The Design of the E Programming Language

Joel E. Richardson
Michael J. Carey
Daniel T. Schuh

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

E is an extension of C++ designed for writing software systems to support persistent applications. Originally designed as a language for implementing database systems, E has evolved into a general persistent programming language. E was the first C++ extension to support transparent persistence, the first C++ implementation to support generic classes, and remains the only C++ extension to provide general-purpose iterators. In addition to its contributions to the C++ programming domain, work on E has made several contributions to the field of persistent languages in general, including several distinct implementations of persistence. This paper describes the main features of E and shows through examples how E addresses many of the problems that arise in building persistent systems.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-88-K-0303, by the National Science Foundation under grant IRI-8657323, by Fellowships from IBM and the University of Wisconsin, by DEC through its Incentives for Excellence program, and by grants from GTE Laboratories, Texas Instruments, and Apple Computer.

Authors' current addresses: Joel E. Richardson, IBM Almaden Research Center (K55/801), 650 Harry Road, San Jose, CA 95120. Michael J. Carey and Daniel T. Schuh, Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706.

1. MOTIVATION

In the mid-1980's, several database research groups, responding to the needs of a variety of emerging applications, began to explore "extensible database systems" [Bato88, Care85, Care90, Daya86, Rowe87, Schw86]. Although different groups have different notions of what "extensible" means, a common desire is to support a high degree of flexibility for customizing the database system to the user's application. Such flexibility is mostly lacking in today's commercial systems, and attempts to provide it through added software layers experienced severe performance penalties [Care85]. These experiences prompted researchers to explore system architectures which could be extended easily and without loss of performance. For example, in an extensible DBMS, it should be easy to augment the collection of "base" types with new user-defined types.[1]

The EXODUS project at the University of Wisconsin has been exploring a toolkit approach to extensibility. The software tools simplify the construction of customized database systems as well as the extension of these systems once they have been built. The first component of the EXODUS toolkit is the EXODUS Storage Manager [Care89]. It provides basic support for objects, files, and transactions. Next is the E programming language and its compiler. This paper describes the E language design; various approaches to the implementation of E are described in [Rich89b, Rich90, Schu90, Shek90]. The third major EXODUS component is the Optimizer Generator [Grae87], which allows a database implementor (DBI) to produce a customized query optimizer from a set of rules describing a given query algebra.

E was originally intended as the language in which to write database system code; that is, abstract data types (e.g., time), access methods (e.g., grid files), and operator methods (e.g., hash join) were all to be written in E. E was also intended as a target language for schema and query compilation, with user-defined schemas being translated into E types and user queries into E procedures [Care86b, Rich87].

The difficulty in building a DBMS in a conventional systems programming language (e.g., C) derives from several factors. First, the DBI must write code whose primary task is to manipulate shared data on secondary storage. A significant portion of the total system code is therefore devoted to interacting with the storage layer, e.g., calling the buffer manager to read a record. Second, the DBI must write code for operators and access methods without knowing a priori the data types on which they might operate. For example, the DBI cannot know that some user will eventually want to build an index, keyed on area, over a set of polygons. Finally, the DBI must build a query processor that converts queries posed by end users into a form that the system can execute. This translation is greatly simplified if the basic operators can be written in a composable manner.

[1] Commercial Ingres now supports user-defined abstract data types along the lines pioneered by ADT-INGRES [Ston86] and POSTGRES [Rowe87].

These and other influences led to the design of the E language. While the original goals are still clearly evident, E has evolved into a general purpose language for coding persistent applications. E is an extension of C++ [Stro86, Elli90] providing generator classes, iterators, and persistent objects. C++ provided a good starting point with its class-based data abstraction features and its expanding popularity as a systems programming language. Parameterized types in the form of generator (or generic) classes were added for their utility both in defining database container types, such as sets and indices, as well as in expressing generic operators, such as select and join. Iterators were added as a useful programming construct in general, and as a mechanism for structuring database queries in particular. Both generators and iterators were inspired by CLU [Lisk77]. Persistence — the ability of a language object to survive from one program run to the next — was added because it is an essential attribute of database objects. In addition, by describing the database or object base in terms of persistent variables, one may then manipulate it directly using expressions within the E language.

Two factors influenced the way in which E's extensions were added to C++. The first was the desire to maintain upward compatibility, that is, that E should be a strict superset of C++. Originally, we were forced to make a small compromise in order to support generator classes: in E, nested class definitions follow nested scope rules, while nested class definitions were exported to the global scope in versions 1.2 and 2.0 of C++ [Stro86]. Now that C++ also supports nesting of class scopes [Elli90], this is no longer a problem. The second important factor concerned efficiency. We desired that the C++ subset of E be compilable into code no less efficient than that produced by a C++ compiler. This primarily influenced the way in which we added persistence to the language: instead of allowing persistence to be a property of any object, we allow persistence only for objects whose type is declared to be a database (db) type. Section 5 explains this point more fully.

The design and implementation of E represent several contributions to the field of persistent programming languages. E and Avalon/C++ [Herl87, Detl88], designed at approximately the same time, were the first to extend C++ with persistence. In Avalon/C++, persistence is based on inheritance from the class recoverable, and access to persistent objects must be within the context of a special *pinning block*. Persistence in E is based on the persistent storage class, and the I/O required to access a persistent object is transparent. The storage class approach has since appeared in at least two more recent language designs, Persistent Modula-3 [Hosk90] and ObjectStore [Lamb91]. E was also the first extension to add generic classes to C++; its design and implementation predated the C++ template design [Stro88]. In addition, E remains the only C++ extension to provide a general-purpose iterator construct.

This remainder of this paper is organized as follows: Section 2 reviews the basics of C++ classes, introducing a simple binary tree example. The following sections then present the three main language extensions that E adds to C++. These sections also compare E's features to similar features found in other languages. Section 3 discusses iterators and uses them to help extend the binary tree to support duplicate keys. Section 4 describes generators and redefines the example as a generic binary tree. Section 5 describes database types and persistence and modifies the example to make the tree a part of a database. Following the presentation of the E language design in Sections 3-5, Section 6 discusses several shortcomings of the current design and/or implementation of E. Since E was first designed and implemented, other persistent extensions of C++ have appeared; several of these are discussed briefly in Section 7. Section 8 concludes the presentation by summarizing the current status of the language. An Appendix is included at the end of the paper to indicate how the parts database application described in [AtkM87] can be expressed in E. Finally, it should be noted that all of the examples in this paper have been compiled and run.

2. C++ REVIEW

The original design of E was an extension of C++ v.1.2 [Stro86], which extended C [Kern78] with classes, operator overloading, type-checked function calls, and several other features. C++ v.2.0 has added multiple inheritance, and E has recently been ported to this version. In order to avoid unnecessary complication, our review of C++ and our examples will include only the data abstraction features, which were present in version 1.2. Where necessary, we will highlight the interaction of C++ inheritance (especially, multiple inheritance) with features added by E, e.g., see Section 5.3.3.

2.1. Classes

A central concept in C++ is the notion of a *class*. A class defines a type, and its definition includes both the representation of any instance of the class as well as the operations that may be performed on an instance. Unlike the abstraction mechanisms provided in CLU [Lisk77] or Smalltalk [Gold83], a C++ class does not necessarily hide the representation of instances. It is up to the designer of a class to declare explicitly which members (data and function) are private and which are public. The data abstraction capabilities that C++ classes provide were one of the main motivations for our selection of C++ as a starting point for the design of E.

In C++ parlance, representation objects are called *data members*, and class operations are called *member functions* (or methods). Member functions are always applied to a specific instance; within the function, any unqualified reference to a data member of the class is bound to that instance. The binding is realized through an implicit parameter, this, which is a pointer to the object on which the method was invoked. Within a member function, an unqualified reference to a member x of the class is equivalent to this→x.

2.2. Inheritance

Another reason that we chose C++ as a starting point for E is that it supports subtyping. Given a class A, we may define a class B that is a subtype of A as follows:

```
class A { ... };
class B : public A { ... };
```

A is called the base class, and B, the derived class. B inherits both the representation of A as well as A's member functions. The **public** keyword in this context specifies that public members of A are also public members of B; without this keyword, public members of A would become private members of B. B may declare additional data members and member functions, and it may reimplement the member functions defined in A.

One of the key contributions of object-oriented programming languages is the late binding of method calls. That is, suppose a method f in type A is reimplemented in B, and suppose a program contains an invocation of f through an object pointer x whose (static) type is A. At runtime, x may actually refer to an instance of type B. Late binding defers the binding of code for f until runtime. At that point, the actual type (A or B) of the object is determined, and the appropriate implementation of f is called. Late binding allows a type hierarchy to be extended with new subtypes without requiring changes to (or even recompilation of) existing code. In C++, late binding is achieved by declaring member functions to be *virtual*.

Although we do not show specific examples in this paper, E extends C++ subtyping mechanisms, including both single- and multiple-inheritance, to the realm of database types and persistent objects. This task has presented several challenges, both in defining the semantics of types and type persistence and in adapting the implementation. Not all of these problems have been completely solved; we discuss the issues further in Section 6.

2.3. An Example

The example in Figures 1a and 1b is a complete C++ definition for a very simple binary tree index. The basic operation of the tree is to map a key value to the address of an entity having that key. In this example, each tree node stores a floating point key and a pointer to the indexed entity along with pointers to its left and right subtrees. The implementation uses a pair of classes: one which defines the nodes in the tree and one which defines the tree itself. The node class is recursive, both in its representation (i.e., nodes point to nodes) and in its operations (i.e., search and insert are recursive methods). The tree class is a simple "wrapper" that encapsulates the nodes. In order to keep the example simple while still showing the major features, the tree is unbalanced, and we limit the operations on the tree to inserting and searching.

Figure 1a gives the definition of the class binaryTreeNode. The representation of each node in the tree follows the class heading. Each node contains a floating point key value (nodeKey), a pointer[2] to the indexed entity (entPtr), and pointers to the left and right subtrees (leftChild and rightChild).

[2] In C++, a void* may legally point to any type of object.

```
class binaryTreeNode {
        float                   nodeKey;
        void                    *entPtr;
        binaryTreeNode          *leftChild;
        binaryTreeNode          *rightChild;
public:
        binaryTreeNode( float, void * );    /* constructor */
        void * search( float );
        void insert( binaryTreeNode* );
};

binaryTreeNode::binaryTreeNode( float insertKey, void * insertPtr ) {
        nodeKey = insertKey;
        entPtr = insertPtr;
        leftChild = rightChild = NULL;
}

void * binaryTreeNode::search( float searchKey ) {
        if( searchKey == nodeKey )
                return entPtr;
        else if( searchKey < nodeKey )
                if( leftChild == NULL )
                        return NULL;
                else
                        return leftChild→search( searchKey );
        else
                if( rightChild == NULL )
                        return NULL;
                else
                        return rightChild→search( searchKey );
}

void binaryTreeNode::insert( binaryTreeNode* newNode )
        if( newNode→nodeKey == this→nodeKey ){
                return;  /* no duplicates allowed */
        else if( newNode→nodeKey < this→nodeKey )
                if( leftChild == NULL )
                        leftChild = newNode;
                else
                        leftChild→insert( newNode );
        else
                if( rightChild == NULL )
                        rightChild = newNode;
                else
                        rightChild→insert( newNode );
}
```

Figure 1a: Class Definition for Binary Tree Nodes

The keyword **public** introduces a set of member declarations that form the public interface to the class. The interface to binaryTreeNode comprises the methods search and insert, as well as one named binaryTreeNode.

