SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:977–1009 (DOI: 10.1002/spe.600)

The implementation of generic smart pointers for advanced defensive programming

Anthony Savidis*,†

Institute of Computer Science, Foundation for Research & Technology—Hellas, Heraklion, Crete, GR-71110, Greece

SUMMARY

Smart pointers denote a well-known technique for collective resource ownership, e.g. sharing dynamic object instances, while usually supporting automatic garbage collection based on reference counting. The original method has been retargeted to serve as a generic defensive programming method for 'exhaustive tracking' of erroneous pointer use in C++. Automatic bug tracking is supported in a unified manner both for pointers to heap memory, i.e. free storage, as well as for pointers to stack or global memory, i.e. auto or static storage. Overall, the presented technique (a) offers a simple contract for memory allocation and use; (b) supports type and indirection depth genericity; (c) implements most operators supported for built-in pointers with embedded bug defense; (d) offers an alternative way of employing a garbage collection facility for memory leak detection; and (e) provides an appropriate collection of utility macros, through which defensive pointers should be used, with an alternative version re-targeted to normal native pointers. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: smart pointers; defensive programming; garbage collection; memory management; recursive meta-programming

INTRODUCTION

In defensive programming, software libraries, design patterns and coding styles are developed that enable programmers to minimize the time interval between (a) the point of execution at which the symptoms/effects of errors, inherent in the software design and coding processes, become somehow observable (e.g. exceptions, malfunctions, data corruption, diagnostic messages, etc); and (b) the exact time of execution of the first offensive statement within the source code that actually causes such errors. Through the employment of advanced programming techniques, capable of turning this time distance to zero, defensive programming aims to *direct defect detection* (i.e. to trap the defect exactly when it is caused). In C++, a large proportion of painful and hardly detectable programming errors are related to pointers, especially when it comes to large-scale software systems. Pointer-use errors

*Correspondence to: Anthony Savidis, Institute of Computer Science, Foundation for Research & Technology—Hellas, Heraklion, Crete, GR-71110, Greece.
†E-mail: as@ics.forth.gr

Received 21 February 2003
Revised 1 August 2003
Accepted 27 November 2003
Published online 21 May 2004
Copyright © 2004 John Wiley & Sons, Ltd.

typically cause illegal memory accesses, leading to a chain of dangerous unpredictable side effects (i.e. the *domino* effect), which quickly turn the identification of the original malicious code to a painful and time demanding task.

In this context, this paper shows how smart pointers can be effectively exploited as a fully implemented design pattern that aims to automatically track down all erroneous uses of pointers (e.g. out of bounds addressing, pointer overrun, null pointer use, double delete, etc.), without sacrificing their original programming convenience and flexibility. Additionally, the paper discusses in detail how the proposed defensive smart pointers encapsulate logic for automatically releasing unused (i.e. unreferenced) dynamic memory, through algorithms primarily based on reference counting, while also resolving cyclic references. In the proposed implementation, smart pointers are typical C++ template classes, which overload most operators that can be used with built-in pointers, while encapsulating error checking and memory management. A complete technique for the implementation of generic defensive smart pointers is discussed, where pointers can only relate to data addresses; a technique to support the use of pointers in member functions, in conjunction with typical smart pointers, is presented in Meyers [1].

There are various design patterns demonstrating the implementation of smart pointer classes [2], as well as existing libraries that offer numerous smart pointer templates for different situations, for example, the Boost library [3,4]. Most of those design patterns offer comprehensive facilities for collective resource ownership, enabling automatic lifetime control for a shared embedded dynamic instance, by usually supporting direct disposal through reference counting policies (e.g. Boost `shared_ptr`). However, in general, they suffer from the following key problems.

(a) The supplied template classes support type genericity, but no indirection depth genericity, meaning that the programmer has to manually specify depth recursion in template instantiations (e.g. `SmartPtr< SmartPtr< SmartPtr<T> > >` for `T***`), thus severely reducing the syntactic ease of use in pointer declarations.
(b) In most approaches, only dynamic memory can be assigned to smart pointer classes (like in the Boost `shared_ptr` template class), thus not enabling programmers to treat pointers to heap, stack and static memory in a uniform way. Such a feature is particularly important when smart pointers are to play a defensive role.
(c) In existing patterns, there are no concrete implementation guidelines for algorithmically enhancing or revisiting the basic reference counting algorithm, which has proven to be inadequate to handle cyclic memory dependencies.
(d) They do not provide the means for flexible deployment, for example, allowing programmer-controlled memory disposal for smart pointers (i.e. explicitly deleting memory even while garbage collection is active), or automatically turning smart pointers into normal pointers in production mode. The latter process requires that garbage collection is activated only for memory leak detection.

The implementation technique presented in this paper enables effective programming defense against the following frequently occurring problems:

- use of already deleted memory;
- use of un-initialized pointers;
- use of pointers in popped stack memory;

Copyright © 2004 John Wiley & Sons, Ltd.

- exceeding memory bounds in read/write;
- out of bounds memory addressing for arrays;
- deleting memory with a pointer that does not point to the beginning of the allocated memory;
- memory leaks (i.e. referenceable unused dynamically allocated user memory, forgotten to be deleted), and lack of memory referenceability (i.e. memory is dynamically allocated, but after a certain runtime point there are no referring pointers to dispose of it);
- deletion of memory with a wrong operator (e.g. `delete` when `delete[]` should be used, and *vice versa*).

Some additional benefits of a fully controlled generic smart-pointer library in a defensive programming perspective include the ability to enable programmers (a) to encompass runtime memory access statistics (e.g. global distribution of memory address access); (b) to monitor pointer operations in time critical units (e.g. count pointer de-references, or total pointer uses, in performance-critical code); and (c) to extend the defensive library by providing thread-safe versions, e.g. through the use of semaphores/mutexes.

The dual role of garbage collection in defensive smart pointers

In the developed library, the garbage collection mechanism can be used in two mutually exclusive ways: (i) as a mechanism of detecting either memory leaks at program termination, or the loss of references to dynamic memory during runtime, meaning that programmers are required to design with explicit memory disposal; and (ii) as a built-in standard functional feature, meaning that programmers are relieved from the burden of manual lifetime control of dynamic memory.

The first use of garbage collection is less intuitive and common than the second. However, it can be employed as a valuable instrument for detecting 'less than prefect' memory management. In the context of the developed method, memory leaks mainly concern dynamic memory not explicitly disposed of by the program after normal termination, even though at that point the necessary pointer variables are available. This information is crucial for independent unit testing in test-fist programming, where units should ensure that any unused resources allocated upon finishing-up their processing are thoroughly free. Additionally, lack of memory referenceability indicates a logical programming error, taking place at a particular point during execution, due to which the program modifies those pointers that would allow explicit memory disposal later on.

EXAMPLES OF USE FOR DEFENSIVE SMART POINTERS

Before proceeding with the detailed presentation of the implementation approach for defensive smart pointers, some representative examples demonstrating various required source code modifications inherent from their employment will be presented. This will provide an overview of both the suggested use style, which, as it will be shown, can be extended/modified through the use of shortcut type definitions and helper macros, and of the potential code-readability effects introduced.

In Figure 1, an example of constructing a template `TreeNode` class is shown, employing defensive smart pointers. The special-purpose macros that hide the more complex declarations are shown in bold typeface. It should be noted that the use of defensive smart pointers is only supported through special-purpose utility macros enabling easier deployment (or otherwise, they tend to be syntactically very

Copyright © 2004 John Wiley & Sons, Ltd.

```
template <class T> class TreeNode {
public:
    typedef DSDECL(TreeNode*) TreeNodeSPtr;
private:
    TreeNodeSPtr left, right;
    TreeNodeSPtr parent, firstChild, lastChild;
    T val;
public:
    void SampleMemberFunc (void) {}
    T& operator*(void) { return val; }
    void Disconnect (void) { // Disconnects sub tree from parent.
        if (DSTRUE(parent)){
            if (parent->firstChild == DSTHIS)
                parent->firstChild = parent->firstChild->right;
            if (parent->lastChild == DSTHIS)
                parent->lastChild = parent->lastChild->left;
        }
        left = right = parent = DSNULL(TreeNode*);
    }
    void InsertChildAtBegin (TreeNodeSPtr& node) {
        if (node == DSTHIS)
            return;
        node->Disconnect(); // In case it is still in some tree.
        if (!firstChild)
            firstChild = lastChild = node;
        else {
            node->right = firstChild;
            firstChild->left = node;
            firstChild = node;
        }
    }
    template <class PrintFunctor>
    void Print (const PrintFunctor& f) { // DFS print.
        if (DSTRUE(firstChild)) // firstChild != DSNULL(TreeNode*)
            firstChild->Print(f);
        if (DSTRUE(right)) // right != DSNULL(TreeNode*)
            right->Print(f);
        f(val);
    }
    TreeNode (T _val) : val(_val){}
    ~TreeNode(){}
};

class PrintIntFunctor {
    public:
    void operator()(int x) const { printf(" %d ", x); }
};

TreeNode<int>::TreeNodeSPtr node1 = DSNEWVAR(new TreeNode<int>(10));
TreeNode<int>::TreeNodeSPtr node2 = DSNEWVAR(new TreeNode<int>(45));

void (TreeNode<int>::*pmf)(void) = &TreeNode<int>:: SampleMemberFunc;
(DSNATIVE(node1)->*pmf)();

node1->InsertChildAtBegin(node2);
node1->Print(PrintIntFunctor());
```

Figure 1. Examples of deployment for defensive smart pointers, in the context of a tree node template class.

Copyright © 2004 John Wiley & Sons, Ltd.

overloaded and clumsy). A normal pointer type T is transformed to its corresponding smart pointer class via `DSDECL(T)`, which works for any indirection depth and pointee type (with the exception of the `void` type). Inside the `DSDECL` macro, as will be explained later in more detail, a recursive template class is employed to extract the particular pure pointee type C and the indirection depth N, which are then used as the actual parameters in the instantiation of the basic `SmartPtr` template class (i.e. `SmartPtr<C,N>`).

