THE “LIGHTWEIGHT CLASSES” design technique, which directly integrates C++ code with existing C language database libraries, helps reduce the need for expensive and complex object database systems. In addition, it eliminates the tedious and error-prone maintenance of “object serialization” code that is often associated with C++ database programming.

“Lightweight classes” occupy contiguous ranges of memory, just like the C and C++ fundamental types int, double, and char. They may not contain virtual functions or pointer data members. At first glance, these restrictions may seem oppressive and contrary to the principles of object-oriented programming (OOP). This article demonstrates, however, that through consistent member function nomenclature, judicious use of templates, and well planned type conversion operators, programmers can enjoy the elegance and type safety of C++ OOP with no loss in efficiency relative to C and C++ fundamental types.

With refinement and standardization, “lightweight classes” could well be the “grand unifying force” of the C versus C++ debate; the key that permanently frees us from the unsafe world of C data types and opens the door to an equally efficient, yet far more elegant, new C++ paradigm. While simplifying database integration was my primary motivation for developing “lightweight classes,” I have found that their optimal time and space efficiency makes them also useful as key elements throughout my development work.

In this article, I will review the challenges of C++/database integration and consider some of the solutions that have been developed to manage these problems. “lightweight classes” will be introduced as an alternative solution, founded in the low level structure of C++ classes. The remainder of the paper demonstrates the use of C++ language features that impart object-oriented elegance and functionality to these otherwise low-level, seemingly rigid constructs. In conclusion, I argue that “lightweight classes” might well serve as alternatives to legacy C fundamental types in a future C++ language standard.

FILE I/O USING C FUNDAMENTAL TYPES Traditionally, Disk record I/O in C has been accomplished by a sequence of function calls. At a low level, the ANSI library calls (such as fopen() and fwrite()) perform these functions. For instance, consider a database record, represented by a struct of aggregated C fundamental types:

```c
struct customer_struct {
    char acctno[8];
    char name[40];
    char address[40];
    ...}
```
LIGHTWEIGHT CLASSES

To write an instance of this struct to disk using ANSI C library functions (without error checking), one can write:

{  
  struct customer_struct customer;  
  ... Code to initialize customer struct elements omitted for clarity  
  FILE* f = fopen("custfile.dat","wb");  
  fwrite(&customer,sizeof(customer),1,f);  
  fclose(f);  
}

In my application work, I employ the Faircom C-tree database system, a C library provided in source code, to manage record structures like those above. C-tree offers automatic file indexing, ad-hoc reporting, SQL and ODBC support, transaction logging, platform and CPU independence, and other functionality that enables our firm to deliver robust applications. While the ideas I will present in this article were the result of my desire to integrate C++ code with Faircom’s C-tree system, I will use more universally familiar ANSI C function calls as examples. I mention my use of C-tree to underscore the fact that “lightweight classes” can function transparently in a complex database environment. They are by no means limited in scope to the ANSI C I/O library calls.

THE C++ DATABASE PROBLEM  When I began to study C++ several years ago, I had hoped to replace struct declarations like the one above with an elegant hierarchy of classes modeled after Coplien’s “Orthodox Canonical Class Form.” An abstract “field” base class formed the root of this hierarchy. Classes for strings, integral values, dates, and money types would be derived from “field.”

Writing a beautiful hierarchy was not too difficult. Integrating it with C-tree for record I/O proved challenging. For instance, consider the following naive rewrite of the above struct and disk write using the familiar ANSI C++ string class:

```
class customer_class {  
public:  
  string acctno;  
  string name;  
  string address;  
};

{  
  customer_class customer;  
  ... Code to initialize customer class members omitted for clarity  
  FILE* f = fopen("custfile.dat","wb");  
  fwrite(&customer,sizeof(customer),1,f);  
  fclose(f);  
}
```

Of course, the fwrite() call will not save any customer information to disk. With the Borland 4.52 C++ compiler, which includes an ANSI modeled C++ string class implementation, the above fwrite() call (in large model) records 12 bytes of “junk.” This is because the sole data member in Borland’s string class declaration is a (4 byte) pointer to a buffer where the actual data is stored.

