Commit 76dd2c59 authored 1 year ago by Alien G

Merge remote-tracking branch 'origin/master'

```
parents 358e73fc a7e535ab Pmaster ....

No related merge requests found
```

```
1 Hinslude Wetdoffy by
```

Showing 8 changed files ▼ with 1570 additions and 0 deletions

```
5 + \author Tristan Bouchard
 6 + \par email: tristan.bouchard\@digipen.edu
7 + \par DigiPen Login: tristan.bouchard
8 + \par Course: CS200
9 + \par Section: A
10 + \par Assignment #N/A
11 + \date 9/19/2019
<u>12</u> + \brief
13 + This file constains the file the implementation of the ObjectAllocator
16 + #include "Action_Memory.h"
17 + #include <cstring>
18 +
19 + /*!
20 + * \brief
21 + * Creates the ObjectManager per the specified values. Throws an exception if the construction
22 + * fails. (Memory allocation problem)
23 + *
<u>24</u> + * \param ObjectSize
25 + * The size of the object to keep track of the memory of
26 + *
<u>27</u> + * \param config
28 + * Settings for the allocator, see ObjectAllocator.h for options
30 + ObjectAllocator::ObjectAllocator(size_t ObjectSize, const OAConfig &config):
31 + Config_(config), InterBlockSize_(0) {
32 + //set the object size
33 + Stats_.ObjectSize_ = ObjectSize;
<u>34</u> +
35 + //set the page size
36 + Stats_.PageSize_ = Get_PageSize();
37 +
38 + //if the memory allocated
<u>39</u> + if (!Config_.UseCPPMemManager_)
40 + {
         InterBlockSize_ = Config_.HBlockInfo_.size_ + Stats_.ObjectSize_;
41 +
42 +
         InterBlockSize_ += Config_.InterAlignSize_ + Config_.PadBytes_ * 2;
43 +
44 +
         //allocate a page and add it to the allocator
<u>45</u> +
         Add_Page();
46 + }
47 + }
48 +
49 + /*!
50 + * \brief
<u>51</u> + * Destroys the ObjectManager (never throws)
53 + ObjectAllocator::~ObjectAllocator() {
54 + //is the memory manager is in use
55 + if (!Config_.UseCPPMemManager )
<u>56</u> + {
```

```
//loop through the page list deleting all allocated memory for the pages
 <u>57</u> +
 58 +
           while (PageList_)
 <u>59</u> +
          {
 60 +
             byte *temp = reinterpret_cast<byte *>(PageList_);
 <u>61</u> +
             PageList_ = PageList_->Next;
 <u>62</u> +
             delete[] temp;
 63 +
           }
 <u>64</u> + }
 <u>65</u> + }
 66 +
 67 + /*!
 68 + * \brief
 69 + * Takes an object from the free list and returns it as allocated memory.
           Will through if there is a memory issue
 <u>71</u> + *
 <u>72</u> + * \param Label
 73 + * A unique name given to the objects header
 <u>74</u> + *
 <u>75</u> + * \return
 <u>76</u> + * returns a pointer to the allocated memory
 77 + */
<u>78</u> +
 79 + void *ObjectAllocator::Allocate(const char *label) {
 80 + void *rtn = nullptr; //pointer to return to the client
 81 +
 82 + //if the memory manager is disabled, return an allocated void pointer
 83 + if (Config_.UseCPPMemManager_)
 84 + {
 85 +
          try
 86 +
 87 +
           //allocate memory and keep track of allocations
 88 +
            ++Stats_.Allocations_;
 89 +
            rtn = new byte[Stats_.ObjectSize_]();
 90 +
          }
          catch (std::bad_alloc &)
 91 +
 92 +
 93
             --Stats_.Allocations_;
 94 +
             throw OAException(OAException::E_NO_MEMORY, "Object can't be allocated");
 95 +
           }
 96 + }
 97 +
         else
98 + {
          //if a new page needs to be allocated
99 +
100 +
           if (Stats_.ObjectsInUse_ == Stats_.PagesInUse_ * Config_.ObjectsPerPage_)
            Add_Page();
101 +
102 +
          //remove the pointer from the free list
103 +
           *reinterpret_cast<GenericObject **>(&rtn) = FreeList_;
104 +
           FreeList_ = FreeList_->Next;
105 +
106 +
107 +
           //update header data
           Allocate_Header(Get_Header(reinterpret_cast<byte *>(rtn)), label);
108 +
109 +
110 +
          //if debug is on, add the bytes
           if (Config_.DebugOn_)
<u>111</u> +
             memset(reinterpret_cast<byte *>(rtn), ALLOCATED_PATTERN, Stats_.ObjectSize_);
112 +
<u>113</u> + }
<u>114</u> +
115 + //update objects in use and free objects
116 + ++Stats_.ObjectsInUse_;
117 + --Stats_.FreeObjects_;
118 +
119 + //check if its the most objects allocated at one time
120 + if (Stats_.ObjectsInUse_ > Stats_.MostObjects_)
           Stats_.MostObjects_ = Stats_.ObjectsInUse_;
<u>121</u> +
<u>122</u> +
123 + //return the pointer
<u>124</u> + return rtn;
125 + }
126 +
127 + /*!
<u>128</u> + * \brief
129 + * Frees memory allocated by the object allocator. Throws an exception if the the
130 + * object can't be freed. (Invalid object)
131 + *
```

```
<u>132</u> + * \param Object
133 + * A unique name given to the objects header
134 + *
<u>135</u> + * \return
136 + * returns a pointer to the allocated memory
137 + */
138 + void ObjectAllocator::Free(void *Object) {
139 + //if the memory manager
140 + if (Config_.UseCPPMemManager_)
<u>141</u> + {
142 +
         //Track deallocations and free the memory
143 +
          ++Stats_.Deallocations_;
144 +
           delete[] reinterpret_cast<byte *>(Object);
145 +
           return;
<u>146</u> + }
147 +
148 + //if the debug option is on, check for bounds, double freed and corruptions
149 + if (Config_.DebugOn_)
<u>150</u> + {
<u>151</u> +
           //check if a block has already been freed throw an exception if so
152 +
           if (Freed(reinterpret_cast<byte *>(Object)))
<u>153</u> +
             throw OAException(OAException::E_MULTIPLE_FREE,
154 +
                               "Object has already been freed or freed out of bounds");
<u>155</u> +
           //check if the object is out of bounds
156 +
           if (OutOfBounds(reinterpret_cast<byte *>(Object)))
<u>157</u> +
158 +
             throw OAException(OAException::E_BAD_BOUNDARY, "Object out of range of its allocated memory");
159 +
160 +
           //check if the pad bytes around the object is corrupted/overwritten. if so, throw an exception
161 +
           if (Corrupted(reinterpret_cast<byte *>(Object)))
162 +
             throw OAException(OAException::E_CORRUPTED_BLOCK,
163 +
                               "Pad bytes adjacent to object ptr has been overwritten");
164 +
165 +
           //if debug is on, add the bytes
166 +
           memset(reinterpret_cast<byte *>(Object), FREED_PATTERN, Stats_.ObjectSize_);
<u>167</u> + }
168 +
169 + //add the object back to the free list
170 +
        GenericObject *obj = reinterpret_cast<GenericObject *>(Object);
<u>171</u> +
        obj->Next = FreeList_;
172 +
        FreeList_ = obj;
<u>173</u> +
174 + //free/update the header block attached to the data block
175 +
        Free_Header(Get_Header(reinterpret_cast<byte *>(Object)));
176 +
<u>177</u> + //update states for freeing
178 + --Stats_.ObjectsInUse_;
179 + ++Stats_.FreeObjects_;
180 + ++Stats_.Deallocations_;
<u>181</u> + }
<u>182</u> +
183 + /*!