Inside class member functions, the expression `DSTHIS` produces a temporary constant smart pointer instance, of the corresponding class, which is to be used as a smart pointer version for the '`this`' pointer. When smart pointer variables are to be used as Boolean expressions, in contrast to normal pointers, they cannot be directly converted to a Boolean value following the 'non-zero means true' rule. For this purpose, to apply a pointer 'true test', the `DSTRUE` macro should be used (as shown in Figure 1), while to apply a pointer 'false test', the original syntax employing the '`!`' operator is directly applicable (e.g. as in the `!firstChild` expression). The use of null pointer expressions requires decoration of the particular pointer type with the `DSNULL` macro, which returns a constant smart pointer instance representing a strongly typed null pointer in the smart pointer library; hence, following the example of Figure 1, the null pointer expression `(TreeNode*) 0` should be written as `DSNULL(TreeNode*)`. Alternatively, it is possible to define one special-purpose null type, like `class dsnull{};`, and provide appropriate versions for the smart pointer class constructor, assignment operator and equality operator, while defining `DSNULL` as `dsnull()`.

In Figure 1, the `DSNEWVAR` 'dynamic allocation adapter' macro is also shown, which should 'surround' all dynamic allocation expressions for simple pointers (i.e. pointers of depth 1), so as to transform them to the appropriate corresponding smart pointer instances. As it is shown, the original syntactic structure of `new` expressions, including the constructor actual parameter passing style, is not affected. The only addition concerns the 'decoration' with the `DSNEWVAR` macro. The two local variables `node1` and `node2` are defined to be of type `TreeNode<int>::TreeNodeSPtr`, following the preceding `typedef` for `TreeNodeSPtr`. Alternatively and equivalently, `DSDECL(TreeNode<int>*)` could be directly used as the type specifier. Finally, since in the presented smart pointer library there is no support for the operator `->*` (calling through pointers to member functions), a smart pointer needs to be converted to the native corresponding C++ pointer, via the `DSNATIVE` macro, so that the operator `->*` can be normally applied.

The next section of the paper presents the key aspects of the adopted software architecture, regarding defensive smart pointer classes, implementation of monitorable reference-aware memory objects, and garbage collection with cycle elimination. Additionally, smaller and more focused examples will be introduced, demonstrating the detailed use features of defensive smart pointers. Overall, smart pointer classes implement in a defensive fashion most pointer operators, such as `+`, `-`, `-=`, `+=`, `++` (pre-increment, post-increment), `--` (pre-decrement, post-decrement), `*` (de-reference), `->`, `[]`, `=`, `==`, `!=`, and the `!` operator.

OVERVIEW OF THE SOFTWARE ARCHITECTURE

In known existing implementations, such as `std::auto_ptr` and Boost `shared_ptr`, as well as in various approaches to the parameterization of resource clean-up methods, such as the one

Copyright © 2004 John Wiley & Sons, Ltd.

suggested for auto_ptr [5], the 'native' pointer to the shared dynamic resource instance is hosted inside the smart pointer class along with the reference counting logic. In the work presented here, in order to support arbitrary sharing and direct cycle elimination, in contrast to delayed/relaxed garbage collection which is initiated later at execution time, usually in regular scheduled intervals, it has been decided to maintain, for each shared resource item, appropriate runtime information regarding *the particular smart pointer instances referring to it*. Implementing appropriate adapter classes for shared memory items, encapsulating the desirable runtime information, while changing smart pointer classes to internally encompass a pointer to such memory-item adapter instances is considered to be an elegant design tactic.

In Figure 2, outlining the key classes and relationships in the proposed architecture, the MemoryObjectSuper class is the super-class of the memory-item adapter class. As it is shown, it encompasses information regarding:

- all referring smart pointers—referTome, which, as it is shown in Figure 2, is reflected through an '$N : 1$' relationship between the SmartPtrSuper class and MemoryObjectSuper::referToMe member;
- reference counter—refCounter;
- size (in typed items, not in bytes) of its associated memory item—size;
- whether its associated memory item is known to be the product of a 'new' expression—isDynamic;
- whether it has been explicitly deleted by the programmer (allowed only if isDynamic is true—wasExplicitlyDeleted); and
- whether it has not yet entered an explicit deletion or collection process—alive.

The reference counting logic for garbage collection together with the algorithm for cycle elimination are implemented inside the MemoryObjectSuper class. The derived template class, named MemoryObject<T>, encompasses a T* userMem native pointer to the referring memory item, and supplies the memory disposal function (i.e. the deletion of a heap memory item upon explicit destruction or garbage collection).

One of the key features of defensive smart pointers, playing a central role in cycle-free garbage collection, is their awareness for physical containment within pointed user memory. This feature is reflected in the owner member of the SmartPtrSuper class, which, for any particular smart pointer instance, is set to be the address of that derived-class memory object (stored as a base MemoryObjectSuper* pointer), whose user memory (i.e. userMem) physically incorporates the complete smart pointer instance. If no such object exists, owner is set to null. In other words, if the memory area occupied by a smart-pointer instance is physically embedded in the user-memory of a particular memory object, then this object is said to 'own' the smart pointer instance. A single memory object may own multiple smart pointers (in Figure 2, this is indicated through the '$N : 1$' relationship between SmartPtrSuper::owner member and the MemoryObjectSuper class).

The implementation of the core functionality for defensive smart pointers is provided within the SmartPtrImpl template class, derived from the SmartPtrSuper class. In SmartPtrImpl, the single template parameter T represents the pointee type. The data members that are encompassed in this class include: (a) a pointer of type MemoryObject<T>* to the referred memory-object instance, stored in the memObj variable; and (b) the current position of the smart pointer instance within the pointed physical user memory (i.e. memObj->userMem + pos), reflecting the result of typical T*

Copyright © 2004 John Wiley & Sons, Ltd.

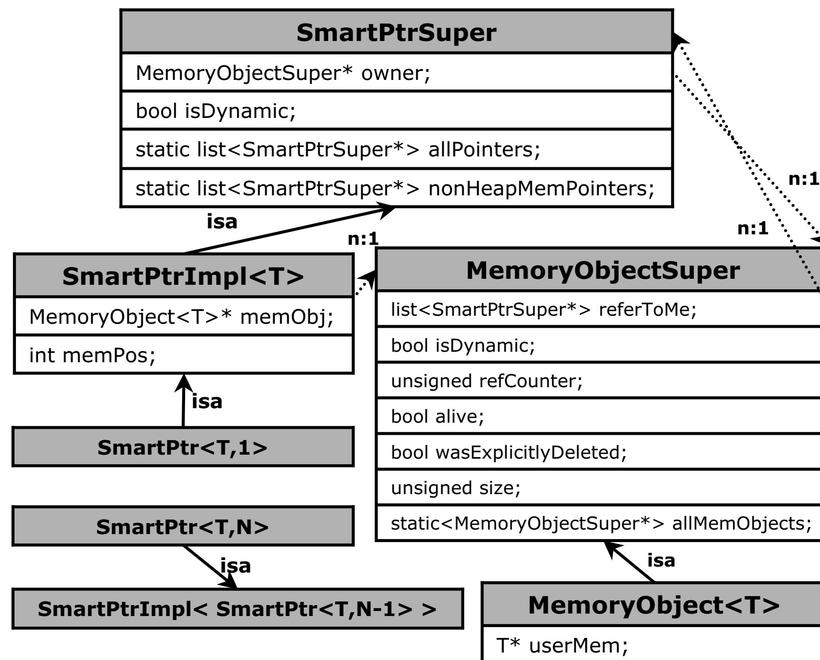

Figure 2. Overview of the key classes, showing the most important data members, and their relationships; dashed arrows represent runtime associations (all of n:1 cardinality).

pointer arithmetic operations applied for the smart pointer instance. The latter is necessary in order to facilitate the flexible use of smart pointers as arrays of continuous items, by supporting standard arithmetic operations and array-style indexing, while also embedding defensive behavior.

Finally, there is one key recursive template class, `SmartPtr`, which is defined to allow easier syntactic deployment of smart pointers, by supporting indirection depth parameterization. As discussed in the next section, another special-purpose recursive template class is also defined, `PtrInfo`, which is used to extract from built-in pointer types (i) the pure pointee type `T`, and (ii) the indirection depth `N` as an `unsigned` number. The combination of those two template classes allows syntactically simple and clean type-specifiers for defensive smart pointers, e.g. `DSDECL(int***)`, to be directly supported. It should be noted that the `PtrInfo` strips off embedded `const` qualifiers (e.g. for a `T*const*const*` only `const T` and `***` will be extracted). As a consequence, the presented template library is not appropriate in its present form for such pointer types.

RECURSIVE TEMPLATES FOR TYPE AND DEPTH GENERICITY

The issue of type and indirection depth genericity will be briefly addressed by presenting the main template-based smart pointer class in which both the basic pointee type `T` and indirection

Copyright © 2004 John Wiley & Sons, Ltd.

```
template <class T, const unsigned char N>
class SmartPtr : public SmartPtrImpl< SmartPtr<T, N-1> > {
    public:
    const SmartPtr& operator= (
        const SmartPtrImpl< SmartPtr<T, N-1> >& ptr
    );
    static const SmartPtr Null (void) { return SmartPtr(); }
    SmartPtr (void){}
    SmartPtr (const SmartPtrImpl< SmartPtr<T, N-1> >& ptr);
    SmartPtr (
        SmartPtr<T, N-1> *attachedMem,
        unsigned int size,
        bool         isDynamic,
        char*        srcFile,
        unsigned int srcLine
    );
    ~SmartPtr(){}
};

template <class T>
class SmartPtr<T, 1> : public SmartPtrImpl<T> {
public:
    const SmartPtr& operator= (const SmartPtrImpl<T>& ptr);
    static const SmartPtr Null (void) { return SmartPtr(); }
    SmartPtr (void){}
    SmartPtr (const SmartPtrImpl<T>& ptr) : SmartPtrImpl<T>(ptr) {}
    SmartPtr (
        T*           attachedMem,
        unsigned int size,
        bool         isDynamic,
        char*        srcFile,
        unsigned int srcLine
    )
    ~SmartPtr(){}
};

template <class T> class PtrInfo {
    public:
    enum { n = 0 };
    typedef T type;
};

template <class T> class PtrInfo<T*> {
    public:
    enum { n = PtrInfo<T>::n + 1 };
    typedef typename PtrInfo<T>::type type;
};

#define DSDECL(T) SmartPtr< PtrInfo<T>::type, PtrInfo<T>::n >
```

Figure 3. The two key recursive template classes, i.e. `SmartPtr` and `PtrInfo`, used inside the `DSDECL` native-type decoration macro, enabling compact defensive smart pointer declarations.

