An Introduction to GNU E

Exodus Project Document

Computer Sciences Department
University of Wisconsin
Madison, WI   53706

1. INTRODUCTION

E is an extension of C++ designed for writing software systems to support persistent applications. GNU E is a new variant based on the GNU C++ compiler, based on the GNU C++ compiler. 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 document is an overview of the E language as implemented by the GNU E compiler. It is an abridged and modified version of [Rich92], which should be refered to for background information and design rational. All examples in this document have been updated to reflect usage in GNU E.

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 been ported to this version. This paper describes the use of the latest E variant, GNU E. GNU E is based on version 2 of the GNU C++ compiler.

GNU E differs from previous implementations based on the AT&T cfront compilation system. The only language extensions directly supported are the db type system and persistent data. Parameterized types in the form of class generators are no longer supported. C++ class and function templates are available and may be used for parmeterized type support. Iterators are no longer supported as part of the language. Instead, a macro package is provided which supports most of the functionality of the original E language construct, albeit in a somewhat clumsier form.

2. C++ REVIEW

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 [Rich92]

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 fol-

```
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

lows the class heading. Each node contains a floating point key value (nodeKey), a pointer[1] to the indexed entity

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

(entPtr), and pointers to the left and right subtrees (leftChild and rightChild).

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

```
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

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.

3.1. Iterators in GNU E

For GNU E, iterators are no longer supported as part of the language. Instead, a macro package is provided which supports most of the functionality of the original E language construct, albeit in a somewhat clumsier form. This macro package is enabled by the preprocessor directive

```
#include <E/iterator.h>
```

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. Under the macro implementation, an iterator function is declared by encapsulating the return type and the function name and prototype argument list as two parameters in the **ITER_DECL** macro:

```
ITER_DECL(int,f())
```

The iterator function is defined by bracketing the function definition with **ITER_DEF** and **END_ITER_DEF** macros. The **ITER_DEF macro takes the iterator function return type and the function name with prototype argument list as 2 parameters, like the ITER_DECL** macro; the **END_ITER_DEF** macro has no parameters.

```
ITER_DEF(int,f())
{
// function body
}
END_ITER_DEF
```

Within the definition of an iterator function, a normal return statement may not be used; values are passed back to the calling iterate loop via **YIELD** and **RETURN** macros. 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 **RETURN** macro to return a final value, or an **IRETURN** macro to terminate without yielding a final value.) Although this example shows only one **YIELD** macro, in general, an i-function may have many.

An iterate loop is implemented via an **ITERATE** macro, or one of the variant forms **ITERATE2**, **ITERATE3**, or **ITERATE4**. The simplest form, **ITERATE**, has 4 parameters: the type returned by the iterator function, the variable name for the iterator return variable, the i-function invocation, and a final parameter which should be an expression or compound statement constituting the iterate loop body. The **ITERATE2** variant has 3 additional arguements, defining an additional type, variable, and i-function invocation, and thus activates 2 iterator functions. Similarly, the **ITERATE3** and **ITERATE4** will activate 3 or 4 iterator functions, with a type, variable name, and i-function call for each i-function invocation as parameters, in addition to a final parameter which should be an expression or compound statement constituting the iterate 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:

```
ITERATE3( 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. The implementation used for the iteration loops allows more parallel activations, the limit of 4 activations is arbitrary.

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 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.

```
ITER_DEF(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 ]);
}
END_ITER_DEF


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

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, the E iterator macro package provides a macro, 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

```
ITERATE2(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** macro 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** macro, val2 has its new value. The **ADVANCE** macro is used to resume the i-function activation associated with one variable; there are variants **ADVANCE2**, **ADVANCE3**, and **ADVANCE4** that allow simultaneous activation of two, three, or four active i-functions, respectively. If any **ADVANCE** macro 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** and the **EMPTY** macros 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.

```
ITER_DEF( int, merge())
{
        ITERATE2(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);
           }
        })
} END_ITER_DEF
```

Figure 3: Using the **ADVANCE** macro

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, a **YIELD_BREAK** macro may be used as an alternative to the **YIELD** macro. The **YIELD_BREAK** macro takes two arguments: the first is the value to be yielded, and the second is is an expression or compound 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_BREAK(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.