<u>184</u> + * \brief
185 + * Calls the function passed in on every object that is currently allocated within the page list
186 + *
<u>187</u> + * \param fn
188 + * function to run on all allocated blocks of data
189 + *
<u>190</u> + * \return
191 + *
            returns the number of blocks of data allocated within the allocator
192
    + */
193
    + unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const {
194
    + //if there are no pad bytes or debug is off, return zero
195
         if (!Config_.DebugOn_ || Config_.PadBytes_)
196 +
           return 0;
<u>197</u> +
        //pointer to the page list
198 +
199 +
         GenericObject *run = PageList_;
200 +
201 +
        //number of objects allocated
         unsigned objs_allocated = 0;
202 +
203 +
204 +
        //loop through all the pages
205 +
         while (run)
206 + {
```

```
207 +
          //get the start of a page
208 +
          byte *page = reinterpret_cast<byte *>(run);
209 +
210 +
          //move to the first data block of data
211 +
          page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
212 +
213 +
          //loop through the page checking if each block is in use
214 +
          for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)</pre>
215 +
216 +
            //check if the memory is in use
217 +
            if (!Freed(page))
218 +
            {
219 +
              //print out the memory and add one to the allocated number
220 +
              fn(page, Stats_.ObjectSize_);
221 +
              ++objs_allocated;
222 +
            }
223 +
          }
224 +
225 +
          //move to the next page
226 +
          run = run->Next;
227 + }
228 +
229 + //return the number of objects in use by the memory
230 + return objs_allocated;
231 + }
232 +
233 + /*!
234 + * \brief
235 + * Calls the callback fn for each block that is potentially corrupted
236 + *
237 + * \param fn
238 + * callback function for using on every block of memory that might be corrupted
239 + *
<u>240</u> + * \return
241 + * the number of objects corrupted
242 + */
243 + unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const {
244 + //if there are no pad bytes or debug is off, return zero
245 + if (!Config_.DebugOn_ || !Config_.PadBytes_)
246 +
          return ∅;
247 +
248 + //pointer to the page list
249 +
        GenericObject *run = PageList_;
250 +
        //number of objects corrupted
<u>251</u> +
252 +
        unsigned objs_corrupted = 0;
253 +
254 +
        //loop through all the pages
        while (run)
<u>255</u> +
<u>256</u> + {
257 +
          //get the start of a page
<u>258</u> +
          byte *page = reinterpret_cast<byte *>(run);
259 +
260 +
          //move to the first data block of data
          page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
261 +
262
          //loop through the page checking if each block is corrupted
263 +
264 +
          for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)</pre>
265
266 +
            //check if the pad bytes around the block corrupted
267 +
            if (Corrupted(page))
268 +
            {
269 +
              //print out the memory and add one to the corrupted count
              fn(page, Stats_.ObjectSize_);
270 +
              ++objs_corrupted;
271 +
272 +
            }
273 +
          }
274 +
<u>275</u> +
          //move to the next page
276 +
          run = run->Next;
277 +
        }
278 +
279 +
        //return the number of objects corrupted
280 +
        return objs_corrupted;
<u>281</u> + }
```

```
282 +
283 + /*!
284 + * \brief
285 + * Frees all empty pages
286 + *
287 + * \return
288 + * returns the number of pages freed
289 + */
290
    + unsigned ObjectAllocator::FreeEmptyPages() {
291
    + GenericObject *run = PageList_; //pointer to the page list
292 +
293 +
        //number of objects corrupted
294 +
        unsigned pages_freed = 0;
295 +
296 + //loop through all the pages
297 +
        while (run)
<u>298</u> + {
          //get the start of a page
299 +
300 +
          byte *page = reinterpret_cast<byte *>(run);
301 +
          if (PageFree(page))
302 +
303 +
            RemoveFreeListPage(page); //reorder the free list to not include the page about to free
304 +
            RemoveNode(&PageList_, run); //remove the page from the page list
305
306 +
            byte *temp = page;
307 +
            run = run->Next;
308 +
            delete[] temp;
309 +
310 +
            //increase freed pages
<u>311</u> +
            --Stats_.PagesInUse_;
312
            Stats_.FreeObjects_ -= Config_.ObjectsPerPage_;
    +
313 +
            ++pages_freed;
314 +
          }
<u>315</u> +
          else
316 +
          {
317 +
            //move to the next page
318 +
            run = run->Next;
<u>319</u> +
          }
<u>320</u> + }
321 +
322 + //return the number of objects corrupted
323 + return pages_freed;
324 + }
<u>325</u> +
326 + // Returns true if FreeEmptyPages and alignments are implemented
327 + /*!
328 + * \brief
329 + * checks if the extra credit is finished
330 + *
331 + * \return
332 + * returns true if the extra credit is implemented
333 + */
334  + bool ObjectAllocator::ImplementedExtraCredit() {
335 + //return true ;)
336 + return true;
<u>337</u> + }
338 +
339 + // Testing/Debugging/Statistic methods
340 + /*!
341 + * \brief
342 + * sets the debug state of the allocator
343 + *
<u>344</u> + * \param State
345 + * true for debug on, false for debug false
346 + */
347 + void ObjectAllocator::SetDebugState(bool State) {
348 + Config_.DebugOn_ = State;
349 + }
350 +
<u>351</u> + /*!
352 + * \brief
353 + * Gets the free list from the object allocator. Cast to GenericObject* to access the linked list
354 + *
355 + * \return
356 + * returns the start of the free list.
```

```
357 + */
358 + const void *ObjectAllocator::GetFreeList() const // returns a pointer to the internal free list
<u>359</u> + {
360 + return FreeList_;
<u>361</u> + }
362 +
<u>363</u> + /*!
364 + * \brief
365 + * Gets the page list from the object allocator. Cast to GenericObject* to access the linked list
366 + *
<u>367</u> + * \return
368 + * returns the head of the page list
369 + */
370 + const void *ObjectAllocator::GetPageList() const // returns a pointer to the internal page list
<u>371</u> + {
372 + return PageList_;
<u>373</u> + }
374 +
<u>375</u> + /*!
376 + * \brief
377 + * Gets a copy of the config used for the allocator
<u>378</u> + *
379 + * \return
380 + * returns a copy of the config object passed in when creating the allocator
381 + */
382 + OAConfig ObjectAllocator::GetConfig() const // returns the configuration parameters
383 + {
384 + return Config_;
<u>385</u> + }
386 +
<u>387</u> + /*!
388 + * \brief
389 + * Gets a copy of the stats used for the allocator
390 + *
391 + * \return
392 + * returns the updated stats of the allocator
393 + */
394 + OAStats ObjectAllocator::GetStats() const // returns the statistics for the allocator
395 + {
396 + return Stats_;
<u>397</u> + }
398 +
399 + /*!
