Commit 76dd2c59 authored 1 year ago by Alien G

Merge remote-tracking branch 'origin/master'

parents 358e73fc a7e535ab master ···

No related merge requests found

```
+ #include "stdafx.h"
```

Showing 8 changed files with **1570 additions** and **0 deletions**

```
+ \author Tristan Bouchard
+ \par    email: tristan.bouchard\@digipen.edu
+ \par    DigiPen login: tristan.bouchard
+ \par    Course: CS200
+ \par    Section: A
+ \par    Assignment #N/A
+ \date   9/19/2019
+ \brief
+     This file constains the file the implementation of the ObjectAllocator
+ */
+ /******************************************************************************/
+ #include "Action_Memory.h"
+ #include <cstring>
+
+ /*!
+  * \brief
+  *   Creates the ObjectManager per the specified values. Throws an exception if the construction
+  *   fails. (Memory allocation problem)
+  *
+  * \param ObjectSize
+  *   The size of the object to keep track of the memory of
+  *
+  * \param config
+  *   Settings for the allocator, see ObjectAllocator.h for options
+  */
+ ObjectAllocator::ObjectAllocator(size_t ObjectSize, const OAConfig &config) :
+         Config_(config), InterBlockSize_(0) {
+   //set the object size
+   Stats_.ObjectSize_ = ObjectSize;
+
+   //set the page size
+   Stats_.PageSize_ = Get_PageSize();
+
+   //if the memory allocated
+   if (!Config_.UseCPPMemManager_)
+   {
+     InterBlockSize_ = Config_.HBlockInfo_.size_ + Stats_.ObjectSize_;
+     InterBlockSize_ += Config_.InterAlignSize_ + Config_.PadBytes_ * 2;
+
+     //allocate a page and add it to the allocator
+     Add_Page();
+   }
+ }
+
+ /*!
+  * \brief
+  *   Destroys the ObjectManager (never throws)
+  */
+ ObjectAllocator::~ObjectAllocator() {
+   //is the memory manager is in use
+   if (!Config_.UseCPPMemManager_)
+   {
```

```
+     //loop through the page list deleting all allocated memory for the pages
+     while (PageList_)
+     {
+       byte *temp = reinterpret_cast<byte *>(PageList_);
+       PageList_ = PageList_->Next;
+       delete[] temp;
+     }
+   }
+ }
+
+ /*!
+  * \brief
+  *   Takes an object from the free list and returns it as allocated memory.
+  *   Will through if there is a memory issue
+  *
+  * \param label
+  *   A unique name given to the objects header
+  *
+  * \return
+  *   returns a pointer to the allocated memory
+  */
+
+ void *ObjectAllocator::Allocate(const char *label) {
+   void *rtn = nullptr; //pointer to return to the client
+
+   //if the memory manager is disabled, return an allocated void pointer
+   if (Config_.UseCPPMemManager_)
+   {
+     try
+     {
+       //allocate memory and keep track of allocations
+       ++Stats_.Allocations_;
+       rtn = new byte[Stats_.ObjectSize_]();
+     }
+     catch (std::bad_alloc &)
+     {
+       --Stats_.Allocations_;
+       throw OAException(OAException::E_NO_MEMORY, "Object can't be allocated");
+     }
+   }
+   else
+   {
+     //if a new page needs to be allocated
+     if (Stats_.ObjectsInUse_ == Stats_.PagesInUse_ * Config_.ObjectsPerPage_)
+       Add_Page();
+
+     //remove the pointer from the free list
+     *reinterpret_cast<GenericObject **>(&rtn) = FreeList_;
+     FreeList_ = FreeList_->Next;
+
+     //update header data
+     Allocate_Header(Get_Header(reinterpret_cast<byte *>(rtn)), label);
+
+     //if debug is on, add the bytes
+     if (Config_.DebugOn_)
+       memset(reinterpret_cast<byte *>(rtn), ALLOCATED_PATTERN, Stats_.ObjectSize_);
+   }
+
+   //update objects in use and free objects
+   ++Stats_.ObjectsInUse_;
+   --Stats_.FreeObjects_;
+
+   //check if its the most objects allocated at one time
+   if (Stats_.ObjectsInUse_ > Stats_.MostObjects_)
+     Stats_.MostObjects_ = Stats_.ObjectsInUse_;
+
+   //return the pointer
+   return rtn;
+ }
+
+ /*!
+  * \brief
+  *   Frees memory allocated by the object allocator.Throws an exception if the the
+  *   object can't be freed. (Invalid object)
+  *
+  * \param Object
+  *   A unique name given to the objects header
+  *
+  * \return
+  *   returns a pointer to the allocated memory
+  */
+ void ObjectAllocator::Free(void *Object) {
+   //if the memory manager
+   if (Config_.UseCPPMemManager_)
+   {
+     //Track deallocations and free the memory
+     ++Stats_.Deallocations_;
+     delete[] reinterpret_cast<byte *>(Object);
+     return;
+   }
+
+   //if the debug option is on, check for bounds, double freed and corruptions
+   if (Config_.DebugOn_)
+   {
+     //check if a block has already been freed throw an exception if so
+     if (Freed(reinterpret_cast<byte *>(Object)))
+       throw OAException(OAException::E_MULTIPLE_FREE,
+                         "Object has already been freed or freed out of bounds");
+
+     //check if the object is out of bounds
+     if (OutOfBounds(reinterpret_cast<byte *>(Object)))
+       throw OAException(OAException::E_BAD_BOUNDARY, "Object out of range of its allocated memory");
+
+     //check if the pad bytes around the object is corrupted/overwritten. if so, throw an exception
+     if (Corrupted(reinterpret_cast<byte *>(Object)))
+       throw OAException(OAException::E_CORRUPTED_BLOCK,
+                         "Pad bytes adjacent to object ptr has been overwritten");
+
+     //if debug is on, add the bytes
+     memset(reinterpret_cast<byte *>(Object), FREED_PATTERN, Stats_.ObjectSize_);
+   }
+
+   //add the object back to the free list
+   GenericObject *obj = reinterpret_cast<GenericObject *>(Object);
+   obj->Next = FreeList_;
+   FreeList_ = obj;
+
+   //free/update the header block attached to the data block
+   Free_Header(Get_Header(reinterpret_cast<byte *>(Object)));
+
+   //update states for freeing
+   --Stats_.ObjectsInUse_;
+   ++Stats_.FreeObjects_;
+   ++Stats_.Deallocations_;
+ }
+
+ /*!
