Atomic Memory Model

Atomic Memory Model

Example Implementation version 2.3 for C++ Reference Manual

2001 – 2012

Miroslav Bonchev Bonchev
miro@mbbsoftware.com

MBBSoftware

Contents

Open Source License

The MIT License

{your product} uses the Atomic Memory Model by Miroslav Bonchev Bonchev.

Utilization of the Atomic Memory Model

The Atomic Memory Model is used in number of large and small, commercial and academic projects including:

Act On File – an all-in-one software solution for file and data processing,
Audio Control – a popular volume control utility for Windows,
Mean Median Map – a solution for the well-known mathematical problem, and others.

If you use or have used the Atomic Memory Model in your project(s), please let us know so that we can reference your project in this column.

Synopsis

The Atomic Memory Model is a powerful technology which handles memory in a consistent, elegant, simple and highly effective way, greatly increasing the quality of code and speed of development. By abstracting the memory and representing it as an encapsulated entity, one can eliminate all of the issues arising around using memory in a digital computer system and make them intrinsically impossible.

Introduction

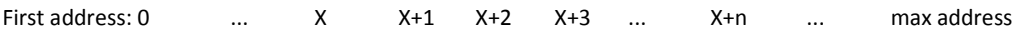
The memory in a computer system is a group of data placeholders of identical size, each of which is uniquely identified by a liner address. Although in some systems, this group might have a more complex nature of identification and accessing e.g. paging, segmentation, dual access, etc, for the purpose of this paper we will consider the flat memory model, where memory is an array of the same size, uniquely identified placeholders, ordered in a linear, sequential and contagious fashion from address zero to max address, as shown below:

First address: 0 ... X X+1 X+2 X+3 ... X+n ... max address

Most commonly, the size and address resolution of these placeholders is one byte, which makes each bit within the entire memory space uniquely identified and accessible via the address of the placeholder and its position within it.

The Problem

When allocated, memory is granted as chunks of single dimension sequence of address locations with size from 0 to n, where n ≤ available memory. Each allocation is a mere transfer of the ownership of an array part of a bigger or global memory array from the memory allocation system to the client (caller). This "transfer" is simply passing the address of the first allocated element (placeholder) to the caller, who automatically becomes the owner of the allocated memory block. By convention the caller assumes that the "received" memory is an array of consecutive, non-interrupted, uniquely identifiable entities at least as big as requested. The "allocated" memory is not moved, isolated, protected, transferred or anything else. The memory management system simply marks the allocated addresses (placeholders/bytes) as not owned by it and it does not use them until they are returned to it. The allocated memory has no type – it is simply an array of sequential addresses with a certain size. For example, if a caller requests memory with size 3 bytes it might receive address X, e.g.:




```
// "Give me 3 bytes of memory."
void* pMy3BytesMemory = malloc( 3 );
```

pMy3BytesMemory is now equal to X. The memory management system has internally marked bytes with addresses X, X+1 and X+2 as not owned by it and will not use them at all until they are returned to it via a corresponding call, e.g.:

```
// "Here are your 3 bytes of memory."
free( pMy3BytesMemory );
```

Using the C++ **new** and **delete** operators makes no difference. In essence, they merely cast the memory but do not change the situation in respect to the memory management. Thus the caller is responsible for:

1. Returning the memory when it is no longer needed.
2. Ensuring that it does not read or write outside of the memory block that was given to it.

When the caller fails to comply with these requirements it initiates one or more of the following erroneous conditions:

- Memory leaks – the caller has failed to return the memory when it is no longer needed. The reasons for this could be:
  - incomplete, imprecise or incorrect algorithm;
  - error in the algorithm implementation;
  - memory leaked during exception handling;
  - memory is released to wrong memory allocation system;
  - memory is released to wrong heap;

- Buffer overruns – the caller has failed to ensure that it does not read or write outside of the memory block that was given to it. The reasons for this could be:
  - incomplete, imprecise or incorrect algorithm;
  - error in the algorithm implementation;

- Program crash – in some systems and/or in some cases the above erroneous conditions may result in access violation, other type of application crash condition or a system exception, particularly when:
  - memory is released to wrong memory allocation system;
  - memory is released to wrong heap;
  - memory is read/written from/to location outside the read/write permitted page/segment;

- memory is read/written from/to location inside the read/write permitted page/segment, but outside of the allocated block, thus causing crash condition (immediately or at later time), e.g. division by zero;

Analysis of the problem

It could be argued that the erroneous conditions described occur simply because it is possible for them to occur, i.e. there is no intrinsic mechanism which somehow stops them from doing so. Any error in the logic of an algorithm or implementation may result in one or more memory related erroneous conditions which are not necessarily immediately obvious. The reasons for the absence of any such intrinsic protective mechanisms which permit these erroneous conditions to emerge lie in the physical and logical structure of digital computer memory and the way it is handled and presented to its customers, including:

1. Memory is allocated with a function call.

2. Memory is released with an explicit function call – failure to call the release function results in a memory leak.

3. Memory is released with a function call – releasing memory with incorrect function or incorrect parameter(s) results in a memory leak and/or application crash/system exception.

4. Memory is allocated without a type – a program has to cast the memory before using it, except if allocated with operator "new" (even though in this case the memory is cast in the body of the operator). An incorrect cast could easily result in read/write buffer overrun and certainly in some other error. There is no way to ensure correct casting.

5. Memory type is mutable - after a memory block is allocated and originally cast to a certain type, it can be cast again unlimited times each time to a different type and used as such a cast (type), regardless of whether it actually is of that type.

6. The pointer to the memory is exposed and used directly. Due to this, it can be incorrectly interpreted, as demonstrated in the previous points, and also be erroneously modified, e.g. by mistake resulting in a read/write buffer overrun and/or a memory leak.

7. Memory is allocated without physical boundaries. Since the allocated memory block is part of a global memory array with exactly the same access rights one could perform buffer overrun and be unaware of it (except if accessing outside of the global memory). This could make buffer

overrun a very difficult to detect erroneous condition resulting in range of different errors – from miscomputation to program crash.

8. Memory is allocated with very limited, nonfunctional and not particularly useful identity. Namely, memory identity is limited to and only to its starting address, which is only useful in "Is Null" or "Is the Same Address" logic and nothing beyond these. Certainly, one could use the starting address as an identification scheme and develop some additional mechanism such as a log file or linked list to track the lifetime of memory allocations, however any such mechanism is resource extensive, difficult to interpret and not necessarily able to give any answer about a memory leak. In practical terms, any such mechanism is useful for nothing more than to only point out that there are one or more memory leaks. One reason for the uselessness of the starting address as an identification scheme is that it does not distinguish between memory block and memory allocation - which means lack of abstraction; a memory block could be expanded/shrank, cast, transferred from one part of the application to another, etc and still have the same starting address.

From the above, we conclude that no abstraction whatsoever is applied to memory. Memory is interpreted and handled as amorphous space without any abstraction, while representing (interpreted as) abstract or semi-abstract entities. Where abstract entity is a type (built-in or user-defined) and semi-abstract is an array from a certain type. This is the most fundamental inconsistency found when examining the problem. This also is the most fundamental inconsistency possible for this Universe of Discourse, therefore this contradiction is the most fundamental reason for the problems associated with use of memory in a digital computer system and it must be removed first.

The Solution

The fundamental reason for problems with use of memory in a digital computer system was identified as the contradiction of memory being handled as amorphous space without any abstraction, while representing abstract entities. The objective now therefore is to remove the identified contradiction. This can be achieved by:

- either removing any abstraction when using memory, which means to use only types that are native to the physical data placeholder (8, 16, 32, 64 bit words). This of course does not solve the problem, and further reverts a high level language to assembly and thus defeats the purpose of using high level languages. For this reason we will not consider this possibility any further.

- abstract the memory handling so that the added abstraction layer(s) correctly reflect the Universe of Discourse (as explained in the previous sections) and appropriately neutralize the possible erroneous conditions.

Atomic Memory Model

The Atomic Memory Model is a notion in which the memory exists only as an Entity. Memory never exists as a piece of space, instead always as an encapsulated self-sustained entity, where one or more abstraction layers have the task to successfully address and resolve all erroneous conditions described earlier.

In the Atomic Memory Model, memory does not exist as memory at any abstract level; instead it exists only as Memory Atoms of a certain type. Within the atom, "in its nucleus", where this abstraction is "undressed", the atom uses memory in the traditional fashion. However, for an external referrer the memory atom is an encapsulated entity, which is available to access only via a set of exposed interfaces.

When a memory atom is created, it allocates the correct amount of memory. When it is destroyed, the memory which it holds is released using the appropriate memory releasing function in its destructor. Since the memory which it owns is always accessed via methods, one cannot gain access to that memory directly: therefore by using defensive code in the methods of the memory atoms, it can be guaranteed that read/write buffer overruns and other erroneous conditions will not occur. By implementing defensive code in the methods of the memory atom, for all relevant erroneous conditions applying to a particular method (e.g. read/write overrun, system out of memory, and suchlike), it can be guaranteed that the memory atom will signal the erroneous condition via exception(s) (or some other appropriate way e.g. ASSERT, returning false, and so on) and will not commit any illegal operations. This guarantees both an extremely stable system (with respect to memory handling), while the code using the Atomic Memory Model is most concise and tidy.

Memory Atom

Definition: *Type* in the Atomic Memory Model consists of three properties **Class**, **Semantics** and **Origin**.

1. **Class** – specialization and granularity. The "class" property specializes the Memory Atom – it specifies the type of the contained elements. The "class" property "granulates" the Memory Atom. For example, specializing a Memory Atom as a Byte means that the Memory Atom will hold Bytes. The class property defines the dimension of the type.

2. **Semantics** – specific (including polymorphic) behavior. The "semantics" property determines the identity of the type (not of an object), i.e. its nature, behavior, characteristics, interfaces etc. For example, a Memory Atom defined as thread safe determines semantics of thread safe access to the contained memory; "Secure" Memory Atom determines semantics that memory content will be erased before memory is released, etc.

3. **Origin** – underlying memory system ultimately owning the memory. The "origin" property specifies from where or how memory will be allocated and respectively released. For example, memory may be allocated from the process heap, other heap, some application pool, COM allocator, C library allocator, or it might not be allocated/released at all, e.g. shell memory, etc.

Definition: Memory Atom is an instance of a particular Memory Atom Type.

Every memory atom is an object which holds memory of type Class, which has particular behavior as determined by its Semantics, and which memory belongs to a particular pool defined by its Origin.

When instantiating a memory atom from a certain type, we declare the three properties class, semantics and origin, in addition to the number of items that the memory atom will contain. The contained items (elements) are from the class of the memory atom, for example a memory atom from class DWORD contains number of DWORDs, and a memory atom from class bool contains a number of bool items. To refer to an item held by a memory atom we use an auxiliary entity called **Memory Unit**.

Memory Unit

Definition: Memory Unit is an integer number classed with memory Type::Class which is the only countable entity used in association with, and in reference to, items held by a memory atom.

Memory Unit is used with Memory Atoms to count, add, subtract, index, etc the elements (items) held in a memory atom. The Memory Unit is always from the same class as the class of the Memory Atom with which it is used. A memory atom can be requested to allocate, reallocate, de-allocate, access, etc items (elements) only using memory units. For example, a memory atom of class DWORD could be requested to allocate a certain number of DWORDs only using a memory unit which must be of class DWORD. After the allocation the memory atom will hold that many DWORDs.

Memory units are introduced and bound to memory atoms to prevent the array of errors which would occur if unbound memory units were to be used when referring to their items. For example, if using integer Allocate(int iUnits) instead of the specialized type Allocate(Unit< class >), one could easily mistakenly pass any value for iUnits, meaning for example, the same amount of units but in bytes instead of DWORDs. In this regard the class property of a memory atom and memory units is its dimension.