One work-around that I explored early on was to add member functions to implement persistence for each of my field classes:

```
virtual void* prepare_buffer_for_write();  
virtual void parse_after_read(const void*);  
```

These functions would take on the responsibility of correctly translating data elements between their complex in-memory formats and the serialized format required for Disk I/O.

There are several problems with this solution:

1. Runtime overhead—For each data member in a class, a translation function must be called on every read and write. For many class designs, pointers must be followed and additional translations performed. While one could argue that the performance hit from this CPU-intensive activity pales in comparison to the traditional notion of disk transfer time, contemporary and emerging disk cache and network architectures give cause for its acknowledgment.

2. Tedious programmer maintenance—as new data members are added to a class, the prepare_buffer_for_write() and parse_after_read() functions must be modified to handle the new class members. In a large application with many record types, hand coding these functions is a tiresome and error-prone task.

Automated preprocessing solutions such as PGen promise some relief from this tedium. However, to synthesize correct translation functions, persistence automation tools must analyze C++ source code keywords prior to their final compilation. This adds build-management complexity and buildtime overhead to the development process, which I would prefer to avoid. Furthermore, these tools focus on a small set of C++ keywords, with which they associate a limited set of translation semantics. In the case of PGen, the resulting restrictions include “The user must mark instances that are to be persistent . . . Objects must be marked for writing and for reading in the same order . . . Char* is always treated as a character string . . . Dynamic arrays are supported, but the user must use the programming convention . . ., etc.” While these restrictions are understandable, and non-trivial system can be designed subject to them, they show that automated preprocessing solutions are no magic bullet.

Object Database Systems claim to be utopian solution to this issue. In fact, the Object Database Management Group’s, “The Object Database Standard: ODMG-93” employs a traditional record-based application (a professor/student/course/section database) in many of their code examples.

In 1992 Object Design’s Tom Atwood wrote, “Initially, OODBMS technology will not compete with record-oriented DBMSs. OODBMS products will meet the needs of applications not served by relational technology and thus expand the entire DBMS world.” Indeed, current sales literature from Object De-
sign, Inc., lists as representative applications: “Documents, spreadsheets, financial data, images, CASE, symbolic data, data dictionaries, geometric models, device diagrams, financial modeling and simulation, DNA sequences, mapping/GIS, petrochemical drilling data.”

When examined from the standpoint of strategic business decision making, ODBs represent a very new, very small, albeit fast-growing, niche in the database software industry. For my development work I require only reads and writes of record-oriented data that is easily managed through the relational model. My firm cannot presently justify a migration to expensive, hardware intensive, ODBMS technologies that offer no direct value to our cost-conscious market segment.

THE LIGHTWEIGHT CLASSES ALTERNATIVE My solution, “lightweight classes,” was initially inspired by a lecture presentation given by Jonathan Shopiro at the Software Development Fall ‘92 conference in Boston. In his conference paper, Shopiro wrote, “The simplest classes, from a storage management point of view, are the small fixed-size classes. . . . The first thing to consider in planning the storage management for such a class is whether the default copy, assignment, and destruction operations are acceptable. If they are, letting the compiler synthesize the copy constructor, assignment function, and destructor will result in the most efficient code.”

Shopiro’s reasoning was both enlightening and startling, as it flew directly against everything I had thought or read about “canonical, good C++ programming style.”