+  * \brief
+  *   Calls the function passed in on every object that is currently allocated within the page list
+  *
+  * \param fn
+  *   function to run on all allocated blocks of data
+  *
+  * \return
+  *   returns the number of blocks of data allocated within the allocator
+  */
+ unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const {
+   //if there are no pad bytes or debug is off, return zero
+   if (!Config_.DebugOn_ || Config_.PadBytes_)
+     return 0;
+
+   //pointer to the page list
+   GenericObject *run = PageList_;
+
+   //number of objects allocated
+   unsigned objs_allocated = 0;
+
+   //loop through all the pages
+   while (run)
+   {
```

```
+    //get the start of a page
+    byte *page = reinterpret_cast<byte *>(run);
+
+    //move to the first data block of data
+    page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+    //loop through the page checking if each block is in use
+    for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)
+    {
+      //check if the memory is in use
+      if (!Freed(page))
+      {
+        //print out the memory and add one to the allocated number
+        fn(page, Stats_.ObjectSize_);
+        ++objs_allocated;
+      }
+    }
+
+    //move to the next page
+    run = run->Next;
+  }
+
+  //return the number of objects in use by the memory
+  return objs_allocated;
+ }
+
+ /*!
+  * \brief
+  *   Calls the callback fn for each block that is potentially corrupted
+  *
+  * \param fn
+  *   callback function for using on every block of memory that might be corrupted
+  *
+  * \return
+  *   the number of objects corrupted
+  */
+ unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const {
+  //if there are no pad bytes or debug is off, return zero
+  if (!Config_.DebugOn_ || !Config_.PadBytes_)
+    return 0;
+
+  //pointer to the page list
+  GenericObject *run = PageList_;
+
+  //number of objects corrupted
+  unsigned objs_corrupted = 0;
+
+  //loop through all the pages
+  while (run)
+  {
+    //get the start of a page
+    byte *page = reinterpret_cast<byte *>(run);
+
+    //move to the first data block of data
+    page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+    //loop through the page checking if each block is corrupted
+    for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)
+    {
+      //check if the pad bytes around the block corrupted
+      if (Corrupted(page))
+      {
+        //print out the memory and add one to the corrupted count
+        fn(page, Stats_.ObjectSize_);
+        ++objs_corrupted;
+      }
+    }
+
+    //move to the next page
+    run = run->Next;
+  }
+
+  //return the number of objects corrupted
+  return objs_corrupted;
+ }
```

```
+
+ /*!
+  * \brief
+  *   Frees all empty pages
+  *
+  * \return
+  *   returns the number of pages freed
+  */
+ unsigned ObjectAllocator::FreeEmptyPages() {
+   GenericObject *run = PageList_; //pointer to the page list
+
+   //number of objects corrupted
+   unsigned pages_freed = 0;
+
+   //loop through all the pages
+   while (run)
+   {
+     //get the start of a page
+     byte *page = reinterpret_cast<byte *>(run);
+     if (PageFree(page))
+     {
+       RemoveFreeListPage(page);    //reorder the free list to not include the page about to free
+       RemoveNode(&PageList_, run); //remove the page from the page list
+
+       byte *temp = page;
+       run = run->Next;
+       delete[] temp;
+
+       //increase freed pages
+       --Stats_.PagesInUse_;
+       Stats_.FreeObjects_ -= Config_.ObjectsPerPage_;
+       ++pages_freed;
+     }
+     else
+     {
+       //move to the next page
+       run = run->Next;
+     }
+   }
+
+   //return the number of objects corrupted
+   return pages_freed;
+ }
+
+ // Returns true if FreeEmptyPages and alignments are implemented
+ /*!
+  * \brief
+  *   checks if the extra credit is finished
+  *
+  * \return
+  *   returns true if the extra credit is implemented
+  */
+ bool ObjectAllocator::ImplementedExtraCredit() {
+   //return true ;)
+   return true;
+ }
+
+ // Testing/Debugging/Statistic methods
+ /*!
+  * \brief
+  *   sets the debug state of the allocator
+  *
+  * \param State
+  *   true for debug on, false for debug false
+  */
+ void ObjectAllocator::SetDebugState(bool State) {
+   Config_.DebugOn_ = State;
+ }
+
+ /*!
+  * \brief
+  *   Gets the free list from the object allocator. Cast to GenericObject* to access the linked list
+  *
+  * \return
+  *   returns the start of the free list.
+  */
+ const void *ObjectAllocator::GetFreeList() const // returns a pointer to the internal free list
+ {
+   return FreeList_;
+ }
+
+ /*!
+  * \brief
+  *   Gets the page list from the object allocator. Cast to GenericObject* to access the linked list
+  *
+  * \return
+  *   returns the head of the page list
+  */
+ const void *ObjectAllocator::GetPageList() const // returns a pointer to the internal page list
+ {
+   return PageList_;
+ }
+
+ /*!
+  * \brief
+  *   Gets a copy of the config used for the allocator
+  *
+  * \return
+  *   returns a copy of the config object passed in when creating the allocator
+  */
+ OAConfig ObjectAllocator::GetConfig() const // returns the configuration parameters
+ {
+   return Config_;
+ }
+
+ /*!
+  * \brief
+  *   Gets a copy of the stats used for the allocator
+  *
+  * \return
+  *   returns the updated stats of the allocator
+  */
+ OAStats ObjectAllocator::GetStats() const // returns the statistics for the allocator
+ {
+   return Stats_;
+ }
+
+ /*!
