

*Oracle Berkeley DB*

*Berkeley DB  
API Reference  
for the STL C++ API*

*11g Release 2*

**ORACLE®**  

---

**BERKELEY DB**



---

## Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: <http://www.oracle.com/technology/software/products/berkeley-db/htdocs/oslicense.html>

Oracle, Berkeley DB, and Sleepycat are trademarks or registered trademarks of Oracle. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Oracle.

Other names may be trademarks of their respective owners.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: <http://forums.oracle.com/forums/forum.jspa?forumID=271>

*Published 11/8/2010*

---

---

# Table of Contents

Preface .....	xi
1. Dbstl Global Public Functions .....	1
close_db .....	3
close_all_dbs .....	4
close_db_env .....	5
close_all_db_envs .....	6
begin_txn .....	7
commit_txn .....	8
abort_txn .....	10
current_txn .....	11
set_current_txn_handle .....	12
register_db .....	13
register_db_env .....	14
open_db .....	15
open_env .....	17
alloc_mutex .....	19
lock_mutex .....	20
unlock_mutex .....	21
free_mutex .....	22
dbstl_startup .....	23
dbstl_exit .....	24
dbstl_thread_exit .....	25
operator== .....	26
set_global_dbfile_suffix_number .....	27
close_db_cursors .....	28
2. Dbstl Container Classes .....	29
3. Db_container .....	30
get_db_open_flags .....	32
get_db_set_flags .....	33
get_db_handle .....	34
get_db_env_handle .....	35
set_db_handle .....	36
set_all_flags .....	37
set_txn_begin_flags .....	38
get_txn_begin_flags .....	39
set_commit_flags .....	40
get_commit_flags .....	41
get_cursor_open_flags .....	42
set_cursor_open_flags .....	43
db_container .....	44
~db_container .....	46
4. Db_vector .....	47
begin .....	49
end .....	51
rbegin .....	52
rend .....	54

---

max_size	55
capacity	56
operator[]	57
at	58
front	60
back	61
operator==	62
operator!=	63
operator<	64
assign	65
push_front	67
pop_front	68
insert	69
erase	71
remove	72
remove_if	73
merge	74
unique	75
sort	76
reverse	77
splice	78
size	80
empty	81
db_vector	82
~db_vector	85
operator=	86
resize	87
reserve	88
push_back	89
pop_back	90
swap	91
clear	92
5. Db_map	93
db_map	95
~db_map	97
insert	98
begin	100
end	102
rbegin	103
rend	105
is_hash	106
bucket_count	107
size	108
max_size	109
empty	110
erase	111
find	113
lower_bound	115
equal_range	117

---

count	119
upper_bound	120
key_eq	122
hash_funct	123
value_comp	124
key_comp	125
operator=	126
operator[]	127
swap	128
clear	129
operator==	130
operator!=	131
6. Db_multimap	132
insert	134
erase	136
equal_range	138
equal_range_N	140
count	142
upper_bound	143
db_multimap	145
~db_multimap	147
operator=	148
swap	149
operator==	150
operator!=	151
7. Db_set	152
db_set	153
~db_set	155
insert	156
operator=	158
value_comp	159
swap	160
operator==	161
operator!=	162
8. Db_multiset	163
db_multiset	164
~db_multiset	166
insert	167
erase	170
operator=	172
swap	173
operator==	174
operator!=	175
9. Dbstl Iterator Classes	176
10. Db_base_iterator	177
refresh	178
close_cursor	179
set_bulk_buffer	180
get_bulk_bufsize	181

---

db_base_iterator .....	182
operator= .....	183
~db_base_iterator .....	184
get_bulk_retrieval .....	185
is_rmw .....	186
is_directdb_get .....	187
11. Iterator Classes for db_vector .....	188
12. Db_vector_base_iterator .....	189
db_vector_base_iterator .....	190
~db_vector_base_iterator .....	191
operator== .....	192
operator!= .....	193
operator< .....	194
operator<= .....	195
operator>= .....	196
operator> .....	197
operator++ .....	198
operator-- .....	199
operator= .....	200
operator+ .....	201
operator+= .....	202
operator- .....	203
operator-= .....	204
operator * .....	205
operator-> .....	206
operator[] .....	207
get_current_index .....	208
move_to .....	209
refresh .....	210
close_cursor .....	211
set_bulk_buffer .....	212
get_bulk_bufsize .....	213
13. Db_vector_iterator .....	214
db_vector_iterator .....	215
~db_vector_iterator .....	216
operator++ .....	217
operator-- .....	218
operator= .....	219
operator+ .....	220
operator+= .....	221
operator- .....	222
operator-= .....	224
operator * .....	225
operator-> .....	226
operator[] .....	227
refresh .....	228
14. Iterator Classes for db_map and db_multimap .....	229
15. Db_map_base_iterator .....	230
db_map_base_iterator .....	231

---

~db_map_base_iterator .....	233
operator++ .....	234
operator-- .....	235
operator== .....	236
operator!= .....	237
operator * .....	238
operator-> .....	239
refresh .....	240
close_cursor .....	241
move_to .....	242
set_bulk_buffer .....	243
get_bulk_bufsize .....	244
operator= .....	245
16. Db_map_iterator .....	246
db_map_iterator .....	247
~db_map_iterator .....	249
operator++ .....	250
operator-- .....	251
operator * .....	252
operator-> .....	253
refresh .....	254
operator= .....	255
17. Iterator Classes for db_set and db_multiset .....	256
18. Db_set_base_iterator .....	257
~db_set_base_iterator .....	258
db_set_base_iterator .....	259
operator++ .....	261
operator-- .....	262
operator * .....	263
operator-> .....	264
refresh .....	265
19. Db_set_iterator .....	266
~db_set_iterator .....	267
db_set_iterator .....	268
operator++ .....	270
operator-- .....	271
operator * .....	272
operator-> .....	273
refresh .....	274
20. Db_reverse_iterator .....	275
operator++ .....	276
operator-- .....	277
operator+ .....	278
operator- .....	279
operator+= .....	280
operator-= .....	281
operator< .....	282
operator> .....	283
operator<= .....	284

operator>=	285
db_reverse_iterator	286
operator=	287
operator[]	288
21. Dbstl Helper Classes	289
22. ElementRef and ElementHolder Wappers	290
23. ElementHolder	291
ElementHolder	292
~ElementHolder	293
operator+=	294
operator-=	295
operator *=	296
operator/=	297
operator%=	298
operator &=	299
operator =	300
operator^=	301
operator>>=	302
operator<<=	303
operator++	304
operator--	305
operator=	306
operator ptype	307
_DB_STL_value	308
_DB_STL_StoreElement	309
24. ElementRef	310
~ElementRef	311
ElementRef	312
operator=	313
_DB_STL_StoreElement	314
_DB_STL_value	315
25. DbstlDbt	316
DbstlDbt	318
~DbstlDbt	319
operator=	320
26. DbstlElemTraits	321
assign	324
eq	325
lt	326
compare	327
length	328
copy	329
find	330
move	331
to_char_type	332
to_int_type	333
eq_int_type	334
eof	335
not_eof	336

---

set_restore_function .....	337
get_restore_function .....	338
set_assign_function .....	339
get_assign_function .....	340
get_size_function .....	341
set_size_function .....	342
get_copy_function .....	343
set_copy_function .....	344
set_sequence_len_function .....	345
get_sequence_len_function .....	346
get_sequence_copy_function .....	347
set_sequence_copy_function .....	348
set_compare_function .....	349
get_compare_function .....	350
set_sequence_compare_function .....	351
get_sequence_compare_function .....	352
set_sequence_n_compare_function .....	353
get_sequence_n_compare_function .....	354
instance .....	355
-DbstlElemTraits .....	356
DbstlElemTraits .....	357
27. BulkRetrievalOption .....	358
BulkRetrievalOption .....	359
operator== .....	360
operator= .....	361
bulk_buf_size .....	362
bulk_retrieval .....	363
no_bulk_retrieval .....	364
28. ReadModifyWriteOption .....	365
operator= .....	366
operator== .....	367
read_modify_write .....	368
no_read_modify_write .....	369
29. Dbstl Exception Classes .....	370
30. DbstlException .....	371
DbstlException .....	372
operator= .....	373
~DbstlException .....	374
31. InvalidDbtException .....	375
InvalidDbtException .....	376
32. FailedAssertionException .....	377
what .....	378
FailedAssertionException .....	379
~FailedAssertionException .....	380
33. InvalidCursorException .....	381
InvalidCursorException .....	382
34. NoSuchKeyException .....	383
NoSuchKeyException .....	384
35. NotEnoughMemoryException .....	385

---

NotEnoughMemoryException .....	386
36. NotSupportedException .....	387
NotSupportedException .....	388
37. InvalidIteratorException .....	389
InvalidIteratorException .....	390
38. InvalidFunctionCall .....	391
InvalidFunctionCall .....	392
39. InvalidArgumentException .....	393
InvalidArgumentException .....	394

---

# Preface

Welcome to Berkeley DB 11g Release 2 (DB). This document describes the C++ STL API for DB library version 11.2.5.0. It is intended to describe the DB API, including all classes, methods, and functions. As such, this document is intended for C++ developers who are actively writing or maintaining applications that make use of DB databases.

## Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in monospaced font, as are method names. For example: "Db::open() is a Db class method."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB\_INSTALL* directory."

Program examples are displayed in a monospaced font on a shaded background. For example:

```
typedef struct vendor {
    char name[MAXFIELD];           // Vendor name
    char street[MAXFIELD];        // Street name and number
    char city[MAXFIELD];          // City
    char state[3];                 // Two-digit US state code
    char zipcode[6];              // US zipcode
    char phone_number[13];        // Vendor phone number
} VENDOR;
```

## Note

Finally, notes of interest are represented using a note block such as this.

## For More Information

Beyond this manual, you may also find the following sources of information useful when building a DB application:

- [Getting Started with Berkeley DB for C++](#)
- [Getting Started with Transaction Processing for C++](#)
- [Berkeley DB Getting Started with Replicated Applications for C++](#)
- [Berkeley DB C API](#) [Berkeley DB C++ API](#)
- [Berkeley DB TCL API](#)
- [Berkeley DB Installation and Build Guide](#)

- 
- [Berkeley DB Programmer's Reference Guide](#)
  - [Berkeley DB Getting Started with the SQL APIs](#)

---

# Chapter 1. Dbstl Global Public Functions

## Public Members

Member	Description
<a href="#">close_db (page 3)</a>	Close pdb regardless of reference count.
<a href="#">close_all_dbs (page 4)</a>	Close all open database handles regardless of reference count.
<a href="#">close_db_env (page 5)</a>	Close specified database environment handle regardless of reference count.
<a href="#">close_all_db_envs (page 6)</a>	Close all open database environment handles regardless of reference count.
<a href="#">begin_txn (page 7)</a>	Begin a new transaction from the specified environment "env".
<a href="#">commit_txn (page 8)</a>	Commit current transaction opened in the environment "env".
<a href="#">abort_txn (page 10)</a>	Abort current transaction of environment "env".
<a href="#">current_txn (page 11)</a>	Get current transaction of environment "env".
<a href="#">set_current_txn_handle (page 12)</a>	Set environment env's current transaction handle to be newtxn.
<a href="#">register_db (page 13)</a>	Register a Db handle "pdb1".
<a href="#">register_db_env (page 14)</a>	Register a DbEnv handle env1, this handle and handles opened in it will be closed by <a href="#">ResourceManager</a> .
<a href="#">open_db (page 15)</a>	Helper function to open a database and register it into dbstl for the calling thread.
<a href="#">open_env (page 17)</a>	Helper function to open an environment and register it into dbstl for the calling thread.
<a href="#">alloc_mutex (page 19)</a>	Allocate a Berkeley DB mutex.
<a href="#">lock_mutex (page 20)</a>	Lock a mutex, wait if it is held by another thread.
<a href="#">unlock_mutex (page 21)</a>	Unlock a mutex, and return immediately.
<a href="#">free_mutex (page 22)</a>	Free a mutex, and return immediately.
<a href="#">dbstl_startup (page 23)</a>	If there are multiple threads within a process that make use of dbstl, then this function should be called in a single thread mutual exclusively before any use of dbstl in a process; Otherwise, you don't need to call it, but are allowed to call it anyway.

---

Member	Description
<a href="#">dbstl_exit (page 24)</a>	This function releases any memory allocated in the heap by code of dbstl.
<a href="#">dbstl_thread_exit (page 25)</a>	This function closes all Berkeley DB handles in the right order, if other threads do not use them.
<a href="#">operator== (page 26)</a>	Operators to compare two Dbt objects.
<a href="#">set_global_dbfile_suffix_number (page 27)</a>	If existing random temporary database name generation mechanism is still causing name clashes, users can set this global suffix number which will be append to each temporary database file name and incremented after each append, and by default it is 0.
<a href="#">close_db_cursors (page 28)</a>	Close cursors opened in dbp1.

**Group**

None

## close\_db

### Function Details

```
void close_db(Db *pdb)
```

Close pdb regardless of reference count.

You must make sure pdb is not used by others before calling this method. You can close the underlying database of a container and assign another database with right configurations to it, if the configuration is not suitable for the container, there will be an [InvalidArgumentException](#) type of exception thrown. You can't use the container after you called close\_db and before setting another valid database handle to the container via [db\\_container::set\\_db\\_handle\(\)](#) function.

### Parameters

#### pdb

The database handle to close.

### Group: Functions to close database/environments.

Normally you don't have to close any database or environment handles, they will be closed automatically.

Though you still have the following API to close them.

### Class

[dbstl\\_global\\_functions](#)

## close\_all\_dbs

### Function Details

```
void close_all_dbs()
```

Close all open database handles regardless of reference count.

You can't use any container after you called `close_all_dbs` and before setting another valid database handle to the container via `db_container::set_db_handle()` function.

### See Also

[close\\_db\(Db \\*\)](#) ;

### Group: Functions to close database/environments.

Normally you don't have to close any database or environment handles, they will be closed automatically.

Though you still have the following API to close them.

### Class

[dbstl\\_global\\_functions](#)

## close\_db\_env

### Function Details

```
void close_db_env(DbEnv *pdbenv)
```

Close specified database environment handle regardless of reference count.

Make sure the environment is not used by any other databases.

### Parameters

#### **pdbenv**

The database environment handle to close.

### **Group: Functions to close database/environments.**

Normally you don't have to close any database or environment handles, they will be closed automatically.

Though you still have the following API to close them.

### **Class**

[dbstl\\_global\\_functions](#)

## close\_all\_db\_envs

### Function Details

```
void close_all_db_envs()
```

Close all open database environment handles regardless of reference count.

You can't use the container after you called `close_db` and before setting another valid database handle to the container via `db_container::set_db_handle()` function.

### See Also

[close\\_db\\_env\(DbEnv \\*\)](#) ;

### Group: Functions to close database/environments.

Normally you don't have to close any database or environment handles, they will be closed automatically.

Though you still have the following API to close them.

### Class

[dbstl\\_global\\_functions](#)

## begin\_txn

### Function Details

```
DbTxn* begin_txn(u_int32_t flags,  
                DbEnv *env)
```

Begin a new transaction from the specified environment "env".

This function is called by dbstl user to begin an external transaction. The "flags" parameter is passed to DbEnv::txn\_begin(). If a transaction created from the same database environment already exists and is unresolved, the new transaction is started as a child transaction of that transaction, and thus you can't specify the parent transaction.

### Parameters

#### flags

It is set to DbEnv::txn\_begin() function.

#### env

The environment to start a transaction from.

### Return Value

The newly created transaction.

### Group: Transaction control global functions.

dbstl transaction API.

You should call these API rather than DB C/C++ API to use Berkeley DB transaction features.

### Class

[dbstl\\_global\\_functions](#)

## commit\_txn

### Function Details

```
void commit_txn(DbEnv *env,  
               u_int32_t flags=0)
```

Commit current transaction opened in the environment "env".

This function is called by user to commit an external explicit transaction.

#### Parameters

##### flags

It is set to DbTxn::commit() funcion.

##### env

The environment whose current transaction is to be committed.

#### See Also

[commit\\_txn\(DbEnv \\*, DbTxn \\*, u\\_int32\\_t\)](#) ;

```
void commit_txn(DbEnv *env, DbTxn *txn,  
               u_int32_t flags=0)
```

Commit a specified transaction and all its child transactions.

#### Parameters

##### txn

The transaction to commit, can be a parent transaction of a nested transaction group, all un-aborted child transactions of it will be committed.

##### flags

It is passed to each DbTxn::commit() call.

##### env

The environment where txn is started from.

#### See Also

[commit\\_txn\(DbEnv \\*, u\\_int32\\_t\)](#) ;

**Group: Transaction control global functions.**

dbstl transaction API.

You should call these API rather than DB C/C++ API to use Berkeley DB transaction features.

**Class**

[dbstl\\_global\\_functions](#)

## abort\_txn

### Function Details

```
void abort_txn(DbEnv *env)
```

Abort current transaction of environment "env".

This function is called by dbstl user to abort an outside explicit transaction.

#### Parameters

##### env

The environment whose current transaction is to be aborted.

#### See Also

[abort\\_txn\(DbEnv \\*, DbTxn \\*\)](#) ;

```
void abort_txn(DbEnv *env,  
              DbTxn *txn)
```

Abort specified transaction "txn" and all its child transactions.

That is, "txn" can be a parent transaction of a nested transaction group.

#### Parameters

##### txn

The transaction to abort, can be a parent transaction of a nested transaction group, all child transactions of it will be aborted.

##### env

The environment where txn is started from.

#### See Also

[abort\\_txn\(DbEnv \\*\)](#) ;

### Group: Transaction control global functions.

dbstl transaction API.

You should call these API rather than DB C/C++ API to use Berkeley DB transaction features.

### Class

[dbstl\\_global\\_functions](#)

## current\_txn

### Function Details

```
DbTxn* current_txn(DbEnv *env)
```

Get current transaction of environment "env".

#### Parameters

**env**

The environment whose current transaction we want to get.

#### Return Value

Current transaction of env.

### Group: Transaction control global functions.

dbstl transaction API.

You should call these API rather than DB C/C++ API to use Berkeley DB transaction features.

### Class

[dbstl\\_global\\_functions](#)

## set\_current\_txn\_handle

### Function Details

```
DbTxn* set_current_txn_handle(DbEnv *env,  
                              DbTxn *newtxn)
```

Set environment env's current transaction handle to be newtxn.

The original transaction handle returned without aborting or committing. This function is used for users to use one transaction among multiple threads.

#### Parameters

##### newtxn

The new transaction to be as the current transaction of env.

##### env

The environment whose current transaction to replace.

#### Return Value

The old current transaction of env. It is not resolved.

### Group: Transaction control global functions.

dbstl transaction API.

You should call these API rather than DB C/C++ API to use Berkeley DB transaction features.

### Class

[dbstl\\_global\\_functions](#)

## register\_db

### Function Details

```
void register_db(Db *pdb1)
```

Register a Db handle "pdb1".

This handle and handles opened in it will be closed by [ResourceManager](#) , so application code must not try to close or delete it. Users can do enough configuration before opening the Db then register it via this function. All database handles should be registered via this function in each thread using the handle. The only exception is the database handle opened by [dbstl::open\\_db](#) should not be registered in the thread of the [dbstl::open\\_db](#) call.

### Parameters

#### **pdb1**

The database handle to register into dbstl for current thread.

### Class

[dbstl\\_global\\_functions](#)

## register\_db\_env

### Function Details

```
void register_db_env(DbEnv *env1)
```

Register a DbEnv handle env1, this handle and handles opened in it will be closed by [ResourceManager](#) .

Application code must not try to close or delete it. Users can do enough config before opening the DbEnv and then register it via this function. All environment handles should be registered via this function in each thread using the handle. The only exception is the environment handle opened by `dbstl::open_db_env` should not be registered in the thread of the `dbstl::open_db_env` call.