These member functions are elaborated following the class declaration. The function binaryTreeNode is a *constructor* for its class. Constructors initialize class instances; the binaryTreeNode constructor initializes all the fields of a newly created node. C++ guarantees that if a class has a constructor, then that constructor will be invoked automatically whenever an instance of the class is created. If the constructor takes arguments, they must be supplied with the object's declaration. For example, in the declaration

```
binaryTreeNode  aNode( 0.0, NULL );
```

aNode is a binaryTreeNode instance initialized with a zero key and a null pointer. The member function search compares the node's key with the argument, searchKey, and either returns the node's entity pointer or recursively searches the appropriate subtree. If that subtree does not exist, the key is not to be found, and the null pointer is returned. The insert member function takes a pointer to a new node which is to be inserted into the tree. We assume that this node has been initialized with its key and pointer values. The routine searches for the proper position and adds the node as a new leaf. Note that duplicate keys are simply rejected; the next section will remedy this shortcoming.

Figure 1b gives the definition of the binaryTree class. As we said earlier, this class is really a thin wrapper around the node class, and it is mainly used to start the recursion, e.g., in a search. The representation of a binaryTree is a pointer to the root node. The binaryTree constructor initializes this pointer to NULL. To search the tree, we first check the root pointer, and if it is not NULL, we search the root node recursively. The insert member function contains an example of creating a node dynamically. The **new** operator returns a pointer to a node which has been allocated on the heap. Again, since we are creating an instance of a class having a constructor, we have provided arguments. If the tree is empty, the new node immediately becomes the root; otherwise, we pass the new node to the root, and the insert proceeds recursively.

3. ITERATORS

We now consider the first of two E extensions that were inspired by the CLU language [Lisk77]. Among its many contributions, CLU demonstrated that separating the production of a sequence of values from the use of those values is both elegant and highly practical. This control abstraction is called an *iterator*, as it generalizes the iteration found in **for** loops. An iterator comprises two cooperating agents, an iterator function (i-function) and an iterate loop (i-loop), that work together to produce and to process a sequence of values. The i-loop is a client of the i-function, which it views simply as the source of a stream of values. The i-function produces that stream by *yielding* each result value one-at-a-time to the client loop. Unlike a return from a normal function, when an i-function yields a value, its local state is preserved. When the i-loop requests another value, the i-function resumes execution. Thus, an i-function can be viewed as a limited form of coroutine, one that may be invoked only within the context of an i-loop.

```
class binaryTree
{
        binaryTreeNode* root;
public:
        binaryTree();    /* constructor */
        void * search( float );
        void insert( float, void * );
};

binaryTree::binaryTree() {
        root = NULL;
}

void * binaryTree::search( float searchKey ) {
        if( root == NULL )
                return NULL;
        else
                return root→search( searchKey );
}

void binaryTree::insert( float insertKey, void * insertPtr ) {
        binaryTreeNode * newNode = new binaryTreeNode( insertKey, insertPtr );
        if( root == NULL )
                root = newNode;
        else
                root→insert( newNode );
}
```

Figure 1b: Class Definition for Binary Trees

3.1. Iterators in E

We originally included iterators in E for their utility in structuring database query processing, although we quickly became convinced of their usefulness as a general programming construct. Syntactically, an iterator function looks like a normal function, except that the keyword **iterator** precedes the return type, and the function body may contain **yield** statements. An i-function may take parameters of any type and may yield values of any type. The code comprising the i-function body is arbitrary; an i-function may invoke other iterators and may be recursive.

Consider the example in Figure 2. The purpose of the i-function bigElements is to yield the elements of an unsorted integer array that are greater than the average of all of the elements. When bigElements is invoked, it first makes one pass through the array in order to compute the average. Then it makes a second pass, yielding each element that is larger than the average. At each yield point, bigElements suspends its execution while the client processes the element; when the client requests the next element, the i-function will resume after the yield point. When the **for** loop terminates, and control "falls out" the bottom, the i-function also terminates. (An iterator may also terminate by executing a normal **return**.) Although this example shows only one **yield** statement, in general, an i-function may have many.

An iterate loop comprises the keyword **iterate**, followed by one or more i-function invocations in parentheses, followed by a statement which forms the loop body. Each invocation supplies actual arguments to the i-function, and it declares a variable to receive the yielded values. For example, the following i-loop activates the i-functions f and g, where the yield types are int and foo, respectively:

```
iterate( int x = f();  foo y = g(); int z = f() ) { ... }
```

Note that there are two simultaneous activations of f, one associated with x and one with z.

An i-function may be invoked *only* within the context of an i-loop. Figure 2 also shows a main program containing an i-loop that uses the bigElements iterator. After initializing the array A, control enters the loop, and the i-function is activated. When control returns to the loop, nextEl holds the first value of the sequence. After the

```
iterator  int bigElements( int * array,  int size ) {
        float     sum = 0.0;
        float     ave = 0.0;

        if(size > 0) {
          /* first compute the average */
          for( int i = 0; i < size; i++ )
                  sum += array[ i ];
          ave = sum / size;
        }

        /* now yield the big elements */
        for( int i = 0; i < size; i++ )
                if( array[ i ] > ave )
                        yield  array[ i ];
}

main() {
        int       A[ 10 ];

        /* Initialize A */
        ...

        /* Now find big elements. */
        iterate( int nextEl = bigElements( A, 10 ) )
                printf("%d ", nextEl);
}
```

Figure 2: A Simple Iterator Example

loop body prints that value, control returns to bigElements if it is still active; if bigElements has terminated, then the loop also terminates, and control flows to the next statement in the program.

3.2. Flow of Control

In the above example, the flow of control through an iterate loop is implicitly defined. That is, at loop entry, and at the top of the loop in each iteration, the i-function is resumed in order to obtain the next value. The number of iterations is determined by the i-function; the loop iterates until the i-function decides to terminate. In addition, a single i-function controls the loop in this simple example. E provides for several variations on this theme, providing the programmer with more general control flow capabilities.

E allows multiple i-functions to be activated concurrently. In this case, the default flow of control resumes *all* i-functions at the top of the loop; the order of resumption is implementation-dependent. The loop terminates when all i-functions have terminated. If some i-functions have terminated while others are still active, the values of the loop variables associated with the terminated i-functions are unaffected by the resumption of the remaining active i-functions. In order to allow the program to determine which i-functions have terminated, E provides a built-in function, empty, that may be applied to any i-loop variable; empty(v) returns 1 if the i-function activation associated with variable v has terminated, and 0 otherwise. We will see an example shortly.

3.2.1. Advance

The default flow of control described above is too restrictive in certain cases. Consider an iterator that is supposed to yield a sorted stream of values by merging two sorted input streams. Naturally, we wish to produce the input streams using iterators, so the merge i-function is also a client of other i-functions. The default flow of control is inappropriate for this task. If we simply try

```
iterate( int val1 = stream1();  int val2 = stream2() ){ ... }
```

then we will march down the streams in lock-step, and the loop body may have to buffer an arbitrary number of values (up to the entire sequence produced by one of the streams). If we try nesting, i.e.,

```
iterate( int val1 = stream1() )
        iterate( int val2 = stream2() ) { ... }
```

then we will repeat the entire inner i-loop for each element considered by the outer. Clearly, more flexible control is needed.

The **advance** statement was introduced in part to meet this need. As an example, consider

```
advance  val2;
```

where this statement appears within the context of either of the two **iterate** loops above. The effect of the statement is to resume the i-function activation associated with val2, in this case, stream2(); after the **advance** statement, val2 has its new value. In its general form, **advance** may have a comma-separated list of variables; the i-function activation associated with each of the variables is resumed (in an implementation-dependent order). If any **advance** statement is executed on a given pass through the body of an i-loop, then *no* default resumptions are carried out, i.e., if any i-functions are advanced, then those are the *only* i-functions advanced for that iteration.

Figure 3 shows how the **advance** statement and the **empty** function may be used to implement the merge example. When control enters the i-loop, two i-functions, stream1 and stream2, are activated, and the loop variables, val1 and val2, receive their initial values. We first test to see if either i-function has terminated, and if so, we simply yield the element from the other stream. The default flow of control will then advance the one active i-function until it is exhausted. If both i-functions are active, then we yield the smaller value and explicitly advance the i-function from which it came; the other i-function will not advance on that iteration. The loop terminates when both i-functions have terminated.

3.2.2. Break

The merge example shows how the client loop can decide which i-function activation to resume on any given iteration. So far, though, loop termination has still been determined by the i-functions, i.e., the client iterates until all i-functions have terminated. Alternatively, a client may decide to **break** out of an i-loop; normally, this causes immediate termination of all active i-functions associated with that loop.

```
iterator int merge() {
        iterate( int val1 = stream1();
                int val2 = stream2() )
        {

          if( empty(val1) )
                  yield val2;
          else if( empty(val2) )
                  yield val1;
          else if( val1 < val2 ){
                  yield val1;
                  advance val1;
          } else {
                  yield val2;
                  advance val2;
          }
        }
}
```

Figure 3: Using the **advance** Statement

A given i-function may sometimes require explicit control over the termination sequence, however. It may, for example, need to release heap space or to perform other bookkeeping tasks. To handle such cases, we have extended the yield syntax with an optional clause. This clause is a statement which is executed only if the client terminates the i-loop while the i-function is suspended at that yield point. For example, suppose that an i-function has built some structure which it must deallocate before terminating, and suppose the variable p points to the root of the structure. Then in the following example, if the client breaks after the i-function has yielded x, the (user-defined) cleanUp routine will be called before the i-function terminates:

```
yield x : cleanUp(p);
```

In the absence of this clause, no further i-function code is executed before termination.

3.3. A Recursive Iterator Example

As a final example, Figure 4 modifies the binary tree implementation from the previous section so that it handles duplicate keys. For brevity, we show only the search routine. Assume that we have amended the insert routine so that it no longer rejects a duplicate entry; instead, if it finds a match, it recursively inserts the new entry into the left subtree. Now, since we must be prepared to find many entries with the same key value, we have rewritten the tree search in Figure 4 as an iterator which yields a sequence of pointers to all of the entities with matching keys. If the search key is greater than the key in the current node, we simply yield the results of searching the right subtree. If the search key is less than or equal to the current node's key, then we search the left subtree, again yielding each

```
iterator void * binaryTreeNode::search( float searchKey )
{
        if( searchKey <= nodeKey )
        {
                if( leftChild != NULL )
                        iterate( void * p = leftChild→search( searchKey ))
                                yield p;

                if( searchKey == nodeKey )
                        yield entPtr;
        }
        else if( rightChild != NULL )
                iterate( void * p = rightChild→search( searchKey ))
                        yield p;

 }
```

Figure 4: Recursive Search Iterator

result to the level above. Finally, if the keys are equal, then we yield the entry in the current node after the subtree search has terminated.