+  * \brief
+  *   Helper function for updating the header block of an object during allocations
+  *
+  * \param header
+  *   the first byte of the header block
+  *
+  * \param label
+  *   the label to attach to the external header block
+  */
+ void ObjectAllocator::Allocate_Header(byte *header, const char *label) {
+   //update allocations_
+   ++Stats_.Allocations_;
+
+   //if basic, alloc # + flag byte
+   if (Config_.HBlockInfo_.type_ == OAConfig::hbBasic)
+   {
+     //add one to the allocation count
+     *reinterpret_cast<unsigned *>(header) = Stats_.Allocations_;
+
+     //set the bool flag to true
+     *reinterpret_cast<bool *>(header + sizeof(unsigned)) = true;
+   }
+
+     //if extended, user-defined + use counter + alloc # + flag byte
+   else
+     if (Config_.HBlockInfo_.type_ == OAConfig::hbExtended)
+     {
+       //add one to the re-use count
+       ++*reinterpret_cast<short *>(header + Config_.HBlockInfo_.additional_);
+
+       //add one to the allocation count
+       *reinterpret_cast<unsigned *>(header + Config_.HBlockInfo_.additional_ +
+                                     sizeof(short)) = Stats_.Allocations_;
+
+       //set the bool flag to true
+       *reinterpret_cast<bool *>(header + Config_.HBlockInfo_.additional_ + sizeof(short) +
+                                 sizeof(unsigned)) = true;
+     }
+
+       //if external, allocate obj
+     else
+       if (Config_.HBlockInfo_.type_ == OAConfig::hbExternal)
+       {
+         //get a pointer size of memory from the page
+         MemBlockInfo **header_block = reinterpret_cast<MemBlockInfo **>(header);
+         try
+         {
+           //allocate the pointer
+           (*header_block) = new MemBlockInfo;
+         }
+         catch (std::bad_alloc &)
+         {
+           //throw if no memory
+           throw OAException(OAException::E_NO_MEMORY, "Unable to allocate external header");
+         }
+
+         //if there is a label to apply
+         if (label)
+         {
+           try
+           {
+             //allocate the char array
+             (*header_block)->label = new char[strlen(label) + 1]();
+           }
+           catch (std::bad_alloc &)
+           {
+             //throw of there is no memory
+             throw OAException(OAException::E_NO_MEMORY, "Unable to allocate external header label");
+           }
+           //copy the label into the headers label
+           strcpy((*header_block)->label, label);
+         }
+
+         //set the in use to in use
+         (*header_block)->in_use = true;
+
+         //set the allocation number
+         (*header_block)->alloc_num = Stats_.Allocations_;
+       }
+ }
+
+ /*!
+  * \brief
+  *   Helper function for updating the header block of an object during freeing
+  *
+  * \param header
+  *   the first byte of the header block
+  */
+ void ObjectAllocator::Free_Header(byte *header) {
+   //if external, allocate obj
+   if (Config_.HBlockInfo_.type_ == OAConfig::hbExternal)
+   {
+     //get pointer to memory
+     MemBlockInfo *head_ptr = *reinterpret_cast<MemBlockInfo **>(header);
+
+     //delete the label attached to the header
+     delete[] head_ptr->label;
+
+     head_ptr->label = nullptr;
+
+     //delete the header
+     delete head_ptr;
+   }
+
+   //set the extended header to zeros keeping the fist two bytes with the object
+   if (Config_.HBlockInfo_.type_ == OAConfig::hbExtended)
+   {
```

```
+     memset(header, 0, Config_.HBlockInfo_.additional_);
+     memset(header + Config_.HBlockInfo_.additional_ + sizeof(short), 0, Config_.HBlockInfo_.size_ - sizeof(short) - Config_.HBlockInfo_.additional_);
+   }
+     //set the memory to zero if a block is present
+   else if (Config_.HBlockInfo_.type_ != OAConfig::hbNone)
+   {
+     memset(header, 0, Config_.HBlockInfo_.size_);
+   }
+ }
+
+ /*!
+  * \brief
+  *   given a header block, check if the data has been freed
+  *
+  * \param object
+  *   the first byte of the object data
+  *
+  * \return
+  *   return if the object passed in has been freed
+  */
+ bool ObjectAllocator::Freed(byte *object) const {
+   if (Config_.HBlockInfo_.type_ != OAConfig::hbNone)
+   {
+     //check to see if the block is extended
+     size_t offset = (Config_.HBlockInfo_.type_ == OAConfig::hbExtended ? Config_.HBlockInfo_.additional_ + sizeof(short): 0);
+
+     //compare the header with an array of 0s
+     byte *arr = new byte[Config_.HBlockInfo_.size_]();
+     bool result = !memcmp(Get_Header(object) + offset, arr, Config_.HBlockInfo_.size_ - offset);
+     delete[] arr;
+     return result;
+   }
+
+   //check if the object passed in is on the free list
+   GenericObject *run = FreeList_;
+   while (run)
+   {
+     //pointer to first element of a page
+     if (object == reinterpret_cast<byte *>(run))
+       return true;
+
+     //move to the next page
+     run = run->Next;
+   }
+
+   return false;
+ }
+
+ /*!
+  * \brief
+  *   Calculates and returns page size
+  * \return
+  *   returns the size of the page in bytes
+  */
+ size_t ObjectAllocator::Get_PageSize() {
+   //check if alignment needs to be calculated
+   if (Config_.Alignment_ > 1)
+   {
+     //calculate the bytes needed for the left alignment
+     size_t front_bytes = sizeof(void *) + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+     Config_.LeftAlignSize_ =
+             Config_.Alignment_ - (static_cast<unsigned>(front_bytes) % Config_.Alignment_);
+
+     //calculate the bytes needed for the inter block alignment
+     size_t inter_bytes = Stats_.ObjectSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_ * 2;
+     Config_.InterAlignSize_ =
+             Config_.Alignment_ - (static_cast<unsigned>(inter_bytes) % Config_.Alignment_);
+   }
+
+   //size of the page
+   size_t size = 0;
+
+   // next page pointer + left alignment block (only the first block has left alignment)
+   size = sizeof(void *) + Config_.LeftAlignSize_;
+
+   // Each object has a header, left padding, the object itself, right padding
+   //   (The left and right padding blocks are the same size)
+   size += Config_.ObjectsPerPage_ *
+           (Config_.HBlockInfo_.size_ + Config_.PadBytes_ + Stats_.ObjectSize_ + Config_.PadBytes_);
+
+   // All but the first block have an inter-block alignment block
+   size += (Config_.ObjectsPerPage_ - 1) * Config_.InterAlignSize_;
+
+   return size;
+ }
+
+ /*!
+  * \brief
+  *   allocates the memory size of one page. will throw if page cant be allocated
+  * \return
+  *   returns allocated pointer
+  */
+ ObjectAllocator::byte *ObjectAllocator::Allocate_Page() {
+   try
+   {
+     //return allocated memory filled with UNALLOCATED_PATTERN value
+     return reinterpret_cast<byte *>(new byte[Stats_.PageSize_]());
+   }
+   catch (std::bad_alloc &)
+   {
+     throw OAException(OAException::E_NO_MEMORY, "Unable to allocate page");
+   }
+ }
+
+ /*!
