The E Reference Manual

1. Introduction

The E programming language is C++ [Strou86], [Strou91], [Elli90] extended with db-types and persistence. It is intended for writing programs needing persistent objects, in particular, database system code. These extensions, together with the additional library classes and funtions described here, have been designed to facilitate such tasks. E is upward compatible with GNU C++.

The variant of E described in E is GNU E. This variant is the language implemented by the GNU E compiler, which is based on the GNU C compiiler, gcc, and its C++ variant, g++. GNU E no longer supports some of the C++ language extentions implemented in previous versions. Previous versions supported class generators for parameterized types; most of the functionality provided by class generators is now available via template classes in standard C++. Iterator functions are no longer supported at the language level; instead, a macro package is provided that supplies most of the functionality of the iterators.

This manual contains four main sections. The first part is written as a delta from the Reference Manual section of [Stro91]. The section numbers and headings correspond as much as possible to those in [Stro91], and only those sections are included which differ from the original. The other three sections describe parts of the E runtime environment that are not part of the language definition. These parts are collections for dynamic allocation of persistent storage, the iterator macro package, and the dbstrings library functions. Section numbers for these e specific sections are prepended with e in place of r prepended for sections corresponding.

This manual is quite terse. A more readable overview of GNU E may be found in [exo92]. A more comprehensive (and readable) description of E may be found in [Rich89b]. This paper, however, describes features that are no longer part of GNU E.

r2. Lexical Conventions

r2.4 Key Words

New key words:

```
dbchar          dbenum          dblong          dbunion
dbclass         dbfloat         dbshort         dbvoid
dbdouble        dbint           dbstruct        persistent
```

Except for the storage class specifier persistent, these keywords all correspond to C++ keywords used as type specifiers, with "db" prepended.

r3.5 Storage Classes

persistent Applicable only to db-type objects (§r3.6.1 and §r3.6.2). These objects retain their values between executions of the program. All runs of the program share the same physical object. If the persistent object is declared in a module which is linked to several different application programs, all programs will share the same object.

Currently persistent objects may not be declared within blocks. Also the current implementation does not support either **static** or **persistent** objects of classes with constructors to be declared in a block. (Sorry.)

A named object that is declared persistent survives until its containing module is explicitly destroyed via *expel* or

until the storage manager volume on which it resides is destroyed.

A persistent object may be created dynamically via the built in template class `collection` (§r7.3.3 and §e1).

r3.6.1 Fundamental Types

E provides a full complement of db-types which are duals of their C++ counterparts: `dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, dbvoid`. The representation of a fundamental db-type is the same as its dual. The lval of a db-type object is a DBREF (§r3.7).

r3.6.2 Derived Types

> *dbclasses*, *dbstructs*, and *dbunions* which look like their non-db counterparts, but are used for describing types in a database;[1]

In general, one may apply type constructors recursively, except that data members of a dbclass *must* be of some db-type. Db-type data members of nondb-type (normal) classes *are* allowed.

An object of type `dbvoid*` can be used to point to objects of unknown type.

r3.7. Objects and Lvalues

An *object* of a db-type is a region either of main memory or of persistent storage. An *lvalue* that refers to a db-type object is represented by an object id (OID) and an offset into the object. This OID/offset pair is called a DBREF. (Nonpersistent db-type objects (i.e. those that are allocated on the heap or stack) have lvalues in which the OID indicates memory, and the offset contains the memory address.)

r4. Conversions

Values of fundamental db-types may be freely converted in the manner of their duals, i.e. for conversion purposes, the fundamental db-types are equivalent to their nondb-type counterparts. For example, it is legal to assign a dbint to a dbchar, to a char, to a float, etc.

r4.6 Pointer Conversions

The constant 0 may be assigned with the same interpretation, i.e. 0 is the universal null db-type pointer.

Any db-type pointer may be converted to a `dbvoid*`.