<u>400</u> + * \brief
401 + * Helper function for updating the header block of an object during allocations
402 + *
<u>403</u> + * \param header
404 + * the first byte of the header block
405 + *
406 + * \param Label
407 + * the label to attach to the external header block
408 + */
409 + void ObjectAllocator::Allocate_Header(byte *header, const char *label) {
410 + //update allocations_
411 + ++Stats_.Allocations_;
412 +
413 + //if basic, alloc # + flag byte
414 + if (Config_.HBlockInfo_.type_ == OAConfig::hbBasic)
415 + {
416 +
          //add one to the allocation count
417 +
           *reinterpret_cast<unsigned *>(header) = Stats_.Allocations_;
418 +
419 +
          //set the bool flag to true
420 +
           *reinterpret_cast<bool *>(header + sizeof(unsigned)) = true;
421 +
422 +
423 +
          //if extended, user-defined + use counter + alloc # + flag byte
424 +
         else
425 +
           if (Config_.HBlockInfo_.type_ == OAConfig::hbExtended)
426 +
            //add one to the re-use count
427 +
            ++*reinterpret_cast<short *>(header + Config_.HBlockInfo_.additional_);
428 +
429 +
430 +
            //add one to the allocation count
             *reinterpret_cast<unsigned *>(header + Config_.HBlockInfo_.additional_ +
431 +
```

```
432
                                             sizeof(short)) = Stats_.Allocations_;
433
434
              //set the bool flag to true
435
              *reinterpret_cast<bool *>(header + Config_.HBlockInfo_.additional_ + sizeof(short) +
436
                                         sizeof(unsigned)) = true;
437
    +
           }
438
439
    +
             //if external, allocate obj
440
            else
<u>441</u>
              if (Config_.HBlockInfo_.type_ == OAConfig::hbExternal)
442
443
                //get a pointer size of memory from the page
444
                MemBlockInfo **header_block = reinterpret_cast<MemBlockInfo **>(header);
<u>445</u>
                try
446 +
                {
447
                  //allocate the pointer
448
                  (*header_block) = new MemBlockInfo;
449 +
450
                catch (std::bad_alloc &)
<u>451</u>
                  //throw if no memory
<u>452</u>
453
                  throw OAException(OAException::E_NO_MEMORY, "Unable to allocate external header");
                }
454
<u>455</u>
                //if there is a label to apply
<u>456</u>
<u>457</u>
                if (label)
                {
<u>458</u>
459
                  try
460
                    //allocate the char array
461
                    (*header_block)->label = new char[strlen(label) + 1]();
462
463
                  catch (std::bad_alloc &)
464
<u>465</u>
                    //throw of there is no memory
466
467
                    throw OAException(OAException::E_NO_MEMORY, "Unable to allocate external header label");
468
                  //copy the label into the headers label
469
                  strcpy((*header_block)->label, label);
<u>470</u>
                }
<u>471</u>
<u>472</u>
<u>473</u>
                //set the in use to in use
474
                (*header_block)->in_use = true;
<u>475</u>
<u>476</u>
                //set the allocation number
<u>477</u>
                (*header_block)->alloc_num = Stats_.Allocations_;
478 +
              }
479 + }
480 +
481 + /*!
<u>482</u>
<u>483</u>
            Helper function for updating the header block of an object during freeing
484
     + * \param header
485
            the first byte of the header block
<u>486</u>
488 + void ObjectAllocator::Free_Header(byte *header) {
489 + //if external, allocate obj
490 +
         if (Config_.HBlockInfo_.type_ == OAConfig::hbExternal)
491 +
492 +
           //get pointer to memory
493 +
           MemBlockInfo *head_ptr = *reinterpret_cast<MemBlockInfo **>(header);
494 +
495 +
           //delete the label attached to the header
           delete[] head_ptr->label;
496 +
497 +
498 +
           head_ptr->label = nullptr;
499 +
500 +
           //delete the header
501 +
            delete head_ptr;
<u>502</u> + }
503 +
<u>504</u> +
         //set the extended header to zeros keeping the fist two bytes with the object
<u>505</u> +
         if (Config_.HBlockInfo_.type_ == OAConfig::hbExtended)
<u>506</u> + {
```

```
memset(header, 0, Config .HBlockInfo .additional );
507 +
           memset(header + Config_.HBlockInfo_.additional_ + sizeof(short), 0, Config_.HBlockInfo_.size_ - sizeof(short) -
508 +
      Config_.HBlockInfo_.additional_);
<u>509</u> + }
510 +
           //set the memory to zero if a block is present
511 + else if (Config_.HBlockInfo_.type_ != OAConfig::hbNone)
<u>512</u> + {
513 +
           memset(header, 0, Config_.HBlockInfo_.size_);
<u>514</u> + }
<u>515</u> + }
<u>516</u> +
517 + /*!
<u>518</u> + * \brief
519 + * given a header block, check if the data has been freed
520 + *
<u>521</u> + * \param object
522 + * the first byte of the object data
523 + *
<u>524</u> + * \return
525 + * return if the object passed in has been freed
526 + */
527 + bool ObjectAllocator::Freed(byte *object) const {
if (Config_.HBlockInfo_.type_ != OAConfig::hbNone)
529 + {
<u>530</u> +
          //check to see if the block is extended
531 +
           size_t offset = (Config_.HBlockInfo_.type_ == OAConfig::hbExtended ? Config_.HBlockInfo_.additional_ +
       sizeof(short): 0);
532 +
533 +
          //compare the header with an array of Os
534 +
           byte *arr = new byte[Config_.HBlockInfo_.size_]();
<u>535</u> +
          bool result = !memcmp(Get_Header(object) + offset, arr, Config_.HBlockInfo_.size_ - offset);
536 +
           delete[] arr;
<u>537</u> +
           return result;
<u>538</u> + }
539 +
540 + //check if the object passed in is on the free list
541 + GenericObject *run = FreeList_;
542 +
         while (run)
<u>543</u> + {
544 +
          //pointer to first element of a page
545 +
          if (object == reinterpret_cast<byte *>(run))
546 +
             return true;
547 +
<u>548</u> +
          //move to the next page
549 +
           run = run->Next;
<u>550</u> + }
551 +
552 + return false;
<u>553</u> + }
<u>554</u> +
555 + /*!
556 + * \brief
<u>557</u> + * Calculates and returns page size
<u>558</u> + * \return
<u>559</u> + * returns the size of the page in bytes
560 + */
561 + size_t ObjectAllocator::Get_PageSize() {
562 + //check if alignment needs to be calculated
         if (Config_.Alignment_ > 1)
563
<u>564</u> +
           //calculate the bytes needed for the left alignment
<u>565</u> +
<u>566</u> +
           size_t front_bytes = sizeof(void *) + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
           Config_.LeftAlignSize_ =
567 +
568 +
                   Config_.Alignment_ - (static_cast<unsigned>(front_bytes) % Config_.Alignment_);
569 +
           //calculate the bytes needed for the inter block alignment
<u>570</u> +
           size_t inter_bytes = Stats_.ObjectSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_ * 2;
<u>571</u> +
<u>572</u> +
           Config_.InterAlignSize_ =
                   Config_.Alignment_ - (static_cast<unsigned>(inter_bytes) % Config_.Alignment_);
<u>573</u> +
<u>574</u> + }
575 +
576 + //size of the page
577 + size_t size = 0;
578 +
579 + // next page pointer + left alignment block (only the first block has left alignment)
```

```
580 + size = sizeof(void *) + Config .LeftAlignSize ;
581 +
582 + // Each object has a header, left padding, the object itself, right padding
583 + // (The left and right padding blocks are the same size)
584 + size += Config_.ObjectsPerPage_ *
585 +
                 (Config_.HBlockInfo_.size_ + Config_.PadBytes_ + Stats_.ObjectSize_ + Config_.PadBytes_);
586 +
587 + // All but the first block have an inter-block alignment block
588 + size += (Config_.ObjectsPerPage_ - 1) * Config_.InterAlignSize_;
<u>589</u> +
590 + return size;
<u>591</u> + }
<u>592</u> +
593 + /*!
<u>594</u> + * \brief
595 + * allocates the memory size of one page. will throw if page cant be allocated
<u>596</u> + * \return
597 + * returns allocated pointer
598 + */
599 + ObjectAllocator::byte *ObjectAllocator::Allocate_Page() {
<u>600</u> + try
601 + {
602
          //return allocated memory filled with UNALLOCATED_PATTERN value
603 +
          return reinterpret_cast<byte *>(new byte[Stats_.PageSize_]());
604 + }
605 + catch (std::bad_alloc &)
606 + {
607 +
          throw OAException(OAException::E_NO_MEMORY, "Unable to allocate page");
608 + }
609 + }
610 +
611 + /*!
