## jdeanmillard / ObjectAllocator.cpp Last active 6 years ago • Report abuse ☆ Star ⟨> Code ← Revisions 2

Implementation for CS 280 Object Allocator

```
ObjectAllocator.cpp
      /*!
     \file
            ObjectAllocator.cpp
  4
     \author Justin Millard
            email: justin.millard\@digipen.edu
           DigiPen login: justin.millard
  6
     \par
  7
            Course: CS280
     \par
            Programming Assignment #1
      \par
  9
      \date 1/21/2013
 10
     \brief
      This file contains the implementation for the Object Allocator.
 11
      */
      13
     #include "ObjectAllocator.h"
 15
 16
     #include <iostream>
 17
     #include <cstring>
     #include <stdlib.h>
 18
 19
     //#include "LeakCheck.h"
 20
     //#undef THIS_FILE
     //static char THIS_FILE[] = __FILE__;
 22
     #define u_char unsigned char
 23
      24
 25
      /*!
       \brief
         Creates the ObjectAllocator per the specified values. Throws an exception
         if the construction fails. (Memory allocation problem)
 29
       \param ObjectSize
         The size of each object than can be allocated.
 32
       \param config
         Contains configuration details for object allocator.
 34
```

```
ObjectAllocator::ObjectAllocator(unsigned ObjectSize, const OAConfig& config) throw(OAException)
     : Config_(config), OAStats_(), PageList(NULL), FreeList(NULL)
38
39
40
       OAStats_.ObjectSize_ = ObjectSize;
41
       if (Config_.Alignment_ != 0)
43
         if (sizeof(void*) + Config_.PadBytes_ + Config_.HeaderBlocks_ <= Config_.Alignment_)</pre>
44
         {
45
           Config .LeftAlignSize = Config .Alignment - Config .PadBytes - sizeof(void*)
46
             Config_.HeaderBlocks_;
47
         }
48
         else
49
         {
           Config_.LeftAlignSize_ = Config_.Alignment_ * 2 - Config_.PadBytes_ - sizeof(void*)
51
             Config .HeaderBlocks;
         }
54
         if (Config_.HeaderBlocks_ + Config_.PadBytes_ * 2 < Config_.Alignment_)</pre>
         {
          Config_.InterAlignSize_ = Config_.Alignment_ - Config_.HeaderBlocks_
56
             - Config_.PadBytes_ * 2;
58
         }
         else
60
         {
           Config_.InterAlignSize_ = Config_.Alignment_ * 2 - Config_.HeaderBlocks_
             - Config_.PadBytes_ * 2;
         }
63
       }
       OAStats_.PageSize_ = Config_.ObjectsPerPage_ * ( ObjectSize + Config_.PadBytes_ * 2
         + Config_.InterAlignSize_ + Config_.HeaderBlocks_ )
         - Config .InterAlignSize + Config .LeftAlignSize + sizeof(void*);
68
       if (Config_.UseCPPMemManager_ == false)
70
       {
71
         try
72
         {
73
           AddNewPage();
74
75
         catch(OAException exception)
76
         {
           throw(exception);
78
         }
79
       }
     }
81
82
     /*!
83
84
       \brief
```

```
85
        Destroys the ObjectAllocator (never throws)
     */
 86
     87
 88
     ObjectAllocator::~ObjectAllocator() throw()
 89
 90
       u_char* page;
 91
       while (PageList != NULL)
93
        page = reinterpret_cast<u_char*>(PageList);
        PageList = PageList->Next;
        delete [] page;
      }
 97
     }
98
99
     /*!
       \brief
102
        Allocates the memory for a new page, and sets up relevant pointers and
103
        signatures.
104
       \param nextPage
106
        The page following the new one.