### Parameters

#### env1

The environment to register into dbstl for current thread.

### Class

[dbstl\\_global\\_functions](#)

## open\_db

### Function Details

```
Db* open_db(DbEnv *penv, const char *filename, DBTYPE dbtype,
            u_int32_t oflags, u_int32_t set_flags, int mode=0644, DbTxn *txn=NULL,
            u_int32_t cflags=0,
            const char *dbname=NULL)
```

Helper function to open a database and register it into dbstl for the calling thread.

Users still need to register it in any other thread using it if it is shared by multiple threads, via [register\\_db\(\)](#) function. Users don't need to delete or free the memory of the returned object, dbstl will take care of that. When you don't use [dbstl::open\\_db\(\)](#) but explicitly call DB C++ API to open a database, you must new the Db object, rather than create it on stack, and you must delete the Db object by yourself.

#### Parameters

**penv**

The environment to open the database from.

**txn**

The transaction to open the database from, passed to Db::open.

**dbtype**

The database type, passed to Db::open.

**oflags**

The database open flags, passed to Db::open.

**filename**

The database file name, passed to Db::open.

**mode**

The database open mode, passed to Db::open.

**cflags**

The create flags passed to Db class constructor.

**dbname**

The database name, passed to Db::open.

## **set\_flags**

The flags to be set to the created database handle.

### **Return Value**

The opened database handle.

### **See Also**

[register\\_db\(Db \\*\)](#) ;

[open\\_db\\_env](#);

## **Class**

[dbstl\\_global\\_functions](#)

## open\_env

### Function Details

```
DbEnv* open_env(const char *env_home, u_int32_t set_flags,  
                u_int32_t oflags=DB_CREATE|DB_INIT_MPOOL,  
                u_int32_t cachesize=4 *1024 *1024, int mode=0644,  
                u_int32_t cflags=0)
```

Helper function to open an environment and register it into dbstl for the calling thread.

Users still need to register it in any other thread if it is shared by multiple threads, via [register\\_db\\_env\(\)](#) function above. Users don't need to delete or free the memory of the returned object, dbstl will take care of that.

When you don't use [dbstl::open\\_env\(\)](#) but explicitly call DB C++ API to open an environment, you must new the DbEnv object, rather than create it on stack, and you must delete the DbEnv object by yourself.

#### Parameters

##### oflags

Environment open flags, passed to DbEnv::open.

##### set\_flags

Flags to set to the created environment before opening it.

##### mode

Environment region files mode, passed to DbEnv::open.

##### cflags

DbEnv constructor creation flags, passed to DbEnv::DbEnv.

##### cachesize

Environment cache size, by default 4M bytes.

##### env\_home

Environment home directory, it must exist. Passed to DbEnv::open.

#### Return Value

The opened database environment handle.

#### See Also

[register\\_db\\_env\(DbEnv \\*\)](#) ;

`open_db ;`

## **Class**

`dbstl_global_functions`

## alloc\_mutex

### Function Details

```
db_mutex_t alloc_mutex()
```

Allocate a Berkeley DB mutex.

#### Return Value

Berkeley DB mutex handle.

### Group: Mutex API based on Berkeley DB mutex.

These functions are in-process mutex support which uses Berkeley DB mutex mechanisms.

You can call these functions to do portable synchronization for your code.

### Class

[dbstl\\_global\\_functions](#)

## lock\_mutex

### Function Details

```
int lock_mutex(db_mutex_t mtx)
```

Lock a mutex, wait if it is held by another thread.

#### Parameters

**mtx**

The mutex handle to lock.

#### Return Value

0 if succeed, non-zero otherwise, call `db_strerror` to get message.

### Group: Mutex API based on Berkeley DB mutex.

These functions are in-process mutex support which uses Berkeley DB mutex mechanisms.

You can call these functions to do portable synchronization for your code.

### Class

[dbstl\\_global\\_functions](#)

## unlock\_mutex

### Function Details

```
int unlock_mutex(db_mutex_t mtx)
```

Unlock a mutex, and return immediately.

#### Parameters

**mtx**

The mutex handle to unlock.

#### Return Value

0 if succeed, non-zero otherwise, call `db_strerror` to get message.

### Group: Mutex API based on Berkeley DB mutex.

These functions are in-process mutex support which uses Berkeley DB mutex mechanisms.

You can call these functions to do portable synchronization for your code.

### Class

[dbstl\\_global\\_functions](#)

## free\_mutex

### Function Details

```
void free_mutex(db_mutex_t mtx)
```

Free a mutex, and return immediately.

#### Parameters

**mtx**

The mutex handle to free.

#### Return Value

0 if succeed, non-zero otherwise, call `db_strerror` to get message.

### Group: Mutex API based on Berkeley DB mutex.

These functions are in-process mutex support which uses Berkeley DB mutex mechanisms.

You can call these functions to do portable synchronization for your code.

### Class

[dbstl\\_global\\_functions](#)

## **dbstl\_startup**

### **Function Details**

```
void dbstl_startup()
```

If there are multiple threads within a process that make use of dbstl, then this function should be called in a single thread mutual exclusively before any use of dbstl in a process; Otherwise, you don't need to call it, but are allowed to call it anyway.

### **Class**

[dbstl\\_global\\_functions](#)

## **dbstl\_exit**

### **Function Details**

```
void dbstl_exit()
```

This function releases memory allocated by dbstl on the heap, and closes all Berkeley DB handles in the right order.

You can call [dbstl\\_exit\(\)](#) before the process exits to release any memory allocated by dbstl that has to persist during the entire process lifetime.

### **Class**

[dbstl\\_global\\_functions](#)

## **dbstl\_thread\_exit**

### **Function Details**

```
void dbstl_thread_exit()
```

This function closes all Berkeley DB handles in the right order, if other threads do not use them.

You can call this function before a thread exits to close unused Berkeley DB handles.

### **Class**

[dbstl\\_global\\_functions](#)

## operator==

### Function Details

```
bool operator==(const Dbt &d1,  
                const Dbt &d2)
```

Operators to compare two Dbt objects.

#### Parameters

**d2**

Dbt object to compare.

**d1**

Dbt object to compare.

```
bool operator==(const DBT &d1,  
                const DBT &d2)
```

Operators to compare two DBT objects.

#### Parameters

**d2**

DBT object to compare.

**d1**

DBT object to compare.

### Class

[dbstl\\_global\\_functions](#)

## set\_global\_dbfile\_suffix\_number

### Function Details

```
void set_global_dbfile_suffix_number(u_int32_t num)
```

If existing random temporary database name generation mechanism is still causing name clashes, users can set this global suffix number which will be append to each temporary database file name and incremented after each append, and by default it is 0.

### Parameters

**num**

Starting number to append to each temporary db file name.

### Class

[dbstl\\_global\\_functions](#)

## close\_db\_cursors

### Function Details

```
size_t close_db_cursors(Db *dbp1)
```

Close cursors opened in dbp1.

#### Parameters

##### dbp1

The database handle whose active cursors to close.

#### Return Value

The number of cursors closed by this call.

### Class

[dbstl\\_global\\_functions](#)

---

## Chapter 2. Dbstl Container Classes

A dbstl container is very much like a C++ STL container.

It stores a collection of data items, or key/data pairs. Each container is backed by a Berkeley DB database created in an explicit database environment or an internal private environment; And the database itself can be created explicitly with all kinds of configurations, or by dbstl internally. For each type of container, some specific type of database and/or configurations must be used or specified to the database and its environment. dbstl will check the database and environment conform to the requirement. When users don't have a chance to specify a container's backing database and environment, like in copy constructors, dbstl will create proper databases and/or environment for it. There are two helper functions to make it easier to create/open an environment or database, they are [dbstl::open\\_db\(\)](#) and [dbstl::open\\_env\(\)](#) ;

### See Also

[dbstl::open\\_db\(\)](#) [dbstl::open\\_env\(\)](#) [db\\_vector](#) [db\\_map](#) [db\\_multimap](#) [db\\_set](#) [db\\_multiset](#)

### Public Members

Member	Description
<a href="#">db_container</a>	db_container
<a href="#">db_map</a>	db_map
<a href="#">db_multimap</a>	db_multimap
<a href="#">db_set</a>	db_set
<a href="#">db_multiset</a>	db_multiset
<a href="#">db_vector</a>	db_vector

### Group

None

---

## Chapter 3. Db\_container

This class is the base class for all db container classes, you don't directly use this class, but all container classes inherit from this class, so you need to know the methods that can be accessed via concrete container classes.

This class is also used to support auto commit transactions. Autocommit is enabled when DB\_AUTO\_COMMIT is set to the database or database environment handle and the environment is transactional.

Inside dbctl, there are transactions begun and committed/aborted if the backing database and/or environment requires auto commit, and there are cursors opened internally, and you can set the flags used by the transaction and cursor functions via set functions of this class.

All dbctl containers are fully multi-threaded, you should not need any synchronization to use them in the correct way, but this class is not thread safe, access to its members are not protected by any mutex because the data members of this class are supposed to be set before they are used, and remain read only afterwards. If this is not the case, you must synchronize the access.

### Public Members

Member	Description
<a href="#">get_db_open_flags (page 32)</a>	Get the backing database's open flags.
<a href="#">get_db_set_flags (page 33)</a>	Get the backing database's flags that are set via Db::set_flags() function.
<a href="#">get_db_handle (page 34)</a>	Get the backing database's handle.
<a href="#">get_db_env_handle (page 35)</a>	Get the backing database environment's handle.
<a href="#">set_db_handle (page 36)</a>	Set the underlying database's handle, and optionally environment handle if the environment has also changed.
<a href="#">set_all_flags (page 37)</a>	Set the flags required by the Berkeley DB functions DbEnv::txn_begin(), DbTxn::commit() and DbEnv::cursor().
<a href="#">set_txn_begin_flags (page 38)</a>	Set flag of DbEnv::txn_begin() call.
<a href="#">get_txn_begin_flags (page 39)</a>	Get flag of DbEnv::txn_begin() call.
<a href="#">set_commit_flags (page 40)</a>	Set flag of DbTxn::commit() call.
<a href="#">get_commit_flags (page 41)</a>	Get flag of DbTxn::commit() call.
<a href="#">get_cursor_open_flags (page 42)</a>	Get flag of Db::cursor() call.
<a href="#">set_cursor_open_flags (page 43)</a>	Set flag of Db::cursor() call.
<a href="#">db_container (page 44)</a>	Default constructor.
<a href="#">~db_container (page 46)</a>	The backing database is not closed in this function.

**Group**

[Dbstl Container Classes \(page 29\)](#)

## get\_db\_open\_flags

### Function Details

```
u_int32_t get_db_open_flags() const
```

Get the backing database's open flags.

#### Return Value

The backing database's open flags.

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_db\_set\_flags

### Function Details

```
u_int32_t get_db_set_flags() const
```

Get the backing database's flags that are set via `Db::set_flags()` function.

#### Return Value

Flags set to this container's database handle.

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of `db_container` are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_db\_handle

### Function Details

```
Db* get_db_handle() const
```

Get the backing database's handle.

#### Return Value

The backing database handle of this container.

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_db\_env\_handle

### Function Details

```
DbEnv* get_db_env_handle() const
```

Get the backing database environment's handle.

#### Return Value

The backing database environment handle of this container.

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## set\_db\_handle

### Function Details

```
void set_db_handle(Db *dbp,  
                  DbEnv *newenv=NULL)
```

Set the underlying database's handle, and optionally environment handle if the environment has also changed.

That is, users can change the container object's underlying database while the object is alive. dbstl will verify that the handles set conforms to the concrete container's requirement to Berkeley DB database/environment handles.

### Parameters

#### **dbp**

The database handle to set.

#### **newenv**

The database environment handle to set.

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## set\_all\_flags

### Function Details

```
void set_all_flags(u_int32_t txn_begin_flags, u_int32_t commit_flags,  
                  u_int32_t cursor_open_flags)
```

Set the flags required by the Berkeley DB functions `DbEnv::txn_begin()`, `DbTxn::commit()` and `DbEnv::cursor()`.

These flags will be set to this container's auto commit member functions when auto commit transaction is used, except that `cursor_oflags` is set to the `Dbc::cursor` when creating an iterator for this container. By default the three flags are all zero. You can also set the values of the flags individually by using the appropriate set functions in this class. The corresponding get functions return the flags actually used.

### Parameters

#### **commit\_flags**

Flags to be set to `DbTxn::commit()`.

#### **cursor\_open\_flags**

Flags to be set to `Db::cursor()`.

#### **txn\_begin\_flags**

Flags to be set to `DbEnv::txn_begin()`.

### **Group: Get and set functions for data members.**

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### **Class**

[db\\_container](#)

## set\_txn\_begin\_flags

### Function Details

```
void set_txn_begin_flags(u_int32_t flag)
```

Set flag of DbEnv::txn\_begin() call.

#### Parameters

##### flag

Flags to be set to DbEnv::txn\_begin().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_txn\_begin\_flags

### Function Details

```
u_int32_t get_txn_begin_flags() const
```

Get flag of DbEnv::txn\_begin() call.

#### Return Value

Flags to be set to DbEnv::txn\_begin().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## set\_commit\_flags

### Function Details

```
void set_commit_flags(u_int32_t flag)
```

Set flag of DbTxn::commit() call.

### Parameters

#### flag

Flags to be set to DbTxn::commit().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_commit\_flags

### Function Details

```
u_int32_t get_commit_flags() const
```

Get flag of DbTxn::commit() call.

#### Return Value

Flags to be set to DbTxn::commit().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## get\_cursor\_open\_flags

### Function Details

```
u_int32_t get_cursor_open_flags() const
```

Get flag of Db::cursor() call.

#### Return Value

Flags to be set to Db::cursor().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## set\_cursor\_open\_flags

### Function Details

```
void set_cursor_open_flags(u_int32_t flag)
```

Set flag of Db::cursor() call.

#### Parameters

##### flag

Flags to be set to Db::cursor().

### Group: Get and set functions for data members.

Note that these functions are not thread safe, because all data members of [db\\_container](#) are supposed to be set on container construction and initialization, and remain read only afterwards.

### Class

[db\\_container](#)

## db\_container

### Function Details

```
db_container()
```

Default constructor.

```
db_container(const db_container &dbctnr)
```

Copy constructor.

The new container will be backed by another database within the same environment unless dbctnr's backing database is in its own internal private environment. The name of the database is coined based on current time and thread id and some random number. If this is still causing naming clashes, you can set a suffix number via "set\_global\_dbfile\_suffix\_number" function; And following db file will suffix this number in the file name for additional randomness. And the suffix will be incremented after each such use. You can change the file name via DbEnv::rename. If dbctnr is using an anonymous database, the newly constructed container will also use an anonymous one.

### Parameters

#### dbctnr

The container to initialize this container.

```
db_container(Db *dbp,  
            DbEnv *envp)
```

This constructor is not directly called by the user, but invoked by constructors of concrete container classes.

The statement about the parameters applies to constructors of all container classes.

### Parameters

#### dbp

Database handle. dbp is supposed to be opened inside envp. Each dbstl container is backed by a Berkeley DB database, so dbstl will create an internal anonymous database if dbp is NULL.

#### envp

Environment handle. And envp can also be NULL, meaning the dbp handle may be created in its internal private environment.

## **Class**

`db_container`

## ~db\_container

### Function Details

```
virtual ~db_container()
```

The backing database is not closed in this function.

It is closed when current thread exits and the database is no longer referenced by any other container instances in this process. In order to make the reference counting work alright, you must call [register\\_db\(Db\\*\)](#) and [register\\_db\\_env\(DbEnv\\*\)](#) correctly.

#### See Also

[register\\_db\(Db\\*\)](#) [register\\_db\\_env\(DbEnv\\*\)](#)

### Class

[db\\_container](#)

---

## Chapter 4. Db\_vector

The `db_vector` class has the union set of public member functions as `std::vector`, `std::deque` and `std::list`, and each method has identical default semantics to that in the `std` equivalent containers.

The difference is that the data is maintained using a Berkeley DB database as well as some Berkeley DB related extensions.

### See Also

[db\\_container db\\_container\(Db\\*, DbEnv\\*\) db\\_container\(const db\\_container&\)](#)

### Class Template Parameters

#### T

The type of data to store.

#### value\_type\_sub

If T is a class/struct type, do not specify anything for this parameter; Otherwise, specify `ElementHolder<T>` to it. `Database(dbp)` and `environment(penv)` handle requirement (applies for all constructors of this class template): `dbp` must meet the following requirement: 1. `dbp` must be a `DB_RECNO` type of database handle. 2. `DB_THREAD` must be set to `dbp`'s open flags. 3. An optional flag `DB_RENUMBER` is required if the container object is supposed to be a `std::vector` or `std::deque` equivalent; Not required if it is a `std::list` equivalent. But `dbstl` will not check whether `DB_RENUMBER` is set to this database handle. Setting `DB_RENUMBER` will cause the index values of all elements in the underlying database to be maintained consecutive and in order, which involves potentially a lot of work because many indices may be updated. See the [db\\_container\(Db\\*, DbEnv\\*\)](#) for more information about the two parameters.

### Public Members

Member	Description
<a href="#">begin (page 49)</a>	Create a read-write or read-only iterator.
<a href="#">end (page 51)</a>	Create an open boundary iterator.
<a href="#">rbegin (page 52)</a>	Create a reverse iterator.
<a href="#">rend (page 54)</a>	Create an open boundary iterator.
<a href="#">max_size (page 55)</a>	Get max size.
<a href="#">capacity (page 56)</a>	Get capacity.
<a href="#">operator[] (page 57)</a>	Index operator, can act as both a left value and a right value.
<a href="#">at (page 58)</a>	Index function.
<a href="#">front (page 60)</a>	Return a reference to the first element.
<a href="#">back (page 61)</a>	Return a reference to the last element.

Member	Description
<a href="#">operator== (page 62)</a>	Container equality comparison operator.
<a href="#">operator!= (page 63)</a>	Container in-equality comparison operator.
<a href="#">operator&lt; (page 64)</a>	Container less than comparison operator.
<a href="#">assign (page 65)</a>	Assign a range [first, last) to this container.
<a href="#">push_front (page 67)</a>	Push an element x into the vector from front.
<a href="#">pop_front (page 68)</a>	Pop out the front element from the vector.
<a href="#">insert (page 69)</a>	Insert x before position pos.
<a href="#">erase (page 71)</a>	Erase element at position pos.
<a href="#">remove (page 72)</a>	Remove all elements whose values are "value" from the list.
<a href="#">remove_if (page 73)</a>	Remove all elements making "pred" return true.
<a href="#">merge (page 74)</a>	Merge content with another container.
<a href="#">unique (page 75)</a>	Remove consecutive duplicate values from this list.
<a href="#">sort (page 76)</a>	Sort this list.
<a href="#">reverse (page 77)</a>	Reverse this list.
<a href="#">splice (page 78)</a>	Moves elements from list x into this list.
<a href="#">size (page 80)</a>	Return the number of elements in this container.
<a href="#">empty (page 81)</a>	Returns whether this container is empty.
<a href="#">db_vector (page 82)</a>	Constructor.
<a href="#">~db_vector (page 85)</a>	
<a href="#">operator= (page 86)</a>	Container assignment operator.
<a href="#">resize (page 87)</a>	Resize this container to specified size n, insert values t if need to enlarge the container.
<a href="#">reserve (page 88)</a>	Reserve space.
<a href="#">push_back (page 89)</a>	Push back an element into the vector.
<a href="#">pop_back (page 90)</a>	Pop out last element from the vector.
<a href="#">swap (page 91)</a>	Swap content with another vector vec.
<a href="#">clear (page 92)</a>	Remove all elements of the vector, make it an empty vector.