Copyright © 2004 John Wiley & Sons, Ltd.

```
#ifdef DSMARTPOINTERS

template <class T>
DSDECL(T*) _DSADDRESSOF (T& x, char* file, unsigned ln) {
    return DSDECL(T*)(&x, 1, false, file, ln);
}

template <class T>
DSDECL(T*) _DSNEWARR (T* expr, unsigned n, char* file, unsigned ln) {
    return DSDECL(T*)(expr, n, true, file, ln);
}

template <class T>
DSDECL(T*) _DSNEWVAR (T* newExpr, char* file, unsigned ln) {
    return DSDECL(T*)(newExpr, 1, true, file, ln) ;
}

template <class T>
const DSDECL(T*) _DSTHIS (T* p, char* file, unsigned ln) {
    return DSDECL(T*)(p, 1, false, file, ln);
}

// "&x" becomes "DSADDRESSOF(x)"
// "new int[10]" becomes "DSNEWARR(int, 10)"
// "new X(x)" becomes "DSNEWVAR(new X(x))"
// "this" becomes "DSTHIS"

#define DSADDRESSOF(x) _DADDRESSOF(x, __FILE__, __LINE__)
#define DSNEWARR(T, n) _DSNEWARR(new T[n], n, __FILE__, __LINE__)
#define DSNEWVAR(expr) _DSNEWVAR(expr, __FILE__, __LINE__)
#define DSTHIS         _DSTHIS(this, __FILE__, __LINE__)

#else

#define DSADDRESSOF(lvalue) &lvalue
#define DSNEWARR(T, n)      new T[n]
#define DSNEWVAR(expr)      expr
#define DSTHIS              this
#endif
```

Figure 4. Helper template functions producing smart pointer instances for various commonly used pointer expressions; when smart pointers are not employed, the function names correspond to macros producing native pointer expressions.

Copyright © 2004 John Wiley & Sons, Ltd.

depth `N` become template parameters. In this template class, instead of the explicit unrolled recursive declarations of the form `SmartPtr<SmartPtr<...<T> > >`, needed in existing implementations, corresponding to built-in pointer type `T*...*` with indirection depth `N`, the programmer may only specify `SmartPtr<T, N>`.

The recursion logic for `SmartPtr` template class is outlined in Figure 2, while the definition of the template class is supplied in Figure 3. As depicted in Figures 2 and 3, a `SmartPtr<T,1>` class for pointee type `T` is a smart pointer implementation template instantiated for type `T` (i.e. `SmartPtrImpl<T>`). Similarly, for the inductive step of recursion, a `SmartPtr<T, N>` class (i.e. a pointer of depth `N`) is a smart pointer implementation instantiated for type `SmartPtr<T,N-1>` (i.e. a pointer to a pointer of depth `N-1`), that is `SmartPtrImpl < SmartPtr<T, N-1> >`.

The implementation of Figure 3 employs partial template specialization [6] together with recursive template definitions, reflecting a technique known as recursive meta-programming [7]. The only member functions that need to be implemented in the context of the `SmartPtr` template class are:

- the default- and copy-constructors;
- the overloaded assignment operator;
- the `Null` static function returning a temporary const instance semantically behaving as a strongly typed null pointer; and
- a special-purpose parameterized constructor, to be discussed later, for assigning pointed user memory to a smart pointer; it should be noted that, as expected from the previous recursive definitions, while for class `SmartPtr<T,1>` the attached user memory is of type `T*`, for `SmartPtr<T,N>` it is of type `SmartPtr<T,N-1>*`.

Finally, to relieve programmers from employing the unusual syntax of template instantiations with explicit depth qualification, it has been considered as a more appropriate solution to allow the use of the original pointer-type specifiers, e.g. `int***`, wrapped-up with a smart-pointer qualification macro, e.g. `DSDECL(int***)`, as also shown in the examples of Figure 1. In Figure 3, the recursive template class `PtrInfo` has been defined to extract the pure pointee type `type` and the total pointer indirection depth `n` from a given 'native' pointer type `T` template parameter. Using this template class, the definition of the `DSDECL(T)` macro (see Figure 4), where `T` is always a native pointer type, hides the syntactically more complex `SmartPtr` template instantiation.

DESIGN CONTRACT FOR MEMORY ALLOCATION AND USE

Before displaying all the implementation details of defensive smart pointers, the primary usage regulations are briefly specified. Such regulations are characterized as a software design contract, since programmers should obey certain deployment semantics, so that smart pointers can certify runtime behavioral correctness. The design contract for smart pointers is that (a) programmers are responsible for supplying pointed user memory with built-in pointer type expressions, by correctly qualifying memory creation origin (i.e. dynamic creation or not), as well as the contextually known size (in number of pointee-type data items); and (b) smart pointers are responsible for accessing, guarding, and disposing (when dynamic) un-referenced user memory. For commonly used expressions, such as those engaging the `this`, `new` or `new[]` operators, or the address-of operator `&`, there are helper

Copyright © 2004 John Wiley & Sons, Ltd.

macros and template functions (see Figure 4) to relieve the programmer from explicitly qualifying memory origin and re-supplying as an extra parameter the number of pointed items.

Following this contract, user memory is actually 'attached' to smart pointers. This is made possible through the provision of the special-purpose parameterized constructor of the `SmartPtr` template class (see Figure 3), accepting five parameters: (i) the attached native user memory (`attachedMem` argument); (ii) the size of user memory in pointee-type items (`size` argument); (iii) a Boolean value to qualify if the attached user-memory expression is actually a `new` expression (i.e. the user memory is created at the time the particular call is made); and (iv) the specific source file and source line in which the smart pointer expression is defined (`srcFile` and `srcLine` arguments).

This design contract introduces no particular semantic variations, in comparison to the use of built-in pointer types. However, it requires explicit qualification in the case of dynamic memory allocation expression, together with the provision of user-memory size information. This overhead is necessary in order to enable the implementation of smart pointers to handle both heap and stack memory, as well as to enable the embedding of more detailed error checking for detecting 'out of bounds' memory access situations. Most of the differences between built-in and smart pointers are hidden under the definition of appropriate utility template functions and helper macros, enabling easier deployment, while also allowing, without altering source code, to turn defensive smart pointers into built-in pointers, and *vice versa*, through the use of a pre-processor flag. The latter works *only* if garbage collection is employed for detecting memory leaks and loss of memory referenceability.

MEMORY OBJECTS WITH REFERENCE COUNTING AND GARBAGE COLLECTION

The assignment of pointed user memory directly to smart pointers is a commonly used technique in existing design patterns for smart pointers, like the one described in Alexandrescu [8]. Typically, pointed user memory is directly embedded inside smart pointer classes in the form of native-pointer private variables. In some more parameterized proposals, the type of shared resources should comply with certain trait patterns, as is the case with `auto_handle` presented in Vlascenau [5]. Most widely employed methods, like the Boost `shared_ptr`, incorporate reference counting for garbage collection inside smart pointer classes, and do not incorporate extra information for guarding erroneous accesses. Additionally, facilities for more flexible deployment of smart pointers are not supported. For example, the Boost library does not support getting the native pointer addresses (this feature is omitted since it is considered a potential source of errors), or explicitly releasing dynamically allocated memory (in the Boost library, this is a limitation of the implementation, since there is no automatic invalidation of other referring smart pointers). Other features which are not included concern book-keeping of user-memory access statistics (apart from `shared_ptr::use_count()`), the ability to also 'guard' stack/global user memory in addition to heap memory, and support for garbage collection with cycle elimination. The main reason for the lack of the previously mentioned defensive features in existing implementations is that such approaches do not have a defensive programming focus. Smart pointer classes offer 'intelligent control' of collective resource ownership, where the particular instantiation of such 'intelligent control' most likely requires delivery of different alternative implementations.

Implementation of the super class

Following the previous overview of the role of memory objects (depicted also in Figure 2), pointed user memory supplied to defensive smart pointers is always encapsulated and managed by heavyweight

Copyright © 2004 John Wiley & Sons, Ltd.

dynamically constructed memory objects. For each different user-memory block starting address, there is a unique 'guardian' memory object instance. The member functions of the `MemoryObjectSuper` class, illustrated in Figure 5, are as follows.