A general integer type is not appropriate for use when referring to items in a memory atom as this would strip out the abstraction and would defeat our purpose to present the memory as an encapsulated self-sustained entity. Further to this when using memory atoms in heterogeneous fashion, for example serializing memory atoms from different classes into a stream, Unit< class > gives intrinsic means to determine the size of the contained memory in bytes or other arbitrary classes. If using a generic integer value when referring to the items of memory atom one will have to compute and convert results, which again removes the abstraction and defeats the objectives. If Unit< class > is not the only way to refer to memory atom items there will be an endless confusion as to when an integer is used as memory unit and when as an absolute integer. In order to avoid any confusion about what class and how many memory units are allocated, every reference to items of a memory atom must be via a specialized memory unit type e.g. Unit< class > with class matching the class of the memory atom and not using a dimensionless type such as signed/unsigned integer, long, etc.

Rule: Memory Units from the class of a Memory Atom are the only way to refer to its items.

Space of Existence

Definition: Space of Existence of a Memory Atom is the stack frame where it is instantiated, or a stack based object holding the memory atom which is responsible for its deletion.

The space of existence can be a simple scope (stack frame) where a Memory Atom is created, lives and on the exit of which is destroyed or it could be not affected directly by the stack frames, but determined by the logic of execution. For example, beside the stack frame itself, the space of existence of memory atoms could be a container, e.g. linked list declared in a relatively outer stack frame which holds memory atoms that are added and destroyed as appropriate. When the stack frame directly or indirectly holding memory atoms is released (normally or abnormally (due exception)) the memory atoms (and the memory contained in them) are released directly or indirectly by the destructor of the container holding them. It is essential that the space of existence is always directly or indirectly engaged with a stack frame so the appropriate destructors are called when unwinding the containing stack frame.

Rule: Memory Atoms are always instantiated on a stack frame or are somehow contained by a stack object, which is responsible for their destruction.

Atomic Memory Model Rules

The Atomic Memory Model rules are:

1. Memory Atoms have a type, constituted from three properties: class, semantics and origin.

2. Memory Atoms do not allow unprotected access to the memory which they hold.

3. Memory Atoms use defensive code in their methods covering all erroneous conditions possible for the method.

4. Memory Units from the class of a Memory Atom are the only way to refer to its items.

5. Memory Atoms are always instantiated on a stack frame or are somehow contained by a stack object, which is responsible for their destruction.

Note that rule 2 may be somewhat relaxed in implementations where the Memory Atoms are interacting with software not designed to accept Memory Atoms, e.g. to allow passing of memory owned by memory atoms to a native OS API functions accepting void*, BYTE*, etc.

Additional Functionality and Reuse of Code

In addition to resolving the erroneous conditions when using computer memory, the Atomic Memory Model could increase its usefulness by defining a common interface for all memory atom types and defining them as a family of polymorphic types. This also significantly simplifies the development and maintenance of different types of memory atoms. For example, adding methods to serializing the content of memory atoms, reading/storing resources, transfer, comparison, casting and other operations.

The Atomic Memory Model allows easy identification of particular instances of memory allocations. This is desired by many memory auditing schemes, however it is practically unused in the Atomic Memory Model as memory does not leak when the model is implemented and used properly.

Conclusion

The benefits of using the Atomic Memory Model are:

- Resolved problems and issues – the Atomic Memory Model successfully addresses all known issues including:
  - Memory leaks;
  - Buffer overruns;
  - Program crash – due one or more of the above.

- Application Performance:
  - Speed:
    - Insignificant performance overhead due to calls to methods for construction.
    - Performance improvement, since function pre-calls querying for the size of the required memory for are no longer needed and any such calls are removed, since the atom can allocate all memory that is required.
  - Memory use:
    - Increase of the size of the compiled code due to the bodies of the methods of the memory atoms – especially when in-lining.
    - There might be an insignificant increase of the compiled code size due the bodies of the Memory Atoms.
  - Robustness – a system using the Atomic Memory Model is extremely robust with respect to memory handling, especially when the memory atoms use defensive code within their methods.

- Development:

- The Atomic Memory Model guarantees concise and tidy program code related to memory allocation, access (use) and de-allocation.
- Development speed is increased multiple times. Code is simplified multiple times. Multiple additional benefits might appear depending on the particular implementation of the Atomic Memory Model, e.g. from reuse of code.
- Debugging – once the Memory Atoms have been developed and fine-tuned so as to be free from errors, memory problems do not occur.
- Memory atomization, i.e. encapsulating it in objects allows for proper instance identification. A few conditionally-compiled lines of code or a conditionally-compiled parent, adding some form of memory atom instance identification, e.g. count, name, etc is all that is necessary. This would be useful when for some reason a developer needs to track the lifecycle of a particular memory atom. In practice, this is very rarely used as memory problems are nonexistent.

- Architecture – The atomic Memory Model provides a consistent, natural and architecturally sound way to handle memory at no cost at all, as opposed to garbage collection which is unnatural, resource wasteful, architecturally unsound, and developer demoralizing approach.

Implementations

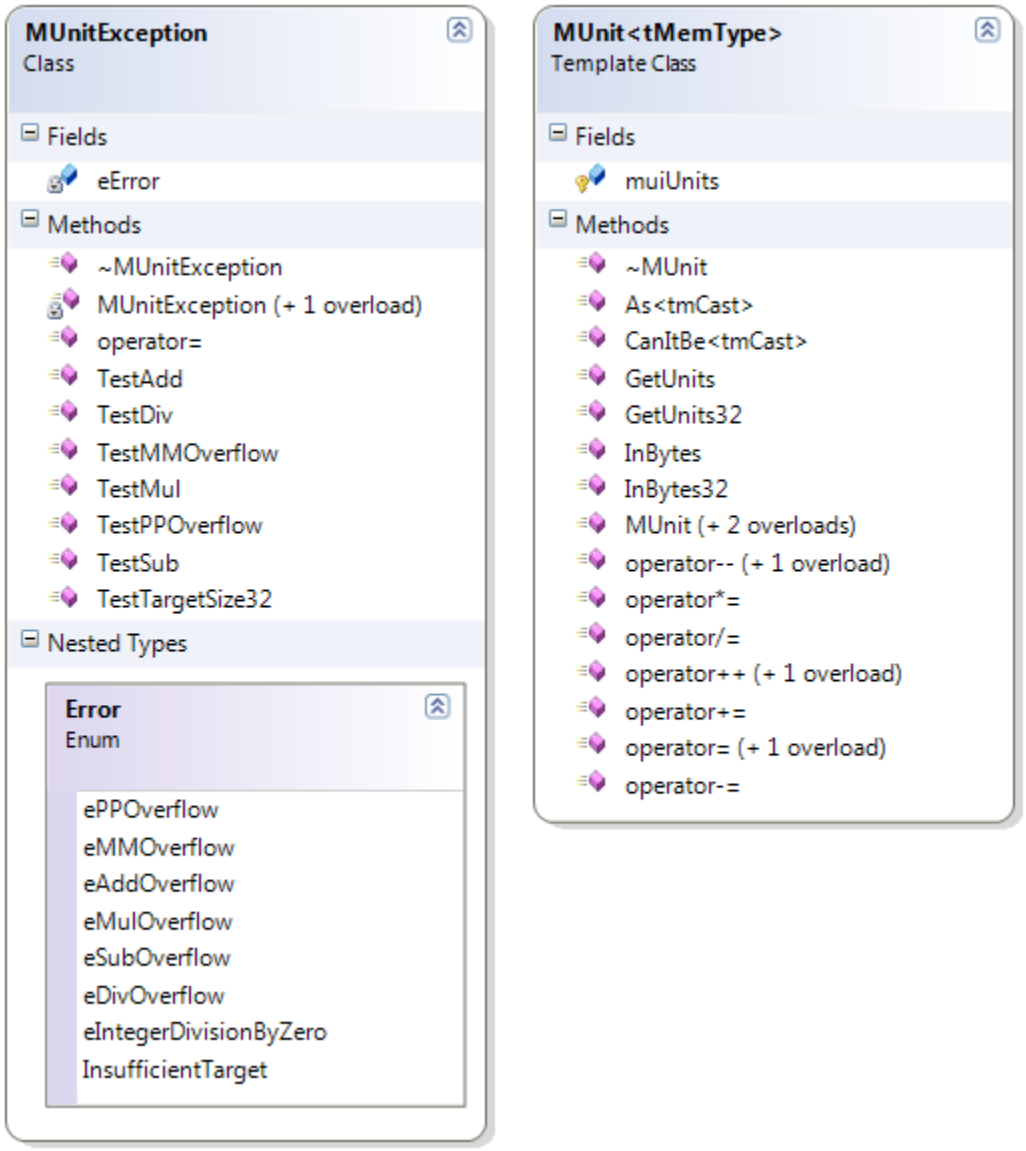
The current implementation is the second and using C++ as strongly typed Object Oriented language with strong type specialization capabilities which are used to guarantee compliance with the model for as much as possible for a non-native implementation achieved by using templates, inheritance and polymorphism. The first implementation had some architectural flaws which are corrected in the second implementation. Using templates guarantees that the class property will be exactly "as declared". The inheritance and polymorphism ensure that some of the semantics property at least in the inherited part will be put into place as declared. There is no guarantee that these will be sufficient for the integrity of the type – however once ensured, it is so. A better approach would be a native support of the model by the language itself. This implementation always throws exceptions in case of errors with exception of a few cases where errors can also be returned via out parameters. Using exception handling ensures consistent and tidy code.

Memory Units and Related Classes

A Memory Unit class is defined as per the Atomic Memory Model definition, and is used for numeric references associated with memory atoms. The Memory Unit throws MUnitException if a memory unit error is encountered, e.g. division by zero, requesting a memory item outside of the memory atom boundary or other. The MUnit exception is defined outside of the MUnit class (not nested, i.e.

namespaced) because if it was nested there would be need from a separate catch block such as catch(MUnit< tMemType >::MUnitException) for every tMemType that is being used in the try block. Use catch(const MUnitException e) to capture MUnitExceptions.

Memory Unit and Memory-Unite-Exception class diagrams.



MUnit has all arithmetic operators overloaded, as well as all numeric comparison operators and several conversion functions. MUnit is 64 bit for 64 bit compilations and 32 bit for 32 bit compilations. The MUI type is a size_t typedef. The only nontrivial methods are the casting operations:

```
template< class tmCast > bool CanItBe() const throw();
- Returns true if the casting conversion is possible and false otherwise.
```

Use:

```
MUnit< BYTE > mu( 4 );
bool bCanConverttoDword = mu.CanItBe< DWORD >();
```

```
template< class tmCast > MUnit< tmCast > As() const throw( MUnitException );
- Returns the number of memory units of the specified class, which are contained in a memory chunk of another class. The function throws exception if the conversion yields a fraction.
```