107
     */
     109
     void ObjectAllocator::AddNewPage(GenericObject* nextPage) throw(OAException)
110
111
       u_char* address;
112
       try
113
       {
114
        address = new u_char[OAStats_.PageSize_];
115
       }
116
       catch(std::bad alloc &)
117
118
        throw(OAException(OAException::E_NO_MEMORY, "Not enough memory."));
119
       }
120
       PageList = reinterpret_cast<GenericObject*>(address);
121
122
       PageList->Next = nextPage;
       address += sizeof(void*);
       ++OAStats_.PagesInUse_;
124
125
126
       GenericObject* nextList;
127
       for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)</pre>
128
129
130
        // Get address to proper alignment
        if (i == 0)
131
132
        {
133
          if (Config_.DebugOn_ == true)
```

```
134
            {
              std::memset(address, ALIGN PATTERN, Config .LeftAlignSize );
135
136
137
            if (i < Config_.ObjectsPerPage_ - 1)</pre>
138
139
              address += Config_.LeftAlignSize_;
            }
140
141
          }
142
          else
143
          {
144
            if (Config_.DebugOn_ == true)
145
146
              std::memset(address, ALIGN PATTERN, Config .InterAlignSize );
147
            }
            address += Config_.InterAlignSize_;
148
149
          }
150
          // Add header
151
          if (Config_.DebugOn_ == true)
152
          {
153
             std::memset(address, 0, Config .HeaderBlocks );
154
          }
155
          address += Config_.HeaderBlocks_;
156
157
          // Add start padding
          if (Config_.DebugOn_ == true)
158
159
160
            std::memset(address, PAD_PATTERN, Config_.PadBytes_);
161
          }
162
          address += Config .PadBytes ;
164
          // Add data bytes and set new free list
          if (Config .DebugOn == true)
165
167
            std::memset(address, UNALLOCATED_PATTERN, OAStats_.ObjectSize_);
168
          }
169
          nextList = FreeList;
          FreeList = reinterpret_cast<GenericObject*>(address);
170
171
          ++OAStats .FreeObjects;
172
          FreeList->Next = nextList;
          address += OAStats_.ObjectSize_;
173
174
175
          // Add end padding
176
          if (Config .DebugOn == true)
177
          {
178
            std::memset(address, PAD_PATTERN, Config_.PadBytes_);
179
180
          address += Config_.PadBytes_;
181
        }
182
```

```
183
     184
     /*!
185
       \brief
186
187
         Take an object from the free list and give it to the client (simulates new)
         Throws an exception if the object can't be allocated. (Memory allocation
188
189
         problem)
190
191
       \return
192
         A pointer to the new object.
193
     194
195
     void *ObjectAllocator::Allocate() throw(OAException)
196
197
       // Ignore most of the function if using CPP Memory Manager
198
       if (Config_.UseCPPMemManager_ == true)
         u_char* address;
200
201
         try
202
         {
203
           address = new u_char[OAStats_.ObjectSize_];
204
         }
205
         catch(std::bad alloc &)
206
           throw(OAException(OAException::E_NO_MEMORY, "Not enough memory."));
207
208
         }
209
         ++OAStats_.ObjectsInUse_;
         if (OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
210
           ++OAStats_.MostObjects_;
213
         }
214
         ++OAStats .Allocations ;
         --OAStats_.FreeObjects_;
216
         return address;
217
       }
218
219
       if (PageList == NULL)
220
       {
         try
222
         {
           AddNewPage();
223
224
225
         catch(OAException exception)
226
         {
227
           throw(exception);
228
         }
229
       }
230
       else if (FreeList == NULL)
```

```
232
         if (Config .MaxPages == 0 || OAStats .PagesInUse < Config .MaxPages )</pre>
233
         {
234
           try
235
           {
236
             AddNewPage(PageList);
237
           }
238
           catch(OAException exception)
239
             throw(exception);
241
           }
242
         }
243
         else
244
         {
245
           throw(OAException(OAException::E_NO_PAGES, "No more pages can be allocated."));
         }
247
       }
248
249
       // If we've gotten this far, object can be properly given to client
250
       u char* newMem = reinterpret cast<u char*>(FreeList);
251
       FreeList = FreeList->Next;
       if (Config .DebugOn == true)
252
253
254
         std::memset(newMem, ALLOCATED PATTERN, OAStats .ObjectSize );
255
       }
       if (Config_.HeaderBlocks_ != 0)
256
257
         u_char* header = newMem - Config_.PadBytes_ - sizeof(u_char);
258
259
         *header = static_cast<u_char>(1);
260
       }
       ++OAStats_.ObjectsInUse_;
       if (OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
264
         ++OAStats_.MostObjects_;
265
       }
266
       ++OAStats .Allocations ;
       --OAStats_.FreeObjects_;
       return newMem;
268
269
     }
270
     271
272
     /*!
