10/5-20-96 JSD

SANDIA REPORT

SAND95-8207 • UC-405

Unlimited Release

Printed April 1996

SMARTARRAY: A C++ Class Template for Self-Describing, Resizable, Error-Resistant Arrays

Kenneth J. Perano, Paul E. Nielan

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.

MASTER

This report has been reproduced from the best available copy.

Available to DOE and DOE contractors from:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge TN 37831

Prices available from (615) 576-8401, FTS 626-8401.

Available to the public from:

National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.
Springfield, VA 22161

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of the contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors or subcontractors.

SAND95-8207 • UC-405

Unlimited Release

Printed April 1996

SMARTARRAY: A C++ Class Template for Self-Describing, Resizable, Error-Resistant Arrays

Kenneth J. Perano

Distributed Information Systems Center

and

Paul E. Nielan

Materials and Engineering Sciences Center

Sandia National Laboratories/California

Abstract

The SmartArray class template supports one-dimensional (single index) arrays and provides four major features that make it superior to built-in C++ arrays: a SmartArray is self-describing (both capacity and content), a SmartArray can be dynamically resized, the index supplied to the [] operator of a SmartArray is bounds checked, and the lower bound of a SmartArray can be chosen by the programmer. Additionally, the SmartArray class provides a full set of traversal functions, an assignment operator, editing functions, and an error handling mechanism—yet remains small, self-contained, portable, efficient, and easy to master.

The class template SmartArray<T> requires that T be either a built-in type or a class that provides an assignment operator, a default (no argument) constructor, a copy constructor, and a destructor. If T does not contain any pointers, the compiler-generated versions of these four functions will probably be adequate.

Table of Contents

1 Introduction	5
2 Key Concepts	6
2.1 Capacity and content	6
2.2 Capacity bounds	6
2.3 Content limits	7
2.4 Example	7
2.5 Value semantics and requirements on class T	8
2.6 Pointers and names of SmartArray objects	8
3 Error handling	9
3.1 Introduction	9
3.2 Production error handling	9
3.3 Development error handling	9
3.4 Overview of error types	11
3.5 Array bound and traversal logic errors	12
3.6 The preset value	12
3.7 Data shredding	13
4 Member functions and operators	13
4.1 Constructors	14
4.2 Destructor	15
4.3 Copy constructor	16
4.4 Item access with bounds checking (operator[])	16
4.5 Capacity inspection functions	17
4.6 Content functions	17
4.7 Capacity change functions	18
4.8 Assignment operator	19
4.9 Editing functions	19
4.10 Traversal functions	22
4.11 Error handling functions	24
5 Advanced features	24
5.1 Advanced traversals	24
5.2 Item access without bounds checking	25
5.3 Reindexing	26
6 Retrofitting SmartArrays into existing code	27
6.1 Replacing built-in arrays with SmartArrays	28
6.2 Interfacing SmartArrays with pointers or built-in arrays	29
7 Programming examples	30
7.1 Constructor and operator[]	30
7.2 Traversal functions	31
7.3 Dynamic array resizing	32
7.4 Use of upperBoundNeeded with dynamic array resizing	33
7.5 Memory allocation error handling	34
7.6 Array bound logic error handling	34
7.7 Editing functions	35
7.8 SmartArray of a user-defined class	35
7.9 Use of the external pointer constructor	36
8 Enhancing the SmartArray class	37
8.1 Inheritance and assignment operators	37
8.2 Protected member functions	39
9 References	40

1 Introduction

The built-in C and C++ notions of an array are powerful low-level tools for building higher-level abstractions. However, for most programming applications they suffer from major limitations. The STL (Standard Template Library) component of the emerging ANSI Standard C++ Library [Reference 10] addresses several, but not all, of these limitations. The `<dynarray>` class template, once proposed for inclusion in the Standard Library, is missing many concepts which we believe are essential for a strong array class.

The Annotated C++ Reference Manual [Reference 4, page 137] states the following:

The C notion of an array—which C++ has adopted without change—is very low-level. Together with the pointer concept and the rules for converting an array to a pointer it provides a mechanism that closely models the memory and address concept of traditional hardware. This concept is simple, general, and potentially maximally efficient.

As a low-level concept, it also suffers from the following limitations:

- [1] An array is of fixed size with its size fixed at compile time.
- [2] An array is one-dimensional; talking about multidimensional arrays in C is simply referring to a conventional use of arrays of arrays.
- [3] An array is not self-describing; that is, given a pointer to an array there is no information available to determine the array's size.

...

Rather than repairing the C array/pointer concept, one ought to use it to build higher level abstractions. Self-describing, range checking arrays or the yet more powerful associative arrays are obvious candidates for user-defined types built from arrays. The C++ concepts of class, operator overloading, and templates support these notions.

This report describes a class template for one-dimensional arrays which addresses these limitations and implements some essential new concepts while remaining small, self-contained, portable, efficient, and easy to master.

The `SmartArray` class template supports one-dimensional (single index) arrays and provides four major features that make it superior to built-in C++ arrays: a `SmartArray` is self-describing (both capacity and content), a `SmartArray` can be dynamically resized, the index supplied to the `[]` operator of a `SmartArray` is bounds checked, and the lower bound of a `SmartArray` can be chosen by the programmer. Additionally, the `SmartArray` class provides a full set of traversal functions, an assignment operator, editing functions, and an error handling mechanism.

In its simplest form the `SmartArray` usage

```
SmartArray<float> a(10);
for (int i = 0; i <= 9; i++) a[i] = i;
```

can replace the built-in C++ array usage

```
float a[10];
for (int i = 0; i <= 9; i++) a[i] = i;
```

The class `SmartArray<T>` requires that `T` be either a built-in type or a class that provides an assignment operator, a default (no argument) constructor, a copy constructor, and a destructor. If `T` does not contain any pointers, the compiler-generated versions of these four functions will probably be adequate.

In this document, the term *array* refers to an instance of the class `SmartArray<T>`, and the term *programmer* refers to an application programmer using the `SmartArray` class.

If you wish to obtain the `SmartArray` class, please contact us.

Our phone numbers and e-mail addresses are:

Ken Perano	(510) 294-2779	perano@ca.sandia.gov
Paul Nielan	(510) 294-2510	pen@sandia.gov

2 Key Concepts

2.1 Capacity and content

The `SmartArray` class is self-describing; that is, information regarding the *capacity* and *content* of the array is embedded within each `SmartArray` object. Although both capacity and content refer to the storage locations of an array, the capacity refers to the storage locations currently *allocated*, while the content refers to the storage locations currently *used*. Since an array always knows its content, it is easy to traverse or manipulate only those array items which are in use.

In practice, for reasons of efficiency and simplicity, the content is approximated as the storage locations from the start of the array through the array item with the highest index used so far. For example, after an array with indices in the range 1 to 10 has been defined, it has a capacity of 10 items and a content of 0 items. If the third and sixth items are filled, then the array still has a capacity of 10 items, but now has a content of 6 items.

Note that the separate concepts of capacity and content are also necessary with built-in C++ arrays, but they are not embedded within the built-in array itself. Typically a programmer will create separate variables to store capacity and content information. By embedding capacity and content concepts within the array class, we relieve the programmer of the need to create separate variables to store and pass this information. Additionally, `SmartArray` member functions access and use capacity and content information as needed—the programmer does not have to supply it via additional arguments.

2.2 Capacity bounds

The capacity of an array refers to the storage locations currently *allocated*. The storage locations are contiguous and are obtained by internal use of the `new` operator. The capacity of an array is established by the lower and upper *bounds* of the array—values

(referred to as `lower_bound` and `upper_bound`) that are stored within and managed by the array. The value of `upper_bound` is set directly with the constructor and the `upperBound` member function, and is automatically updated as necessary by the member functions `upperBoundNeeded`, `put`, `append`, `insert`, `import`, `operator=`, and `reindex`. The value of `lower_bound` is set directly with the constructor and is changed as necessary by the member functions `operator=` and `reindex`.

An error results when an index outside the bounds of an array is supplied as an argument to applicable member functions.

The number of items in the capacity of an array is calculated (by the `capacity` member function) from the upper and lower bounds using the expression: `upper_bound - lower_bound + 1`.

2.3 Content limits

The content of an array refers to the storage locations currently *used*. The content of an array is established by the lower and upper *content limits* of the array—values (referred to as `lower_content_limit` and `upper_content_limit`) that are stored within and managed by the array. The lower content limit is defined to be equal to the lower bound, and the upper content limit is defined to be equal to the highest array index used so far. The upper content limit is automatically updated through use of the member functions `put`, `append`, `remove`, `insert`, `import`, `operator[]`, `operator=`, `empty`, and `reindex`. It can also be set directly with the member function `upperContentLimit`, as explained in Section 4.6.

The concept of array content could be refined by tracking the usage status of each storage location, but for efficiency and simplicity we have chosen to track only the upper content limit.