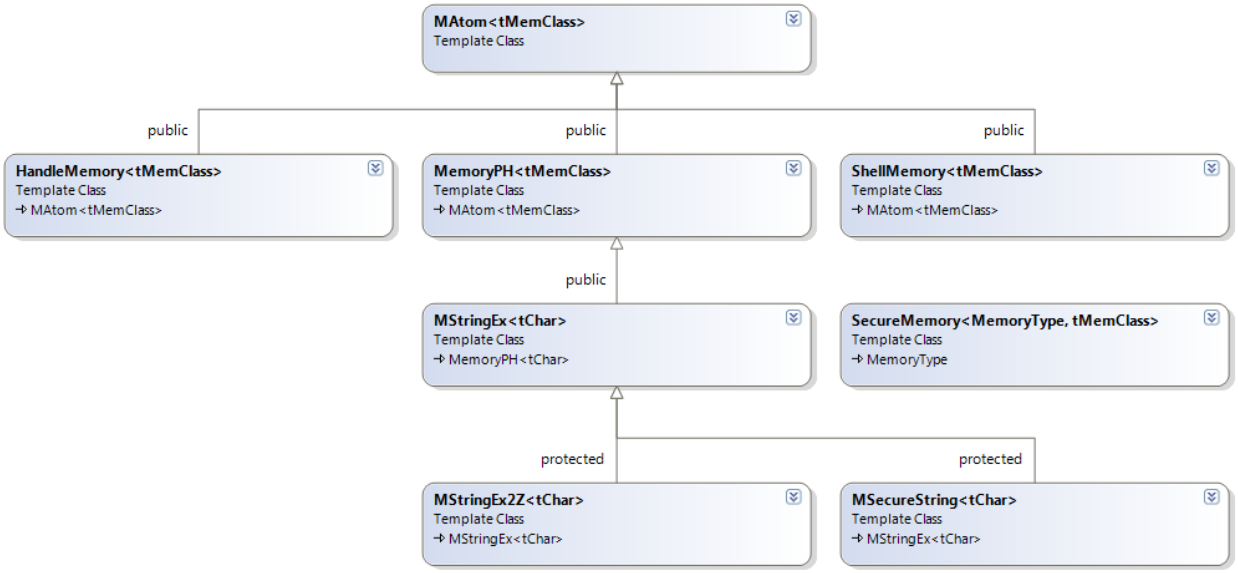
Use:

```
MUnit< DWORD > m1( MUnit< __int64 >( 256 ).As< DWORD >() );
```

Memory Atom and Related Classes

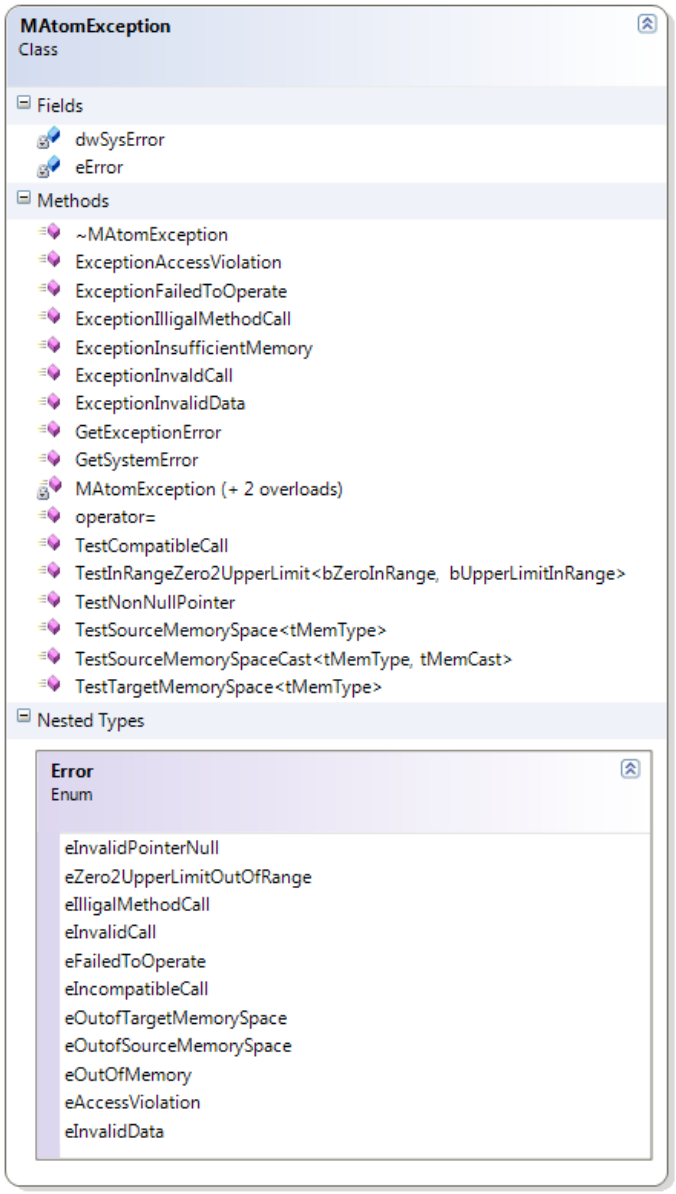
MAtom defines common interface for all memory atoms from this implementation.

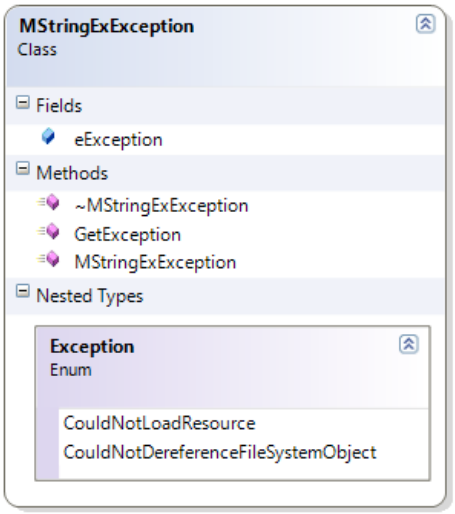
Class Diagram for the memory atoms of example implementation two:



The memory atoms in this implementation throw MAtomException. Note that the string classes also throw MStringExException. The MAtom exception is defined outside of the MAtom class to avoid an unnecessary exception handler for every specialization of memory atoms used in the try block e.g. catch(MAtom< tMemType >::MAtomException). Use catch(const MAtomException e) to capture MAtomExceptions. Use MAtomException::Error MAtomException::GetExceptionError() const to retrieve the exception error for bad calls. Use DWORD MAtomException::GetSystemError() const to retrieve system errors.

Class diagrams of MAtomException and MStringExException.

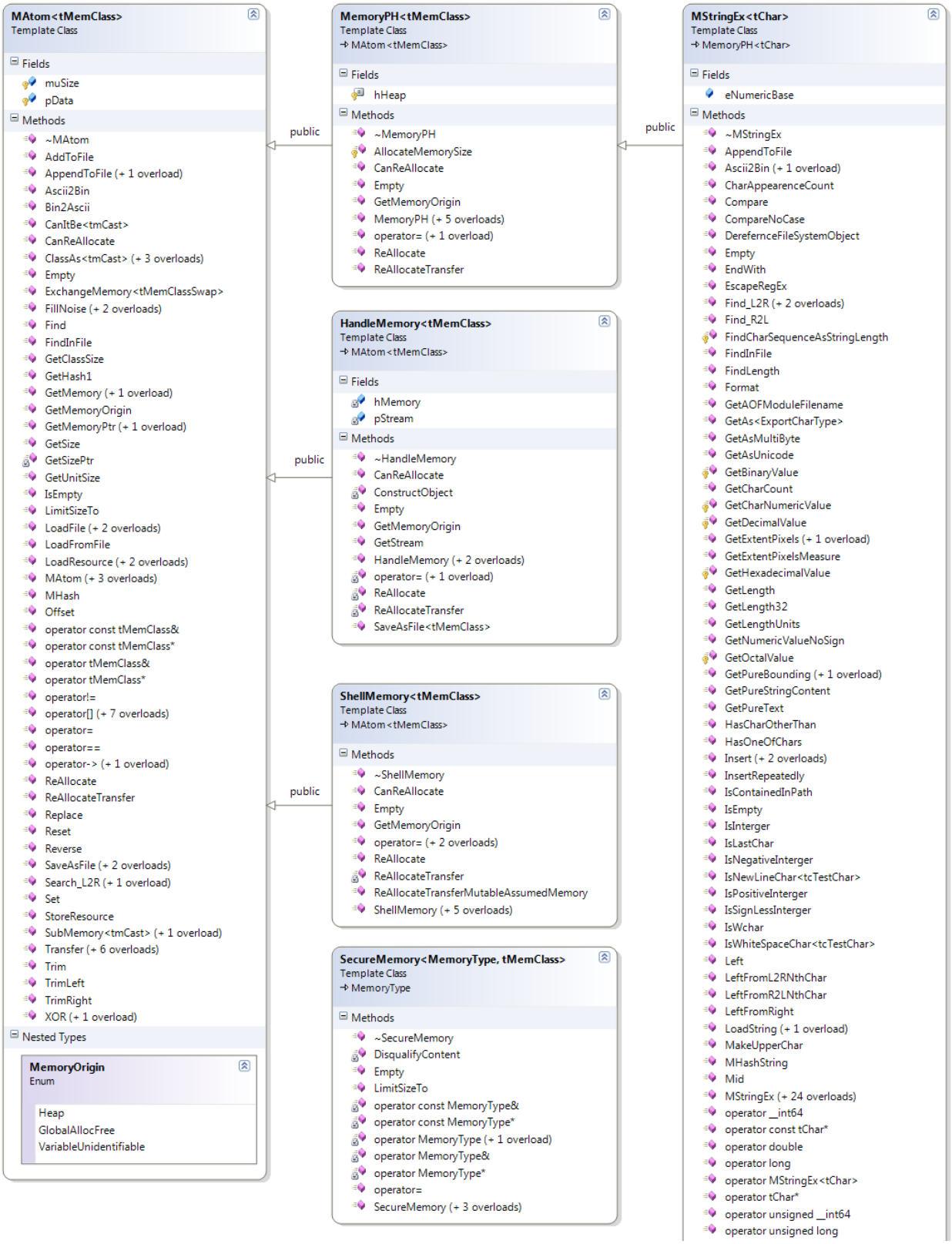


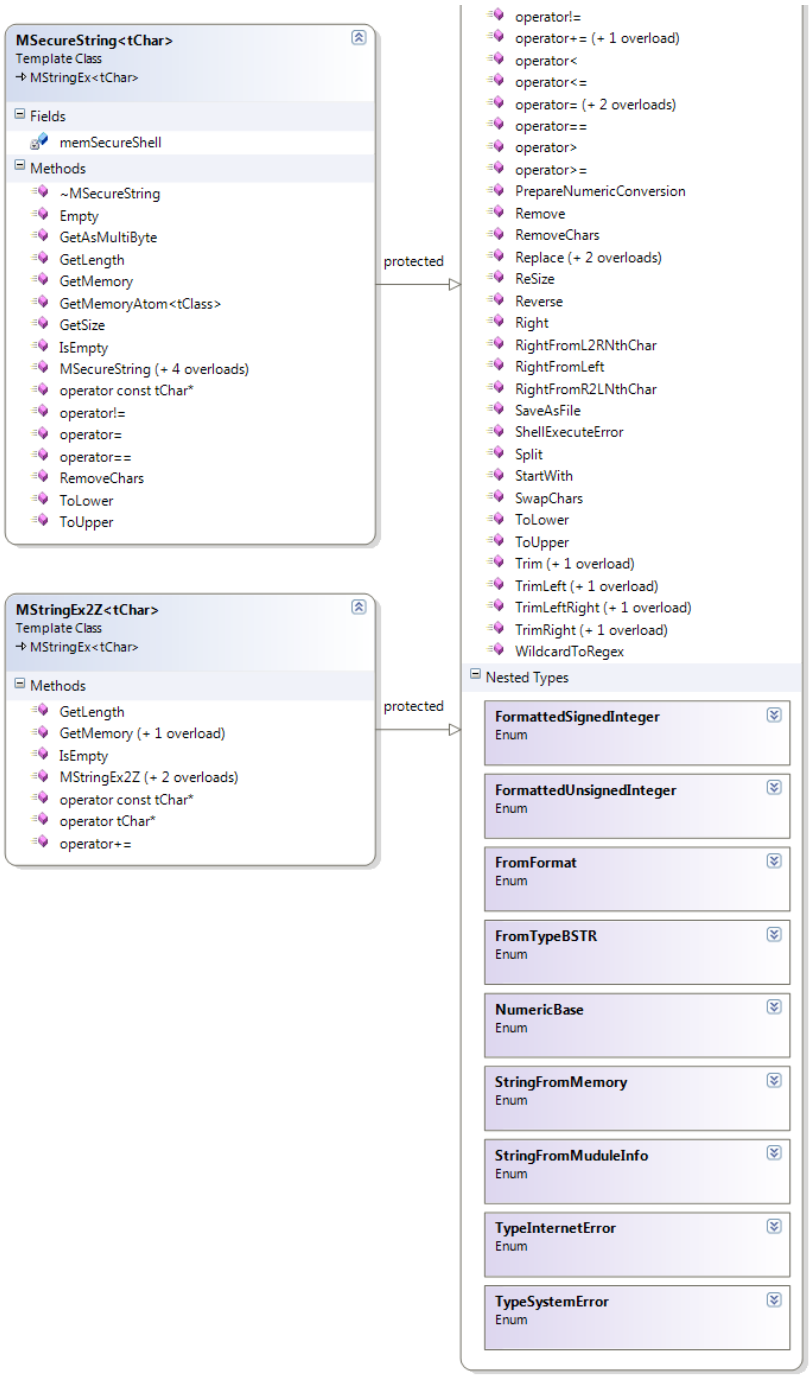


MAtom< class tMemType >

The fundamental abstract class defining common interface for all memory atoms in implementation two is `template< class tMemType > class MAtom;` The current class diagram of the hierarchy follows.