273
       \brief
         Returns an object to the free list for the client (simulates delete).
274
         Throws an exception if the the object can't be freed. (Invalid object)
275
276
       \param Object
277
278
         The object.
279
      280
```

```
void ObjectAllocator::Free(void *Object) throw(OAException)
282
      {
        // Skips most of the function if using CPP memory manager
283
284
        if (Config .UseCPPMemManager == true)
285
286
          u char* block = reinterpret cast<u char*>(Object);
287
          delete [] block;
288
289
          --OAStats_.ObjectsInUse_;
290
          ++OAStats .Deallocations ;
          ++OAStats_.FreeObjects_;
293
          return:
        }
294
295
296
        GenericObject* page = PageList;
        u char* objAddress = reinterpret cast<u char*>(Object);
298
        // Only perform checks when debugging
        if (Config .DebugOn == true)
        {
          bool onBound = false;
          bool corrupted = false;
          bool alreadyFreed = true;
          while (page != NULL)
          {
            // Check if object is within page boundaries
308
            if ( reinterpret_cast<u_char*>(page) < objAddress</pre>
310
              && objAddress < reinterpret_cast<u_char*>(page) + OAStats_.PageSize_)
              u char* blockStart = reinterpret cast<u char*>(page) + sizeof(void*) +
                   Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;
              size_t separation = OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 +
                                   Config .InterAlignSize + Config .HeaderBlocks ;
              // Check if object is on a block boundary
              if ( 0 == (objAddress - blockStart) % separation )
317
              {
                onBound = true;
              }
            }
            page = page->Next;
          }
          if (onBound == false)
            throw(OAException(OAException::E_BAD_BOUNDARY, "Object not on a block boundary"));
328
          }
          else
```

```
330
            // Check data block
            for (unsigned i = sizeof(u_char*); i < OAStats_.ObjectSize_; ++i)</pre>
              if ( objAddress[i] != FREED_PATTERN )
              {
                 alreadyFreed = false;
                 break;
              }
339
            }
341
            // Check padding blocks
            u char* leftPad = objAddress - Config .PadBytes ;
            u_char* rightPad = objAddress + OAStats_.ObjectSize_;
343
            for (unsigned i = 0; i < Config_.PadBytes_; ++i)</pre>
            {
              if (leftPad[i] != PAD PATTERN || rightPad[i] != PAD PATTERN)
347
              {
                 corrupted = true;
                 break;
              }
350
351
            }
352
          }
          if (alreadyFreed == true)
            throw(OAException(OAException::E_MULTIPLE_FREE, "Block already freed"));
          }
358
          else if (corrupted == true)
359
          {
            throw(OAException(OAException::E_CORRUPTED_BLOCK, "Block is corrupted"));
          }
        }
        // If we've gotten this far, we can finally free memory
        if (Config_.HeaderBlocks_ != 0)
        {
          u_char* header = objAddress - Config_.PadBytes_ - sizeof(char);
           *header = 0;
368
        }
        if (Config_.DebugOn_ == true)
370
371
           std::memset(Object, FREED PATTERN, OAStats .ObjectSize );
372
374
        GenericObject* block = reinterpret cast<GenericObject*>(Object);
        block->Next = FreeList;
        FreeList = block;
377
        --OAStats_.ObjectsInUse_;
378
```

```
++OAStats .Deallocations ;
       ++OAStats_.FreeObjects_;
381
     }
382
     383
       \brief
385
        Calls the callback fn for each block in use.
387
388
       \param fn
389
        A function pointer for the callback function.
       \return
        The number of objects in use.
393
     unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
       GenericObject* page = PageList;
       u char* block;
       while (page)
400
401
        block = reinterpret cast<u char*>(page) + sizeof(void*) + Config .LeftAlignSize
402
             + Config_.HeaderBlocks_ + Config_.PadBytes_;
        for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)</pre>
403
404
405
          if (*block == ALLOCATED PATTERN)
406
407
            fn(block, OAStats_.ObjectSize_);
408
          }
409
          if (i < Config_.ObjectsPerPage_ - 1)</pre>
410
            block += OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 + Config_.InterAlignSize_
411
412
             + Config_.HeaderBlocks_;
413
          }
414
        }
415
        page = page->Next;
416
417
       return OAStats .ObjectsInUse ;
418
419
     420
     /*!