- The `UserMemory` pure virtual function, implemented by the derived template class `MemoryObject`, which returns the starting address of the attached user memory.
- The two reference control functions `AddReference` (adding a smart pointer to the referral list, while increasing reference counter) and `RemoveReference` (removing a smart pointer from the referral list, while decreasing reference counter).
- The reference-counting cycle-free garbage collection functions:
  - `IncRefCounter,` which increases the reference counter variable;
  - `DecRefCounter`, which collects the object if either the reference counter becomes zero, or the memory object is considered *no longer referenceable*;
  - `IsReferencable`, which is the cornerstone of the cycle-free garbage collection algorithm, and decides if an object is still referenceable by the programmer (this will be explained in detail later on).
- The explicit memory disposal function `Delete`, that first invalidates all referring pointers and then performs a brute force (but completely safe) 'self destruction' call, i.e. `delete this`. As will be shown, the real disposal of the `T* userMem` pointed user memory is performed inside the derived class destructor (i.e. `~MemoryObject<T>`). Also, the `Delete` function accepts a Boolean parameter, `bool asArray`, indicating whether the programmer explicitly qualified the disposal call as being applied to a dynamic array (i.e. a call to `delete[]`); the latter must be defined in the library via special purpose macros, enabling the detection of typical `new[]` / `delete` or `new[]` / `delete` mismatches.
- The `InvalidateReferringPointers` function, which invalidates all smart pointers that refer to the caller memory object, i.e. those in the `referToMe` list (each smart pointer calls the `SmartPtrSuper::Invalidate` member).
- The `InsideUserMemory` function, which returns `true` if a given address falls inside the user memory of the caller memory object; this function is used to identify the owner memory object (if any) for smart pointer instances.
- The `Find` static function, which returns the unique memory object instance (if any) for which the call to `UserMemory`, providing the starting address of user memory returns a value equal to the actual parameter `userMem`.

Implementation of the pointee-type parameterized derived class

The derived template class `MemoryObject` is parameterized for the pointee type `T`, while supplying the functionality that can only be implemented with pointee-type related information (see Figure 6).

- The user-memory destruction logic, implemented inside the destructor function ~ `MemoryObject`; if pointed user memory has been dynamically allocated, is appropriately deleted (based on the value of the `size` variable) either via simple `delete` or through `delete[]`.

```
class MemoryObjectSuper {
    private:
    virtual void* UserMemory (void) = 0;

    public:
    static MemoryObjectSuper* Find (void* userMem);
    virtual bool InsideUserMemory (void* addr) const = 0;
    void   AddReference (SmartPtrSuper* from)
               { referToMe.push_back(from); IncRefCounter(); }
    void   RemoveReference (SmartPtrSuper* from)
               { referToMe.remove(from); DecRefCounter(); }
    void   IncRefCounter (void) { ++refCounter; }
    void   DecRefCounter (void);
    bool   IsReferenceable (void) const;
    void   Delete (bool asArray);
    bool   IsDeleted (void) const { return wasExplicitlyDeleted; }
    void   InvalidateReferringPointers (void);

    MemoryObjectSuper(unsigned size, bool isNewExpression);
    virtual ~ MemoryObjectSuper();
};

void MemoryObjectSuper::DecRefCounter (void) {
    if (!alive)  // Destructor call is active.
        --refCounter;
    else
    if (!--refCounter || !IsReferenceable())
        { alive = false; delete this; }
}

void MemoryObjectSuper::Delete (bool asArray) {
    assert(asArray || size==1);
    assert(!asArray || size > 1);
    assert(isDynamic && alive && !wasExplicitlyDeleted);

    wasExplicitlyDeleted = true;
    alive = false;
    InvalidateReferringPointers();
    delete this;
}
```

Figure 5. The member functions of the `MemoryObjectSuper` class, mainly implementing the garbage collection algorithm, while supporting explicit destruction.

Copyright © 2004 John Wiley & Sons, Ltd.

```
template <class T> class MemoryObject : public MemoryObjectSuper {
    private:
    T* userMem;
    void* UserMemory (void) { return userMem; }

    public:
    T* GetPtr (unsigned index) {
          assert(userMem && !wasExplicitlyDeleted && alive);
          assert(index < size);
          return userMem + index;
    }

    T& Get (unsigned index) { return *GetPtr(index); }

    bool InsideUserMemory (void* addr) const {
          unsigned long d1 = (unsigned long) userMem;
          unsigned long d2 = d1 + size * sizeof(T);
          return d1 <= ((unsigned long) addr) &&
                 d2 >= ((unsigned long) addr);
    }

    MemoryObject (
        T*       _userMem,
        unsigned _size,
        bool     _isDynamic,
        char*    srcFile,
        unsigned srcLine

    ) : MemoryObjectSuper(srcFile, srcLine, _size, _isDynamic),
        userMem(_userMem)
        { SmartPtrSuper::CheckMemoryOwnership(this); }

     ~MemoryObject() {
          if (isDynamic)
              if (size > 1)
                  delete[] userMem;
              else
                  delete userMem;
       }
};
```

Figure 6. The member functions of the `MemoryObject` template derived class, mainly implementing user-memory-content-access control and user-memory disposal logic.

Copyright © 2004 John Wiley & Sons, Ltd.

- The user-memory content extraction functions `Get` and `GetPtr`, embedding defensive behaviour.
- The implementation of the `UserMemory` (being private, since it is only used by the `MemoryObjectSuper::Find` function) and the `InsideUserMemory` pure virtual functions previously described.
- The constructor, supplying the appropriate arguments for the super-class instance initialization, i.e. `MemoryObjectSuper(srcFile, srcLine, _size, _isDynamic)`, while also requesting an exhaustive ownership check for all smart pointer instances, i.e. `SmartPtr-Super::CheckMemoryOwnership(this)` (to be explained in the next section).

SMART POINTER CLASS DETAILS

Implementation of the super class

As previously discussed (see also Figure 1), the super class for smart pointers, `SmartPtrSuper`, encapsulates a local member of `MemoryObjectSuper*` type, which is the *owner* memory object (can be null). It is legal to have a single memory object owning multiple smart pointers, since it is possible for the user memory of a memory object to physically encompass multiple smart pointer instances. In Figure 7, the super class for smart pointers is outlined, encompassing the following members.

- The `GetOwner` and `SetOwner` members for manipulation of the owner memory object.
- The `Invalidate` pure virtual member, implemented by derived pointee-type specific smart pointer classes, which 'nullifies' the smart pointer instance (i.e. prohibits further access for either reading or writing to the attached user memory).
- The `GetUserMemory` pure virtual member returning the starting address of the pointed user memory.
- The static member `CheckMemoryOwnership`, which, as previously explained, is called in the `MemoryObject<T>` constructor (see Figure 6).
- The static member function `InvalidateAllUsingNonDynamic`, which iterates through all smart pointer instances, and invalidates those pointing to non-heap user memory with a starting address equal to the actual parameter `userMem`.
- The `IsExternalReference`, which will be explained later in the discussion on the garbage collection algorithm.
- The embedded public class `StackMemPopGuard`, which is employed to guard the use of stack user memory as follows.
  - When stack memory is used, e.g. taking the address of a local variable (or array), a `StackMemPopGuard` instance is declared at the point in the code exactly following the declaration of the subject variable (or array), always within the same block, supplying its memory address as the constructor argument. As a result, when the runtime control exits the enclosing block, the destructor of this temporary instance is called, inherently invalidating all pointers using the recently popped stack memory, via a call to the `SmartPtrSuper::InvalidateAllUsingNonDynamic` function. The declaration of the temporary instance is hidden under the utility macro `DSLOCALVAR` (for an example of use, see the bottom section of Figure 7).

Copyright © 2004 John Wiley & Sons, Ltd.

```
class SmartPtrSuper {
    private:
    MemoryObjectSuper* owner;

    public:
    MemoryObjectSuper* GetOwner (void) const { return owner; }
    void             SetOwner (MemoryObjectSuper* _owner)
                         { owner= _owner; }
    virtual void     Invalidate (void) = 0;
    virtual void*    GetUserMemory (void) = 0;
    static void      CheckMemoryOwnership (MemoryObjectSuper* memObj);
    static void      InvalidateAllUsingNonDynamic (void* userMem);
    bool             IsExternalReference (void) const
                         { return !owner || !owner->IsDynamic(); }
    class StackMemPopGuard {
    private:
    void* userMem;
    public:
    StackMemPopGuard (void* userMem) :
        userMem(_userMem) { assert(userMem); }
     ~StackMemPopGuard()
        { SmartPtrSuper::InvalidateAllUsingNonDynamic(userMem); }
    };
};

#define DSLOCALVAR(var) \
        SmartPtrSuper::StackMemPopGuard lifetime##var(&var)

// Example of use for DSLOCALVAR.
//
DSDECL(int*) iPtr;
{ int temp; DSLOCALVAR(temp); iPtr = DSADDRESSOF(temp); }
*iPtr = 20; // Error trapped, since 'iPtr' has been invalidated.
```

Figure 7. The member functions of the `SmartPtrSuper` class.

Implementation of the core functionality in the pointee-type parameterized derived class

The body of the `SmartPtrImpl` template class is depicted in Figure 8. The template parameter `T` corresponds to the pointee type. According to the previous overview of the software architecture, smart pointers encompass a variable of type `MemoryObject<T>*`, denoted as `memObj`, holding the address of its runtime referred memory object instance (i.e. the memory object that holds the user memory attached to the smart pointer).

When a smart pointer is 'redirected' to refer to a memory object other than the currently referred `memObj`, via an internal call to `SetMemory`, the `CollectGarbage` function is called, internally calling `memObj->RemoveReference` (a member of `MemoryObjectSuper` class). The latter call appropriately performs garbage collection.

Copyright © 2004 John Wiley & Sons, Ltd.

```
template <class T> class SmartPtrImpl : public SmartPtrSuper {
    private:
    MemoryObject<T>* memObj;
    int              pos;

    void CollectGarbage (void) {
        if (memObj)
            memObj->RemoveReference(this);
    }
    void SetMemory (MemoryObject<T>* mem) {
         if (memObj != mem)
             { CollectGarabge(); memObj = mem; pos = 0; }
    }
    bool ToBoolean (void) const { return !!memObj; }

    public:
    void Attach (T* userMem, unsigned size, bool isNewExpression);
    T* Native (void) const
      { return memObj ? memObj->GetPtr(0) : (C*) 0; }
    T& operator*(void) const
      { assert(memObj); return memObj->Get(pos); }
    T* operator->(void) const
      { assert(memObj); return memObj->GetPtr(pos); }
    T& operator[] (int index) const
      { assert(memObj); return memObj->Get((unsigned) (pos+index)); }

    // The defensive arithmetic and boolean pointer
    // operators are inserted at this point.

    const SmartPtrImpl& operator= (const SmartPtrImpl& Rvalue) {
         assert(this != &Rvalue);
         if (memObj != Rlvalue.memObj) {
             CollectGarbage();
             if (memObj = Rvalue.memObj) // Assignment on purpose
                 memObj->AddReference(this);
         }
         pos = Rvalue.pos;
         return *this;
    }

   SmartPtrImpl (void);
   SmartPtrImpl (T* userMem, unsigned size, bool isNewExpression);
   SmartPtrImpl (const SmartPtrImpl& ptr);
   virtual ~SmartPtrImpl() { CollectGarbage(); }
};
```

Figure 8. The member functions of the `SmartPtrImpl<T>` template class, parameterized for pointee type `T`, including de-reference, arrow, subscript and assignment operators, with defensive implementations.