Use of the content limits is made by many of the `SmartArray` member functions, as well as being directly available to the programmer via the `lowerContentLimit` and `upperContentLimit` member functions.

The number of items in the content of an array is calculated (by the `content` member function) from the upper and lower content limits using the expression: `upper_content_limit - lower_content_limit + 1`.

2.4 Example

As an illustration of these concepts, consider the following code fragment.

```
SmartArray<int> b(7,100);
b[7] = 7; b[8] = 8; b[9] = 9; b[10] = 10; b[11] = 11; b[12] = 12;
b.upperBound(200);
b[102] = 5;
```

Immediately before the call to `upperBound`, the lower bound is 7, the upper bound is 100, the capacity is 94 items, the upper content limit is 12, and the content is 6 items. After the assignment to `b[102]`, the lower bound is 7, the upper bound is 200, the capacity is 194 items, the upper content limit is 102, and the content is 96 items.

2.5 Value semantics and requirements on class T

The `SmartArray<T>` class has *value semantics* [Reference 5, page 166]. This means that the code:

```
SmartArray<T> container(20);
T x;
container[10] = x;
```

creates a new `T` in the array `container` (by calling the class `T` assignment operator) with the same value as `x`. Subsequent changes to `x` do not affect `container[10]` and subsequent changes to `container[10]` do not affect `x`. It also means that when `container` is destroyed, the objects within it are destroyed, but `x` is unaffected.

The behavior of `T`'s assignment operator, default constructor, copy constructor, and destructor must be consistent with the value semantics behavior of the `SmartArray` class. The primary requirement is that the assignment operator and copy constructor create independent copies of the object they operate on. This is always true for built-in types and for classes that: 1) do not contain pointers and 2) rely on the compiler-generated assignment operator, default constructor, copy constructor, and destructor. An object of class `T` is said to have *remote ownership* [Reference 12] if the object contains pointers and is responsible for the data to which they point. The four compiler-generated functions (assignment operator, default constructor, copy constructor, and destructor) for objects having remote ownership will not be consistent with the value semantics of `SmartArray<T>`. Therefore these four functions must be defined by the programmer.

2.6 Pointers and names of SmartArray objects

With built-in C++ arrays, an array name is converted by the compiler to a pointer to its first element in most cases [Reference 11, Chapter 9].¹ Such a conversion is not allowed for a `SmartArray`. For example, if `b` is a `SmartArray`, then the expression `*(b+4)` is not equivalent to `b[4]` and will produce a compiler error.

When a `SmartArray` is passed to a function by value, the copy constructor for the `SmartArray` class is invoked. This in turn calls the assignment operator for type `T` for each item in the content of the array. To avoid the cost of the copy constructor when passing a `SmartArray` to a function, either a pointer or a reference to the `SmartArray` must be used.

¹ This conversion of a built-in array name to a pointer to its first element can create serious problems if a function that is expecting a built-in array of items of type `base` is passed a built-in array of items of type `derived`. The array name in the calling statement is converted to a pointer to the first element of the array, *i.e.*, it is converted to a pointer to `derived`. In the called function, this pointer is passed by value to a pointer of type `pointer-to-base` (a legitimate assignment). The compiler will then attempt to access the elements of the array using offsets based on `sizeof(base)`. Unless the size of `base` and the size of `derived` are equal (which is unlikely) serious problems will result. Since this conversion only takes place with built-in arrays, this problem cannot occur with `SmartArrays`.

3 Error handling

3.1 Introduction

In *Writing Solid Code* [Reference 2], Maguire recommends that programmers maintain two versions of their code—a *production* version and a *development* version. The production version is fast and sleek, while the development version is slow and fat because it contains *extra* error detection checks. Both versions are maintained in the same source files, and the preprocessor is used to conditionally include or exclude the extra checks.

The primary goal of production error handling is to help protect the user in case anything goes wrong; the primary goal of development error handling is to call logic errors to the programmer's attention as soon as possible after the erroneous code is executed.

Both production and development versions must consider two types of errors: *resource errors* and *logic errors*. We define resource errors as errors which can be expected to occur in a correctly coded program. An example of a resource error is `new` failing to return the requested amount of memory. Robust programs must make provisions for handling resource errors. We define logic errors as errors which should never occur in a program that is written properly—they result from incorrect code. An example of a logic error is an array bounds violation that results from improperly filtered user input.

3.2 Production error handling

Production error handling in a class library should never terminate the application program. Rather, when an error is detected by class library code, the application programmer should be notified and thus given the opportunity to take corrective action. Exceptions were added to C++ specifically to address this issue. However, because exceptions are not supported by many current compilers, the SmartArray class does not yet use them to handle errors². Instead the SmartArray class uses a combination of object error flags and member function return values. Whenever an error occurs, an internal array error flag and error message are set. The status of this array error flag can be checked at any time with the `errorCheck` member function. If this flag is set, the corresponding error message can be retrieved with the `errorMessage` member function.

3.3 Development error handling

Development error handling for logic errors is provided by the SmartArray class for increased error detection during development. It is in force whenever `assert` statements (a standard C/C++ feature) are enabled. *We recommend this approach throughout the development process because it catches programming errors even if the programmer is not checking error flags and function return values during early versions of a program.*

² Other arguments for not using exceptions for error handling are presented in References 7 and 9, but our decision was not based these arguments. Future versions of the SmartArray class will likely support error handling by throwing exceptions.

Note that the `assert` statements used by this approach will trigger (aborting the program) before the member function in which an error is detected returns.

If `assert` statements are enabled, all logic errors will be asserted and thus the application will be halted. Of the five categories of errors summarized in Table 3.1, only memory allocation resource errors do not result in program termination. Memory allocation resource errors are handled identically whether or not `assert` statements are enabled.

For increased speed and to allow error recovery, we recommend that any production version of a program using the `SmartArray` class be compiled with `assert` statements disabled, and that error flags and member functions be checked in the application program as appropriate.

3.4 Overview of error types

Error Type	Production Error Handling (asserts disabled)	Development Error Handling (asserts enabled)
<i>memory allocation resource errors</i> occur when internal use of the new operator cannot provide the requested storage.	Store an error message in the array and set the error flag to true, but do not terminate the program. The programmer can define a new_handler function to detect and handle the error if desired.	
<i>array bound logic errors</i> occur when an index outside the bounds of an array is supplied as an argument to applicable member functions or operators.	Store an error message in the array and set the error flag to true, but do not terminate the program. The [] operator returns a reference to an auxiliary object of type T. (See the errorValue member function and the discussion in Section 3.5) The item member function, which is designed for maximum performance, does not check for array bound logic errors. However, this function contains assert statements that do check bounds to facilitate debugging during development.	Terminate program on an assert statement.
<i>traversal logic errors</i> occur when current, currentItem, or index are called without appropriate previous calls to start and next.	For current, store an error message in the array and set the error flag to true, but do not terminate the program. In addition, return a reference to an auxiliary object of type T. (See the errorValue member function and the discussion in Section 3.5) The currentItem and index member functions do not check for traversal logic errors. However, these functions contains assert statements that do check traversals to facilitate debugging during development.	Terminate program on an assert statement.
<i>general logic errors</i> are programmer errors which do not fit into any of the other categories. They include attempting to stop a traversal which has not been started and supplying arguments outside of the content limits of an array.	Store an error message in the array and set the error flag to true, but do not terminate the program. The put, append, remove, and insert member functions return a value of false when supplied index arguments outside the content limits of the source array.	Terminate program on an assert statement.
<i>library logic errors</i> occur when internal consistency checks detect that the SmartArray is in an invalid state. The SmartArray class is designed so that this should never happen.	Store an error message in the array and set the error flag to true, but do not terminate the program.	Terminate program on an assert statement.

Table 3.1 Summary of Error Handling in the SmartArray Class

3.5 Array bound and traversal logic errors

If an attempt is made to access an array item with an index outside of the range `lower_bound ... upper_bound` with the `[]` operators, an array bound logic error will occur. If a traversal is attempted without appropriate calls to `start` and `next`, a traversal logic error will occur. Array bound and traversal logic errors are handled similarly. This section expands on the summary provided in Table 3.1.

Development error handling: With development error handling enabled, the program will terminate on an `assert` statement when an array bound or traversal logic error occurs.

Production error handling: With production error handling enabled, we have chosen not to halt the application program when an array bound or traversal logic error occurs. Instead, a legitimate reference to an auxiliary object of type *T* is returned (since some reference must be returned by `operator[]` and `current`). The value stored in this auxiliary object is called the *error value* and is initially the value put there by the default constructor for *T*. The error value for built-in types is initially undefined since built-in types do not have a default constructor. The error value can be retrieved with the `errorValue()` member function, and set with the `errorValue(value)` member function.

Four alternative approaches for production array bound and traversal logic error detection are possible:

First, the programmer could check the value of the array error flag using the `errorCheck` member function.

Second, the programmer could set the auxiliary object to an object of type *T* designed to flag an error. For the case of simple built-in types this could simply be an invalid value that would cause problems on attempted use. For a user-defined type, this could be an object designed to issue a message to gracefully terminate the application on attempted use.