In his lecture, he explained that, for small classes (classes whose footprint would fit in a machine register), one should neither declare a copy constructor nor override the default assignment operator. Shopiro showed that, if a small class lacks a handwritten default copy constructor and assignment operator, the C++ compiler (Cfront in his case) is freed to perform an interesting optimization. It can generate code that references the class through, ultimately, simple register loads and saves. Consider the comments in this example:

```cpp
struct small_class {
    short A, B;
    A=B;       // resolves to a register load and store
    anyfunc(A); // A is loaded into a register and pushed onto the stack.
};
```

More relevant to my problem was Shopiro’s reminder that such classes can be aggregated into larger classes that possess this same efficiency. Thus, I could write:

```cpp
class customer_class {
    public:
        class_lacking_handwritten_copy_constructor name;
        class_lacking_handwritten_copy_constructor address;
};
```

and have confidence that this assignment operation:

```cpp
{ 
    customer_class customerA, customerB;
    customerA=customerB;
}
```

would resolve to a single copy operation.

I have verified this by reviewing the assembly language output of the Borland compiler (-S option). When I aggregate field classes possessing a handwritten replacement of the default assignment operators into a larger record class, the compiler resolves the assignment RecordA=RecordB by appending a synthesized static scope function to the end of each module that makes such an assignment. This compiler synthesized static function calls, for each field class in the record class, the field’s handwritten assignment operator function.

On the other hand, when I let the compiler simply synthesize the default assignment operator for the field classes, the resulting assembly code for RecordA=RecordB is a single call to a class-independent memcpy()-like function. No additional code is synthesized by the compiler.

Thus, when it is practical, to avoid code bloat and preserve low-level object copying efficiencies, one should abstain from the “canonical, good C++” practice of handwriting copy constructors and assignment operators.

With a further restriction of class design, such that pointer members and virtual functions are disallowed, it follows that complete instance data for the class can be directly written to disk using the ANSI C fwrite()/fread() function calls.

The “Commentary” section of ARM, Chapter 10 and Stroustrup’s discussion of the “struct is-a-class notion” in The Design and Evolution of C++ reinforce this conclusion.

DEFINING LIGHTWEIGHT CLASSES I define a “Lightweight Class” to be any class design that satisfies the following stringent criteria:

- It has no virtual functions
- Either it has no ancestor classes, or its ancestor classes are all “lightweight classes”
- All its data members are either C fundamental types (char, short, int, etc.) or “lightweight classes”
- It has no members that are pointers
- It either has an empty handwritten default constructor {}, or no constructors whatsoever. (The latter restriction must be observed if you wish to use “lightweight classes” in unions)
- It has no destructor
- It has no constructs that inhibit the compiler’s generation of a fast, bitwise, copy constructor. (Examples of forbidden constructs include handwritten copy constructors and constructors that take a non-const argument. See the ARM for a complete enumeration of these constructs.)
- It has no constructs that inhibit the compiler’s generation of a fast, bitwise, assignment operator. (This restriction is analogous to the above rule for the copy constructor.)

While at first glance, these restrictions may appear to be a wholesale abandonment of OOP, this need not be the case. The remainder of this article shows how I have been able to incorporate...
LIGHTWEIGHT CLASSES

polymorphism, inheritance, message-passing, and other OOP features to compose not-so-simple “lightweight classes.” The key to success requires consistent class design and the careful use of templates to represent strings of various sizes.

CANONICAL LIGHTWEIGHT CLASS  The following example, “class minimal_class,” is a conceptual skeleton for all my “lightweight classes”:

```cpp
class minimal_class {
    public:
        enum {strbuflen=max_length_required_for_ASCII_representation};   // Output ASCII representation
    int to(const char *s) const;   // Create from ASCII representation
    void from(const char *s);     // Output ASCII representation to an ostream
    friend ostream& operator <<(ostream &os, const minimal_object &m);

    private:
        // Typically a short, long, char array, etc.
        some_C_fundamental_type value;
};
```

