# D T L

**Delphi Template Library.**

## Contents



## Overview

Delphi Template Library (DTL) is a collection of classes and functions, which are written in the extended Delphi language for using of the DEEX preprocessor. The DTL provides several generic containers and functions to utilize and manipulate these containers. Features of the Standard Library are declared within {$I 'dtl.inc'}.

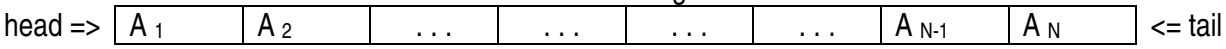
## 1. Standard containers

**Description**

Containers in DTL are classes and can exist in two version: standard (derived from smart objects) and unsafe (derived from TObject).

DTL containers can be extended with user-declared methods and properties. User-declared methods must be defined after usage of template and before final "end" in type declaration.

Containers have head and tail that used for iteration through containers.



## 1.1. Stack

**Description**
Stack is data structure based on the principle of “Last In, First Out” (LIFO).
DTL implementation of Stack allows forward and backward list iteration through its elements.
DTL implementation of Stack is adapter of Deque.

**Template**
DTL_DECLARE STACK OF `elemtype AS `containerclassname

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the stack. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the stack of `elemtype. | <identifier> |

**Example**

```
type
  DTL_DECLARE STACK OF Char AS TStackOfChar END;

procedure Bar;
var stack : TStackOfChar;
begin
   stack := TStackOfChar.Create;
   stack.push('a');
   stack.push(['1', '2', '3']);
   while not stack.is_empty do writeln(stack.pop);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the stack contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the stack. |
| function **peek** : `elemtype; | Returns a value of the element at the head of the stack.<br>Precondition: is_empty is false. |
| procedure **push** (const value : `elemtype); | Inserts value(s) at the head of the stack.<br>Postconditions: count will be incremented by number of elements, and peek will be equal to last inserted element. |
| procedure **push** (const values : array of `elemtype); | |
| function **pop** : `elemtype; | Removes the element at the head of the stack, and returns its value.<br>Precondition: is_empty is false.<br>Postcondition: count will be decremented by 1. |
| procedure **insert** (const value : `elemtype); | Default inserter for stack. This is the same as push. |

## 1.2. Queue

**Description**
Queue is data structure based on the principle of “First In, First Out” (FIFO).
DTL implementation of Queue allows forward and backward list iteration through its elements.
DTL implementation of Queue is adapter of Deque.

**Template**
DTL_DECLARE QUEUE OF `elemtype AS `containerclassname

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the queue. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the queue of `elemtype. | <identifier> |

**Example**

```
type
  DTL_DECLARE QUEUE OF Char AS TQueueOfChar END;

procedure Bar;
var queue : TQueueOfChar;
begin
   queue := TQueueOfChar.Create;
   queue.push('a');
   queue.push(['1', '2', '3']);
   while not queue.is_empty do writeln(queue.pop);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the queue contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the queue. |
| function **peek** : `elemtype; | Returns a value of the element at the head of the queue.<br>Precondition: is_empty is false. |
| procedure **push** (const value : `elemtype); | Inserts value(s) at the tail of the queue.<br>Postconditions: count will be incremented by number of elements |
| procedure **push** (const values : array of `elemtype); | |
| function **pop** : `elemtype; | Removes the element at the head of the queue, and returns its value.<br>Precondition: is_empty is false.<br>Postcondition: count will be decremented by 1. |
| procedure **insert** (const value : `elemtype); | Default inserter for queue. This is the same as push. |

## 1.3. Priority Queue

**Description**

A Priority queue is a data structure that supporting the following three operations:

- insert an element to the queue with an associated priority [uses O(log n) time]
- remove the element from the queue that has the highest priority, and return it [uses O(log n) time]
- peek at the element with highest priority without removing it [uses constant O(1) time]

DTL implementation of Priority queue allows forward and backward list iteration through its elements.
DTL implementation of Priority queue is adapter of Multiset.