At the top level, the client picks up the return values one-by-one. At any point in the client loop, there is a stack of active i-functions corresponding to levels of the tree. This situation is illustrated in Figure 5. On the right is a sample tree built by inserting the keys, 5, 3, 8, 5, 2, 5 (in that order). Assume that the client has asked to search the tree for all entries with the key value 5. To the left of the tree, we show the stack of recursive iterator activations at a point immediately before execution of the first **yield**. (The arrow next to the stack indicates the direction of growth.) Each activation is labeled with its current line number from the code of Figure 4.

We note that the client may choose to break out of the loop before all duplicate entries have been yielded; this event triggers a cascading termination of all active i-functions. Although the yield statements in the binary tree search iterator do not contain termination clauses (since none are needed), any such clauses would be executed as described above, beginning with the newest activation on the stack.

3.4. Iterators in Other Languages

Iterators have appeared in various forms in many languages. Contemporaneously with CLU, Alphard allowed the programmer to define *generators* (i-functions) and provided both a for loop and a first loop for iterating over the results [Shaw77]. To write a generator in Alphard, the programmer defined an Alphard *form* (similar to a C++ class) that provided two functions: init and next. The execution of a for (or first) loop would first invoke init and then repeatedly invoke next. A similar solution is possible (and advocated) in C++, which does not provide any special support for iterators [Stro86]. That is, one defines a class whose data members maintain the state of the

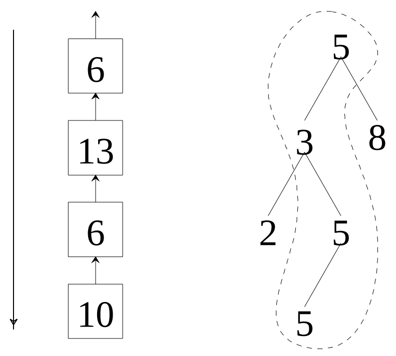

Figure 5: Operation of Recursive Search Iterator

iteration and whose member functions include one to initialize the state and one to return a value and advance the state. While C++ does not define a special construct for invoking iterators, a fairly clean solution is possible using the standard for loop along with a few simple coding conventions.

The Alphard and C++ approach to writing iterators has two disadvantages relative to languages like CLU and E. First, for each type of object to be iterated over, one must usually define a separate class (or form) for each kind of iterative access desired. This is required because the programmer is made responsible for explicitly saving the state of an iterator between invocations, and the data types involved in the iteration state depend on the type of the target object of the iteration. Second, coding an iterator that has multiple **yield** points can significantly complicate the logic required in the next function[3], as the programmer must then explicitly save and resume the control state as well as the data state of the iterator.

Neither of these problems occur with CLU-like iterators, since preservation of data and control state is part of their semantics. Such iterators have appeared in several other languages since CLU, e.g., Trellis/Owl [Scha86]. Several database programming languages (DBPLs), which are languages designed to simplify the task of writing database applications, offer a restricted form of iterator that yields the results of a database query. Examples include Pascal/R [Schm77], Plain [Wass79], and O++ [Agra89]. These languages allow the programmer to write arbitrary i-loops, but i-functions can be specified only implicitly via predicates over sets of objects. Rigel [Rowe79], another DBPL, offers a more general design; it allows both implicit i-functions, for iterating over built-in container types (such as lists, arrays, and sets), and programmer-provided i-functions.

In comparing E to these languages, recall that E was intended to be a systems programming language, where control flow must be under explicit programmer control. Moreover, since E was intended to be useful as a target language for database query compilation, E must provide low-level iteration support sufficient for implementing queries. Thus, E does not provide the high-level, query-oriented iterators found in many DBPLs. Instead, all i-functions in E are programmer-defined, as in CLU and Trellis/Owl. On the other hand, E supports parallel invocation of i-functions, something not found in CLU or Trellis/Owl.[4] Rigel and O++ both support a restricted form of parallel invocation, permitting the individual tuples satisfying the join predicate in a query to be obtained and processed. In E, parallel iterators are provided to support general stream-merging computations. E's **advance** statement is unique in allowing the explicit resumption of i-functions required for such processing. Finally, both Rigel

[3][Shaw77] mentions the difficulty of writing certain kinds of iterators such as those to compute recurrence relations. As a simple example, consider generating the Fibonacci numbers. The first two numbers are computed differently from the rest, so the next function must know whether it is returning the first, second, or n-th (n > 2) number. In E, we would simply write an i-function having three yield points.

[4]This flexibility is not without cost, however. The more restricted semantics of CLU iterators allow for an efficient stack-oriented implementation [AtkR78], while i-function activations in E must be allocated on the heap.

and E provide a means for i-functions to clean up after the premature termination of an i-loop. Rigel permits just one cleanup clause for each i-function as a whole, whereas E permits one cleanup clause per yield point.

4. GENERATOR CLASSES

As mentioned in the introduction, one of the problems facing the DBI is that much of the code for a large and flexible system such as a DBMS must be written without knowledge of the specific types of objects that the code will eventually manipulate. A traditional DBMS has knowledge of a few basic attribute types "wired in." The basic operators and access methods can operate on any of these types, essentially by switching on the type of the attribute at hand. One obvious problem with this approach is that the set of basic types is fixed, and therefore the system is difficult to extend. Another problem is that, in order to handle different record types, offset and length information must be passed explicitly to each routine. In addition, the programmer is responsible for coding offset calculations and for interpreting untyped buffer pages. One of the original goals of E was to make such mechanical tasks implicit. We were inspired by CLU *generators* [Lisk77] (not to be confused with Alphard generators) as providing an elegant solution to the problem.

A generator is a parameterized type, i.e., one that is defined in terms of one or more unknown (formal) types. The classic example is the generic type `stack[ T ]`, which, given any element type `T`, defines the type of a stack of `T` elements. In the case of our binary tree, we will make it a generic class by introducing two type parameters: the type of the key and the type of the entity being indexed.

4.1. Generator Classes in E

E introduces parameterized types in the form of *generator classes*[5]. A generator class may have any number of class parameters; the formal class names may be used freely within the generator like regular type names, e.g., as data member types, as argument or return types for member functions, or as a basis for other type definitions. Syntactically, a generator class has the form of a regular class, except that the formal parameters are specified in square brackets following the class name. The parameters themselves have the form of (skeletal) class declarations. For example, the definition of a bounded generic stack class is shown in Figure 6. We shall omit showing the `stack` member functions, since the only notable feature is that `T` is used wherever the name of the element type is needed.

[5]We note that generics have recently been added to C++ (v.3.0 and beyond) in the form of *templates*. E generator classes and C++ templates will be compared in Section 4.3.2.

4.1.1. Instantiation

In order to use a generic class, we must first *instantiate* a specific class by supplying actual arguments to the generator. For example, assume we have defined class frame; we can then define the type of a stack of frames by:

```
class frameStack : stack[ frame ];
```

Given this definition, we can now declare and use frameStack instances. For example:

```
frameStack      S1;     // S1 is a frameStack instance
frame           f;      // f is a frame
S1.push( f );           // push f onto S1
```

Attempting to push anything but a frame onto S1 will be flagged as a type error at compile time.

4.1.2. Constraints on Class Parameters

If a class parameter is specified with an empty body, as in the stack example, then any type may be used to instantiate the generator. We could, for example, define intStack to be stack[int], even though int is not really a class. However, the programmer may also specify constraints on instantiating types by "fleshing out" the parameter class body with member function declarations; only classes having member functions with the same names and type signatures can be used to instantiate the generic class. Furthermore, within the generator, these member functions may be invoked on objects of the parameter type. For example, the binary tree class can be made generic by introducing two type parameters, one of which is the key type. In order for a key type to be useful, however, we must be able to compare two key values to determine ordering. One means of accomplishing this is to constrain the key type:

```
class stack[ class T { } ]
{
        int     top;            // top-of-stack index
        T       stk[ 100 ];     // the elements
public:
                stack();        // constructor
        T       pop();
        void    push( T );
        int     isEmpty();
};
```

Figure 6: A Generic Stack Class Declaration

```
class binaryTreeNode
[
        class keyType
                { public: int compare( keyType* ); },

        class entityType { }
]
{ ... };
```

With this declaration, an actual class may be bound to keyType only if it has a public member function compare that takes a keyType pointer and returns an integer. Within the search routine, we can now compare keys as follows:

```
int cmpVal = searchKey.compare( &nodeKey );
if( cmpVal < 0 )
        { ... }
else if( cmpVal == 0 )
        { ... }
else
        { ... }
```

Of course, there is an additional requirement that the integer returned by the compare function be less than, equal to, or greater than zero corresponding to the ordering of the two keys. Such semantic constraints cannot be expressed within the type system, however.

4.1.3. Function Parameters

One shortcoming of the above approach is that, while the names of class parameters are formal names, the names of any member functions included as constraints are actual names. In the example above, any class may instantiate keyType provided that it has a member function whose name is literally compare. While this may be useful in some contexts, in others it may be too restrictive. For example, we may have a pre-existing class that has a comparison routine, but the routine's name may not be compare. We may also have a class defining several different comparison routines corresponding to different criteria for ordering instances. To overcome this problem, E allows member functions of a parameter class to be named as separate, formal parameters to the generic. For example, we could redefine our generic node class as follows:

```
class binaryTreeNode
[
        class   keyType{ },
        class   entityType{ },
        int     keyType::compare( keyType* )
]
{ ... };
```

The parameter keyType may be instantiated with any class having some member function that takes a keyType pointer and returns an integer; the name of the specific member function must be supplied at instantiation time. If the programmer wishes, the formal or actual member functions (or both) may even be overloaded operators. (For our examples, we have chosen to pass a single compare routine, but we could have instead passed a pair of overloaded operators, < and ==.)

To illustrate these points, assume that we have a class dataPoint for recording experimental observations and that we wish to build an index over such points. The key is to be a complex number taken from the experimental data, where complex is defined as follows:

```
class complex {
        /* representation... */
public:
        int cmpImag( complex* );    // compare imaginary parts
        int cmpReal( complex* );    // compare real parts
};
```

We may then instantiate a node type in which the keys are ordered by their imaginary parts as follows:

```
class complexNode : binaryTreeNode[ complex, dataPoint, complex::cmpImag ];
```

Despite the flexibility provided by member function class parameters, the approach above still falls short in some cases. One remaining problem is that it is not possible to directly instantiate a binary tree with a fundamental key type (e.g. float), as such types are not classes and do not have comparison methods. Another problem is that it is a relatively common C++ coding practice to define comparison routines (and other symmetric binary operators) for a class as **friend** functions[6] rather than as member functions of the class. Thus, while the actual key type might be a class, the comparison routine might not be a member function. To handle these cases, E also allows normal functions to be generic class parameters, as the following example illustrates:

```
class binaryTreeNode
[
        class   keyType{ },
        class   entityType{ },
        int     compare( keyType*, keyType* )
]
{ ... };
```

Here, any type (including a fundamental type) may instantiate keyType, and any (non-member) function having a matching signature may used to instantiate compare. This is the solution we will use in our ongoing binary tree example.