**Group**

[Dbstl Container Classes \(page 29\)](#)

## begin

### Function Details

```
iterator begin(ReadModifyWriteOption rmw=
    ReadModifyWriteOption::no_read_modify_write(), bool readonly=false,
    BulkRetrievalOption bulk_read=BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true)
```

Create a read-write or read-only iterator.

We allow users to create a readonly iterator here so that they don't have to use a const container to create a const\_iterator. But using const\_iterator is faster. The flags set via `db_container::set_cursor_oflags()` is used as the cursor open flags.

#### Parameters

##### **directdb\_get**

Whether always read key/data pair from backing db rather than using the value cached in the iterator. The current key/data pair is cached in the iterator and always kept updated on iterator movement, but in some extreme conditions, errors can happen if you use cached key/data pairs without always refreshing them from database. By default we are always reading from database when we are accessing the data the iterator sits on, except when we are doing bulk retrievals. But your application can gain extra performance promotion if you can set this flag to false.

##### **readonly**

Whether the iterator is created as a readonly iterator. Read only iterators can not update its underlying key/data pair.

##### **bulk\_read**

Whether read database key/data pairs in bulk, by specifying `DB_MULTIPLE_KEY` flag to underlying cursor's `Dbc::get` function. Only readonly iterators can do bulk retrieval, if iterator is not read only, this parameter is ignored. Bulk retrieval can accelerate reading speed because each database read operation will read many key/data pairs, thus saved many database read operations. The default bulk buffer size is 32KB, you can set your desired bulk buffer size by specifying `BulkRetrievalOpt::bulk_retrieval(your_bulk_buffer_size)`; If you don't want bulk retrieval, set `BulkRetrievalOpt::no_bulk_retrieval()` as the real parameter.

##### **rmw**

Whether this iterator will open a Berkeley DB cursor with `DB_RMW` flag set. If the iterator is used to read a key/data pair, then update it and store back to db, it is good to set the `DB_RMW` flag, by specifying `RMWltrOpt::read_modify_write()` If you don't want to set the `DB_RMW` flag, specify `RMWltrOpt::no_read_modify_write()`, which is the default behavior.

**Return Value**

The created iterator.

**See Also**

db\_container::set\_cursor\_oflags();

```
const_iterator begin(BulkRetrievalOption bulkretrieval=
    (BulkRetrievalOption::no_bulk_retrieval()),
    bool directdb_get=true) const
```

Create a const iterator.

The created iterator can only be used to read its referenced data element. Can only be called when using a const reference to the container object. The parameters have identical meanings and usage to those of the other non-const begin function.

**Parameters****directdb\_get**

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

**bulkretrieval**

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

**Return Value**

The created const iterator.

**See Also**

begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

**Class**

[db\\_vector](#)

## end

### Function Details

```
iterator end()
```

Create an open boundary iterator.

#### Return Value

Returns an invalid iterator denoting the position after the last valid element of the container.

```
const_iterator end() const
```

Create an open boundary iterator.

#### Return Value

Returns an invalid const iterator denoting the position after the last valid element of the container.

### Class

[db\\_vector](#)

# rbegin

## Function Details

```
reverse_iterator rbegin(ReadModifyWriteOption rmw=
    ReadModifyWriteOption::no_read_modify_write(), bool readonly=false,
    BulkRetrievalOption bulk_read=BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true)
```

Create a reverse iterator.

This function creates a reverse iterator initialized to sit on the last element in the underlying database, and can be used to read/write. The meaning and usage of its parameters are identical to the above begin function.

### Parameters

#### directdb\_get

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

#### bulk\_read

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

#### rmw

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

#### readonly

Same as that of begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

### Return Value

The created iterator.

### See Also

begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);

```
const_reverse_iterator rbegin(BulkRetrievalOption bulkretrieval=
    BulkRetrievalOption(BulkRetrievalOption::no_bulk_retrieval()),
    bool directdb_get=true) const
```

Create a const reverse iterator.

This function creates a const reverse iterator initialized to sit on the last element in the backing database, and can only read the element, it is only available to const [db\\_vector](#) containers. The meaning and usage of its parameters are identical as above.

### Parameters

#### **directdb\_get**

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

#### **bulkretrieval**

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

### Return Value

The created iterator.

### See Also

`begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

### Class

[db\\_vector](#)

# rend

## Function Details

```
reverse_iterator rend()
```

Create an open boundary iterator.

### Return Value

Returns an invalid iterator denoting the position before the first valid element of the container.

```
const_reverse_iterator rend() const
```

Create an open boundary iterator.

### Return Value

Returns an invalid const iterator denoting the position before the first valid element of the container.

## Class

[db\\_vector](#)

## max\_size

### Function Details

```
size_type max_size() const
```

Get max size.

The returned size is not the actual limit of database. See the Berkeley DB limits to get real max size.

#### Return Value

A meaningless huge number.

### Group: Huge return

These two functions return  $2^{30}$ , denoting a huge number that does not overflow, because dbstl does not have to manage memory space.

But the return value is not the real limit, see the Berkeley DB database limits for the limits.

### Class

[db\\_vector](#)

## capacity

### Function Details

```
size_type capacity() const
```

Get capacity.

### Group: Huge return

These two functions return  $2^{30}$ , denoting a huge number that does not overflow, because dbstdl does not have to manage memory space.

But the return value is not the real limit, see the Berkeley DB database limits for the limits.

### Class

[db\\_vector](#)

## operator[]

### Function Details

```
reference operator[](index_type n)
```

Index operator, can act as both a left value and a right value.

#### Parameters

**n**

The valid index of the vector.

#### Return Value

The reference to the element at specified position.

```
const_reference operator[](index_type n) const
```

Read only index operator.

Only used as a right value, no need for assignment capability. The return value can't be used to update the element.

#### Parameters

**n**

The valid index of the vector.

#### Return Value

The const reference to the element at specified position.

### Group: Element access functions.

The `operator[]` and `at()` only come from `std::vector` and `std::deque`, If you are using `db_vector` as `std::list`, you don't have to set `DB_RENUMBER` flag to the backing database handle, and you get better performance, but at the same time you can't use these functions.

Otherwise if you have set the `DB_RENUMBER` flag to the backing database handle, you can use this function though it is an `std::list` equivalent.

### Class

`db_vector`

## at

### Function Details

```
reference at(index_type n)
```

Index function.

#### Parameters

**n**

The valid index of the vector.

#### Return Value

The reference to the element at specified position, can act as both a left value and a right value.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/at.html>

```
const_reference at(index_type n) const
```

Read only index function.

Only used as a right value, no need for assignment capability. The return value can't be used to update the element.

#### Parameters

**n**

The valid index of the vector.

#### Return Value

The const reference to the element at specified position.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/at.html>

### Group: Element access functions.

The operator[] and [at\(\)](#) only come from `std::vector` and `std::deque`, If you are using [db\\_vector](#) as `std::list`, you don't have to set `DB_RENUMBER` flag to the backing database handle, and you get better performance, but at the same time you can't use these functions.

Otherwise if you have set the DB\_RENUMBER flag to the backing database handle, you can use this function though it is an `std::list` equivalent.

## **Class**

[db\\_vector](#)

# front

## Function Details

```
reference front()
```

Return a reference to the first element.

### Return Value

Return a reference to the first element.

### See Also

<http://www.cplusplus.com/reference/stl/vector/front.html>

```
const_reference front() const
```

Return a const reference to the first element.

The return value can't be used to update the element.

### Return Value

Return a const reference to the first element.

### See Also

<http://www.cplusplus.com/reference/stl/vector/front.html>

## Group: Element access functions.

The operator[] and [at\(\)](#) only come from `std::vector` and `std::deque`, If you are using [db\\_vector](#) as `std::list`, you don't have to set `DB_RENUMBER` flag to the backing database handle, and you get better performance, but at the same time you can't use these functions.

Otherwise if you have set the `DB_RENUMBER` flag to the backing database handle, you can use this function though it is an `std::list` equivalent.

## Class

[db\\_vector](#)

## back

### Function Details

```
reference back()
```

Return a reference to the last element.

#### Return Value

Return a reference to the last element.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/back.html>

```
const_reference back() const
```

Return a reference to the last element.

The return value can't be used to update the element.

#### Return Value

Return a reference to the last element.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/back.html>

### Group: Element access functions.

The operator[] and [at\(\)](#) only come from `std::vector` and `std::deque`, If you are using [db\\_vector](#) as `std::list`, you don't have to set `DB_RENUMBER` flag to the backing database handle, and you get better performance, but at the same time you can't use these functions.

Otherwise if you have set the `DB_RENUMBER` flag to the backing database handle, you can use this function though it is an `std::list` equivalent.

### Class

[db\\_vector](#)

## operator==

### Function Details

```
bool operator==(const db_vector< T2,  
                T3 > &v2) const
```

Container equality comparison operator.

This function supports auto-commit.

#### Parameters

#### v2

The vector to compare against.

#### Return Value

Compare two vectors, return true if they have identical sequences of elements, otherwise return false.

```
bool operator==(const self &v2) const
```

Container equality comparison operator.

This function supports auto-commit.

#### Return Value

Compare two vectors, return true if they have identical elements, otherwise return false.

### Group: Compare functions.

<http://www.sgi.com/tech/stl/Vector.html>

### Class

[db\\_vector](#)

## operator!=

### Function Details

```
bool operator!=(const db_vector< T2,  
                T3 > &v2) const
```

Container in-equality comparison operator.

This function supports auto-commit.

#### Parameters

**v2**

The vector to compare against.

#### Return Value

Returns false if elements in each slot of both containers equal; Returns true otherwise.

```
bool operator!=(const self &v2) const
```

Container in-equality comparison operator.

This function supports auto-commit.

#### Parameters

**v2**

The vector to compare against.

#### Return Value

Returns false if elements in each slot of both containers equal; Returns true otherwise.

### Group: Compare functions.

<http://www.sgi.com/tech/stl/Vector.html>

### Class

[db\\_vector](#)

## operator<

### Function Details

```
bool operator<(const self &v2) const
```

Container less than comparison operator.

This function supports auto-commit.

#### Parameters

#### v2

The container to compare against.

#### Return Value

Compare two vectors, return true if this is less than v2, otherwise return false.

### Group: Compare functions.

<http://www.sgi.com/tech/stl/Vector.html>

### Class

[db\\_vector](#)

## assign

### Function Details

```
void assign(InputIterator first, InputIterator last,  
            bool b_truncate=true)
```

Assign a range [first, last) to this container.

#### Parameters

##### **b\_truncate**

See its member group doc for details.

##### **last**

The range open boundary.

##### **first**

The range closed boundary.

```
void assign(const_iterator first, const_iterator last,  
            bool b_truncate=true)
```

Assign a range [first, last) to this container.

#### Parameters

##### **b\_truncate**

See its member group doc for details.

##### **last**

The range open boundary.

##### **first**

The range closed boundary.

```
void assign(size_type n, const T &u,  
            bool b_truncate=true)
```

Assign n number of elements of value u into this container.

### Parameters

#### **b\_truncate**

See its member group doc for details. This function supports auto-commit.

#### **u**

The value of elements to insert.

#### **n**

The number of elements in this container after the call.

### Group: Assign functions

See the function documentation for the correct usage of `b_truncate` parameter.

The following four member functions have default parameter `b_truncate`, because they require all key/data pairs in the database be deleted before the real operation, and by default we use `Db::truncate` to truncate the database rather than delete the key/data pairs one by one, but `Db::truncate` requires no open cursors on the database handle, and the four member functions will close any open cursors of backing database handle in current thread, but can do nothing to cursors of other threads opened from the same database handle. So you must make sure there are no open cursors of the database handle in any other threads. On the other hand, users can specify "false" to the `b_truncate` parameter and thus the key/data pairs will be deleted one by one. Other than that, they have identical behaviors as their counterparts in `std::vector`.

<http://www.cplusplus.com/reference/stl/vector/assign.html>

### Class

[db\\_vector](#)

## push\_front

### Function Details

```
void push_front(const T &x)
```

Push an element *x* into the vector from front.

#### Parameters

**x**

The element to push into this vector. This function supports auto-commit.

### Group: Functions specific to deque and list

These functions come from `std::list` and `std::deque`, and have identical behaviors to their counterparts in `std::list/stddeque`.

[http://www.cplusplus.com/reference/stl/deque/pop\\_front.html](http://www.cplusplus.com/reference/stl/deque/pop_front.html) [http://www.cplusplus.com/reference/stl/deque/push\\_front.html](http://www.cplusplus.com/reference/stl/deque/push_front.html)

### Class

[db\\_vector](#)

## pop\_front

### Function Details

```
void pop_front()
```

Pop out the front element from the vector.

This function supports auto-commit.

### Group: Functions specific to deque and list

These functions come from `std::list` and `std::deque`, and have identical behaviors to their counterparts in `std::list/stddeque`.

[http://www.cplusplus.com/reference/stl/deque/pop\\_front.html](http://www.cplusplus.com/reference/stl/deque/pop_front.html) [http://www.cplusplus.com/reference/stl/deque/push\\_front.html](http://www.cplusplus.com/reference/stl/deque/push_front.html)

### Class

[db\\_vector](#)

# insert

## Function Details

```
iterator insert(iterator pos,
               const T &x)
```

Insert x before position pos.

### Parameters

**x**

The element to insert.

**pos**

The position before which to insert.

```
void insert(iterator pos, size_type n,
           const T &x)
```

Insert n number of elements x before position pos.

### Parameters

**x**

The element to insert.

**pos**

The position before which to insert.

**n**

The number of elements to insert.

```
void insert(iterator pos, InputIterator first,
           InputIterator last)
```

Range insertion.

Insert elements in range [first, last) into this vector before position pos.

### Parameters

**last**

The open boundary of the range.

**pos**

The position before which to insert.

**first**

The closed boundary of the range.

```
void insert(iterator pos, const_iterator first,  
            const_iterator last)
```

Range insertion.

Insert elements in range [first, last) into this vector before position pos.

**Parameters****last**

The open boundary of the range.

**pos**

The position before which to insert.

**first**

The closed boundary of the range.

**Group: Insert functions**

The iterator pos in the functions must be a read-write iterator, can't be read only.

<http://www.cplusplus.com/reference/stl/vector/insert.html>

**Class**

[db\\_vector](#)

## erase

### Function Details

```
iterator erase(iterator pos)
```

Erase element at position pos.

#### Parameters

**pos**

The valid position in the container's range to erase.

#### Return Value

The next position after the erased element.

```
iterator erase(iterator first,  
              iterator last)
```

Erase elements in range [first, last).

#### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

#### Return Value

The next position after the erased elements.

### Group: Erase functions

The iterator pos in the functions must be a read-write iterator, can't be read only.

<http://www.cplusplus.com/reference/stl/vector/erase.html>

### Class

[db\\_vector](#)

## remove

### Function Details

```
void remove(const T &value)
```

Remove all elements whose values are "value" from the list.

This function supports auto-commit.

#### Parameters

#### value

The target value to remove.

#### See Also

<http://www.cplusplus.com/reference/stl/list/remove/>

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

## remove\_if

### Function Details

```
void remove_if(Predicate pred)
```

Remove all elements making "pred" return true.

This function supports auto-commit.

### Parameters

#### pred

The binary predicate judging elements in this list.

### See Also

[http://www.cplusplus.com/reference/stl/list/remove\\_if/](http://www.cplusplus.com/reference/stl/list/remove_if/)

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

## merge

### Function Details

```
void merge(self &x)
```

Merge content with another container.

This function supports auto-commit.

#### Parameters

**x**

The other list to merge with.

#### See Also

<http://www.cplusplus.com/reference/stl/list/merge/>

```
void merge(self &x,  
           Compare comp)
```

Merge content with another container.

This function supports auto-commit.

#### Parameters

**x**

The other list to merge with.

**comp**

The compare function to determine insertion position.

#### See Also

<http://www.cplusplus.com/reference/stl/list/merge/>

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

## unique

### Function Details

```
void unique()
```

Remove consecutive duplicate values from this list.

This function supports auto-commit.

#### See Also

<http://www.cplusplus.com/reference/stl/list/unique/>

```
void unique(BinaryPredicate binary_pred)
```

Remove consecutive duplicate values from this list.

This function supports auto-commit.

#### Parameters

### binary\_pred

The compare predicate to determine uniqueness.

#### See Also

<http://www.cplusplus.com/reference/stl/list/unique/>

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

## sort

### Function Details

```
void sort()
```

Sort this list.

This function supports auto-commit.

#### See Also

<http://www.cplusplus.com/reference/stl/list/sort/>

```
void sort(Compare comp)
```

Sort this list.

This function supports auto-commit.

#### Parameters

## comp

The compare operator to determine element order.

#### See Also

<http://www.cplusplus.com/reference/stl/list/sort/>

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

## reverse

### Function Details

```
void reverse()
```

Reverse this list.

This function supports auto-commit.

#### See Also

<http://www.cplusplus.com/reference/stl/list/reverse/>

### Group: std::list specific functions

<http://www.cplusplus.com/reference/stl/list/>

### Class

[db\\_vector](#)

# splice

## Function Details

```
void splice(iterator position,  
            self &x)
```

Moves elements from list x into this list.

Moves all elements in list x into this list container at the specified position, effectively inserting the specified elements into the container and removing them from x. This function supports auto-commit.

### Parameters

#### position

Position within the container where the elements of x are inserted.

#### x

The other list container to splice from.

### See Also

<http://www.cplusplus.com/reference/stl/list/splice/>

```
void splice(iterator position, self &x,  
            iterator i)
```

Moves elements from list x into this list.

Moves elements at position i of list x into this list container at the specified position, effectively inserting the specified elements into the container and removing them from x. This function supports auto-commit.

### Parameters

#### i

The position of element in x to move into this list.

#### position

Position within the container where the elements of x are inserted.

#### x

The other list container to splice from.

**See Also**

<http://www.cplusplus.com/reference/stl/list/splice/>

```
void splice(iterator position, self &x, iterator first,
            iterator last)
```

Moves elements from list x into this list.

Moves elements in range [first, last) of list x into this list container at the specified position, effectively inserting the specified elements into the container and removing them from x. This function supports auto-commit.

**Parameters****position**

Position within the container where the elements of x are inserted.

**first**

The range's closed boundary.

**last**

The range's open boundary.

**x**

The other list container to splice from.

**See Also**

<http://www.cplusplus.com/reference/stl/list/splice/>

**Group: std::list specific functions**

<http://www.cplusplus.com/reference/stl/list/>

**Class**

[db\\_vector](#)

## size

### Function Details

```
size_type size() const
```

Return the number of elements in this container.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/size.html>

### Class

[db\\_vector](#)

## empty

### Function Details

```
bool empty() const
```

Returns whether this container is empty.

#### Return Value

True if empty, false otherwise.

### Class

[db\\_vector](#)

## db\_vector

### Function Details

```
db_vector(Db *dbp=NULL,  
          DbEnv *penv=NULL)
```

Constructor.

Note that we do not need an allocator in db-stl container, but we need backing up Db\* and DbEnv\*, and we have to verify that the passed in bdb handles are valid for use by the container class. See class detail for handle requirement.

#### Parameters

##### dbp

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

##### penv

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

#### See Also

[db\\_container\(Db\\*, DbEnv\\*\)](#) ;

```
db_vector(size_type n, const T &val=T(), Db *dbp=NULL,  
          DbEnv *penv=NULL)
```

Constructor.

This function supports auto-commit. Insert n elements of T type into the database, the value of the elements is the default value or user set value. See class detail for handle requirement.

#### Parameters

##### dbp

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

##### penv

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

##### val

The value of elements to insert.

**n**

The number of elements to insert.

**See Also**

[db\\_vector\(Db\\*, DbEnv\\*\)](#) ; [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

```
db_vector(const self &x)
```

Copy constructor.