A nondb-type pointer may be converted to a db-type pointer, but *not* vice versa. In the former case, a DBREF is manufactured (§r3.7).

```
int   x, *p;
dbint dx, *dp;

dp = &x;    // OK
p  = &dx;   // error
```

[1]Throughout this manual, the term "dbclass" is generally used to denote "dbclass, dbstruct, or dbunion." Furthermore, "class" often denotes "class or dbclass". Exceptions should be clear from context. In addition, the term *db-type* means a dbclass or any type derived from a dbclass, e.g. pointer-to or array-of. *Nondb-type* refers to normal C++ types.

• **CAVEAT:** If a db pointer is passed to a function that expects a regular pointer or vice versa, and if that parameter is unchecked (i.e., because the function has an ellipsis in its formal parameter list), an error will occur at runtime. For example, if `p` is a `dbchar*`, then

```
printf("The string is: %s", p );
```

will not do what the programmer expects because `printf` is defined as:

```
int printf(const char* ...);
```

Due to the ellipsis, the compiler cannot catch the fact that a db pointer is being passed where a `char*` is expected. The following example will fail for similar reasons:

```
dbint x;
scanf("%d", &x);
```

The dbString library (§e3) alleviates the problem of handling strings of `dbchar`. Currently, the only solution to the problem with `scanf` is to read into an `int` and then copy into the `dbint`:

```
dbint x;
int         y;
scanf("%d", &y);
x = y;
```

r5. Expressions

r5.3 Unary Operators

The "address-of" (&) operator is applicable to db-type objects. The result is a db-type lvalue (a DBREF). (§r3.7)

r5.4 Explicit Type Conversion

Integers may be converted to db-type pointers, but not vice versa.

A db-type pointer may be explicitly converted to another db-type pointer. For example, it is legal to convert (cast) a `dbchar*` to a `foo*`, provided `foo` is a db-type.

r7.3.3 New

The `new` operator may be used to allocate db type objects dynamically. The include file <E/collection.h> defines a set of overloaded new operators to accommodate the creation of objects in a collection (§e1). The placement overloading allows for the optional specification of a collection instance and the optional specification of a near hint. If no collection is specified, `new` allocates the object on the heap (as usual).

The predefined new overloading has the form:

`new (`*collection* `[ ,`*pointer* `]`$_{opt}$ `)` *new-specification*

where *collection* is an expression evaluating to a collection, *pointer* is a pointer valued expression, and *new-specification* is a type specification as in C++ (i.e. it may include constructor arguments). For example, suppose `emps` is a `collection < emp >`.

```
emp * ep;
emp * ep2;
ep  = new (emps) emp;   // create an employee in emps
ep2 = new (emps,ep) emp;      // create another near the first
```

If the type has a constructor that takes arguments, provide them in the usual way:

```
ep = new (emps) emp( ++eid, "Jones", 45000 );
```

The type of object created (i.e. the type following `new`) must be the same type or a publically derived subtype of

the type of object contained by the collection.

```
dbclass emp2 : public emp { ... };
dbclass emp3 : emp { ... };
new (emps) emp2;        // OK
new (emps) emp3;        // error - not publically derived
new (emps) dbint;       // error - not subtype
```

You may create a vector of T objects in a collection of T:

```
new (emps) emp[10];           // error
```

However, in this case the collection scan operater will return only a single pointer for the array; it will not return individual pointers for each array element.

The `delete` operator may be applied to db-type pointers. The referenced object is deallocated from the heap or from the collection, as appropriate. If the object is of a type with a destructor, that destructor will be called first.

r5.7 Additive Operators

Normal pointer arithmetic is allowed on db-type pointers. Arithmetic affects the offsets only; it is not possible change the OID in a pointer via arithmetic operations.

r5.9 Relational Operators

Relational operators are applicable to db-type pointers. If p1 and p2 point to the same storage manager object (i.e. have the same OID), then the relational operators compare the offsets. If p1 and p2 point to different objects, then only the inequality operator will return true.

r5.10 Equality Operators

Db-type pointers may be compared, using `==` and `!=`. Two pointers are equal if they point to the same object and have the same offset.