612 + * \brief
613 + * Populates a page with all debug pattern bytes
<u>614</u> + * \param page
615 + *
          the first byte of a page
616 + */
617 + void ObjectAllocator::Populate_Page(byte *page) {
618 + //set Null values for Page list pointer
619 +
        memset(page, 0, sizeof(void *));
620 +
621 + //set the alignment bytes for the left block
        memset(page + sizeof(void *), ALIGN_PATTERN, Config_.LeftAlignSize_);
622 +
623 +
624 +
        //sets header, padding, datam padding
625 +
        auto Set_Block_Bytes = [&](byte *head_pos) {
            //header
626 +
627 +
            memset(head_pos, 0, Config_.HBlockInfo_.size_);
628 +
629 +
            //padding
            memset(head_pos + Config_.HBlockInfo_.size_, PAD_PATTERN, Config_.PadBytes_);
630 +
631 +
632 +
            //padding
633 +
            memset(head_pos + Config_.HBlockInfo_.size_ + Config_.PadBytes_ + Stats_.ObjectSize_,
                   PAD_PATTERN, Config_.PadBytes_);
634 +
             return void(0);
635 +
<u>636</u> + };
637 +
        //The total size of the inter block bytes minus the alignment
639 +
         size_t InterBlockSizeIsh = InterBlockSize_ - Config_.InterAlignSize_;
640 +
641 +
        //Run through the list setting the bytes of each inter block
642
         for (size_t i = 0, offset = sizeof(void *) + Config_.LeftAlignSize_;
643
             i < Config_.ObjectsPerPage_; ++i, offset += InterBlockSizeIsh)</pre>
644 +
645 +
          //alignment when i != 0
646 +
          if (i)
647 +
648 +
            memset(page + offset, ALIGN_PATTERN, Config_.InterAlignSize_);
649 +
             offset += Config_.InterAlignSize_;
650 +
          }
651 +
           //offsets in relation to the start of the header
652 +
653 +
           Set_Block_Bytes(page + offset);
654 + }
```

```
655 + }
656 +
657 + /*!
658 + * \brief
659 + *
           allocates and adds a page to the front of the page list
660 + */
661 + void ObjectAllocator::Add_Page() {
662 + //checks if page count max has been reached
if (Config_.MaxPages_ && (Stats_.PagesInUse_ == Config_.MaxPages_))
664 +
          throw OAException(OAException::E_NO_PAGES, "Max pages have been reached");
665 +
666 + //allocate a new block of memory
667 +
        byte *new_page = Allocate_Page();
668 +
669 + //populate the page for debugging
670 + if (Config_.DebugOn_)
<u>671</u> + {
672 +
          //mem set the whole page with unallocated pattern
673 +
          memset(new_page, UNALLOCATED_PATTERN, Stats_.PageSize_);
674 +
675 +
          //populate the page with all the proper patterns
676 +
          Populate_Page(new_page);
<u>677</u> + }
678 +
679 +
        //update pages in use and Free objects
        ++Stats_.PagesInUse_;
680 +
<u>681</u> +
        Stats_.FreeObjects_ += Config_.ObjectsPerPage_;
682 +
683 + //insert the new page into the from of the page list
684 +
        GenericObject *front_bytes = reinterpret_cast<GenericObject *>(new_page);
685 + front_bytes->Next = PageList_;
686 +
        PageList_ = front_bytes;
687 +
688 + //move to where the block of data would be
689 +
        new_page +=
690 +
                sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
691 +
692 + //setup pointers for freed list
        for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, new_page += InterBlockSize_)</pre>
693 +
694 + {
695 +
          reinterpret_cast<GenericObject *>((new_page))->Next = FreeList_;
          FreeList_ = reinterpret_cast<GenericObject *>(new_page);
696 +
697 + }
698 + }
699 +
700 + /*!
701 + * \brief
702 + * given a block of data, gets the header attached to that block
703 + *
<u>704</u> + * \param object
705 + * the first byte of the object that you want to get the header of
706 + * \return
707 + */
708 + ObjectAllocator::byte *ObjectAllocator::Get_Header(byte *object) const {
709 + return object - Config_.PadBytes_ - Config_.HBlockInfo_.size_;
<u>710</u> + }
<u>711</u> +
712 + /*!
713 + * \brief
714 + * checks if a block of data is corrupted or not
715 + *
<u>716</u> + * \param object
717 + * the first byte of the object you want to check for corruption for
718 + *
719 + * \return
720 + * returns true if the block is corrupted
721 + */
722 + bool ObjectAllocator::Corrupted(byte *object) const {
723 + try
724 + {
725 +
          //allocate a string to compare with the pad bytes on the left and right
<u>726</u> +
           const byte *comp_str = reinterpret_cast<byte *>(memset(new byte[Config_.PadBytes_ + 1](),
                                                                PAD_PATTERN, Config_.PadBytes_));
727 +
<u>728</u> +
           if (memcmp(comp_str, object + Stats_.ObjectSize_, Config_.PadBytes_) ||
              memcmp(comp_str, object - Config_.PadBytes_, Config_.PadBytes_))
729 +
```

```
<u>730</u> +
731 +
             //return that the block has been corrupted
732
             delete[] comp_str;
733 +
             return true;
<u>734</u> +
735 +
           delete[] comp_str;
<u>736</u> + }
737 +
           //throw if the comp string couldn't be allocated
738 + catch (std::bad_alloc &)
739 + {
740 +
           throw OAException(OAException::E_NO_MEMORY,
<u>741</u> +
                             "String for comparing pad bytes unable to be allocated");
<u>742</u> + }
743 +
744 + //the memory is not corrupted
745 + return false;
<u>746</u> + }
747 +
748 + /*!
749 + * \brief
750 + * Checks if an object is out of bounds
751 + *
<u>752</u> + * \param object
753 + * the object to check
754 + *
755 + * \return
756 + * returns true if the object passed in is out of bounds
757 + */
758 + bool ObjectAllocator::OutOfBounds(byte *object) const {
<u>759</u> + //pointer to the head of the page list
        GenericObject *run = PageList_;
760 +
761 +
         size_t Block_Alignment =
762 +
                 sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
763 +
764 + //go though all the pages
765 +
         while (run)
<u>766</u> + {
767 +
           //pointer to first element of a page
768 +
           byte *page = reinterpret_cast<byte *>(run);
769 +
          long long int difference = object - page;
770 +
<u>771</u> +
          //check if the object is within the page
772 +
           if (difference >= 0 && difference < static_cast<long long int>(Stats_.PageSize_))
773 +
            //check if the position of the object is a multiple of all the other objects
774 +
<u>775</u> +
             return (difference % (InterBlockSize_) != Block_Alignment);
776 +
           }
<u>777</u> +
778 +
           //move to the next page
779 +
           run = run->Next;
<u>780</u> + }
<u>781</u> +
782 + return false;
<u>783</u> + }
<u>784</u> +
<u>785</u> + /*!
786 + * \brief
787 + * Checks if a page contains only elements within the free list
788 + *
789 + * \param page
790 + * the first byte of the page
791 + *
792 + * \return
793 + * returns true if the page is free
794 + */
795 + bool ObjectAllocator::PageFree(byte *page) const {
796 + //move to the first data block of data
<u>797</u> +
        page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
798 +
799 + //loop through the page checking if each block is corrupted
800 + for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)</pre>
801 + {
802 +
           //check if the pad bytes around the block corrupted
803 +
           if (!Freed(page))
             return false;
804 +
```

```
805 + }
806 +
807 + return true;
808 + }
809 +
810 + /*!
<u>811</u> + * \brief
812 + * Reorders the free list to not include the page passed in
813 + *
<u>814</u> + * \param page
815 + * the page to remove the free list from
816 + */
817 + void ObjectAllocator::RemoveFreeListPage(byte *page) {
818 + //move to where the block of data would be
# page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
820 +
821 + //setup pointers for freed list
# for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)</pre>
823 + {
824 +
          RemoveNode(&FreeList_, reinterpret_cast<GenericObject *>((page)));
825 + }
826 + }
827 +
828 + /*!
