Atomic Memory Model

M. Bonchev

Key Words: *Memory leaks and overruns, Memory abstraction.*

Abstract. *The task to track down and remove memory leaks is known as one of the most tedious and unpleasant ones that a software engineer might encounter. In some cases once part of a program memory leaks it might become impossible to remove. Buffer overruns are another problem often even more difficult and unpleasant to solve. This paper suggests an effective way to eradicate the above issues and other problems related to memory in a computer program. This is achieved by enforcing a preemptive methodology for memory handling in addition to abstraction of the memory and representing it as encapsulated entity with three properties: class, semantics and origin. The Atomic Memory Model is a consistent and elegant methodology as opposed to garbage collecting. It significantly improves the quality of code making most of the issues related to memory intrinsically impossible to occur and increasing the speed and ease of development.*

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.

History and Background

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 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, simply the memory management system 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 certain size. For example if a caller requests memory with size 3 bytes it might receive address X, e.g.:

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

```
// „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 );
```

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 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 results in one or more 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 thought 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, which could be so wrong that even the sizes of the allocated memory block and the entity that the memory was „converted“ to may not match.
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 not only be incorrectly interpreted, as demonstrated in the previous points, but may also be unwantedly modified, e.g. by mistake resulting in 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 crash.
8. Memory is allocated with very limited, non functional 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 in not much more beyond these. Certainly one could use the starting address as identification scheme and develop some additional mechanism such as 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 is one or more memory leaks. One reason for the uselessness of the starting address as identification scheme is because 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. Although memory identification is necessary the most intuitive and commonly used identification scheme using the memory block start address is not useful to any degree in an abstract and very little in a physical sense.

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 (build in or user defined) and semi-abstract is an array from 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. Ttherefore the objective now is to remove the identified contradiction. This can be achieved by:

• either removing any abstraction when using memory, which means to band any sort of data type except the native for the physical memory data placeholder (most commonly BYTE). This will revert the high level language to assembly or even machine code and thus defeat the purpose of high level languages. Such a solution would obviously add a whole new host of problems and possible errors, to escape from which the high level languages were developed on first place. For this reason we will not consider this possibility any further.

• or somehow 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.

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 no longer exists as a memory at any abstract level; instead it exists as a Memory

Atom(s) of certain type. One can understand this as „memory does not exist“, memory atoms do. 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 encapsulated entity, which is available to access only via set of exposed interfaces.

When a memory atom is created, it allocates 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 is a type which has the following properties:

1. Class – specialization and granularity. The „class“ property specializes the Memory Atom – it specifies the type of the contained elements. Therefore the „class“ property also „granulates“ the Memory Atom. For example specializing a Memory Atom as 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. what its nature is, and how it will behave, what characteristics it will possess, what interfaces it will expose. 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, might not be allocated/released at all, e.g. shell memory, etc.

The *Type* of memory atom is defined by the three properties *Class*, *Semantics* and *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 number of bool items. To refer to an item held by a memory atom we use an auxiliary entity called Memory Unit.

Memory Unit

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 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.The reason why memory units are bound to memory atoms is because there would be an assortment of confusions if a non-bound to the class of the memory atom type of memory units are used when referring to its items. For example if using integer Allocate(int iUnits) instead of specialized type Allocate(Unit< class >) one could easily by mistake pass any value for iUnits meaning for example the same amount of units but in bytes instead of DWORDs, etc. In this regard the class property of a memory atom and memory units is its dimension:

The <u>class</u> property of a Memory Atoms and Memory Units defines the <u>dimension</u> of these entities.

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 this 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 class. If using a generic integer values 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 units 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.

Memory Units from the class of a Memory Atom are the only way to refer to it's items.

Space of Existence

Instantiated memory atoms as any other entity exist in a space, therefore we can strictly define space of existence where a memory atom is created, used and on the exit of it destroyed. The space of existence can be a simple scope (stack frame) where Memory Atom (as any other object) is created, lives and on the exit of which is destroyed or it could be not affected directly by the stack frames enter/exit, but determined by the logic of execution and current circumstances of that execution. For example a space of existence for memory atoms could be a linked list declared in a relatively outer for the current execution stack frame, e.g. in the global stack frame or in the „main“

function stack frame. Memory atoms are created, used and destroyed as appropriat during the program execution according to the program logic and the current circumstances of execution and not dependent on the scope(s) where the execution passes through. When the stack frame where the space of existence is instantiated is released (normally or abnormally (due exception)) the space of existence is destroyed as a usual stack object and all contained memory atoms are released by the Space of Existence destructor. It is essential that the space of existence is always directly or indirectly engaged with a stack frame so its destructor is called when releasing (exiting) the containing stack frame.

