Annex D
Linked Lists

(Informative)
The List package implements a classic list data-structure, and is analogous to the STL *(Standard Template Librar*y) List container that is popular with C++ programmers. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type.

D.1 List definitions

list —A list is a doubly linked list, where every element has a predecessor and successor. A list is a sequence that supports both forward and backward traversal, as well as amortized constant time insertion and removal of elements at the beginning, end, or middle.

container—A container is a collection of data of the same type. Containers are objects that contain and manage other data. Containers provide an associated *iterator* that allows access to the contained data.

iterator—Iterators are objects that represent positions of elements in a container. They play a role similar to that of an array subscript, and allow users to access a particular element, and to traverse through the container.

D.2 List declaration

The List package supports lists of any arbitrary predefined type, such as integer, string, or class object.

To declare a specific list, users must first include the generic List class declaration from the standard include area and then declare the specialized list type:

```
'include
"List.vh"
...
List#(type) dl;          // dl is a List of 'type' elements
```

Editor's Note: Standard include may use different syntax or file extension

D.2.1 Declaring list variables

List variables are declared by providing a specialization of the generic list class:

```
List#(integer) il;                  // Object il is a list of integer
typedef List#(Packet) PList;        // Class Plist is a list of Packet objects
```

The List specialization declares a list of the indicated type. The type used in the list declaration determines the type of the data stored in the list elements.

D.2.2 Declaring list iterators

List iterators are declared by providing a specialization of the generic List_Iterator class:

```
List_Iterator#(string) s;          // Object s is a list-of-string iterator
List_Iterator#(Packet) p, q;       // p and q are iterators to a list-of-Packet
```

D.3 Linked List class prototypes

The following class prototypes describe the generic List and List_Iterator classes. Only the public interface is included here.

D.3.1 List_Iterator class prototype

```
class List_Iterator#(parameter type T);
        extern function void next();
        extern function void prev();
        extern function int neq( List_Iterator#(T) iter );
        extern function int eq( List_Iterator#(T) iter );
        extern function T data();
endclass
```

D.3.2 List class prototype

```
class List#(parameter type T);
        extern function new();
        extern function int size();
        extern function int empty();
        extern function void push_front( T value );
        extern function void push_back( T value );
        extern function T front();
        extern function T back();
        extern function void pop_front();
        extern function void pop_back();
        extern function List_Iterator#(T) start();
        extern function List_Iterator#(T) finish();
        extern function void insert( List_Iterator#(T) position, T value );
        extern function void insert_range( List_Iterator#(T) position, first, last );
        extern function void erase( List_Iterator#(T) position );
        extern function void erase_range( List_Iterator#(T) first, last );
        extern function void assign( List_Iterator#(T) first, last );
        extern function void swap( List#(T) lst );
        extern function void clear();
        extern function void purge();
endclass
```

D.4 List_Iterator methods

The List_Iterator class provides methods to iterate over the elements of lists. These methods are described below.

D.4.1 next()

function void next();

next changes the iterator so that it refers to the next element in the list.

D.4.2 prev()

function void prev();

prev changes the iterator so that it refers to the previous element in the list:

D.4.3 eq()

function int eq(List_Iterator#(T) iter);

eq compares two iterators, and returns 1 if both iterators refer to the same list element. Otherwise, it returns 0.

```
if( i1.eq(i2) ) $display( "both iterators refer to the same element" );
```

D.4.4 neq()

function int neq(List_Iterator#(T) iter);

neq is the Negation of **eq()**; it compares two iterators, and returns 0 if both iterators refer to the same list element. Otherwise, it returns 1.

D.4.5 data()

function T data();

data returns the data stored in the element at the given iterator location.

D.5 List methods

The List class provides methods to query the size of the list, obtain iterators to the head or tail of the list, retrieve the data stored in the list, and methods to add, remove, and reorder the elements of the list.

D.5.1 size()

function int size();

size returns the number of elements stored in the list.

```
while( list1.size > 0 ) begin    // loop while there are still elements in the list.
        ...
end
```

D.5.2 empty()

function int empty();

empty returns 1 if the number elements stored in the list is zero, 0 otherwise.

```
if( list1.empty )
        $display( "list is empty" );
```

D.5.3 push_front()

function void push_front(T value);

push_front Inserts the specified value at the front of the list.

```
List#(int) numbers;
numbers.push_front(10);
numbers.push_front(5);                    // numbers contains { 5 , 10 }
```

D.5.4 push_back()

function void push_back(T value);

push_back inserts the specified value at the end of the list.

```
List#(string) names;
names.push_back("Donald");
names.push_back("Mickey");          // names contains { "Donald", "Mickey" }
```

D.5.5 front()

function T front();

front returns the data stored in the first element of the list (valid only if the list is not empty).

