ShumWengSang / CS280

Code Issues Pull requests Actions Projects Security Insights

master

CS280 / Assignment1 / src / ObjectAllocator.h

ShumWengSang Seems to be done History

1 contributor

Raw Blame

327 lines (282 sloc) | 13.3 KB

```
/******************************************************************************/
/*!
\file   ObjectAllocator.h
\author Roland Shum
\par    email: roland.shum\@digipen.edu
\par    DigiPen login: roland.shum
\par    Course: CS280
\par    Assignment #1
\date   1/24/2020
\brief
  This is the interface file for all member functions
  of class ObjectAllocator. OAConfig and OAStats are
  classes that are paired with Object Allocator.
  OAException is the class that the OAAllocator would
  throw if it fails.

*/
/******************************************************************************/
//----------------------------------------------------------------------------
#ifndef OBJECTALLOCATORH
#define OBJECTALLOCATORH
//----------------------------------------------------------------------------


#include <string>  // string
#include <iostream>
// If the client doesn't specify these:
static const int DEFAULT_OBJECTS_PER_PAGE = 4;//! Default Objects Per Page
```

```cpp
static const int DEFAULT_MAX_PAGES = 3;       //! Default Maximum Amount Of Pages

/******************************************************************************/
/*!
  \class OAException
  \brief
        Exception class for the object allocator. Possible Exceptions are:
        - NO_MEMORY
        - NO_PAGES
        - BAD_BOUNDARY
        - MULTIPLE_FREE
        - CORRUPTED_BLOCK
*/
/******************************************************************************/
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)
    };

    OAException(OA_EXCEPTION ErrCode, const std::string &Message) :
        error_code_(ErrCode), message_(Message) {};

    virtual ~OAException()
    {
    }

    OA_EXCEPTION code(void) const
    {
        return error_code_;
    }

    virtual const char *what(void) const
    {
        return message_.c_str();
    }


private:
    OA_EXCEPTION error_code_;    //! Exception error code
    std::string message_;        //! Exception message
};
```

```

// 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 type of header it is.
    enum HBLOCK_TYPE
    {
        hbNone,     //! No Header
        hbBasic,    //! Basic Header
        hbExtended, //! Extended Header
        hbExternal  //! External Header
    };

    /**************************************************************************/
    /*!
      \class HeaderBlockInfo
      \brief
        Class used in OAConfig to determine what type of header to use.
        User can choose from None, Basic, Extended, and External.
        None means no header. Basic enables checking allocation number and
        checking if block is active. Extended extends basic to include
        a user defined byte field, and a use counter. External use an external
        memory block to moniter the data.
    */
    /**************************************************************************/
    struct HeaderBlockInfo
    {
        HBLOCK_TYPE type_;  //! Describes the type of header.
        size_t size_;       //! Size of the header block
        size_t additional_; //! Additional sizing from user

        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;
        };

    };

    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);

    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 required to align first block
    unsigned InterAlignSize_;   //! number of alignment bytes required between data blocks
};

/*****************************************************************************/
/*!
  \class OAStats
  \brief
    ObjectAllocator statistical info
*/
/*****************************************************************************/
// ObjectAllocator statistical info
struct OAStats
{
    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
};

/*****************************************************************************/
/*!
  \class GenericObject

  \brief
    This class allows us to treat generic objects as raw pointers.
*/
/*****************************************************************************/
```

```
struct GenericObject
{
    GenericObject *Next; //! Pointer to next object in linked list.
};

/******************************************************************************/
/*!
  \class MemBlockInfo
  \brief
    This class defines what the external header is. When an external header is
    configured for the OA, the header would be a pointer to a MemBlockInfo
    object.
*/
/******************************************************************************/
struct MemBlockInfo
{
    bool in_use;        //! Is the block free or in use?
    char *label;        //! A dynamically allocated NUL-terminated string
    unsigned alloc_num; //! The allocation number (count) of this block

    MemBlockInfo(unsigned alloc_num, const char* label);
    ~MemBlockInfo();

    // Deleted default functions
    MemBlockInfo(const MemBlockInfo& ) = delete;
    MemBlockInfo& operator=(const MemBlockInfo&) = delete;