**Template**

DTL_DECLARE PRIORITYQUEUE OF `elemtype AS `containerclassname ORDER_BY `comp_fn

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the priority queue. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the priority queue of `elemtype. | <identifier> |
| `comp_fn | Name of global function used to determine whether one element is smaller than another element. Function must take two arguments and return integer value.<br>= -1, if x < y.<br>= 0, if x = y.<br>= 1, if x > y. | <identifier> |

**Example**

```
# CHAR_COMP `a `b => _BEGIN_ sign(byte(`a) - byte(`b)) _END_;

type
  DTL_DECLARE PRIORITYQUEUE OF Char AS TPrQueueOfChar ORDER_BY CHAR_COMP END;

procedure Bar;
var pqueue : TPrQueueOfChar;
begin
   pqueue := TPrQueueOfChar.Create;
   pqueue.push(['1', '2', '3']);
   pqueue.push('a');
   pqueue.push(['1', '2', '3']);
   while not pqueue.is_empty do writeln(pqueue.pop);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : Boolean; | Returns true if the priority queue contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the priority queue. |
| function **peek** : `elemtype; | Returns a value of the element at the head of the priority queue that has the highest priority.<br>Precondition: is_empty is false. |
| procedure **push** (const value : `elemtype); | Inserts value(s) to the queue with an associated priority.<br>Postconditions: count will be incremented by number of elements |

| | |
|---|---|
| procedure **push** (const values : array of `elemtype); | |
| function **pop** : `elemtype; | Removes the element of the queue that has the highest priority, and returns its value.<br>Precondition: is_empty is false.<br>Postcondition: count will be decremented by 1. |
| procedure **insert** (const value : `elemtype); | Default inserter for priority queue. This is the same as push. |

## 1.4. Deque

**Description**
Deque is data structure for which elements can be added to or removed from the head or tail.
DTL implementation of Deque allows forward and backward list iteration through its elements.

**Template**
DTL_DECLARE DEQUE OF `elemtype AS `containerclassname

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the deque. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the deque of `elemtype. | <identifier> |

**Example**

```
type
  DTL_DECLARE DEQUE OF Char AS TDequeOfChar END;

procedure Bar;
var deque : TDequeOfChar;
begin
   deque := TDequeOfChar.Create;
   deque.push_head('a');
   deque.push_tail(['1', '2', '3']);
   while not deque.is_empty do writeln(deque.pop_tail);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the deque contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the deque. |
| function **peek_head** : `elemtype; | Returns a value of the element at the head of the deque.<br>Precondition: is_empty is false. |
| function **peek_tail** : `elemtype; | Returns a value of the element at the tail of the deque.<br>Precondition: is_empty is false. |
| procedure **push_head** (const value : `elemtype); | Inserts value(s) at the head of the deque.<br>Postconditions: count will be incremented by number of elements, and head will be equal to last inserted element. |
| procedure **push_head** (const values : array of `elemtype); | |
| procedure **push_tail** (const value : `elemtype); | Inserts value(s) at the tail of the deque.<br>Postconditions: count will be incremented by number of elements, and tail will be equal to last inserted element. |
| procedure **push_ tail** (const values : array of `elemtype); | |
| function **pop_head** : `elemtype; | Removes the element at the head of the deque.<br>Precondition: is_empty is false.<br>Postcondition: count will be decremented by 1. |
| function **pop_tail** : `elemtype; | Removes the element at the tail of the deque.<br>Precondition: is_empty is false.<br>Postcondition: count will be decremented by 1. |
| procedure **insert** (const value : `elemtype); | Default inserter for deque. This is the same as push_tail. |

## 1.5. Vector

**Description**
Vector or dynamic array is data structure for which elements can be added to or removed from the tail and update elements by their index.
DTL implementation of Vector allows indexed iteration through its elements.

**Template**
DTL_DECLARE VECTOR OF `elemtype AS `containerclassname

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the vector. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the vector of `elemtype. | <identifier> |