```
1  ITER_DEF( void *, binaryTreeNode::search( float searchKey ))
2  {
3          if( searchKey <= nodeKey )
4          {
5                  if( leftChild != NULL )
6                          ITERATE( void *, p, leftChild→search( searchKey ),
7                                  YIELD(p));
8
9                  if( searchKey == nodeKey )
10                         YIELD(entPtr);
11         }
12         else if( rightChild != NULL )
13                 ITERATE( void *, p, rightChild→search( searchKey ),
14                         YIELD(p));
15
16  } END_ITER_DEF
```

Figure 4: Recursive Search Iterator

If the search key is less than or equal to the current node's key, then we search the left subtree, again yielding each 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.

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. Here all values are returned via the **YIELD** macro and not the **YIELD_BREAK** variant. If the **YIELD_BREAK** variant had been used to assure execution of termination clauses, any such clauses would be executed as described above, beginning with the newest activation on the stack.

4. GENERATOR CLASSES

GNU E no longer supports parameterized types in the form of generator classes. It relies on C++ templates as implemented in GNU C++ for parameterized type support. For more information on C++ templates see [Sto91] or [Stro88]. In the remainder of this paper, all the examples which previously used E generator classes in [Rich92] have been rewritten to use template classes.

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[2], it is created at that time. While this scheme is a substantial improvement over the original implementation, there are still problems concerning naming [RICH92].

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

5.3. Collections

In GNU E, the standard include file <E/collection.h> defines a collection class template. Instances of this collection template class may be used for 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 collection class template with the argument dbvoid.

5.3.1. Creating Objects in a Collection

In most current versions of C++ the **new** operator can be overloaded 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:

```
#include <E/collection.h>
dbclass person { ... };
collection<person> 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). 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

To facilitate scanning colletions, a collection_scan class template is provided allong with the collection template. This class template has the following public interface:

```
template <dbclass T> class collection_scan
{
  public:
  collection_scan( collection<T> & base_collection)
  T * next();
};
```

A collection_scan<T> object encapsulates the state needed to traverse all the elements in a collection<T> once. The construction for the collection_scan<T> variable is passed a collection<T> used to initialize the scan. Subsequent calls to the member function next of the collection_scan<T> object will return successive elements contained in the collection<T> used to initialize the collection_scan<T> variable, until all obects in the collection<T> have been scanned. After all elements of the collection have been scanned, the next member function will return the null pointer.

The following example processes all of the people in Madison:

```
{
  collections_scan<person> Madison_scan(Madison);
  person * scan_pt;
  while ((scan_pt= Madison_scan.next()) != NULL)
  {
    // process given element, pointed to by p.
  }
}
```

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.

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 is equivalent to scanning a collection and deleting each component element as it is scanned. 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. For any class t used as the key parameter, there must exist a function

```
#include <E/collection.h>
#include <E/iterator.h>

template < dbclass keyType, dbclass entityType, >  dbclass binaryTreeNode
{
        keyType                 nodeKey;
        entityType              *entPtr;
        binaryTreeNode          *leftChild;
        binaryTreeNode          *rightChild;
public:
        binaryTreeNode( keyType insertKey, entityType * insertPtr ) {
                nodeKey = insertKey;
                entPtr = insertPtr;
                leftChild = rightChild = NULL;
        };

        ITER_DEF( entityType * 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));
        } END_ITER_DEF;

        void 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 Template

```
int compare(t*, t*)
```

that determines an ordering on the class t analogous to the way the C library function determines a lexical ordering on C character strings. 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 first create a new nested typedef, btn, which is local to the binaryTree template and which, for any instantiation of the binaryTree template, represents a binaryTreeNode instantiated with the same class parameters. The binary tree itself is then represented as allNodes, a collection<btn> 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

```
template < dbclass keyType, dbclass entityType, > dbclass binaryTree
{
        typedef binaryTreeNode<keyType,entityType> btn;
        collection<btn> allNodes;
        btn             *root;

public:

        binaryTree() { root = NULL; };    /* constructor */

        ITER_DEF(entityType *, search( keyType searchKey )) {
                if( root == NULL )
                        return;
                else
                        ITERATE( entityType *, p,  root→search( searchKey ),
                                YIELD( p));
        } END_ITER_DEF;

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

Figure 8b: The Binary Tree Class

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{ ... };
persistent collection<student>   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;
}
persistent binaryTree<dbfloat,student> 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 template class, then the number of keys that will fit on a page varies with the specific key type. One approach is to define the template class with a constant parameter, 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 template, 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 class template 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 template 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 integer giving the current number of entries in the array. Finally, the maximum number of array entries is the

```
template <
        dbclass         keyType,
        dbclass         entityType,
>
dbclass BTreeLeaf
{
        /* 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

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.

5.5.1. 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. It is important to note that all accesses to persistent data must be with a transaction. We also expect to integrate E’s transaction support with the recently introduced facilities for handling exceptions in C++ [Elli90].

REFERENCES

[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.

[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.

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

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

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

[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.

[Rich92] J. Richardson, M. Carey, and D. Schuh, "The Design of the E Programming Language," ACM Transactions on Programming Languages and Systems, to appear.

[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.

[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.

[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 {
        collection<basePart>            bPartsFile;     // all base parts
        collection<compositePart>       cPartsFile;     // all composite parts
        collection<Use>                 Uses;           // all uses/used-in links


        Part *  findPart( char* );
};

persistent PartsDb  Database;
```

Figure A1: Task 1: Describe the Database

```
void Task2() {
        collection_scan<basePart> bpScan(Database.bPartsFile);
        basePart * p;
        while (( p = bpScan.next()) != NULL)
                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