Class Diagrams of MAtom, MemoryPH, HandleMemory, Shellmemory, SecureMemory & various strings.





The MAtom class can and will be further extended by adding more operations such as unit-wise operations and others.

Method - operator=

```
Replace the content of the memory atom with that of the parameter.

virtual MAtom< tMemClass >& operator=( typename const MAtom< tMemClass >& maSource ) = 0;
```

Method - GetMemoryOrigin

Pure polymorphic method. Returns the Memory Origin.

```
virtual MemoryOrigin GetMemoryOrigin() const = 0;
```

Method - CanReAllocate

Pure polymorphic method. Returns true if the memory atom is able to (re)allocate memory, and false if not. For example, memory atoms with memory from a heap are typically able to (re)allocate, whilst shell memory atoms cannot.

```
virtual bool CanReAllocate()     const throw() = 0;
```

Method - IsEmpty

Pure polymorphic method. Returns true if the memory atom is empty, and false otherwise.

```
virtual bool IsEmpty() const throw( MAtomException );
```

Method - Empty

Pure polymorphic method. Empties the memory atom from any contained memory.

```
virtual MAtom< tMemType >& Empty() throw( MAtomException ) = 0;
```

Method - GetSize

Returns the size of the memory contained by the atom memory it memory units.

```
MUnit< tMemType > GetSize() const throw();
```

Method - GetUnitSize

Returns the size of the memory unit specializing the atom, i.e. size of the Type::Class property.

```
static size_t GetUnitSize() throw();
```

Method - GetClassSize

Returns the size of the class specializing the memory atom.

```
static size_t GetClassSize() throw();
```

Method - operator==

Compare two memory atoms to see if they have the same content byte by byte.
Returns true if the atoms are identical and false otherwise.

```
virtual bool operator==( const MAtom< tMemType >& mtAtom ) const throw();
```

Method - operator!=

Compare two memory atoms to see if they have different content byte by byte.
Returns true if the atoms have different content and false otherwise.

```
virtual bool operator!=( const MAtom< tMemType >& mtAtom ) const throw();
```

Methods Family – GetMemory - Get Direct Access to Memory

Returns direct access to the memory. This is relaxation of the Atomic Memory Model rule 2 allowing the use of memory atoms in non-Atomic Memory Model compliant environment.

```
const tMemType*   GetMemory()    const throw() { return( pData  ); }
      tMemType*   GetMemory()          throw() { return( pData  ); }
const tMemType**  GetMemoryPtr() const throw() { return( &pData ); }
      tMemType**  GetMemoryPtr()       throw() { return( &pData ); }
```

Methods Family - operator->

Returns direct access to the memory. This is relaxation of the Atomic Memory Model rule 2 allowing the use of memory atoms in non Atomic Memory Model compliant environment.

```
const tMemType* operator->() const throw() { return( pData ); }
      tMemType* operator->()       throw() { return( pData ); }
```

Methods Family - operator & - Get Direct Access to Memory

Returns direct access to the memory. This is relaxation of the Atomic Memory Model rule 2 allowing the use of memory atoms in non-Atomic Memory Model compliant environment.

```
operator       tMemType& ()         throw() { return( *pData ); }
operator const tMemType& ()   const throw() { return( *pData ); }
operator       tMemType* ()         throw() { return(  pData ); }
operator const tMemType* ()   const throw() { return(  pData ); }
```

Methods Family - operator[]

Indexer returning reference to a unit contained in the memory atom.

```
tMemType& operator[]( const int iIndex ) throw( MAtomException );
const tMemType& operator[]( const int iIndex ) const throw( MAtomException );

tMemType& operator[]( const unsigned long iIndex ) throw( MAtomException );
const tMemType& operator[]( const unsigned long iIndex ) const throw( MAtomException );

tMemType& operator[]( const MUI muiIndex ) throw( MAtomException );

const tMemType& operator[]( const MUI muiIndex ) const throw( MAtomException );

tMemType& operator[]( typename const MUnit< tMemType >& muIndex ) throw( MAtomException );
const tMemType& operator[]( typename const MUnit< tMemType >& muIndex ) const throw( MAtomException );
```

Methods Family - SubMemory

Wraps number of units inside a memory atom and returns a shell memory atom containing them.

const MUnit< tMemType >& muFrom – starting memory unit, zero based index.

```
const MUnit< tmCast >& muSize   - number of units to be included in the new memory atom

Return ShellMemory atom referencing the requested number of memory units in the source (this) memory atom
starting from the appointed position.

template< class tmCast > ShellMemory< tmCast > SubMemory( typename const MUnit< tMemType >& muFrom,
typename const MUnit< tmCast >& muSize ) const throw( MAtomException );

template< class tmCast > ShellMemory< tmCast > SubMemory( typename const MUnit< tMemType >& muFrom,
typename const MUnit< tmCast >& muSize ) throw( MAtomException );
```

Automatic Casting Operations of non-compatible classes, e.g. cast [MAtom Derivate]< BYTE > to [MAtom Derivate]< DWORD >.

These methods returns an instance of ShellMemory because it is impossible to return re-specialized MAtom referencing "this" without encountering serious problems. Returning a re-specialized MAtom would require an expression similar to return(*(MAtom< tmCast >*)this);, which is clearly wrong since the muSize will be the same value but in a different class of units. For example, Memory< BYTE >(MUnit< BYTE >(10)) classed to DWORD will appear as MAtom< DWORD >::pData pointing to the same data, and having the same amount of units = 10 but of type DWORD, so the object would incorrectly claim total size of 10 * sizeof(DWORD) instead of 10 * sizeof(BYTE).

In the first Atomic Memory Model implementation, the memory size was held in bytes, and this conversion was possible. However in this implementation (2) such immediate casting is impossible for the above reason as well as for reason that while virtual functions will be called correctly on the return object, non-virtual functions will be called inconsistently. Consider the following example:

```
class X                     | class A : public X          | class B : public X
{                           | {                           | {
public:                     | public:                     | public:
   virtual void f1() = 0;   |    virtual void f1()        |    virtual void f1()
                            |    {                        |    {
                            |       printf( "A::f1()\n" );|       printf( "B::f1()\n" );
                            |    }                        |    }
                            |                             |
   void f2()                |    void f2()                |    void f2()
   {                        |    {                        |    {
      printf( "X::f2()" );  |       printf( "A::f2()\n" );|       printf( "B::f2()\n" ); }
   }                        |    }                        |    }
};                          | };                          | };
```

```
void main()
{
   A  a;
   B& b( *(B*)&a ); - effectively what AppearAs() in Atomic Memory Model Phase One does
   a.f1();
   a.f2();
   b.f1();
   b.f2();
}

Results:    A::f1()
            A::f2()
            A::f1() --same object with different "identity" - inconsistent behavior.
            B::f2() -/
```

In order to maintain consistency, the Atomic Memory Model implementation 2 uses appropriately constructed and specialized shell memory objects which are returned by value for any automatic casting operations.

Methods Family - ClassAs – entire memory atom

Cast (specialize) the memory atom with another class. Returns a shell memory atom classed (specialized) with the new cast type.

```
template< class tmCast > ShellMemory< tmCast > ClassAs() throw( MAtomException );
template< class tmCast > const ShellMemory< tmCast > ClassAs() const throw( MAtomException );
```

Methods Family - ClassAs – portion from memory atom

Cast (specialize) the memory atom with another class.

const MUnit< tMemType > muOffsetThis - the beginning of the memory referenced by the returned shell memory atom in the units of the referenced (this) memory atom.
const MUnit< tmCast > muUnitsCast - the size of the memory referenced in the returned shell memory atom in memory units of the returned shell memory atom.

Returns a shell memory atom classed (specialized) with the new cast type.

```
template< class tmCast > ShellMemory< tmCast > ClassAs( typename const MUnit< tMemType > muOffsetThis,
typename const MUnit< tmCast > muUnitsCast ) throw( MAtomException );
template< class tmCast > const ShellMemory< tmCast > ClassAs( typename const MUnit< tMemType >
muOffsetThis, typename const MUnit< tmCast > muUnitsCast ) const throw( MAtomException );
```

Method - CanItBe

Test if a memory atom as a whole can be casted directly with another memory class.

```
template< class tmCast > bool CanItBe() const throw();
```

Method - ReAllocate

Pure polymorphic method. Release the contained memory and allocate the new requested amount. Not all types of memory atoms are able to re-allocate memory.

```
virtual MAtom< tMemType >& ReAllocate( typename const MUnit< tMemType >& muUnits ) throw( MAtomException )
= 0;
```

Method - Transfer A

Copy part or the entire memory content of the passed memory atom to the memory contained in this memory atom. Memory is copied in a byte by byte fashion.

MUnit< tMemType >& muOffsetThis - indicates the offset from where the new data will be written in units of (this) to the target memory atom.
const bool bUpdateOffsetThis - indicates whether muOffsetThis will be updated after successful data transfer with the number of copied memory units.
const MAtom< tMemType >& maSource - source memory atom.
MUnit< tMemType >& muOffsetSource - indicates the offset from where to begin the data transfer in the source memory atom in its own units.
const bool bUpdateOffsetSource - indicates whether muOffsetSource will be updated after successful data transfer with the number of copied memory units.
const MUnit< tMemType >& muTransfer - indicates the number of memory units to be copied from the source memory atom to this (target) one.

There must be enough memory in both the source and target memory atoms for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename MUnit< tMemType >& muOffsetThis,
                                     const bool bUpdateOffsetThis,
                                     typename const MAtom< tMemType >& maSource,
```

```
                                    typename MUnit< tMemType >& muOffsetSource,
                                    const bool bUpdateOffsetSource,
                                    typename const MUnit< tMemType >& muTransfer ) throw( MAtomException
);
```

Method - Transfer B

Sequentially copy part or the entire memory content of the passed memory atom to the memory contained in this memory atom starting at the target's beginning.

MAtom< tMemType >& maSource - source memory atom.
MUnit< tMemType >& muOffsetSource - indicates the offset from where to begin the data transfer in the source memory atom in its own units.
const bool bUpdateOffsetSource - indicates whether muOffsetSource will be updated after successful data transfer with the number of copied memory units.
const MUnit< tMemType >& muTransfer - indicates the number of memory units to be copied from the source memory atom to this (target) one.

There must be enough memory in both the source and target memory atoms for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename const MAtom< tMemType >& maSource,
                                    typename MUnit< tMemType >& muOffsetSource,
                                    const bool bUpdateOffsetSource,
                                    typename const MUnit< tMemType >& muTransfer ) throw( MAtomException
);
```

Method - Transfer C

Sequentially copy part or the entire memory content of the passed memory atom to the memory contained in this memory atom starting from the sources's beginning.