421
       \brief
422
423
        Calls the callback fn for each block that is potentially corrupted.
424
425
       \param fn
426
        A function pointer for the callback function.
427
```

```
428
        \return
429
          The number of corrupted blocks.
430
      431
432
      unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
433
      {
434
        if (Config .PadBytes == 0)
435
436
          return 0;
437
        }
438
439
        unsigned corruptedBlocks = 0;
440
        GenericObject* page = PageList;
441
        u_char* block;
        u_char* leftPad;
442
443
        u char* rightPad;
444
        while (page)
445
446
447
          block = reinterpret cast<u char*>(page) + sizeof(void*) + Config .LeftAlignSize
448
            + Config .HeaderBlocks + Config .PadBytes ;
          for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)</pre>
449
450
          {
451
            // Check padding blocks
            leftPad = block - Config_.PadBytes_;
452
            rightPad = block + OAStats .ObjectSize ;
453
454
            for (unsigned j = 0; j < Config_.PadBytes_; ++j)</pre>
455
456
              if (leftPad[j] != PAD PATTERN || rightPad[j] != PAD PATTERN)
457
              {
458
                fn(block, OAStats_.ObjectSize_);
459
                ++corruptedBlocks;
460
                break;
461
              }
462
            }
463
            // Prevents pointer arithmetic overrun
464
465
            if (i < Config .ObjectsPerPage - 1)</pre>
466
              block += OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 + Config_.InterAlignSize_
467
468
                + Config .HeaderBlocks;
469
            }
470
          }
471
          page = page->Next;
472
        }
473
474
        return corruptedBlocks;
475
      }
476
```

```
477
     /*!
478
       \brief
479
480
        Frees all empty pages (pages whose blocks are all on the FreeList.)
481
482
       \return
483
        The number of empty pages.
484
     485
486
     unsigned ObjectAllocator::FreeEmptyPages()
487
488
       unsigned emptyPages = 0;
489
       GenericObject* currentPage = PageList;
       u_char* mem;
490
491
492
       if (!currentPage)
493
494
        return emptyPages;
495
496
497
       // Delete first page if it is empty. If subsequent page(s) are empty, delete
       // those too. Set PageList pointer at first non-empty page
498
499
       while (currentPage && IsEmpty(currentPage))
       {
        ++emptyPages;
        -- OAStats . PagesInUse ;
        RemovePageBlocks(currentPage);
        mem = reinterpret_cast<u_char*>(currentPage);
        currentPage = currentPage->Next;
        PageList = currentPage;
        delete [] mem;
       while (currentPage)
510
       {
511
        while (currentPage->Next && IsEmpty(currentPage->Next))
512
513
          ++emptyPages;
          --OAStats .PagesInUse ;
515
          RemovePageBlocks(currentPage->Next);
          mem = reinterpret_cast<u_char*>(currentPage->Next);
516
517
          currentPage->Next = currentPage->Next->Next;
          delete [] mem;
519
        }
520
        currentPage = currentPage->Next;
       }
522
       return emptyPages;
523
524
     525
```

```
/*!
526
       \brief
527
        Checks whether or not a page is empty (all its objects on the FreeList.)
528
529
530
       \param page
531
        The page to be checked.
532
533
       \return
534
        Whether or not it is empty.
535
     536
537
     bool ObjectAllocator::IsEmpty(GenericObject* page)
538
     {
539
       u_char* pageBlock = reinterpret_cast<u_char*>(page) + sizeof(void*) +
540
        Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;
541
       size_t separation = OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 +
542
        Config .InterAlignSize + Config .HeaderBlocks ;
543
544
       // Go through different blocks
545
       for (unsigned i = 0; i < Config .ObjectsPerPage ; ++i)</pre>
546
        // If any block is on the Free List, page is not empty
547
548
        if (OnFreeList(pageBlock) == false)
549
        {
550
          return false;
551
552
        if (i < Config_.ObjectsPerPage_ - 1)</pre>
553
554
          pageBlock += separation;
555
        }
556
       }
557
558
       return true;
559
     }
560
     561
562
     /*!
       \brief
564
        Checks whether or not a block is on the FreeList.
565
566
       \param block
        The address of block.