This function supports auto-commit. Insert all elements in x into this container.

**See Also**

[db\\_container\(const db\\_container&\)](#)

```
db_vector(Db *dbp, DbEnv *penv, InputIterator first,
          InputIterator last)
```

Insert a range of elements into this container.

The range is [first, last), which contains elements that can be converted to type T automatically. See class detail for handle requirement.

**Parameters****dbp**

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

**first**

Range closed boundary.

**last**

Range open boundary.

**penv**

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

**See Also**

[db\\_vector\(Db\\*, DbEnv\\*\)](#) ;

```
db_vector(const_iterator first, const_iterator last, Db *dbp=NULL,
```

---

```
DbEnv *penv=NULL)
```

Range constructor.

This function supports auto-commit. Insert the range of elements in [first, last) into this container. See class detail for handle requirement.

### Parameters

#### **dbp**

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

#### **first**

Range closed boundary.

#### **last**

Range open boundary.

#### **penv**

The same as that of [db\\_container\(Db\\*, DbEnv\\*\)](#) ;

### See Also

[db\\_vector\(Db\\*, DbEnv\\*\)](#) ;

### Class

[db\\_vector](#)

## ~db\_vector

### Function Details

```
virtual ~db_vector()
```

### Class

[db\\_vector](#)

## operator=

### Function Details

```
const self& operator=(const self &x)
```

Container assignment operator.

This function supports auto-commit. This [db\\_vector](#) is assumed to be valid for use, only copy content of x into this container.

### Parameters

**x**

The right value container.

### Return Value

The container x's reference.

### Class

[db\\_vector](#)

## resize

### Function Details

```
void resize(size_type n,  
            T t=T())
```

Resize this container to specified size *n*, insert values *t* if need to enlarge the container.

This function supports auto-commit.

#### Parameters

**t**

The value to insert when enlarging the container.

**n**

The number of elements in this container after the call.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/resize.html>

### Class

[db\\_vector](#)

## reserve

### Function Details

```
void reserve(size_type)
```

Reserve space.

The vector is backed by Berkeley DB, we always have enough space. This function does nothing, because dbstl does not have to manage memory space.

### Class

[db\\_vector](#)

## push\_back

### Function Details

```
void push_back(const T &x)
```

Push back an element into the vector.

This function supports auto-commit.

### Parameters

**x**

The value of element to push into this vector.

### See Also

[http://www.cplusplus.com/reference/stl/vector/push\\_back.html](http://www.cplusplus.com/reference/stl/vector/push_back.html)

### Class

[db\\_vector](#)

## pop\_back

### Function Details

```
void pop_back()
```

Pop out last element from the vector.

This function supports auto-commit.

#### See Also

[http://www.cplusplus.com/reference/stl/vector/pop\\_back.html](http://www.cplusplus.com/reference/stl/vector/pop_back.html)

### Class

[db\\_vector](#)

## swap

### Function Details

```
void swap(self &vec)
```

Swap content with another vector vec.

#### Parameters

#### vec

The other vector to swap content with. This function supports auto-commit.

#### See Also

<http://www.cplusplus.com/reference/stl/vector/swap.html>

### Class

[db\\_vector](#)

## clear

### Function Details

```
void clear(bool b_truncate=true)
```

Remove all elements of the vector, make it an empty vector.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

Same as that of [db\\_vector::assign\(\)](#) .

#### See Also

<http://www.cplusplus.com/reference/stl/vector/clear.html>

### Class

[db\\_vector](#)

---

## Chapter 5. Db\_map

[db\\_map](#) has identical methods to `std::map` and the semantics for each method is identical to its `std::map` counterpart, except that it stores data into underlying Berkeley DB btree or hash database.

Passing a database handle of btree or hash type creates a [db\\_map](#) equivalent to `std::map` and `std::hashmap` respectively. Database(`dbp`) and environment(`penv`) handle requirement(applyes to all constructors in this class template): 0. The `dbp` is opened inside the `penv` environment. Either one of the two handles can be NULL. If `dbp` is NULL, an anonymous database is created by `dbstl`. 1. Database type of `dbp` should be `DB_BTREE` or `DB_HASH`. 2. No `DB_DUP` or `DB_DUPSORT` flag set in `dbp`. 3. No `DB_RECNUM` flag set in `dbp`. 4. No `DB_TRUNCATE` specified in `dbp`'s database open flags. 5. `DB_THREAD` must be set if you are sharing the `dbp` across multiple threads directly, or indirectly by sharing the container object across multiple threads.

### See Also

[db\\_container](#) [db\\_container\(Db\\*, DbEnv\\*\)](#) [db\\_container\(const db\\_container&\)](#)

### Class Template Parameters

#### **kdt**

The key data type.

#### **ddt**

The data data type. [db\\_map](#) stores key/data pairs.

#### **value\_type\_sub**

Do not specify anything if `ddt` type is a class/struct type; Otherwise, specify `ElementHolder<ddt>` to it.

#### **iterator\_t**

Never specify anything to this type parameter. It is only used internally.

### Public Members

Member	Description
<a href="#">db_map</a> (page 95)	Create a <code>std::map</code> / <code>hash_map</code> equivalent associative container.
<a href="#">~db_map</a> (page 97)	
<a href="#">insert</a> (page 98)	Insert a single key/data pair if the key is not in the container.
<a href="#">begin</a> (page 100)	Begin a read-write or readonly iterator which sits on the first key/data pair of the database.

Member	Description
<a href="#">end (page 102)</a>	Create an open boundary iterator.
<a href="#">rbegin (page 103)</a>	Begin a read-write or readonly reverse iterator which sits on the first key/data pair of the database.
<a href="#">rend (page 105)</a>	Create an open boundary iterator.
<a href="#">is_hash (page 106)</a>	Get container category.
<a href="#">bucket_count (page 107)</a>	Only for <code>std::hash_map</code> , return number of hash bucket in use.
<a href="#">size (page 108)</a>	This function supports auto-commit.
<a href="#">max_size (page 109)</a>	Get max size.
<a href="#">empty (page 110)</a>	Returns whether this container is empty.
<a href="#">erase (page 111)</a>	Erase a key/data pair at specified position.
<a href="#">find (page 113)</a>	Find the key/data pair with specified key x.
<a href="#">lower_bound (page 115)</a>	Find the greatest key less than or equal to x.
<a href="#">equal_range (page 117)</a>	Find the range within which all keys equal to specified key x.
<a href="#">count (page 119)</a>	Count the number of key/data pairs having specified key x.
<a href="#">upper_bound (page 120)</a>	Find the least key greater than x.
<a href="#">key_eq (page 122)</a>	Function to get key compare functor.
<a href="#">hash_funct (page 123)</a>	Function to get hash key generating functor.
<a href="#">value_comp (page 124)</a>	Function to get value compare functor.
<a href="#">key_comp (page 125)</a>	Function to get key compare functor.
<a href="#">operator= (page 126)</a>	Container content assignment operator.
<a href="#">operator[] (page 127)</a>	Retrieve data element by key.
<a href="#">swap (page 128)</a>	Swap content with container mp.
<a href="#">clear (page 129)</a>	Clear contents in this container.
<a href="#">operator== (page 130)</a>	Map content equality comparison operator.
<a href="#">operator!= (page 131)</a>	Container unequality comparison operator.

## Group

[Dbstl Container Classes \(page 29\)](#)

## db\_map

### Function Details

```
db_map(Db *dbp=NULL,  
       DbEnv *envp=NULL)
```

Create a `std::map/hash_map` equivalent associative container.

See the handle requirement in class details to pass correct database/environment handles.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

#### See Also

[db\\_container\(Db\\*, DbEnv\\*\)](#)

```
db_map(Db *dbp, DbEnv *envp, InputIterator first,  
       InputIterator last)
```

Iteration constructor.

Iterates between `first` and `last`, setting a copy of each of the sequence of elements as the content of the container object. Create a `std::map/hash_map` equivalent associative container. Insert a range of elements into the database. The range is `[first, last)`, which contains elements that can be converted to type `ddt` automatically. See the handle requirement in class details to pass correct database/environment handles. This function supports auto-commit.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

##### last

The open boundary of the range.

**first**

The closed boundary of the range.

**See Also**

[db\\_container\(Db\\*, DbEnv\\*\)](#)

```
db_map(const db_map< kdt, ddt, value_type_sub,  
        iterator > &x)
```

Copy constructor.

Create an database and insert all key/data pairs in x into this container. x's data members are not copied. This function supports auto-commit.

**Parameters****x**

The other container to initialize this container.

**See Also**

[db\\_container\(const db\\_container&\)](#)

**Class**

[db\\_map](#)

## ~db\_map

### Function Details

```
virtual ~db_map()
```

### Class

[db\\_map](#)

## insert

### Function Details

```
insert(const value_type &x)
```

Insert a single key/data pair if the key is not in the container.

#### Parameters

**x**

The key/data pair to insert.

#### Return Value

A pair P, if insert OK, i.e. the inserted key wasn't in the container, P.first will be the iterator sitting on the inserted key/data pair, and P.second is true; otherwise P.first is an invalid iterator and P.second is false.

```
iterator insert(iterator position,  
                const value_type &x)
```

Insert with hint position.

We ignore the hint position because Berkeley DB knows better where to insert.

#### Parameters

**position**

The hint position.

**x**

The key/data pair to insert.

#### Return Value

The iterator sitting on the inserted key/data pair, or an invalid iterator if the key was already in the container.

```
void insert(const db_map_base_iterator< kdt, realddt, ddt > &first,  
            const db_map_base_iterator< kdt, realddt,  
            ddt > &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
void insert(InputIterator first,
           InputIterator last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

## Group: Insert Functions

They have similar usage as their C++ STL equivalents.

Note that when secondary index is enabled, each [db\\_container](#) can create a [db\\_multimap](#) secondary container, but the insert function is not functional for secondary containers.

<http://www.cplusplus.com/reference/stl/map/insert/>

## Class

[db\\_map](#)

# begin

## Function Details

```
iterator begin(ReadModifyWriteOption rmw=
    ReadModifyWriteOption::no_read_modify_write(), bool readonly=false,
    BulkRetrievalOption bulkretrieval=
    BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true)
```

Begin a read-write or readonly iterator which sits on the first key/data pair of the database.

### Parameters

#### directdb\_get

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool)`;

#### readonly

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool)`;

#### rmw

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool)`;

#### bulkretrieval

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool)`;

### Return Value

The created iterator.

### See Also

[db\\_vector::begin](#) ([ReadModifyWriteOption](#) , bool, [BulkRetrievalOption](#) , bool)

```
const_iterator begin(BulkRetrievalOption bulkretrieval=
    BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true) const
```

Begin a read-only iterator.

### Parameters

#### directdb\_get

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool)`;

**bulkretrieval**

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

**Return Value**

The created const iterator.

**See Also**

[db\\_vector::begin](#) (ReadModifyWrite, bool, [BulkRetrievalOption](#) , bool);

**Group: Iterator Functions**

The parameters in begin functions of this group have identical meaning to thoes in [db\\_vector::begin](#) , refer to those functions for details.

[db\\_vector::begin\(\)](#)

**Class**

[db\\_map](#)

## end

### Function Details

```
iterator end()
```

Create an open boundary iterator.

#### Return Value

Returns an invalid iterator denoting the position after the last valid element of the container.

#### See Also

[db\\_vector::end\(\)](#)

```
const_iterator end() const
```

Create an open boundary iterator.

#### Return Value

Returns an invalid const iterator denoting the position after the last valid element of the container.

#### See Also

[db\\_vector::end\(\) const](#)

### Group: Iterator Functions

The parameters in begin functions of this group have identical meaning to thoes in [db\\_vector::begin](#) , refer to those functions for details.

[db\\_vector::begin\(\)](#)

### Class

[db\\_map](#)

## rbegin

### Function Details

```
reverse_iterator rbegin(ReadModifyWriteOption rmw=
    ReadModifyWriteOption::no_read_modify_write(), bool read_only=false,
    BulkRetrievalOption bulkretrieval=
    BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true)
```

Begin a read-write or readonly reverse iterator which sits on the first key/data pair of the database.

#### Parameters

##### directdb\_get

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

##### read\_only

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

##### rmw

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

##### bulkretrieval

Same as that of `db_vector::begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

#### Return Value

The created iterator.

#### See Also

[db\\_vector::begin \(ReadModifyWriteOption , bool, BulkRetrievalOption , bool\)](#)

[db\\_vector::begin \(ReadModifyWrite, bool, BulkRetrievalOption , bool\);](#)

```
const_reverse_iterator rbegin(BulkRetrievalOption bulkretrieval=
    BulkRetrievalOption::no_bulk_retrieval(),
    bool directdb_get=true) const
```

Begin a read-only reverse iterator.

#### Parameters

##### directdb\_get

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

**bulkretrieval**

Same as that of `begin(ReadModifyWrite, bool, BulkRetrievalOption, bool);`

**Return Value**

The created const iterator.

**See Also**

[db\\_vector::begin](#) (ReadModifyWrite, bool, [BulkRetrievalOption](#) , bool);

**Group: Iterator Functions**

The parameters in begin functions of this group have identical meaning to thoes in [db\\_vector::begin](#) , refer to those functions for details.

[db\\_vector::begin\(\)](#)

**Class**

[db\\_map](#)

# rend

## Function Details

```
reverse_iterator rend()
```

Create an open boundary iterator.

### Return Value

Returns an invalid iterator denoting the position before the first valid element of the container.

### See Also

[db\\_vector::rend\(\)](#)

```
const_reverse_iterator rend() const
```

Create an open boundary iterator.

### Return Value

Returns an invalid const iterator denoting the position before the first valid element of the container.

### See Also

[db\\_vector::rend\(\) const](#)

## Group: Iterator Functions

The parameters in begin functions of this group have identical meaning to thoes in [db\\_vector::begin](#) , refer to those functions for details.

[db\\_vector::begin\(\)](#)

## Class

[db\\_map](#)

## is\_hash

### Function Details

```
bool is_hash() const
```

Get container category.

Determines whether this container object is a `std::map<>` equivalent (when returns false) or that of `hash_map<>` class (when returns true). This method is not in `stl`, but it may be called by users because some operations are not supported by both `type(map/hash_map)` of containers, you need to call this function to distinguish the two types. `dbstl` will not stop you from calling the wrong methods of this class.

### Return Value

Returns true if this container is a hash container based on a Berkeley DB hash database; returns false if it is based on a Berkeley DB btree database.

### Group: Metadata Functions

These functions return metadata about the container.

### Class

[db\\_map](#)

## bucket\_count

### Function Details

```
size_type bucket_count() const
```

Only for `std::hash_map`, return number of hash bucket in use.

This function supports auto-commit.

### Return Value

The number of hash buckets of the database.

### Group: Metadata Functions

These functions return metadata about the container.

### Class

[db\\_map](#)

## size

### Function Details

```
size_type size(bool accurate=true) const
```

This function supports auto-commit.

#### Parameters

#### accurate

This function uses database's statistics to get the number of key/data pairs. The statistics mechanism will either scan the whole database to find the accurate number or use the number of last accurate scanning, and thus much faster. If there are millions of key/data pairs, the scanning can take some while, so in that case you may want to set the "accurate" parameter to false.

#### Return Value

Return the number of key/data pairs in the container.

### Group: Metadata Functions

These functions return metadata about the container.

### Class

[db\\_map](#)

## max\_size

### Function Details

```
size_type max_size() const
```

Get max size.

The returned size is not the actual limit of database. See the Berkeley DB limits to get real max size.

#### Return Value

A meaningless huge number.

#### See Also

[db\\_vector::max\\_size\(\)](#)

### Group: Metadata Functions

These functions return metadata about the container.

### Class

[db\\_map](#)

## empty

### Function Details

```
bool empty() const
```

Returns whether this container is empty.

This function supports auto-commit.

#### Return Value

True if empty, false otherwise.

### Group: Metadata Functions

These functions return metadata about the container.

### Class

[db\\_map](#)

## erase

### Function Details

```
void erase(iterator pos)
```

Erase a key/data pair at specified position.

#### Parameters

##### pos

An valid iterator of this container to erase.

```
size_type erase(const key_type &x)
```

Erase elements by key.

All key/data pairs with specified key x will be removed from underlying database. This function supports auto-commit.

#### Parameters

##### x

The key to remove from the container.

#### Return Value

The number of key/data pairs removed.

```
void erase(iterator first,  
           iterator last)
```

Range erase.

Erase all key/data pairs within the valid range [first, last).

#### Parameters

##### last

The open boundary of the range.

##### first

The closed boundary of the range.

## **Group: Erase Functions**

<http://www.cplusplus.com/reference/stl/map/erase/>

## **Class**

[db\\_map](#)

## find

### Function Details

```
const_iterator find(const key_type &x) const
```

Find the key/data pair with specified key x.

#### Parameters

**x**

The target key to find.

#### Return Value

The valid const iterator sitting on the key x, or an invalid one.

#### See Also

<http://www.cplusplus.com/reference/stl/map/find/>

```
iterator find(const key_type &x,  
             bool readonly=false)
```

Find the key/data pair with specified key x.

#### Parameters

**x**

The target key to find.

**readonly**

Whether the returned iterator is readonly.

#### Return Value

The valid iterator sitting on the key x, or an invalid one.

#### See Also

<http://www.cplusplus.com/reference/stl/map/find/>

### Group: Searching Functions

The following functions are returning iterators, and they by default return read-write iterators.

If you intend to use the returned iterator only to read, you should call the const version of each function using a const reference to this container. Using const iterators can potentially promote concurrency a lot. You can also set the readonly parameter to each non-const version of the functions to true if you don't use the returned iterator to write, which also promotes concurrency and overall performance.

## **Class**

[db\\_map](#)

## lower\_bound

### Function Details

```
const_iterator lower_bound(const key_type &x) const
```

Find the greatest key less than or equal to x.

#### Parameters

x

The target key to find.

#### Return Value

The valid const iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/map/lower\\_bound/](http://www.cplusplus.com/reference/stl/map/lower_bound/)

```
iterator lower_bound(const key_type &x,  
                    bool readonly=false)
```

Find the greatest key less than or equal to x.

#### Parameters

x

The target key to find.

#### readonly

Whether the returned iterator is readonly.

#### Return Value

The valid iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/map/lower\\_bound/](http://www.cplusplus.com/reference/stl/map/lower_bound/)

### Group: Searching Functions

The following functions are returning iterators, and they by default return read-write iterators.

If you intend to use the returned iterator only to read, you should call the const version of each function using a const reference to this container. Using const iterators can potentially promote concurrency a lot. You can also set the readonly parameter to each non-const version of the functions to true if you don't use the returned iterator to write, which also promotes concurrency and overall performance.

## **Class**

[db\\_map](#)

## equal\_range

### Function Details

```
equal_range(const key_type &x) const
```

Find the range within which all keys equal to specified key x.

#### Parameters

**x**

The target key to find.

#### Return Value

The range [first, last).

#### See Also

[http://www.cplusplus.com/reference/stl/map/equal\\_range/](http://www.cplusplus.com/reference/stl/map/equal_range/)

```
equal_range(const key_type &x,  
            bool readonly=false)
```

Find the range within which all keys equal to specified key x.

#### Parameters

**x**

The target key to find.

**readonly**

Whether the returned iterator is readonly.

#### Return Value

The range [first, last).

#### See Also

[http://www.cplusplus.com/reference/stl/map/equal\\_range/](http://www.cplusplus.com/reference/stl/map/equal_range/)

### Group: Searching Functions

The following functions are returning iterators, and they by default return read-write iterators.

If you intend to use the returned iterator only to read, you should call the const version of each function using a const reference to this container. Using const iterators can potentially promote concurrency a lot. You can also set the readonly parameter to each non-const version of the functions to true if you don't use the returned iterator to write, which also promotes concurrency and overall performance.

## **Class**

[db\\_map](#)

## count

### Function Details