[6]A class can declare any function to be a **friend**. Such a function is not a member of the class, but it is permitted to access the class's representation.

4.1.4. Constant Parameters

The last kind of class parameter that E supports is a constant value. The parameter may be of any fundamental type and, within the generator class, it may be used freely as a **const**. The value used in an instantiation must be a compile time constant. This is particularly useful in defining array members whose size depends on the particular instantiation. For example, we may define a generic stack class such that the maximum number of elements is a class parameter:

```
class stack [ class T { }, int STKMAX ]
{
        int     top;
        T       stk[ STKMAX ];
        ...
};
```

We may then define a class for stacks of one hundred integers as follows:

```
class intStack : stack[ int, 100 ];
```

4.2. Nested Instantiations

We have shown how to define binaryTreeNode as a generator class. In order to define the wrapper class, binaryTree, as a generic, we must arrange for binaryTreeNode to be instantiated automatically whenever a user's program instantiates binaryTree.

E allows a new type to be defined within the scope of a class, along the lines of C++ 2.1 [Elli90]. In E, creating types in this way includes defining a type by instantiating a generic. Furthermore, within the context of a generator class GA, we may instantiate another generator GB by supplying GA's parameters to GB; then any instantiation of GA with actual parameters causes a nested instantiation of GB. Thus, we can make the binaryTree class a generator as shown in Figure 7. Within the context of binaryTree, a new class btn is instantiated from binaryTreeNode by passing along the parameters supplied to binaryTree. We will complete this class definition in the next section.

4.3. Discussion and Comparison with Related Work

In E, a generic class is not a true type, and no objects can be declared of such a class. This is in contrast to languages like ML [Miln84] and Fun [Card85], whose type systems support truly generic objects, e.g., a generic identity function in which the return type depends on the type of the actual parameter. E follows the more common practice of defining a generic to be a "mold" for creating new types, and instantiation of a type is defined to be (essentially) equivalent to macro expansion of the generic class definition. Note, however, that this does not imply a macro expansion implementation. (In fact, E follows CLU [AtkR78] in compiling generic code that is shared by all instantiations.) Other languages that define generics similarly include CLU [Lisk77], Ada [Ichb79], Trellis/Owl

```
class binaryTree
[
        class   keyType{ },
        class   entityType{ },
        int     compare( keyType*, keyType* )
] {
        class btn : binaryTreeNode[ keyType, entityType, compare ];
        btn  *root;
public:
        binaryTree();
        entityType * search( keyType );
        void insert( keyType, entityType* );
};
```

Figure 7: A Generic Binary Tree Class

[Scha85], and Eiffel [Meye86].

Another language that supports generics of this sort is the most recent version of C++, i.e., C++ v.3.0 [Elli90]. A generic class or function in C++ v.3.0 is called a **template.** Unlike an E generator class, a template class cannot specify constraints on the instantiating type. However, the member functions of a template class can invoke methods of the (unknown) actual parameter type; this design implies that instantiating a template with a type that does not supply the required methods is an error that may not be detected until link time [Elli90]. In E, such an error would be caught at compile time. Unlike C++ templates, E does not directly provide generic functions. The same effect can be achieved in E, however, by defining the function as a **static** member of a generic class.[7]

In C++ v.3.0, a class created from a template can serve as a base class for further subtyping. Furthermore, a template class T1 may inherit from a non-template class C or from another template class T2. In the former case, any class instantiated from T1 becomes a subclass of C. In the latter case, any class instantiated from T1 becomes a subclass of one instantiated (automatically) from T2 with the same parameters. For example, in C++ v.3.0, we can define Set<T> to inherit from Collection<T> so that for a specific type T', Set<T'> is a subtype of Collection<T'>. Trellis/Owl [Scha85] and Eiffel [Meye88] also support this kind of definition.

Similar to C++, E allows a class instantiated from a class generator to serve as a base class for further subtyping. E also allows a generator class to inherit from other types or generics, thus supporting the incremental definition of generic types. Unfortunately, E does not define any subtype relationships between types instantiated

[7] If a member function of a C++ class is declared **static**, then it exists independently of any object of the class and may be invoked directly by using the scope resolution operator (::). For example, C::f( ); is a valid invocation of static member function f of class C.

from such classes. From a language design standpoint, this is certainly a flaw; fortunately, it has not proved to be a problem in practice. On the other hand, E's support for constraints on class parameters has been very useful. Finally, E's implementation of generics in terms of shared, generic code has allowed us to write large, separately-compiled generic modules (e.g., a B+ tree index). Such large packages would have been impractical with a macro-expanding implementation.

Generator classes were designed and implemented prior to the appearance of the C++ template design [Stro88]. Thus, E diverged from C++ in this respect. Later, while templates were still only a design, we were content to continue. Now that templates are finally available, however, we have had to reconsider. Despite the relative strengths of E's generator design and the possibility of correcting its flaws, we currently plan to drop generator classes in favor of templates when we move to version 3.0. Clearly, it would be a mistake to support both generators *and* templates, as the result would likely be impossible to understand or to maintain. Equally clearly, the C++ community is going to become familiar with templates and will be (justifiably) reluctant to program in E if they are required to switch to generators. Given these realities, adopting C++ templates in future releases of E seems like the best choice.

5. DB TYPES & PERSISTENCE

In the discussion so far, we have described language extensions in E that allow the programmer to process sequences of values and to define parameterized types. Both features are important for database-style programming. However, the data objects available to the program thus far are still volatile objects whose lifetimes are bounded by a program run. We now introduce the features of E that allow a program to create and use persistent objects and thus to describe a database or object base, together with its operations, strictly within the language.

5.1. Database Types

E mirrors the existing C++ types and type constructors with corresponding database types (db types) and type constructors. Any type definable in C++ can be analogously defined as a db type. Db types are used to describe the types of objects in the database, i.e., the database schema. However, not every db type object is necessarily part of a database; db type objects may also be allocated on the stack or in the heap. We will shortly convert the binary tree class into a db type.

Let us informally define a db type to be:

(1) one of the fundamental db types: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid. Fundamental db types are fully interchangeable with their non-db counterparts, e.g., it is legal to multiply an int and a dbshort, assign a dbint to a float, etc.

(2) a dbclass (or dbstruct, or dbunion). Every data member of a dbclass must be of a db type. The argument and return types of member functions may be either db or nondb types.

(3) a pointer to a db type object. The usual kinds of pointer arithmetic are legal on db pointers, and casting is allowed between one db pointer type and another. It is *not* possible to convert a db pointer into a normal (non-db) pointer, nor into any non-pointer type (e.g., int).

(4) an array of db type objects. As in C or C++, an array name is equivalent to a pointer to its first element.

(5) a reference to a db type object.

5.2. The persistent Storage Class

Having db types allows the E programmer to define the types of objects in the database. The **persistent** storage class provides the basis for populating the database. If the declaration of a db type variable specifies that its storage class is **persistent**, then that variable survives across all runs of the program and across crashes. A simple example is a program that counts the number of times it has ever been run:

```
persistent dbint count = 0;
main() { printf("This program has been run %d times.", count++ ); }
```

Here, the integer count is a persistent variable whose initial value is set to 0. Each time the program runs, it prints the current value of count and then increments it. Note that there are no explicit calls to read or write count, and there are no references to any external files; I/O is implicit in the program. The great convenience of language support for persistence is that it allows the programmer to concentrate on the algorithm at hand rather than on the details of moving data between disk and main memory [AtkM83].

In the first implementation of persistence, the E compiler interacted with the runtime system to reserve a storage location for the object; this address was compiled into the code as a constant [Rich89b]. The current implementation uses a more flexible scheme that defers binding to a storage location until runtime. The persistent store keeps a map for translating the variable names in each E source file into their corresponding storage locations in the EXODUS Storage Manager. When an E program starts to run, it first interacts with the persistent store to obtain the current object address of every persistent variable named in the program. If an object has no current address, either because the program is running for the first time or because the object has been deleted[8], it is created at that time. While this scheme is a substantial improvement over the original implementation, there are still problems concerning naming that we shall discuss in Section 6.

[8]Deletion of named persistent variables is not defined within the language. We provide a separate utility for this task.

5.3. Collections

E provides the built-in generator db class collection[T] to allow the dynamic creation and deletion of objects. For a specific type T', a collection[T'] may contain objects of type T' or any subtype of T'. The lifetime of an object within a collection is bounded by the lifetime of the collection; in particular, if a program creates an object in a persistent collection, then that object will also be persistent. Like any generic class, the programmer must first instantiate a specific type of collection before declaring a collection object. As with objects of any db type, a given collection may be volatile or persistent, depending on the declaration, and it may be a data member of another class. It should be noted that it is also possible to define heap-like collections, capable of containing objects of any type, by instantiating the built-in collection class with the argument dbvoid.

5.3.1. Creating Objects in a Collection

In C++ v.2.0, one may overload the **new** operator in order to take control of storage allocation. E uses this mechanism to allow programs to create objects in a collection. As an example, suppose that person is defined as a **dbclass** and that it has a constructor that takes a character string containing the person's name. The following E code defines a type describing collections of persons, declares an instance of that type, and creates two people within the collection:

```
dbclass person { ... };
dbclass City : collection[ person ];
City Madison;
person * p1 = new( Madison ) person("Jane");
person * p2 = new( Madison, p1 ) person("Toby");
```

As shown above, the overloaded **new** operation takes one or two additional arguments. The first argument specifies the containing collection for the newly allocated object; the argument can be any expression that evaluates to a collection, as long as the type specified for the new object is the same as or a subtype of the type of entity in the collection. The compiler can verify this condition since both the type of the collection and the type of the object being created are manifest. In this example, we are creating instances of person in a collection of persons, but if, for example, student were a subtype of person, we could also create student instances within this collection.

Note that a collection in E is similar to a typed heap in that objects are allocated and deallocated in them, rather than inserted or removed. Since an object can exist in only one collection, some tasks can be slightly awkward. For example, to simulate the object's appearing in another collection C, we would have to declare C as a collection of pointers. However, this design is in keeping with the purpose of E: to be a low-level language supporting the implementation of higher-level data models. For example, a class extent in a higher-level data model could be implemented as a collection of objects, while sets could be implemented as collections of pointers.

5.3.2. Physical Clustering

When objects are stored on disk, their locations relative to one another can have a significant impact on overall performance. Generally speaking, objects that will be used together should be stored together if possible. The second **new** argument, which is optional, allows the programmer to communicate physical clustering hints to the storage layer. The second **new** in the example above requests that the new person (Toby) be created near the object referenced by p1 (Jane).[9] In general, the second argument to **new** may be any pointer-valued expression, and the referenced object need not be of the same type nor in the same collection as the newly created object. It is up to the implementation of the underlying storage layer to determine what "near" means, and at worst, the hint will be ignored. In the current implementation of the EXODUS Storage Manager, the search for a nearby location begins on the same disk page if the objects are allocated in the same collection, and on the same disk cylinder otherwise [Care89].