There is currently no easy way to determine whether two db pointers refer to the same object (ignoring differences in offset).

r7. Declarations

r7.1.1 Storage Class Specifiers

E adds an additional storage class specifier, `persistent`. The storage classes `auto`, `static`, `register`, and `extern` also apply to dbclass objects. All persistent objects must be declared with the persistent storage class specifier. If the extern specifier is not included in the declaration, the declaration is a definition and there must be exactly one definition for that object in a program. If an object is declared extern persistent, the object must be defined elsewhere. For any object defined as persistent, any non-defining declarations of the object must declare it as extern persistent.

`persistent` (§3.5) cannot be applied to functions or iterators.

r7.1.6 Type Specifiers

Add: `dbclass`, `dbenum`, `dbstruct`, `dbunion`.

r7.2 Enumeration Declarations

`dbenum` is not yet implemented.

r8.6 Initializers

A `persistent` object declared without an initializer is guaranteed to be initialized to all 0's.

r9 Classes

r9.2 Class Members

A class currently cannot have a persistent (§3.5) data member.

r10.2 Virtual Functions

A dbclass may have virtual functions.

r12.1 Constructors

If an object of a dbclass with a constructor is declared `persistent`, it is guaranteed that the constructor will be called exactly once in the object's lifetime, and that this event will occur before any user program references the object.

r12.4 Destructors

A dbclass may have a destructor which will be called whenever an object of the class is to be destroyed (i.e. goes out of scope or is destroyed with `delete`.)

If a collection (§e1) is to be destroyed such that the type of object in the collection has a destructor, that destructor will be invoked on each object in the collection first.

e1. Collections

The template class `collection` provides the E programmer with a means of manipulating sets of dynamically allocated objects. In particular, it allows E programs to create and destroy persistent objects at runtime; if an object is created in a persistent collection, then that object is persistent also. To use the collection template class, the program must include the template class definition, via

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

e1.1 Instantiation

A collection template class may be instantiated with any db type, e.g.

```
persistent collection< dbint > intColl;
// variable intColl is a collection of dbint

dbclass emp{ ... };
persistent collection< emp > empColl;
// variable empColl is a collection of emp
```

The collection template class has a destructor, which will destroy delete all the objects in the collection, Usually collections should be declared persistent; if they are not persistent, the objects in the collection may be unintentionally destroyed when the collection variable goes out of scope.

A collection instantiated with a specific type `T` may contain objects of type `T`. If `T` is a class, the collection may also contain objects of any (sub)type such that `T` is a public base class (directly or indirectly).

A collection of dbints should contain ONLY dbints, and similarly for other fundamental types except dbvoid. The gnu E implementation does not enforce this restriction, however.

A collection of dbvoid may contain any kind of object.

e1.2 Creating Objects in a Collection

The "placement" overloaded for new is used for creation of objects within collections. The definition of the approriate overloading of new is in the standard include file <E/collection.h>

e1.3 Destroying Objects

The `delete` operator is used to delete objects from a collection (§7.2.4).

e1.4 Scanning Collections

A class created from the collection template has the following member functions:

```
T * get_first();
T * get_last();
T * get_next( T * );
T * get_prev( T * );
```

In addition, to facilitate scanning colletions, a collection_scan template class is provided allong with the collection template. This template class 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 reference to a collection<T> which indicates the collection to be scanned. Subsequent calls to the member function next of the collection_scan<T> object will return pointers to successive elements contained in that collection, until all obects in the collection<T> have been scanned. After a pointer to each element has been returned, the collection_scan next member function will return the null pointer.

The following example processes all of the people in Madison:

```
collection<dbint> C;
int sum = 0;
{
  collection_scan<dbint> Cscan<C>;
  dbint *p; dbint sum = 0;
  while ( p = Cscan.next())
      sum += *p;
}
```

Note that if the collection contains subtypes of `T`, the scan still gives back a `T` pointer. (This is the reason for the the public-base-type restriction, above.)

e1.5 Deleting Collections

If a collection is destroyed (e.g. by going out of scope) any objects that it contains are also destroyed. If the objects' type has a destructor, then that destructor is invoked on each object before the collection is destroyed.

