jdeanmillard / ObjectAllocator.cpp

Last active 6 years ago • Report abuse

Star

Code Revisions 2

Implementation for CS 280 Object Allocator

ObjectAllocator.cpp

```
/******************************************************************************/
/*!
\file   ObjectAllocator.cpp
\author Justin Millard
\par    email: justin.millard\@digipen.edu
\par    DigiPen login: justin.millard
\par    Course: CS280
\par    Programming Assignment #1
\date   1/21/2013
\brief
  This file contains the implementation for the Object Allocator.
*/
/******************************************************************************/

#include "ObjectAllocator.h"
#include <iostream>
#include <cstring>
#include <stdlib.h>
//#include "LeakCheck.h"
//#undef THIS_FILE
//static char THIS_FILE[] = __FILE__;
#define u_char unsigned char

/******************************************************************************/
/*!
  \brief
    Creates the ObjectAllocator per the specified values. Throws an exception
    if the construction fails. (Memory allocation problem)

  \param ObjectSize
    The size of each object than can be allocated.

  \param config
    Contains configuration details for object allocator.
*/
```

```
/******************************************************************************/
ObjectAllocator::ObjectAllocator(unsigned ObjectSize, const OAConfig& config) throw(OAException)
: Config_(config), OAStats_(), PageList(NULL), FreeList(NULL)
{
  OAStats_.ObjectSize_ = ObjectSize;
  if (Config_.Alignment_ != 0)
  {
    if (sizeof(void*) + Config_.PadBytes_ + Config_.HeaderBlocks_ <= Config_.Alignment_)
    {
      Config_.LeftAlignSize_ = Config_.Alignment_ - Config_.PadBytes_ - sizeof(void*)
        - Config_.HeaderBlocks_;
    }
    else
    {
      Config_.LeftAlignSize_ = Config_.Alignment_ * 2 - Config_.PadBytes_ - sizeof(void*)
        - Config_.HeaderBlocks_;
    }

    if (Config_.HeaderBlocks_ + Config_.PadBytes_ * 2 < Config_.Alignment_)
    {
      Config_.InterAlignSize_ = Config_.Alignment_ - Config_.HeaderBlocks_
        - Config_.PadBytes_ * 2;
    }
    else
    {
      Config_.InterAlignSize_ = Config_.Alignment_ * 2 - Config_.HeaderBlocks_
        - Config_.PadBytes_ * 2;
    }
  }
  OAStats_.PageSize_ = Config_.ObjectsPerPage_ * ( ObjectSize + Config_.PadBytes_ * 2
    + Config_.InterAlignSize_ + Config_.HeaderBlocks_ )
    - Config_.InterAlignSize_ + Config_.LeftAlignSize_ + sizeof(void*);

  if (Config_.UseCPPMemManager_ == false)
  {
    try
    {
      AddNewPage();
    }
    catch(OAException exception)
    {
      throw(exception);
    }
  }
}

/******************************************************************************/
/*!
  \brief
```

```
    Destroys the ObjectAllocator (never throws)
*/
/******************************************************************************/
ObjectAllocator::~ObjectAllocator() throw()
{
  u_char* page;
  while (PageList != NULL)
  {
    page = reinterpret_cast<u_char*>(PageList);
    PageList = PageList->Next;
    delete [] page;
  }
}

/******************************************************************************/
/*!
  \brief
    Allocates the memory for a new page, and sets up relevant pointers and
    signatures.

  \param nextPage
    The page following the new one.