Copyright © 2004 John Wiley & Sons, Ltd.

```
// The following definitions go inside the
// SmartPtrImpl<T> template class.

SmartPtrImpl (MemoryObject<T>* _memObj, int _pos) {
    if (memObj = _memObj) { // Assignment on purpose
        memObj->AddReference(this);
        if (!memObj->IsDynamic())
            SmartPtrSuper::RegisterNonDynamic(this);
    }
    pos = _pos;
}
const SmartPtrImpl operator++ (int) { // Post-increment
    assert(memObj);
    ++pos;
    return SmartPtrImpl(memObj, pos-1);
}
const SmartPtrImpl& operator++ (void) { // Pre-increment
    assert(memObj);
    --pos;
    return *this;
}
const SmartPtrImpl& operator-- (void);
const SmartPtrImpl operator-- (int);
const SmartPtrImpl operator+ (int offset) const;
const SmartPtrImpl operator- (int offset) const;
const SmartPtrImpl& operator-= (int offset) const;
const SmartPtrImpl& operator+= (int offset) const;

bool operator==(const SmartPtrImpl& ptr) const;
bool operator!=(const SmartPtrImpl& ptr) const;

bool operator!(void) const
    { return !ToBoolean(); }
```

Figure 9. The common arithmetic and comparison pointer operators with embedded bug defense for the `SmartPtrImpl<T>` template class.

Common pointer expressions and embedded bug defense

In the `SmartPtrImpl` class, the local variable `pos` holds the current position (initially 0) of the pointer within user memory (i.e. `memObj->userMem`). As previously mentioned in the discussion on `MemoryObject<T>` (see Figure 6, member `GetPtr`), within the implementation of the operators `*` and `->` (or `[]` with `index`, respectively), the value of `pos` (or `pos+index`) is asserted to always fall inside the interval `[0, memObj->size`, thus forbidding illegal memory accesses.

In the assignment operator, if `this` already shares the same memory object with `Rvalue`, it only sets the position to be equal to `Rvalue.pos`. Otherwise, the two smart pointers refer to different memory objects, so that a call to `CollectGarbage` is first made to cancel reference of `this` smart

Copyright © 2004 John Wiley & Sons, Ltd.

pointer to `memObj`, followed by an assignment of the new referred memory object `Rvalue.memObj` to the local `memObj` variable. Then, in case the new value of `memObj` is not null, a call to `memObj->AddReference` member is made, so that `memObj` can internally update the referral list by adding the newly established `this` smart pointer reference.

The full range of arithmetic and operators allowed for built-in pointers is overloaded for smart pointers too, as indicated in Figure 9. For clarity, only the implementation of the most representative operators is shown, while the rest follow similar implementation patterns. The creation of a temporary smart pointer instance may be noticed, which is returned by the post-increment (or post-decrement) overloaded operator; this temporary instance holds the old position in user memory before the increment (or decrement) is applied.

CYCLIC REFERENCES' ELIMINATION ALGORITHM

Limitations of reference counting in cyclic dependencies

Before introducing some code fragments which cause cyclic dependencies at runtime, an appropriate definition is formally provided as follows.

- When a pointer P refers to user memory M of a memory object O, a formal relationship *refers(P, O)*, informally represented as $P \rightarrow O$, is established.
- When the physical memory occupied by pointer P is encapsulated inside the user memory M of a memory object O, a formal relationship *owns(O, P)*, informally represented as $O \rightarrow P$, is established.
- The informal representation of the *owns* and *refers* relationships during runtime produces a directed graph. Any cyclic path in this graph identifies cyclic dependencies among all the engaged pointer variables and memory objects.

In Figure 10, a very simple code fragment is depicted with a sample agent class implementation, supporting agent instance hierarchies, which engage cyclic pointer references. During runtime, the execution of the last line of code of Figure 10 results in the production of cyclic references, as outlined in Figure 11. When the `topAgent` smart pointer variable is assigned a new value, the root of the agent instance hierarchy is no longer syntactically accessible, meaning the whole hierarchy is practically unreachable. However, while the reference counter of the objects engaged in this 'dangling' instance hierarchy is decreased, it does not reach zero, since for each of the memory objects (of Figure 11) there is at least one referring smart pointer. Consequently, if garbage collection is based solely on reference counting information, this instance hierarchy, though not being syntactically referenceable by the programmer, cannot be automatically disposed.

External and internal references

Let us assume that during runtime all smart pointers referring to a particular memory object either directly (i.e. through their `memObj` value), or indirectly (i.e. by following the arrow edges in the ownership/reference graph) are owned by memory objects of dynamically allocated user memory (e.g. the scenario of Figure 11). Then, effectively, there are no available smart pointer variables, which

Copyright © 2004 John Wiley & Sons, Ltd.

```
const MAX_MANAGED_AGENTS = 256;
class Agent {
    private:
    DSDECL(Agent*) managed[MAX_OWNED_AGENTS];
    DSDECL(Agent*) parent;
    unsigned curr;

    public:
    void AppendManaged (DSDECL(Agent*) agent) {
        assert(curr < MAX_MANAGED_AGENTS);
        managed[curr++] = agent;
    }

    Agent (DSDECL(Agent*) _parent = DSNULL(Agent*)):
        curr(0),
        parent(_parent)
        { if (DSTRUE(parent) parent->AppendManaged(DSTHIS); }
};

Agent* topAgent = DSNEWVAR(new Agent);
DSNEWVAR(new Agent(topAgent));
DSNEWVAR(new Agent(topAgent));
...
topAgent = anotherTopAgent; // Assume later the pointer is changed.
```

Figure 10. An example using a very simple agent class, from which cyclic pointer references are produced (i.e. reference counter does not become zero, even though memory should be collected).

normally occupy global (static) or auto (stack) storage, to access such user memory either directly or indirectly. Consequently, even though the `refCounter` of the subject memory object is not zero, the attached user memory becomes unusable and should be collected. This reveals the key role of typical programmer-defined pointer variables in identifying whether user-memory is still reachable. In this context, pointers occupying static or auto storage are called *external references*, and all pointers occupying free storage are called *internal references*. In the described implementation, a smart pointer is an external reference *if and only if* the expression '`!owner || !owner->IsDynamic()`' is true (i.e. either the smart pointer has no owner, or it is owned by a memory object of non-heap user memory, see Figure 7, `IsExternalReference` member).

In Figure 12, a runtime snapshot indicating external and internal pointer references is outlined. The heap memory unit A is represented only by a pointer variable falling inside A. This pointer, which is embedded in heap memory, is an internal reference. Clearly, due to the presence of the reference $A^{\text{pointer}} \rightarrow A^{\text{memory object}}$, the reference counter of the A memory object will never actually become zero. However, since the user memory of the A memory object is no longer referenceable, it has to be collected as garbage. Heap memory unit B, although being referred to cyclically via $B \rightarrow C \rightarrow B$, engaging solely internal references, is indirectly reachable by an external reference. This reference is actually the pointer variable embedded in stack/global memory unit D, while the *reference path* is

Copyright © 2004 John Wiley & Sons, Ltd.

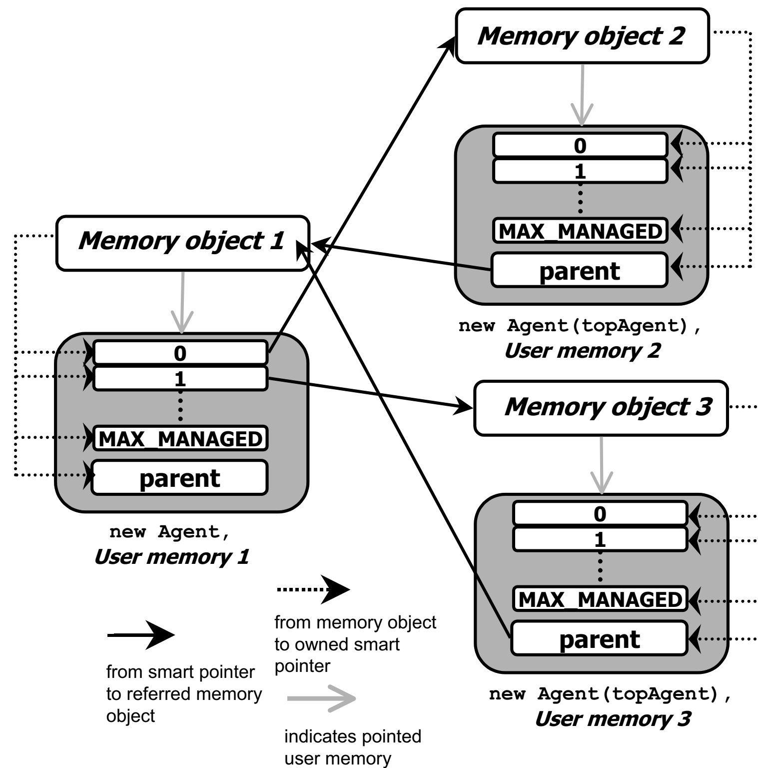

Figure 11. Runtime picture of the various links corresponding to the example of Figure 10, after the last statement is executed.