5.3.3. Scanning Collections

The collection generator class has an iterator member function, scan(), for scanning all of the elements in a collection. This iterator returns a sequence of pointers to the objects in the collection. The following example processes all of the people in Madison:

```
iterate( person * p = Madison.scan() ){ ... }
```

Note that even though a collection of T may contain objects of a subtype of T, a scan always returns T pointers. For example, the preceding scan always yields a person*, although some instances in the collection might be of type student. The introduction of multiple inheritance in C++ v.2.0 posed some challenges in the implementation of collection scans. The challenges stem from the fact that an E collection is implemented as an EXODUS Storage Manager file, which only records the OIDs of the objects that it contains. A file scan thus produces a pointer to the beginning of each object. However, this is not necessarily the correct pointer to return to the E program. For example, suppose class C has supertypes A and B, and we create a C object in a collection[B]. Now suppose that we scan the collection, obtaining a B pointer to each object in turn. When the scan encounters the C object, the returned pointer must be adjusted to refer to the "B part" of the object. Mechanisms to accomplish this and to handle virtual base classes have been implemented.[10] Note that the existing C++ mechanism for moving up

[9] In E v.1.2, **in** and **near** clauses were added as **new** syntax extensions. We have elected to drop this syntax in favor of the **new** operator overloading capability introduced in C++ v.2.0. Under the old E syntax, the second **new** example would have read:

```
p2 = in( Madison ) near( p1 ) new person("Toby");
```

[10] The offset of the B part is stored in the object header (provided by the EXODUS Storage Manager), and the scan iterator adjusts the pointer by this amount before returning it to the client. If B is a virtual base class, then the object header contains the offset of the virtual base pointer within the object, in which case the scan iterator retrieves the pointer out of the object itself and returns it. In either case, the appropriate B part must be unambiguous or else an error will be reported by the compiler when it processes the new statement that attempts to create a C object in the collection[B].

the type hierarchy (in which the compiler inserts code to adjust the pointer) does not apply in this situation, as it is not known until runtime that a particular object returned by the scan is actually of type C.

5.3.4. Destroying Objects and Collections

The usual C++ **delete** operator may be used to remove an object from a collection. For example, we can delete Toby (in the previous example) with:

```
delete p2;
```

If the object's type has a destructor, the destructor will be called first and then the object will be destroyed.

If a collection is destroyed, the objects that it contains are also destroyed. If the collection contains objects of a class having a destructor, then the destructor will be invoked on each object before the collection is destroyed. Assume that we wish to delete Madison, which is a collection[person]. Conceptually, this process involves the following steps:

```
iterate( person * p = Madison.scan() ) { delete p; }
/* now destroy the empty collection... */
```

For performance reasons, however, our implementation does not actually destroy the objects individually. Rather, the destructor calls de-initialize each object, and then the entire collection is destroyed en masse.

The semantics of persistent object destruction parallel those of volatile object destruction and consequently inherit all of the same problems. In particular, dangling references are possible. Since the storage layer never reuses object ids, however, we prevent the worst effect of dangling references, i.e., the overwriting of random data. Other problems associated with explicit deletion, such as creation of garbage, are not addressed by our design. C++ relies on destructors to ensure proper cleanup when an object is deleted. In designing E, we elected to extend the existing semantics to persistent objects rather than attempting to define implicit deletion semantics for C++.

5.4. The Binary Tree Example Revisited

Let us now (finally) reimplement our binary tree example as a db type. Unlike the previous incremental examples, here we reproduce the entire implementation for comparison with the original C++ version. The node class shown in Figure 8a has changed from the C++ version of Figure 1a in the following ways: The insert routine accepts duplicates, and the search routine is an iterator (as developed in Section 3). The key and entity types are type parameters, and the key comparison routine is a function parameter (as developed in Section 4). The class itself is a **dbclass** (as developed in this section).

Figure 8b shows the binary tree class. In order to define this class, we must first instantiate two new classes which we will then use. The class btn is binaryTreeNode instantiated with the same parameters as for binaryTree, i.e., this is a nested instantiation as described in Section 4. Next, btnSet is instantiated as a type of

```
dbclass binaryTreeNode [
        dbclass         keyType{ },
        dbclass         entityType{ },
        int             compare( keyType*, keyType* )
] {
        keyType                 nodeKey;
        entityType              *entPtr;
        binaryTreeNode          *leftChild;
        binaryTreeNode          *rightChild;
public:
        binaryTreeNode( keyType, entityType * );    /* constructor */
        iterator entityType * search( keyType );
        void insert( binaryTreeNode * );
};

binaryTreeNode::binaryTreeNode( keyType insertKey, entityType * insertPtr ) {
        nodeKey = insertKey;
        entPtr = insertPtr;
        leftChild = rightChild = NULL;
}

iterator entityType * binaryTreeNode::search( keyType searchKey ) {
        int cmp = compare( &searchKey, &nodeKey );
        if( cmp <= 0 ){
                if( leftChild != NULL )
                        iterate( entityType * p = leftChild→search( searchKey ))
                                yield p;
                if( cmp == 0 )
                        yield entPtr;
        } else
                if( rightChild != NULL )
                        iterate( entityType * p = rightChild→search( searchKey ))
                                yield p;
}

void binaryTreeNode::insert( binaryTreeNode * newNode ) {
        int cmp = compare( &(newNode→nodeKey), &nodeKey );
        if( cmp <= 0 )
                if( leftChild == NULL )
                        leftChild = newNode;
                else
                        leftChild→insert( newNode );
        else
                if( rightChild == NULL )
                        rightChild = newNode;
                else
                        rightChild→insert( newNode );
}
```

Figure 8a: The Binary Tree Node Class

```
dbclass binaryTree
[
        dbclass         keyType{ },
        dbclass         entityType{ },
        int             compare( keyType*, keyType* )
]
{
        dbclass btn : binaryTreeNode[ keyType, entityType, compare ];
        dbclass btnSet : collection[ btn ];

        btnSet          allNodes;
        btn             *root;

public:

        binaryTree();    /* constructor */
        iterator entityType * search( keyType );
        void insert( keyType, entityType * );
};

binaryTree::binaryTree()
{
        root = NULL;
}

iterator entityType * binaryTree::search( keyType searchKey )
{
        if( root == NULL )
                return;
        else
                iterate( entityType * p =  root→search( searchKey ))
                        yield p;
}

void binaryTree::insert( keyType insertKey, entityType * insertPtr )
{
        btn * newNode = new( allNodes ) btn( insertKey, insertPtr );
        if( root == NULL )
                root = newNode;
        else
                root→insert( newNode );
}
```

Figure 8b: The Binary Tree Class

collection containing btn nodes. The binary tree itself is now represented as allNodes, a collection containing the nodes, and root, a pointer to the root node. On an insert, the new node is allocated in the tree's collection. Other changes to the binary tree class parallel those made for the node class, i.e., the use of type parameters, and the definition of search as an iterator.

Finally, Figure 8c shows an example using a persistent binary tree index. This program builds an index over students keyed on grade point average (gpa). Since the students must persist, we first define school as a collection of students, and we declare a persistent instance, UWmadison, of this type. We then define a comparison routine for floating point numbers, and we use this routine, along with the types student and dbfloat, to instantiate a specific index type. Next we declare a persistent index, gpaIndex. Finally, the main program shows examples of creating a new student and adding the corresponding index entry and of iterating over all students with a given gpa.

5.5. Implementing a Disk-Based Index

The binary tree example developed in this paper is clearly hinting at the implementation of "real" database index structures, e.g., B+ trees, in which each node contains many keys. In defining such structures, an essential constraint is that each index node must fit on one disk page and must make maximal use of the space on that page.

```
dbclass student{ ... };
dbclass school : collection[ student ];
persistent school   UWmadison;

int compare( dbfloat * x, dbfloat * y )
{
        float cmp = (*x - *y);

        if( cmp < 0 )
                return -1;
        else if( cmp == 0 )
                return 0;
        else
                return 1;
}
dbclass gpaIndexType : binaryTree[ dbfloat, student, compare ];
persistent gpaIndexType gpaIndex;

main() {
        student *   s;

        s = new( UWmadison ) student( ... );
        gpaIndex.insert( s→gpa, s );

        iterate( student * s = gpaIndex.search( 3.0 ))
                ...

}
```

Figure 8c: Example Using a Persistent Binary Tree

If we define the node type as a generic class, then the number of keys that will fit on a page varies with the specific key type. One approach is to define the generator with a constant parameter, as we did in the stack example of Section 4. However, this approach forces the user of the class to compute the maximal number of keys for each instantiation.

An easier approach is to make use of the fact that within a generator, the expression sizeof(T), where T is a type parameter, is treated as a constant and may be used in declaring array bounds. For example, assume that PAGESIZE is a constant giving the size of a disk page in bytes. In Figure 9, we have outlined the definition of a simplified generic class describing leaf nodes in a B+ tree; each node is to contain an array of key-pointer pairs where the number of array elements is the maximum that will fit on one page. Like the binary tree example, this class is parameterized by the key and entity types and by the key comparison routine. For convenience, we have defined an auxiliary type, kpp, for key-pointer pairs; the tree node is an array of these structures. Note that since kpp is defined in terms of class parameters, it is also a generic type, and it is implicitly instantiated with each instantiation of BTreeLeaf. We then define two macros for convenience. The amount of usable space on a page is the size of the page minus any overhead for control information; in this simple example, the only control data is an

```
dbclass BTreeLeaf
[
        dbclass         keyType{ },
        dbclass         entityType{ },
        int             compare( keyType*, keyType* )
] {
        /* auxiliary definitions */
        dbstruct        kpp {
                keyType         keyVal;
                entityType *    entPtr;
        };

#define MAXSPACE        (PAGESIZE - sizeof(dbint))
#define MAXENTRIES      (MAXSPACE / sizeof(kpp))

        /* data members */
        dbint   nKeys;
        kpp     kpPairs[ MAXENTRIES ];

public:
        ...
};
```

Figure 9: A Generic DbClass for B+ Tree Leaf Nodes

integer giving the current number of entries in the array. Finally, the maximum number of array entries is the amount of available space divided by the size of an entry, i.e., by sizeof(kpp). The data member, kpPairs, is then defined to be an array whose dimension is this maximum.

5.6. Comparison with Other Persistence Mechanisms

5.6.1. Reachability

The style of persistence provided by E might be termed "allocation-based" persistence. That is, an object is persistent only if it is created as such, either by being declared a **persistent** variable or by being created within a persistent collection. This approach is quite different from reachability-based persistence, in which an object persists if it is reachable from one or more distinguished roots. A number of persistent languages and OODBMSs to date have adopted the latter approach, e.g., PS-Algol [AtkM83], Galileo [Alba85], GemStone [Maie86], Zeitgeist [Ford88], and PGraphite [Tarr90]. Reachability is perhaps a more convenient mechanism for the programmer, who no longer needs to worry about persistent objects containing dangling references, e.g., to a reclaimed volatile object. It also allows greater flexibility in that a program can decide whether an object should become persistent after it already exists. In E, a volatile object can be "made" persistent only by copying its value into a persistent object.