Third, the programmer could compare the value of any reference returned by the `[]` operator to the error value—equality indicating an error.

Finally, the programmer could compare the *address* of any reference returned by the `[]` operators to the address of the reference returned by `errorValue`—with equality again indicating an error.

3.6 The preset value

The problem: What value does an array item have when it has never been initialized by the programmer? For example, after the declaration of a `SmartArray`

```
SmartArray<Animals> b(100);
```

what is the value of `b[10]`? The answer is that `b[10]` will have the value set by the default constructor for class `Animals`. For the following declaration,

```
SmartArray<int> b(100);
```

the value of `b[10]` will be undefined (and may be different each time the program is run). This is because the built-in type `int` does not have a default constructor. If the uninitialized `b[10]` is later used in the program, the behavior of the program may not be repeatable from run to run, making it difficult to debug.

The solution: The SmartArray class allows the programmer to specify a *preset value* for the array both to enable recognition of uninitialized array items and to ensure repeatability. Each item in the capacity of the array is assigned this value when the array is constructed. Then, whenever this value is detected (in a debugger, with program code, or in program output), it is clear that the array item was never initialized. The preset value is specified using a SmartArray constructor.

If the content and capacity of an array could not change, then SmartArray's use of the preset value at construction only would be sufficient. However, since the content and capacity can change, the preset value must be properly used by the SmartArray during its lifetime. For example, when the upper bound of an array is increased, the added items are set to the preset value. Additionally, if the upper content limit of an array is decreased, then the items moved out of the content are restored to the preset value.

In summary, specification of the preset value in the constructor will enable the SmartArray class to ensure that any items which are either outside the content of the array (but within the capacity) or which are within the content of the array but not yet set by the programmer will contain the preset value. See also the `put` member function (Section 4.9) which, unlike the use of the preset value during construction, can fill all or part of an array with data items resulting in an array with non-zero content.

3.7 Data shredding

Data shredding—intentionally overwriting all items in an array's capacity with a special known value—is done only if the SmartArray class has been compiled with `assert` statements enabled. It provides increased error detection during development.

The problem: As the capacity of a SmartArray changes, it requests memory from and returns memory to the heap using `new` and `delete`. However, because SmartArrays provide the application programmer with references to memory where array data items are stored, it is possible that the application program could retain references or pointers to released memory. Furthermore, if that memory has not been overwritten by the “system memory manager”, such a pointer or reference could appear to point to valid data.

The solution: Before a SmartArray releases memory to the heap, it shreds the data by overwriting each item in the array's capacity with the error value explained in Section 3.5. Shredding prevents the programmer from retrieving data that appears valid from a released portion of memory [Reference 2]. Generally such a retrieval would have to be fairly subversive (*i.e.*, involve pointers or references) since both `operator[]` and the `item` member functions (when `assert` statements are enabled) will never allow such access.

4 Member functions and operators

This section describes the public member functions and operators provided by the SmartArray class. It is divided into the following sections:

- Constructors
- Destructor

- Copy constructor
- Item access with bounds checking (`operator[]`)
- Capacity inspection functions
- Content functions
- Capacity change functions
- Assignment operator
- Editing functions
- Traversal functions
- Error handling functions

4.1 Constructors

`SmartArray(int upper_bound)`³

This constructor creates an array with a capacity of $(\text{upper_bound} + 1)$ items of type T. The default constructor for type T is called to initialize each item.

If the constructor is unable to allocate sufficient memory, a memory allocation resource error occurs, and an array with zero capacity will be constructed.

Note that for built-in C++ arrays the statement

```
int a[10];
```

allocates 10 integer storage locations (from 0 to 9), while for `SmartArray` objects the statement

```
SmartArray<int> a(10)
```

allocates 11 integer storage locations (from 0 to 10).

`SmartArray(int lower_bound, int upper_bound)`

This constructor creates an array with a capacity of $(\text{upper_bound} - \text{lower_bound} + 1)$ items of type T. The default constructor for type T is called to initialize each item.

If the constructor is unable to allocate sufficient memory, a memory allocation resource error occurs, and an array with zero capacity will be constructed.

`SmartArray(int lower_bound, int upper_bound, T presetValue)`

This is the most general form of the array constructor. It creates an array with a capacity of $(\text{upper_bound} - \text{lower_bound} + 1)$ items of type T. The default constructor for type T is called to initialize each item.

³ Single-argument constructors enable the compiler to perform implicit type conversions. Since such conversions are often undesirable (they can lead to subtle coding errors), the `SmartArray` class currently uses a proxy class to disable implicit type conversions. Future versions of `SmartArray` will use the new `explicit` keyword to accomplish this. See Meyers [Reference 18, Item 5] for a complete discussion of this issue.

Then each item is set to the value `presetValue`. The `presetValue` is used to aid detection of uninitialized array items as described in Section 3.6. The content of the array remains zero items.

If the constructor is unable to allocate sufficient memory, a memory allocation resource error occurs, and an array with zero capacity will be constructed.

SmartArray(int lower_bound, int upper_bound, T *pExternalData)

This constructor allows a `SmartArray` to be constructed from a pointer to external data. It creates a full-featured `SmartArray` with one exception—the capacity of the array cannot be changed. The programmer is responsible both for allocating the memory pointed to by the `pExternalData` pointer and for releasing this memory after the array is destroyed. Additionally, data shredding is not done since the array does not own (and therefore is not releasing) the memory pointed to by `pExternalData`.

Unlike all other constructors (except the copy constructor), this constructor results in an array with non-zero content. It sets `upper_content_limit` to `upper_bound`, thus including all of the external data in the content of the array at construction.

The only restriction on the use of an array constructed this way is that the capacity of the array cannot be changed. Therefore any attempt to directly call the `upperBound` member function to change the array's capacity will result in a general logic error. In addition, the member functions `upperBoundNeeded`, `put`, `append`, `insert`, `import`, and `operator=` will result in a general logic error if they attempt to increase the capacity of the array.

See Sections 6.2 and 7.9 for examples using this constructor.

SmartArray()

The default (no argument) constructor creates an array with a capacity of zero items. No memory is allocated, the lower bound is set to 0, and the upper bound is set to -1. This results in both a content of zero items and a capacity of zero items. The array capacity can later be changed directly by the programmer with the `upperBound` member function and indirectly by several other member functions (`upperBoundNeeded`, `put`, `append`, `insert`, `import`, and `operator=`). This constructor is particularly useful when the size of the array is not known *a priori*.

4.2 Destructor

~SmartArray()

The destructor calls the class `T` destructor for each item in the array whose index lies within the range `lower_bound ... upper_bound`. (Note, however, that this step is skipped for arrays using external data—see Sections 4.1 and 6.2.) In addition, any memory directly allocated with `new` by member functions of the `SmartArray` class during the lifetime of the array is deleted.

In addition to its traditional role of releasing resources that were allocated to an array during its lifetime, the destructor takes two steps to aid the programmer in detecting the often difficult to find bug of using a pointer to an object which has

been previously deleted. First, the destructor leaves the array in an error-flagged state, and, second, the capacity and content are set to zero items. Thus, until the system reuses the memory in which the deleted array object is stored, any call to `errorCheck` will return `true`, and attempts to access array data via array member functions will cause an array bound logic error (since the capacity is zero items).

4.3 Copy constructor

`SmartArray(const SmartArray<T>& source)`

The expression `SmartArray<T> a = source` results in `a` being an exact copy of `source`, except that the capacity of `a` is just sufficient to hold the content of `source`.

Thus, the upper bound of `a` is just sufficient to hold the content of `source`, *i.e.*, the upper bound of `a` and the upper content limit of `a` will be equal. This avoids the cost of allocating more memory than necessary to hold the content. The assignment operator for type `T` is used to copy the array items from the array `source` to the newly constructed array `a`.

4.4 Item access with bounds checking (operator[])

The `[ ]` operator allows the `SmartArray` class to behave much like built-in C++ arrays by returning a reference to an item of type `T`.

This operator returns a reference that can be invalidated by subsequent operations on the array which result in a change in the array's capacity. The reference should be used only in a localized context.

Unlike built-in C++ arrays, the `[ ]` operator performs bounds checking on the supplied index. If the index is outside of the range `lower_bound ... upper_bound`, an array bound logic error occurs. See Chapter 3 and Section 4.11 for a complete discussion of error handling in the `SmartArray` class. The `item` member function (Section 5.2), which does not perform bounds checking, is provided for those cases in which code performance is critical.

Note that the `[ ]` operator will never result in the array changing its capacity. This is because capacity changes always move array data to a different location in memory. Such data movement can invalidate other references returned by the `[ ]` operator elsewhere in the same expression. In contrast, the `put` member function (described in Section 4.9) causes the array to grow automatically as items are put into it.

Indices outside of the content limits but within the bounds of an array are handled differently by non-const and const arrays as described below.