**Example**

```
type
  DTL_DECLARE VECTOR OF Char AS TVectorOfChar END;

procedure Bar;
var vector : TVectorOfChar;
begin
   vector := TVectorOfChar.Create;
   vector.insert('a');
   vector.insert(['1', '2', '3']);
   vector[1] := 'b';
   for I := 0 to vector.Count - 1 do writeln(vector[i]);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the vector contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the vector. |
| property **Items**[index : integer]: `elemtype; | Get and set a value of the element at the position pointed by index. Precondition: 0 <= index < count. |
| procedure **insert** (const value : `elemtype); | Inserts value(s) at the tail of the deque. Postconditions: count will be incremented by number of elements, and head will be equal to last inserted element. |
| procedure **insert** (const values : array of `elemtype); | |
| function **remove** : `elemtype; | Returns a value of the element at the tail of the vector. Precondition: is_empty is false. |

## 1.6. Array

**Description**
Array is data structure with fixed number of elements.
DTL implementation of Array allows indexed iteration through its elements.

**Template**
DTL_DECLARE ARRAY OF `elemtype `size AS `containerclassname

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the array. | Any ordinal or declared type |
| `size | The number of elements stored in the array. | Integer or integer constant |
| `containerclassname | The classname of a new container that will implement the array of `elemtype. | <identifier> |

**Example**

```
type
  DTL_DECLARE ARRAY OF Char 10 AS TArrayOfChar END;

procedure Bar;
var arrayc : TArrayOfChar;
begin
   arrayc := TArrayOfChar.Create;
   arrayc[1] := 'b';
   for I := 0 to arrayc.Count - 1 do writeln(arrayc[i]);
end;
```

**Members**

| Member | Description |
|---|---|
| function **count** : integer; | Returns fixed size of the array. |
| property **Items**[index : integer]: `elemtype; | Get and set a value of the element at the position pointed by index. Precondition: 0 <= index < count. |

## 1.7. Set

**Description**
Set is data structure based that stores unique objects.
DTL implementation of Set allows forward and backward list iteration through its elements.

**Template**
DTL_DECLARE SET OF `elemtype AS `containerclassname ORDER_BY `comp_fn

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the set. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the set of `elemtype. | <identifier> |
| `comp_fn | Name of global function used to determine whether one element is smaller than another element. Function must take two arguments and return integer value.<br>= -1, if x < y.<br>= 0, if x = y.<br>= 1, if x > y. | <identifier> |

**Example**

```
# CHAR_COMP `a `b => _BEGIN_ sign(byte(`a) - byte(`b)) _END_;

type
  DTL_DECLARE SET OF Char AS TSetOfChar ORDER_BY CHAR_COMP END;

procedure Bar;
var setc : TSetOfChar;
begin
   setc := TSetOfChar.Create;
   setc.insert('a');
   setc.insert(['1', '2', '3']);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the set contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the set. |
| procedure **insert** (const value : `elemtype); | Inserts value(s) into the set.<br>Postconditions: count will be incremented by number of elements that have been inserted |
| procedure **insert** (const values : array of `elemtype); | |
| procedure **remove** (const value : `elemtype); | Removes value(s) from the set.<br>Postconditions: count will be decremented by number of elements that have been removed |
| procedure **remove** (const values : array of `elemtype); | |

## 1.9. Multiset

**Description**
Multiset differs from a set in that each member has a multiplicity, which is a natural number indicating how many times it is a member in the multiset.
DTL implementation of Multiset allows forward and backward list iteration through its elements.

**Template**
DTL_DECLARE MULTISET OF `elemtype AS `containerclassname ORDER_BY `comp_fn

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `elemtype | The type of object stored in the multiset. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the multiset of `elemtype. | <identifier> |
| `comp_fn | Name of global function used to determine whether one element is smaller than another element. Function must take two arguments and return integer value.<br>= -1, if x < y.<br>= 0, if x = y.<br>= 1, if x > y. | <identifier> |