This is not yet implemented; destroying a collection destroys its objects without calling destructors. If destructor calls are necessary, the program must destroy each object explicitly (and individually) before destroying the collection.

```
{ collection_scan<emp> empScan(empRelation);
  emp * emp_pt;
  while (emp_pt = empScan.next())
      delete emp_pt;
}
```

e2. Iterator function macro package.

Previous versions of E supported Clu - style iterators as a language construct. Iterators are no longer supported with this version of E. Much of their functionality is available in a similar form through use of the macro packaged described below. This package is enabled via the preprocessor directive

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

e2.1 Iterator Function Definitions

The macro ITER_DECL is used to declare iterator functions. It takes 2 arguments, first, the type of the values to be yielded from the iterator function, and the second is the name and argument list for the declaration:

```
ITER_DECL(int,  stream_of_ints( args ));
```

Iterator function definitions are bracketed by the ITER_DEF and END_ITER_DEF macros; like ITER_DECL, the ITER_DEF macro has two arguments, the type to be yielded by the iterator and the function name and arguement list for the function declaration.

```
ITER_DEF(int,  stream_of_ints( int arg  ))
{
  for (int i = 0; i<arg; i++)
    YIELD(i);
} END_ITER_DEF
```

Within an iterator definition, the use of `YIELD` is legal (and expected!). The use of the normal C++ return statement is illegal. Alternatives to the return statement are described under RETURN below.

Like functions, iterators have yield-by-value semantics. Unlike functions, it is perfectly reasonable for an iterator to yield a reference to a local variable. However, clients must not attempt to use such an address after the iterator has terminated.

Iterators may themselves be clients of other iterators, and in general, may be recursive. The following example shows a (dumb) recursive iterator which yields the sequence of integers from 0 to the specified maximum:

```
ITER_DEF(int, sequence( int max ))
{
    if(max > 0)
      ITERATE(int, val, sequence( max - 1 ),
          YIELD( val));

    YIELD( max);

} END_ITER_DEF
```

e2.1.1 YIELD and YIELD_BREAK macros

There are 2 variants of the YIELD macro that may be used to return values from an iterator function:

```
YIELD (expression );
YIELD_BREAK (expression, break_response );
```

where *break_response* is an expression or block statement. `YIELD` and `YIELD_BREAK` may only be used within iterator functions. These macros returns the value of *expression* and suspends current local state for future resumption; control is returned to the invoking iterate loop.

Normally, the iterator will resume at the first statement after the `YIELD` or `YIELD_BREAK`. However, if the client loop executes a `BREAK`, and the last value yielded by the iterator was through a `YIELD_BREAK` macro, the iterator will resume at the <break_response> statement associated with that `YIELD_BREAK`; this gives the iterator a chance to "clean up" before terminating. If the client loop executes a `BREAK` (§e2.2.1) to terminate the iterator function and the last yield point was via a YIELD macro, the implied <break_responce> action is an IRETURN, as described below.

e2.1.2 RETURN and IRETURN macros

A return statement should not be executed within an iterator function. The effects of executing a return statement are undefined. A RETURN macro, which takes 1 parameter, may be used to terminate the iterator execution while yielding a final value, and an IRETURN macro, which takes no parameters, may be used to terminate the iterator function without returning a final value. Executing the RETURN or IRETURN macros terminate the iterator. It is possible, though not generally useful, to return a value. If the client loop is not also terminated (i.e. because there are other iterators still active), then the value returned will be assigned to the associated loop variable. In addition, applying `EMPTY( )` (§e2.2.4) to the loop variable will return true.