```
size_type count(const key_type &x) const
```

Count the number of key/data pairs having specified key x.

#### Parameters

**x**

The key to count.

#### Return Value

The number of key/data pairs having x as key within the container.

#### See Also

<http://www.cplusplus.com/reference/stl/map/count/>

### Group: Searching Functions

The following functions are returning iterators, and they by default return read-write iterators.

If you intend to use the returned iterator only to read, you should call the const version of each function using a const reference to this container. Using const iterators can potentially promote concurrency a lot. You can also set the readonly parameter to each non-const version of the functions to true if you don't use the returned iterator to write, which also promotes concurrency and overall performance.

### Class

[db\\_map](#)

## upper\_bound

### Function Details

```
const_iterator upper_bound(const key_type &x) const
```

Find the least key greater than x.

#### Parameters

**x**

The target key to find.

#### Return Value

The valid iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/map/upper\\_bound/](http://www.cplusplus.com/reference/stl/map/upper_bound/)

```
iterator upper_bound(const key_type &x,  
                    bool readonly=false)
```

Find the least key greater than x.

#### Parameters

**x**

The target key to find.

**readonly**

Whether the returned iterator is readonly.

#### Return Value

The valid iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/map/upper\\_bound/](http://www.cplusplus.com/reference/stl/map/upper_bound/)

### Group: Searching Functions

The following functions are returning iterators, and they by default return read-write iterators.

If you intend to use the returned iterator only to read, you should call the const version of each function using a const reference to this container. Using const iterators can potentially promote concurrency a lot. You can also set the readonly parameter to each non-const version of the functions to true if you don't use the returned iterator to write, which also promotes concurrency and overall performance.

## **Class**

[db\\_map](#)

## key\_eq

### Function Details

```
key_equal key_eq() const
```

Function to get key compare functor.

Used when this container is a hash\_map, hash\_multimap, hash\_set or hash\_multiset equivalent.

#### Return Value

[key\\_equal](#) type of compare functor.

#### See Also

[http://www.sgi.com/tech/stl/hash\\_map.html](http://www.sgi.com/tech/stl/hash_map.html)

### Class

[db\\_map](#)

## hash\_funct

### Function Details

```
hasher hash_funct() const
```

Function to get hash key generating functor.

Used when this container is a hash\_map, hash\_multimap, hash\_set or hash\_multiset equivalent.

#### Return Value

The hash key generating functor.

#### See Also

[http://www.sgi.com/tech/stl/hash\\_map.html](http://www.sgi.com/tech/stl/hash_map.html)

### Class

[db\\_map](#)

## value\_comp

### Function Details

```
value_compare value_comp() const
```

Function to get value compare functor.

Used when this container is a `std::map`, `std::multimap`, `std::set` or `std::multiset` equivalent.

#### Return Value

The value compare functor.

#### See Also

[http://www.cplusplus.com/reference/stl/map/value\\_comp/](http://www.cplusplus.com/reference/stl/map/value_comp/)

### Class

[db\\_map](#)

## key\_comp

### Function Details

```
key_compare key_comp() const
```

Function to get key compare functor.

Used when this container is a `std::map`, `std::multimap`, `std::set` or `std::multiset` equivalent.

#### Return Value

The key compare functor.

#### See Also

[http://www.cplusplus.com/reference/stl/map/key\\_comp/](http://www.cplusplus.com/reference/stl/map/key_comp/)

### Class

[db\\_map](#)

## operator=

### Function Details

```
const self& operator=(const self &x)
```

Container content assignment operator.

This function supports auto-commit.

### Parameters

**x**

The other container whose key/data pairs will be inserted into this container. Old content in this containers are discarded.

### See Also

<http://www.cplusplus.com/reference/stl/map/operator=>

### Class

[db\\_map](#)

## operator[]

### Function Details

```
data_type_wrap operator[](const key_type &x)
```

Retrieve data element by key.

This function returns an reference to the underlying data element of the specified key x. The returned object can be used to read or write the data element of the key/data pair. Do use a data\_type\_wrap of [db\\_map](#) or value\_type::second\_type(they are the same) type of variable to hold the return value of this function.

#### Parameters

**x**

The target key to get value from.

#### Return Value

Data element reference.

```
const ddt operator[](const key_type &x) const
```

Retrieve data element by key.

This function returns the value of the underlying data element of specified key x. You can only read the element, but unable to update the element via the return value of this function. And you need to use the container's const reference to call this method.

#### Parameters

**x**

The target key to get value from.

#### Return Value

Data element, read only, can't be used to modify it.

### Class

[db\\_map](#)

## swap

### Function Details

```
void swap(db_map< kdt, ddt, value_type_sub > &mp,  
         bool b_truncate=true)
```

Swap content with container mp.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

See [db\\_vector::swap\(\)](#) for details.

#### **mp**

The container to swap content with.

#### See Also

<http://www.cplusplus.com/reference/stl/map/swap/> [db\\_vector::clear\(\)](#)

### Class

[db\\_map](#)

## clear

### Function Details

```
void clear(bool b_truncate=true)
```

Clear contents in this container.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

See [db\\_vector::clear\(bool\)](#) for details.

#### See Also

[db\\_vector::clear\(bool\)](#)

#### Class

[db\\_map](#)

## operator==

### Function Details

```
bool operator==(const db_map< kdt, ddt,  
                value_type_sub > &m2) const
```

Map content equality comparison operator.

This function does not rely on key order. For a set of keys S1 in this container and another set of keys S2 of container m2, if set S1 contains S2 and S2 contains S1 (S1 equals to S2) and each data element of a key K in S1 from this container equals the data element of K in m2, the two db\_map<> containers equal. Otherwise they are not equal.

### Parameters

**m2**

The other container to compare against.

### Return Value

Returns true if they have equal content, false otherwise.

### Class

[db\\_map](#)

## operator!=

### Function Details

```
bool operator!=(const db_map< kdt, ddt,  
                value_type_sub > &m2) const
```

Container unequality comparison operator.

#### Parameters

**m2**

The container to compare against.

#### Return Value

Returns false if equal, true otherwise.

### Class

[db\\_map](#)

---

## Chapter 6. Db\_multimap

This class is the combination of `std::multimap` and `hash_multimap`.

By setting database handles as `DB_BTREE` or `DB_HASH` type respectively, you will be using an equivalent of `std::multimap` or `hash_multimap` respectively. Database(`dbp`) and environment(`penv`) handle requirement: The `dbp` handle must meet the following requirement: 1. Database type should be `DB_BTREE` or `DB_HASH`. 2. Either `DB_DUP` or `DB_DUPSORT` flag must be set. Note that so far Berkeley DB does not allow `DB_DUPSORT` be set and the database is storing identical key/data pairs, i.e. we can't store two (1, 2), (1, 2) pairs into a database D with `DB_DUPSORT` flag set, but only can do so with `DB_DUP` flag set; But we can store a (1, 2) pair and a (1, 3) pair into D with `DB_DUPSORT` flag set. So if your data set allows `DB_DUPSORT` flag, you should set it to gain a lot of performance promotion. 3. No `DB_RECNUM` flag set. 4. No `DB_TRUNCATE` specified in database open flags. 5. `DB_THREAD` must be set if you are sharing the database handle across multiple threads directly, or indirectly by sharing the container object across multiple threads.

### See Also

[db\\_container](#) [db\\_map](#)

### Class Template Parameters

#### **kdt**

The key data type.

#### **ddt**

The data data type. [db\\_multimap](#) stores key/data pairs.

#### **value\_type\_sub**

Do not specify anything if `ddt` type is a class/struct type; Otherwise, specify `ElementHolder<ddt>` to it.

#### **iterator\_t**

Never specify anything to this type parameter. It is only used internally.

### Public Members

Member	Description
<a href="#">insert (page 134)</a>	Range insertion.
<a href="#">erase (page 136)</a>	Erase elements by key.
<a href="#">equal_range (page 138)</a>	Find the range within which all keys equal to specified key x.
<a href="#">equal_range_N (page 140)</a>	Find equal range and number of key/data pairs in the range.

---

Member	Description
<a href="#">count (page 142)</a>	Count the number of key/data pairs having specified key x.
<a href="#">upper_bound (page 143)</a>	Find the least key greater than x.
<a href="#">db_multimap (page 145)</a>	Constructor.
<a href="#">~db_multimap (page 147)</a>	
<a href="#">operator= (page 148)</a>	Container content assignment operator.
<a href="#">swap (page 149)</a>	Swap content with another multimap container.
<a href="#">operator== (page 150)</a>	Returns whether the two containers have identical content.
<a href="#">operator!= (page 151)</a>	Container inequality comparison operator.

**Group**

[Dbstl Container Classes \(page 29\)](#)

# insert

## Function Details

```
void insert(InputIterator first,
            InputIterator last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
void insert(const_iterator &first,
            const_iterator &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
iterator insert(const value_type &x)
```

Insert a single key/data pair if the key is not in the container.

### Parameters

**x**

The key/data pair to insert.

### **Return Value**

A pair P, if insert OK, i.e. the inserted key wasn't in the container, P.first will be the iterator sitting on the inserted key/data pair, and P.second is true; otherwise P.first is an invalid iterator and P.second is false.

### **Group: Insert Functions**

<http://www.cplusplus.com/reference/stl/multimap/insert/>

### **Class**

[db\\_multimap](#)

## erase

### Function Details

```
size_type erase(const key_type &x)
```

Erase elements by key.

All key/data pairs with specified key x will be removed from underlying database. This function supports auto-commit.

#### Parameters

**x**

The key to remove from the container.

#### Return Value

The number of key/data pairs removed.

```
void erase(iterator pos)
```

Erase a key/data pair at specified position.

#### Parameters

**pos**

An valid iterator of this container to erase.

```
void erase(iterator first,  
           iterator last)
```

Range erase.

Erase all key/data pairs within the valid range [first, last).

#### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

## **Group: Erase Functions**

<http://www.cplusplus.com/reference/stl/multimap/erase/>

## **Class**

[db\\_multimap](#)

## equal\_range

### Function Details

```
equal_range(const key_type &x) const
```

Find the range within which all keys equal to specified key x.

#### Parameters

**x**

The target key to find.

#### Return Value

The range [first, last).

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/equal\\_range/](http://www.cplusplus.com/reference/stl/multimap/equal_range/)

```
equal_range(const key_type &x,  
            bool readonly=false)
```

Find the range within which all keys equal to specified key x.

#### Parameters

**x**

The target key to find.

**readonly**

Whether the returned iterator is readonly.

#### Return Value

The range [first, last).

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/equal\\_range/](http://www.cplusplus.com/reference/stl/multimap/equal_range/)

### Group: Searching Functions

See of db\_map's searching functions group for details about iterator, function version and parameters.

[db\\_map](#)

**Class**

[db\\_multimap](#)

## equal\_range\_N

### Function Details

```
equal_range_N(const key_type &x,  
              size_t &nelem) const
```

Find equal range and number of key/data pairs in the range.

This function also returns the number of elements within the returned range via the out parameter nelem.

#### Parameters

**x**

The target key to find.

**nelem**

The output parameter to take back the number of key/data pair in the returned range.

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/equal\\_range/](http://www.cplusplus.com/reference/stl/multimap/equal_range/)

```
equal_range_N(const key_type &x, size_t &nelem,  
              bool readonly=false)
```

Find equal range and number of key/data pairs in the range.

This function also returns the number of elements within the returned range via the out parameter nelem.

#### Parameters

**x**

The target key to find.

**nelem**

The output parameter to take back the number of key/data pair in the returned range.

**readonly**

Whether the returned iterator is readonly.

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/equal\\_range/](http://www.cplusplus.com/reference/stl/multimap/equal_range/)

## **Group: Searching Functions**

See of `db_map`'s searching functions group for details about iterator, function version and parameters.

[db\\_map](#)

## **Class**

[db\\_multimap](#)

## count

### Function Details

```
size_type count(const key_type &x) const
```

Count the number of key/data pairs having specified key x.

#### Parameters

**x**

The key to count.

#### Return Value

The number of key/data pairs having x as key within the container.

#### See Also

<http://www.cplusplus.com/reference/stl/multimap/count/>

### Group: Searching Functions

See of db\_map's searching functions group for details about iterator, function version and parameters.

[db\\_map](#)

### Class

[db\\_multimap](#)

## upper\_bound

### Function Details

```
const_iterator upper_bound(const key_type &x) const
```

Find the least key greater than x.

#### Parameters

**x**

The target key to find.

#### Return Value

The valid iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/upper\\_bound/](http://www.cplusplus.com/reference/stl/multimap/upper_bound/)

```
iterator upper_bound(const key_type &x,  
                    bool readonly=false)
```

Find the least key greater than x.

#### Parameters

**x**

The target key to find.

**readonly**

Whether the returned iterator is readonly.

#### Return Value

The valid iterator sitting on the key, or an invalid one.

#### See Also

[http://www.cplusplus.com/reference/stl/multimap/upper\\_bound/](http://www.cplusplus.com/reference/stl/multimap/upper_bound/)

### Group: Searching Functions

See of db\_map's searching functions group for details about iterator, function version and parameters.

[db\\_map](#)

**Class**

[db\\_multimap](#)

## db\_multimap

### Function Details

```
db_multimap(Db *dbp=NULL,  
            DbEnv *envp=NULL)
```

Constructor.

See class detail for handle requirement.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

#### See Also

[db\\_map::db\\_map\(Db\\*, DbEnv\\*\)](#) [db\\_vector::db\\_vector\(Db\\*, DbEnv\\*\)](#)

```
db_multimap(Db *dbp, DbEnv *envp, InputIterator first,  
            InputIterator last)
```

Iteration constructor.

Iterates between first and last, setting a copy of each of the sequence of elements as the content of the container object. This function supports auto-commit. See class detail for handle requirement.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

##### last

The open boundary of the range.

##### first

The closed boundary of the range.

**See Also**

[db\\_map::db\\_map\(Db\\*, DbEnv\\*, InputIterator, InputIterator\)](#) [db\\_vector::db\\_vector\(Db\\*, DbEnv\\*\)](#)

```
db_multimap(const self &x)
```

Copy constructor.

Create an database and insert all key/data pairs in x into this container. x's data members are not copied. This function supports auto-commit.

**Parameters**

**x**

The other container to initialize this container.

**See Also**

[db\\_container\(const db\\_container&\)](#) [db\\_map\(const db\\_map&\)](#)

**Class**

[db\\_multimap](#)

## **~db\_multimap**

### **Function Details**

```
virtual ~db_multimap()
```

### **Class**

[db\\_multimap](#)

## operator=

### Function Details

```
const self& operator=(const self &x)
```

Container content assignment operator.

This function supports auto-commit.

### Parameters

**x**

The other container whose key/data pairs will be inserted into this container. Old content in this containers are discarded.

### See Also

<http://www.cplusplus.com/reference/stl/multimap/operator=>

### Class

[db\\_multimap](#)

## swap

### Function Details

```
void swap(db_multimap< kdt, ddt, value_type_sub > &mp,  
          bool b_truncate=true)
```

Swap content with another multimap container.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

See [db\\_map::swap\(\)](#) for details.

#### **mp**

The other container to swap content with.

#### See Also

[db\\_vector::clear\(\)](#)

### Class

[db\\_multimap](#)

## operator==

### Function Details

```
bool operator==(const db_multimap< kdt, ddt,  
                value_type_sub > &m2) const
```

Returns whether the two containers have identical content.

This function does not rely on key order. For a set of keys S1 in this container and another set of keys S2 of container m2, if set S1 contains S2 and S2 contains S1 (S1 equals to S2) and each set of data elements of any key K in S1 from this container equals the set of data elements of K in m2, the two db\_multimap<> containers equal. Otherwise they are not equal. Data element set comparison does not rely on order either.

#### Parameters

**m2**

The other container to compare against.

#### Return Value

Returns true if they are equal, false otherwise.

### Class

[db\\_multimap](#)

## operator!=

### Function Details

```
bool operator!=(const db_multimap< kdt, ddt,  
                value_type_sub > &m2) const
```

Container unequality comparison operator.

#### Parameters

**m2**

The container to compare against.

#### Return Value

Returns false if equal, true otherwise.

### Class

[db\\_multimap](#)

---

## Chapter 7. Db\_set

This class is the combination of `std::set` and `hash_set`.

By setting database handles of `DB_BTREE` or `DB_HASH` type, you will be using the equivalent of `std::set` or `hash_set`. This container stores the key in the key element of a key/data pair in the underlying database, but doesn't store anything in the data element. Database and environment handle requirement: The same as that of [db\\_map](#) .

### See Also

[db\\_map](#) [db\\_container](#)

### Class Template Parameters

#### **kdt**

The key data type.

#### **value\_type\_sub**

If `kdt` is a class/struct type, do not specify anything in this parameter; Otherwise specify `ElementHolder<kdt>`.

### Public Members

Member	Description
<a href="#">db_set</a> (page 153)	Create a <code>std::set</code> / <code>hash_set</code> equivalent associative container.
<a href="#">~db_set</a> (page 155)	
<a href="#">insert</a> (page 156)	Insert a single key/data pair if the key is not in the container.
<a href="#">operator=</a> (page 158)	Container content assignment operator.
<a href="#">value_comp</a> (page 159)	Get value comparison functor.
<a href="#">swap</a> (page 160)	Swap content with another container.
<a href="#">operator==</a> (page 161)	Set content equality comparison operator.
<a href="#">operator!=</a> (page 162)	Inequality comparison operator.

### Group

[Dbstl Container Classes](#) (page 29)

## db\_set

### Function Details

```
db_set(Db *dbp=NULL,  
       DbEnv *envp=NULL)
```

Create a `std::set/hash_set` equivalent associative container.

See the handle requirement in class details to pass correct database/environment handles.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

#### See Also

[db\\_map\(Db\\*, DbEnv\\*\)](#) [db\\_container\(Db\\*, DbEnv\\*\)](#)

```
db_set(Db *dbp, DbEnv *envp, InputIterator first,  
       InputIterator last)
```

Iteration constructor.

Iterates between `first` and `last`, copying each of the elements in the range into this container. Create a `std::set/hash_set` equivalent associative container. Insert a range of elements into the database. The range is `[first, last)`, which contains elements that can be converted to type `dbt` automatically. This function supports auto-commit. See the handle requirement in class details to pass correct database/environment handles.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

##### last

The open boundary of the range.

**first**

The closed boundary of the range.

**See Also**

[db\\_map\(Db\\*, DbEnv\\*, InputIterator, InputIterator\)](#)

```
db_set(const self &x)
```

Copy constructor.

Create a database and insert all key/data pairs in x into this container. x's data members are not copied. This function supports auto-commit.

**Parameters****x**

The source container to initialize this container.

**See Also**

[db\\_map\(const db\\_map&\) db\\_container\(const db\\_container&\)](#)

**Class**

[db\\_set](#)

## ~db\_set

### Function Details

```
virtual ~db_set()
```

### Class

[db\\_set](#)

## insert

### Function Details

```
insert(const value_type &x)
```

Insert a single key/data pair if the key is not in the container.

#### Parameters

**x**

The key/data pair to insert.

#### Return Value

A pair P, if insert OK, i.e. the inserted key wasn't in the container, P.first will be the iterator positioned on the inserted key/data pair, and P.second is true; otherwise P.first is an invalid iterator equal to that returned by [end\(\)](#) and P.second is false.

```
void insert(const_iterator &first,  
            const_iterator &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

#### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
void insert(iterator &first,  
            iterator &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

#### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
iterator insert(iterator position,
               const value_type &x)
```

Insert with hint position.

We ignore the hint position because Berkeley DB knows better where to insert.

**Parameters****position**

The hint position.

**x**

The key/data pair to insert.

**Return Value**

The iterator positioned on the inserted key/data pair, or an invalid iterator if the key was already in the container.