**Example**

```
# CHAR_COMP `a `b => _BEGIN_ sign(byte(`a) - byte(`b)) _END_;

type
  DTL_DECLARE MULTISET OF Char AS TMSetOfChar ORDER_BY CHAR_COMP END;

procedure Bar;
var setc : TMSetOfChar;
begin
   setc := TMSetOfChar.Create;
   setc.insert('a');
   setc.insert(['1', '2', '3']);
   setc.insert(['1', '3']);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the multiset contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the multiset. |
| function **count_of**(const value : `elemtype) : integer; | Returns the number of element copies contained in the multiset. |
| procedure **insert** (const value : `elemtype); | Inserts value(s) into the multiset.<br>Postconditions: count will be incremented by number of elements that have been inserted |
| procedure **insert** (const values : array of `elemtype); | |
| procedure **remove** (const value : `elemtype); | Removes value(s) from the multiset.<br>Postconditions: count will be decremented by number of elements that have been removed |
| procedure **remove** (const values : array of `elemtype); | |

## 1.9. Map

**Description**
Map (Associative array) is data type composed of a collection of keys and a collection of values, where each key is associated with one value.
DTL implementation of Map allows forward and backward list iteration through its elements.

**Template**
DTL_DECLARE MAP OF `keytype `valuetype AS `containerclassname ORDER_BY `comp_fn

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `keytype | The type of key stored in the map. | Any ordinal or declared type |
| `valuetype | The type of value stored in the map. | Any ordinal or declared type |
| `containerclassname | The classname of a new container that will implement the map of `elemtype. | <identifier> |
| `comp_fn | Name of global function used to determine whether one element is smaller than another element. Function must take two arguments and return integer value.<br>= -1, if x < y.<br>= 0, if x = y.<br>= 1, if x > y. | <identifier> |

**Example**

```
# CHAR_COMP `a `b => _BEGIN_ sign(byte(`a) - byte(`b)) _END_;

type
  DTL_DECLARE MAP OF Char Integer AS TMapOfCharInt ORDER_BY CHAR_COMP END;

procedure Bar;
var mapci : TMapOfCharInt;
begin
   mapci := TMapOfCharInt.Create;
   mapci.insert('a', 1);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the map contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the map. |
| function **is_exists** (const key : `keytype); | Returns true if the map contains key. |
| procedure **insert** (const key : `keytype; const value : `valuetype); | Inserts (key, value) pair into the map. |
| procedure **remove** (const key : `keytype); | Removes value(s) from the map by key |
| property **Values**[const key : `keytype]: `valuetype; **default;** | Get and set a value of the element by key. |

## 1.10. Multimap

**Description**
Multimap is generalization of a map in which more than one value may be associated with and returned for a given key.
DTL implementation of Multimap allows forward and backward list iteration through its elements.

**Template**
DTL_DECLARE MULTIMAP OF `keytype `valuetype IN `valuecontainer AS `containerclassname ORDER_BY `comp_fn

**Template parameters**

| Parameter | Description | Type of parameter |
|---|---|---|
| `keytype | The type of key stored in the multimap. | Any ordinal or declared type |
| `valuetype | The type of values stored in the multimap. | Any ordinal or declared type |
| `valuecontainer | The type of container for storing values associated with key. | Declared type of DTL container |
| `containerclassname | The classname of a new container that will implement the map of `elemtype. | <identifier> |
| `comp_fn | Name of global function used to determine whether one element is smaller than another element. Function must take two arguments and return integer value.<br>= -1, if x < y.<br>= 0, if x = y.<br>= 1, if x > y. | <identifier> |

**Example**

```
# CHAR_COMP `a `b => _BEGIN_ sign(byte(`a) - byte(`b)) _END_;

type
  DTL_DECLARE MULTIMAP OF Char Integer AS TMMapOfCharInt ORDER_BY CHAR_COMP
  END;

procedure Bar;
var mmapci : TMMapOfCharInt;
begin
   mmapci := TMMapOfCharInt.Create;
   mmapci.insert('a', 1);
   mmapci.insert('b', [2,3,4]);
end;
```

**Members**

| Member | Description |
|---|---|
| function **is_empty** : boolean; | Returns true if the set contains no elements, and false otherwise. |
| function **count** : integer; | Returns the number of elements contained in the set. |
| function **is_exists** (const key : `keytype); | Returns true if the map contains key. |
| function **count_of**(const key : `keytype) : integer; | Returns the number of element copies contained in the multimap with key. |
| procedure **insert** (const key : `keytype; const value : `valuetype); | Inserts (key, value) pair(s) into the multimap. |

| | |
|---|---|
| procedure **insert** (const key : `keytype; const values : array of `valuetype); | |
| procedure **remove** (const key : `keytype); | Removes key and value(s) from the multimap by key |
| property **Items**[const key : `keytype]: `valuecontainer; **default;** | Get container of key |

# 2. Algoritms

**Description**

## 2.1. Foreach

# 3. Iterators

**Description**

## 3.1. Forward

## 3.2. Backward

## 3.3. Range