Iterator functions as described above cannot currently be declared as inline member functions of classes. Attempts

to use the ITER_DEF macro within a class definition will result in syntax errors due to problems with the GNU E parser.

e2.2 ITERATE macros

There are four variants of the `ITERATE` macro, which will activate from 1 to four iterator functions:

```
ITERATE( type_var_call , statement_arg)
ITERATE2( t_v_c , t_v_c , statement_arg)
ITERATE3( t_v_c , t_v_c , t_v_c , statement_arg)
ITERATE4( t_v_c , t_v_c , t_v_c, t_v_c , statement_arg)

t_v_c ::= type_name , identifier , iterator_invocaton
iterator_invocation ::= primary ( arg_list )
statement_arg ::= block
```

The syntactic group *t_v_c* above is a triple which specifies the type of value the iterator is expected to yield, the name of the local variable which will contain the yielded values, and the call used to invoke the iterator function. The set of macros allows activation of one to four iterator functions in parallel; the total number of of arguments is thus 4 for `ITERATE`, 7 for `ITERATE2`, 10 for `ITERATE3`, and 13 for `ITERATE4`, When control enters an iterate loop, each of the iterators in the list is invoked in some order (which is undefined), and each of the iterator variables is initialized with the results of the first **yield**. The loop terminates if (and when) control reaches the top of the loop and all iterators have terminated. If control reaches the bottom of the loop, and no `ADVANCE` operations (§e2.2.3) have been performed during that iteration, then a `advance` is performed on ALL iterators. The scope of iterator variables is that of the loop. However, the name of the iterator variable is also used to form program labels used in the expansion of the `ITERATE` constructs; therefore, the names of iterator variables should not be reused within other `ITERATE` macros in the same function.

```
ITERATE2( int x, iter1(...), float, y, iter2(...), {
    printf("%d, %d", x, y);
})
```

is equivalent to

```
ITERATE2( int, x, iter1(...), float, y, iter2(...), {
    printf("%d, %d", x, y);
    ADVANCE2( x, y);
})
```

Note that within the body of the iterate loop, it is possible to change objects which were passed to an iterator. If these objects were passed by reference, then the changes will be visible inside the iterator function.

• It is not currently possible to cast the return type of an iterator. E.g., if iterator `f` yields `void*`, then the following is illegal:

```
ITERATE( int*, ip , (int*) f(), { ... })
```

Of course, the value of the iterator variable may be cast within the body of the loop.

e2.2.1 BREAK macros

A set of `BREAK` macros is provided to allow termination of active iterator functions from within iterate loops. A break within an iterate loop (§e2.2) has the form:

```
BREAK(id)
BREAK2(id1,id2)
BREAK3(id1,id2,id3)
```

A C++ `break` statement _list is equivalent to a break in which the *id_list* names all the iterator variables in the smallest lexically enclosing iterate loop.

A BREAK macro resumes the iterator associated with each of the variables given as arguments to the macro; if the last yield point in the iterator function was defined via YIELD_BREAK, the point of resumption, however, is not the normal resume-point, but is the *break_response* action of the last yield point used in the corresponding iterator activation.

The last action of the break is an implicit continue (§9.9). Note that BREAK and its variants do not necessarily exit the loop, since some iterators may still be active.

e2.2.2 Continue Statement within ITERATE loops

The loop continuation point of the iterate loop is:

```
ITERATE( ... ,{
    ...
contin:
})
```

That is, if a continue is executed within an iterate loop, control passes immediately to the end of the block statement given as the body of the loop.

e2.2.3 ADVANCE macro

The ADVANCE macros can advance the iterator functions associated with from 1 to 4 iterator activations. The ADVANCE macros may only be used within the scope of an ITERATE macro. There are 4 variants, depending on the number of arguments:

```
ADVANCE(iter_var)
ADVANCE2(iter_var1,iter_var2)
ADVANCE3(iter_var1,iter_var2,iter_var3)
ADVANCE4(iter_var1,iter_var2,iter_var3,iter_var4)
```

Advance is valid only inside the scope of an iterate loop, defined via one of the ITERATE macros, and the names given as arguments must be iterator variables in a lexically enclosing iterate loop.

The result of this macro is that for each of the variables given as arguments, its associated iterator function is resumed (normally) to yield the next value. The order in which iterators are resumed is not defined. If the iterator associated with variable x has terminated, then ADVANCE(x) has no effect.