With this minimal class, one can write the following useful code fragments:

```cpp
{  // 1
    // Allocate a new instance of minimal_class on the stack
    // no initialization of minimal_object is performed.
    minimal_class minimal_object;  // 2

    // Set the contents of minimal_object to “1234”
    minimal_object.from(“1234”);  // 3

    // Write ASCII to standard output
    cout << minimal_object;  // 4

    // Write an ASCII representation to a character buffer
    buf[minimal_class::strbuflen];
    minimal_object.to(buf);   // 5

    minimal_class another_one;
    another_one = minimal_object;  // 6

    // Write minimal_object to a disk file
    fwrite(&minimal_object,sizeof(minimal_object),1,FILEptr);  // 7
}
```

Since no constructors have been written, the compiler outputs code that simply decrements the stack pointer at line 2. No other time-consuming or heap-greedy construction/destruction activities are performed. This truth has proven to be a great comfort in coding and debugging!

In line 3, a “from” function is called. In many cases, an assignment operator will make for a more readable syntax. However, when the right-hand side of the assignment is inherently dissimilar (“1234” is a const char* in this case), the explicit use of the from() conversion function syntax better documents the complexity of the “assignment” operation.

In line 4, we write minimal_class to an ostream.

In line 5, minimal_class is formatted into a character buffer using C’s zero byte string termination convention. A strong argument could be made that this functionality is redundant with line 4, and not typesafe. However, in the “lightweight classes” world, buf[minimal_class::strbuflen] is guaranteed to be large enough to contain the output. Furthermore, I have found that there are many cases when it is useful to perform high-speed conversion to ASCII, without enduring the runtime overhead of the stream libraries.

Since we have not supplied an assignment operation, in line 6 the compiler-generated code performs a fast, bitwise copy of another_one to minimal_object.

In line 7, fwrite() is able to write minimal_object to disk storage because the footprint of minimal_class is a contiguous region of memory with length of sizeof(some_C_fundamental_type).

At line 8, no destruction activities are performed. The stack pointer is simply incremented back to its value from the start of the block.

As I will show in later examples, these efficiencies are preserved as “lightweight classes” are aggregated into more complex record structures.

CONCEPTUAL INHERITANCE  I like to think of the relationship between minimal_class and my real “lightweight classes” as one of “conceptual inheritance.” All my “lightweight classes” have, at minimum, the functionality of minimal_class. The IS-A relationship is unmistakable. However, it cannot be documented through the usual C++ inheritance syntax for two illuminating reasons:

• “lightweight classes” may not contain virtual functions
• An idealized root class in a hierarchy of “lightweight classes” would contain no data members

Data members would be specified in the descendant classes. Unfortunately for “lightweight classes,” the ARM dictates that objects of an empty class must have non-zero size.

I have designed lightweight classes that represent all the numerical types in the C-tree database library. These include: 2-byte integer, 4-byte integer, 2-byte unsigned, and 4-byte unsigned. These are easy to craft using a singular class data member called “value” that is one of the respective C built-in types: short, long, unsigned short, and unsigned long.

Many of the operations (++, --, <=, >, etc.) on “value,” are identical across all my numerical class definitions. To avoid large chunks of repetitious code, I use macros to fill in many repeated elements in my “lightweight class” definitions. While macros are unpleasant to look at and even less pleasant to debug, the maintenance and debugging of one small, ugly piece of code is an easier job than the maintenance and debugging of a half dozen elegant replications of that code.

Following is an example of this approach, a macro that provides relational operators to all of my numerical “lightweight classes” that possess a singular data member called value:

```cpp
#define __relational_macro(classname) \
    int operator < (classname second) const \ 
    { return value < second.value; } \ 
    int operator > (classname second) const \ 
    { return value > second.value; } \ 
    int operator <= (classname second) const \ 
    { return value <= second.value; } \ 
    int operator >= (classname second) const \ 
    { return value >= second.value; } \ 
    int operator == (classname second) const \ 
    { return value == second.value; } \ 
    int operator != (classname second) const \ 
    { return value != second.value; }
```

In line 4, we write minimal_class to an ostream.
A "LIGHTWEIGHT" MONEY CLASS

A more robust example of a Lightweight Class is the C-tree database "money" type. Using C, this C-tree database type would be stored as a long integer—a four-byte quantity of pennies.