We elected not to base E's persistence on reachability for several reasons. As mentioned earlier, we first envisioned E as being a target language for the compilation of higher level data models. We felt that in such a language, the persistence of a given object should be an explicit property rather than an implicit side-effect of being part of a particular data structure. Perhaps a more compelling reason (for E) is that reachability fits most naturally into a garbage-collected language. That is, the reachability traversal for determining persistence requires the same information as that for garbage collection. Because E is an extension of C++, basing E's persistence on reachability would have required that database types (at least) be quite different from their C++ counterparts, something that we wanted to avoid. For example, we would probably have had to disallow persistent **union**s (or change the semantics of **union**) since they do not carry enough information to determine which member is "active."

5.6.2. Collections and Class Extents

A central issue in the design of a database programming language is how (or whether) collections of persistent objects are defined. Pascal/R [Schm77] introduced **relation** as a type constructor; tuples could be added or deleted under program control, although the relations themselves could only be named variables. One implication of this restriction is that nested relations were not allowed. Other DBPL's, e.g., Rigel [Rowe79] and Plain [Wass79], took a similar approach with similar restrictions. PS-Algol [AtkM83], the first language providing fully general (orthogonal) persistence, made the runtime heap the basis for persistence. Any object reachable from a distinguished "database root" pointer would persist. However, the persistent heap had no notion of a collection of objects; such a collection would have to be coded explicitly as a persistent data structure. E takes an intermediate approach. Like the DBPL's, a given collection stores a specific type of object, and there are facilities for processing all of the objects in a collection. Like PS-Algol, there are no restrictions on the type of object that may be persistent (except that it must be a db type); for example, one may define collections of collections. E does *not* provide an implicit persistent heap, however; the dynamic creation of a persistent object in E requires the specification of a collection in which to create the new object.

Another popular approach to persistence is to support class extents. An extent is the set of all instances of a class; when an instance is created, the system automatically places the object within the proper extent. Programs can then use class extents to form the basis of queries. Most systems that support extents also define inclusion semantics for subtypes, so the extent of a class includes the extents of all subclasses. A query over an extent may specify whether or not to include subclass extents. Examples of systems that support extents include Orion [Bane87], O2 [Banc88], PCLOS [Paep88], and O++ [Agra89].

Extents are a simple, convenient, and fairly natural way to add persistence to an object-oriented language, especially if one is focusing on database applications. In fact, extents may be seen as an extension of relational DBMS semantics: just as a relation defines both a tuple type and all existing tuples, so does a class define an object type and all existing instances. However, we did not feel that extents were appropriate for E, since they associate persistence with types rather than with instances. In a general-purpose implementation language like E, we wanted to be able to implement extents if desired, but not to force them in all cases.

6. DISCUSSION

So far, we have presented the design of the E language and have shown examples of its use. Of course, the true test of a language's practicality is in its implementation and actual use. We have built several E compilers, and we have explored five distinct implementations of persistence [Rich89b, Rich90, Schu90, Shek90, Whit92]. Furthermore, we (and others) have used E to build a number of persistent applications. This experience has revealed the language's weaknesses as well as its strengths, and it uncovered some interesting implementation challenges. In this section, we describe some of the more interesting issues. A detailed evaluation of E will appear in a forthcoming paper.

6.1. Language Design Issues

6.1.1. Naming Persistent Objects

The idea of defining a new storage class struck us as a very clean approach to extending C++ with a persistence mechanism. Just as an **auto** variable lives for one procedure invocation, and a **static** variable lives for the lifetime of a process, a **persistent** variable was to live across processes, as a kind of "super" static object. Under this definition, the name of a persistent variable is not merely a handle for some object in the persistent store, but it is actually a denotation of the object itself.

In retrospect, this approach has drawbacks in the area of name space management. Since the names of all top-level persistent objects are variable names, the name space seen by an E program must obey C++ scope rules. The sufficiency of these rules for writing large non-persistent applications is questionable, and they are even more restrictive for organizing a large database. For example, sharing can be hampered by the fact that a program cannot include two persistent variables (from two different E modules) that have the same name, just as clashes in transient global variable names are disallowed. Similarly, long term program evolution is hampered since it is not currently possible to change the name of a persistent E object once it has been created. Finally, writing general-purpose code can be impeded by the early binding of names to objects, although the use of procedures can largely alleviate this problem.

While it is possible to circumvent most of these problems, doing so requires a fair amount of foresight and care when developing a large E program. An alternative would have been to separate the names of objects in the persistent store from the names used in the program text. For example, we could have chosen to limit persistent variable access to pointer traversals, requiring pointer variables to be bound to specific objects via runtime calls (similar to opening a file). Our feeling was that adding persistence as a storage class was more natural, somehow; moreover, it doesn't prevent E programmers from simulating the latter approach by building and using a directory class to manage the name space of persistent objects dynamically.

6.1.2. Orthogonality

One of the more controversial design points was our decision to give E a "two-headed" type system, rather than simply to introduce persistence as an orthogonal property of all types. Orthogonality is, after all, often cited as a desirable feature of a persistent language [AtkM83, AtkM87], and some users of E have complained about its dual approach [Hans91]. The motivation for E's use of db types stems both from philosophy as well as implementation concerns. First, E was originally conceived as a language in which to write database management systems. In such systems, there is a clear distinction between those objects that persist and those that are volatile. For example, lock tables and transaction descriptors are definitely not persistent, while objects in the database definitely are. The "db" attribute of a type distinguishes between objects that *may be* persistent and those that are definitely volatile. We note that O++, which was designed more recently than E, makes a similar distinction [Agra89].

The separation of normal types from db types also has a strong grounding in performance considerations. Those same system resources that are known to be volatile (e.g., those mentioned in the previous paragraph) are often the ones that are accessed with the highest frequency. If every object reference might be a persistent reference, then every access must check that the needed object is in memory.[11] Even if the check costs only one boolean test, we might still significantly increase the cost of accessing these critical system resources. In E, accesses to non-db type objects suffer *no* loss of performance over the same accesses in C++.

In addition to the cost of a pointer dereference, another factor in our decision to introduce db types was the representation of pointers. On a VAX, a pointer is 32 bits, giving an address space of approximately 4 GB. However, databases are already exceeding this limit and, in fact, are moving into the terabyte range. If persistence were orthogonal to all types, then we would be faced with several not very appealing alternatives. We could make all pointers 32 bits, and thus guarantee that E would already be obsolete for real world applications. Alternatively, we could make all pointers be "large enough" for our projected needs. In the current implementation, E uses the EXODUS Storage Manager [Care89] as its persistent store, so every E pointer involves an EXODUS object id (12 bytes) plus an offset (4 bytes). Thus, this approach would quadruple the space needed, and also the copying cost, for every pointer; neither would be desirable from the standpoint of achieving good program performance [Rich89a].

6.2. Implementation Issues

While the preceding sections discussed issues related to the language design, in this section we turn our attention to shortcomings in its implementation. As we shall see, these problems stem largely from the C/Unix model of creating and running programs, a model which we attempted to preserve in E. It has become all too apparent that this model is insufficient for supporting a persistent language and that an integrated programming environment is needed.

6.2.1. Type Persistence

In C and C++, type definitions are shared between compilation units via textual inclusion of header files. There is no notion of a type existing outside of any particular run of the compiler. A given type, `T`, used to compile one module may or may not be the same `T` used to compile another. This mechanism is acceptable (though not particularly desirable) for C and C++ programming. For E programs, however, it is critical that the definition of a type remain consistent across compilations. If an object created with type `T` is later manipulated by a program with a different definition of `T`, the database can be easily corrupted. If persistent objects were limited to simple C structures, then very careful programming could avoid this problem. For an object-oriented language with persistence,

[11] This is not necessarily true for all persistent language implementations. For example, it is possible to implement an object-faulting mechanism that relies on the machine's paging hardware (e.g., see [Shek90, Lamb91]). However, such mechanisms tend to limit either the space of addressable objects [Shek90] or the space of concurrently addressable objects [Lamb91] to the size of the machine's virtual address space; they can also have performance problems when programs operate on large (relative to physical memory) databases [Shek90].

however, even careful programming is insufficient. In order to support late binding of method invocations, each object must carry enough information to identify its own type uniquely. In the face of persistence, the identity of a type must also persist.

This issue of persistent type identity is fundamentally at odds with the existing model of writing C programs under Unix. A complete solution to these problems requires an integrated programming environment that provides, among other features, a type library in which a type has an identity independent of any particular compilation. It also requires answering some hard questions such as: How are types shared between programmers? If a type changes, is it still the same type? If so, what happens to existing objects of that type, and if not, how can old programs use the new type?[12]

These issues extend well beyond the scope of our original intentions in designing E. Thus, our implementation provides only a partial solution to the type identity problem, a solution based on computing a hash value from the class definition. An approximation to type identity is achieved in this way, without environment support, since the same class definition will hash to the same value in different compilations. When a persistent class instance is created, the class's hash value is stored in the object, later identifying the object's type during method dispatch. Of course, hash collisions are possible, though extremely unlikely; an E program that includes two classes with the same hash value simply terminates itself at startup time. While we considered several alternative designs, this one seemed to be the best initial compromise.

6.2.2. Type Availability

While computing hash values provides an initial solution to the type persistence problem, there is another related issue that it does not address. The problem is that it is possible for a program to encounter an object whose type was not known when the program was compiled. To see how this problem arises, let us first consider the implementation of C++. Here, it is assumed that every object encountered by a program was created by that program. As a result, for every type of object encountered, there is at least one module of the program (i.e., the one that created the object) that knows about its type. The implementation of virtual function dispatch can then assume that both the method dispatch table (vtbl) and the method code itself are somewhere in the program's address space.

Unfortunately, these assumptions can break down in the presence of persistence. Suppose we have a persistent graph, G, whose nodes are of type T1. Suppose that we write a program P1 that traverses G, invoking a virtual function on each node. In order to compile P1, we need only include definitions for T1 and its base classes (if

[12]The latter questions relate to the problem of schema evolution, a well-known hard problem in database systems. Researchers in object-oriented database systems have offered several approaches to the problem [Bane87, Penn87, Skar87], although none seems entirely satisfactory.

any). Now suppose we write another program P2 that defines T2 as a subtype of T1 and adds a T2 node to G. If we then run P1, the program will terminate with an error when it invokes the virtual function on the new node. Note that this is not a type error; T2 is a subtype of T1, and the invocation is legal. The problem is that neither the vtbl for T2 nor any of T2's methods are available to P1.

As was the case with type persistence, the root of the problem lies not in the language design itself, but in the language's implementation within the context of an environment that does not adequately support persistence. Providing such support would involve a significant amount of effort. Programs would have to become objects in the database, and an execution would involve incremental dynamic linking of method code. The environment would have to track dependencies between programs and the persistent objects they create. Tools would have to be provided to allow users to build, execute, debug, and evolve programs within the environment. This, in turn, would require us to define a model of programs and the programming process. While these are all very interesting and important problems, they fall far outside the scope of the EXODUS project. Thus, at least for the present, maintaining E programs under Unix can sometimes be awkward.