If an ADVANCE macro is executed during a given iteration of an iterate loop, then no default resumptions are carried out (i.e., the iterators that are advanced are the *only* ones to advance that iteration).

e2.2.4 EMPTY macro

Within an iterate loop, the user can test whether the iterator associated with a particular variable has terminated. The EMPTY macro is used for this test. For example, the following iterator merges two sorted streams of integers. (Assume the iterator "ordered" is defined which yields the elements of an integer set in sorted order.)

```
ITER_DEF( int,  merge( intset s1, intset s2 ))
{
      ITERATE2( int, i1, ordered( s1 ), int, i2, ordered( s2 ), {
            if( EMPTY(i1) )
                  YIELD( i2);
            else if( EMPTY(i2) )
                  YIELD (i1);
            else if( i1 < i2 ){
                  YIELD (i1);
                  ADVANCE (i1);
            } else {
                  YIELD (i2);
                  ADVANCE (i2);
            }
      }) // iterate
} // merge
END_ITER_DEF
```

e3 db String functions

The E runtime library contains a set of routines that are analagous to the C library string and byte string manipulation routines. These routines can manipulate either persistent objects or transient memory that is pointed to by db pointers. The declarations for these routines can be enabled by the preprocessor directive

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

In addition to imitating the functionality of the 'usual' string routines (e.g. strcmp, bcopy, etc.), there are also routines to do byte insertion, deletion, reading, and writing, etc. Those routines that are analogues to the C library functions ones have identical names and are resolved base on overloading of the argurment types.

The declarations for these routines are listed below. These routines are all compiled with C++ linkage.

```
dbchar *    strcat(dbchar * s, const dbchar * append);
dbchar *    strncat(dbchar * s, const dbchar * append, int count);
int         strcmp(const dbchar * s1, const dbchar * s2);
int         strncmp(const dbchar * s1, const dbchar * s2, int count);
int         strcasecmp(const dbchar * s1, const dbchar * s2);
int         strncasecmp(const dbchar * s1, const dbchar * s2, int count);
dbchar *    strcpy(dbchar * to, const dbchar * from);
dbchar *    strncpy(dbchar * to, const dbchar * from, int count);
int         strlen(const dbchar * s);
dbchar *    index(const dbchar * s, int c);
dbchar *    rindex(const dbchar * s, int c);

dbchar *    strins( dbchar * s, const dbchar * insert, int length = -1 );
dbchar *    strdel( dbchar * s, int length  = -1 );

int         strload( dbchar * s, FILE * f, int n = -1 );
int         strdump( const dbchar * s, FILE * f, int n = -1 );
int   strread( dbchar * s );
int   strprint( const dbchar * s );

void        bcopy(const dbvoid *, dbvoid *, int );
int         bcmp( const dbvoid *, const dbvoid *, int );
void        bzero( dbvoid *, int, int = 0 );
int         bload( dbvoid *, FILE *, int );
int         bdump( const dbvoid *, FILE *, int );
void        bins( dbvoid*, const dbvoid *, int );
void        bdel( dbvoid *, int );
void        bpos( const dbvoid *, OID&, int&, int& );
```

e3.1 String Routines

The following describes the routines that operate on null terminated strings. Note that, unlike their conventional counterparts, these routines must deal with the possibility of reaching the physical end of a Storage Manager object. We will denote this position as **eob**. For the purposes of terminating a string, **eob** is equivalent to a null character. Note, however, that **eob** is *not* a character, and so the routines **index** and **rindex** cannot be used to search for it. (They *can* be used to search for the null character.) Those routines that might possibly expand the length of a string (e.g. strcat) will usually expand the physical object if **eob** is reached before the operation is completed. If expansion is not desired, one can use the routines that take an explicit length parameter (supplying the correct length, of course). As a final note, these routines (with the exception of those doing insertion and deletion) may operate on normal (in-memory) strings of characters; this is because E allows a (char*) to be cast as a (dbchar*). Such strings have no **eob**.