T& operator[] (int index)

This operator returns a reference to an item of type `T` stored at location `index`.

If `index` is greater than `upper_content_limit` (but within the range `lower_bound ... upper_bound`), `upper_content_limit` is set to `index`. This allows the array to automatically increase its content.

```
const T& operator[] (int index) const
```

This operator returns a `const` reference to an item of type `T` stored at location `index`.

If `index` is greater than `upper_content_limit` (but within the range `lower_bound ... upper_bound`), a general logic error occurs. This is because any attempt to reference items outside the content limits of a `const` array must necessarily be a logic error.

4.5 Capacity inspection functions

```
int capacity() const
```

Returns the number of items in the capacity of an array.

```
int lowerBound() const
```

Returns the lower bound of the array.

Note that once `lower_bound` is set by the constructor it can be changed only by the assignment operator or the `reindex` member function. An array on the left-hand side of an assignment operator assumes the lower bound of the right-hand side array.

```
int upperBound() const
```

Returns the upper bound of the array.

This member function name is overloaded (see Section 4.7).

```
bool isDataExternal() const
```

Returns `true` if the array was constructed using a pointer to external data.

4.6 Content functions

```
int content() const
```

Returns the number of items in the content of an array.

```
int lowerContentLimit() const
```

Returns the lower content limit of the array (which is always equal to the array's lower bound.)

```
int upperContentLimit() const
```

Returns the upper content limit of the array.

```
bool upperContentLimit(int newUpperContentLimit)
```

This member function sets the upper content limit to `newUpperContentLimit`. Since the upper content limit is set internally when the `[ ]` operator or similar functions (such as `put`) are used, the programmer should generally not use this member function. However, when traversal functions (see Section 4.10) are used to fill an array, the upper content limit must be set before the traversal is started. This member function returns `true` if `newUpperContentLimit` is within the bounds of the

array. Otherwise an array bound logic error will occur and the function will return `false`.

`void empty()`

This member function sets the upper content limit to `lower_bound - 1`, resulting in a content of zero items. It also resets the traversal stack (Section 5.1). It does not change the bounds of the array and consequently does not free memory allocated for the array. It does not change the preset value (Section 3.6) or the value of the auxiliary object (Section 3.5).

4.7 Capacity change functions

The capacity of an array can be changed only by changing the upper bound. In order to change the capacity of an array, sufficient memory to hold both the old and new capacities must momentarily coexist.

`bool upperBound(int newUpperBound)`

The upper bound of the array is changed to `newUpperBound`. A new area of memory is allocated for the range `lower_bound ... newUpperBound`, the contents are copied item by item from the old area of memory using the class `T` assignment operator, and finally the original area of memory is deleted, causing the class `T` destructor to be called for each of the original items. Although this approach can be expensive, it is more robust than a block memory move [Reference 12, items 208 and 210]. In a properly designed program, array resizing should occur only occasionally.

If this member function is used to increase the upper bound of the array, the newly added locations are set using the default constructor or the `presetValue` (see Section 3.6).

A memory allocation resource error occurs and this member function returns `false` if the attempt to change the capacity of the array fails. A return value of `true` indicates success.

This member function name is overloaded (see Section 4.5).

`bool upperBoundNeeded(int minUpperBound)`

This member function will ensure that the upper bound of the array is at least `minUpperBound`. If `minUpperBound` is less than or equal to the current upper bound, no action is taken. If the `minUpperBound` is larger than the current upper bound, then the upper bound will be increased such that the capacity is at least doubled.

By using `upperBoundNeeded` (inside a loop, for example) the programmer can avoid repetitive and expensive calls to `upperBound`. The geometric growth of the capacity (allocated memory) helps to ensure that a minimum number of capacity resizes will be done. The resulting array can, however, have nearly twice the capacity necessary to hold its content. If the desired final capacity is known *a priori*, it should be set directly with the `upperBound` member function outside the loop. If `upperBoundNeeded` is used within a loop, the resulting array can have its capacity subsequently reduced (set equal to its content) with the statement `a.upperBound(a.upperContentLimit())`.

Consider the following code fragment.

```
SmartArray<int> b(1,3);
for (int i = 1; i <= 10; i++) {
    b.upperBoundNeeded(i);
    b[i] = i;
}
```

As the loop progresses, the upper bound of the array will change from the original 3 to 6, and then to 12.

A memory allocation resource error occurs and this member function returns `false` if an attempt to increase the capacity of the array fails. A return value of `true` indicates success.

4.8 Assignment operator

```
SmartArray<T>& operator= (const SmartArray<T>& rhs)
```

Let both `a` and `b` be instances of the `SmartArray` class. The expression `a = b` results in `a` being an exact copy of `b`, except that the capacity of `a` is not increased unless necessary.

Specifically, the expression `a = b` results in the content of `a` becoming the same as the content of `b`. The lower bound of `a` will be changed if necessary to match the lower bound of `b`. The upper bound of `a` will be increased only if necessary to allow `a` to hold the content of `b`. Therefore the upper bounds of `a` and `b` will not necessarily be equal after the assignment. The array `a` will be assigned `b`'s error value, preset value (if any), error flag, and error message. In addition, array `a` will be assigned the current state of any of `b`'s in-progress traversals.

The assignment operator for type `T` is used to copy the array items.

A memory allocation resource error occurs if an attempt to increase the capacity of the left-hand side array fails.

4.9 Editing functions

```
bool put(const T& sourceValue)
bool put(const T& sourceValue, int index)
bool put(const T& sourceValue, int from, int to)
```

The `put` member function is used to overwrite array items, specified by a range of indices, with value `sourceValue`. If called with only a `sourceValue`, the entire capacity of the array from `lower_bound` to `upper_bound` will be overwritten. If called with a second argument `index`, the one item in the array specified by `index` will be overwritten (this is equivalent to `a[index] = sourceValue`). If called with three arguments, the items from `index from` to `index to` will be overwritten. The assignment operator for type `T` is used to put `sourceValue` into the array. The content of the array will be increased, if necessary, to include any locations written to by `put`.

In either of the last two cases, if either `index`, `from`, or `to` is less than `lower_bound` an array bounds logic error will occur and `put` will return `false`. If `from` is greater than `to` a general logic error will occur and `put` will return `false`. If either `index`, `from`, or `to` is larger than `upper_bound`, the member function `upperBoundNeeded` will be called to increase `upper_bound`.

A memory allocation resource error occurs and this member function returns `false` if an attempt to increase the capacity of the array fails. A return value of `true` indicates success.

```
bool insert(const T& sourceItem, int index)
bool insert(const SmartArray<T>& sourceArray, int index)
bool insert(const SmartArray<T>& sourceArray, int from, int
            to, int index)
```

Causes an array to increase its content by inserting:

- the single item `sourceItem`
- all the items from an array `sourceArray`
- a sequence of items from an array `sourceArray`

at the index given by `index`. Items originally at indices `index` through `upper_content_limit`, if any, are shifted to higher indices by an amount equal to the number of source items. The array will increase its capacity if necessary. The assignment operator for type `T` is used to copy the items.

Insertion at the end of the array occurs if `index = upper_content_limit + 1`. See the `append` member function below.

A general logic error occurs and `false` is returned if `index` is outside the range `lower_content_limit ... upper_content_limit + 1`, or if `from` is greater than `to`.

A memory allocation resource error occurs and this member function returns `false` if an attempt to increase the capacity of the array fails. A return value of `true` indicates success.

```
bool append(const T& sourceItem)
bool append(const SmartArray<T>& sourceArray)
bool append(const SmartArray<T>& sourceArray, int from, int to)
```

Causes an array to increase its content by appending:

- the single item `sourceItem`
- all the items from an array `sourceArray`
- a sequence of items from an array `sourceArray`

at the end of the content of the array.

The expressions

```
a.append(t)
a.append(b)
a.append(b, from, to)
```

are equivalent to

```

a.insert(t, a.upperContentLimit()+1)
a.insert(b, a.upperContentLimit()+1)
a.insert(b, from, to, a.upperContentLimit()+1)

```

A memory allocation resource error occurs and this member function returns `false` if an attempt to increase the capacity of the array fails. A return value of `true` indicates success.

```
bool remove(int from, int to)
```

Causes an array to reduce its content by removing the items in the array from index `from` through index `to`. Items originally at indices `to + 1` through `upper_content_limit`, if any, are shifted to lower indices by an amount equal to the number of items in the range `from` through `to`. The assignment operator for type `T` is used to move the items.

A general logic error occurs and `false` is returned if either index is outside the range `lower_content_limit ... upper_content_limit`, or if `from` is greater than `to`. A return value of `true` indicates success.

```
void reverse()
```

Reverses the order of the items in the content of the array in place. This can be useful after a sort operation. Sorting is not provided by the `SmartArray` class.

```
bool import(const T* pSource, int length)
bool import(const T* pSource, int length, int index)
```

The purpose of this member function is to transfer data to a `SmartArray` from a built-in C++ array or other location specified by a pointer to `T`.