D.5.6 back()

function T back();

back returns the data stored in the last element of the list (valid only if the list is not empty).

```
List#(int) numbers;
numbers.push_front(3);
numbers.push_front(2);
numbers.push_front(1);
```

```
$display( numbers.front, numbers.back );      // displays 1 3
```

D.5.7 pop_front()

function void pop_front();

pop_front removes the first element of the list. If the list is empty, this method is illegal and may generate an error.

D.5.8 pop_back()

function void pop_back();

pop_back removes the last element of the list. If the list is empty, this method is illegal and may generate an error.

```
while( lp.size > 1 ) begin // remove all but the center element from an odd-sized list lp
        lp.pop_front();
        lp.pop_back();
end
```

D.5.9 start()

function List_Iterator#(T) start();

start returns an iterator to the position of the first element in the list.

D.5.10 finish()

function List_Iterator#(T) finish();

finish returns an iterator to a position just past the last element in the list. The last element in the last can be accessed using finish.prev.

```
List#(int) lst;                    // display contents of list lst in position order
for( List_Iterator#(int) p = lst.start; p.neq(lst.finish); p.next )
        $display( p.data );
```

D.5.11 insert()

function void insert(List_Iterator#(T) position, T value);

insert inserts the given data (value) into the list at the position specified by the iterator (before the element, if any, that was previously at the iterator's position). If the iterator is not a valid position within the list then this operation is illegal and may generate an error.

```
function void add_sort( List#(byte) L, byte value );
    for( List_Iterator#(byte) p = L.start; p.neq(L.finish) ; p.next )
        if( p.data > value ) begin
```

```
            lst.insert( p, value );     // Add element to sorted list (ascending order)
            return;
        end
    endfunction
```

D.5.12 insert_range()

```
function void insert range( List Iterator#(T) position, first, last );
```

insert_range inserts the elements contained in the list range specified by the iterators first and last at the specified list position (before the element, if any, that was previously at the position iterator). All the elements from first up to, but not including, last are inserted into the list. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list. The range iterators can specify a range either in another list or in the same list as being inserted.

If the position iterator is not a valid position within the list, or if the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and may generate an error.

D.5.13 erase()

```
function void erase( List Iterator#(T) position );
```

erase removes form the list the element at the specified position. After **erase()** returns, the position iterator becomes invalid.

```
list1.erase( list1.start );         // same as pop_front
```

If the position iterator is not a valid position within the list, this operation is illegal and may generate an error.

D.5.14 erase_range()

function void erase_range(List_Iterator#(T) first, last);

erase removes from a list the range of elements specified by the first and last iterators. This operation removes elements from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list1.erase_range( list1.start, list1.finish ); // Remove all elements from list1
```

If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and may generate an error.

D.5.15 assign()

function void assign(List_Iterator#(T) first, last);

assign assigns to the list object the elements that lie in the range specified by the first and last iterators. After this method returns, the modified list will have a size equal to the range specified by first and last. This method copies the data from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list2.assign( list1.start, list2.finish );        // list2 is a copy of list1
```

If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and may generate an error.

D.5.16 swap()

function void swap(List#(T) lst);

swap exchanges the contents of two equal-size lists.

```
list1.swap( list2 );          // swap the contents of list1 to list2 and vice-versa
```

Swapping a list with itself has no effect. If the lists are of different sizes, this method may issue a warning.

D.25.17 clear()

function void clear();

clear removes all the elements from a list, but not the list itself (i.e., the list header itself).

```
list1.clear();          // list1 becomes empty
```

D.5.18 purge()

function void purge();

purge removes all the list elements (as in clear) and the list itself. This accomplishes the same effect as assigning **null** to the list. A purged list must be re-created using **new** before it can be used again.

```
list1.purge();          // same as list1 = null
```

Editors Note: If the class scope operator "::" is supported then the iterator can be declared within the list itself, as in C++, with no need for another global List_Iterator class. List#(*type*)::iterator