$D \rightarrow C \rightarrow B$. Hence, B is accessible by the programmer through the program object D. Finally, heap memory unit C is also directly accessible by the same external reference (i.e. the pointer variable inside D), so it can be reached by the programmer.

Overview of existing garbage collection policies

Cyclic dependencies, such as those outlined in Figure 12, are very common in most software systems, where collections of dynamic object instances are organized and linked together in numerous ways reflecting the underlying architectural plan and application logic. Generally, the two most challenging issues in a garbage collection algorithm are (a) to collect memory when it can no longer be referenced by the programmer (i.e. recycle memory blocks that become unreachable by program variables); and (b) to detect as quickly as possible when the previous condition holds. In general purpose, low-level memory managers, there are usually centralized garbage collectors that deal with unreachable memory in two ways: (i) mainly heuristic methods that maintain use statistics, based on the assumption that

Copyright © 2004 John Wiley & Sons, Ltd.

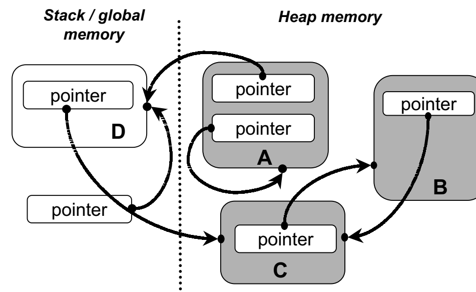

Figure 12. A snapshot of runtime references. The representation of pointer variables embedded in memory units implies that the embedding unit corresponds to an aggregate data structure that includes a declaration of the embedded pointer variable.

if it is not accessed for a long time, then there is a high probability it is unreachable like hand-made memory managers for games, where the lifetime of dynamic objects has an acceptably small threshold [9]; and (ii) by performing *tracing* garbage collection policies [10], engaging potentially demanding graph searches (i.e. a recycled object may result in recycling another object, and so on).

Usually, 'late probabilistic detection' methods are not memory efficient, since they tend to leave unused memory spread for a while, effectively slowing down execution, at sometimes undesirable points, when the collection cycle is triggered during execution. However, they may work reliably and effectively when (a) applications continuously create dynamic objects; and (b) there is a small upper bound regarding the lifetime of the created objects (like in particle systems or video games).

Sophisticated tracing collection techniques (i.e. performing graph searches, ignoring cyclic references), such as the very early approach described in Dijkstra *et al.* [11] or more recent approaches such as the one described in Pirinen [12], reflect generic algorithms that do not make any particular assumptions on the target programming language, nor limit their scope to any particular memory object model. The main issue that increases the complexity of such algorithms is that, when deciding to recycle a memory block, the algorithm has to recursively calculate the effect on other potentially dependent blocks, causing a *domino effect*.

Overview of the garbage collection method

Instead of choosing a general-purpose garbage collection method, a simplification of a tracing collector has been designed. However, this is only applicable to the specific types of memory objects engaged in the smart object library. The key point in the algorithm is the handling of the 'domino effect' by taking advantage of a genuine C++ implementation. More specifically, based on the previously introduced definitions, when allocated user memory is to be collected, its corresponding memory object instance is destroyed. Effectively, this results in two actions: (a) calling the destructor of the memory object instance, which, in the developed implementation, informs all referring pointers of its destruction (thus updating references as needed); and (b) deleting the allocated user memory, which may further

Copyright © 2004 John Wiley & Sons, Ltd.

result in the destruction of any embedded (owned) smart pointer variables, subsequently leading to another garbage collection cycle.

The previously described characteristics allow the garbage collection algorithm to concentrate solely on the subject memory object (i.e. the one losing a reference), needing only to quickly detect the existence of external references. Then, when dynamic user-memory-owning smart pointers are collected, the destructors of the embedded smart pointers will be automatically called, leading to subsequent collection rounds due to the updates caused on the memory object references. Consequently, the core of the garbage collection algorithm, given a target memory object, concerns the *fast detection of a path towards an external reference*.

Fast tracing of external references

Within the defensive smart pointer class there is no explicit representation of the runtime reference/ownership graph by means of a centralized data structure. Instead, as previously mentioned, the graph is implicitly stored, being distributed in the local data of smart pointers and memory objects, by engaging two types of edges.

- Memory object *reference edges*, of the form $1 : N, N > 0$, associating one memory object to one or more smart pointer instances:
  - during runtime, every memory object keeps a list (i.e. the `referToMe` member) of all smart pointer instances *referring to it*, i.e. pointing to its user memory.
- Smart pointer *ownership edges*, of the form $1 : N, N >= 0$, associating one memory object to zero or more owned smart pointers:
  - smart pointers encompass a `MemoryObjectSuper*` member named `owner`, being the *owner* memory object. As already mentioned, ownership implies that the smart pointer instance occupies memory physically embedded in the user memory of the *owner* memory object.

As previously discussed, the garbage collection algorithm reflects the fact that *if for a memory object M there is no path towards an external reference, then M cannot be referenced, hence it should be recycled*. The resulting implementation of the `IsReferenceable` key member of the `MemoryObjectSuper` class is outlined in Figure 13. This function implements a typical graph node search algorithm. The search terminates positively, i.e. an external reference is found, in one of the following cases.

- The subject memory object holds stack/global user memory (i.e. there are programming variables to directly access the user memory, while, since it is not allocated from the heap, it is an error to delete it).
- The subject memory object is represented directly or indirectly by a smart pointer that has no *owner* memory object (i.e. the smart pointer is a declared variable occupying stack/global memory, being an external reference).
- The subject memory object is represented directly or indirectly by a smart pointer that is owned by a memory object of stack/global user memory (i.e. the smart pointer is embedded in an aggregate data structure that occupies stack/global memory, being an external reference).

Copyright © 2004 John Wiley & Sons, Ltd.

```
bool MemoryObjectSuper::IsReferenceable (void) {
    if (user-memory is not from the heap)
        return true;
    start = this;
    result = IsReferenceableRecursion();
    for (each smart pointer Y in visit list)
        Mark Y as not visited;
    Clear visit list;
    return result;
}

bool MemoryObjectSuper::IsReferenceableRecursion (void) {
    for (each smart pointer X that refers to me, where X not visited) do {
        Mark X as visited and append X in visit list;
        if (X.IsExternalReference())
            return true;
        else
        if (X.GetOwner() notequal start and
            X.GetOwner().IsReferenceableRecursion())
            return true;
    }
    return false;
}
```

Figure 13. Deciding user-memory referenceability as a typical graph node search algorithm of $O(N)$ worst-case complexity ($N =$ number of smart pointer instances + number of memory object instances), by tracing external references.

DETAILS OF ATTACHING USER MEMORY TO SMART POINTERS

Following the presentation of the compact garbage collection algorithm, the implementation of the key member to attach user memory to smart pointer instances of the `SmartPtrImpl<T>` template-based class, `Attach`, is discussed. The complete implementation of `Attach` is depicted in Figure 14. The lower part of Figure 14 shows how the `Attach` member is actually called inside the `SmartPtrImpl<T>` constructor.

In case the user memory supplied in the `userMem` formal argument is null, the smart pointer is reset by passing via `SetMemory` a null target memory object value, as indicated in the last line of the code. Otherwise, the first action is to identify if for the supplied `userMem` there is a previously created memory object instance, since only one guardian memory object instance is allowed per distinct user-memory address. This is accomplished via a call to `MemoryObjectSuper::Find`, and the result is assigned to the `existingMem` local variable.

In case there is no such memory object, a new one is constructed via `new MemoryObject <T>(userMem, size, isNewExpression)`, supplied as the parameter to `SetMemory` for the caller smart pointer instance. Otherwise, the already existing memory object `existingMem` is passed to `SetMemory`. Then, in case the supplied `userMem` is actually a ‘new’ expression (i.e. `isNewExpression==true`), it is asserted that the `existingMem` memory object has not

Copyright © 2004 John Wiley & Sons, Ltd.

```
// Part of the SmartPtrImpl<T> template class.

void Attach (T*       userMem,
             unsigned size,
             bool     isNewExpression,
             unsigned srcLine,
             char*    srcLine) {

  if (userMem) {

     MemoryObjectSuper* knownObj = MemoryObjectSuper::Find(userMem);
     bool sameMemObj = memObj && memObj==knownObj;

     if (knownObj) { // A mem object for userMem exists

       SetMemory((MemoryObject<T>*) knownObj); // Just refer to it.

       if (isNewExpression) {
          assert(!memObj->IsDynamic());
          memObj->SetAsDynamic();
       }

       if (size > memObj->GetSize()) {
          assert(isNewExpression || !memObj->IsDynamic());
          memObj->AssumeLargerSize(size);
       }
       else
       if (size < memObj->GetSize())
          assert(!isNewExpession); // Was used with larger size!
     }
     else
       SetMemory( // Make a new mem object for this user-memory.
          new MemoryObject<T>(
             userMem, size, isNewExpression, srcLine, srcFile
          )
       );

     if (!sameMemObj) { // Different attached memory than before.
       memObj->AddReference(this);
       if (!memObj->IsDynamic()) {
          SetRegisteredAsNonDynamic(true);
          SmartPtrSuper::RegisterNonDynamic(this);
       }
       else
          SetRegisteredAsNonDynamic(false);
     }
  }
  else // Null user memory supplied.
     SetMemory( (MemoryObject<T>*) 0); // Nullify smart pointer.
}

SmartPtrImpl (T* userMem, unsigned size, bool isNewExpression) :
    memObj((MemoryObject<T>*) 0),
    pos(0)
    { Attach(userMem, size, isDynamic); }
```

Figure 14. The key function for attaching user memory to smart pointers.