+  * \brief
+  *   Populates a page with all debug pattern bytes
+  * \param page
+  *   the first byte of a page
+  */
+ void ObjectAllocator::Populate_Page(byte *page) {
+   //set Null values for Page list pointer
+   memset(page, 0, sizeof(void *));
+
+   //set the alignment bytes for the left block
+   memset(page + sizeof(void *), ALIGN_PATTERN, Config_.LeftAlignSize_);
+
+   //sets header, padding, datam padding
+   auto Set_Block_Bytes = [&](byte *head_pos) {
+       //header
+       memset(head_pos, 0, Config_.HBlockInfo_.size_);
+
+       //padding
+       memset(head_pos + Config_.HBlockInfo_.size_, PAD_PATTERN, Config_.PadBytes_);
+
+       //padding
+       memset(head_pos + Config_.HBlockInfo_.size_ + Config_.PadBytes_ + Stats_.ObjectSize_,
+              PAD_PATTERN, Config_.PadBytes_);
+       return void(0);
+   };
+
+   //The total size of the inter block bytes minus the alignment
+   size_t InterBlockSizeIsh = InterBlockSize_ - Config_.InterAlignSize_;
+
+   //Run through the list setting the bytes of each inter block
+   for (size_t i = 0, offset = sizeof(void *) + Config_.LeftAlignSize_;
+        i < Config_.ObjectsPerPage_; ++i, offset += InterBlockSizeIsh)
+   {
+     //alignment when i != 0
+     if (i)
+     {
+       memset(page + offset, ALIGN_PATTERN, Config_.InterAlignSize_);
+       offset += Config_.InterAlignSize_;
+     }
+
+     //offsets in relation to the start of the header
+     Set_Block_Bytes(page + offset);
+   }
+ }
+
+ /*!
+  * \brief
+  *   allocates and adds a page to the front of the page list
+  */
+ void ObjectAllocator::Add_Page() {
+   //checks if page count max has been reached
+   if (Config_.MaxPages_ && (Stats_.PagesInUse_ == Config_.MaxPages_))
+     throw OAException(OAException::E_NO_PAGES, "Max pages have been reached");
+
+   //allocate a new block of memory
+   byte *new_page = Allocate_Page();
+
+   //populate the page for debugging
+   if (Config_.DebugOn_)
+   {
+     //mem set the whole page with unallocated pattern
+     memset(new_page, UNALLOCATED_PATTERN, Stats_.PageSize_);
+
+     //populate the page with all the proper patterns
+     Populate_Page(new_page);
+   }
+
+   //update pages in use and Free objects
+   ++Stats_.PagesInUse_;
+   Stats_.FreeObjects_ += Config_.ObjectsPerPage_;
+
+   //insert the new page into the from of the page list
+   GenericObject *front_bytes = reinterpret_cast<GenericObject *>(new_page);
+   front_bytes->Next = PageList_;
+   PageList_ = front_bytes;
+
+   //move to where the block of data would be
+   new_page +=
+           sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+   //setup pointers for freed list
+   for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, new_page += InterBlockSize_)
+   {
+     reinterpret_cast<GenericObject *>((new_page))->Next = FreeList_;
+     FreeList_ = reinterpret_cast<GenericObject *>(new_page);
+   }
+ }
+
+ /*!
+  * \brief
+  *   given a block of data, gets the header attached to that block
+  *
+  * \param object
+  *   the first byte of the object that you want to get the header of
+  * \return
+  */
+ ObjectAllocator::byte *ObjectAllocator::Get_Header(byte *object) const {
+   return object - Config_.PadBytes_ - Config_.HBlockInfo_.size_;
+ }
+
+ /*!
+  * \brief
+  *   checks if a block of data is corrupted or not
+  *
+  * \param object
+  *   the first byte of the object you want to check for corruption for
+  *
+  * \return
+  *   returns true if the block is corrupted
+  */
+ bool ObjectAllocator::Corrupted(byte *object) const {
+   try
+   {
+     //allocate a string to compare with the pad bytes on the left and right
+     const byte *comp_str = reinterpret_cast<byte *>(memset(new byte[Config_.PadBytes_ + 1](),
+                                                             PAD_PATTERN, Config_.PadBytes_));
+     if (memcmp(comp_str, object + Stats_.ObjectSize_, Config_.PadBytes_) ||
+         memcmp(comp_str, object - Config_.PadBytes_, Config_.PadBytes_))
+     {
+       //return that the block has been corrupted
+       delete[] comp_str;
+       return true;
+     }
+     delete[] comp_str;
+   }
+     //throw if the comp string couldn't be allocated
+   catch (std::bad_alloc &)
+   {
+     throw OAException(OAException::E_NO_MEMORY,
+                       "String for comparing pad bytes unable to be allocated");
+   }
+
+   //the memory is not corrupted
+   return false;
+ }
+
+ /*!
+  * \brief
+  *   Checks if an object is out of bounds
+  *
+  * \param object
+  *   the object to check
+  *
+  * \return
+  *   returns true if the object passed in is out of bounds
+  */
+ bool ObjectAllocator::OutOfBounds(byte *object) const {
+   //pointer to the head of the page list
+   GenericObject *run = PageList_;
+   size_t Block_Alignment =
+           sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+   //go though all the pages
+   while (run)
+   {
+     //pointer to first element of a page
+     byte *page = reinterpret_cast<byte *>(run);
+     long long int difference = object - page;
+
+     //check if the object is within the page
+     if (difference >= 0 && difference < static_cast<long long int>(Stats_.PageSize_))
+     {
+       //check if the position of the object is a multiple of all the other objects
+       return (difference % (InterBlockSize_) != Block_Alignment);
+     }
+
+     //move to the next page
+     run = run->Next;
+   }
+
+   return false;
+ }
+
+ /*!
+  * \brief
+  *   Checks if a page contains only elements within the free list
+  *
+  * \param page
+  *   the first byte of the page
+  *
+  * \return
+  *   returns true if the page is free
+  */
+ bool ObjectAllocator::PageFree(byte *page) const {
+   //move to the first data block of data
+   page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+   //loop through the page checking if each block is corrupted
+   for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)
+   {
+     //check if the pad bytes around the block corrupted
+     if (!Freed(page))
+       return false;
+   }
+
+   return true;
+ }
+
+ /*!