It overwrites the content of the array with `length` items copied from `pSource`. The first data item (transferred from `pSource[0]`) will have index `lower_bound` if `index` is not specified and will have index `index` otherwise. If specified, `index` must be greater than or equal to the lower bound.

The array will increase its capacity if necessary. Any items in the array prior to the call to `import` and not within the range affected by `import` (`index` through `index + length - 1`) are unchanged. The assignment operator for type `T` is used to copy the items.

A memory allocation resource error occurs and this member function returns `false` if an attempt to increase the capacity of the array fails. A return value of `true` indicates success.

```
bool export(T* pDestination, int length) const
bool export(T* pDestination, int length, int index) const
```

The purpose of this member function is to transfer data from a `SmartArray` to a built-in C++ array or other location specified by a pointer to `T`.

It overwrites the memory starting at location `pDestination` with `length` items copied from the `SmartArray`. The first data item copied from the `SmartArray` will have index `lower_bound` if `index` is not specified and will have index `index`

otherwise. If specified, `index` must be greater than or equal to the lower content limit and less than or equal to the upper content limit.

The programmer must ensure that sufficient memory has been allocated, beginning at location `pDestination`, to hold `length` items of type `T`. The assignment operator for type `T` is used to copy the items.

If the value of `length` and/or `index` is such that the array's upper content limit is reached prior to `length` items being copied, a general logic error occurs and `export` returns `false`. This indicates the failure of `export` to copy all requested items.

A general logic error occurs and `false` is returned if `index` is outside the range `lower_content_limit ... upper_content_limit`.

A return value of `true` indicates that `length` items were copied to location `pDestination`.

4.10 Traversal functions

The iterator class approach, often provided by C++ container classes to facilitate traversal of a container, can be useful for arrays also. This approach avoids the “off-by-one” errors which often plague programmers accessing arrays with an integer counter and `operator[]`. However, the required use of the additional iterator object clutters the traversal code. Therefore, rather than supply an accompanying iterator class for the `SmartArray` class, we have chosen to supply built-in traversal functions. The traditional objection to built-in traversal functions—difficulty with simultaneous traversals—is overcome by supplying a traversal stack as described in Section 5.1. We believe eliminating an extra iterator object results in clearer code.

The traversal functions allow the items in the content of an array to be processed without explicit reference to the content limits.

In the simplest form, an array traversal consists of

- a call to `start` to initialize the traversal.
- calls to `next` to increment the traversal position
- calls to `current` to access the current traversal item
- a call to `stop` to halt the traversal.

For example:

```
SmartArray<int> a(1,8);
a[1] = 22;
a[2] = 32;
a[3] = 36;
// at this point the content limits of a are 1 and 3.

a.start();
while (a.next())
    f(a.current());
a.stop();
```

This loop will call `f` with `a[1]`, `a[2]` and `a[3]`. Note that the coding of the traversal loop does not directly use the bounds or content limits of the array.

See Section 5.1 for a description of advanced traversals.

bool start() const

This member function must be called before any traversal is begun. It initializes the traversal, setting the traversal position so that `next` must be used before attempting to access the first item. It returns `true` if the traversal initialization was successful, `false` otherwise.

An unsuccessful initialization can occur only when `new` fails to allocate the small amount of memory necessary to maintain a traversal state stack. As discussed in Section 3.4, such a memory allocation resource error can be handled by a programmer-defined `new_handler` function.

bool next() const

This member function increments the traversal position. It returns `true` if one or more items are available for access via the `current` member function. Otherwise it returns `false`. A traversal starts at the item with index `lower_content_limit` and proceeds through the item with index `upper_content_limit`.

T& current()

const T& current() const

Return a reference to the current item in the traversal. A `const` reference is returned for a `const` array. A call to `current` while a traversal is not in progress results in a traversal logic error.

int index() const

Return the index of the current item in the traversal. A call to `index` while a traversal is not in progress results in a traversal logic error, and the returned value is undefined.

bool stop() const

All traversals (which must be started with a call to `start`) must be halted by the `stop` member function. Calls to the member functions `start` and `stop` must be balanced. Thus any transfer out of a traversal loop must ensure that the `stop` member function is called.

If `stop` is called when a traversal is not in progress (`start` has not been called), a general logic error will result and `stop` will return `false`. Otherwise it returns `true`.

The destructor will assert that all traversals have been properly halted.

Note: When traversal functions are used to fill an array the upper content limit must be increased before the traversal is started. This is done with the `upperContentLimit(int)` member function (see Sections 4.6 and 7.3).

4.11 Error handling functions

bool errorCheck() const

Returns `true` if the error flag for a `SmartArray` is set, `false` otherwise. When an error causes an array's error flag to be set, this flag remains set until cleared with the `errorClear` member function. The text of the error message can be retrieved with the `errorMessage` member function. The error flag will be set for any `SmartArray` using the contents of an array whose error flag is set. Member functions which cause an array to use the contents of another array are: `append`, `insert`, `operator=`, and the copy constructor.

const char* errorMessage() const

Returns a pointer to the error message text. This message is updated as new errors are encountered.

void errorClear() const

Clears the error flag and the error message to set the array to a non-error state.

void errorValue(const T& value)

const T& errorValue() const

The `errorValue()` member function returns a reference to the auxiliary object described in Section 3.5. The value stored in this auxiliary object is called the *error value* and is initially the value put there by the default constructor for `T`. The error value for built-in types is initially undefined since built-in types do not have a default constructor. However, the error value can be retrieved with the `errorValue()` member function, and set with the `errorValue(value)` member function.

bool isPresetUsed() const

Returns `true` if a preset value has been specified for the array.

const T& presetValue() const

Returns a reference to the preset value. The referent is undefined if a preset value has not been specified for the array.

5 Advanced features

5.1 Advanced traversals

As stated in Section 4.10, rather than supply an accompanying iterator class for the `SmartArray` class, we have chosen to supply traversal functions. A possible objection to this strategy is that simultaneous, independent traversals of an array are not possible using traversal functions. This is partially addressed by providing a traversal state stack

allowing nested traversals.⁴ This stack is automatically maintained by the `start` and `stop` traversal functions—with `start` always pushing the stack and `stop` always popping it. The stack automatically expands itself resulting in an effectively infinite stack, limited only by available machine memory. Traversals may thus be nested to any depth.

As mentioned earlier, all traversals must be terminated with a `stop` member function call—including traversals which are terminated before a complete traversal of every item in the array. During development error handling (`assert` statements enabled) the destructor will `assert` that all `start` member functions had a matching `stop`. The programmer can use the `traversalStackDepth` member function any time a traversal is not in progress to verify that `start` and `stop` member function calls are balanced. If `stop` is called when a traversal is not in progress, a general logic error will result since `stop` will attempt to pop an empty stack.

```
int traversalStackDepth() const
```

This member function can be checked at any time to determine the number of traversal states pushed onto the traversal state stack. This stack depth check is useful to check the logic of nested traversals. After the code for a nested traversal has been completed, the stack depth should be checked to ensure that it is zero (*i.e.*, the stack is empty). As noted in the above discussion on nested traversals, the destructor will verify (with an `assert` statement) that the traversal stack is empty.

The `isFinished` member function is not limited to nested traversals, but is described here because of its limited applicability.

```
bool isFinished()
```

This member function is primarily for internal use by the `next` member function and its use by the programmer is discouraged. However, since it is useful in certain forms of traversal loops it has been made available. It will evaluate to `true` if the previous call to `next` completed the traversal (*i.e.*, `next` returned `false`) and a new traversal has not been started by calling `start`.

5.2 Item access without bounds checking

```
T& item (int index)
```

```
const T& item (int index) const
```

This member function returns a reference to an item of type `T` stored at location `index` and is thus similar to the `[ ]` operator. Unlike the `[ ]` operator (Section 4.4), however, *the item member function does not perform bounds checking for production code* (`assert` statements disabled) and consequently should be used only for those cases in which code performance is critical. The programmer is responsible for ensuring that the index is within the range `lower_bound ... upper_bound`. If not, no error will be detected and a reference to a memory location

⁴ Of course, fully independent simultaneous traversals can be programmed directly using `for` loops and the content limit functions.

outside of the array will be returned. This will likely result in a bug that is difficult to detect.

Note also that, unlike the `[]` operator, *the `item` member function does not change the upper content limit*. If the array is being written to (as in the following example) the programmer must update the upper content limit with a call to the `upperContentLimit` member function.

Our recommendation is to always use the `[]` operator to reference array items during development and switch to the `item` member function (and programmer setting of the upper content limit) only where absolutely necessary to achieve peak performance in production code. If `assert` statements are enabled (which we recommend for development) bounds checking will be done via `assert` statements in the `item` member functions. This protection is lost, however, as soon as `assert` statements are disabled—which they typically are for a production release of the code. Therefore, bounds checking logic should always be provided by the programmer if the `item` member function is used.