Copyright © 2004 John Wiley & Sons, Ltd.

```
class Window { public: Window (void){} };

class Button {
    DSDECL(Window*) parent;
    public:
    Button (DSDECL(Window*) win) : parent( win ){}
};

class ConfirmBox : public Window {
    private:
    DSDECL(Button*) ok;
    DSDECL(Button*) cancel;

    public:
    ConfirmBox (void) {
       ok = DSNEWVAR(new Button( DUPCAST(Window, DSTHIS ) ));
       cancel = DSNEWVAR(new Button( DUPCAST(Window, DSTHIS ) ));
    }
};

{ DSDECL(ConfirmBox*) confBox = DSNEWVAR( new ConfirmBox ); }
```

Figure 15. Assigning legally dynamic user memory to pointers even before the 'new' expression, which actually allocates this memory, returns.

been previously registered as being associated to dynamic user-memory (see the first assertion in bold type in Figure 14). The following question may arise at this point: *of how is it possible in such a call to already have an existing memory object for* `userMem`, *given that* `userMem` *is supplied via a* `new` *expression in the same call*. In Figure 15, one such typical scenario is illustrated. Within the block shown at the lower part of Figure 15, object construction takes place before the `new ConfirmBox` expression actually returns, and the result of the latter expression is subsequently assigned to the `confBox` local variable. However, the address of the object created through the `new ConfirmBox` expression is already used inside the `ConfirmBox` constructor, before the 'new' expression returns, through the `DSTHIS` expression. To cater for such natural scenarios, in the construction of the smart pointer instance returned by the `_DSTHIS` template function, which is used by the `DSTHIS` macro (discussed earlier, see Figure 4), the actual value `false` is passed to the formal argument `isNewExpression`. Another helper macro appearing in Figure 15 is the `DUPCAST(T,ptr)` macro, which is used to convert a smart pointer instance `ptr` of pointee type C to a smart pointer instance of pointee type T, if and only if C is derived from T. This will be elaborated upon later, while discussing the implementation of casting operators for smart pointers.

Tracking type-size violations when using dynamic user memory

Type-size violations concern the access of dynamic user memory by casting to a data type $T1$ with a size larger than the original type $T2$ for which dynamic allocation has been performed. Continuing the previous discussion on the `Attach` member (see Figure 14), there can be only one place in code

Copyright © 2004 John Wiley & Sons, Ltd.

```
class A {
   int a; DSDECL(A*) self;
   public: A(void){ self = DSTHIS; } ~A(){}
};

class B : public A {
   int b; DSDECL(B*) self;
   public: B(void){ self = DSTHIS; } ~B(){}
};

class C : public B {
   int c; DSDECL(C*) self;
   public: C(void){ self = DSTHIS; } ~C(){}
};

DSDECL(C*) c;
c = DSNEWVAR( new C );
```

Figure 16. Engaging legally the same user memory with different size information (sizeof A < sizeof B < sizeof C), while assigning to pointer variables.

where a particular heap user memory address is engaged via its creational 'new' expression. Hence, if `isNewExpression` is `true`, the expression `memObj->IsDynamic()` for the already existing `memObj` should be asserted to be `false`, in order to verify that the same user memory has not been previously qualified as being a 'new' expression. Then, via the call to `SetAsDynamic`, the fact that the 'new' creational expression for `userMem` of `memObj` has been processed is reflected. Next, it is checked if the `size` value supplied for user memory in the call is larger than the currently known size (i.e. `memObj->GetSize()`). Again, this is a very natural scenario, since the same user memory address can be legally qualified as having different sizes within different calls. In Figure 16, an example demonstrating this case is outlined, where, for each of the three `DSTHIS` smart pointer construction expressions, producing `this` compatible smart pointers, the supplied size information differs (`sizeof(A)`, `sizeof(B)` and `sizeof(C)`, respectively). Going back to Figure 14, the supplied user memory `size` is set via a call to `AssumeLargerSize` only if it is larger than the current user memory size of the target memory object instance. However, in this case it is should be asserted that either the supplied user memory is engaged with its creational 'new' expression in this call, or the creational call has not yet been processed. In other words, this assertion states that *if user memory is to be created via a 'new' expression for type T1, it should not be used afterwards with a type T2 of larger size*.

Similarly, if the supplied user memory size is smaller than the stored size, actually `size < memObj->GetSize()`, then it should be asserted that the present call is not the creational 'new' expression. Otherwise, it is implied that *the supplied user memory, created via 'new' in the present call for type T1, has been illegally used before with a type T2 of larger size*. This gives a clear indication of the presence of erroneous statements that have already been executed. These statements can easily be traced automatically in our smart pointer library, by introducing the recording of memory access history within memory objects.

Copyright © 2004 John Wiley & Sons, Ltd.

Final details

Finally, continuing with Figure 14, in case the target memObj of the caller smart pointer has actually changed, i.e. !sameMemObj is true, the call memObj->AddReference(this) is made, so that memObj adds the caller smart pointer in its referToMe list. Then, if the newly referred memObj does not 'guard' heap memory, the smart pointer calls SetRegisterAsNonDynamic(true), while also registering itself in the global list of pointers to stack/global memory via a call to RegisterNonDynamic. Such an explicit registration is important in order to automatically invalidate the appropriate smart pointer instances when their associated stack user memory is popped (as discussed in Figure 7), through the use of the SmartPtrImpl::StackMemPopGuard utility class.

IMPLEMENTATION OF TYPE CASTING OPERATORS

Pointer type casting concerns the automatic up-casting, and the special purpose casting operators static_cast, dynamic_cast and reinterpret_cast. For all those cases, the type casting logic has been implemented inside the SmartPtrImpl<T> template class. The programmer needs to employ the corresponding utility macros (see Figure 17). As shown, the implementation of those macros relies on the internal application of the type casting to the pointed user memory, and the subsequent construction of a corresponding smart pointer instance, i.e. the Caster template member function of the SmartPtrImpl<T> template class, which is used to implement the four type casting members, namely UpCaster, StaticCaster, DynamicCaster and ReinterpretCaster.

While the use of the three explicit casting operators via the utility macros appears quite natural, the use of the DUPCAST macro to explicitly qualify a conversion, which had to be automatically performed by the compiler, seems less straightforward. As an alternative, to support automatic conversion, it is possible to provide template versions of all the various copy constructors of SmartPtr<T,N> and SmartPtrImpl<T> template classes, with template type parameter C (where T is derived from C), so that a smart pointer argument for the C base class pointee type can also be accepted. The implementation resembles in style the way the arguments are supplied to the construction of the _temp instance, within the Caster template function of Figure 17.

FINAL DETAILS OF SMART POINTER UTILITY MACROS

The use of smart pointers engages different syntax mainly for pointer declarations and user-memory assignment in comparison to built-in pointers. This is true not only for the particular defensive programming library implementation discussed, but also for all other existing libraries, such as the Boost library [3]. While defensive smart pointers practically help track down erroneous pointer use, they tend to degrade performance, since they introduce a considerable execution overhead for each pointer operation. As a result, in production mode, software developers may wish to completely substitute the use of smart pointers with conventional built-in pointers, without, however, introducing explicit code updates. Naturally, the possibility to strip off smart pointers applies only if garbage collection is not considered as a standard feature in production mode (i.e. programmers still handle

Copyright © 2004 John Wiley & Sons, Ltd.

```
// Part of the SmartPtrImpl<T> template class.

template <class C> static C* UpCaster (T* native)
    { return native; }
template <class C> static C* StaticCaster (T* native)
    { return static_cast<C*>(native); }
template <class C> static C* DynamicCaster (T* native)
    { return dynamic_cast<C*>(native); }
template <class C> static C* ReinterpretCaster (T* native)
    { return reinterpret_cast<C*>(native); }

template <class C, C* (*Fcast)(T*)>
SmartPtrImpl<C> Caster (void) const {
    SmartPtrImpl<C> _temp (
        Fcast((T*) memObj->GetPtr(0)), // Casting function applied.
        memObj->Size(),
        memObj->IsDynamic(),
        memObj->GetSrcFile(),
        memObj->GetSrcLine()
    );
    if (memObj)
        _temp += pos; // Apply current pointer offset.
    return _temp;
}

template <class C> SmartPtrImpl<C> UpCast (void) const
    { return Caster<C, UpCaster<C> >(); }

template <class C> SmartPtrImpl<C> StaticCast (void) const
    { return Caster<C, StaticCaster<C> >(); }

template <class C> SmartPtrImpl<C> DynamicCast (void) const
    { return Caster<C, DynamicCaster<C> >(); }

template <class C> SmartPtrImpl<C> ReinterpretCast (void) const
    { return Caster<C, ReinterpretCaster<C> >(); }

// Part of global definitions of the utility macros.

#define DUPCAST(T,ptr) ptr.UpCast<T>()
#define DSTATIC_CAST(T,ptr) ptr.StaticCast<T>()
#define DDYNAMIC_CAST(T,ptr) ptr.DynamicCast<T>()
#define DREINTERPRET_CAST(T,ptr) ptr.ReinterpretCast<T>()
```

Figure 17. Implementation of type casting operators, and the corresponding utility macros that must be employed.

Copyright © 2004 John Wiley & Sons, Ltd.

```
#define DSNATIVE(ptr)    ((ptr).Native())
#define DSDELETE(ptr)    (ptr).Delete(false)
#define DSDELARR(ptr)    (ptr).Delete(true)
#define DSNULL(T)        (DSDECL(T)::Null())
#define DSTRUE(ptr)      (!!ptr)
#define DSBYNATIVE(addr) _DSBYNATIVE(addr, __FILE__, __LINE__)
#define DSNEWOBJ(type, args) \
                          DSNEWVAR(new type##args) // 'args' in ()
template <class T>
DSDECL(T*) _DSBYNATIVE (T* addr, char* file, unsigned line) {
    DSDECL(T*)(addr, 1, false, file, line);
}
```

Figure 18. The remaining utility macros and template functions.

memory disposal, but exploit garbage collection for detecting memory leaks and lost references). Additionally, employed third-party libraries are likely to produce and require built-in pointers, thus requiring the ability to easily convert from or to smart pointers.