MUnit< tMemType >& muOffsetThis - indicates the offset from where the new data will be written in units of (this) to the target memory atom.
const bool bUpdateOffsetThis - indicates whether muOffsetThis will be updated after successful data transfer with the number of copied memory units.
const MAtom< tMemType >& maSource - source memory atom.
const MUnit< tMemType >& muTransfer - indicates the number of memory units to be copied from the source memory atom to this (target) one.

There must be enough memory in both the source and target memory atoms for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename MUnit< tMemType >& muOffsetThis,
                                    const bool bUpdateOffsetThis,
                                    typename const MAtom< tMemType >& maSource,
                                    typename const MUnit< tMemType >& muTransfer ) throw( MAtomException
);
```

Method - Transfer D

Sequentially copy the entire memory content of the passed memory atom starting at its beginning, to the memory contained by the target (this).

MUnit< tMemType >& muOffsetThis - indicates the offset from where the new data will be written in units of (this) to the target memory atom.
const MAtom< tMemType >& maSource - source memory atom.

There must be enough memory in the target memory atom for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename MUnit< tMemType > muOffsetThis, typename const MAtom<
tMemType >& maSource ) throw( MAtomException );
```

Method - Transfer E

Sequentially copy the entire memory content of the passed memory atom starting at its beginning, to the memory contained by the target (this).

MUnit< tMemType >& muOffsetThis - indicates the offset from where the new data will be written in units of (this) to the target memory atom.
const bool bUpdateOffsetThis - indicates whether muOffsetThis will be updated after successful data transfer with the number of copied memory units.
const MAtom< tMemType >& maSource - source memory atom.

There must be enough memory in the target memory atom for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename MUnit< tMemType >& muOffsetThis, const bool bUpdateOffsetThis, typename const MAtom< tMemType >& maSource ) throw( MAtomException );
```

Method - Transfer F

Sequentially copy the entire memory content of the passed memory atom to the memory contained by the target (this), starting at their beginnings.

ShellMemory< tMemType >& maSource - source memory atom.

Both memory atoms must have the same size in order for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename const MAtom< tMemType >& maSource ) throw( MAtomException );
```

Method - Transfer G

Sequentially copy part or the entire memory content of the passed memory atom to the memory contained by the target (this) memory atom.

const MUnit< tMemType > muOffsetThis - indicates the offset from where the new data will be written in units of (this) to the target memory atom.
const MAtom< tMemType >& maSource - source memory atom.
const MUnit< tMemType > muOffsetSource - indicates the offset from where to begin the data transfer in the source memory atom in its own units.
const MUnit< tMemType > muTransfer - indicates the number of memory units to be copied from the source memory atom to this (target) one.
There must be enough memory in both the source and target memory atoms for the request to be completed. The return value is reference to this (target) memory atom.

```
virtual MAtom< tMemType >& Transfer( typename const MUnit< tMemType > muOffsetThis,
                                     typename const MAtom< tMemType >& maSource,
                                     typename const MUnit< tMemType > muOffsetSource,
                                     typename const MUnit< tMemType > muTransfer ) throw( MAtomException );
```

Method - ReAllocateTransfer

Pure polymorphic method. Release contained memory, allocate new memory with the size of the passed memory atom and copy its content to the newly allocated memory of the target (this). Not all types of memory atoms are able to re-allocate memory and thus some types of memory atoms may always throw exception from this method.

const MAtom< tMemType >& maSource – size of the new memory to allocate in memory units.

Returns reference to the modified this memory atom.

```
virtual MAtom< tMemType >& ReAllocateTransfer( typename const MAtom< tMemType >& maSource ) throw( MAtomException ) = 0;
```

Methods Family - LoadResource

Load resource in the memory atom from the hDefaultTextResourceModule module. The HMODULE hDefaultTextResourceModule must be defined in a cpp file and loaded with a valid module handle (or set to null to load the resource from the current executable module) from which the resource will be loaded.

```
const TCHAR* strModule - path to the exe/dll containing the resource to be loaded.
const HMODULE hModule  - handle to the module containing the resource to be loaded.
const TCHAR* strType   - resource type.
const DWORD dwName     - resource name.
```

The function returns reference to this with the loaded resource. The function throws an exception if an invalid call is made, and returns an empty object if it fails to load the resource.

```
virtual MAtom< tMemType >& LoadResource( const TCHAR* strType, const DWORD dwName ) throw( MAtomException );

virtual MAtom< tMemType >& LoadResource( const TCHAR* strModule, const TCHAR* strType, const DWORD dwName ) throw( MAtomException );

virtual MAtom< tMemType >& LoadResource( const HMODULE hModule, const TCHAR* strType, const DWORD dwName ) throw( MAtomException );
```

Method - LoadResource

Store the memory atom content as a resource in a module. The calling process must have sufficient rights to write in the module, and the module must be available for writing.

```
LPCTSTR strModule                  - filename and path to the module where the content of the memory atom will be stored as a resource.
LPCTSTR strType                    - type of the stored resource.
const DWORD dwName                 - name of the stored resource.
const USHORT usLanguage            - language code for the stored resource, e.g. LANG_ENGLISH.
const USHORT usSubLanguage         - sub-language code for the stored resource, e.g. SUBLANG_ENGLISH_US.
const BOOL bDeleteExistingResources - "true" deletes any existing resource with the specified name; "false" leaves existing resources intact unless they are overwritten.
```

The function returns true if the request is processed successfully and false in case of failure. Use GetLastError() or string(string::eGetSystemError) to review the error occurred.

```
virtual bool StoreResource( LPCTSTR strModule, LPCTSTR strType, const DWORD dwName, const USHORT usLanguage, const USHORT usSubLanguage, const BOOL bDeleteExistingResources ) const;
```

Method - Offset

Move the contained memory up/down a memory stream. This method is meaningful only for memory types which represent a memory segment within a stream, such as the ShellMemory memory type.

```
const MUnit< tMemType >& muOffset - memory units to move the beginning of the "window" in the stream.
const bool bPositiveDirection     - true makes the beginning of the "window" move towards the end of the stream, and false towards the beginning.
const bool bConstantSize          - true maintains the size of the memory contained by the memory atom constant, while false shrinks/expands the contained memory with muOffset.
```

The function returns a reference to the modified memory atom ("window") in the other stream. The function throws exception in case of invalid call as usual.

```
virtual MAtom< tMemType >& Offset( typename const MUnit< tMemType >& muOffset, const bool bPositiveDirection, const bool bConstantSize ) throw( MAtomException );
```

Method - LimitSizeTo

Decreases the number of the contained memory units to the muSize passed as parameter.
The function returns a reference to the modified memory atom.

```
virtual MAtom< tMemType >& LimitSizeTo( typename const MUnit< tMemType >& muSize ) throw( MAtomException
);
```

Method - SaveAsFile A

Writes the content of the memory held by the memory atom at the beginning of the opened file represented with the hFile handle.

const HANDLE hFile – handle to file where to write the memory atom data.

The function returns reference to this memory atom. The method returns retaining the file handle hFile opened and pointing just after the newly written data. This method and virtual MAtom< tMemType >& AddToFile(const HANDLE hFile) can be used in conjunction to serialize number of memory atoms storing them sequentially in a file.

```
virtual const MAtom< tMemType >& SaveAsFile( const HANDLE hFile ) const throw();
```

Method - SaveAsFile B

Writes the content of the memory held by the memory atom as a file.

*const TCHAR *ptcFilename - filename to be used for saving the data*
const DWORD dwCreationDisposition - same meaning as the dwCreationDisposition in CreateFile file function.
const DWORD dwFlagsAndAttributes - same meaning as the dwFlagsAndAttributes in CreateFile file function.
DWORD pdwLastError - indicates the required behavior in case of error. If this parameter is not NULL and an error occurs the function will place the error code in the variable pointed by pdwLastError and return false. If pdwLastError is NULL the function will throw an exception which will contain the error code indicating the error. Using a returned error code instead of exception helps creating a more concise logic in certain cases, e.g. when using disposition flag - CREATE_NEW, and inspecting the returned code, and other cases such as sequences of if blocks.*

The method requires GENERIC_WRITE access rights, exclusive sharing rights and uses default security attributes.
The function returns true in case of success, and false or throws an exception in case of error depending on the content of pdwLastError.

```
virtual bool SaveAsFile( const TCHAR *ptcFilename, const DWORD dwCreationDisposition, const DWORD
dwFlagsAndAttributes, DWORD* pdwLastError ) const throw();
```

Method - SaveAsFile C

Writes the content of the memory held by the memory atom as a file.

*const TCHAR *ptcFilename - filename to be used for saving the data*
const DWORD dwCreationDisposition - same meaning as the dwCreationDisposition in CreateFile file function.
const DWORD dwFlagsAndAttributes - same meaning as the dwFlagsAndAttributes in CreateFile file function.
DWORD pdwLastError - indicates the required behavior in case of error. If this parameter is not NULL and an error occurs the function will place the error code in the variable pointed by pdwLastError and return false. If pdwLastError is NULL the function will throw an exception which will contain the error code indicating the error. Using a returned error code instead of exception helps creating a more concise logic in certain cases, e.g. when using disposition flag - CREATE_NEW, and inspecting the returned code, and other cases such as sequences of if blocks.*

The method requires GENERIC_WRITE access rights, exclusive sharing rights and uses default security attributes.
The function returns true in case of success, and false or throws an exception in case of error depending on the content of pdwLastError.

```
virtual bool SaveAsFile( const TCHAR *ptcFilename, const DWORD dwFlagsAndAttributes, DWORD* pdwLastError )
const throw();
```

Method - AppendToFile A

Writes the content of the memory held by the memory atom at the end a file. If the file does not exist the function creates it.

*const TCHAR *ptcFilename - filename to be used for saving the data.*

The method requires GENERIC_WRITE access rights, exclusive sharing rights and uses default security attributes.
The function returns reference to this memory atom.

```
virtual const MAtom< tMemType >& AppendToFile( const TCHAR *ptcFilename ) const throw();
```

Method - AppendToFile B

Writes the content of the memory held by the memory atom at the end of opened file.

const HANDLE hFile - handle to file at the end of which to add the memory atom content. The handle must have write permission attributes.

The function returns reference to this memory atom.

```
virtual const MAtom< tMemType >& AppendToFile( const HANDLE hFile ) const throw();
```

Method - AddToFile

Writes the content of the memory held by the memory atom in file.

const HANDLE hFile - handle to file where to write the memory atom content. The data will be written at the current file pointer position. The handle must have write permission attributes.

The function returns reference to this memory atom.

```
virtual const MAtom< tMemType >& AddToFile( const HANDLE hFile ) const throw();
```

Method - LoadFile A

Load an entire file in a memory atom.

*const TCHAR *ptcFilename - path to the file to be loaded.*

The size of the loaded file must be a multiple of the granularity of the memory atom. For example a file with size 6 bytes cannot be loaded in memory atom of class DWORD.
The function returns reference to this memory atom loaded with the data from the file.

```
virtual MAtom< tMemType >& LoadFile( const TCHAR *ptcFilename ) throw();
```

Method - LoadFile B

Load an entire file in a memory atom and returns an open handle to the file.

*const TCHAR *ptcFilename - path to the file to be loaded.*
MHandle& hFile - reference to a HANDLE wrapper which will assume the handle to the open file. The caller must close the handle, which MHandle does automatically. After the function returns the handle points at the end of the file.

The size of the loaded file must be a multiple of the granularity of the memory atom. For example a file with size 6 bytes cannot be loaded in memory atom of class DWORD.
The function returns reference to this memory atom loaded with the data from the file.

```
virtual MAtom< tMemType >& LoadFile( const TCHAR *ptcFilename, MHandle& hFile ) throw();
```

Method - LoadFile C

Load an entire file in a memory atom and returns an open handle to the file.