The "Lightweight" money class I have created uses a wide variety of C++ features to make it a transparent addition to the C++ language. It is as efficient as a long, and a far more elegant representation of money:

class money { public: // See minimal_class above for explanations on these first 4 functions
enum {strbuflen = 14};
int to(char *s) const;
void from(const char *s);
// Output money as d.pp format
friend ostream& operator <<(ostream &os, money m);
// It is sometimes convenient to initialize a money object from a
double as doubles can be written directly into C++ source code
money(double d) {value = d * 100;}
void from(double d) {value = d * 100;}
money& operator =(double d) {from(d); return *this;}
// Since we have declared a constructor above, the compiler will not
// generate a default constructor. We must declare it here.
money(void) {}
// It would be an error to allow a money class user to code:
// x.from(int);
// These 2 special names force the class user to explicitly
// describe what is happening
void from_pennies(long l) {value = l;}
void from_dollars(long l) {value = l * 100;}

// The overloading of all the arithmetic operators further cements the
// money type into the programming language.
money& operator =(money m) {value += m.value; return *this;}
money& operator =(long l) {value = l; return *this;}
money& operator =(int i) {value = i; return *this;}
money& operator =(money &m) {value += m.value; return *this;}

money operator -(void) const
{money temp; temp.value = -value; return temp;}

friend money operator - (money first,money second)
{money temp; temp.value = (first.value-second.value); return temp;}

friend money operator * (long_quantity,money_price)
{money temp; temp.value = (price.value * _quantity); return temp;}

friend money operator / (money_price,long_quantity)
{money temp; temp.value = (price.value / _quantity); return temp;}

// In a numerical class declaration, it is customary to define an
// "operator int()" that at least returns the zero or non-zero status
// of an object. However, this opens the door to other class users
// comparing money instances to other integers - a meaningless or
// at least ambiguous programming error. I employ the pennies()
// member function to force class users to write clearer code.

long pennies(void) const {return value;}

// This lack of operator int() mandates the following
// functions for code readability
bool iszero(void) const {return value == 0l;}
bool int isnonzero(void) const {return value != 0l;}
bool isnegative(void) const {return value < 0l;}
bool ispositive(void) const {return value > 0l;}

// Fill in the relational operators using macro discussed previously
__relational_macro(money)

private:
// The private declarations of the following prevent any type
// conversions of ints or longs to double, that would erroneously
// satisfy the assignment to double operator declared above.

money& operator =(long l); // makes a money from a long
money& operator =(int i); // makes a money from an int
from(long l); // copies a long to money
from(int i); // copies an int to money

// The singular data item in the money class declaration
long value;
};

A "LIGHTWEIGHT" STRING CLASS

Another type in the C-tree database system is a fixed length, zero terminated, C-style string. The C++ template mechanism can be used to implement this type easily:

template<int bufsize> class cfs { // Member functions shown and explained later
…
protected:
// The singular data item for this "Lightweight" string class.
char buf[bufsize];
};

C++ views each template class instantiation as a unique, independent type. You can write:
cfs<10> string10A;
cfs<10> string10B;
string10B.from("Hello");

and get a fast bitwise copy of stringB to stringA. However, without help, the compiler cannot resolve the following code:
cfs<8> string8;

string8 = string10A; // error - compiler does not know how to
// assign one cfs<> type to another

This code can be compiled, however, if the C++ compiler can transform string10A, via one unambiguous type conversion, into a type that one of stringB's assignment operator functions can digest.
The small and fast `cbuf_handle` class is this type in my “Lightweight Class” library.

```c
// cbuf_handle: A class that encapsulates a fixed length char buffer
// I elect to not include terminating zero bytes in the buffer
class cbuf_handle {
  const char *buf;
  int buflen;
public:
  // Inline constructor quickly assigns arguments to members
  cbuf_handle(const char* _buf, int _buflen)
    : buf(_buf), buflen(_buflen) {}

  int len(void) const {return buflen;}
  const char* charptr(void) const {return buf;}
};
```