```
void insert(InputIterator first,
           InputIterator last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

**Parameters****last**

The open boundary of the range.

**first**

The closed boundary of the range.

**Group: Insert Functions**

<http://www.cplusplus.com/reference/stl/set/insert/>

**Class**

[db\\_set](#)

## operator=

### Function Details

```
const self& operator=(const self &x)
```

Container content assignment operator.

This function supports auto-commit.

#### Parameters

**x**

The source container whose key/data pairs will be inserted into the target container. Old content in the target container is discarded.

#### Return Value

The container x's reference.

#### See Also

http://www.cplusplus.com/reference/stl/set/operator=/

### Class

[db\\_set](#)

## value\_comp

### Function Details

```
value_compare value_comp() const
```

Get value comparison functor.

#### Return Value

The value comparison functor.

#### See Also

[http://www.cplusplus.com/reference/stl/set/value\\_comp/](http://www.cplusplus.com/reference/stl/set/value_comp/)

### Class

[db\\_set](#)

## swap

### Function Details

```
void swap(db_set< kdt, value_type_sub > &mp,  
          bool b_truncate=true)
```

Swap content with another container.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

See [db\\_vector::swap](#) 's b\_truncate parameter for details.

#### **mp**

The container to swap content with.

#### See Also

[db\\_map::swap\(\)](#) [db\\_vector::clear\(\)](#)

### Class

[db\\_set](#)

## operator==

### Function Details

```
bool operator==(const db_set< kdt,  
                value_type_sub > &m2) const
```

Set content equality comparison operator.

Return if the two containers have identical content. This function does not rely on key order. Two sets A and B are equal if and only if A contains B and B contains A.

### Parameters

**m2**

The container to compare against.

### Return Value

Returns true if the two containers are equal, false otherwise.

### Class

[db\\_set](#)

## operator!=

### Function Details

```
bool operator!=(const db_set< kdt,  
                value_type_sub > &m2) const
```

Inequality comparison operator.

### Class

[db\\_set](#)

---

## Chapter 8. Db\_multiset

This class is the combination of `std::multiset` and `hash_multiset`.

By setting database handles of `DB_BTREE` or `DB_HASH` type respectively, you will be using the equivalent of `std::multiset` or `hash_multiset` respectively. This container stores the key in the key element of a key/data pair in the underlying database, but doesn't store anything in the data element. Database and environment handle requirement: The requirement to these handles is the same as that to [db\\_multimap](#) .

### See Also

[db\\_multimap](#) [db\\_map](#) [db\\_container](#) [db\\_set](#)

### Class Template Parameters

#### **kdt**

The key data type.

#### **value\_type\_sub**

If `kdt` is a class/struct type, do not specify anything in this parameter; Otherwise specify `ElementHolder<kdt>`.

### Public Members

Member	Description
<a href="#">db_multiset</a> (page 164)	Create a <code>std::multiset</code> / <code>hash_multiset</code> equivalent associative container.
<a href="#">~db_multiset</a> (page 166)	
<a href="#">insert</a> (page 167)	Insert a single key if the key is not in the container.
<a href="#">erase</a> (page 170)	Erase elements by key.
<a href="#">operator=</a> (page 172)	Container content assignment operator.
<a href="#">swap</a> (page 173)	Swap content with another container.
<a href="#">operator==</a> (page 174)	Container content equality compare operator.
<a href="#">operator!=</a> (page 175)	Inequality comparison operator.

### Group

[Dbstl Container Classes](#) (page 29)

## db\_multiset

### Function Details

```
db_multiset(Db *dbp=NULL,  
            DbEnv *envp=NULL)
```

Create a `std::multiset/hash_multiset` equivalent associative container.

See the handle requirement in class details to pass correct database/environment handles.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

#### See Also

[db\\_multimap\(Db\\*, DbEnv\\*\)](#)

```
db_multiset(Db *dbp, DbEnv *envp, InputIterator first,  
            InputIterator last)
```

Iteration constructor.

Iterates between `first` and `last`, copying each of the elements in the range into this container. Create a `std::multi/hash_multiset` equivalent associative container. Insert a range of elements into the database. The range is `[first, last)`, which contains elements that can be converted to type `dbt` automatically. This function supports auto-commit. See the handle requirement in class details to pass correct database/environment handles.

#### Parameters

##### dbp

The database handle.

##### envp

The database environment handle.

##### last

The open boundary of the range.

**first**

The closed boundary of the range.

**See Also**

[db\\_multimap\(Db\\*, DbEnv\\*, InputIterator, InputIterator\)](#)

```
db_multiset(const self &x)
```

Copy constructor.

Create a database and insert all key/data pairs in x into this container. x's data members are not copied. This function supports auto-commit.

**Parameters****x**

The source container to initialize this container.

**See Also**

[db\\_multimap\(const db\\_multimap&\)](#) [db\\_container\(const db\\_container&\)](#)

**Class**

[db\\_multiset](#)

## **~db\_multiset**

### **Function Details**

```
virtual ~db_multiset()
```

### **Class**

[db\\_multiset](#)

## insert

### Function Details

```
iterator insert(const value_type &x)
```

Insert a single key if the key is not in the container.

#### Parameters

**x**

The key to insert.

#### Return Value

An iterator positioned on the newly inserted key. If the key `x` already exists, an invalid iterator equal to that returned by `end()` function is returned.

```
iterator insert(iterator position,  
               const value_type &x)
```

Insert a single key with hint if the key is not in the container.

The hint position is ignored because Berkeley DB controls where to insert the key.

#### Parameters

**x**

The key to insert.

**position**

The hint insert position, ignored.

#### Return Value

An iterator positioned on the newly inserted key. If the key `x` already exists, an invalid iterator equal to that returned by `end()` function is returned.

```
void insert(InputIterator first,  
           InputIterator last)
```

Range insertion.

Insert a range `[first, last)` of key/data pairs into this container.

**Parameters****last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
void insert(db_set_iterator< kdt, value_type_sub > &first,
            db_set_iterator< kdt,
            value_type_sub > &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

**Parameters****last**

The open boundary of the range.

**first**

The closed boundary of the range.

```
void insert(db_set_base_iterator< kdt > &first,
            db_set_base_iterator< kdt > &last)
```

Range insertion.

Insert a range [first, last) of key/data pairs into this container.

**Parameters****last**

The open boundary of the range.

**first**

The closed boundary of the range.

**Group: Insert Functions**

<http://www.cplusplus.com/reference/stl/multiset/insert/>

## **Class**

`db_multiset`

## erase

### Function Details

```
size_type erase(const key_type &x)
```

Erase elements by key.

All key/data pairs with specified key x will be removed from the underlying database. This function supports auto-commit.

#### Parameters

**x**

The key to remove from the container.

#### Return Value

The number of key/data pairs removed.

```
void erase(iterator pos)
```

Erase a key/data pair at specified position.

#### Parameters

**pos**

A valid iterator of this container to erase.

```
void erase(iterator first,  
           iterator last)
```

Range erase.

Erase all key/data pairs within the valid range [first, last).

#### Parameters

**last**

The open boundary of the range.

**first**

The closed boundary of the range.

## **Group: Erase Functions**

<http://www.cplusplus.com/reference/stl/multiset/erase/>

## **Class**

[db\\_multiset](#)

## operator=

### Function Details

```
const self& operator=(const self &x)
```

Container content assignment operator.

This function supports auto-commit.

#### Parameters

**x**

The source container whose key/data pairs will be inserted into the target container. Old content in the target container is discarded.

#### Return Value

The container x's reference.

#### See Also

<http://www.cplusplus.com/reference/stl/multiset/operator=>

### Class

[db\\_multiset](#)

## swap

### Function Details

```
void swap(db_multiset< kdt, value_type_sub > &mp,  
          bool b_truncate=true)
```

Swap content with another container.

This function supports auto-commit.

#### Parameters

#### **b\_truncate**

See [db\\_multimap::swap\(\)](#) for details.

#### **mp**

The container to swap content with.

#### See Also

[db\\_map::swap\(\)](#) [db\\_vector::clear\(\)](#)

### Class

[db\\_multiset](#)

## operator==

### Function Details

```
bool operator==(const self &m2) const
```

Container content equality compare operator.

This function does not rely on key order. Two sets A and B are equal if and only if for each and every key K having n occurrences in A, K has n occurrences in B, and for each and every key K` having N occurrences in B, K` has n occurrences in A.

### Parameters

**m2**

The container to compare against.

### Return Value

Returns true if the two containers are equal, false otherwise.

### Class

[db\\_multiset](#)

## operator!=

### Function Details

```
bool operator!=(const self &m2) const
```

Inequality comparison operator.

### Class

[db\\_multiset](#)

---

## Chapter 9. Dbstl Iterator Classes

Common information for all dbstl iterators:.

1. Each instance of a dbstl iterator uniquely owns a Berkeley DB cursor, so that the key/data pair it currently sits on is always valid before it moves elsewhere. It also caches the current key/data pair values in order for member functions like `operator*` / `operator->` to work properly, but caching is not compatible with standard C++ Stl behavior --- the C++ standard requires the iterator refer to a shared piece of memory where the data is stored, thus two iterators of the same container sitting on the same element should point to the same memory location, which is false for dbstl iterators.
2. There are some functions common to each child class of this class which have identical behaviors, so we will document them here.

This class is the base class for all dbstl iterators, there is no much to say about this class itself, and users are not supposed to directly use this class at all. So we will talk about some common functions of dbstl iterators in this section.

### See Also

[db\\_vector\\_base\\_iterator](#) [db\\_vector\\_iterator](#) [db\\_map\\_base\\_iterator](#) [db\\_map\\_iterator](#)  
[db\\_set\\_base\\_iterator](#) [db\\_set\\_iterator](#)

### Public Members

Member	Description
<a href="#">db_base_iterator</a>	<a href="#">db_base_iterator</a>
<a href="#">db_reverse_iterator</a>	<a href="#">db_reverse_iterator</a>
<a href="#">db_map_iterator</a>	<a href="#">db_map_iterator</a>
<a href="#">Iterator classes for db_map and db_multimap.</a>	Iterator classes for <a href="#">db_map</a> and <a href="#">db_multimap</a> .
<a href="#">Iterator classes for db_set and db_multiset.</a>	Iterator classes for <a href="#">db_set</a> and <a href="#">db_multiset</a> .
<a href="#">Iterator classes for db_vector.</a>	Iterator classes for <a href="#">db_vector</a> .

### Group

None

---

## Chapter 10. Db\_base\_iterator

### Public Members

Member	Description
<a href="#">refresh (page 178)</a>	Read data from underlying database via its cursor, and update its cached value.
<a href="#">close_cursor (page 179)</a>	Close its cursor.
<a href="#">set_bulk_buffer (page 180)</a>	Call this function to modify bulk buffer size.
<a href="#">get_bulk_bufsize (page 181)</a>	Return current bulk buffer size.
<a href="#">db_base_iterator (page 182)</a>	Default constructor.
<a href="#">operator= (page 183)</a>	Iterator assignment operator.
<a href="#">~db_base_iterator (page 184)</a>	Destructor.
<a href="#">get_bulk_retrieval (page 185)</a>	Get bulk buffer size.
<a href="#">is_rmw (page 186)</a>	Get DB_RMW setting.
<a href="#">is_directdb_get (page 187)</a>	Get direct database get setting.

### Group

[Dbstl Iterator Classes \(page 176\)](#)

## refresh

### Function Details

```
int refresh(bool from_db=true)
```

Read data from underlying database via its cursor, and update its cached value.

#### Parameters

#### from\_db

Whether retrieve data from database rather than using the cached data in this iterator.

#### Return Value

0 if succeeded. Otherwise an [DbstlException](#) exception will be thrown.

### Class

[db\\_base\\_iterator](#)

## **close\_cursor**

### **Function Details**

```
void close_cursor() const
```

Close its cursor.

If you are sure the iterator is no longer used, call this function so that its underlying cursor is closed before this iterator is destructed, potentially increase performance and concurrency. Note that the cursor is definitely closed at iterator destruction if you don't close it explicitly.

### **Class**

[db\\_base\\_iterator](#)

## set\_bulk\_buffer

### Function Details

```
bool set_bulk_buffer(u_int32_t sz)
```

Call this function to modify bulk buffer size.

Bulk retrieval is enabled when creating an iterator, so users later can only modify the bulk buffer size to another value, but can't enable/disable bulk read while an iterator is already alive.

### Parameters

**sz**

The new buffer size in bytes.

### Return Value

true if succeeded, false otherwise.

### Class

[db\\_base\\_iterator](#)

## get\_bulk\_bufsize

### Function Details

```
u_int32_t get_bulk_bufsize()
```

Return current bulk buffer size.

Returns 0 if bulk retrieval is not enabled.

### Class

[db\\_base\\_iterator](#)

## db\_base\_iterator

### Function Details

```
db_base_iterator()
```

Default constructor.

```
db_base_iterator(db_container *powner, bool directdbget, bool b_read_only,  
                u_int32_t bulk,  
                bool rmw)
```

Constructor.

```
db_base_iterator(const db_base_iterator &bi)
```

Copy constructor. Copy all members of this class.

### Class

[db\\_base\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &bi)
```

Iterator assignment operator.

Iterator assignment will cause the underlying cursor of the right iterator to be duplicated to the left iterator after its previous cursor is closed, to make sure each iterator owns one unique cursor. The key/data cached in the right iterator is copied to the left iterator. Consequently, the left iterator points to the same key/data pair in the database as the the right value after the assignment, and have identical cached key/data pair.

### Parameters

**bi**

The other iterator to assign with.

### Return Value

The iterator bi's reference.

### Class

[db\\_base\\_iterator](#)

## ~db\_base\_iterator

### Function Details

```
virtual ~db_base_iterator()
```

Destructor.

### Class

[db\\_base\\_iterator](#)

## get\_bulk\_retrieval

### Function Details

```
u_int32_t get_bulk_retrieval() const
```

Get bulk buffer size.

Return bulk buffer size. If the size is 0, bulk retrieval is not enabled.

### Class

[db\\_base\\_iterator](#)

## **is\_rmw**

### **Function Details**

```
bool is_rmw() const
```

Get DB\_RMW setting.

Return true if the iterator's cursor has DB\_RMW flag set, false otherwise. DB\_RMW flag causes a write lock to be acquired when reading a key/data pair, so that the transaction won't block later when writing back the updated value in a read-modify-write operation cycle.

### **Class**

[db\\_base\\_iterator](#)

## is\_directdb\_get

### Function Details

```
bool is_directdb_get() const
```

Get direct database get setting.

Return true if every operation to retrieve the key/data pair the iterator points to will read from database rather than using the cached value, false otherwise.

### Class

[db\\_base\\_iterator](#)

---

## Chapter 11. Iterator Classes for `db_vector`

`db_vector` has two iterator classes --- `db_vector_base_iterator` and `db_vector_iterator` .

The differences between the two classes are that the `db_vector_base_iterator` can only be used to read its referenced value, so it is intended as `db_vector`'s const iterator; While the other class allows both read and write access. If your access pattern is readonly, it is strongly recommended that you use the const iterator because it is faster and more efficient. The two classes have identical behaviors to `std::vector::const_iterator` and `std::vector::iterator` respectively. Note that the common public member function behaviors are described in the `db_base_iterator` section.

### See Also

[db\\_base\\_iterator](#)

### Public Members

Member	Description
<a href="#">db_vector_base_iterator</a>	<code>db_vector_base_iterator</code>
<a href="#">db_vector_iterator</a>	<code>db_vector_iterator</code>

### Group

[Dbstl Iterator Classes \(page 176\)](#)

---

## Chapter 12. Db\_vector\_base\_iterator

This class is the const iterator class for [db\\_vector](#) , and it is inherited by the [db\\_vector\\_iterator](#) class, which is the iterator class for [db\\_vector](#) .

### Public Members

Member	Description
<a href="#">db_vector_base_iterator</a> (page 190)	
<a href="#">~db_vector_base_iterator</a> (page 191)	
<a href="#">operator==</a> (page 192)	Equality comparison operator.
<a href="#">operator!=</a> (page 193)	Unequal compare, identical to ! <a href="#">operator(==itr)</a> .
<a href="#">operator&lt;</a> (page 194)	Less than comparison operator.
<a href="#">operator&lt;=</a> (page 195)	Less equal comparison operator.
<a href="#">operator&gt;=</a> (page 196)	Greater equal comparison operator.
<a href="#">operator&gt;</a> (page 197)	Greater comparison operator.
<a href="#">operator++</a> (page 198)	Pre-increment.
<a href="#">operator--</a> (page 199)	Pre-decrement.
<a href="#">operator=</a> (page 200)	Assignment operator.
<a href="#">operator+</a> (page 201)	Iterator movement operator.
<a href="#">operator+=</a> (page 202)	Move this iterator backward by n elements.
<a href="#">operator-</a> (page 203)	Iterator movement operator.
<a href="#">operator-=</a> (page 204)	Move this iterator forward by n elements.
<a href="#">operator *</a> (page 205)	Dereference operator.
<a href="#">operator-&gt;</a> (page 206)	Arrow operator.
<a href="#">operator[]</a> (page 207)	Iterator index operator.
<a href="#">get_current_index</a> (page 208)	Get current index of within the vector.
<a href="#">move_to</a> (page 209)	Iterator movement function.
<a href="#">refresh</a> (page 210)	Refresh iterator cached value.
<a href="#">close_cursor</a> (page 211)	Close underlying Berkeley DB cursor of this iterator.
<a href="#">set_bulk_buffer</a> (page 212)	Modify bulk buffer size.
<a href="#">get_bulk_bufsize</a> (page 213)	Get bulk retrieval buffer size in bytes.

### Group

[Iterator Classes for db\\_vector](#) (page 188)

## db\_vector\_base\_iterator

### Function Details

```
db_vector_base_iterator(const db_vector_base_iterator< T > &vi)
```

```
db_vector_base_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
    bool rmw=false, bool directdbget=true,  
    bool readonly=false)
```

```
db_vector_base_iterator()
```

### Group: Constructors and destructor

Do not construct iterators explicitly using these constructors, but call `db_vector::begin() const` to get an valid iterator.

`db_vector::begin() const`

### Class

[db\\_vector\\_base\\_iterator](#)

## ~db\_vector\_base\_iterator

### Function Details

```
virtual ~db_vector_base_iterator()
```

### Group: Constructors and destructor

Do not construct iterators explicitly using these constructors, but call `db_vector::begin() const` to get an valid iterator.

`db_vector::begin() const`

### Class

[db\\_vector\\_base\\_iterator](#)

## operator==

### Function Details

```
bool operator==(const self &itr) const
```

Equality comparison operator.

Invalid iterators are equal; Valid iterators sitting on the same key/data pair equal; Otherwise not equal.

### Parameters

**itr**

The iterator to compare against.

### Return Value

True if this iterator equals to itr; False otherwise.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator!=

### Function Details

```
bool operator!=(const self &itr) const
```

Unequal compare, identical to !operator(==itr).

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

False if this iterator equals to itr; True otherwise.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator<

### Function Details

```
bool operator<(const self &itr) const
```

Less than comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

True if this iterator is less than itr.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator<=

### Function Details

```
bool operator<=(const self &itr) const
```

Less equal comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

True if this iterator is less than or equal to itr.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator>=

### Function Details

```
bool operator>=(const self &itr) const
```

Greater equal comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

True if this iterator is greater than or equal to itr.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator>

### Function Details

```
bool operator>(const self &itr) const
```

Greater comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

True if this iterator is greater than itr.

### Group: Iterator comparison operators

The way to compare two iterators is to compare the index values of the two elements they point to.

The iterator sitting on an element with less index is regarded to be smaller. And the invalid iterator sitting after last element is greater than any other iterators, because it is assumed to have an index equal to last element's index plus one; The invalid iterator sitting before first element is less than any other iterators because it is assumed to have an index -1.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Pre-increment.

Move the iterator one element backward, so that the element it sits on has a bigger index. Use ++iter rather than iter++ where possible to avoid two useless iterator copy constructions.

#### Return Value

This iterator after incremented.