6.2.3. Transactions

The first implementations of E had only a primitive notion of transaction. Each program run constituted one transaction. Current support is somewhat better: library calls are available to begin, commit, or abort a transaction. These calls are supported by the current implementation of the EXODUS Storage Manager, which provides atomic, recoverable transactions. The persistent data touched by a transaction is locked in a two-phase manner, and recovery is provided via write-ahead logging. At present, a program is limited to executing a series of independent, flat transactions; in the future, we may provide nested transactions. We also expect to integrate E's transaction support with the recently introduced facilities for handling exceptions in C++ [Elli90].

7. OTHER RELATED WORK

Throughout this paper we have compared particular features in E with other languages having similar features. Thus, most of the important comparisons with other languages have already been made. In this section, we focus our attention on comparing E with several other languages that have also extended C++ with persistence.

7.1. Avalon/C++

Avalon/C++ [Herl87, Detl88] is a language designed to support reliable distributed computing. This language utilizes the inheritance mechanism of C++ to allow programmers to design data types having customized synchronization and recovery properties. Persistence is then modeled as a set of objects encapsulated by a server; a server may recover the state of its objects after a crash. E differs from this approach in that its main goal is to provide transparent persistence for structuring large object bases and transparent I/O for manipulating them. In E, persistent objects exist independently of any active process.

Using inheritance to declare db types might have been a reasonable alternative for adding this feature into C++. Instead of defining a type as a **dbclass**, one would define it as a **class** that inherits (directly or indirectly) from a special predefined class, db. There are two problems with such an approach, however. The first is that the fundamental C++ types (e.g., int) are not classes, so we cannot define their db analogs in terms of inheritance. The second problem is that with the addition of multiple inheritance, it would be possible to define a class that inherits from both db and non-db classes. It is not clear how one would define a consistent meaning for such a class, and we might end up having to simply disallow such cases.

7.2. O++

Researchers at AT&T Bell Laboratories designed and implemented a language, O++, that seeks to blend both high-level and systems-level programming features [Agra89]. Like E, O++ is also an extension of C++ including persistence. However, O++ maintains class extents, and it provides support for integrity constraints and triggers. Like most DBPLs, O++ also provides a form of iterator for expressing calculus-like queries over type extents; two variations of this looping construct allow for querying either the extent of a single type or the extents of a type and all of its subtypes. Despite its higher-level (more application-oriented) features, O++ still shares many of the basic limitations and problems that E inherits from C++, e.g., the possibility of storing a persistent pointer to a volatile object.

One interesting point of comparison between E and O++ is in their requirements for defining the type of a *possibly* persistent object. In E, one associates the "db" attribute with a type; a pointer to that type can potentially refer to a persistent object. In O++, one associates the type modifier "dual" with the declaration of a given pointer to indicate that it might refer to a persistent object.[13] (Thus, all db pointers in E are essentially "dual" pointers.) While at first glance, O++ may seem to afford more flexibility, the two methods are essentially equivalent. Consider defining a class which contains a pointer data member. If objects of this class may persist, then E requires that the class be defined as a db type, while O++ requires that the pointer be given the "dual" type modifier. Now consider a function which takes a pointer argument. Again, if we desire the function to be able to handle pointers to persistent as well as nonpersistent objects, then E requires that the pointer refer to a db type, while O++ requires that the argument declaration have the "dual" type modifier. A potential advantage of E's approach is that the "possibly persistent" attribute need appear in only one place, i.e., with the type's definition. However, a potential advantage of the O++ design is that the syntax isolates the essence of the difference between persistent and nonpersistent types to

[13]O++ also has a type modifier "persistent" (not to be confused with E's **persistent** storage class) which indicates that the pointer *definitely* refers to a persistent object.

the one place where it really makes a difference, i.e., pointers.

7.3. ObjectStore

In the past two years, several startup companies have introduced object-oriented database products based on a C++ data model. Of particular relevance here is the ObjectStore system from Object Design [Lamb91]. Unlike most of the commercial offerings, which offer persistence only through library interfaces, ObjectStore also extended the C++ language with persistence in order to provide a tightly integrated, language-based interface to their database system. ObjectStore was designed with essentially the same goals as O++, having been intended for use in complex, data-intensive applications such as CAD, CAE, CASE, and geographic information systems (GIS).

ObjectStore has a number of features in common with E. ObjectStore shares E's basic allocation-based approach to persistence: C++ is extended with a `persistent` storage class, and persistent objects are obtained by declaring variables of this storage class or allocating objects within a persistent collection. ObjectStore improves on E's approach by partitioning its persistent storage class into named databases, thereby alleviating the naming problem discussed earlier. Unlike E, persistence is orthogonal to type in ObjectStore; all pointers are the same size as normal C++ pointers, and virtual memory mapping techniques and auxiliary data structures are used to manage I/O and to handle databases larger than virtual memory [Lamb91]. (A driving goal in the design of ObjectStore was the desire to get as close to C++ pointer dereferencing speeds as possible for applications whose working sets fit in main memory.) Like E, ObjectStore provides a type-parameterized collection class — a C++ template — for use in representing sets of objects. Since ObjectStore (like O++) is intended as an end-user DBPL, it provides several variants of its collection type in the form of an object class library. Other more advanced features of ObjectStore include support for inverse members (which model relationships between objects), associative queries and indexing, versioned data, and cooperative transactions.

8. CURRENT STATUS

We currently have two E v.2.1 compilers working, both based on the AT&T cfront v.2.1 translator. These two E compilers differ in how they manage I/O for persistent objects. One caches objects in the buffers of the EXODUS Storage Manager and uses a hash table to locate cached objects [Schu90]; the other copies objects into virtual memory and swizzles pointers, converting them from object id-based pointers into virtual memory addresses while their target objects are resident [Whit92]. Other E implementations have experimented with compile-time approaches to scheduling and optimizing calls to the storage layer [Rich89b, Rich90], and we have also explored an implementation of persistence based on virtual memory mapping under the Mach operating system [Shek90]. Our experience shows that each implementation can do well for certain classes of applications, and research is continuing on how to make E programs fast and robust across a wide range of database sizes.

In addition to E v.2.1, a version of E based on C++ v.3.0 is currently under development. This version will be based on the Gnu g++ compiler, which will enable us to freely distribute E via anonymous ftp (which is how other EXODUS software is already handled). The implementation of E v.3.0 is well underway, and we expect to be distributing this version of E by late Spring of 1992. As mentioned earlier, E v.3.0 will depart from earlier versions of E in that generic types will be supported via the C++ template design; the collection generator class will be appropriately redefined as a template. Other features of E (iterators and persistence) will remain as they are in E v.2.1.

The EXODUS toolkit has been distributed to over 35 external sites to date. E has been (and is being) used to construct a variety of data and object management prototypes, including a small demonstration relational DBMS at the University of Wisconsin [SIG88], a DBMS with an integrated production-rule system at Wright State University [Hans89], a nested relational DBMS at the Air Force Institute of Technology [Harv91], an object manager to support the ARCADIA program development toolset at the University of Colorado [Heim91], the EXTRA/EXCESS and MOOSE object-oriented database systems at the University of Wisconsin [Care88, Ioan89], and a database system to support the CAPITL programming environment project at the University of Wisconsin [Solo92]. The initial experiences of two of our external user sites were reported recently in [Hans91].

ACKNOWLEDGEMENTS

We would like to thank all of the members of the EXODUS project for their willingness to play the role of guinea pigs so patiently. In particular, Mike Zwilling wrote numerous programs for the E test suite and was an invaluable aid in finding v.1.2 compiler bugs. Paul Bober and Dan Lieuwen contributed similarly to both v.1.2 and v.2.0 through their extensive use of the language. We would also like to thank Larry Rowe for his suggestions regarding iterators, Marvin Solomon for pointing us in the direction of C++, and (most of all) David DeWitt for numerous helpful discussions and constant feedback. Finally, we would like to thank the referees for comments that helped us to improve the presentation significantly.

REFERENCES

[Agra89] Agrawal, R., and Gehani, N.H., "ODE (Object Database and Environment): The Language and the Data Model," *Proc. ACM-SIGMOD Conf.,* Portland, OR, June, 1989.

[Alba85] Albano, A., Cardelli, L., and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Trans. on Database Systems*, 10(2), June 1985.

[AtkM83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R., "An Approach to Persistent Programming," *Computer Journal*, 26(4), 1983.

[AtkM87] Atkinson, M.P., and Buneman, O.P., "Types and Persistence in Database Programming Languages," *ACM Comp. Surveys,* 19(2), June, 1987.

[AtkR78] Atkinson, R., Liskov, B., and Scheifler, R., "Aspects of Implementing CLU," *Proc. ACM National Conf.,* 1978.

[Atwo90] Atwood, T., and Hanna, S., "Two Approaches to Adding Persistence to C++," *Proc. 4th Int'l Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Banc88] Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lecluse, C., Pfeffer, P., Richard, P., and Velez, F., "The Design and Implementation of O2, an Object-Oriented Database System," *Proc. 2nd Int'l. Workshop on Object-Oriented Database Systems*, Bad Munster, FRG, Sept., 1988.

[Bane87] Banerjee, J., Kim, W., Kim, H.J., and Korth, H.F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May, 1987.

[Bato88] Batory, D.S., Leung, T.Y., and Wise, T.E., "Implementation Concepts for an Extensible Data Model and Data Language," *ACM Trans. on Database Systems,*, 13(3), Sept., 1988.

[Card85] Cardelli, L., and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, 17(4), December, 1985.

[Care86a] Carey, M.J., and DeWitt, D., "Extensible Database Systems," in *On Knowledge Base Management: Integrating Artificial Intelligence and Database Technologies*, M. Brodie and J. Myopoulos, eds., Springer-Verlag, 1986.

[Care86b] Carey, M.J., DeWitt, D.J., Frank, D., Graefe, G., Richardson, J.E., Shekita, E.J., and Muralikrishna, M., "The Architecture of the EXODUS Extensible DBMS," *Proc. Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986.

[Care88] Carey, M.J., DeWitt, D.J., and Vandenberg, S.L., "A Data Model and Query Language for EXODUS," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June, 1988.

[Care89] Carey, M.J., DeWitt, D.J., Richardson, J.E., and Shekita, E.J., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley Publishing Co., 1989.

[Care90] Carey, M.J., DeWitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., and Vandenberg, S.L., "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, eds., Morgan-Kaufman, 1990.

[Daya86] Dayal, U., and Smith, J., "PROBE: A Knowledge-Oriented Database Management System," in *On Knowledge Base Management: Integrating Artificial Intelligence and Database Technologies*, M. Brodie and J. Myopoulos, eds., Springer-Verlag, 1986.

[Detl88] Detlefs, D., Herlihy, M., and Wing, J., "Inheritance of Synchronization and Recovery Properties in Avalon/C++," *IEEE Computer*, 21(12), December, 1988.