In addressing the above issues, it is important to enable programmers to structure a 'unified' implementation, irrespective of whether smart or built-in pointers are used, by providing macros that hide the differences under common declaration specifiers, expressions and operators. Those macros are provided with two different versions of the required definitions, one binding directly to built-in pointer semantics, and the other binding to defensive smart pointer classes. In Figure 18, the definitions for the remaining utility macros are provided.

In the right part of Figure 19, examples of using the utility macros are illustrated. The original source code, programmed via built-in pointers, appears on the left side. Utility macros enable switching to built-in pointers through the use of a pre-processor flag. Even though the use of the pre-processor is a less preferred technique in C++, it is currently the only way to support conditional compilation, so as to enable intuitive code version control.

Support for explicit memory disposal

Another key feature of defensive smart pointers is the support for explicit memory disposal, i.e. the `Delete` member of smart pointers, as well as the `DSDELETE` and `DSDELARR` utility macros. These allow programmers to employ smart pointers only as an effective diagnostic library, without shifting from the original C++ programming style for manual memory allocation control. In such cases, one more assertion needs to be added to the `MemoryObjectSuper` class, within the `DecRefCounter` function, in the *if* block encompassing the memory collection logic (see Figure 20). These assertions specify that either the associated user memory has not been allocated from the heap (i.e. `isDynamic==false`), or that the associated user memory has already been deleted by the programmer (i.e. `wasExplicitlyDelete==true`). In other words, the assertion reflects the logic that *if the garbage collector decides to dispose of dynamically allocated memory, the programmer no longer has syntactic access to it.*

Copyright © 2004 John Wiley & Sons, Ltd.

`int* a;`	`DSDECL(int*) a;`
`int** a1 = &a;`	`DSDECL(int**) a2 = DSADDRESSOF(a);`
`a = new int[10];`	`a = DSNEWARR(int, 10);`
`a[10] = 10;`	**`a[10] = 10;`** ← ***Out of bounds access trapped***
`int* b = (int*) 0;`	`DSDECL(int*) b = DSNULL(int*);`
`*b = 3;`	**`*b = 3;`** ← ***Null pointer access trapped***
`{ int c; b = &c; }`	`{ int c; DSLOCALVAR(c); b=DSADDRESSOF(c); }`
`*b = 4;`	**`*b = 4;`** ← ***Popped memory access trapped***
`delete a;`	**`DSDELETE(a);`** ← ***Missed use of delete[] trapped***
`++a;`	**`++a;`** ← ***Offset on deleted pointer trapped***
`void* p = a;`	`void* p = DSNATIVE(a);` ← ***Void pointers stay built-in !***
`new int(10);`	`DSNEWVAR(new int(10));` ← ***Lost reference trapped***
`a = (int*) p;`	`a = DSBYNATIVE((int*) p);` ← ***Casting needed for void****
`extern int add(int*,int);`	`extern int add(int*,int);`
`int sum = add(a,10);`	`int sum = add(DSNATIVE(a), 10);` ← ***Gets native pointer***
`int const* icp;`	`DSDECL(int const*) icp;` ← ***Const-ness allowed***
`icp = a;`	`icp = a;` ← ***Error, incompatible types (non-const)***
`int x;`	`int x;`
`icp = &x;`	`icp = DSADDRESSOF((const) x);` ← ***Needs qualification***
`*icp = 23;`	`*icp = 23;` ← ***Assignment to const is illegal***

Figure 19. Use of built-in pointer types (left), and respective employment of the utility macros for smart pointer engagement (right).

```
void MemoryObjectSuper::DecRefCounter (void) {
    ...
    if (!--refCounter || !IsReferenceable()) { // Memory is collected.
        #ifdef DSCATCH_MEMORY_LEAKS
        assert(!isDynamic || wasExplicitlyDeleted);
        #endif
        ...
    }
}
```

Figure 20. Catching 'on the spot' loss of referenceability for dynamically allocated memory, via the garbage collection mechanism.

Copyright © 2004 John Wiley & Sons, Ltd.

SUMMARY AND CONCLUSIONS

The presented library for defensive smart pointers has been used as a diagnostic package, for bug defense, in software that engaged large collections of interconnected dynamic object instances during runtime, helping trap erroneous use of pointers, memory leaks, and report statistics for memory allocation and reference. Its design has been based on previous smart pointer patterns, while emphasizing genericity, encapsulating the ability to trap most potential pointer errors, supporting garbage collection that resolves cyclic references, and enabling the use of garbage collection for 'less than perfect' memory-use diagnosis. Since there are many resources available that discuss smart pointers, outlining alternative programming patterns, the emphasis has mainly been on presenting the most difficult hidden features that are less frequently revealed and discussed.

- The employment of recursive templates to implement type, as well as indirection depth genericity.
- The design of reference-counted memory objects that are aware of embedded smart pointer variables (i.e. the ownership relationship).
- The policy on dealing with specific runtime C++ operational semantic details, such as (a) use of pointers to objects before the allocation expression returns; and (b) variable size information when pointers of object classes in an inheritance hierarchy are engaged for the same derived instance.
- The technique to defend against erroneous accesses on stack/global memory apart from heap memory.
- Garbage collection that resolves cycles by fast tracing of external references.
- Utility macros to unify smart pointers and built-in pointers by supporting the definition of smart pointer types via a decoration macro over the original C++ pointer type, through the pointee type and indirection depth extractor template.

There is more to code than meets the eye

It is a fact that there is more in the program code than what can be observed and noticed by simple code reviewing, whatever the reviewing tactic and coding standard is. As a software system or package grows, and while dynamic memory allocation is heavily used, debugging often turns out to a painful 'pointer safari'. Real practice has shown that performance degrades when smart pointers are in control, while the coding process itself takes more time until someone becomes familiar with the related style. An important contribution to the runtime performance penalty is due to the cycle-eliminating garbage collection algorithm, which naturally trades speed for memory size, since each time pointers stop referring to user memory, the tracing algorithm is called. Some of the programmers that have used the presented utility macros have found them to be a little difficult when it comes to typing, while others have said that the overall code readability was reduced.

In this context, there have been a few stylistic proposals made by some experienced programmers for changing the 'look' of utility macros:

(i) employment of low-case lettering for macro identifiers, putting only 's' as a prefix, as opposed to 'DS' (standing for 'debug smart'), e.g. *snull(int*)*, *sdecl(char*)*;

Copyright © 2004 John Wiley & Sons, Ltd.

(ii) use of uniform *typedef*s for the most common types, so that the use of the macros can be reduced, e.g. *typedef DSDECL(int*) int1;*, making the declaration *int1 p* look far more readable than *DSDECL(int*) p*;

(iii) use of extra macros for commonly used expressions, e.g. *#define sintarr(n) DSNEWARR(int, n)*, reducing text size and thus enhancing ease of use.

It is left to the reader to consider if these proposed modifications may potentially enhance the quality of use of the library.

From our experience in employing defensive smart pointers, pointer debugging has been largely automated due to large-scale direct defect detection, i.e. the vast majority of pointer-specific bugs are detected at the point they are actually created. Programmers using defensive smart pointers have acknowledged that the extra effort made in assimilating and deploying the smart pointer library has been actually paid back by severely reducing the overall debugging time. However, detailed comparative studies and measurements to provide a precise account on the reduction of the debugging time, or the relative increase in the source editing time, have not been applied so far. The main reason for this is that all programmers engaged in the target project (which was actually an independent research project, not a tester of the defensive library) were required to employ defensive pointers. Additionally, it has been observed that senior programmers were initially less willing to employ the library, while less experienced programmers were far more receptive and sometimes enthusiastic to use it.

Currently, the presented library offers most of the features in one package. Therefore, programmers are not able to selectively activate the desirable functionality as needed. For instance, one could need to completely disengage the garbage collection facility. Alternatively, some programmers may only be interested in memory usage statistics, while others may wish to turn off pointer arithmetic defense, and focus merely on memory leak detection. To overcome the current limitation, it is planned to organize all the smart pointer features in operationally orthogonal modules, enabling programmers to control via compile flags the type of defensive functionality to be activated.

REFERENCES

1. Meyers S. Implementing *operator->** for smart pointers. *Dr. Dobb's Journal* 1999; **12**(10).
2. Sharon Y. Smart pointers—what, why, which? http://ootips.org/yonat/4dev/smart-pointers.html [1999].
3. C++ Boost Library. Smart pointers. http://www.boost.org/libs/smart_ptr/smart_ptr.htm [2002].
4. Karlsson B. Smart pointers in Boost. *C/C++ Users Journal* 2002; **20**(4).
5. Vlascenau C. Generalizing the concepts behind auto_ptr. *C/C++ Users Journal* 2001; **19**(8):36–45.
6. C++ ISO Standard, ISO/IEC 1482. *Template Declarations*, ch. 14.5, 1998.
7. Veldhulzen T. Using C++ template meta-programs. *C++ GEMS*, Lippman S (ed.). SIGS: New York, 1996; 459–473.
8. Alexandrescu A. Smart pointers. *Modern C++ Design*. Addison-Wesley: Reading, MA, 2001; 157–194.
9. Hixon B, Martin D, Moore R, Schaefer G, Wifall R. Play by play: Effective memory management. *Gamasutra on line magazine*. http://www.gamasutra.com/features/20020802/hixon_01.htm [2 August 2002].
10. The memory management reference beginner's guide overview. http://www.memorymanagement.org/articles/begin.html [2002].
11. Dijkstra E, Lamport L, Martin A, Scholten C, Steffens E. *On-the-fly Garbage Collection: An Exercise in Cooperation* (*Lecture Notes in Computer Science*, vol. 46). Springer: Berlin, 1976.
12. Pirinen P. Barrier techniques for incremental tracing. *ACM ISMM'98*. ACM, 1998; 20–25.

Copyright © 2004 John Wiley & Sons, Ltd.