```
self operator++(int)
```

Post-increment.

Move the iterator one element backward, so that the element it sits on has a bigger index. Use ++iter rather than iter++ where possible to avoid two useless iterator copy constructions.

#### Return Value

A new iterator not incremented.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Pre-decrement.

Move the iterator one element backward, so that the element it sits on has a smaller index. Use --iter rather than iter-- where possible to avoid two useless iterator copy constructions.

#### Return Value

This iterator after decremented.

```
self operator--(int)
```

Post-decrement.

Move the iterator one element backward, so that the element it sits on has a smaller index. Use --iter rather than iter-- where possible to avoid two useless iterator copy constructions.

#### Return Value

A new iterator not decremented.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &itr)
```

Assignment operator.

This iterator will point to the same key/data pair as itr, and have the same configurations as itr.

#### Parameters

**itr**

The right value of the assignment.

#### Return Value

This iterator's reference.

#### See Also

[db\\_base\\_iterator::operator=](#)

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator+

### Function Details

```
self operator+(difference_type n) const
```

Iterator movement operator.

Return another iterator by moving this iterator forward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move forward by |n| element.

#### Return Value

The new iterator at new position.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator+=

### Function Details

```
const self& operator+=(difference_type n)
```

Move this iterator backward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move forward by |n| element.

#### Return Value

Reference to this iterator at new position.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator-

### Function Details

```
self operator-(difference_type n) const
```

Iterator movement operator.

Return another iterator by moving this iterator backward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move backward by |n| element.

#### Return Value

The new iterator at new position.

```
difference_type operator-(const self &itr) const
```

Iterator distance operator.

Return the index difference of this iterator and itr, so if this iterator sits on an element with a smaller index, this call will return a negative number.

#### Parameters

**itr**

The other iterator to subtract. itr can be the invalid iterator after last element or before first element, their index will be regarded as last element's index + 1 and -1 respectively.

#### Return Value

The index difference.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## operator-=

### Function Details

```
const self& operator-=(difference_type n)
```

Move this iterator forward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move backward by |n| element.

#### Return Value

Reference to this iterator at new position.

### Group: Iterator movement operators.

When we talk about iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_vector\\_base\\_iterator](#)

## **operator \***

### **Function Details**

```
reference operator *() const
```

Dereference operator.

Return the reference to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type. The returned value can only be used to read its referenced element.

#### **Return Value**

The reference to the element this iterator points to.

### **Class**

[db\\_vector\\_base\\_iterator](#)

## **operator->**

### **Function Details**

```
pointer operator->() const
```

Arrow operator.

Return the pointer to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type. The returned value can only be used to read its referenced element.

### **Return Value**

The address of the referenced object.

### **Class**

[db\\_vector\\_base\\_iterator](#)

## operator[]

### Function Details

```
value_type_wrap operator[](difference_type _Off) const
```

Iterator index operator.

If `_Off` not in a valid range, the returned value will be invalid. Note that you should use a `value_type_wrap` type to hold the returned value.

### Parameters

#### `_Off`

The valid index relative to this iterator.

### Return Value

Return the element which is at position `*this + _Off`. The returned value can only be used to read its referenced element.

### Class

[db\\_vector\\_base\\_iterator](#)

## **get\_current\_index**

### **Function Details**

```
index_type get_current_index() const
```

Get current index of within the vector.

Return the iterators current element's index (0 based). Requires this iterator to be a valid iterator, not end\_itr\_.

#### **Return Value**

current index of the iterator.

### **Class**

[db\\_vector\\_base\\_iterator](#)

## move\_to

### Function Details

```
void move_to(index_type n) const
```

Iterator movement function.

Move this iterator to the index "n". If n is not in the valid range, this iterator will be an invalid iterator equal to end() iterator.

### Parameters

**n**

target element's index.

### See Also

[db\\_vector::end\(\)](#) ;

### Class

[db\\_vector\\_base\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true)
```

Refresh iterator cached value.

#### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

#### See Also

[db\\_base\\_iterator::refresh\(bool\)](#) .

### Class

[db\\_vector\\_base\\_iterator](#)

## **close\_cursor**

### **Function Details**

```
void close_cursor() const
```

Close underlying Berkeley DB cursor of this iterator.

#### **See Also**

[db\\_base\\_iterator::close\\_cursor\(\) const](#)

### **Class**

[db\\_vector\\_base\\_iterator](#)

## set\_bulk\_buffer

### Function Details

```
bool set_bulk_buffer(u_int32_t sz)
```

Modify bulk buffer size.

Bulk read is enabled when creating an iterator, so you later can only modify the bulk buffer size to another value, but can't enable/disable bulk read while an iterator is already alive.

### Parameters

#### **sz**

The new size of the bulk read buffer of this iterator.

### Return Value

Returns true if succeeded, false otherwise.

### See Also

[db\\_base\\_iterator::set\\_bulk\\_buffer\(u\\_int32\\_t sz\)](#)

### Class

[db\\_vector\\_base\\_iterator](#)

## **get\_bulk\_bufsize**

### **Function Details**

```
u_int32_t get_bulk_bufsize()
```

Get bulk retrieval buffer size in bytes.

#### **Return Value**

Return current bulk buffer size, or 0 if bulk retrieval is not enabled.

#### **See Also**

[db\\_base\\_iterator::get\\_bulk\\_bufsize\(\)](#)

### **Class**

[db\\_vector\\_base\\_iterator](#)

---

## Chapter 13. Db\_vector\_iterator

### Public Members

Member	Description
<a href="#">db_vector_iterator</a> (page 215)	
<a href="#">~db_vector_iterator</a> (page 216)	
<a href="#">operator++</a> (page 217)	Pre-increment.
<a href="#">operator--</a> (page 218)	Pre-decrement.
<a href="#">operator=</a> (page 219)	Assignment operator.
<a href="#">operator+</a> (page 220)	Iterator movement operator.
<a href="#">operator+=</a> (page 221)	Move this iterator backward by n elements.
<a href="#">operator-</a> (page 222)	Iterator movement operator.
<a href="#">operator-=</a> (page 224)	Move this iterator forward by n elements.
<a href="#">operator *</a> (page 225)	Dereference operator.
<a href="#">operator-&gt;</a> (page 226)	Arrow operator.
<a href="#">operator[]</a> (page 227)	Iterator index operator.
<a href="#">refresh</a> (page 228)	Refresh iterator cached value.

### Group

[Iterator Classes for db\\_vector](#) (page 188)

## db\_vector\_iterator

### Function Details

```
db_vector_iterator(const db_vector_iterator< T,  
                  value_type_sub > &vi)
```

```
db_vector_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
                  bool brmw=false, bool directdbget=true,  
                  bool b_read_only=false)
```

```
db_vector_iterator()
```

```
db_vector_iterator(const db_vector_base_iterator< T > &obj)
```

### Group: Constructors and destructor

Do not construct iterators explicitly using these constructors, but call [db\\_vector::begin](#) to get an valid iterator.

[db\\_vector::begin](#)

### Class

[db\\_vector\\_iterator](#)

## ~db\_vector\_iterator

### Function Details

```
virtual ~db_vector_iterator()
```

### Group: Constructors and destructor

Do not construct iterators explicitly using these constructors, but call [db\\_vector::begin](#) to get an valid iterator.

[db\\_vector::begin](#)

### Class

[db\\_vector\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Pre-increment.

#### Return Value

This iterator after incremented.

#### See Also

[db\\_vector\\_base\\_iterator::operator++\(\)](#)

```
self operator++(int)
```

Post-increment.

#### Return Value

A new iterator not incremented.

#### See Also

[db\\_vector\\_base\\_iterator::operator++\(int\)](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Pre-decrement.

#### Return Value

This iterator after decremented.

#### See Also

[db\\_vector\\_base\\_iterator::operator--\(\)](#)

```
self operator--(int)
```

Post-decrement.

#### Return Value

A new iterator not decremented.

#### See Also

[db\\_vector\\_base\\_iterator::operator--\(int\)](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &itr)
```

Assignment operator.

This iterator will point to the same key/data pair as itr, and have the same configurations as itr.

### Parameters

**itr**

The right value of the assignment.

### Return Value

This iterator's reference.

### See Also

[db\\_base\\_iterator::operator=\(const self&\)](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## operator+

### Function Details

```
self operator+(difference_type n) const
```

Iterator movement operator.

Return another iterator by moving this iterator backward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move forward by |n| element.

#### Return Value

The new iterator at new position.

#### See Also

[db\\_vector\\_base\\_iterator::operator+\(difference\\_type n\) const](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## operator+=

### Function Details

```
const self& operator+=(difference_type n)
```

Move this iterator backward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move forward by |n| element.

#### Return Value

Reference to this iterator at new position.

#### See Also

[db\\_vector\\_base\\_iterator::operator+=\(difference\\_type n\)](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## operator-

### Function Details

```
self operator-(difference_type n) const
```

Iterator movement operator.

Return another iterator by moving this iterator forward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move backward by |n| element.

#### Return Value

The new iterator at new position.

#### See Also

[db\\_vector\\_base\\_iterator::operator-\(difference\\_type n\) const](#)

```
difference_type operator-(const self &itr) const
```

Iterator distance operator.

Return the index difference of this iterator and itr, so if this iterator sits on an element with a smaller index, this call will return a negative number.

#### Parameters

**itr**

The other iterator to subtract. itr can be the invalid iterator after last element or before first element, their index will be regarded as last element's index + 1 and -1 respectively.

#### Return Value

The index difference.

#### See Also

[db\\_vector\\_base\\_iterator::operator-\(const self& itr\) const](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

## **Class**

`db_vector_iterator`

## operator-=

### Function Details

```
const self& operator-=(difference_type n)
```

Move this iterator forward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move backward by |n| element.

#### Return Value

Reference to this iterator at new position.

#### See Also

[db\\_vector\\_base\\_iterator::operator-=\(difference\\_type n\)](#)

### Group: Iterator movement operators.

These functions have identical behaviors and semantics as those of [db\\_vector\\_base\\_iterator](#) , so please refer to equivalent in that class.

### Class

[db\\_vector\\_iterator](#)

## **operator \***

### **Function Details**

```
reference operator *() const
```

Dereference operator.

Return the reference to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type. The returned value can be used to read or update its referenced element.

#### **Return Value**

The reference to the element this iterator points to.

### **Class**

[db\\_vector\\_iterator](#)

## **operator->**

### **Function Details**

```
pointer operator->() const
```

Arrow operator.

Return the pointer to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type. The returned value can be used to read or update its referenced element.

### **Return Value**

The address of the referenced object.

### **Class**

[db\\_vector\\_iterator](#)

## operator[]

### Function Details

```
value_type_wrap operator[](difference_type _Off) const
```

Iterator index operator.

If `_Off` not in a valid range, the returned value will be invalid. Note that you should use a `value_type_wrap` type to hold the returned value.

### Parameters

#### `_Off`

The valid index relative to this iterator.

### Return Value

Return the element which is at position `*this + _Off`, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type. The returned value can be used to read or update its referenced element.

### Class

[db\\_vector\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true)
```

Refresh iterator cached value.

#### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

#### See Also

[db\\_base\\_iterator::refresh\(bool\)](#)

### Class

[db\\_vector\\_iterator](#)

---

## Chapter 14. Iterator Classes for `db_map` and `db_multimap`

`db_map` has two iterator class templates -- `db_map_base_iterator` and `db_map_iterator` .

They are the const iterator class and iterator class for `db_map` and `db_multimap` .  
`db_map_iterator` inherits from `db_map_base_iterator` .

The two classes have identical behaviors to `std::map::const_iterator` and `std::map::iterator` respectively. Note that the common public member function behaviors are described in the `db_base_iterator` section.

The differences between the two classes are that the `db_map_base_iterator` can only be used to read its referenced value, while `db_map_iterator` allows both read and write access. If your access pattern is readonly, it is strongly recommended that you use the const iterator because it is faster and more efficient.

### Public Members

Member	Description
<a href="#">db_map_base_iterator</a>	<code>db_map_base_iterator</code>
<a href="#">db_map_iterator</a>	<code>db_map_iterator</code>

### Group

[Dbstl Iterator Classes](#) (page 176)

---

## Chapter 15. Db\_map\_base\_iterator

### Public Members

Member	Description
<a href="#">db_map_base_iterator (page 231)</a>	Copy constructor.
<a href="#">~db_map_base_iterator (page 233)</a>	Destructor.
<a href="#">operator++ (page 234)</a>	Pre-increment.
<a href="#">operator-- (page 235)</a>	Pre-decrement.
<a href="#">operator== (page 236)</a>	Equal comparison operator.
<a href="#">operator!= (page 237)</a>	Unequal comparison operator.
<a href="#">operator * (page 238)</a>	Dereference operator.
<a href="#">operator-&gt; (page 239)</a>	Arrow operator.
<a href="#">refresh (page 240)</a>	Refresh iterator cached value.
<a href="#">close_cursor (page 241)</a>	Close underlying Berkeley DB cursor of this iterator.
<a href="#">move_to (page 242)</a>	Iterator movement function.
<a href="#">set_bulk_buffer (page 243)</a>	Modify bulk buffer size.
<a href="#">get_bulk_bufsize (page 244)</a>	Get bulk retrieval buffer size in bytes.
<a href="#">operator= (page 245)</a>	Assignment operator.

### Group

[Iterator Classes for db\\_map and db\\_multimap \(page 229\)](#)

## db\_map\_base\_iterator

### Function Details

```
db_map_base_iterator(const self &vi)
```

Copy constructor.

#### Parameters

**vi**

The other iterator of the same type to initialize this.

```
db_map_base_iterator(const base &vi)
```

Base copy constructor.

#### Parameters

**vi**

Initialize from a base class iterator.

```
db_map_base_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
    bool rmw=false, bool directdbget=true,  
    bool readonly=false)
```

Constructor.

#### Parameters

**b\_bulk\_retrieval**

The bulk read buffer size. 0 means bulk read disabled.

**directdbget**

Whether do direct database get rather than using key/data values cached in the iterator whenever read.

**readonly**

Whether open a read only cursor. Only effective when using Berkeley DB Concurrent Data Store.

**powner**

The container which creates this iterator.

**rmw**

Whether set DB\_RMW flag in underlying cursor.

```
db_map_base_iterator()
```

Default constructor, dose not create the cursor for now.

**Group: Constructors and destructor**

Do not create iterators directly using these constructors, but call [db\\_map::begin](#) or [db\\_multimap\\_begin](#) to get instances of this class.

[db\\_map::begin\(\)](#) [db\\_multimap::begin\(\)](#)

**Class**

[db\\_map\\_base\\_iterator](#)

## ~db\_map\_base\_iterator

### Function Details

```
virtual ~db_map_base_iterator()
```

Destructor.

### Group: Constructors and destructor

Do not create iterators directly using these constructors, but call [db\\_map::begin](#) or [db\\_multimap\\_begin](#) to get instances of this class.

[db\\_map::begin\(\)](#) [db\\_multimap::begin\(\)](#)

### Class

[db\\_map\\_base\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Pre-increment.

#### Return Value

This iterator after incremented.

```
self operator++(int)
```

Post-increment.

#### Return Value

Another iterator having the old value of this iterator.

### Group: Iterator increment movement functions.

The two functions moves the iterator one element backward, so that the element it sits on has a bigger key.

The btree/hash key comparison routine determines which key is greater. Use ++iter rather than iter++ where possible to avoid two useless iterator copy constructions.

### Class

[db\\_map\\_base\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Pre-decrement.

#### Return Value

This iterator after decremented.

```
self operator--(int)
```

Post-decrement.

#### Return Value

Another iterator having the old value of this iterator.

### Group: Iterator decrement movement functions.

The two functions moves the iterator one element forward, so that the element it sits on has a smaller key.

The btree/hash key comparison routine determines which key is greater. Use --iter rather than iter-- where possible to avoid two useless iterator copy constructions.

### Class

[db\\_map\\_base\\_iterator](#)

## operator==

### Function Details

```
bool operator==(const self &itr) const
```

Equal comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

Returns true if equal, false otherwise.

### Group: Compare operators.

Only equal comparison is supported.

### Class

[db\\_map\\_base\\_iterator](#)

## operator!=

### Function Details

```
bool operator!=(const self &itr) const
```

Unequal comparison operator.

#### Parameters

**itr**

The iterator to compare against.

#### Return Value

Returns false if equal, true otherwise.

#### See Also

bool [operator==\(const self&itr\) const](#)

### Group: Compare operators.

Only equal comparison is supported.

### Class

[db\\_map\\_base\\_iterator](#)

## operator \*

### Function Details

```
reference operator *() const
```

Dereference operator.

Return the reference to the cached data element, which is an `pair<Key_type, T>`. You can only read its referenced data via this iterator but can not update it.

### Return Value

Current data element reference object, i.e. [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_map\\_base\\_iterator](#)

## operator->

### Function Details

```
pointer operator->() const
```

Arrow operator.

Return the pointer to the cached data element, which is an `pair<Key_type, T>`. You can only read its referenced data via this iterator but can not update it.

### Return Value

Current data element reference object's address, i.e. address of [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_map\\_base\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true) const
```

Refresh iterator cached value.

#### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

#### See Also

[db\\_base\\_iterator::refresh\(bool\)](#)

### Class

[db\\_map\\_base\\_iterator](#)

## **close\_cursor**

### **Function Details**

```
void close_cursor() const
```

Close underlying Berkeley DB cursor of this iterator.

#### **See Also**

[db\\_base\\_iterator::close\\_cursor\(\) const](#)

### **Class**

[db\\_map\\_base\\_iterator](#)

## move\_to

### Function Details

```
int move_to(const kdt &k,  
            int flag=DB_SET) const
```

Iterator movement function.

Move this iterator to the specified key *k*, by default moves exactly to *k*, and update cached data element, you can also specify `DB_SET_RANGE`, to move to the biggest key smaller than *k*. The btree/hash key comparison routine determines which key is bigger. When the iterator is on a multiple container, `move_to` will move itself to the first key/data pair of the identical keys.

#### Parameters

**k**

The target key value to move to.

**flag**

Flags available: `DB_SET`(default) or `DB_SET_RANGE`. `DB_SET` will move this iterator exactly at *k*; `DB_SET_RANGE` moves this iterator to *k* or the smallest key greater than *k*. If fail to find such a key, this iterator will become invalid.

#### Return Value

0 if succeed; non-0 otherwise, and this iterator becomes invalid. Call `db_strerror` with the return value to get the error message.

### Class

[db\\_map\\_base\\_iterator](#)

## set\_bulk\_buffer

### Function Details

```
bool set_bulk_buffer(u_int32_t sz)
```

Modify bulk buffer size.

Bulk read is enabled when creating an iterator, so users later can only modify the bulk buffer size to another value, but can't enable/disable bulk read while an iterator is already alive.

### Parameters

#### **SZ**

The new size of the bulk read buffer of this iterator.

### Return Value

Returns true if succeeded, false otherwise.

### See Also

[db\\_base\\_iterator::set\\_bulk\\_buffer\(u\\_int32\\_t \)](#)

### Class

[db\\_map\\_base\\_iterator](#)

## **get\_bulk\_bufsize**

### **Function Details**

```
u_int32_t get_bulk_bufsize()
```

Get bulk retrieval buffer size in bytes.

#### **Return Value**

Return current bulk buffer size or 0 if bulk retrieval is not enabled.