};

/******************************************************************************/
/*!
  \class ObjectAllocator
  \brief
    The class that is the object allocator. User contructs an allocator with
    a OAConfig instance, and can use Allocate and Free to use memory.

    Internally, allocates a pool based on the given configuration and allocates
    using that pool. If debug is turned on, it would be possible to detect
    more errors.

    Operations include:
    - Allocate
    - Free

    Debug Operations:

    - DumpMemoryInUse
    - ValidePages
    - FreeEmptyPages
    - SetDebugState
    - GetFreeList
    - GetPageList
    - GetConfig
    - GetStats
*/
/********************************************************************************/
// This memory manager class
class ObjectAllocator
{
public:
    // Defined by the client (pointer to a block, size of block)
    typedef void (*DUMPCALLBACK)(const void *, size_t);

    typedef void (*VALIDATECALLBACK)(const void *, size_t);

    // Predefined values for memory signatures
    static const unsigned char UNALLOCATED_PATTERN = 0xAA;
    static const unsigned char ALLOCATED_PATTERN = 0xBB;
    static const unsigned char FREED_PATTERN = 0xCC;
    static const unsigned char PAD_PATTERN = 0xDD;
    static const unsigned char ALIGN_PATTERN = 0xEE;

    // Creates the ObjectManager per the specified values
    // Throws an exception if the construction fails. (Memory allocation problem)
    ObjectAllocator(size_t ObjectSize, const OAConfig &config);
    // Destroys the ObjectManager (never throws)
    ~ObjectAllocator();

    // Deleted default functions
    ObjectAllocator(const ObjectAllocator& oa) = delete;
    ObjectAllocator& operator=(const ObjectAllocator& oa) = delete;

    // 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 = nullptr);

    // 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(void);
```

```cpp

    // Returns true if FreeEmptyPages and alignments are implemented
    static bool ImplementedExtraCredit(void);

    // Testing/Debugging/Statistic methods
    void SetDebugState(bool State);       // true=enable, false=disable
    const void *GetFreeList(void) const;  // returns a pointer to the internal free list
    const void *GetPageList(void) const;  // returns a pointer to the internal page list
    OAConfig GetConfig(void) const;       // returns the configuration parameters
    OAStats GetStats(void) const;         // returns the statistics for the allocator

private:
    OAStats stats;                          //! Stats of the object allocator
    OAConfig configuration;                 //! User defined configuration of the allocator
    size_t headerSize;                      //! The size of the header part of the page, post alig
    size_t dataSize;                        //! The size of each data part of the page (but not th
    size_t totalDataSize;                   //! The size of the sum of data part of the page, post
    GenericObject *PageList_ = nullptr;     //! The beginning of the list of pages
    GenericObject *FreeList_ = nullptr;     //! The beginning of the list of free objects

    void allocate_new_page_safe(GenericObject* &PageList);       // allocates another page of obje
    GenericObject* allocate_new_page(size_t pageSize);                  // Calls the actual new fo
    void put_on_freelist(GenericObject*Object); // puts Object onto the free list

    // Given a page address, removes all the objects in it from the freelist
    void removePageObjs_from_freelist(GenericObject* pageAddr);
    void freePage(GenericObject* temp);

    // For allocate
    void incrementStats();

    void freeHeader(GenericObject* Object, OAConfig::HBLOCK_TYPE headerType, bool ignoreThrow = fa
    // Given an addr, creates a handle at that point according to header type and config
    void updateHeader(GenericObject* Object, OAConfig::HBLOCK_TYPE headerType, const char* label =
    // Builds a header when initialized from page. No checks
    void buildBasicHeader(GenericObject* addr);
    // Called when we allocate. Builds the external header for user. No checks
    void buildExternalHeader(GenericObject* Object, const char* label);
    // Called when allocate. Builds the extended header
    void buildExtendedHeader(GenericObject* Object);
    // Check boundaries full check. Slower
    void check_boundary_full(unsigned char* addr) const;

    // Check padding

    bool isPaddingCorrect(unsigned char* paddingAddr, size_t size) const;
    bool checkData(GenericObject* objectdata, const unsigned char pattern) const;
    bool isInPage(GenericObject* pageAddr, unsigned char* addr) const;
    bool isPageEmpty(GenericObject* page) const;
    bool isObjectAllocated(GenericObject* object) const;

    // Given an address to an object, returns the address of the object's header file.
    unsigned char* toHeader(GenericObject* obj) const;
    unsigned char* toLeftPad(GenericObject* obj)const;
    unsigned char* toRightPad(GenericObject* obj)const;

    // Generice function to insert at the head of linked list
    void InsertHead(GenericObject* &head, GenericObject* node);
};

#endif
```