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

In summary the golden rules of the Atomic Memory Model are:

1. Memory Atoms have *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 it’s 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.

As defined the Atomic Memory Model abstracts memory and successfully resolves all possible erroneous conditions described earlier as follows:

Memory leaks	
Reason	Problem resolved by
- incomplete, imprecise or incorrect algorithm	Object destructor is implicitly called by the compiler on exit c scope.
- error in the algorithm implementation	Object destructor is implicitly called by the compiler on exit c scope.
- memory leaked during exception handling	Object destructor is implicitly called by the compiler on exit c scope. The structured exceptio system guarantees that all stack frames are properly exited and all objects within them properl destroyed.
- memory is released to wrong memory system	The correct destructor is called implicitly by the compiler.
- memory is released to wrong heap	The correct destructor is called implicitly by the compiler.
Buffer overruns	
Reason	Problem resolved by
- Incomplete, imprecise or incorrect algorithm	The raw memory is not directl accessible while the memory atom methods are tuned, free o errors and using defensive code
- Error in the algorithm implementation	The raw memory is not directl accessible while the memory atom methods are tuned, free o errors and using defensive code
Program crash – since this condition is consequence from the above erroneous conditions, and since they do not occur when using the Atomic Memory Model, this condition does not occur either when using the Ator Memory Model.	

Additional Functionality

Resolving the erroneous conditions when using memory is not the only benefit that one might receive when using the Atomic Memory Model. To add additional value to a particular implementation of the Atomic Memory Model one can use inheritance and define a set of useful common mandatory operations that all types of memory atoms will support in addition to the private for their semantics. A common abstract parent could enforce that common interface adding polymorphic capabilities to all memory atom types and defining a family of types. This also significantly simplifies the development and maintenance of different types of memory atoms. Some operations declared (and defined – if possible, depending on the operation) in a common abstract parent could be:

- Construction and destruction
  - Some constructor – depending on the implementation
  - Virtual destructor
- Memory Allocation operations – abstract methods
  - Operator =
  - Allocate
  - Re-Allocate
  - Release
  - Reallocate Transfer
  - Empty
  - Clone
  - ...
- Capability Check Operations – abstract methods
  - Can Reallocate
  - ...
- Logical Operations
  - Operator ==
  - Operator !=
  - ...
- Status Operations
  - Get Size In Bytes
  - Get Size In Units
  - Get Size
  - Get Size Unit
  - Get Size Unit Cast
  - Is Empty

  - ...
- Access Operations
  - Get Memory
  - Operator []
  - Operator [class] ()
  - Get Item Cast
  - Get Item Offset Cast
  - ...
- Casting Operations
  - Appear As
  - ...
- Memory Units Operations
  - Invert
  - ^=
  - Shift Left
  - Shift Right
  - Fill Noise
  - Memory Set
  - ...
- Search Operations
  - Find
  - ...
- Identity Operations
  - Signature Operations
  - ...
- Storage Operations
  - Load File
  - Load Resource
  - Store to File
  - Store to Resource
  - Add to File
  - ...

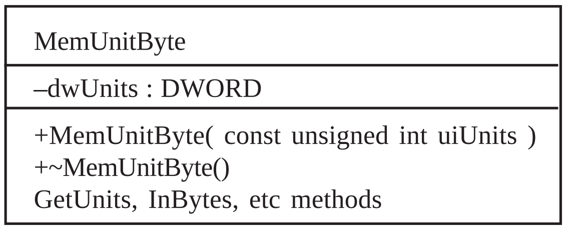
These are only a few example operations and types of operations that might be useful and which could be declared in a common ancestor Memory Atom. The particular operations and their implementation, depend on the particular implementation of the Atomic Memory Model and on how the hierarchy of Memory Atom types is constructed, and upon the general design considerations.