+  * \brief
+  *   Reorders the free list to not include the page passed in
+  *
+  * \param page
+  *   the page to remove the free list from
+  */
+ void ObjectAllocator::RemoveFreeListPage(byte *page) {
+   //move to where the block of data would be
+   page += sizeof(void *) + Config_.LeftAlignSize_ + Config_.HBlockInfo_.size_ + Config_.PadBytes_;
+
+   //setup pointers for freed list
+   for (size_t i = 0; i < Config_.ObjectsPerPage_; ++i, page += InterBlockSize_)
+   {
+     RemoveNode(&FreeList_, reinterpret_cast<GenericObject *>((page)));
+   }
+ }
+
+ /*!
+  * \brief
+  *   removes a node from a list
+  *
+  * \param list
+  *   the list you want to remove the node from
+  *
+  * \param node
+  *   the node to remove
+  */
+ void ObjectAllocator::RemoveNode(GenericObject **list, GenericObject *node)
+ {
+   GenericObject *hold = *list;   //pointer holder for removed node
+   GenericObject *remove = *list; //pointer holder for removed node
+
+   //move to where the node is
+   while (remove != node && remove->Next)
+   {
+     hold = remove;
+     remove = remove->Next;
+   }
+
+   //to avoid issues with null
+   if (!remove) return;
+
+   if (remove == *list)    //if its the front of the list
+     *list = remove->Next; //set new head
+   else
+     hold->Next = remove->Next; //move to next node
+ }
\ No newline at end of file
```

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

```
+ /******************************************************************************/
+ /*!
+ \file   ObjectAllocator.h
+ \author Tristan Bouchard
+ \par    email: Tristan.Bouchard\@digipen.edu
+ \par    DigiPen login: tristan.bouchard
+ \par    Course: CS280
+ \par    Section A
+ \par    Assignment #1
+ \date   9/19/2019
+ \brief
+   This file contains the prototypes for objects and functions for the
+   ObjectAllocator
+ */
+ /******************************************************************************/
+ #ifndef ACTION_MEMORY_H
+ #define ACTION_MEMORY_H
+ #include <string>
```

```
+
+ //! If not specified, the objects per page will be this
+ static const int DEFAULT_OBJECTS_PER_PAGE = 4;
+
+ //! If not specified, the max pages will be this
+ static const int DEFAULT_MAX_PAGES = 3;
+
+ /*!
+   Exception class
+ */
+ class OAException
+ {
+   public:
+     /*!
+       Possible exception codes
+     */
+     enum OA_EXCEPTION
+     {
+       E_NO_MEMORY,      //!< out of physical memory (operator new fails)
+       E_NO_PAGES,       //!< out of logical memory (max pages has been reached)
+       E_BAD_BOUNDARY,   //!< block address is on a page, but not on any block-boundary
+       E_MULTIPLE_FREE,  //!< block has already been freed
+       E_CORRUPTED_BLOCK //!< block has been corrupted (pad bytes have been overwritten)
+     };
+
+     /*!
+       Constructor
+
+       \param ErrCode
+         One of the 5 error codes listed above
+
+       \param Message
+         A message returned by the what method.
+     */
+     OAException(OA_EXCEPTION ErrCode, const std::string& Message) : error_code_(ErrCode), message_(Message) {};
+
+     /*!
+       Destructor
+     */
+     virtual ~OAException() {
+     }
+
+     /*!
+       Retrieves the error code
+
+       \return
+         One of the 5 error codes.
+     */
+     OA_EXCEPTION code() const {
+       return error_code_;
+     }
+
+     /*!
+       Retrieves a human-readable string regarding the error.
+
+       \return
+         The NUL-terminated string representing the error.
+     */
+     virtual const char *what() const {
+       return message_.c_str();
+     }
+   private:
+     OA_EXCEPTION error_code_; //!< The error code (one of the 5)
+     std::string message_;     //!< The formatted string for the user.
+ };
+
+
+ /*!
+   ObjectAllocator configuration parameters
+ */
+ struct OAConfig
+ {
+   static const size_t BASIC_HEADER_SIZE = sizeof(unsigned) + 1; //!< allocation number + flags
+   static const size_t EXTERNAL_HEADER_SIZE = sizeof(void*);     //!< just a pointer
+
+ /*!
+   The different types of header blocks
+ */
+ enum HBLOCK_TYPE{hbNone, hbBasic, hbExtended, hbExternal};
+
+ /*!
+   POD that stores the information related to the header blocks.
+ */
+ struct HeaderBlockInfo
+ {
+   HBLOCK_TYPE type_;  //!< Which of the 4 header types to use?
+   size_t size_;       //!< The size of this header
+   size_t additional_; //!< How many user-defined additional bytes
+
+   /*!
+     Constructor
+
+     \param type
+       The kind of header blocks in use.
+
+     \param additional
+       The number of user-defined additional bytes required.
+
+   */
+   HeaderBlockInfo(HBLOCK_TYPE type = hbNone, unsigned additional = 0) : type_(type), size_(0), additional_(additional)
+   {
+     if (type_ == hbBasic)
+       size_ = BASIC_HEADER_SIZE;
+     else if (type_ == hbExtended) // alloc # + use counter + flag byte + user-defined
+       size_ = sizeof(unsigned int) + sizeof(unsigned short) + sizeof(char) + additional_;
+     else if (type_ == hbExternal)
+       size_ = EXTERNAL_HEADER_SIZE;
+   };
+ };
+
+ /*!
+   Constructor
+
+   \param UseCPPMemManager
+     Determines whether or not to by-pass the OA.
+
+   \param ObjectsPerPage
+     Number of objects for each page of memory.
+
+   \param MaxPages
+     Maximum number of pages before throwing an exception. A value
+     of 0 means unlimited.
+
+   \param DebugOn
+     Is debugging code on or off?
+
+   \param PadBytes
+     The number of bytes to the left and right of a block to pad with.
+
+   \param HBInfo
+     Information about the header blocks used
+
+   \param Alignment
+     The number of bytes to align on.