*/
/******************************************************************************/
void ObjectAllocator::AddNewPage(GenericObject* nextPage) throw(OAException)
{
  u_char* address;
  try
  {
    address = new u_char[OAStats_.PageSize_];
  }
  catch(std::bad_alloc &)
  {
    throw(OAException(OAException::E_NO_MEMORY, "Not enough memory."));
  }

  PageList = reinterpret_cast<GenericObject*>(address);
  PageList->Next = nextPage;
  address += sizeof(void*);
  ++OAStats_.PagesInUse_;

  GenericObject* nextList;

  for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)
  {
    // Get address to proper alignment
    if (i == 0)
    {
      if (Config_.DebugOn_ == true)
      {
        std::memset(address, ALIGN_PATTERN, Config_.LeftAlignSize_);
      }
      if (i < Config_.ObjectsPerPage_ - 1)
      {
        address += Config_.LeftAlignSize_;
      }
    }
    else
    {
      if (Config_.DebugOn_ == true)
      {
        std::memset(address, ALIGN_PATTERN, Config_.InterAlignSize_);
      }
      address += Config_.InterAlignSize_;
    }
    // Add header
    if (Config_.DebugOn_ == true)
    {
      std::memset(address, 0, Config_.HeaderBlocks_);
    }
    address += Config_.HeaderBlocks_;

    // Add start padding
    if (Config_.DebugOn_ == true)
    {
      std::memset(address, PAD_PATTERN, Config_.PadBytes_);
    }
    address += Config_.PadBytes_;

    // Add data bytes and set new free list
    if (Config_.DebugOn_ == true)
    {
      std::memset(address, UNALLOCATED_PATTERN, OAStats_.ObjectSize_);
    }
    nextList = FreeList;
    FreeList = reinterpret_cast<GenericObject*>(address);
    ++OAStats_.FreeObjects_;
    FreeList->Next = nextList;
    address += OAStats_.ObjectSize_;

    // Add end padding
    if (Config_.DebugOn_ == true)
    {
      std::memset(address, PAD_PATTERN, Config_.PadBytes_);
    }
    address += Config_.PadBytes_;
  }
}
```

```

/******************************************************************************/
/*!
  \brief
    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)

  \return
    A pointer to the new object.
*/
/******************************************************************************/
void *ObjectAllocator::Allocate() throw(OAException)
{
  // Ignore most of the function if using CPP Memory Manager
  if (Config_.UseCPPMemManager_ == true)
  {
    u_char* address;
    try
    {
      address = new u_char[OAStats_.ObjectSize_];
    }
    catch(std::bad_alloc &)
    {
      throw(OAException(OAException::E_NO_MEMORY, "Not enough memory."));
    }
    ++OAStats_.ObjectsInUse_;
    if (OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
    {
      ++OAStats_.MostObjects_;
    }
    ++OAStats_.Allocations_;
    --OAStats_.FreeObjects_;
    return address;
  }

  if (PageList == NULL)
  {
    try
    {
      AddNewPage();
    }
    catch(OAException exception)
    {
      throw(exception);
    }
  }
  else if (FreeList == NULL)
  {
    if (Config_.MaxPages_ == 0 || OAStats_.PagesInUse_ < Config_.MaxPages_)
    {
      try
      {
        AddNewPage(PageList);
      }
      catch(OAException exception)
      {
        throw(exception);
      }
    }
    else
    {
      throw(OAException(OAException::E_NO_PAGES, "No more pages can be allocated."));
    }
  }

  // If we've gotten this far, object can be properly given to client
  u_char* newMem = reinterpret_cast<u_char*>(FreeList);
  FreeList = FreeList->Next;
  if (Config_.DebugOn_ == true)
  {
    std::memset(newMem, ALLOCATED_PATTERN, OAStats_.ObjectSize_);
  }
  if (Config_.HeaderBlocks_ != 0)
  {
    u_char* header = newMem - Config_.PadBytes_ - sizeof(u_char);
    *header = static_cast<u_char>(1);
  }
  ++OAStats_.ObjectsInUse_;
  if (OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
  {
    ++OAStats_.MostObjects_;
  }
  ++OAStats_.Allocations_;
  --OAStats_.FreeObjects_;
  return newMem;
}

/******************************************************************************/
/*!
  \brief
    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)

  \param Object
    The object.