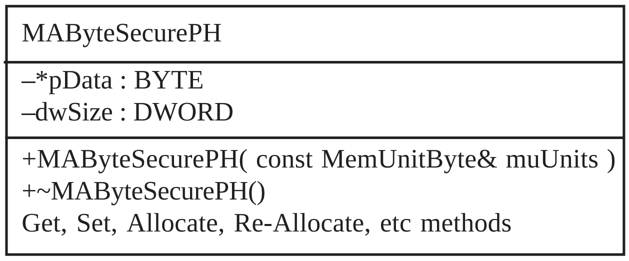
Example Implementations (Pseudo Code)

The types of memory atoms that could be defined and implemented is subject of general design considerations based on particular conditions and needs. Any program written in any language that intrinsically supports construction/destruction functionality can benefit from using the Atomic Memory Model. An example for the most basic Memory Atom in UML and a pseudo language similar to C++ but for simplicity not supporting inheritance and parameterized types is shown below. Not using inheritance, and parameterized types forces an artificial expression (instead of intrinsically embedded) of the properties of the type via its name. In this example the memory atom will contain BYTEs, have „secure“ semantics and use memory from the process heap - it’s name will be MAByteSecurePH:

- Class – „Byte“ – holds byte data
- Semantics – „Secure“ – implements „secure“ behavior i.e. erase the memory before freeing
- Origin – „PH“ – uses HeapAlloc/HeapFree in conjunction with GetProcessHeap

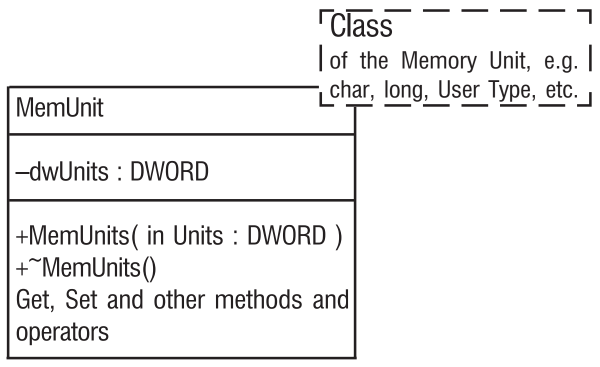
The „MA“ prefix stands for Memory Atom. There is no formal way to guarantee that the content and behavior of the Atom will correspond to the properties as they are declared.

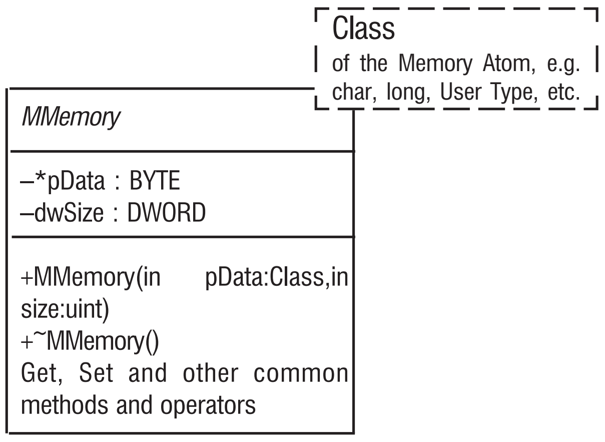

MemUnitByte
–dwUnits : DWORD
+MemUnitByte(const unsigned int uiUnits)
+~MemUnitByte()
GetUnits, InBytes, etc methods

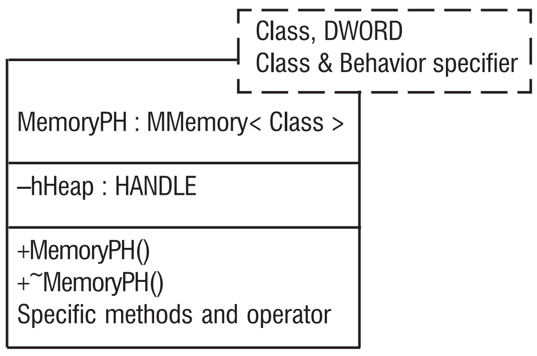

MAByteSecurePH
–*pData : BYTE
–dwSize : DWORD
+MAByteSecurePH(const MemUnitByte& muUnits)
+~MAByteSecurePH()
Get, Set, Allocate, Re-Allocate, etc methods