As mentioned in the previous code, `cbuf_handle` is returned via a type conversion operator in the `cfs<>` class. This member of `cfs<>` is the inline function:

```c
operator const cbuf_handle() const {
  cbuf_handle cbh(buf,bufsize-1); return cbh;
}
```

The ability to initialize a `cfs<>` object is handled by the new class member:

```c
cfs<bufsize>& operator =(const cbuf_handle cbh);
```

**Aside:** This handwritten assignment operator takes a `const` argument. Thus, it does not impede the compiler’s output of a fast, bitwise assignment when `cfs<>` instances of the same instantiation signature (ex. `string10A = string10B`) are involved.

Now, the code fragment:

```c
string8 = string10A;
```

can compile and run as expected. The generated code will function like:

```c
    cbuf_handle temp = string10A.operator const cbuf_handle();
    stringA.operator = (temp);
```

**MAXIMIZING TEMPLATE PORTABILITY AND MINIMIZING COMILATION TIME**

Unfortunately, there are wide differences in template instantiation schemes between C++ compiler implementations. These differences have been discussed in many *C++ Report* articles.

My direct experience with Borland’s C++ compiler for DOS/Windows and SCO’s Cfront implementation for UNIX has led me to the following sanity saving recommendation:

To retain reasonable compilation times and portability as you employ “lightweight classes”, declare all template functions as small inlines that call supporting functions outside the template class declaration.

To this end, I have written all the `cfs<>` class functions as inlines that make fast calls to static functions in a special “`cfs_support`” class. By declaring the `cfs_support` class member functions static, they work precisely as ordinary C functions, but they are accessible only through the intentionally awkward “`cfs_support`” class name:

```c
class cfs_support {
  public:
    static int to(char *s, const char* cfs_buf, int cfs_bufsize);
    static void from(char* cfs_buf, int cfs_bufsize, const char *s, int slen);
    static ostream& to_ostream(ostream& os, const char* cfs_buf, int cfs_bufsize);
};
```

Given the above explanations of the `cbuf_handle` class, and `cfs_support` static functions, the complete `cfs` template class can finally be presented:

```c
template<int bufsize> class cfs {
  public:
    enum {strbuflen = bufsize+1};
    int to(char *s) const {return cfs_support::to(s,buf,bufsize);}
    void from(const char *s) {cfs_support::from(buf,bufsize,s,::strlen(s));}
    cfs<bufsize>& operator =(const cbuf_handle cbh) {cfs_support::from(buf,bufsize,cbh.charptr(),cbh.len()); return *this}
    cfs(const cbuf_handle cbh) {cfs_support::from(buf,bufsize,cbh.charptr(),cbh.len());}
    operator const cbuf_handle() const {cbuf_handle cbh(buf,bufsize-1); return cbh;}
  protected:
    char buf[bufsize];
};
```

It should be understood that the money and `cfs<>”lightweight classes” I have presented here are greatly reduced from my development versions. In actual practice, I have added member functions to interact with user interface data structures, and perform other application domain-specific tasks.

I should also note that inheritance can be used to represent more specific kinds of strings. For example, it is straightforward to derive from class `cfs<11>` a “phone number” class that adds new output formatting functions to write the data as (XXX) XXX-XXXX. In the context of “lightweight classes,” inheritance without virtual functions is surprisingly safe. I hope to discuss this in a future article.