+ */
+ OAConfig(bool UseCPPMemManager = false,
+          unsigned ObjectsPerPage = DEFAULT_OBJECTS_PER_PAGE,
+          unsigned MaxPages = DEFAULT_MAX_PAGES,
+          bool DebugOn = false,
+          unsigned PadBytes = 0,
+          const HeaderBlockInfo &HBInfo = HeaderBlockInfo(),
+          unsigned Alignment = 0) : UseCPPMemManager_(UseCPPMemManager),
+                                    ObjectsPerPage_(ObjectsPerPage),
+                                    MaxPages_(MaxPages),
+                                    DebugOn_(DebugOn),
+                                    PadBytes_(PadBytes),
+                                    HBlockInfo_(HBInfo),
+                                    Alignment_(Alignment)
+ {
```

```
+     HBlockInfo_ = HBInfo;
+     LeftAlignSize_ = 0;
+     InterAlignSize_ = 0;
+   }
+
+   bool UseCPPMemManager_;      //!< by-pass the functionality of the OA and use new/delete
+   unsigned ObjectsPerPage_;    //!< number of objects on each page
+   unsigned MaxPages_;          //!< maximum number of pages the OA can allocate (0=unlimited)
+   bool DebugOn_;               //!< enable/disable debugging code (signatures, checks, etc.)
+   unsigned PadBytes_;          //!< size of the left/right padding for each block
+   HeaderBlockInfo HBlockInfo_; //!< size of the header for each block (0=no headers)
+   unsigned Alignment_;         //!< address alignment of each block
+   unsigned LeftAlignSize_;     //!< number of alignment bytes required to align first block
+   unsigned InterAlignSize_;    //!< number of alignment bytes required between remaining blocks
+ };
+
+
+ /*!
+   POD that holds the ObjectAllocator statistical info
+ */
+ struct OAStats
+ {
+   /*!
+     Constructor
+   */
+   OAStats() : ObjectSize_(0), PageSize_(0), FreeObjects_(0), ObjectsInUse_(0), PagesInUse_(0),
+                   MostObjects_(0), Allocations_(0), Deallocations_(0) {};
+
+   size_t ObjectSize_;      //!< size of each object
+   size_t PageSize_;        //!< size of a page including all headers, padding, etc.
+   unsigned FreeObjects_;   //!< number of objects on the free list
+   unsigned ObjectsInUse_;  //!< number of objects in use by client
+   unsigned PagesInUse_;    //!< number of pages allocated
+   unsigned MostObjects_;   //!< most objects in use by client at one time
+   unsigned Allocations_;   //!< total requests to allocate memory
+   unsigned Deallocations_; //!< total requests to free memory
+ };
+
+ /*!
+   This allows us to easily treat raw objects as nodes in a linked list
+ */
+ struct GenericObject
+ {
+   GenericObject *Next = 0; //!< The next object in the list
+ };
+
+ /*!
+   This is used with external headers
+ */
+ struct MemBlockInfo
+ {
+   bool in_use = 0;        //!< Is the block free or in use?
+   char *label = 0;        //!< A dynamically allocated NUL-terminated string
+   unsigned alloc_num = 0; //!< The allocation number (count) of this block
+ };
+
+ /*!
+   This class represents a custom memory manager
+ */
+ class ObjectAllocator
+ {
+   public:
+       // Defined by the client (pointer to a block, size of block)
+     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
+
+       // Predefined values for memory signatures
+     static const unsigned char UNALLOCATED_PATTERN = 0xAA; //!< New memory never given to the client
+     static const unsigned char ALLOCATED_PATTERN =   0xBB; //!< Memory owned by the client
+     static const unsigned char FREED_PATTERN =       0xCC; //!< Memory returned by the client
+     static const unsigned char PAD_PATTERN =         0xDD; //!< Pad signature to detect buffer over/under flow
+     static const unsigned char ALIGN_PATTERN =       0xEE; //!< For the alignment bytes
+
+       // Creates the ObjectManager per the specified values
+       // Throws an exception if the construction fails. (Memory allocation problem)
+    ObjectAllocator(size_t ObjectSize, const OAConfig& config) noexcept(false);
+
+      // Destroys the ObjectManager (never throws)
+    ~ObjectAllocator();
+
+      // 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 problem)
+    void *Allocate(const char *label = 0);
+
+      // Returns an object to the free list for the client (simulates delete)
+      // Throws an exception if the the object can't be freed. (Invalid object)
+    void Free(void *Object);
+
+      // Calls the callback fn for each block still in use
+    unsigned DumpMemoryInUse(DUMPCALLBACK fn) const;
+
+      // Calls the callback fn for each block that is potentially corrupted
+    unsigned ValidatePages(VALIDATECALLBACK fn) const;
+
+      // Frees all empty pages (extra credit)
+    unsigned FreeEmptyPages();
+
+      // Returns true if FreeEmptyPages and alignments are implemented
+    static bool ImplementedExtraCredit();
+
+      // Testing/Debugging/Statistic methods
+    void SetDebugState(bool State);   // true=enable, false=disable
+    const void *GetFreeList() const;  // returns a pointer to the internal free list
+    const void *GetPageList() const;  // returns a pointer to the internal page list
+    OAConfig GetConfig() const;       // returns the configuration parameters
+    OAStats GetStats() const;         // returns the statistics for the allocator
+
+      // Prevent copy construction and assignment
+    ObjectAllocator(const ObjectAllocator &oa) = delete;            //!< Do not implement!
+    ObjectAllocator &operator=(const ObjectAllocator &oa) = delete; //!< Do not implement!
+  private:
+    typedef unsigned char byte; //!< byte typedef for understanding code better
+
+    // Some "suggested" members (only a suggestion!)