829 + * \brief
830 + * removes a node from a list
831 + *
<u>832</u> + * \param List
833 + * the list you want to remove the node from
834 + *
<u>835</u> + * \param node
836 + * the node to remove
837 + */
# + void ObjectAllocator::RemoveNode(GenericObject **list, GenericObject *node)
839 + {
840 + GenericObject *hold = *list; //pointer holder for removed node
841 + GenericObject *remove = *list; //pointer holder for removed node
842 +
843 + //move to where the node is
844 + while (remove != node && remove->Next)
845 + {
846 +
          hold = remove;
847 +
        remove = remove->Next;
848 + }
849 +
850 + //to avoid issues with null
851 + if (!remove) return;
852 +
853 + if (remove == *list) //if its the front of the list
          *list = remove->Next; //set new head
854 +
<u>855</u> + else
          hold->Next = remove->Next; //move to next node
856 +
857 + }
      \ No newline at end of file
```

Rainier/src/Core/Actions/Action Memory.h 0 → 100644

```
<u>3</u> + \file ObjectAllocator.h
4 + \author Tristan Bouchard
5 + \sqrt{par}
          email: Tristan.Bouchard\@digipen.edu
6 + \par DigiPen Login: tristan.bouchard
<u>7</u> + \par Course: CS280
8 + \par Section A
9 + \par Assignment #1
<u>10</u> + \date 9/19/2019
<u>11</u> + \brief
12 + This file contains the prototypes for objects and functions for the
<u>13</u> + ObjectAllocator
14 + */
16 + #ifndef ACTION_MEMORY_H
17 + #define ACTION_MEMORY_H
18 + #include <string>
```

```
<u>19</u> +
<u>20</u> + //! If not specified, the objects per page will be this
21 + static const int DEFAULT_OBJECTS_PER_PAGE = 4;
22 +
23 + //! If not specified, the max pages will be this
4 + static const int DEFAULT_MAX_PAGES = 3;
<u>25</u> +
26 + /*!
27 + Exception class
28 + */
<u>29</u> + class OAException
30 + {
<u>31</u> + public:
<u>32</u> +
           /*!
<u>33</u> +
            Possible exception codes
<u>34</u> +
           */
<u>35</u> +
           enum OA_EXCEPTION
<u>36</u> +
<u>37</u> +
             E_NO_MEMORY,
                                //!< out of physical memory (operator new fails)</pre>
38 +
             E_NO_PAGES,
                                //!< out of logical memory (max pages has been reached)</pre>
<u>39</u> +
             E_BAD_BOUNDARY, //!< block address is on a page, but not on any block-boundary
<u>40</u> +
             E_MULTIPLE_FREE, //!< block has already been freed</pre>
<u>41</u> +
             E_CORRUPTED_BLOCK //!< block has been corrupted (pad bytes have been overwritten)
<u>42</u> +
           };
43 +
44 +
           /*!
45 +
             Constructor
<u>46</u> +
<u>47</u> +
             \param ErrCode
              One of the 5 error codes listed above
48 +
49 +
<u>50</u> +
             \param Message
<u>51</u> +
               A message returned by the what method.
<u>52</u> +
           */
53 +
           OAException(OA_EXCEPTION ErrCode, const std::string& Message) : error_code_(ErrCode), message_(Message) {};
<u>54</u> +
<u>55</u> +
           /*!
<u>56</u> +
           Destructor
<u>57</u> +
           */
<u>58</u> +
           virtual ~OAException() {
<u>59</u> +
60 +
<u>61</u> +
           /*!
            Retrieves the error code
62 +
63 +
             \return
64 +
               One of the 5 error codes.
65 +
66 +
           OA_EXCEPTION code() const {
<u>67</u> +
68 +
             return error_code_;
<u>69</u> +
           }
<u>70</u> +
<u>71</u> +
<u>72</u> +
             Retrieves a human-readable string regarding the error.
<u>73</u> +
<u>74</u> +
               The NUL-terminated string representing the error.
<u>75</u> +
<u>76</u> +
           virtual const char *what() const {
77 +
<u>78</u> +
             return message_.c_str();
<u>79</u> +
80 + private:
81 + OA_EXCEPTION error_code_; //!< The error code (one of the 5)</pre>
           std::string message_; //!< The formatted string for the user.</pre>
<u>83</u> + };
84 +
<u>85</u> +
86 + /*!
<u>87</u> + ObjectAllocator configuration parameters
88 + */
89 + struct OAConfig
90 + {
91 + static const size_t BASIC_HEADER_SIZE = sizeof(unsigned) + 1; //!< allocation number + flags
92 + static const size_t EXTERNAL_HEADER_SIZE = sizeof(void*); //!< just a pointer</pre>
93 +
```

```
94 + /*!
 95
           The different types of header blocks
 96
         */
 97
         enum HBLOCK_TYPE{hbNone, hbBasic, hbExtended, hbExternal};
 98
    +
99 +
         /*!
100
           POD that stores the information related to the header blocks.
101
         */
         struct HeaderBlockInfo
<u>103</u> + {
104 +
           HBLOCK_TYPE type_; //!< Which of the 4 header types to use?
           size_t size_;
105 +
                              //!< The size of this header
106 +
           size_t additional_; //!< How many user-defined additional bytes</pre>
107 +
108 +
           /*!
109 +
             Constructor
110 +
<u>111</u> +
             \param type
112 +
              The kind of header blocks in use.
113 +
114 +
             \param additional
<u>115</u> +
               The number of user-defined additional bytes required.
116 +
<u>117</u> +
           HeaderBlockInfo(HBLOCK_TYPE type = hbNone, unsigned additional = 0): type_(type), size_(0),
118 +
       additional_(additional)
119 +
           {
120 +
             if (type_ == hbBasic)
               size_ = BASIC_HEADER_SIZE;
121
122
             else if (type_ == hbExtended) // alloc # + use counter + flag byte + user-defined
123
               size_ = sizeof(unsigned int) + sizeof(unsigned short) + sizeof(char) + additional_;
124
             else if (type_ == hbExternal)
125 +
               size_ = EXTERNAL_HEADER_SIZE;
126 +
           };
127 +
         };
128 +
129 +
         /*!
130 +
           Constructor
131 +
132 +
           \param UseCPPMemManager
133 +
            Determines whether or not to by-pass the OA.
<u>134</u> +
135 +
           \param ObjectsPerPage
            Number of objects for each page of memory.
136 +
137 +
138 +
           \param MaxPages
             Maximum number of pages before throwing an exception. A value
139 +
140 +
             of 0 means unlimited.
141 +
142 +
           \param DebugOn
             Is debugging code on or off?
143 +
144 +
145 +
           \param PadBytes
             The number of bytes to the left and right of a block to pad with.
146 +
147 +
           \param HBInfo
148 +
             Information about the header blocks used
149 +
150 +
           \param Alignment
<u>151</u>
             The number of bytes to align on.
152
<u>153</u> +
<u>154</u> +
         OAConfig(bool UseCPPMemManager = false,
<u>155</u> +
                  unsigned ObjectsPerPage = DEFAULT_OBJECTS_PER_PAGE,
                  unsigned MaxPages = DEFAULT_MAX_PAGES,
156 +
<u>157</u> +
                  bool DebugOn = false,
<u>158</u> +
                  unsigned PadBytes = 0,
                  const HeaderBlockInfo &HBInfo = HeaderBlockInfo(),
159 +
                  unsigned Alignment = 0) : UseCPPMemManager_(UseCPPMemManager),
160 +
                                            ObjectsPerPage_(ObjectsPerPage),
161 +
162 +
                                            MaxPages_(MaxPages),
<u>163</u> +
                                            DebugOn_(DebugOn),
164 +
                                            PadBytes_(PadBytes),
165 +
                                            HBlockInfo_(HBInfo),
166
                                            Alignment_(Alignment)
167 +
```

```
168 +
           HBlockInfo = HBInfo;
169 +
           LeftAlignSize_ = 0;
170 +
          InterAlignSize_ = 0;
<u>171</u> + }
<u>172</u> +
<u>173</u> +
        bool UseCPPMemManager_;
                                   //!< by-pass the functionality of the OA and use new/delete
174 + unsigned ObjectsPerPage_; //!< number of objects on each page
<u>175</u> +
        unsigned MaxPages_;
                                    //!< maximum number of pages the OA can allocate (0=unlimited)
<u>176</u> + bool Debug0n_;
                                     //!< enable/disable debugging code (signatures, checks, etc.)</pre>
177 + unsigned PadBytes_;
                                   //!< size of the left/right padding for each block
178 + HeaderBlockInfo HBlockInfo_; //!< size of the header for each block (0=no headers)
179 + unsigned Alignment_; //!< address alignment of each block
180 + unsigned LeftAlignSize_; //!< number of alignment bytes required to align first block
181 + unsigned InterAlignSize_; //!< number of alignment bytes required between remaining blocks
<u>182</u> + };
183 +
184 +
185 + /*!