*const TCHAR *ptcFilename - path to the file to be loaded.*
const MUnit< tMemType >& muMaxUnits2Load - the maximum number of memory units to load.
MHandle phFile - optional reference to a HANDLE wrapper which if supplied will assume the handle to the open file. Pass NULL to avoid returning file handle, pass to a valid MHandle pointer to request returning of a valid file handle. The caller must close the handle when it is no longer needed, which MHandle does automatically. After the function returns the handle points at the end of the file.*

The size of the loaded file must be a multiple of the granularity of the memory atom or be bigger than muMaxUnits2Load in bytes. For example a file with size 6 bytes cannot be loaded in memory atom of class DWORD where muMaxUnits2Load is 2 units, but its first DWORD can be loaded if muMaxUnits2Load is 1.
The function returns reference to this memory atom loaded with the data from the file.

```
virtual MAtom< tMemType >& LoadFile( const TCHAR *ptcFilename, const MUnit< tMemType >& muMaxUnits2Load,
MHandle* phFile ) throw();
```

Method - LoadFromFile

Loads a memory atom with data from file reading at the current position onwards.

const MHandle& hFile - handle of open file. The data put into the memory atom is read from the current pointer position of this handle. If there is not enough data in the file (from the current pointer position to the end of the file) to fill up the memory atom, then the memory atom will adjust itself to reflect the amount of data available. The available data must be multiple to the granularity of the memory atom. For example a file with size 6 bytes, and handle pointer at 3, cannot be loaded in memory atom of class DWORD.

The function returns reference to this memory atom loaded with the data from the file.

```
virtual MAtom< tMemType >& LoadFromFile( const MHandle& hFile ) throw( MAtomException );
```

Method - FillNoise A

Fill the content of the memory atom with random data.

const bool bSeed - set to true to use the uiSeed to restart the random generator, set to false to ignore the uiSeed parameter and use the random generator at its current state.
const unsigned int uiSeed - restarting value of the random generator. This parameter is ignored if bSeed is false.

The function returns reference to this memory atom loaded with random data.

```
MAtom< tMemType >& FillNoise( const bool bSeed, const unsigned int uiSeed ) throw();
```

Method - FillNoise B

Fill the content of the memory atom with random data.

const unsigned int uiSeed - restart the random generator using this value.

The function returns reference to this memory atom loaded with random data.

```
MAtom< tMemType >& FillNoise( const unsigned int uiSeed )                    throw();
```

Method - FillNoise C

Fill the content of the memory atom with random data, using the current state of the random generator.
The function returns reference to this memory atom loaded with random data.

```
MAtom< tMemType >& FillNoise()                                               throw();
```

Methods Family - XOR

Unit-wise XOR of this memory atom and a memory atom of the same type and size, with result of every per-unit XOR saved in this.

const MAtom< tMemType >& maXORMemory - memory atom to XOR this. The operation is this-atom[index] = this-atom[index] XOR maXORMemory[index]
const tMemType& mtXORUnit - memory unit to XOR this. The operation is this-atom[index] = this-atom[index] XOR mtXORUnit

The function returns reference to this memory atom unit-wise XOR-ed with the memory units of the parameter memory atom.

```
MAtom< tMemType >& XOR( const MAtom< tMemType >& maXORMemory )              throw();
MAtom< tMemType >& XOR( const tMemType& mtXORUnit )                         throw();
```

Method - Set

Fill the content of a memory atom with the value.

const tMemType& objPattern - value used to set every unit in the memory atom.

The function returns reference to this memory atom filled with objPattern.

```
MAtom< tMemType >& Set( typename const tMemType& objPattern ) throw();
```

Method - Reset

Byte-wise fill the content of a memory atom with the value.

const BYTE b1Wipe - byte value used to set every byte in the memory atom to. Every byte of the contained memory is set to this value.

The function returns reference to this memory atom filled with b1Wipe.

```
MAtom< tMemType >& Reset( typename const BYTE b1Wipe )        throw();
```

Method - Reverse

Unit-wise reversing of the content of the memory atom.
The function returns reference to the modified (this) memory atom.

```
MAtom< tMemType >& Reverse() throw();
```

Method - Replace

Replace every occurrence of a value in the contained memory units with another value.

const tMemType tReplaceValue - value to replace.
const tMemType tNewValue   - new value to replace all occurrences of tReplaceValue.

The function returns reference to the modified (this) memory atom.

```
MAtom< tMemType >& Replace( const tMemType tReplaceValue, const tMemType tNewValue ) throw();
```

Method - TrimLeft

Remove all units with a particular value, found on the left hand side of the memory contained in a memory atom. The function returns reference to the modified (this) memory atom.

```
MAtom< tMemType >& TrimLeft( const tMemType tTrimValue ) throw();
```

Method - TrimRight

Remove all units with a particular value, found on the right hand side of the memory contained in a memory atom. The function returns reference to the modified (this) memory atom.

```
MAtom< tMemType >& TrimRight( const tMemType tTrimValue ) throw();
```

Method - Trim

Remove all units with a particular value, found on both sides of the memory contained in a memory atom. The function returns reference to the modified (this) memory atom.

```
MAtom< tMemType >& Trim( const tMemType tTrimValue ) throw();
```

Method - Find

Find the zero based index of the first occurrence of a particular value with type of the class of the atom.

const tMemType tSearchValue - value to search.

The function returns the zero based index of the first match. If there is no match the function returns a memory unit with the size of the memory atom in class units, which is the first/smallest invalid index.

```
MUnit< tMemType >  Find( const tMemType tSearchValue ) const throw();
```

Method - Search_L2R A

Search for a memory match in this memory atom from left to right (from the beginning towards the end).

const MAtom< tMemType >& memSearchFor - memory atom for the content of which to look for in this memory atom.
const bool bPartialMatchAtTheEnd - flag indicating whether a partial match at the end of the contained memory is to be considered a match, true - yes, false - no.

Returns an empty shell memory atom if there is no match. Returns a non-empty shell memory atom if there is a complete match, or if there is an incomplete match and the partial memory flag is set to true. For example a partial match in the middle of the memory is meaningless and an empty shell object will be always returned. If the partial flag is set to true and there is a partial match at the end of the memory, there may be a full match is there is more data behind the end of the memory atom, so the method will return a partial match, i.e. a shell memory atom wrapping the memory from the beginning of the match to the end of the memory in this atom. To distinguish a full from a partial match use the size of the returned shell memory atom and memSearchFor.

```
const ShellMemory< tMemType > Search_L2R( typename const MAtom< tMemType >& memSearchFor, const bool
bPartialMatchAtTheEnd ) const;
```

Method - Search_L2R B

Search for a memory match in this memory atom from left to right (from the beginning towards the end). This method is applicable only for memory atoms with class sizes of only one or two bytes.

const MAtom< tMemType >& memSearchFor - memory atom for the content of which to look for in this memory atom.
const bool bPartialMatchAtTheEnd - flag indicating whether a partial match at the end of the contained memory is to be considered a match, true - yes, false - no.
const bool bCaseSensitive - set to true for case sensitive search, and false otherwise.

Returns an empty shell memory atom if there is no match. Returns a non-empty shell memory atom if there is a complete match, or if there is an incomplete match and the partial memory flag is set to true. For example a partial match in the middle of the memory is meaningless and an empty shell object will be always returned. If the partial flag is set to true and there is a partial match at the end of the memory, there may be a full match is there is more data behind the end of the memory atom, so the method will return a partial match, i.e. a shell memory atom wrapping the memory from the beginning of the match to the end of the memory in this atom. To distinguish a full from a partial match use the size of the returned shell memory atom and memSearchFor.

```
const ShellMemory< tMemType > Search_L2R( typename const MAtom< tMemType >& memSearchFor, const bool bPartialMatchAtTheEnd, const bool bCaseSensitive ) const;
```

Method - FindInFile

Find this memory atom content in a file.

const MHandle& hFile - handle to an open file to search in. The handle must have read and seek permissions.
const unsigned __int64 uiSearchFrom - starting position, in bytes, to start the search in the file.
unsigned __int64 puiFoundAt - optional parameter to hold the starting position of the first found match from the beginning of the file in bytes. NULL - starting position is not required.*
const HANDLE heStop - synchronization event handle to asynchronously terminate the search.

Returns true if the file contains data identical to the content of this memory atom, and false if a match was not found.

```
bool  FindInFile( const MHandle& hFile, const unsigned __int64 uiSearchFrom, unsigned __int64* puiFoundAt, const HANDLE heStop ) const;
```

Method - Bin2Ascii

Encode the content of the memory atom as ASCII in chars. Every byte is split to more/less significant 4 bits, which number is converted to ASCII. For example ABCD\0, i.e. 0x41 0x42 0x43 0x44 0x00 becomes 0x34 0x31 0x34 0x32 0x34 0x33 0x34 0x34 0x30 0x30. The function returns a new memory atom with the encoded data from this.

```
MemoryPH< char >   Bin2Ascii() const;
```

Method - Ascii2Bin

Decode ASCII encoded binary data. This function is inverse of MemoryPH< char > Bin2Ascii() const;.

```
MAtom< tMemType >& Ascii2Bin();
```

Method - MHash

Creates and returns an 8 byte integer hash code for the content of the memory atom.

```
unsigned __int64 MHash() const;
```

Method - GetHash1

Creates and returns a hash code of the contained by the memory atom memory as a floating point number with double precision.

```
virtual double GetHash1() const;
```

MemoryPH< class tMemType >

Generic Memory Atom using the Process Heap for memory atom Type::Origin property, i.e. as memory resource. MemoryPH Inherits directly from MAtom< tMemType >. The tMemType - defines the Type::Class property of the memory atom, i.e. granularity and type of the memory units contained by the memory atom.

Method - GetMemoryOrigin

Pure polymorphic method.
Returns the Memory Origin.

```
virtual MemoryOrigin GetMemoryOrigin() const;
```

Constructor - MemoryPH A

Standard default constructor - allocates zero bytes.

```
MemoryPH();
```

Constructor - MemoryPH B

Standard copy constructor - creates a new memory object using the source object.

const MemoryPH< tMemType >& maSource - source memory atom used to construct this object.

The constructor allocates exactly as much memory as maSource holds and copies its content onto the newly allocated memory.

```
MemoryPH( typename const MemoryPH< tMemType >& maSource );
```

Constructor - MemoryPH C

Copy constructor from any MAtom< tMemType > descendent memory atom type - creates a new memory object using the source object.

const MAtom< tMemType >& maSource - source memory atom used to construct this object.

The constructor allocates exactly as much memory as maSource holds and copies its content onto the newly allocated memory.

```
MemoryPH( typename const MAtom< tMemType >& maSource );
```

Constructor - MemoryPH D

Constructor allocating required number of memory units.

const MUnit< tMemType >& muUnits - number of memory units that the newly created memory atom must allocate.

The template specialization parameters determine the class of the allocated memory, which must match to the class of muUnits, and whether allocated/reallocated memory must be zeroed or not.
Use: allocate 100 * sizeof(DWORD) bytes.
MemoryPH< DWORD, true > ma100ZeroedDWords(MUnit< DWORD >(100));

```
MemoryPH( typename const MUnit< tMemType >& muUnits );
```

Constructor - MemoryPH E

Constructor allocating required number of memory units, and initializing them with a value.

const MUnit< tMemType >& muUnits - number of memory units that the newly created memory atom must allocate.
const tMemType& tobjInitValue - value used to initialize the allocated memory unites.

The template specialization parameters determine the class of the allocated memory, which must match to the class of muUnits, and whether reallocated memory must be zeroed or not.
Use: allocate 100 * sizeof(DWORD) bytes, and initialize every unit in the memory atom with 0xA5A5A5A5.
MemoryPH< DWORD, true > ma100ZeroedDWords(MUnit< DWORD >(100), 0xA5A5A5A5);

```
MemoryPH( typename const MUnit< tMemType >& muUnits, typename const tMemType& tobjInitValue );
```

Constructor - MemoryPH F

Constructor creating a memory atom coping non Atomic Memory Model managed memory.

const tMemType pSrcData - pointer to a non-Atomic Memory Model managed memory to be copied in the memory atom.*
const MUnit< tMemType >& muUnits - size of the source memory in memory units.