568
       \return
569
        Whether or not the block is on the FreeList.
571
     572
573
     bool ObjectAllocator::OnFreeList(unsigned char* block)
574
```

```
575
      GenericObject* currentBlock = FreeList;
576
577
      while (currentBlock != NULL)
578
        if (block == reinterpret_cast<u_char*>(currentBlock))
579
        {
581
          return true;
582
        }
583
        currentBlock = currentBlock->Next;
584
      }
585
      return false;
587
     }
588
     589
590
     /*!
      \brief
591
        Gets previous block from FreeList. Returns null if none can be found or if
592
593
        block is head of list.
594
      \param FreeListBlock
595
        The address of the current block.
596
598
      \return
        The previous block (if it exists.)
599
     601
     GenericObject* ObjectAllocator::GetPreviousBlock(GenericObject* FreeListBlock)
602
603
604
      GenericObject* currentBlock = FreeList;
606
      while (currentBlock->Next)
607
608
        if (currentBlock->Next == FreeListBlock)
609
        {
610
          return currentBlock;
611
        }
612
        currentBlock = currentBlock->Next;
613
      }
614
615
      return NULL;
616
     }
617
     618
     /*!
619
620
      \brief
        Travels through FreeList and removes the blocks that are found on the
621
622
        designated page.
623
```

```
624
        \param page
625
         The page containing the blocks to remove.
626
627
       \return
628
         The previous block (if it exists.)
629
      630
631
      void ObjectAllocator::RemovePageBlocks(GenericObject* page)
632
      {
633
       size t separation = OAStats .ObjectSize + Config .PadBytes * 2 +
634
         Config_.InterAlignSize_ + Config_.HeaderBlocks_;
635
636
       GenericObject* currentBlock;
637
       GenericObject* previousBlock;
       u_char* address = reinterpret_cast<u_char*>(page) + sizeof(void*) +
638
639
         Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;
       // Go through different blocks
641
642
       for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)</pre>
643
644
         currentBlock = reinterpret cast<GenericObject*>(address);
         previousBlock = GetPreviousBlock(currentBlock);
645
646
         // Change FreeList pointer if FreeList is found on this page
647
         if (currentBlock && FreeList == currentBlock)
648
649
           --OAStats_.FreeObjects_;
650
           FreeList = currentBlock->Next;
651
         }
652
         // Change next pointers if they point to object from this page
653
         if (previousBlock)
654
         {
655
           -- OAStats .FreeObjects;
656
           previousBlock->Next = currentBlock->Next;
657
         }
658
         if (i < Config .ObjectsPerPage - 1)</pre>
659
           address += separation;
661
         }
662
       }
664
      665
      /*!
666
       \brief
667
         Returns true if FreeEmptyPages and alignments are implemented
670
       \return
671
         Whether or not alignments and FreeEmptyPages are implemented.
672
```

```
673
674
   bool ObjectAllocator::ImplementedExtraCredit()
675
676
    return true;
677
678
   679
   /*!
680
    \brief
681
682
     Sets debug state of allocator.
683
    \param State
685
     Whether or not the allocator is in debug mode.
686
   687
   void ObjectAllocator::SetDebugState(bool State)
689
690
    Config_.DebugOn_ = State;
691
692
   693
694
695
    \brief
696
     Returns the first object on the FreeList.
697
    \return
698
699
     The first object on the FreeList.
   701
702
   const void * ObjectAllocator::GetFreeList() const
704
    return FreeList;
   707
   /*!
708
    \brief
709
710
     Returns the first object on the PageList.
711
712
    \return
713
     The first object on the PageList.
714
   715
   const void * ObjectAllocator::GetPageList() const
716
717
   {
718
    return PageList;
719
720
```

```
/*!
722
723
    \brief
      Returns the config details of the ObjectAlloator.
724
725
726
     \return
727
      A copy of the allocator's config details.
728
    729
730
    OAConfig ObjectAllocator::GetConfig() const
731
732
    return Config_;
733
734
    735
736
    /*!
    \brief
737
      Returns the stat details of the ObjectAlloator.
738
739
    \return
740
741
      A copy of the allocator's stat details.
742
    743
744
    OAStats ObjectAllocator::GetStats() const
745
746
    return OAStats_;
747
```