[Elli90] Ellis, M., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Ford88] Ford, S., Joseph, J., Langworthy, D., Lively, D., Pathak, G., Perez, E., Peterson, R., Sparacin, D., Thatte, S., Wells, D., and Agarwala, S., "ZEITGEIST: Database Support for Object-Oriented Programming," *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.

[Gold83] Goldberg, A., and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Grae87] Graefe, G., and DeWitt, D., "The EXODUS Optimizer Generator," *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May, 1987.

[Hans89] Hanson, E., "An Initial Report on the Design of Ariel," *SIGMOD Record* 18(3), Sept. 1989.

[Hans91] Hanson, E., Harvey, T., and Roth, M., "Experiences in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Phoenix, AZ, Oct. 1991.

[Harv91] Harvey, T., Schnepf, C., and Roth, M., "The Design of the Triton Nested Relational Database System," *SIGMOD Record* 20(3), Sept. 1991.

[Heim91] Heimbigner, D., personal communication, Oct. 1991.

[Herl87] Herlihy, M., and Wing, J., "Avalon: Language Support for Reliable Distributed Systems," *Proc. 17th Int'l. Symp. on Fault-Tolerant Computing,* Pittsburgh, PA, July, 1987.

[Hosk90] Hosking, A., and Moss, J.E.B., "Towards Compile-Time Optimizations for Persistence," *Proc. 4th Int'l Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Ichb79] Ichbiah, J., Barnes, J., Heliard, J., Krieg-Bruckner, B., Roubine, O., and Wichmann, B., "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices,* 14(6), 1979.

[Ioan89] Ioannidis, Y., and Livny, M., "MOOSE: Modeling Objects in a Simulation Environment", *Proc. IFIP 1989, 11th World Computer Congress*, San Francisco, CA, August 1989, pp. 821-826.

[Kern78] Kernighan, B., and Ritchie, D., *The C Programming Language,* Prentice-Hall, 1978.

[Lamb91] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D., "The ObjectStore Database System," *Comm. ACM,* 34(10), Oct., 1991.

[Lisk77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU," *Comm. ACM,* 20(8), Aug., 1977.

[Maie86] Maier, D., Stein, J., Otis, A., and Purdy, A., "Development of an Object-Oriented DBMS," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, 1986.

[Meye86] Meyer, B., "Genericity Versus Inheritance," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Sept. 1986.

[Meye88] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[Miln84] Milner, R., "A Proposal for Standard ML," *Proc. 1984 Symposium on Lisp and Functional Programming,* New York, August, 1984.

[Paep88] Paepke, A., "PCLOS: A Flexible Implementation of CLOS Persistence," *Proc. European Conference on Object-Oriented Programming,* Oslo, Norway, 1988.

[Penn87] Penny, D.J., and Stein, J., "Class Modification in the GemStone Object-Oriented DBMS," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Orlando, FL, October, 1987.

[Rich87] Richardson, J.E., and Carey, M.J., "Programming Constructs for Database System Implementation in EXODUS," *Proc. ACM SIGMOD Conf.,* San Francisco, CA, May, 1987.

[Rich89a] Richardson, J.E., "E: A Persistent Systems Implementation Language," Ph.D. Thesis, University of Wisconsin, Madison, August, 1989.

[Rich89b] Richardson, J.E., and Carey, M.J., "Persistence in the E Language: Issues and Implementation," *Software—Practice & Experience* 19(12), Dec. 1989.

[Rich90] Richardson, J.E., "Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language," *Proc. 4th Int'l Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Rowe79] Rowe, L., and Schoens, K., "Data Abstraction, Views, and Updates in RIGEL," *Proc. ACM SIGMOD Conf.,* 1979.

[Rowe87] Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," *Proc. 13th VLDB Conf.,* Brighton, England, 1987.

[Scha85] Schaffert, C., Cooper, T., and Wilpolt, C., "Trellis Object-Based Environment: Language Reference Manual," DEC-TR-372, November, 1985.

[Scha86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C., "An Introduction to Trellis/Owl," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Sept. 1986.

[Schm77] Schmidt, J.W., "Some High Level Language Constructs for Data of Type Relation," *ACM Trans. on Database Systems*, 2(3), 1977.

[Schu90] Schuh, D.T., Carey, M.J., and DeWitt D.J., "Persistence in E Revisited — Implementation Experiences," *Proc. 4th Int'l Workshop on Persistent Object Systems,* Martha's Vineyard, MA, Sept. 1990.

[Schw86] Schwarz, P., Chang, W., Freytag, J.C., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H., "Extensibility in the Starburst Database System," *Proc. Int'l. Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, 1986.

[Shaw77] Shaw, M., Wulf, W., and London, R., "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," *Comm. ACM*, 20(8), Aug., 1977.

[Shek90] Shekita, E.J. and Zwilling, M., "Cricket: A Mapped, Persistent Object Store," *Proc. 4th Int'l Workshop on Persistent Object Systems,* Martha's Vineyard, MA, Sept. 1990.

[SIG88] EXODUS Software Demonstration, presented at the ACM SIGMOD Conference, Chicago, IL, May, 1988.

[Skar87] Skarra, A., and Zdonik, S.B., "Type Evolution in an Object-Oriented Database," in *Research Directions in Object-Oriented Programming,* B. Shriver and P. Wegner, eds., MIT Press, 1987.

[Solo92] Solomon, M., personal communication, Jan. 1992.

[Ston86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," *Proc. 2nd Int'l Conf. on Data Engineering,* Los Angeles, CA, Feb., 1986.

[Stro86] Stroustrup, B., *The C++ Programming Language,* Addison Wesley, 1986.

[Stro88] Stroustrup, B., "Parameterized Types for C++," *Proc. USENIX C++ Conference,* Denver, CO, October, 1988.

[Stro91] Stroustrop, B., "The C++ Programming Language (2nd Edition)", Addison-Wesley Publishing Co., 1991.

[Tarr90] Tarr, P., Wileden, J., and Clarke, L., "Extending and Limiting PGraphite-style Persistence," *Proc. 4th Int'l Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Wass79] Wasserman, A. "The Data Management Facilities of PLAIN," *Proc. ACM SIGMOD Conf.*, 1979.

[Whit92] White, S., and DeWitt, D., "Pointer Swizzling in Virtual Memory: An Alternative Approach to Supporting Persistence in the E Programming Language," submitted for publication. (Also available as a Computer Sciences Dept. Tech. Report, Univ. of Wisconsin, Madison, Feb. 1992.)

APPENDIX: A Parts Database

In [AtkM87], Atkinson and Buneman proposed a set of four tasks to evaluate the expressiveness of database programming languages. We have coded and run their example tasks in E, and we present here excerpts from that code. The example is a parts database in which a given part is either a base part or a composite part. The four tasks are:

(1) Describe the database.

(2) Print the name, cost, and mass of all base parts that cost more than $100.

(3) Compute the total mass and total cost of a given composite part.

(4) Record a new manufacturing step in the database, that is, how a new composite part is manufactured from subparts.

Our implementation follows in the spirit of those described in [AtkM87]. That is, a part may either be a basePart or a compositePart. In our implementation, these classes are defined as subtypes of Part. A two-way linked list of Use objects maintains the many-to-many usage relation between parts. Every part P keeps a UsedIn list of other parts in which P is an immediate subpart. In addition, each composite part keeps a Uses list of its immediate subparts.

The database is defined as PartsDb, a class containing three collections: base parts, composite parts, and usage records. (Alternatively, we could have combined the first two into a single collection of parts.) This class provides a search routine that looks for a part with a given name. Finally, Database is declared as a persistent instance of this class.

To perform Task 2, the reporting of expensive parts, we simply iterate over the base parts in the database, printing the desired information for each qualifying record. Task 3, the recursive calculation of a composite part's cost and mass, is somewhat more interesting due to its use of virtual functions (the bodies of which are not shown). A base part simply returns its own cost and mass; a composite part recursively sums the cost and mass of its subparts, and then adds its own incremental cost and mass.

Finally, Task 4 adds a new composite part definition to the database. This routine assumes that it is given the name of the new composite part, its cost and mass increments, and a list of its subparts. The routine completes the task by creating a new composite part instance and then adding this instance to the UsedIn list of each of its subparts.

```
const MAXSTRING  = 16;
typedef dbchar String[MAXSTRING];

dbstruct Part;   // forward decl.
dbstruct Use {
        Part *          Uses;           // the subpart
        Part *          UsedIn;         // the composite part that uses it
        dbint           Quantity;       // how many of the subpart
        Use *           NextUses;       // next entry on composite part's uses chain
        Use *           NextUsedIn;     // next entry on subpart's used-in chain
};

dbstruct Part {
        String          Name;   // name of part
        Use *           UsedIn; // subpart-of chain

                        Part( char*, Use * );
        int             Match( char* );
        virtual void    costAndMass(dbint&, dbint&);
};

dbstruct basePart : public Part {
        dbint           Cost;   // cost of base part
        dbint           Mass;   // weight of base part

                        basePart( char*, Use *, dbint, dbint );
        virtual void    costAndMass(dbint&, dbint&);
};

dbstruct compositePart : public Part {
        Use *           MadeFrom;               // list of components
        dbint           AssemblyCost;           // additional cost to assemble components
        dbint           MassIncrement;          // additional mass to assemble components

                        compositePart( char*, Use *, dbint, dbint, Use *);
        virtual void    costAndMass(dbint&, dbint&);
};

dbstruct PartsDb {
        dbclass                 bpCollType  : collection[ basePart ];
        dbclass                 cpCollType  : collection[ compositePart ];
        dbclass                 useCollType : collection[ Use ];

        bpCollType              bPartsFile;     // all base parts
        cpCollType              cPartsFile;     // all composite parts
        useCollType             Uses;           // all uses/used-in links


        Part *  findPart( char* );
};

persistent PartsDb  Database;
```

Figure A1: Task 1: Describe the Database

```
void Task2() {
	iterate( basePart * p = Database.bPartsFile.scan() )
		if( p→Cost >= 100 ){
			printf("name = ");
			strprint( p→Name );
			printf(" cost = %d mass = %d", p→Cost, p→Mass);
		}
}
```

Figure A2: Task 2 - Print Out Expensive Parts

```
void Task3(char * name) {
	Part * p;
	int d = 0;
	int m = 0;

	p = Database.findPart( name );
	p→costAndMass( d, m );
	printf("name = %s, cost = %d, mass = %d", name, d, m );
}
```

Figure A3: Task 3 - Find Cost and Mass of a Part

```
void Task4( char* name, dbint cost, dbint massInc, Use * usesList ) {
	/* Assume usesList points to each subpart, but that subparts
	   have not yet been recorded as being used in this new part.
	*/

	compositePart * newPart;
	newPart = new( Database.cPartsFile )
	          compositePart( name,NULL,cost,massInc,usesList );

	for( Use * up = usesList; up != NULL; up = up→NextUses ) {
		// this use is due to the new composite part
		up→UsedIn = newPart;
		// insert use-record at head of subpart's used-in chain
		up→NextUsedIn = up→Uses→UsedIn;
		up→Uses→UsedIn = up;
	}
}
```

Figure A4: Task 4 - Add New Manufacturing Step