The template specialization parameter determines the class of the allocated memory.
Use: allocate 100 * sizeof(DWORD) bytes, and copy the content of 100 * sizeof(DWORD) bytes pointed by pDWordSourceMemory.
MemoryPH< DWORD > ma100DWords(pDWordSourceMemory, MUnit< DWORD >(100));

```
MemoryPH( const tMemType* pSrcData, typename const MUnit< tMemType >& muUnits );
```

Method - Empty

Empty the memory atom. The memory contained by the atom is released to the process heap, and a new zero size chunk is allocated.
The method returns reference to this memory atom emptied.

```
virtual MAtom< tMemType >& Empty() throw( MAtomException );
```

Methods Family - operator=

Operator equal replicating the content of this object with that of the parameter memory atom.

const MAtom< tMemType >& maSource - source memory atom used to modify this object.
const MemoryPH< tMemType >& maSource - source memory atom used to modify this object.

The operator frees the contained memory and then allocates exactly as much memory as maSource holds, after which it copies its content onto the newly allocated memory of this atom.
The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemType >& operator=( typename const MAtom< tMemType >& maSource );
virtual MemoryPH< tMemType >& operator=( typename const MemoryPH< tMemType >& maSource );
```

Method - CanReAllocate

Virtual override function always returning true for this type as it is able to reallocate memory.

```
virtual bool CanReAllocate() const throw();
```

Method - ReAllocate

Free the contained memory and reallocate new memory with the required size.

const MUnit< tMemType >& muUnits - size of the new memory to allocate in memory units.

The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemType >& ReAllocate( typename const MUnit< tMemType >& muUnits ) throw( MAtomException );
```

Method - ReAllocateTransfer

Free the contained memory and replicate the content of the MAtom< tMemType > descendent memory atom passed as parameter.

const MAtom< tMemType >& maSource - source memory atom used to modify this object.

The method frees the contained memory and then allocates exactly as much memory as maSource holds, after which it copies its content onto the newly allocated memory of this atom.
The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemType >& ReAllocateTransfer( typename const MAtom< tMemType >& maSource ) throw( MAtomException );
```

Method – operator+

Concatenates two memory atoms into a new MemoryPH memory atom.

const MAtom< tMemType >& maSummandL - the content of this memory atom will be placed at the beginning of the result memory atom.
const MAtom< tMemType >& maSummandR - the content of this memory atom will be placed after the content of maSummandL in the result memory atom.

The method returns a new MemoryPH< tMemType > memory atom holding copy of the contents of the parameters as follow: [maSummandL][maSummandr].
Note that the returned memory atom is passed back in a standard manner through a standard copy constructor, and thus this is not a speed and memory efficient function, which should be used primarily for memory atoms not containing large amounts of memory. For such cases use the Transfer(parameters) functions.

```
friend inline MemoryPH< tMemType > operator +( typename const MAtom< tMemType >& maSummandL, typename const MAtom< tMemType >& maSummandR ) throw( MAtomException );
```

ShellMemory< tMemClass >

The Shell Memory Atom Type does not hold own memory but instead wraps/references memory owned by another entity. ShellMemory Inherits directly from MAtom< tMemType >. The tMemType - defines the Type::Class property of the memory atom, i.e. granularity and type of the memory units referenced by the memory atom. This type of memory atoms are very useful to help supply the MAtom< tMemClass

> interface to non-Atomic Memory Model owned memory, give a MAtom< tMemClass > interface to a window within memory owned by other memory atoms, as well as for helping functions.

Method - GetMemoryOrigin

Pure polymorphic method.
Returns the Memory Origin.

```
virtual MemoryOrigin GetMemoryOrigin() const;
```

Constructor - ShellMemory A

Default constructor - no memory is allocated. The memory object is set to point to NULL memory which has zero size.

```
ShellMemory();
```

Constructor - ShellMemory B

Copy constructor from any MAtom< tMemClass > descendent memory atom type - creates a new memory object using the source object.

const MAtom< tMemClass >& maSource - source memory atom used to construct this object.

The newly created object does not allocate any memory, instead it points to the same memory, with the same size, which maSource owns/references.

```
ShellMemory( const MAtom< tMemClass >& maSource );
```

Constructor - ShellMemory C

Constructor initializing a new ShellMemory memory atom representing a portion of memory owned/referenced by another memory atom.

MAtom< tMemClass >& maSource - the memory atom part of whose content will be represented by the newly created memory atom.
const MUnit< tMemClass >& muOffset - zero based offset of the first memory unit in the source memory atom which will be represented by this.
const MUnit< tMemClass >& muSize - number of memory units which will be referenced by the newly created memory atom.

This constructor does not allocate any memory. Instead it simply creates a ShellMemory atom which points memory with certain size owned by another entity.

```
ShellMemory( MAtom< tMemClass >& maSource, const MUnit< tMemClass >& muOffset, const MUnit< tMemClass >& muSize ) : MAtom< tMemClass >( maSource, muOffset, muSize );
```

Constructor - ShellMemory D

Standard copy constructor - creates a new memory object using the source object.

const ShellMemory< tMemClass >& maSource - source memory atom used to construct this object.

This constructor does not allocate any memory. Instead it simply creates a ShellMemory atom which points the memory with the same size pointed by the source object.

```
ShellMemory( const ShellMemory< tMemClass >& maSource ) : MAtom< tMemClass >( maSource );
```

Constructor - ShellMemory E

Constructor initializing a new ShellMemory memory atom using the abstract parent of all memory atom classes in the Atomic Memory Atom implementation I.

MMemory< tMemClass >& maSource - source memory atom used to construct this object.

This constructor does not allocate any memory. Instead it simply creates a ShellMemory atom which points the memory with the same size pointed by the source object. This constructor is useful for creating interfacing memory atoms for use in code utilizing memory atoms from implementation one and two.

```
ShellMemory( MMemory< tMemClass >& maSource );
```

Constructor - ShellMemory F

Constructor creating a memory atom referencing non Atomic Memory Model managed memory.

tMemClass pSrcData - pointer to a non Atomic Memory Model managed memory to be referenced by the memory atom.*
const MUnit< tMemClass >& muUnits - size of the source memory in memory units.

The template specialization parameter determines the class of the assumed memory.
Use - references 100 * sizeof(DWORD) bytes pointed by pDWordSourceMemory
ShellMemory< DWORD > ma100DWords(pDWordSourceMemory, MUnit< DWORD >(100));

```
ShellMemory( tMemClass* pSrcData, typename const MUnit< tMemClass >& muUnits );
```

Method - Empty

Empty the memory atom. The memory object is set to point to NULL memory which has zero size.
The method returns reference to this memory atom pointing to NULL memory with zero size.

```
virtual MAtom< tMemClass >& Empty() throw( MAtomException );
```

Methods Family - operator=

Operator equal replicating the content of this object with that of the parameter memory atom.

MAtom< tMemClass >& maSource - source memory atom used to modify this object.
ShellMemory< tMemClass >& maSource - source memory atom used to modify this object.

The operator assumes the memory owned/referenced by the source memory atom. Notice that the passed memory object is NOT const since this memory atom is mutable and assumes the same memory chunk which the source atom holds and it could change it. The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemClass >& operator=( typename MAtom< tMemClass >& maSource );
virtual ShellMemory< tMemClass >& operator=( typename ShellMemory< tMemClass >& maSource );
```

Method - CanReAllocate

```
// Virtual override function always returns false for this type as it is not able to reallocate memory.

virtual bool CanReAllocate() const throw();
```

Method - ReAllocate

This ReAllocate is permitted only if this object refers to NULL memory, allowing it to be used as information carrier.

const MUnit< tMemClass >& muNewSize - new size of the MULL memory.

Generally this method is inconsistent since const memory assumed by mutable object may be changed, hence the above limitation.
The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemClass >& ReAllocate( typename const MUnit< tMemClass >& muNewSize ) throw( MAtomException );
```

Method - ReAllocateTransferMutableAssumedMemory

Assumes content of any MAtom< tMemClass > descendent memory atom passed as parameter.

const MAtom< tMemClass >& maSource - source memory atom used to modify this object.

Generally this method is inconsistent since const memory assumed by mutable object may be changed. However there are cases where the method can be helpful, hence the unambiguous name ReAllocate-Transfer-Mutable-Assumed-Memory clarifying the resolution of the issue. The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemClass >& ReAllocateTransferMutableAssumedMemory( typename const MAtom< tMemClass >& maSource ) throw( MAtomException );
```

Prohibited Method - operator=

Impossible: Operator equal assuming constant reference to the content of any MAtom< tMemClass > descendent memory atom type by this object.

Reason: This operator cannot exist because it is impossible to guarantee that the object (this) will remain constant after passing the const source to it.

```
virtual MAtom< tMemClass >& operator=( typename const MAtom< tMemClass >& /*maSource*/ );
```

Prohibited Method - ReAllocateTransfer

Impossible: ReAllocateTransfer on constant source memory.

Reason: This method cannot exist because the const attribute cannot be enforced after the const source memory is delegated to a mutable object.

```
virtual MAtom< tMemClass >& ReAllocateTransfer( typename const MAtom< tMemClass >& /*maSource*/ ) throw( MAtomException );
```

HandleMemory< tMemClass >

Generic Memory Atom using the GlobalAlloc for memory atom Type::Origin property, i.e. as memory resource, and is implementing the system IStream interface. HandleMemory Inherits directly from MAtom< tMemType >. The tMemType - defines the Type::Class property of the memory atom, i.e. granularity and type of the memory units contained by the memory atom.

Constructor – HandleMemory A

Constructor from resource located in the current executable module, the resource type and name.

```
const TCHAR* strType   - resource type.
const DWORD dwName     - resource name.

explicit HandleMemory( const TCHAR* strType, const DWORD dwName );
```

Constructor – HandleMemory B

Constructor from handle to module, resource type and name.

```
const HMODULE hmModule - handle to the module containing the resource to be loaded in the newly created memory atom.
const TCHAR* strType   - resource type.
const DWORD dwName     - resource name.

explicit HandleMemory( const HMODULE hmModule, const TCHAR* strType, const DWORD dwName );
```

Constructor – HandleMemory C

Constructor from path to module, resource type and name.

```
const TCHAR* strType   - resource type.
const DWORD dwName     - resource name.
const TCHAR* strModule - path to the module containing the resource to be loaded in the newly created memory atom.

explicit HandleMemory( const TCHAR* strModule, const TCHAR* strType, const DWORD dwName );
```

Method – GetStream

```
// Return IStream pointer to the allocated memory.

IStream* GetStream();
```

Method - GetMemoryOrigin

Pure polymorphic method.
Returns the Memory Origin.

```
virtual MemoryOrigin GetMemoryOrigin() const;
```

Method – Empty

Empty the memory atom. The memory contained by the atom is released, and the memory object is set to point to NULL memory which has zero size.
The method returns reference to this memory atom pointing to NULL memory with zero size.

```
virtual MAtom< tMemClass >& Empty() throw( MAtomException );
```

Method – CanReAllocate

Virtual override function always returns false for this type as it is not able to reallocate memory.

```
virtual bool CanReAllocate() const throw();
```

Not Yet Implemented Methods

To be appropriately implemented in later releases.

```
virtual MAtom< tMemClass >& operator=( typename const MAtom< tMemClass >& /*maSource*/ );
virtual MAtom< tMemClass >& ReAllocate( typename const MUnit< tMemClass >& /*muUnits*/  );
virtual MAtom< tMemClass >& ReAllocateTransfer( typename const MAtom< tMemClass >& /*maSource*/ );
virtual MAtom< tMemClass >& operator=( typename MAtom< tMemClass >& /*maSource*/ );
```

SecureMemory< MemoryType, tMemClass >

Specialized Memory Atom derivate of any MAtom< tMemClass > insatiable memory atom type, with Type::Semantics to zero any memory before it is released. The MemoryType must be a MAtom< tMemClass > derivate memory atom which does the memory management and performs the fundamental memory handling and operations. The tMemClass - defines the Type::Class property of the memory atom, and must be the same as the Type::Class property of the superclass MemoryType.