**AGGREGATING “LIGHTWEIGHT CLASSES;” THE CUSTOMER_CLASS REVISITED**

Given a palette of “lightweight classes” from that to build more complex record structures and applications, we can now easily design a `customer_class` that can be directly written to disk:
class customer_class {
public:
    cfs<40> name;
    cfs<26> address;
    ...
    money current;
    money past_due;
    ...
    money balance(void) const {return current+past_due;};
};

The above customer_class can be written directly to disk using C
library calls:

```c
{ customer_class customer;
  FILE* f;
  customer.from("Joe Smith");
  customer.address.from("1234 Main St");
  customer.current.from(1234.99);
  customer.past_due.from_pennies(0);
  f= fopen("custfile.dat","wb");
  fwrite(&customer,sizeof(customer),1,f);
  fclose(f);
}
```

SUMMARY

- Powerful classes can be constructed without employing the
canonically “OOPish” features of C++: copy constructors, de-
sstructors, virtual functions, and pointers. These “lightweight
classes” can be written to disk with ease, and exhibit all the
efficiencies of C built-in types.

- A conceptual inheritance, implemented as a set of common fun-
damental member functions, possibly aided by macros, can be
used to leverage implementation and documentation across a
wide variety of “lightweight classes.”

- Through the use of small, fast, descriptive “handle” classes,
template based “Lightweight” classes with different instanta-
siation signatures can intercommunicate easily.

- The aggregation of “Lightweight Classes” into complex data-
base records results in a new “Lightweight Class” that can be
efficiently instantiated, copied, or written to disk storage.

THE FUTURE OF “LIGHTWEIGHT CLASSES” AND C++

I now use “light-
weight classes” as fundamental elements for all my development
work. I have come to view the C fundamental types (int, short, long,
etc.) much as I viewed machine registers in my early “C” days—
useful for writing high speed loops, but too unsafe and awkward
for general use.

It would be a welcome addition to the C++ language if a core
group of “lightweight classes” were adopted by the design com-
mittes. A natural nomenclature for these classes would be a capi-
tal ‘C’ followed by the legacy C type name as in: Cshort, Cint, etc.
Through the use of a robust, standardized skeletal class member
function interface, C++ programmers could write code that is as
efficient as C, without having to navigate all the inconsistencies in
the ASCII C library type conversion and formatting functions.

While C++ has been heavily criticized by object-oriented design
purists for its use of the legacy C fundamental types, it has been
embraced by legions of commercial programmers for its low-level
efficiency.

It is my hope that the needs of both camps can, in large part,
be satisfied by the concepts underlying “lightweight classes.”

ACKNOWLEDGEMENT Many thanks to Dr. Steve Schach of the Com-
puter Science Department at Vanderbilt University for his help and
encouragement in the development of this paper.

References

1. Atwood, T. “Applying the Object Paradigm to Databases,”
   Proceedings of the Software Development ’92 Developers’ Con-
2. Atwood, T., J. Duhl, G. Ferran, M. Loomis and D. Wade. The
   Object Database Standard: ODMG-93. Morgan Kaufman, San
4. Ellis, M. and B. Stroustrup. The Annotated C++ Reference
   pp. 4–8.
8. “Objectstore, The Critical Component for Object- Oriented
   Development,” and accompanying sales literature. Object
9. “Chapter 7, Template instantiation in C++”—overview taken
directly from a paper by Glen McCluskey and Robert B. Mur-
ray, SCO C++ Language System Selected Readings. The Santa
10. Shopiro, J. “Storage Management Techniques for Fixed Size
    Objects in C++,” Proceedings of the Software Development ’92
11. Stroustrup, B. The Design and Evolution of C++. Addison-
12. Tichy, W., J. Heilig and F. Newberry Paulisch, “A generative
    and generic approach to persistence,” C++ Report. Jan. 1994,
    pp. 22–33.
13. Tristram, C. “Do you really need… Object Databases,” Open

Chris Moth has been President of Daisy Systems, Inc., a vertical software firm
specializing in retail point of sale automation systems, since 1982. He received
a Master of Science Degree from Vanderbilt University in 1995. He can be
reached at cmoth@daisynet.com.