*/
/******************************************************************************/
```

```
void ObjectAllocator::Free(void *Object) throw(OAException)
{
  // Skips most of the function if using CPP memory manager
  if (Config_.UseCPPMemManager_ == true)
  {
    u_char* block = reinterpret_cast<u_char*>(Object);
    delete [] block;

    --OAStats_.ObjectsInUse_;
    ++OAStats_.Deallocations_;
    ++OAStats_.FreeObjects_;

    return;
  }

  GenericObject* page = PageList;
  u_char* objAddress = reinterpret_cast<u_char*>(Object);

  // Only perform checks when debugging
  if (Config_.DebugOn_ == true)
  {
    bool onBound = false;
    bool corrupted = false;
    bool alreadyFreed = true;

    while (page != NULL)
    {
      // Check if object is within page boundaries
      if ( reinterpret_cast<u_char*>(page) < objAddress
        && objAddress < reinterpret_cast<u_char*>(page) + OAStats_.PageSize_)
      {
        u_char* blockStart = reinterpret_cast<u_char*>(page) + sizeof(void*) +
            Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;
        size_t separation = OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 +
                            Config_.InterAlignSize_ + Config_.HeaderBlocks_;
        // Check if object is on a block boundary
        if ( 0 == (objAddress - blockStart) % separation )
        {
          onBound = true;
        }
      }
      page = page->Next;
    }

    if (onBound == false)
    {
      throw(OAException(OAException::E_BAD_BOUNDARY, "Object not on a block boundary"));
    }
    else
    {
      // Check data block
      for (unsigned i = sizeof(u_char*); i < OAStats_.ObjectSize_; ++i)
      {
        if ( objAddress[i] != FREED_PATTERN )
        {
          alreadyFreed = false;
          break;
        }
      }

      // Check padding blocks
      u_char* leftPad = objAddress - Config_.PadBytes_;
      u_char* rightPad = objAddress + OAStats_.ObjectSize_;
      for (unsigned i = 0; i < Config_.PadBytes_; ++i)
      {
        if (leftPad[i] != PAD_PATTERN || rightPad[i] != PAD_PATTERN)
        {
          corrupted = true;
          break;
        }
      }
    }

    if (alreadyFreed == true)
    {
      throw(OAException(OAException::E_MULTIPLE_FREE, "Block already freed"));
    }
    else if (corrupted == true)
    {
      throw(OAException(OAException::E_CORRUPTED_BLOCK, "Block is corrupted"));
    }
  }

  // If we've gotten this far, we can finally free memory
  if (Config_.HeaderBlocks_ != 0)
  {
    u_char* header = objAddress - Config_.PadBytes_ - sizeof(char);
    *header = 0;
  }
  if (Config_.DebugOn_ == true)
  {
    std::memset(Object, FREED_PATTERN, OAStats_.ObjectSize_);
  }
  GenericObject* block = reinterpret_cast<GenericObject*>(Object);
  block->Next = FreeList;
  FreeList = block;

  --OAStats_.ObjectsInUse_;
  ++OAStats_.Deallocations_;
  ++OAStats_.FreeObjects_;
}

/******************************************************************************/
/*!
  \brief
    Calls the callback fn for each block in use.

  \param fn
    A function pointer for the callback function.

  \return
    The number of objects in use.
*/
/******************************************************************************/
unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
{
  GenericObject* page = PageList;
  u_char* block;
  while (page)
  {
    block = reinterpret_cast<u_char*>(page) + sizeof(void*) + Config_.LeftAlignSize_
          + Config_.HeaderBlocks_ + Config_.PadBytes_;
    for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)
    {
      if (*block == ALLOCATED_PATTERN)
      {
        fn(block, OAStats_.ObjectSize_);
      }
      if (i < Config_.ObjectsPerPage_ - 1)
      {
        block += OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 + Config_.InterAlignSize_
          + Config_.HeaderBlocks_;
      }
    }
    page = page->Next;
  }
  return OAStats_.ObjectsInUse_;
}

/******************************************************************************/
/*!