186 + POD that holds the ObjectAllocator statistical info
187 + */
188 + struct OAStats
<u>189</u> + {
190 + /*!
191 +
          Constructor
192 + */
193 + OAStats(): ObjectSize_(0), PageSize_(0), FreeObjects_(0), ObjectsInUse_(0), PagesInUse_(0),
194 +
                        MostObjects_(0), Allocations_(0), Deallocations_(0) {};
195 +
196 + size_t ObjectSize_; //!< size of each object</pre>
197 + size_t PageSize_;
                              //!< size of a page including all headers, padding, etc.
198 + unsigned FreeObjects_; //!< number of objects on the free list
199 + unsigned ObjectsInUse_; //!< number of objects in use by client
200 + unsigned PagesInUse_; //!< number of pages allocated</pre>
201 + unsigned MostObjects_; //!< most objects in use by client at one time
<u>202</u> + unsigned Allocations_; //!< total requests to allocate memory
203 + unsigned Deallocations_; //!< total requests to free memory</pre>
<u>204</u> + };
205 +
206 + /*!
207 + This allows us to easily treat raw objects as nodes in a linked list
208 + */
209 + struct GenericObject
210 + {
211 + GenericObject *Next = 0; //!< The next object in the list
<u>212</u> + };
213 +
214 + /*!
215 + This is used with external headers
216 + */
217 + struct MemBlockInfo
218 + {
                              //!< Is the block free or in use?
<u>219</u> + bool in_use = 0;
<u>220</u> + char *label = 0;
                             //!< A dynamically allocated NUL-terminated string
221 + unsigned alloc_num = 0; //!< The allocation number (count) of this block
<u>222</u> + };
223 +
224 + /*!
225 + This class represents a custom memory manager
226 + */
227 + class ObjectAllocator
228 + {
229 +
        public:
230 +
             // Defined by the client (pointer to a block, size of block)
231 +
           typedef void (*DUMPCALLBACK)(const void *, size_t); //!< Callback function when dumping memory leaks
           typedef void (*VALIDATECALLBACK)(const void *, size_t); //!< Callback function when validating blocks
232 +
233 +
234 +
            // Predefined values for memory signatures
           static const unsigned char UNALLOCATED_PATTERN = 0xAA; //!< New memory never given to the client
235 +
           static const unsigned char ALLOCATED_PATTERN = 0xBB; //!< Memory owned by the client</pre>
236 +
237 +
           static const unsigned char FREED_PATTERN =
                                                           0xCC; //!< Memory returned by the client</pre>
           static const unsigned char PAD_PATTERN =
                                                           0xDD; //!< Pad signature to detect buffer over/under flow</pre>
238 +
           static const unsigned char ALIGN_PATTERN =
239 +
                                                           0xEE; //!< For the alignment bytes</pre>
240 +
<u>241</u> +
             // Creates the ObjectManager per the specified values
             // Throws an exception if the construction fails. (Memory allocation problem)
242 +
```

```
243 +
           ObjectAllocator(size_t ObjectSize, const OAConfig& config) noexcept(false);
244 +
245 +
            // Destroys the ObjectManager (never throws)
246 +
          ~ObjectAllocator();
247 +
            // Take an object from the free list and give it to the client (simulates new)
248 +
249 +
            // Throws an exception if the object can't be allocated. (Memory allocation problem)
          void *Allocate(const char *label = 0);
250 +
251 +
252 +
            // Returns an object to the free list for the client (simulates delete)
253 +
            // Throws an exception if the the object can't be freed. (Invalid object)
254 +
          void Free(void *Object);
255 +
256 +
            // Calls the callback fn for each block still in use
257 +
          unsigned DumpMemoryInUse(DUMPCALLBACK fn) const;
258 +
259 +
            // Calls the callback fn for each block that is potentially corrupted
260 +
          unsigned ValidatePages(VALIDATECALLBACK fn) const;
261 +
262 +
            // Frees all empty pages (extra credit)
263 +
          unsigned FreeEmptyPages();
264 +
265 +
            // Returns true if FreeEmptyPages and alignments are implemented
266 +
          static bool ImplementedExtraCredit();
267 +
268 +
            // Testing/Debugging/Statistic methods
          void SetDebugState(bool State); // true=enable, false=disable
269 +
270 +
          const void *GetFreeList() const; // returns a pointer to the internal free list
271 +
           const void *GetPageList() const; // returns a pointer to the internal page list
272 +
          OAConfig GetConfig() const;
                                         // returns the configuration parameters
273 +
          OAStats GetStats() const;
                                           // returns the statistics for the allocator
274 +
275 +
            // Prevent copy construction and assignment
           ObjectAllocator(const ObjectAllocator &oa) = delete; //!< Do not implement!
<u>276</u> +
277 +
           ObjectAllocator &operator=(const ObjectAllocator &oa) = delete; //!< Do not implement!
278 +
        private:
          typedef unsigned char byte; //!< byte typedef for understanding code better
279 +
280 +
281 +
          // Some "suggested" members (only a suggestion!)