Consider the following loop:

```
for ( int index = a.lowerBound(); index <= a.upperBound(); index++) {
    // The following expression is equivalent to a[index] =
    // fillValue; but without bounds checking.
    a.item(index) = fillValue;
}
// update the upper content limit since item does not
a.upperContentLimit(a.upperBound());
```

This code is safe since a valid range of the index has been provided by the array.

T& currentItem()

const T& currentItem() const

This member function returns a reference to the current item in the traversal and is thus similar to `current`. Unlike the `current` member function, however, *the `currentItem` member function does not perform bounds checking for production code* (`assert` statements disabled) and consequently should be used only for those cases in which code performance is critical. The relationship between `currentItem` and `current` is similar to that between `item` and `operator[]`.

5.3 Reindexing

An array's indexing scheme, selected at construction by the specification of the array's bounds, is typically not changed during its lifetime. A notable exception to this is an array on the left-hand side of an assignment statement which assumes the content and indexing scheme of the right-hand side array. However, it is occasionally necessary to reindex an array containing data without modifying the data.

Consider a routine expecting to receive data in an array with a lower bound of zero. The programmer instead passes this routine an array with a lower bound of one. Such a routine can then either flag an error condition or reindex the user-supplied array and continue. The `reindex` member function enables the second solution.


```
bool reindex(int offset)
```

This member function changes the complete indexing scheme for the array. It adds the value `offset` to the array's lower bound, upper bound, lower content limit, upper content limit, and (conceptually) the index of every data item currently in the array. Neither the array's capacity or content is changed and no data is physically moved in memory but rather the pointer to the array's data is adjusted—making this routine quite efficient. Traversals which are in progress (even nested ones) will successfully complete (*i.e.*, no items will be missed) if the array is reindexed during the traversal. This member function always returns `true`.

Note: *the indices of all data items in the array are changed by a call to this member function.*

Consider the following example:

```
SmartArray<int> a(1,3);
a[1] = 11;
a[2] = 12;

// The following assertions are all true.
assert (a.lowerBound() == 1);
assert (a.upperBound() == 3);
assert (a.lowerContentLimit() == 1);
assert (a.upperContentLimit() == 2);
assert (a[1] == 11);
assert (a[2] == 12);

a.reindex(-1);

// The following assertions are all true.
assert (a.lowerBound() == 0);
assert (a.upperBound() == 2);
assert (a.lowerContentLimit() == 0);
assert (a.upperContentLimit() == 1);
assert (a[0] == 11);
assert (a[1] == 12);
```

Thus the data initially deposited at the lower bound of the array remains at the lower bound of the array after the call to `reindex` but is indexed differently.

6 Retrofitting SmartArrays into existing code

The `SmartArray` class has been designed for usage that closely parallels that of a built-in C++ array. Retrofitting `SmartArrays` into existing code is therefore a viable option for programmers desiring the features offered by the `SmartArray` class.

There are two major considerations when retrofitting the `SmartArray` class into existing code:

- replacing built-in arrays with `SmartArrays`
- interfacing `SmartArrays` with existing arrays or libraries in the program.

6.1 Replacing built-in arrays with SmartArrays

Replacing a built-in array with a SmartArray involves two steps:

- replacing the definitions of built-in arrays with definitions of SmartArrays
- eliminating any use of the built-in array name which resulted in its conversion to a pointer as discussed in Section 2.6.

Replacing definitions

In many occurrences, the built-in C++ array definition:

```
float a[10];
```

can be replaced by the SmartArray definition:

```
SmartArray<float> a(10);
```

Note that the square brackets of the built-in array definition are replaced by the parentheses of the SmartArray constructor call.

In the example above, the single argument constructor is called with the same value (10) as appears in the built-in array definition. Note that while the built-in array definition shown results in `a[0]` through `a[9]` being allocated, the SmartArray definition results in `a[0]` through `a[10]` being allocated since the constructor's argument is the upper bound of the array—not the array's size. To avoid “wasting” the single array location `a[10]`, the constructor can be called with an argument one less than the index used in the built-in array definition as in:

```
SmartArray<float> a(9);
```

If `a` is being declared as part of a `class` or `struct` declaration, the argument to the SmartArray constructor is not allowable C++ syntax. That is,

```
class b {
    float a[10];
};
```

is allowable, but

```
class b {
    SmartArray<float> a(10);    // syntax error
};
```

is not. Instead, `class b` must be declared as

```
class b {
    SmartArray<float> a;
    b() : a(10) {};
};
```

where the constructor for `a` is called from the initializer list of `b`'s constructor. Alternatively, the member function `a.upperBound(10)` could be called from within `b`'s constructor.

Eliminating array name-pointer conversions

Since a built-in C++ array name is converted to a pointer to its first element, any expression containing a built-in array name without square brackets must be rewritten when SmartArrays are substituted. For example, the expression `*(b+4)` must be replaced by `b[4]`. Loops (within routines that have been passed a pointer to an array) using

pointer notation such as `*b++ = i` must be rewritten to use indicial notation such as `b[j] = i`.

When a built-in array is passed to a function, the function receives a pointer to the array's first element and must be prototyped accordingly. For example:

```
void myFunction(float *b); // OR void myFunction(float b[]);
...
void main () {
    float a[10];
    ...
    myFunction(a);
```

When retrofitted to use a `SmartArray`, this code becomes:

```
void myFunction(SmartArray<float>& b);
...
void main () {
    SmartArray<float> a(10);
    ...
    myFunction(a);
```

Thus references to `SmartArrays` replace pointers to the first item of built-in arrays. Note that only the function prototype and definition—not any of the function calls—need to be changed.

6.2 Interfacing SmartArrays with pointers or built-in arrays

There are three common types of interaction that `SmartArrays` can have with a pointer to contiguous memory such as a built-in array name or heap memory allocated by `new` and accessed by a pointer.

- copying `SmartArray` data to or from a pointer to contiguous memory
- accessing `SmartArray` data via a pointer to contiguous memory
- constructing a `SmartArray` by embedding a pointer to externally allocated contiguous memory.

Copying SmartArray data to or from a pointer

The `SmartArray` class provides two member functions, `import` and `export` (Section 4.9), to ease the task of copying data into or out of a `SmartArray`.

Given a built-in array `b`, the data in `b` is copied into the `SmartArray` `a` with the code

```
int b[10];
...
SmartArray<int> a(9);
a.import(b,10); // copy 10 data items from b into SmartArray a
```

And given a built-in array `c`, the data in `SmartArray` `a` is copied into `c` with the code

```
SmartArray<int> a(9);
...
int c[10];
a.export(c,10); // copy 10 data items from SmartArray a into c
```

Accessing SmartArray data via a pointer

Accessing SmartArray data via a pointer is essential when working with libraries that expect a pointer to contiguous memory. The SmartArray class provides a mechanism for obtaining a pointer to a SmartArray's data. Data items in a SmartArray are stored in contiguous memory which was obtained by SmartArray's use of the `new` operator. Because the SmartArray `operator[]` returns a reference to a data item, using the address-of operator (`&`) on any such reference will yield a pointer to this data item. As with built-in arrays, the programmer must ensure that any use of the pointer will not attempt to access data outside the current bounds of the SmartArray. Consider the following example:

```
void sort(int *pa, int n);          // prototype
SmartArray<int> a(9);
...
int *pa = &a[0]; // get a pointer to first data item in a
sort(pa, 10); // sort the 10 items in SmartArray a in place
```

Constructing a SmartArray by embedding a pointer to externally allocated contiguous memory

Normally a SmartArray allocates and deallocates the memory used for array data. With this approach, filling a SmartArray with existing data (using `import` for example) requires making an independent copy of the data. This can be prohibitively expensive when working with large amounts of data. To avoid this copying, a SmartArray can be constructed by embedding a pointer to externally allocated contiguous memory—memory that is allocated and deallocated outside of the SmartArray.

The net effect of using the external pointer constructor is to “wrap” a SmartArray around external data (memory) for the purpose of providing a SmartArray interface. When the SmartArray is destroyed, the external data remains intact.

Consider the following example where the programmer has the data in a built-in array and wishes to call a sorting routine expecting a SmartArray:

```
void sort(SmartArray<int>& a); // prototype
int b[10];
...
SmartArray<int> a(0,9,b);      // "wrap" SmartArray "a" around b's data
sort(a); // sort the 10 items in built-in array b in place
```

Alternatively, a temporary SmartArray could be constructed simply for the call to `sort`:

```
sort(SmartArray<int> (0,9,b)); // sort the 10 items in built-in array b in place
```

See Section 4.1 for documentation of the external pointer constructor, Section 4.5 for documentation of the `isDataExternal` function, and Section 7.9 for another example.