  \brief
    Calls the callback fn for each block that is potentially corrupted.

  \param fn
    A function pointer for the callback function.

  \return
    The number of corrupted blocks.
*/
/******************************************************************************/
unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
{
  if (Config_.PadBytes_ == 0)
  {
    return 0;
  }

  unsigned corruptedBlocks = 0;
  GenericObject* page = PageList;
  u_char* block;
  u_char* leftPad;
  u_char* rightPad;

  while (page)
  {
    block = reinterpret_cast<u_char*>(page) + sizeof(void*) + Config_.LeftAlignSize_
      + Config_.HeaderBlocks_ + Config_.PadBytes_;
    for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)
    {
      // Check padding blocks
      leftPad = block - Config_.PadBytes_;
      rightPad = block + OAStats_.ObjectSize_;
      for (unsigned j = 0; j < Config_.PadBytes_; ++j)
      {
        if (leftPad[j] != PAD_PATTERN || rightPad[j] != PAD_PATTERN)
        {
          fn(block, OAStats_.ObjectSize_);
          ++corruptedBlocks;
          break;
        }
      }

      // Prevents pointer arithmetic overrun
      if (i < Config_.ObjectsPerPage_ - 1)
      {
        block += OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 + Config_.InterAlignSize_
          + Config_.HeaderBlocks_;
      }
    }
    page = page->Next;
  }

  return corruptedBlocks;
}

```

```
/******************************************************************************/
/*!
  \brief
    Frees all empty pages (pages whose blocks are all on the FreeList.)

  \return
    The number of empty pages.
*/
/******************************************************************************/
unsigned ObjectAllocator::FreeEmptyPages()
{
  unsigned emptyPages = 0;
  GenericObject* currentPage = PageList;
  u_char* mem;

  if (!currentPage)
  {
    return emptyPages;
  }

  // Delete first page if it is empty. If subsequent page(s) are empty, delete
  // those too. Set PageList pointer at first non-empty page
  while (currentPage && IsEmpty(currentPage))
  {
    ++emptyPages;
    --OAStats_.PagesInUse_;
    RemovePageBlocks(currentPage);
    mem = reinterpret_cast<u_char*>(currentPage);
    currentPage = currentPage->Next;
    PageList = currentPage;
    delete [] mem;
  }
  while (currentPage)
  {
    while (currentPage->Next && IsEmpty(currentPage->Next))
    {
      ++emptyPages;
      --OAStats_.PagesInUse_;
      RemovePageBlocks(currentPage->Next);
      mem = reinterpret_cast<u_char*>(currentPage->Next);
      currentPage->Next = currentPage->Next->Next;
      delete [] mem;
    }
    currentPage = currentPage->Next;
  }
  return emptyPages;
}

/******************************************************************************/
```

```
/*!
  \brief
    Checks whether or not a page is empty (all its objects on the FreeList.)

  \param page
    The page to be checked.

  \return
    Whether or not it is empty.
*/
/******************************************************************************/
bool ObjectAllocator::IsEmpty(GenericObject* page)
{
  u_char* pageBlock = reinterpret_cast<u_char*>(page) + sizeof(void*) +
    Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;
  size_t separation = OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 +
    Config_.InterAlignSize_ + Config_.HeaderBlocks_;

  // Go through different blocks
  for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)
  {
    // If any block is on the Free List, page is not empty
    if (OnFreeList(pageBlock) == false)
    {
      return false;
    }
    if (i < Config_.ObjectsPerPage_ - 1)
    {
      pageBlock += separation;
    }
  }

  return true;
}

/******************************************************************************/
/*!
  \brief
    Checks whether or not a block is on the FreeList.

  \param block
    The address of block.

  \return
    Whether or not the block is on the FreeList.