282 +
          GenericObject *PageList_ = { 0 }; //!< the beginning of the list of pages</pre>
283 +
          GenericObject *FreeList_ = { 0 }; //!< the beginning of the list of objects</pre>
284 +
          OAConfig Config_;
                                       //!< config for the OA
          OAStats Stats_;
                                           //!< stats gotten for the OA at run time
285 +
           size_t InterBlockSize_;
                                          //!< the inter block size from the OA
286 +
287 +
288 +
          //sets alignment and returns the page size
          size_t Get_PageSize();
289 +
290 +
291 +
          //allocates the memory needed for a page
292 +
          byte *Allocate_Page();
293 +
          //populates the page with debug pad bytes
294 +
          void Populate_Page(byte *page);
295 +
296 +
          //adds a page to the page list
297 +
298 +
          void Add_Page();
299 +
300 +
          //returns the header of an object
           byte *Get_Header(byte *object) const;
301 +
302 +
303 +
           //updates the header of the allocated block of data passed to the client
           void Allocate_Header(byte *header, const char *label);
304 +
305 +
           //updates the header of the freed header
306 +
           void Free_Header(byte *header);
<u>307</u> +
308 +
          //checks if an object has been freed
309 +
310 +
           bool Freed(byte *object) const;
<u>311</u> +
312 +
          //checks if an object has been corrupted
313 +
           bool Corrupted(byte *object) const;
314 +
315 +
          //checks if an object is out of bounds
316 +
           bool OutOfBounds(byte *object) const;
317 +
```

```
318 +
           //check if a page has no objects allocated on it
319 +
           bool PageFree(byte *page) const;
320 +
           //reorder the free list to not include the page passed in
321 +
322 +
           void RemoveFreeListPage(byte *page);
323 +
324 +
           //generic node removal
           void RemoveNode(GenericObject **list,GenericObject *node);
<u>325</u> +
<u>326</u> + };
327 +
328 + #endif
```

Rainier/src/Core/Actions/Action Parser.cpp 0 → 100644

```
1 + #include "stdafx.h"
 2 + #include "Action_Parser.h"
 3 + #include <cstring>
4 + #include <algorithm>
 6 + std::map<std::string, std::map<std::string,std::string>> ActionParser::ActionData_;
7 + std::string ActionParser::currentActionGroup_;
8 +
9 + void ActionParser::Parse(std::string Filename)
10 + {
11 + std::ifstream infile(Filename);
12 + if (infile.is_open())
          PreProccess(infile);
<u>13</u> +
<u>14</u> + }
<u>15</u> +
16 + void ActionParser::PreProccess(std::ifstream &infile)
<u>17</u> + {
18 + std::string line;
19 +
        auto GetNextChar = [](std::string str, size_t &index){
20 +
<u>21</u> +
          while (isspace(str[index])) ++index;
<u>22</u> + };
23 +
        while (!infile.eof())
24 +
<u>25</u> + {
          size_t index = 0;
26 +
<u>27</u> +
          std::getline(infile, line);
28 +
<u>29</u> +
          GetNextChar(line, index);
          if (line[index] == '@')
30 +
<u>31</u> +
            //clear the old group name and set the new one
32 +
<u>33</u> +
            currentActionGroup_ = std::string(line.begin() + ++index, line.end());
            std::transform(currentActionGroup_.begin(), currentActionGroup_.end(),
<u>34</u> +
<u>35</u> +
                            currentActionGroup_.begin(), ::toupper);
<u>36</u> +
          }
          else if (line[index] == '$')
<u>37</u> +
38 +
<u>39</u> +
            GetNextChar(line, index); //get the first char of the name
            int tempIndex = ++index;
40 +
41 +
            //get the end of the word
42 +
            while (!isspace(line[index]) && line[index] != '=') ++index;
43 +
44 +
             std::string name(line.begin() + tempIndex, line.begin() + index); //get the action name
             std::transform(name.begin(), name.end(), name.begin(), ::toupper);
46 +
<u>47</u> +
            while (line[index] != '=') ++index;
48 +
            tempIndex = ++index;
49 +
<u>50</u> +
            while (line[index] != ';') ++index;
<u>51</u> +
            std::string expression(line.begin() + tempIndex, line.begin() + index);
<u>52</u> +
            ActionData_[currentActionGroup_][name] = expression;
<u>53</u> +
<u>54</u> +
          }
<u>55</u> + }
<u>56</u> +
                                        // clear fail and eof bits
57 + infile.clear();
58 + infile.seekg(0, std::ios::beg); // back to the start
<u>59</u> + }
```

Rainier/src/Core/Actions/Action Parser.h 0 → 100644

```
<u>1</u> + #define ACTION_PARSER_H
 2 + #ifdef ACTION_PARSER_H
 3 + #include <fstream>
4 + #include <string>
5 + #include <map>
7 + class ActionParser
<u>8</u> + {
9 + public:
          static std::map<std::string, std::string, std::string>> ActionData_; //action group <action name,</pre>
10 +
     expression>
11 + static void Parse(std::string Filename = "0");
          static std::string GetExpression(std::string actionGroup, std::string actionName); //for the action tree
12 +
          static std::string GetCurrentActionGroup(); //for the action tree
<u>13</u> +
14 + private:
          static void PreProccess(std::ifstream &infile); //maps actions to expressions
<u>15</u> +
          static std::string currentActionGroup_;
<u>16</u> +
<u>17</u> + };
<u>18</u> +
<u>19</u> + #endif
      \ No newline at end of file
```

▼ Rainier/src/Core/Actions/Action Tree.cpp 0 → 100644

```
1 + #include "stdafx.h"
2 + #include "Action_Parser.h"
 3 + #include "Actions.h"
4 +
5 + void ActionTree::Generate(std::string expression)
<u>6</u> + {
7 + FreeTree(root_);
8 + root_ = nullptr;
9 + expression_ = expression;
10 + GetToken();
11 + CheckOr(root_);
12 + }
13 +
+ ActionTree::~ActionTree()
<u>15</u> + {
16 + FreeTree(root_);
<u>17</u> + }
<u>18</u> +
19 + ActionTree::TreeNode::TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool()> inputCallBack):
20 + left_(left), right_(right), type_(type), input_(inputCallBack) {}
21 +
22  + ActionTree::TreeNode::TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool(bool,bool)>
      opCallBack):
4 left_(left), right_(right), type_(type), operator_(opCallBack) {}
25 + bool ActionTree::EvaluateTree() const
27 + return Evaluate(root_);
<u>28</u> + }
29 +
30 + bool ActionTree::Evaluate(TreeNode *tree) const
<u>31</u> + {
32 + if (tree->type_ == TreeNode::ANDOP || tree->type_ == TreeNode::OROP)
          return tree->operator_(Evaluate(tree->left_), Evaluate(tree->right_));
33 +
34 + return tree->input_();
<u>35</u> + }
36 +
37 + //evaluate the boolean expression
38 + //bool ActionTree::Evaluate()
39 + //{
40 + // return false;
41 + //}
42 +
43 + void ActionTree::GetToken()
45 + currToken_.clear(); //clear the token
46 +
47 + //go to the next non-whitespace char
48 +
        while (isspace(expression_[currPos_])) ++currPos_;
49 +
50 + currToken_.push_back(expression_[currPos_++]);
```

```
<u>51</u> +
 <u>52</u> + //check if we got an operator
        if (IsOperator(currToken_))
 54 +
           return;
 <u>55</u> +
 56 + //get the rest of the input hash and return
         while (IsIdentifier(expression_[currPos_]))
 <u>58</u> +
           currToken_.push_back(expression_[currPos_++]);
 <u>59</u> +
 60 + //uppercase everything
 61 + std::transform(currToken_.begin(), currToken_.end(), currToken_.begin(), ::toupper);
 <u>62</u> + }
 63 +
 64 + void ActionTree::CheckOr(TreeNode *&tree) // |
 65 + {
 66 + CheckAnd(tree);
 67 +
 <u>68</u> +
         while (currToken_ == "|")
 <u>69</u> + {
 <u>70</u> +
           auto temp = new TreeNode(nullptr, nullptr, TreeNode::OROP, [](bool L,bool R) { return L | R; });
 <u>71</u> +
           temp->left_ = tree;
 <u>72</u> +
           tree = temp;
 <u>73</u> +
           GetToken();
 <u>74</u> +
           CheckAnd(tree->right_);
 <u>75</u> + }
<u>76</u> + }
 <u>77</u> +
 78 + void ActionTree::CheckAnd(TreeNode *&tree) // &
 79 + {
 80 + CheckNot(tree);
         while (currToken_ == "&")
 <u>81</u> +
 82 + {
 83 +
           auto temp = new TreeNode(nullptr, nullptr, TreeNode::ANDOP, [](bool L,bool R) { return L & R; });
 84 +
           temp->left_ = tree;
 85 +
           tree = temp;
 86 +
           GetToken();
 <u>87</u> +
           CheckNot(tree->right_);
 88 + }
 89 + }
 90 +
 91 + void ActionTree::CheckNot(TreeNode *&tree) // !
92 + {
 93
    + //go to the end of all the not(s) and set the value there
 94 + if (currToken_ == "!")