+    GenericObject *PageList_ = { 0 }; //!< the beginning of the list of pages
+    GenericObject *FreeList_ = { 0 }; //!< the beginning of the list of objects
+    OAConfig Config_;                 //!< config for the OA
+    OAStats Stats_;                   //!< stats gotten for the OA at run time
+    size_t InterBlockSize_;           //!< the inter block size from the OA
+
+    //sets alignment and returns the page size
+    size_t Get_PageSize();
+
+    //allocates the memory needed for a page
+    byte *Allocate_Page();
+
+    //populates the page with debug pad bytes
+    void Populate_Page(byte *page);
+
+    //adds a page to the page list
+    void Add_Page();
+
+    //returns the header of an object
+    byte *Get_Header(byte *object) const;
+
+    //updates the header of the allocated block of data passed to the client
+    void Allocate_Header(byte *header, const char *label);
+
+    //updates the header of the freed header
+    void Free_Header(byte *header);
+
+    //checks if an object has been freed
+    bool Freed(byte *object) const;
+
+    //checks if an object has been corrupted
+    bool Corrupted(byte *object) const;
+
+    //checks if an object is out of bounds
+    bool OutOfBounds(byte *object) const;
+
+     //check if a page has no objects allocated on it
+     bool PageFree(byte *page) const;
+ 
+     //reorder the free list to not include the page passed in
+     void RemoveFreeListPage(byte *page);
+ 
+     //generic node removal
+     void RemoveNode(GenericObject **list,GenericObject *node);
+ };
+ 
+ #endif
```

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

```
+ #include "stdafx.h"
+ #include "Action_Parser.h"
+ #include <cstring>
+ #include <algorithm>
+ 
+ std::map<std::string, std::map<std::string,std::string>> ActionParser::ActionData_;
+ std::string ActionParser::currentActionGroup_;
+ 
+ void ActionParser::Parse(std::string Filename)
+ {
+   std::ifstream infile(Filename);
+   if (infile.is_open())
+     PreProccess(infile);
+ }
+ 
+ void ActionParser::PreProccess(std::ifstream &infile)
+ {
+   std::string line;
+ 
+   auto GetNextChar = [](std::string str, size_t &index){
+     while (isspace(str[index])) ++index;
+   };
+ 
+   while (!infile.eof())
+   {
+     size_t index = 0;
+     std::getline(infile, line);
+ 
+     GetNextChar(line, index);
+     if (line[index] == '@')
+     {
+       //clear the old group name and set the new one
+       currentActionGroup_ = std::string(line.begin() + ++index, line.end());
+       std::transform(currentActionGroup_.begin(), currentActionGroup_.end(),
+                      currentActionGroup_.begin(), ::toupper);
+     }
+     else if (line[index] == '$')
+     {
+       GetNextChar(line, index); //get the first char of the name
+       int tempIndex = ++index;
+ 
+       //get the end of the word
+       while (!isspace(line[index]) && line[index] != '=') ++index;
+ 
+       std::string name(line.begin() + tempIndex, line.begin() + index); //get the action name
+       std::transform(name.begin(), name.end(), name.begin(), ::toupper);
+ 
+       while (line[index] != '=') ++index;
+       tempIndex = ++index;
+ 
+       while (line[index] != ';') ++index;
+       std::string expression(line.begin() + tempIndex, line.begin() + index);
+       ActionData_[currentActionGroup_][name] = expression;
+     }
+   }
+ 
+   infile.clear();                 // clear fail and eof bits
+   infile.seekg(0, std::ios::beg); // back to the start
+ }
```

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

```
+ #define ACTION_PARSER_H
+ #ifdef ACTION_PARSER_H
+ #include <fstream>
+ #include <string>
+ #include <map>
+
+ class ActionParser
+ {
+   public:
+     static std::map<std::string, std::map<std::string, std::string>> ActionData_; //action group <action name, expression>
+     static void Parse(std::string Filename = "0");
+     static std::string GetExpression(std::string actionGroup, std::string actionName); //for the action tree
+     static std::string GetCurrentActionGroup(); //for the action tree
+   private:
+     static void PreProccess(std::ifstream &infile); //maps actions to expressions
+     static std::string currentActionGroup_;
+ };
+
+ #endif
  \ No newline at end of file
```

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

```
+ #include "stdafx.h"
+ #include "Action_Parser.h"
+ #include "Actions.h"
+
+ void ActionTree::Generate(std::string expression)
+ {
+   FreeTree(root_);
+   root_ = nullptr;
+   expression_ = expression;
+   GetToken();
+   CheckOr(root_);
+ }
+
+ ActionTree::~ActionTree()
+ {
+   FreeTree(root_);
+ }
+
+ ActionTree::TreeNode::TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool()> inputCallBack) :
+ left_(left), right_(right), type_(type), input_(inputCallBack) {}
+
+ ActionTree::TreeNode::TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool(bool,bool)> opCallBack) :
+ left_(left), right_(right), type_(type), operator_(opCallBack) {}
+
+ bool ActionTree::EvaluateTree() const
+ {
+   return Evaluate(root_);
+ }
+
+ bool ActionTree::Evaluate(TreeNode *tree) const
+ {
+   if (tree->type_ == TreeNode::ANDOP || tree->type_ == TreeNode::OROP)
+     return tree->operator_(Evaluate(tree->left_), Evaluate(tree->right_));
+   return tree->input_();
+ }
+
+ //evaluate the boolean expression
+ //bool ActionTree::Evaluate()
+ //{
+ //  return false;
+ //}
+
+ void ActionTree::GetToken()
+ {
+   currToken_.clear(); //clear the token
+
+   //go to the next non-whitespace char
+   while (isspace(expression_[currPos_])) ++currPos_;
+
+   currToken_.push_back(expression_[currPos_++]);
```

```
+
+   //check if we got an operator
+   if (IsOperator(currToken_))
+     return;
+
+   //get the rest of the input hash and return
+   while (IsIdentifier(expression_[currPos_]))
+     currToken_.push_back(expression_[currPos_++]);
+
+   //uppercase everything
+   std::transform(currToken_.begin(), currToken_.end(), currToken_.begin(), ::toupper);
+ }
+
+ void ActionTree::CheckOr(TreeNode *&tree) // |
+ {
+   CheckAnd(tree);
+
+   while (currToken_ == "|")
+   {
+     auto temp = new TreeNode(nullptr, nullptr, TreeNode::OROP, [](bool L,bool R) { return L | R; });
+     temp->left_ = tree;
+     tree = temp;
+     GetToken();
+     CheckAnd(tree->right_);
+   }
+ }
+
+ void ActionTree::CheckAnd(TreeNode *&tree) // &
+ {
+   CheckNot(tree);
+   while (currToken_ == "&")
+   {
+     auto temp = new TreeNode(nullptr, nullptr, TreeNode::ANDOP, [](bool L,bool R) { return L & R; });
+     temp->left_ = tree;
+     tree = temp;
+     GetToken();
+     CheckNot(tree->right_);
+   }
+ }
+
+ void ActionTree::CheckNot(TreeNode *&tree) // !
+ {
+   //go to the end of all the not(s) and set the value there
+   if (currToken_ == "!")
