
1183    /**************************************************************************/
1184    /*!
1185
1186      Contructor for the configuration file.
1187
1188      \param UseCPPMemManager
1189        Head of the linked list to insert the new node in.
1190      \param ObjectsPerPage
1191        Node to insert before the head of the linkedl list.
1192      \param MaxPages
1193        The maximum number of pages the OA is allowed to allocate
1194      \param DebugOn
1195        Turn on debug settings for the OA
1196      \param PadBytes
1197        The number of bytes used for padding
1198      \param HBInfo
1199        The information about the header to use
1200      \param Alignment
1201        The number of bytes to align to.
1202
1203    */
1204    /**************************************************************************/
1205    OAConfig::OAConfig(bool UseCPPMemManager, unsigned int ObjectsPerPage,
1206                       unsigned int MaxPages, bool DebugOn,
1207                       unsigned int PadBytes, const OAConfig::HeaderBlockInfo &HBInfo,
1208                       unsigned int Alignment)
1209            : UseCPPMemManager_(UseCPPMemManager),
1210              ObjectsPerPage_(ObjectsPerPage),
1211              MaxPages_(MaxPages),
1212              DebugOn_(DebugOn),
1213              PadBytes_(PadBytes),
1214              HBlockInfo_(HBInfo),
1215              Alignment_(Alignment)
1216    {
1217        HBlockInfo_ = HBInfo;
1218
1219
1220        // We need to calc what is left align and the interblock alignment
1221        unsigned leftHeaderSize = static_cast<unsigned>(PTR_SIZE + HBInfo.size_ + static_cast<size_t>(
1222        LeftAlignSize_ = static_cast<unsigned>(align(leftHeaderSize, this->Alignment_) - leftHeaderSiz
1223
1224        InterAlignSize_ = 0;
1225    }
```