*/
/******************************************************************************/
bool ObjectAllocator::OnFreeList(unsigned char* block)
{
  GenericObject* currentBlock = FreeList;

  while (currentBlock != NULL)
  {
    if (block == reinterpret_cast<u_char*>(currentBlock))
    {
      return true;
    }
    currentBlock = currentBlock->Next;
  }

  return false;
}

/******************************************************************************/
/*!
  \brief
    Gets previous block from FreeList. Returns null if none can be found or if
    block is head of list.

  \param FreeListBlock
    The address of the current block.

  \return
    The previous block (if it exists.)
*/
/******************************************************************************/
GenericObject* ObjectAllocator::GetPreviousBlock(GenericObject* FreeListBlock)
{
  GenericObject* currentBlock = FreeList;

  while (currentBlock->Next)
  {
    if (currentBlock->Next == FreeListBlock)
    {
      return currentBlock;
    }
    currentBlock = currentBlock->Next;
  }

  return NULL;
}

/******************************************************************************/
/*!
  \brief
    Travels through FreeList and removes the blocks that are found on the
    designated page.

  \param page
    The page containing the blocks to remove.

  \return
    The previous block (if it exists.)
*/
/******************************************************************************/
void ObjectAllocator::RemovePageBlocks(GenericObject* page)
{
  size_t separation = OAStats_.ObjectSize_ + Config_.PadBytes_ * 2 +
    Config_.InterAlignSize_ + Config_.HeaderBlocks_;

  GenericObject* currentBlock;
  GenericObject* previousBlock;
  u_char* address = reinterpret_cast<u_char*>(page) + sizeof(void*) +
    Config_.LeftAlignSize_ + Config_.PadBytes_ + Config_.HeaderBlocks_;

  // Go through different blocks
  for (unsigned i = 0; i < Config_.ObjectsPerPage_; ++i)
  {
    currentBlock = reinterpret_cast<GenericObject*>(address);
    previousBlock = GetPreviousBlock(currentBlock);
    // Change FreeList pointer if FreeList is found on this page
    if (currentBlock && FreeList == currentBlock)
    {
      --OAStats_.FreeObjects_;
      FreeList = currentBlock->Next;
    }
    // Change next pointers if they point to object from this page
    if (previousBlock)
    {
      --OAStats_.FreeObjects_;
      previousBlock->Next = currentBlock->Next;
    }
    if (i < Config_.ObjectsPerPage_ - 1)
    {
      address += separation;
    }
  }
}

/******************************************************************************/
/*!
  \brief
    Returns true if FreeEmptyPages and alignments are implemented

  \return
    Whether or not alignments and FreeEmptyPages are implemented.
*/
```

```
/******************************************************************************/
bool ObjectAllocator::ImplementedExtraCredit()
{
  return true;
}

/******************************************************************************/
/*!
  \brief
    Sets debug state of allocator.

  \param State
    Whether or not the allocator is in debug mode.
*/
/******************************************************************************/
void ObjectAllocator::SetDebugState(bool State)
{
  Config_.DebugOn_ = State;
}

/******************************************************************************/
/*!
  \brief
    Returns the first object on the FreeList.

  \return
    The first object on the FreeList.
*/
/******************************************************************************/
const void * ObjectAllocator::GetFreeList() const
{
  return FreeList;
}

/******************************************************************************/
/*!
  \brief
    Returns the first object on the PageList.

  \return
    The first object on the PageList.
*/
/******************************************************************************/
const void * ObjectAllocator::GetPageList() const
{
  return PageList;
}

/******************************************************************************/
```

```
/*!
  \brief
    Returns the config details of the ObjectAlloator.

  \return
    A copy of the allocator's config details.
*/
/******************************************************************************/
OAConfig ObjectAllocator::GetConfig() const
{
  return Config_;
}

/******************************************************************************/
/*!
  \brief
    Returns the stat details of the ObjectAlloator.

  \return
    A copy of the allocator's stat details.
*/
/******************************************************************************/
OAStats ObjectAllocator::GetStats() const
{
  return OAStats_;
}
```