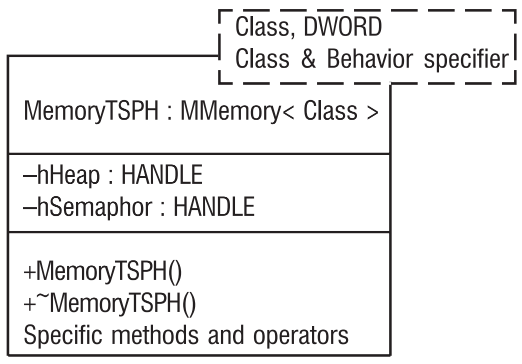
```
class MemUnitByte
{
private:
   DWORD dwSize;
public:
   MemUnitByte( const unsigned int uiUnits ) : dwSize( uiUnits )
   {
   }
   ~MemUnitByte()
   {
   }
  DWORD GetUnits() const
   {
      return( dwSize );
   }
   DWORD InBytes() const
    {
      return( dwSize * sizeof( BYTE ) );
    }
   other methods ...
};
class MAByteSecurePH
{
private:
   BYTE *pData;
   DWORD dwSize;
public:
   MAByteSecurePH( const MemUnitByte& muUnits )
    {
      pData = HeapAlloc( GetProcessHeap(), 0, dwSize = muUnits.InBytes());
    }
   ~MAByteSecurePH()
  {
     SecureZeroMemory( pData, dwSize );
     HeapFree( GetProcessHeap(), 0, pData );
  }
  Get, Set, Allocate, Re-Allocate, etc methods ...
};
```

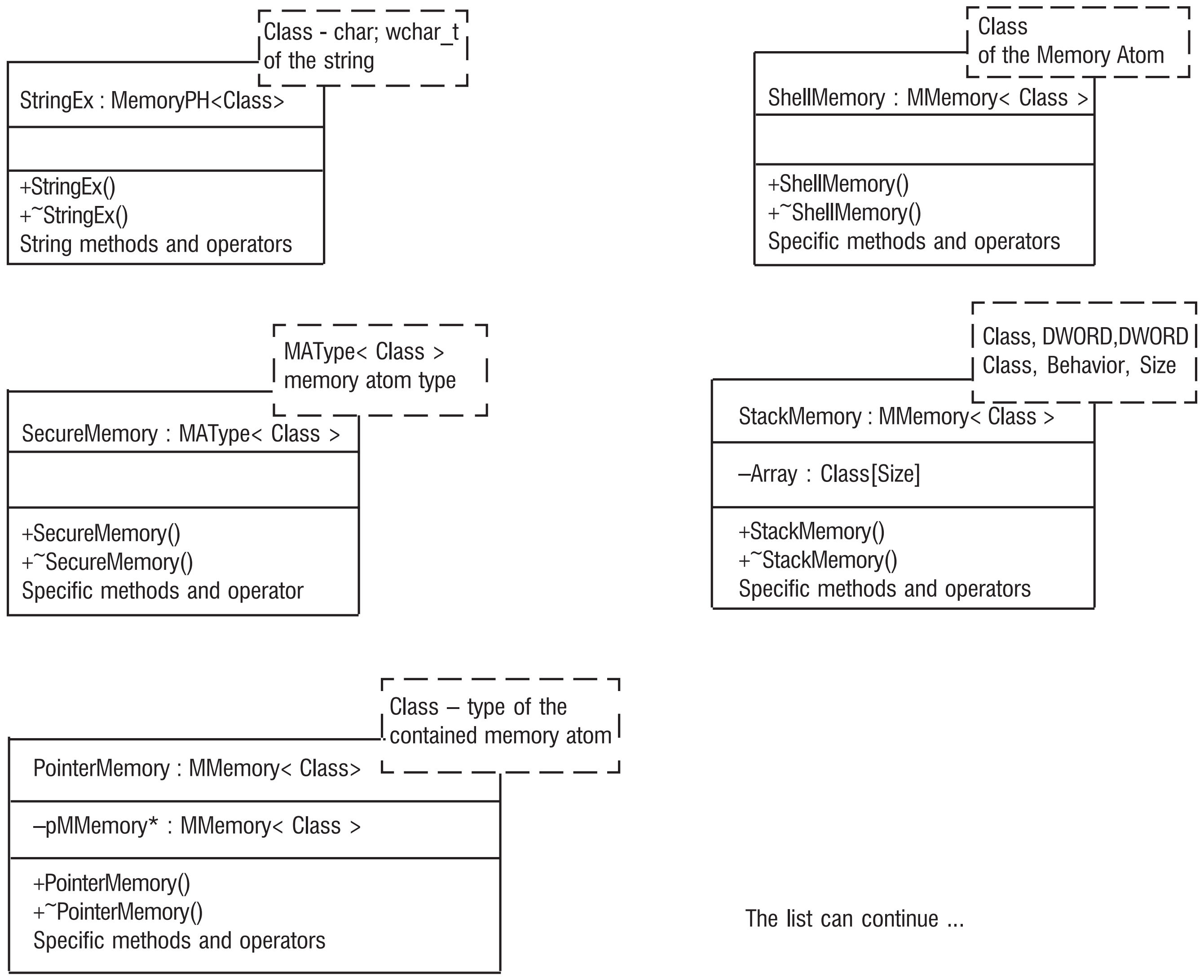
When using Object Oriented language such as C++, which was the language in which the Atomic Memory Model was developed some of the properties can be guaranteed. This is achieved by using templates, inheritance and polymorphism. The template guarantees that the class property will be exactly „as declared“. The inheritance and polymorphism ensure that some of the semantics property at least the inherited part will be into place as declared. There is no guarantee that these will be sufficient for integrity of the type – however once ensured it is so. An example of a more advanced hierarchy of memory atoms in UML and C++ is:


Class
of the Memory Unit, e.g. char, long, User Type, etc.
MemUnit
–dwUnits : DWORD
+MemUnits(in Units : DWORD)
+~MemUnits()
Get, Set and other methods and operators


Class
of the Memory Atom, e.g. char, long, User Type, etc.
MMemory
–*pData : BYTE
–dwSize : DWORD
+MMemory(in pData:Class,in size:uint)
+~MMemory()
Get, Set and other common methods and operators


Class, DWORD
Class & Behavior specifier
MemoryPH : MMemory< Class >
–hHeap : HANDLE
+MemoryPH()
+~MemoryPH()
Specific methods and operator


Class, DWORD
Class & Behavior specifier
MemoryTSPH : MMemory< Class >
–hHeap : HANDLE
–hSemaphor : HANDLE
+MemoryTSPH()
+~MemoryTSPH()
Specific methods and operators


Class - char; wchar_t of the string
StringEx : MemoryPH<Class>
+StringEx()
+~StringEx()
String methods and operators
Class of the Memory Atom
ShellMemory : MMemory< Class >
+ShellMemory()
+~ShellMemory()
Specific methods and operators
MAType< Class > memory atom type
SecureMemory : MAType< Class >
+SecureMemory()
+~SecureMemory()
Specific methods and operator
Class, DWORD,DWORD Class, Behavior, Size
StackMemory : MMemory< Class >
–Array : Class[Size]
+StackMemory()
+~StackMemory()
Specific methods and operators
Class – type of the contained memory atom
PointerMemory : MMemory< Class>
–pMMemory* : MMemory< Class >
+PointerMemory()
+~PointerMemory()
Specific methods and operators
The list can continue ...

Sketching this hierarchy of Types of Memory Atoms using C++ further demonstrates how the template specialization and type inheritance complement each other in a powerful and elegant model.

```
template< class tMemType >
class MemUnit
{
  // This type is used in all methods referring to memory units.
  // No dimensionless types such as signed/unsigned int, long,
etc are used when referring to memory units.
  // Methods for construction, destruction, compare, adding, con-
version, etc.
}

template< class tMemType >
class MMemory
{
  // Defines default behavior for all descendent ...
}

template< class tMemType, DWORD dwMemFlags >
class MemoryPH : public MMemory< tMemType >
{
  // Conforms to the default behavior, uses the Process Heap.
}

template< class tMemType, DWORD dwMemFlags >
class MemoryTSPH : public MMemory< tMemType >
{
  // Conforms to the default behavior and thread safe, uses the
Process Heap.
}

template< class tChar, DWORD dwMemFlags >
class StringEx : public MMemoryPH< tChar, dwMemFlags >
{
  // Conforms to the default behavior and a proper string class,
uses the Process Heap.
}

template< class MMemoryType, class tMemType >
class SecureMemory : public MMemoryType
{
  // Adds semantics to MMemoryType removing any sensible
information at memory release.
}
```

```
template< class tMemType, DWORD dwMemFlags >
class ShellMemory : public MMemory< tMemType >
{
  // Points to memory – not responsible for allocation, release,
etc ...
}

template< class tMemType, DWORD dwItemCount, DWORD
dwMemFlags >
class StackMemory : public MMemory< tMemType >
{
  // Conforms to the default behavior, uses the stack.
}

template< class tMemType >
class PointerMemory : public MMemory< tMemType >
{
// Conforms to the default behavior, holds a Memory Atom Ob-
ject.
}
```