+   {
+     GetToken();
+     CheckRest(tree);
+     std::function<bool()> temp = tree->input_;
+     tree->input_ = [temp](){return !temp();};
+   }
+   else
+     CheckRest(tree);
+ }
+
+ void ActionTree::CheckRest(TreeNode *&tree) // ( or input
+ {
+   if (currToken_ == "(")
+   {
+     GetToken();
+     CheckOr(tree);
+   }
+
+   //check if an action is within the preprocessed map
+   auto NestedActionCheck = ActionParser::ActionData_[ActionManager::CurrectActionGroup()].find(currToken_);
+   if (NestedActionCheck != ActionParser::ActionData_[ActionManager::CurrectActionGroup()].end())
+   {
+     //maybe move this within the lambda as a static member
+     //play around with this. Pointers might be an issue here. Or its very expensive
+     tree = new TreeNode(nullptr, nullptr, TreeNode::INPUT, [=]() {
+       ActionTree nested_tree;
+       nested_tree.Generate(NestedActionCheck->second);
+       return nested_tree.EvaluateTree();});
+   }
+   //TODO: put callback to input system here (check if key is within the action system)
+   else if(IsInput(currToken_))
+     tree = new TreeNode(nullptr, nullptr, TreeNode::INPUT, []() {return true; });
+
+   GetToken();
+ }
+
+ bool ActionTree::IsOperator(std::string token)
+ {
+   return (token == "|" || token == "&" || token == "!" || token == "(");
+ }
+
+ bool ActionTree::IsIdentifier(char token)
+ {
+   return (token >= 'a' && token <= 'z') || (token >= 'A' && token <= 'Z') || token == '_';
+ }
+
+ bool ActionTree::IsInput(std::string token)
+ {
+   //index into the map that has the look up keys
+   return token != ")" && !IsOperator(token);
+ }
+
+ void ActionTree::FreeTree(TreeNode *tree)
+ {
+   if (!tree)
+     return;
+
+   FreeTree(tree->left_);
+   FreeTree(tree->right_);
+   FreeNode(tree);
+ }
+
+ void ActionTree::FreeNode(TreeNode *node)
+ {
+   delete node;
+ }
+
+ //std::function<bool()> ActionTree::GetInputCallBack(std::string input)
+ //{
+ //  if (input[0] == 'k' || input[0] == 'K')
+ //  {
+ //    //check if input is within the input system
+ //    //return a callback function to the keyboard input system
+ //  }
+ //  if (input[0] == 'c' || input[0] == 'C')
+ //  {
+ //    //check if input is within the input system
+ //    //return a callback to the controller input system
+ //  }
+ //
+ //  //return a temp dummy function
+ //  return [](){ return true; };
+ //}
  \ No newline at end of file
```

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

```
+ #define ACTION_TREE_H
+ #ifdef ACTION_TREE_H
+ #include <string>
+ #include <map>
+ #include <functional>
+
+ class ActionTree
+ {
+   struct TreeNode;
+ public:
+   void Generate(std::string expression);
+   ~ActionTree();
+   bool EvaluateTree() const;
+ private:
+   bool Evaluate(TreeNode *tree) const;
+   TreeNode *root_ = nullptr;
+   unsigned currPos_ = 0;
+   std::string currToken_;
+   std::string expression_;
```

```
+
+  struct TreeNode
+  {
+    enum NodeType {ANDOP, OROP, INPUT};
+    TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool()> inputCallBack);
+    TreeNode(TreeNode *left, TreeNode *right, NodeType type, std::function<bool(bool,bool)> opCallBack);
+    TreeNode *left_;   // left sub-expression
+    TreeNode *right_;  // right sub-expression
+    NodeType type_;    // kind of node this is
+    std::function<bool()> input_;
+    std::function<bool(bool,bool)> operator_;
+  };
+
+  void GetToken();
+
+  void CheckOr(TreeNode *&tree);   // |
+  void CheckAnd(TreeNode *&tree);  // &
+  void CheckNot(TreeNode *&tree);  // !
+  void CheckRest(TreeNode *&tree); // ( or input
+
+  void FreeTree(TreeNode *tree);
+  void FreeNode(TreeNode *node);
+
+  bool IsOperator(std::string token);
+  bool IsIdentifier(char token);
+  bool IsInput(std::string token);
+ };
+
+ #endif
  \ No newline at end of file
```

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

```
+ #include "stdafx.h"
+ #include "Actions.h"
+ #include "Action_Parser.h"
+ #include <fstream>
+
+ std::map<std::string, std::map<std::string, ActionTree>> ActionManager::actionManager_;
+ std::string ActionManager::currentActionGroup_;
+
+ void ActionManager::Import(std::string Filename)
+ {
+   ActionParser::Parse(Filename);
+   auto groupItter = ActionParser::ActionData_.begin();
+   while (groupItter != ActionParser::ActionData_.end())
+   {
+     //loop through each action within a group generating their ActionTree's
+     auto actionItter = groupItter->second.begin();
+     while (actionItter != groupItter->second.end())
+     {
+       actionManager_[groupItter->first][actionItter->first].Generate(actionItter->second);
+       ++actionItter;
+     }
+
+     ++groupItter;
+   }
+ }
+
+ void ActionManager::Reimport(std::string Filename)
+ {
+   actionManager_.clear();
+   Import(Filename);
+ }
+
+ bool ActionManager::Action_Pressed(std::string action)
+ {
+   return false;
+ }
+
+ bool ActionManager::Action_Held(std::string action)
+ {
+   std::transform(action.begin(), action.end(), action.begin(), ::toupper);
+   return actionManager_[currentActionGroup_][action].EvaluateTree();
+ }
```

```
+
+ bool ActionManager::Action_Released(std::string action)
+ {
+   return false;
+ }
+
+ void ActionManager::LoadActionGroup(std::string actionGroup)
+ {
+   std::transform(actionGroup.begin(), actionGroup.end(), actionGroup.begin(), ::toupper);
+   currentActionGroup_ = actionGroup;
+ }
+
+ std::string ActionManager::CurrectActionGroup()
+ {
+   return currentActionGroup_;
+ }
  \ No newline at end of file
```

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

```
+ #define ACTIONS_H
+ #ifdef ACTIONS_H
+ #include "Action_Tree.h"
+ #include <string>
+ #include <map>
+
+ class ActionManager
+ {
+   public:
+     static void Import(std::string Filename);
+     static void Reimport(std::string Filename);
+
+     static bool Action_Pressed(std::string action);
+     static bool Action_Held(std::string action);
+     static bool Action_Released(std::string action);
+
+     static void LoadActionGroup(std::string actionGroup);
+     static std::string CurrectActionGroup();
+   private:
+     static std::map<std::string, std::map<std::string, ActionTree>> actionManager_;
+     static std::string currentActionGroup_;
+ };
+
+ #endif
  \ No newline at end of file
```

Please register or sign in to comment