7 Programming examples

7.1 Constructor and operator[]

This example illustrates the use of the constructor and `operator[]`.

```

#include "SmartArray.h"

void main() {
    SmartArray<int> a(1,5);
    for (int i = 1, i <= 5; i++)
        a[i] = i;
}

```

7.2 Traversal functions

This example illustrates use of the traversal functions as well as the `put` member function. An alternate approach using the content limit functions is also shown.

```

#include <iostream.h>
#include "SmartArray.h"

void main() {
    SmartArray<int> a(1,5);

    a.put(100,1,2);
    a.put(200,3,5);

    cout << sum(a);
}

int sum(SmartArray<int>& b) {
    int answer = 0;

    b.start();
    while (b.next())
        answer += b.current();
    b.stop();

    return answer;
}

```

The printed output is:

800

By using content functions and a `for` loop instead of the traversal functions, the function `sum` could also be written as:

```

int sum(SmartArray<int>& b) {
    int answer = 0;

    for (int i = b.lowerContentLimit(); i <= b.upperContentLimit(); i++)
        answer += b[i];

    return answer;
}

```

7.3 Dynamic array resizing

This example illustrates the `upperBound` member function, which dynamically changes the size of an array. It also illustrates use of the `upperContentLimit` member function prior to a traversal—a common paradigm.

```
#include <iostream.h>
#include "SmartArray.h"

bool concat(SmartArray<int>& a, SmartArray<int>& b, SmartArray<int>& c);

void main() {
    // The lower bound for a, b, c is 1.
    SmartArray<int> a(1,5);
    SmartArray<int> b(1,10);
    SmartArray<int> c(1,1);

    // fill the a and b arrays
    int i;
    for (i = 1, i <= 5; i++)
        a[i] = i;
    for (i = 1, i <= 10; i++)
        b[i] = i;

    concat(a, b, c);

    cout << c[6];
}
```

For illustration, the function `concat` is written without using the editing functions.

```
bool concat(SmartArray<int>& a, SmartArray<int>& b, SmartArray<int>& c) {

    // concat is restricted to arrays having a lower bound of 1, this
    // should be checked for a, b, and c or changed with a call to
    // reindex.

    // set the upper bound of c
    c.upperBound(a.content() + b.content());

    // set upper content limit in preparation for a traversal
    // that alters c
    c.upperContentLimit(c.upperBound());

    // traverse c
    c.start();
    // traverse a
    a.start();
    while (a.next()) {
        c.next();
        c.current() = a.current();
    }
    a.stop();

    // traverse b
    b.start();
```

```

        while (b.next()) {
            c.next();
            c.current() = b.current();
        }
        b.stop();
        c.stop();
        return true; // no error checking in this version
    }
}

```

The printed output is:

1

Of course, main would normally be written as follows using `append` to replace `concat`.

```

void main() {
    // The lower bound for a, b, c is 1.
    SmartArray<int> a(1,5);
    SmartArray<int> b(1,10);
    SmartArray<int> c(1,1);

    // fill the a and b arrays
    int i;
    for (i = 1, i <= 5; i++)
        a[i] = i;
    for (i = 1, i <= 10; i++)
        b[i] = i;

    c = a;
    c.append(b);
    cout << c[6];
}

```

The printed output is again:

1

7.4 Use of `upperBoundNeeded` with dynamic array resizing

This example illustrates how the size of an array can be expanded as necessary. Using the `upperBoundNeeded` member function described in Section 4.7, the `while` loop reads integers until `cin` is exhausted.

```

#include <iostream.h>
#include "SmartArray.h"

void main() {
    SmartArray<int> a(1,1);

    int num;
    int index = a.lowerBound();           // better than index = 1;
    while (cin >> num) {                  // Ref. 3 p. 331
        a.upperBoundNeeded(index);
        a[index++] = num;
    }
}

```

If 21 integers were supplied, the upper bound of the array would be 32 at the end of the program.

7.5 Memory allocation error handling

This example illustrates how the function `concat` from Section 7.3 would be rewritten to handle a memory allocation resource error.

```
bool concat(SmartArray<int>& a, SmartArray<int>& b, SmartArray<int>& c) {

    // concat is restricted to arrays having a lower bound of 1, this
    // should be checked for a, b, and c or changed with a call to
    // reindex.

    // set the upper bound of c and check for memory allocation
    // resource error
    c.upperBound(a.content() + b.content());
    if (c.errorCheck())
        return false;

    // the rest of the function is the same as in Example 3
    ...
}
```

7.6 Array bound logic error handling

This example illustrates the handling of an array bound logic error.

```
#include <iostream.h>
#include <stdlib.h>
#include "SmartArray.h"

void main() {
    SmartArray<int> a(1,10);

    for (int i = 1; i <= 12; i++)
        a[i] = i;

    if (a.errorCheck()) {
        cerr << "array error" << endl;
        exit(0);
    }
}
```

If `assert` statements are enabled for the `SmartArray` class, then this program will assert and terminate when the instruction `a[11] = 11` is encountered.

If `assert` statements are disabled for the `SmartArray` class, then the instructions `a[11] = 11` and `a[12] = 12` will each cause an array bounds logic error to occur. As explained in Section 3.5, these instructions will cause the integer values 11 and 12 to be successively written to an auxiliary object, since the array locations `a[11]` and `a[12]` do not exist. Any subsequent use of `errorCheck` will return true.

7.7 Editing functions

This example illustrates the copy constructor, as well as the `insert` and `remove` editing functions.

```
#include <iostream.h>
#include "SmartArray.h"

void main() {
    SmartArray<int> a(1,10);

    for (int i = 1; i <= 10; i++)
        a[i] = i*i;

    SmartArray<int> b=a;

    // at this point a and b are equal

    b.remove(3,4);

    // at this point a and b are different

    b.insert(16,3);
    b.insert(9,3);    // when 9 is inserted at b[3], 16 moves to b[4].

    // at this point a and b are equal
}
```

7.8 SmartArray of a user-defined class

This example illustrates the definition of a class for use with the `SmartArray` class template.

```
#include <iostream.h>
#include "SmartArray.h"

class pair {
public:
    pair() : first(0), second(0) {};
    pair(int m, int n) : first(m), second(n) {};
    friend pair operator+ (const pair& a, const pair& b);
    // the compiler-generated destructor and operator= are adequate.
public:
    int first;
    int second;
}

pair operator+ (const pair& a, const pair& b) {
    return pair(a.first + b.first, a.second + b.second);
}

void main() {
    SmartArray<pair> a(1,10);
```

```

        a.put(pair(1,-1));

        SmartArray<pair> b=a;

        pair sum;
        a.start();
        while (a.next())
            sum = sum + a.current();
        a.stop();

        cout << sum.first << " " << sum.second << endl;

    }

```

This program will print:

```
10 -10
```

7.9 Use of the external pointer constructor

This example illustrates the use of the external pointer constructor.

```

#include "SmartArray.h"

void main() {
    int builtIn[10];

    // put data into builtIn
    builtIn[2] = 2;
    builtIn[3] = 3;
    builtIn[4] = 4;
    builtIn[5] = 5;

    // Construct a SmartArray b with a lower bound of -1, and upper bound
    // of 1 and using the data of builtIn[2] through builtIn[4]. Therefore
    // b[-1] and builtIn[2] occupy the same memory.
    SmartArray<int> b(-1, 1, &builtIn[2]);

    // verify that data is stored "externally"
    assert(b.isDataExternal());

    // verify that builtIn's data is used by b
    assert(b.content() == 3);
    assert(b[-1] == 2);
    assert(b[0] == 3);
    assert(b[1] == 4);

    // modify b[-1] and verify builtIn[2] is changed
    b[-1] = 20;
    assert(b[-1] == 20);
    assert(builtIn[2] == 20);

    // verify that b[-1] and builtIn[2] occupy the same memory
    assert(&b[-1] == &builtIn[2]);

    // use put member function to modify b[0] and b[1]

```

```

        b.put(30,0);
        b.put(40,1);

    // reindex b to a lower bound of zero and verify data is properly reindexed
        b.reindex(0 - b.lowerBound());
        assert(b[0] == 20);
        assert(b[1] == 30);
        assert(b[2] == 40);

    // verify builtIn is accessing the data correctly
        assert(builtIn[2] == 20);
        assert(builtIn[3] == 30);
        assert(builtIn[4] == 40);
        assert(builtIn[5] == 5);
    }

```

8 Enhancing the SmartArray class

Some convenient features (such as equality, print, and sort) have been intentionally omitted from the SmartArray class to avoid additional requirements on type T (for example, support for `operator==`). This functionality and more can be obtained by deriving a class from SmartArray. Classes derived from SmartArray should augment—not modify—the behavior of the base class.

The proper implementation of a typical derived class in C++ involves subtle interactions between inheritance, assignment operators, references, and pointers. Section 8.1 summarizes these issues in a general context.

Section 8.2 describes the protected member functions provided by SmartArray for use in implementing derived classes.

8.1 Inheritance and assignment operators

Compiler-generated assignment operators are badly broken—they do not work properly with references and pointers to classes in an inheritance hierarchy. The proper implementation of assignment requires virtual functions that make use of run-time type information (RTTI). Our scheme for implementing assignment operators, presented in *Correct Assignment Operators in C++* [Reference 17], follows this approach. In addition our scheme requires no changes be made to the base class or existing derived classes as new derived classes are added to the inheritance hierarchy. This scheme is used to implement the assignment operator in SmartArray.