The list can continue. An example use for some of the these memory atoms is:

```
// Construct an empty memory object.
MMemoryPH< DWORD, HEAP_ZERO_MEMORY > memObject1;

// Construct memory object holding 10 DWORDs.
MMemoryPH< DWORD, HEAP_ZERO_MEMORY > memObject2(
MemUnit< DWORD >( 10 ) );

// Construct a Space of Existence for memory objects.
MList< MMemoryPH< DWORD, HEAP_ZERO_MEMORY > >
listSomeMemorySpaceOfExistence;
```

The hierarchy of types of the memory atom that one may build will be their decision based on their needs and particular circumstances. The important points from the Atomic Memory Model perspective is complying with the five rules of the model, the first of which is the encapsulation of the memory into abstract entities and defining the *Type* of these entities through the three properties of *Class*, *Semantics* and *Origin*.For a fully working commercially used implementation examples of the Atomic Memory Model refer to the program code Implementation Example „Phase One“ and Implementation Example „Phase Two“ which supplements this thesis.

The Benefits

The benefits of using the Atomic Memory Model are:

1. **Resolved problems and issues** – the Atomic Memory Model successfully addresses all known issues related to memory use in a digital computing system as explained earlier, namely:
   - **Memory leaks;**
   - **Buffer overruns;**
   - **Program crash** – due one or more of the above.
2. **Application Performance:**
   - **Speed** – the effect of using the Atomic Memory Model may vary from insignificant slowdown to significant speedup. One might observe:
     - Insignificant performance overhead due to calls to methods for construction, access and destruction of memory atom objects – consider in-lining to improve performance.
     - Significant performance improvement, since function pre-calls are no longer needed and any such calls are removed. Function pre-call is a means for a caller to interrogate a function as to how much memory it will require to perform its action using a specific set of parameters, thus the caller supplies memory that is needed by the function (service). Traditionally the caller calls the function once with a flag indicating that no memory is being passed, meaning that it only asks for the size of the required memory. Then it calls again passing a pointer to memory with the required size. When using the Atomic Memory Model one passes a reference to a Memory Atom to the function. The function requests the Memory Atom to allocate the necessary space and uses it through the Memory Atom methods, thus function pre-call is no longer needed.
   - **Memory use** – depending on the use of in-lining and the size of the Memory Atoms:
     - There will be somewhat increase of the size of the compiled code due to the bodies of the methods of the memory atoms– especially when in-lining.
     - The use of RAM will be insignificantly increased from 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.
3. **Development:**
   - The Atomic Memory Model guarantees concise and tidy program code related to memory allocation, access (use) and de-allocated.
   - 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 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 reason is never memory leaks as they clearly do not occur.

4. Garbage collection becomes a void concept. Some software systems rely on garbage collection to free unreferenced memory. When using the Atomic Memory Model the concept for garbage collection becomes void. Using the Atomic Memory Model over garbage collection improves both system performance and system memory usage.

Conclusion

The Atomic Memory Model is an extremely powerful memory handling technology which enormously increases the quality of code and speed of development. The Atomic Memory Model makes concepts such as garbage collection redundant, since it combines intrinsic safety with increased performance and a more efficient use of memory. Languages featuring templates and inheritance are naturally better suited for using the Atomic Memory Model than others without these features: however the minimum prerequisite a language must possess in order to be able to implement the Atomic Memory model, is the capacity to define entities (user types) and to make implicit destructor calls regarding those entities.

Example implementations of the method can be downloaded from http://www.mbbsoft.com/Software/AtomicMemoryModel/Download.aspx - MIT Open Source License.

Reference

1. Bogdanov, D., I. Mustakerov. Ezik za programirane C. Sofia, Tehnika. 2003.(in Bulgarian)

2. Lukas, Paul J. The C++ Programmer's Handbook. Prince Hall Inc., 1992.

3. Louis, Dirk. C/C++. Prentice Hall, 2001.

4. Stroustrup, Bjarne. The C++ Programming Language. Addison-Wesley, 2000.

5. Fowler, Martin. UML Distilled, Pearson, 2003.

Manuscript received on 05.08. 2007

Miroslav B. Bonchev, *was born in Gabrovo, Bulgaria. Graduated M.Sc. in Automation and Computer Technique and Technology at the Technical University of Gabrovo, specializations in Control Systems and Computer Networks. Worked as software engineer, hardware designer, software architect and head of development in Bulgaria and Great Britain for companies such as Pitney Bowes and others. Author of the popular volume control utility Audio Control, the new coming Act On File and founder of MBBSoft. Currently, M. Sci. student in pure mathematics at Queen Mary University of London.*

Contacts:

e-mail: mbb@mbbsoft.com