Strcat concatenates the string *append* to the string *s*. **Strncat** concatenates at most *count* characters. Reaching **eob** in the source string terminates the copy. Reaching **eob** in the target causes the target to be expanded. Both routines return a pointer to the null (or **eob**) terminated result.

Strcmp returns an integer value less than, equal to, or greater than zero as string *s1* is lexicographically less than, equal to, or greater than string *s2*. **Strncmp** compares at most count characters. **Strcasecmp** and **strncasecmp** are like **strcmp** and **strncmp**, respectively, except that they are case-insensitive. Comparisons reflect a conversion to lower case.

Strcpy copies the string *from* to *to*, returning a pointer to the null-terminated result. **Strncpy** copies at most *count* characters, returning a pointer to the result. That result may not be null terminated if strlen(from) > count. (However, assuming that the destination is an object in the storage manager, the result *will* be **eob** terminated.)

Strlen returns the number of non-null characters in string *s*.

Index returns a pointer to the first (leftmost) occurrence of character *c* in string *s*. **Rindex** searches for the last (rightmost) occurrence of *c*. Both routines return a null pointer if *c* does not occur in *s*. Setting *c* to the null character works.

Strins inserts characters from the string *insert* in front of the character pointed to by *s*. The *count* argument is optional, and if not given, defaults to -1. If *count* is -1, all of the characters of *insert* (up to, but not including, the trailing null) are inserted into *s*. If *count* > 0, at most that many characters are inserted. **Strdel** deletes characters, beginning with the one pointed to be *s*. The optional *count* parameter defaults to -1. If *count* is -1, then all characters up to, but not including, the trailing null are deleted. Otherwise, at most *count* characters are deleted. Neither **strins** nor **strdel** has any effect on in-memory objects.

Strload copies characters from the stream *f* to the string *s*. If *count* is -1 (the default), then characters are read from *f* until the first null, or until EOF (which is converted into a null). Otherwise, at most *count* characters are read. The target object is expanded as needed to hold the characters. **Strdump** copies characters from the string *s* to the stream *f*. If *count* is -1 (the default), then characters are copied to *f* until the null is reached. Otherwise, at most *count* characters are copied.

e3.2 Byte-string Routines

The following routines operate on strings of bytes. They do not look for null characters as the above routines do.

Bcopy copies *length* characters from *from* to *to*. Reaching **eob** in the source terminates the copy; reaching **eob** in the target expands the target. Like the conventional bcopy, the source and destination arguments are reversed from strcpy. In the current release, if the source and destination are overlapping byte ranges, the results are undefined.

Bcmp compares the first *length* bytes in *s1* to those in *s2*, returning a value less than, equal to, or greater than zero as the first range is lexicographically less than, equal to, or greater than the second.

Bzero writes *length* null bytes to *s*. If the optional parameter *force* is 0 (the default), then reaching **eob** terminates the operation; otherwise, the remaining number of nulls are appended to the object.

Bload, **bdump**, **bins**, and **bdel** all operate as their string counterparts, except that null characters are not checked for. In the case of **bload**, the EOF character is preserved (instead of being cenverted to a null).

Bpos gives the programmer low-level information on the object pointed to by *s*. The OID is returned in *oid*. If the object being pointed to is a Storage Manager object, then the current object size is returned in *size*, and the current offset of *s* is returned in *pos*. (In this way, the programmer can determine, for example, how much room there is between the current pointer position and **eob**.) Note that bytes are numbered starting from 0, i.e. a pointer to the beginning of an object has offset 0. If the pointer refers to an in-memory object, the size returned is -1 (since there is no definable **eob** for such objects), and the actual memory address is returned in *pos*. If *s* is an invalid pointer, then one return, *size* == *pos* == -1.

REFERENCES

[Rich89b] Richardson, J., Carey, M., and Schuh, D., "The Design of the E Programming Language," Computer Sciences Technical Report, University of Wisconsin-Madison, February, 1989. (Submitted for publication.)

[exo92] *An Introduction to GNU E,* Exodus Project Document

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

[Stro86] Stroustrup, Bjarne, *The C++ Programming Language, 2nd edition,* Addison-Wesley, 1986.

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