Constructor – SecureMemory A

Standard default constructor - allocates zero bytes.

```
SecureMemory();
```

Constructor – SecureMemory B

Effectively standard copy constructor - creates a new memory object using the source object.

const MemoryType maSource - source memory atom used to construct this object.

The constructor passes the control to the standard copy constructor of the specializing parent class which handles the memory operations.

```
SecureMemory( const MemoryType maSource );
```

Constructor – SecureMemory C

Copy constructor from any MAtom< tMemClass > descendent memory atom type - creates a new memory object using the source object.

const MAtom< tMemClass >& maSource - source memory atom used to construct this object.

The constructor passes the control to the same prototype copy constructor of the specializing parent class which handles the memory operations.

Use: SecureMemory< MemoryPH< BYTE >, BYTE > memData(MUnit< BYTE >(40));

```
SecureMemory( const MAtom< tMemClass >& maSource );
```

Constructor – SecureMemory D

Constructor creating a memory atom coping non Atomic Memory Model managed memory.

```
const tMemClass* pSrcData         - pointer to a non-Atomic Memory Model managed memory to be copied in the memory atom.
const MUnit< tMemClass >& muUnits - size of the source memory in memory units.
```

The template specialization parameter determines the class of the allocated memory.

```
SecureMemory( const tMemClass* pSrcData, typename const MUnit< tMemClass >& muUnits );
```

Method – Empty

Empty the memory atom. First erases the memory by forcing zeroing and then releases it via the specializing parent Empty() operation.
The method returns reference to this memory atom emptied.

```
virtual MAtom< tMemClass >& Empty() throw( MAtomException );
```

Method – operator=

Operator equal replicating the content of this object with that of any MAtom< tMemClass > descendent memory atom type.

const MAtom< tMemClass >& maSource - source memory atom used to modify this object.

The operator frees the contained memory after erasing it, and then passes the control to the identical operator= of the specializing parent class which handles the memory operations to appropriately handle the allocation and copy operations.
The method returns reference to the modified (this) memory atom.

```
virtual MAtom< tMemClass >& operator=( typename MAtom< tMemClass >& maSource );
```

Method – LimitSizeTo

Decreases the number of the contained memory units to the muSize passed as parameter, after cleaning the extra memory. The function returns a reference to the modified memory atom.

```
virtual MAtom< tMemClass >& LimitSizeTo( typename const MUnit< tMemClass >& muSize ) throw( MAtomException
);
```

Prohibited Methods Family

Methods banned for security reasons.

```
operator       MemoryType  () const throw() { return( *this ); }
operator       MemoryType  ()       throw() { return( *this ); }
operator const MemoryType& () const throw() { return( *this ); }
operator       MemoryType& ()       throw() { return( *this ); }
operator const MemoryType* () const throw() { return(  this ); }
operator       MemoryType* ()       throw() { return(  this ); }
```

MStringEx

The MstringEx< tChar > also typedef-ed as “string“ is a generic string class based on MemoryPH as follows:

```
template< class tChar > class MStringEx : public MemoryPH< tChar >
```

Note that in addition to MUnitException and MAtomException, the string class also throws MStringExException.

Since the operation of the string class is not immediately relevant to the Atomic Memory Model we will not include the methods of the string class in this document. They are however self-explanatory by simply considering their respective names, prototypes and descriptions where available in the source file.

MSecureString

The MSecureString is a sting class inheriting from MStringEx which has the same semantics as the SecureMemory memory class to zero memory before releasing it.

```
template< class tChar > class MSecureString : protected MStringEx< tChar >
```

Since the operation this string class is not immediately relevant to the Atomic Memory Model we will not include the methods of the string class in this document. They are however self-explanatory by simply considering their respective names, prototypes and descriptions where available in the source file.

MStringEx2Z

The MString2Z is a sting class inherits from MStringEx. Objects from MString2Z have double zero ending at the end of the string.

```
template< class tChar > class MStringEx2Z : protected MStringEx< tChar >
```

Since the operation this string class is not immediately relevant to the Atomic Memory Model we will not include the methods of the string class in this document. They are however self-explanatory by simply considering their respective names, prototypes and descriptions where available in the source file.

Note:
Example implementation one has additional memory types, such as SegmentedMemory, StackMemory and others, some of which will be gradually transferred to implementation two.

Example of use of the Atomic Memory Model

```
#include <windows.h>
#include <stdio.h>
#include "..\MAtom.h"
#include "..\StringEx.h"


// Only needed if the MStringEx< tChar > is included in the project.
// Set this handle if you use the string class to load resources.
HMODULE hDefaultTextResourceModule = NULL;


MemoryPH< DWORD > fun1( MemoryPH< DWORD > m )
{
   printf(TEXT("fun1: memory units = %d, bytes = %d   "), m.GetSize().GetUnits(), m.GetSize().InBytes());


   for( MUnit< DWORD > muCounter( 0 ); muCounter < m.GetSize(); muCounter++ )
   {
      printf( TEXT(" %d "), m[muCounter] );
   }


   printf( TEXT("\n\n") );


   // Modify the second DWORD.
   m[1] = 7;


   // Return the modified object.
   return( m );
}


void fun2( const MemoryPH< DWORD >& m )
{
   printf(TEXT("fun2: memory units = %d, bytes = %d   "), m.GetSize().GetUnits(), m.GetSize().InBytes());


   for( MUnit< DWORD > muCounter( 0 ); muCounter < m.GetSize(); muCounter++ )
   {
      printf( TEXT(" %d "), m[muCounter] );
   }


   printf( TEXT("\n\n") );
}


void fun3( const MAtom< DWORD >& m )
{
   printf(TEXT("fun3: memory units = %d, bytes = %d   "), m.GetSize().GetUnits(), m.GetSize().InBytes());


   for( MUnit< DWORD > muCounter( 0 ); muCounter < m.GetSize(); muCounter++ )
   {
      printf( TEXT(" %d "), m[muCounter] );
   }


   printf( TEXT("\n\n") );
}


int main()
{
   try
```

```
{
   // Create a memory unit integer holding 5 in reference to BYTEs.
   MUnit< BYTE > mu( 5 );


   // Test if the memory units of BYTEs can be converted to memory units of DWORDs.
   bool bCanConverttoDword = mu.CanItBe< DWORD >();


   // Create a 256 __int64 memory integer, and convert it to DWORD, i.e. 256 __int64 in DWORDs.
   MUnit< DWORD > m1( MUnit< __int64 >( 256 ).As< DWORD >() );


   // Create an empty memory atom on the process heap, to be holding bytes.
   MemoryPH< BYTE >  m;


   // Create 2 empty memory atoms to be holding DWORDs and using the process heap memory.
   MemoryPH< DWORD > ma1, ma2;


   // Create a memory atom on the heap holding 3 DWORDs, each initialized with 0xA5A5A5A5.
   MemoryPH< DWORD > ma3( MUnit< DWORD >( 3 ), 0xA5A5A5A5 );


   // Create simple DWORD array on the stack.
   DWORD dwa[3] = { 1, 2, 3 };


   // Create a memory atom coping the content of the stack array.
   MemoryPH< DWORD > ma4( dwa, MUnit< DWORD >( 3 ) );


   // Create a shell memory atom, which refers to the stack array.
   ShellMemory< DWORD > ma5( dwa, MUnit< DWORD >( 3 ) );


   // Call fun1 passing the ma4 to print its content, and assign the returned modified memory to ma2.
   ma2 = fun1( ma4 );


   // Call fun2 passing the ma2 to print its content.
   fun2( ma2 );


   // Call fun3 passing the ma2 to print its content.
   fun3( ma5 );


   // Join the Stack content of ma5 and the heap content of ma2, and assign it to ma2.
   ma2 = ma5 + ma2;


   // Print the m2 content.
   fun3( ma2 );


   // Print the middle 4 DWORDS of m1.
   fun3( ma2.SubMemory< DWORD >( MUnit< DWORD >( 1 ), MUnit< DWORD >( 4 ) ) );


   // Create a zero terminated string object using the typedef string. You can also use
   // MStringEx< char > strName( "string" );  to specialize the particular object using
   // wchat_t, TCHAR or another appropriate char definition.
   string strName( "Hello!" );


   // Print the strName string.
   fun4( strName );
```

```
      // Create a shell memory atom referencing the content of strName except the terminating zero.
      ShellMemory< TCHAR > maName( strName.SubMemory( MUnit< TCHAR >( 0 ), strName.GetSize() - 1 ) );


      // Print the maName memory atom.
      fun4( maName );


//      const DWORD d( ma1[25] ); // Generate exception - index is outside of memory.


      DWORD dw( 6 );
      int   i = 3;
      unsigned int ui = 4;

      MUnit< DWORD > mu1( 2 );
      MUnit< DWORD > mu2( 3 );
      MUnit< DWORD > mu3( dw );
      MUnit< DWORD > mu4( i );
      MUnit< DWORD > mu5( ui );


//      mu2 += MUnit< DWORD >( -3 ); // Generate exception - memory units overflow.


      mu5 = MUnit< DWORD >( 3 );


      mu2 = 12 / mu2;


//      mu4 = mu5 - 3 * (3 + mu2); // Generate exception - negative memory units.
   }
   catch( const MUnitException e )
   {
      printf( TEXT( "Memory unit exception.\n\n" ) );
   }
   catch( MAtomException e )
   {
      printf( TEXT( "Memory atom exception.\n\n" ) );
   }
   catch( MStringExException e )
   {
      printf( TEXT( "String exception.\n\n" ) );
   }
   catch( ... )
   {
      printf( TEXT( "Unknown exception.\n\n" ) );
   }


   return( 0 );
}
```

Including the Atomic Memory Model in a Project

To start using the Atomic Memory Model in a project simply copy the supplied header files in a folder on a path included in your project and `#include "MAtom.h"` header file in the files where memory atoms are required. You may also wish to explicitly include some of the other files provided with the Atomic Memory Model depending on your needs. For example, if you need any of the string classes `#include "StringEx.h"` header file which contains the various string classes definitions. If you need the Memory Atoms available in example implementation one, which are not yet transferred to example

implementation two, `#include "MMemory.h"` header file. To use other facilities delivered with the Atomic Memory Model, such as generic list class, smart pointer class, smart handle class etc., include the respective header file as required. The Atomic Memory Model files are:

`#include "Common.h"`
- non-essential file, includes MASSERT macro. This file can be removed provided that the required macro is otherwise available.

`#include "MException.h"`
- essential file, containing exception definitions.

`#include "MAtom.h"`
- essential file, containing the example implementation two main memory atom classes definitions.

`#include "StringEx.h"`
- essential file, if strings are required, containing definition of string classes based on MemoryPH.

`#include "MMemory.h"`
- non-essential file, containing example implementation one. Unless required for some of its capabilities this file can be removed. You will have to also remove the MMemory& constructor of MemoryPH.

`#include "MHandle.h"`
- not part of the Atomic Memory model, but providing MHandle class used in some functions of Atomic Memory Model classes.

`#include "MSmartPtr.h"`
- not part of the Atomic Memory model, but providing a smart pointer class used in some functions of Atomic Memory Model classes.

`#include "MList.h"`
- non-essential file, providing generic list class used in the Segmented Memory atom available in example implementation one, which will be added to example implementation two in future.

```
#include "ResId.h"
```

- non-essential file, providing ResId (Resource Identifier) class used in the MStringEx string semantics memory atom class. The ResId class is an integer wrapper, analogous to the Memory Unit class and is used purely for the purpose of disambiguation.