 95 + {
 96 +
           GetToken();
 97 +
           CheckRest(tree);
           std::function<bool()> temp = tree->input_;
 98 +
           tree->input_ = [temp](){return !temp();};
 99 +
<u>100</u> + }
<u>101</u> + else
102 +
           CheckRest(tree);
<u>103</u> + }
104 +
105 + void ActionTree::CheckRest(TreeNode *&tree) // ( or input
106 + {
<u>107</u> + if (currToken_ == "(")
108 + {
109 +
           GetToken();
110 +
           CheckOr(tree);
<u>111</u> + }
112 +
113 +
         //check if an action is within the preprocessed map
         auto NestedActionCheck = ActionParser::ActionData_[ActionManager::CurrectActionGroup()].find(currToken_);
114 +
<u>115</u> +
         if (NestedActionCheck != ActionParser::ActionData_[ActionManager::CurrectActionGroup()].end())
<u>116</u> + {
<u>117</u> +
           //maybe move this within the lambda as a static member
           //play around with this. Pointers might be an issue here. Or its very expensive
<u>118</u> +
           tree = new TreeNode(nullptr, nullptr, TreeNode::INPUT, [=]() {
119 +
120 +
             ActionTree nested_tree;
121 +
             nested_tree.Generate(NestedActionCheck->second);
             return nested_tree.EvaluateTree();});
122 +
123 + }
124 + //TODO: put callback to input system here (check if key is within the action system)
125 + else if(IsInput(currToken_))
```

```
126 +
          tree = new TreeNode(nullptr, nullptr, TreeNode::INPUT, []() {return true; });
127 +
<u>128</u> + GetToken();
129 + }
130 +
131 + bool ActionTree::IsOperator(std::string token)
132 + {
133 + return (token == "|" || token == "&" || token == "!" || token == "(");
134 + }
135 +
136 + bool ActionTree::IsIdentifier(char token)
137 + {
138 + return (token >= 'a' && token <= 'z') || (token >= 'A' && token <= 'Z') || token == '_';
<u>139</u> + }
140 +
141 + bool ActionTree::IsInput(std::string token)
142 + {
143 + //index into the map that has the Look up keys
144 + return token != ")" && !IsOperator(token);
<u>145</u> + }
146 +
147 + void ActionTree::FreeTree(TreeNode *tree)
148 + {
<u>149</u> + if (!tree)
150 +
           return;
<u>151</u> +
152 + FreeTree(tree->left_);
153 + FreeTree(tree->right_);
154 + FreeNode(tree);
<u>155</u> + }
156 +
157 + void ActionTree::FreeNode(TreeNode *node)
158 + {
159 + delete node;
160 + }
161 +
162 + //std::function<bool()> ActionTree::GetInputCallBack(std::string input)
163 + //{
<u>164</u> + // if (input[0] == 'k' || input[0] == 'K')
165 + // {
166 + //
           //check if input is within the input system
167 + // //return a callback function to the keyboard input system
168 + // }
<u>169</u> + // if (input[0] == 'c' || input[0] == 'C')
170 + // {
171 + // //check if input is within the input system
172 + // //return a callback to the controller input system
<u>173</u> + // }
174 + //
175 + // //return a temp dummy function
<u>176</u> + // return [](){ return true; };
<u>177</u> + //}
```

Rainier/src/Core/Actions/Action Tree.h 0 → 100644

\ No newline at end of file

```
1 + #define ACTION_TREE_H
 2 + #ifdef ACTION_TREE_H
 3 + #include <string>
4 + #include <map>
5 + #include <functional>
6 +
7 + class ActionTree
8 + {
9 + struct TreeNode;
<u>10</u> + public:
11 + void Generate(std::string expression);
12 + ~ActionTree();
13 + bool EvaluateTree() const;
<u>14</u> + private:
15 + bool Evaluate(TreeNode *tree) const;
16 + TreeNode *root_ = nullptr;
17 + unsigned currPos_ = 0;
18 + std::string currToken_;
19 + std::string expression_;
```

```
20 +
<u>21</u> +
        struct TreeNode
22 + {
23 +
          enum NodeType {ANDOP, OROP, INPUT};
24 +
          TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool()> inputCallBack);
          TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool(bool,bool)> opCallBack);
25 +
26 +
          TreeNode *left_; // left sub-expression
<u>27</u> +
          TreeNode *right_; // right sub-expression
          NodeType type_; // kind of node this is
28 +
29 +
          std::function<bool()> input_;
<u>30</u> +
          std::function<bool(bool,bool)> operator_;
<u>31</u> + };
<u>32</u> +
33 +
        void GetToken();
<u>34</u> +
35 +
        void CheckOr(TreeNode *&tree); // |
<u>36</u> +
        void CheckAnd(TreeNode *&tree); // &
37 +
        void CheckNot(TreeNode *&tree); // !
38 +
        void CheckRest(TreeNode *&tree); // ( or input
39 +
40 +
       void FreeTree(TreeNode *tree);
41 +
        void FreeNode(TreeNode *node);
<u>42</u> +
43 + bool IsOperator(std::string token);
44 + bool IsIdentifier(char token);
45 + bool IsInput(std::string token);
<u>46</u> + };
<u>47</u> +
<u>48</u> + #endif
      \ No newline at end of file
```

Rainier/src/Core/Actions/Actions.cpp 0 → 100644

```
1 + #include "stdafx.h"
 2 + #include "Actions.h"
 3 + #include "Action_Parser.h"
4 + #include <fstream>
 <u>5</u> +
 6 + std::map<std::string, std::map<std::string, ActionTree>> ActionManager::actionManager_;
7 + std::string ActionManager::currentActionGroup_;
8 +
9 + void ActionManager::Import(std::string Filename)
10 + {
11 + ActionParser::Parse(Filename);
        auto groupItter = ActionParser::ActionData_.begin();
        while (groupItter != ActionParser::ActionData_.end())
<u>13</u> +
<u>14</u> + {
        //loop through each action within a group generating their ActionTree's
<u>15</u> +
          auto actionItter = groupItter->second.begin();
16 +
          while (actionItter != groupItter->second.end())
<u>17</u> +
<u>18</u> +
            actionManager_[groupItter->first][actionItter->first].Generate(actionItter->second);
19 +
            ++actionItter;
20 +
21 +
          }
22 +
23 +
          ++groupItter;
<u>24</u> + }
<u>25</u> + }
27 + void ActionManager::Reimport(std::string Filename)
28 + {
29 + actionManager_.clear();
30 + Import(Filename);
<u>31</u> + }
<u>32</u> +
33 + bool ActionManager::Action_Pressed(std::string action)
<u>34</u> + {
35 + return false;
<u>36</u> + }
<u>37</u> +
38 + bool ActionManager::Action_Held(std::string action)
<u>39</u> + {
40 + std::transform(action.begin(), action.end(), action.begin(), ::toupper);
41 + return actionManager_[currentActionGroup_][action].EvaluateTree();
<u>42</u> + }
```

```
43 +
44 + bool ActionManager::Action_Released(std::string action)
<u>45</u> + {
46 + return false;
<u>47</u> + }
48 +
49 + void ActionManager::LoadActionGroup(std::string actionGroup)
<u>50</u> + {
51 + std::transform(actionGroup.begin(), actionGroup.end(), actionGroup.begin(), ::toupper);
52 + currentActionGroup_ = actionGroup;
<u>53</u> + }
<u>54</u> +
55 + std::string ActionManager::CurrectActionGroup()
<u>56</u> + {
57 + return currentActionGroup_;
<u>58</u> + }
     \ No newline at end of file
```

Rainier/src/Core/Actions/Actions.h 0 → 100644

```
1 + #define ACTIONS_H
 2 + #ifdef ACTIONS_H
3 + #include "Action_Tree.h"
<u>4</u> + #include <string>
<u>5</u> + #include <map>
<u>6</u> +
7 + class ActionManager
8 + {
9 + public:
           static void Import(std::string Filename);
<u>10</u> +
<u>11</u> +
           static void Reimport(std::string Filename);
<u>12</u> +
<u>13</u> +
           static bool Action_Pressed(std::string action);
<u>14</u> +
           static bool Action_Held(std::string action);
<u>15</u> +
           static bool Action_Released(std::string action);
<u>16</u> +
<u>17</u> +
           static void LoadActionGroup(std::string actionGroup);
<u>18</u> +
           static std::string CurrectActionGroup();
<u>19</u> + private:
20 +
           static std::map<std::string, std::map<std::string, ActionTree>> actionManager_;
<u>21</u> +
           static std::string currentActionGroup_;
<u>22</u> + };
23 +
<u>24</u> + #endif
      \ No newline at end of file
```

Please <u>register</u> or <u>sign in</u> to comment