Assignment implementation issues are of course not peculiar to SmartArray. They arise whenever a class has a public assignment operator and serves as a base class—a common occurrence in C++ programming. Furthermore, assignment is only the most prominent example of functions where these issues arise. Another example is the equality operator (`operator==`).

The problems with compiler-generated assignment

Because compiler-generated assignment operators are based on static type information, they do not work properly with references and pointers to classes in an inheritance hierarchy.

An example of this is the so-called “slicing” problem. Consider the following:

```
Derived d1, d2;
Base &r1 = d1;
Base &r2 = d2;
```

the assignment

```
r1 = r2;
```

will result in `operator=` for the base class being called—with the result that slicing or *partial assignment* occurs, *i.e.*, any derived class data is not copied. This slicing occurs even though `r1` and `r2` refer to objects of class `Derived`.

Many authors discuss the implementation of assignment in the context of actual object variables while failing to mention that these implementations often behave incorrectly (*i.e.*, slice) with references or dereferenced pointers. And yet objects are often passed by reference—thus setting the stage for the introduction of subtle bugs.

Correct assignment operators must know and use the dynamic (run-time) types of both the left-hand side *and* right-hand side of an assignment. The dynamic type of the left-hand side can be determined indirectly with the virtual function mechanism. The dynamic type of the right-hand side can be determined by using or emulating a new feature of C++ called run-time type information (RTTI). Versions of `SmartArray` using both built-in and emulated RTTI are available.

Some writers [References 14, 15, 18, 19] have placed a high emphasis on compile-time detection of improper assignment. However, in the presence of references or pointers, *compile-time detection is not possible* since the types of the referents are not known at compile time.

As these writers point out, compile-time detection of improper assignment is only achievable when inheritance is restricted to the point that the types of the referents *are* known at compile time. This knowledge can only be achieved at high cost—by prohibiting derivation from concrete classes. While this restrictive approach may make sense in some instances, in general it is throwing out the baby with the bath water. In particular, it disallows polymorphic assignment statements and it leads to an unnecessarily complicated class hierarchy. Yet, polymorphic behavior and inheritance hierarchies that clearly reflect design intent are central goals of C++. Wide scale complication of natural class hierarchies to circumvent language deficiencies (the compiler-generated assignment operator) cannot be the right approach.

The scheme proposed by us in Reference 17 allows *both* inheritance from concrete classes *and* correct assignment operators while requiring no changes be made to the base class or existing derived classes as new derived classes are added. The `SmartArray` class uses this scheme to implement correct assignment operators. The files `SmartArrayPlus.cc` and `SmartArrayPlus.h`, which accompany the `SmartArray` source code, demonstrate the implementation of this scheme in a derived class. These two files should be used as the starting point for array classes derived from `SmartArray`.

8.2 Protected member functions

Several protected member functions are documented here because they are potentially useful to derived classes.

virtual void assign(const SmartArray<T>& rhs)

Used in the assignment operator scheme documented in Reference 17.

bool audit() const

Performs a consistency check on the state of an array. Since there is some overhead cost for this, the consistency check is done only if `assert` statements are enabled; otherwise, `audit` always returns `true`. If any problems are found, `audit` will halt execution of the application program with an `assert` statement.

bool checkRange(int index, int low, int hi) const

Returns `true` if `low <= index && index <= hi`, otherwise returns `false`.

void errorMessageSet(const char *text) const

Copies `text` to the error message. This text is retrieved via the public member function `errorMessage`.

The error message is treated as `mutable` and therefore can be changed for a `const` array.

void errorSet() const

Sets the error flag. The status of this flag is checked via the public function `errorCheck`. The error flag is cleared with the public `errorClear` member function.

The error flag is treated as `mutable` and therefore can be changed for a `const` array.

9 References

- [1] Meyers, S., *Effective C++*, Addison-Wesley, 1992.
- [2] Maguire, S., *Writing Solid Code*, Microsoft Press, 1993.
- [3] Stroustrup, B., *The C++ Programming Language*, Second Edition, Addison-Wesley, 1993.
- [4] Ellis, M. and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- [5] Murray, R., *C++ Strategies and Tactics*, Addison-Wesley, 1993.
- [6] Cargill, T., *C++ Programming Style*, Addison-Wesley, 1992.
- [7] Carroll, M. and Ellis, M., *Error handling in C++ library code*, C++ Report, May, 1994.
- [8] Koenig, A., *When constructors fail*, C++ Report, May, 1994.
- [9] Cargill, T., *Exception handling: a false sense of security*, C++ Report, November-December, 1994.
- [10] Plauger, P., *The Draft Standard C++ Library*, Prentice Hall PTR, 1995.
- [11] van der Linden, P., *Expert C Programming*, Prentice Hall, 1994.
- [12] Cline, M. and Lomow G., *C++ FAQs: Frequently Asked Questions*, Addison-Wesley, 1995.
- [13] Taligent Press, *Taligent's Guide to Designing Programs*, Addison-Wesley, 1994.
- [14] Meyers, S., *Code reuse, concrete classes, and inheritance*, C++ Report, July-August, 1994.
- [15] Meyers, S., *operator=: The readers fight back*, C++ Report, November-December, 1994.
- [16] Stroustrup, B., *The Design and Evolution of C++*, Addison-Wesley, 1994.
- [17] Perano, K. and Nielan, P., *Correct assignment operators in C++*, SAND96-8220 (to be published).
- [18] Meyers, S., *More Effective C++*, Addison-Wesley, 1996.
- [19] Carroll, M. and Ellis, M., *Designing and Coding Reusable C++*, Addison-Wesley, 1996.

UNLIMITED RELEASE

INITIAL DISTRIBUTION

Mark Braaten
 Building K1 ES207
 General Electric Research Center
 1 River Road, P. O. Box 8
 Schenectady, New York 12301

Joe Fridy (10)
 Aluminum Company of America
 Alcoa Technical Center
 100 Technical Drive
 Alcoa Center, Pennsylvania 15069-0001

Victor Kaliakin
 Department of Civil Engineering
 University of Delaware
 Newark, Delaware 19716

Steve Maguire
 Storm Development
 916 NE 65th St., #172
 Seattle, WA 98115

John Micco
 Symantec Corp.
 135 South Street
 Bedford, MA 01730-2307

Kyran Mish
 2 Laguna Ct.
 Chico, CA 95928

Robert Neely
 Mailstop L035
 Lawrence Livermore National Laboratory
 Livermore, CA 94550

Ivan Otero
 Mailstop L035
 Lawrence Livermore National Laboratory
 Livermore, CA 94550

T. K. Parnell
 Failure Analysis Associates
 190 Commonwealth Dr.
 Menlo Park, California 94205

Christopher Skelly
 Insight Resource Inc.
 130 New Broadway
 North Tarrytown, NY 10591

0437 S. W. Attaway, 9118
 0439 D. R. Martinez, 9234
 0441 P. L. Stanton, 9225
 0443 G. D. Sjaardema, 9117
 0458 J. S. Rottler, 5139
 0819 J. M. McGlaun, 9231

0819 J. S. Peery, 9231
 0826 J. A. Schutt, 9111
 0833 J. H. Biffle, 9103
 1109 A. L. Hale, 9224
 1110 R. C. Allen, 9222
 1110 W. J. Camp, 9200
 1111 S. S. Dosanjh, 9221
 1111 J. N. Shadid, 9226
 1320 K. G. Budge, 6748
 9001 T. O. Hunter, 8000
 Attn: L. A. West, 8200
 W. J. McLean, 8300
 L. A. Hiles, 8800

9003 D. L. Crawford, 8900
 9004 M. E. John, 8100
 9007 R. C. Wayne, 8400
 9011 R. C. Armstrong, 8920
 9011 E. L. Bloesch, 8920
 9011 R. L. Clay, 8920
 9011 J. E. Costa, 8920
 9011 H. D. Jones, 8920
 9011 C. M. Pancerella, 8920
 9011 K. J. Perano, 8920 (25)
 9011 T. W. Strayer, 8920
 9011 R. A. Whiteside, 8920
 9042 R. J. Kee, 8303
 9042 P. E. Nielan, 8742 (25)
 9105 C. A. Lajeunesse, 8412
 9214 E. J. Friedman-Hill, 8117
 9214 C. L. Janssen, 8117
 9214 W. E. Mason, 8117
 9214 J. C. Meza, 8117
 9214 L. M. Napolitano, 8117
 9405 M. L. Callabresi, 8743
 9405 T. M. Dyer, 8700
 9405 B. V. Hess, 9220
 9671 J. R. Smith, 8220
 9021 Technical Communications Department, 8815 for
 OSTI (10)
 9021 Technical Communications Department, 8815 /
 Technical Library, 4414
 8099 Technical Library, 4414 (4)
 9018 Central Technical Files, 8950-2 (3)