#### **See Also**

[db\\_base\\_iterator::get\\_bulk\\_bufsize\(\)](#)

### **Class**

[db\\_map\\_base\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &itr)
```

Assignment operator.

This iterator will point to the same key/data pair as itr, and have the same configurations as itr.

### Parameters

**itr**

The right value of assignment.

### Return Value

The reference of itr.

### See Also

[db\\_base\\_iterator::operator=\(const self&\)](#)

### Class

[db\\_map\\_base\\_iterator](#)

---

## Chapter 16. Db\_map\_iterator

### Public Members

Member	Description
<a href="#">db_map_iterator (page 247)</a>	Copy constructor.
<a href="#">~db_map_iterator (page 249)</a>	Destructor.
<a href="#">operator++ (page 250)</a>	Pre-increment.
<a href="#">operator-- (page 251)</a>	Pre-decrement.
<a href="#">operator * (page 252)</a>	Dereference operator.
<a href="#">operator-&gt; (page 253)</a>	Arrow operator.
<a href="#">refresh (page 254)</a>	Refresh iterator cached value.
<a href="#">operator= (page 255)</a>	Assignment operator.

### Group

[Dbstl Iterator Classes \(page 176\)](#)

## db\_map\_iterator

### Function Details

```
db_map_iterator(const db_map_iterator< kdt, ddt,  
                value_type_sub > &vi)
```

Copy constructor.

#### Parameters

**vi**

The other iterator of the same type to initialize this.

```
db_map_iterator(const db_map_base_iterator< kdt, realddt,  
                ddt > &vi)
```

Base copy constructor.

#### Parameters

**vi**

Initialize from a base class iterator.

```
db_map_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
                bool brmw=false, bool directdbget=true,  
                bool b_read_only=false)
```

Constructor.

#### Parameters

**b\_bulk\_retrieval**

The bulk read buffer size. 0 means bulk read disabled.

**brmw**

Whether set DB\_RMW flag in underlying cursor.

**powner**

The container which creates this iterator.

**directdbget**

Whether do direct database get rather than using key/data values cached in the iterator whenever read.

**b\_read\_only**

Whether open a read only cursor. Only effective when using Berkeley DB Concurrent Data Store.

```
db_map_iterator()
```

Default constructor, dose not create the cursor for now.

**Group: Constructors and destructor**

Do not create iterators directly using these constructors, but call [db\\_map::begin](#) or [db\\_multimap\\_begin](#) to get instances of this class.

[db\\_map::begin\(\)](#) [db\\_multimap::begin\(\)](#)

**Class**

[db\\_map\\_iterator](#)

## ~db\_map\_iterator

### Function Details

```
virtual ~db_map_iterator()
```

Destructor.

### Group: Constructors and destructor

Do not create iterators directly using these constructors, but call [db\\_map::begin](#) or [db\\_multimap\\_begin](#) to get instances of this class.

[db\\_map::begin\(\)](#) [db\\_multimap::begin\(\)](#)

### Class

[db\\_map\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Pre-increment.

#### Return Value

This iterator after incremented.

#### See Also

[db\\_map\\_base\\_iterator::operator++\(\)](#)

```
self operator++(int)
```

Post-increment.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_base\\_iterator::operator++\(int\)](#)

### Class

[db\\_map\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Pre-decrement.

#### Return Value

This iterator after decremented.

#### See Also

[db\\_map\\_base\\_iterator::operator--\(\)](#)

```
self operator--(int)
```

Post-decrement.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_base\\_iterator::operator--\(int\)](#)

### Class

[db\\_map\\_iterator](#)

## operator \*

### Function Details

```
reference operator *() const
```

Dereference operator.

Return the reference to the cached data element, which is an `pair<Key_type, ElementRef<T>>` object if T is a class type or an `pair<Key_type, ElementHolder<T>>` object if T is a C++ primitive data type.

#### Return Value

Current data element reference object, i.e. [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_map\\_iterator](#)

## operator->

### Function Details

```
pointer operator->() const
```

Arrow operator.

Return the pointer to the cached data element, which is an `pair<Key_type, ElementRef<T>>` object if T is a class type or an `pair<Key_type, ElementHolder<T>>` object if T is a C++ primitive data type.

#### Return Value

Current data element reference object's address, i.e. address of [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_map\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true) const
```

Refresh iterator cached value.

### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

### See Also

[db\\_base\\_iterator::refresh\(bool \)](#)

### Class

[db\\_map\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &itr)
```

Assignment operator.

This iterator will point to the same key/data pair as itr, and have the same configurations as itr.

### Parameters

**itr**

The right value of assignment.

### Return Value

The reference of itr.

### See Also

[db\\_base\\_iterator::operator=\(const self&\)](#)

### Class

[db\\_map\\_iterator](#)

---

## Chapter 17. Iterator Classes for `db_set` and `db_multiset`

[db\\_set\\_base\\_iterator](#) and [db\\_set\\_iterator](#) are the const iterator and iterator class for [db\\_set](#) and [db\\_multiset](#) .

They have identical behaviors to `std::set::const_iterator` and `std::set::iterator` respectively.

The difference between the two classes is that the [db\\_set\\_base\\_iterator](#) can only be used to read its referenced value, while [db\\_set\\_iterator](#) allows both read and write access. If the access pattern is readonly, it is strongly recommended that you use the const iterator because it is faster and more efficient.

The two classes inherit several functions from [db\\_map\\_base\\_iterator](#) and [db\\_map\\_iterator](#) respectively.

### See Also

[db\\_map\\_base\\_iterator](#) [db\\_map\\_iterator](#)

### Public Members

Member	Description
<a href="#">db_set_base_iterator</a>	<a href="#">db_set_base_iterator</a>
<a href="#">db_set_iterator</a>	<a href="#">db_set_iterator</a>

### Group

[Dbstl Iterator Classes](#) (page 176)

---

## Chapter 18. Db\_set\_base\_iterator

### Public Members

Member	Description
<a href="#">~db_set_base_iterator (page 258)</a>	Destructor.
<a href="#">db_set_base_iterator (page 259)</a>	Constructor.
<a href="#">operator++ (page 261)</a>	Post-increment.
<a href="#">operator-- (page 262)</a>	Post-decrement.
<a href="#">operator * (page 263)</a>	Dereference operator.
<a href="#">operator-&gt; (page 264)</a>	Arrow operator.
<a href="#">refresh (page 265)</a>	Refresh iterator cached value.

### Group

[Iterator Classes for db\\_set and db\\_multiset \(page 256\)](#)

## ~db\_set\_base\_iterator

### Function Details

```
virtual ~db_set_base_iterator()
```

Destructor.

### Group: Constructors and destructor

Do not use these constructors to create iterators, but call `db_set::begin() const` or `db_multiset::begin() const` to create valid iterators.

### Class

[db\\_set\\_base\\_iterator](#)

## db\_set\_base\_iterator

### Function Details

```
db_set_base_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
                    bool brmw=false, bool directdbget=true,  
                    bool b_read_only=false)
```

Constructor.

#### Parameters

##### **b\_bulk\_retrieval**

The bulk read buffer size. 0 means bulk read disabled.

##### **brmw**

Whether set DB\_RMW flag in underlying cursor.

##### **powner**

The container which creates this iterator.

##### **directdbget**

Whether do direct database get rather than using key/data values cached in the iterator whenever read.

##### **b\_read\_only**

Whether open a read only cursor. Only effective when using Berkeley DB Concurrent Data Store.

```
db_set_base_iterator()
```

Default constructor, dose not create the cursor for now.

```
db_set_base_iterator(const db_set_base_iterator &s)
```

Copy constructor.

#### Parameters

##### **s**

The other iterator of the same type to initialize this.

```
db_set_base_iterator(const base &bo)
```

Base copy constructor.

### Parameters

**bo**

Initialize from a base class iterator.

### Group: Constructors and destructor

Do not use these constructors to create iterators, but call `db_set::begin()` const or `db_multiset::begin()` const to create valid iterators.

### Class

[db\\_set\\_base\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Post-increment.

#### Return Value

This iterator after incremented.

#### See Also

[db\\_map\\_base\\_iterator::operator++\(\)](#)

```
self operator++(int)
```

Pre-increment.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_base\\_iterator::operator++\(int\)](#)

### Group: Iterator movement operators.

These functions are identical to those of [db\\_map\\_base\\_iterator](#) and [db\\_map\\_iterator](#) and [db\\_set\\_iterator](#) .

Actually the iterator movement functions in the four classes are the same.

### Class

[db\\_set\\_base\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Post-decrement.

#### Return Value

This iterator after decremented.

#### See Also

[db\\_map\\_base\\_iterator::operator--\(\)](#)

```
self operator--(int)
```

Pre-decrement.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_base\\_iterator::operator--\(int\)](#)

### Group: Iterator movement operators.

These functions are identical to those of [db\\_map\\_base\\_iterator](#) and [db\\_map\\_iterator](#) and [db\\_set\\_iterator](#) .

Actually the iterator movement functions in the four classes are the same.

### Class

[db\\_set\\_base\\_iterator](#)

## operator \*

### Function Details

```
reference operator *()
```

Dereference operator.

Return the reference to the cached data element, which is an object of type T. You can only use the return value to read its referenced data element, can not update it.

### Return Value

Current data element reference object, i.e. [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_set\\_base\\_iterator](#)

## operator->

### Function Details

```
pointer operator->() const
```

Arrow operator.

Return the pointer to the cached data element, which is an object of type T. You can only use the return value to read its referenced data element, can not update it.

### Return Value

Current data element reference object's address, i.e. address of [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_set\\_base\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true) const
```

Refresh iterator cached value.

### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

### See Also

[db\\_base\\_iterator::refresh\(bool\)](#)

### Class

[db\\_set\\_base\\_iterator](#)

---

## Chapter 19. Db\_set\_iterator

### Public Members

Member	Description
<a href="#">~db_set_iterator (page 267)</a>	Destructor.
<a href="#">db_set_iterator (page 268)</a>	Constructor.
<a href="#">operator++ (page 270)</a>	Pre-increment.
<a href="#">operator-- (page 271)</a>	Pre-decrement.
<a href="#">operator * (page 272)</a>	Dereference operator.
<a href="#">operator-&gt; (page 273)</a>	Arrow operator.
<a href="#">refresh (page 274)</a>	Refresh iterator cached value.

### Group

[Iterator Classes for db\\_set and db\\_multiset \(page 256\)](#)

## ~db\_set\_iterator

### Function Details

```
virtual ~db_set_iterator()
```

Destructor.

### Group: Constructors and destructor

Do not use these constructors to create iterators, but call [db\\_set::begin\(\)](#) or [db\\_multiset::begin\(\)](#) to create valid ones.

### Class

[db\\_set\\_iterator](#)

## db\_set\_iterator

### Function Details

```
db_set_iterator(db_container *powner, u_int32_t b_bulk_retrieval=0,  
               bool brmw=false, bool directdbget=true,  
               bool b_read_only=false)
```

Constructor.

#### Parameters

##### **b\_bulk\_retrieval**

The bulk read buffer size. 0 means bulk read disabled.

##### **brmw**

Whether set DB\_RMW flag in underlying cursor.

##### **powner**

The container which creates this iterator.

##### **directdbget**

Whether do direct database get rather than using key/data values cached in the iterator whenever read.

##### **b\_read\_only**

Whether open a read only cursor. Only effective when using Berkeley DB Concurrent Data Store.

```
db_set_iterator()
```

Default constructor, dose not create the cursor for now.

```
db_set_iterator(const db_set_iterator &s)
```

Copy constructor.

#### Parameters

##### **s**

The other iterator of the same type to initialize this.

```
db_set_iterator(const base &bo)
```

Base copy constructor.

### Parameters

**bo**

Initialize from a base class iterator.

```
db_set_iterator(const db_set_base_iterator< kdt > &bs)
```

Sibling copy constructor.

Note that this class does not derive from [db\\_set\\_base\\_iterator](#) but from [db\\_map\\_iterator](#) .

### Parameters

**bs**

Initialize from a base class iterator.

## Group: Constructors and destructor

Do not use these constructors to create iterators, but call [db\\_set::begin\(\)](#) or [db\\_multiset::begin\(\)](#) to create valid ones.

## Class

[db\\_set\\_iterator](#)

## operator++

### Function Details

```
self& operator++()
```

Pre-increment.

Identical to those of [db\\_map\\_iterator](#) .

#### Return Value

This iterator after incremented.

#### See Also

[db\\_map\\_iterator::operator++\(\)](#)

```
self operator++(int)
```

Post-increment.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_iterator::operator++\(int\)](#)

### Class

[db\\_set\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Pre-decrement.

#### Return Value

This iterator after decremented.

#### See Also

[db\\_map\\_iterator::operator--\(\)](#)

```
self operator--(int)
```

Post-decrement.

#### Return Value

Another iterator having the old value of this iterator.

#### See Also

[db\\_map\\_iterator::operator--\(int\)](#)

### Class

[db\\_set\\_iterator](#)

## operator \*

### Function Details

```
reference operator *()
```

Dereference operator.

Return the reference to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type.

### Return Value

Current data element reference object, i.e. [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_set\\_iterator](#)

## operator->

### Function Details

```
pointer operator->>() const
```

Arrow operator.

Return the pointer to the cached data element, which is an `ElementRef<T>` object if `T` is a class type or an `ElementHolder<T>` object if `T` is a C++ primitive data type.

### Return Value

Current data element reference object's address, i.e. address of [ElementHolder](#) or [ElementRef](#) object.

### Class

[db\\_set\\_iterator](#)

## refresh

### Function Details

```
virtual int refresh(bool from_db=true) const
```

Refresh iterator cached value.

#### Parameters

#### from\_db

If not doing direct database get and this parameter is true, we will retrieve data directly from db.

#### See Also

[db\\_base\\_iterator::refresh\(bool\)](#)

### Class

[db\\_set\\_iterator](#)

---

## Chapter 20. Db\_reverse\_iterator

This class is the reverse class adaptor for all dbstl iterator classes.

It inherits from real iterator classes like [db\\_vector\\_iterator](#) , [db\\_map\\_iterator](#) or [db\\_set\\_iterator](#) . When you call `container::rbegin()`, you will get an instance of this class.

### See Also

[db\\_vector\\_base\\_iterator](#) [db\\_vector\\_iterator](#) [db\\_map\\_base\\_iterator](#) [db\\_map\\_iterator](#)  
[db\\_set\\_base\\_iterator](#) [db\\_set\\_iterator](#)

### Public Members

Member	Description
<a href="#">operator++</a> (page 276)	Move this iterator forward by one element.
<a href="#">operator--</a> (page 277)	Move this iterator backward by one element.
<a href="#">operator+</a> (page 278)	Iterator shuffle operator.
<a href="#">operator-</a> (page 279)	Iterator shuffle operator.
<a href="#">operator+=</a> (page 280)	Iterator shuffle operator.
<a href="#">operator-=</a> (page 281)	Iterator shuffle operator.
<a href="#">operator&lt;</a> (page 282)	Less compare operator.
<a href="#">operator&gt;</a> (page 283)	Greater compare operator.
<a href="#">operator&lt;=</a> (page 284)	Less equal compare operator.
<a href="#">operator&gt;=</a> (page 285)	Greater equal compare operator.
<a href="#">db_reverse_iterator</a> (page 286)	Constructor. Construct from an iterator of wrapped type.
<a href="#">operator=</a> (page 287)	Assignment operator.
<a href="#">operator[]</a> (page 288)	Return the reference of the element which can be reached by moving this reverse iterator by Off times backward.

### Group

[Dbstl Iterator Classes](#) (page 176)

## operator++

### Function Details

```
self& operator++()
```

Move this iterator forward by one element.

#### Return Value

The moved iterator at new position.

```
self operator++(int)
```

Move this iterator forward by one element.

#### Return Value

The original iterator at old position.

### Group: Reverse iterator movement functions

When we talk about reverse iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_reverse\\_iterator](#)

## operator--

### Function Details

```
self& operator--()
```

Move this iterator backward by one element.

#### Return Value

The moved iterator at new position.

```
self operator--(int)
```

Move this iterator backward by one element.

#### Return Value

The original iterator at old position.

### Group: Reverse iterator movement functions

When we talk about reverse iterator movement, we think the container is a uni-directional range, represented by [begin, end), and this is true no matter we are using iterators or reverse iterators.

When an iterator is moved closer to "begin", we say it is moved forward, otherwise we say it is moved backward.

### Class

[db\\_reverse\\_iterator](#)

## operator+

### Function Details

```
self operator+(difference_type n) const
```

Iterator shuffle operator.

Return a new iterator by moving this iterator forward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move towards reverse direction.

#### Return Value

A new iterator at new position.

### Group: Operators for random reverse iterators

Methods below only applies to random iterators.

```
/////
```

Return a new iterator by moving this iterator backward or forward by n elements.

### Class

[db\\_reverse\\_iterator](#)

## operator-

### Function Details

```
self operator-(difference_type n) const
```

Iterator shuffle operator.

Return a new iterator by moving this iterator backward by n elements.

#### Parameters

**n**

The amount and direction of movement. If negative, will move towards reverse direction.

#### Return Value

A new iterator at new position.

```
difference_type operator-(const self &itr) const
```

Return the negative value of the difference of indices of elements this iterator and itr are sitting on.

#### Parameters

**itr**

The other reverse iterator.

#### Return Value

itr.index - this->index.

### Group: Operators for random reverse iterators

Methods below only applies to random iterators.

```
/////
```

Return a new iterator by moving this iterator backward or forward by n elements.

### Class

[db\\_reverse\\_iterator](#)

## operator+=

### Function Details

```
const self& operator+=(difference_type n)
```

Iterator shuffle operator.

Move this iterator forward by n elements and then return it.

#### Parameters

**n**

The amount and direction of movement. If negative, will move towards reverse direction.

#### Return Value

This iterator at new position.

### Group: Operators for random reverse iterators

Move this iterator backward or forward by n elements and then return it.

### Class

[db\\_reverse\\_iterator](#)

## operator-=

### Function Details

```
const self& operator-=(difference_type n)
```

Iterator shuffle operator.

Move this iterator backward by n elements and then return it.

#### Parameters

**n**

The amount and direction of movement. If negative, will move towards reverse direction.

#### Return Value

This iterator at new position.

### Group: Operators for random reverse iterators

Move this iterator backward or forward by n elements and then return it.

### Class

[db\\_reverse\\_iterator](#)

## operator<

### Function Details

```
bool operator<(const self &itr) const
```

Less compare operator.

### Group: Operators for random reverse iterators

Reverse iterator comparison against reverse iterator itr, the one sitting on elements with less index is returned to be greater.

### Class

[db\\_reverse\\_iterator](#)

## operator>

### Function Details

```
bool operator>(const self &itr) const
```

Greater compare operator.

### Group: Operators for random reverse iterators

Reverse iterator comparison against reverse iterator itr, the one sitting on elements with less index is returned to be greater.

### Class

[db\\_reverse\\_iterator](#)

## operator<=

### Function Details

```
bool operator<=(const self &itr) const
```

Less equal compare operator.

### Group: Operators for random reverse iterators

Reverse iterator comparison against reverse iterator itr, the one sitting on elements with less index is returned to be greater.

### Class

[db\\_reverse\\_iterator](#)

## operator>=

### Function Details

```
bool operator>=(const self &itr) const
```

Greater equal compare operator.

### Group: Operators for random reverse iterators

Reverse iterator comparison against reverse iterator itr, the one sitting on elements with less index is returned to be greater.

### Class

[db\\_reverse\\_iterator](#)

## db\_reverse\_iterator

### Function Details

```
db_reverse_iterator(const iterator &vi)
```

Constructor. Construct from an iterator of wrapped type.

```
db_reverse_iterator(const self &ritr)
```

Copy constructor.

```
db_reverse_iterator(const db_reverse_iterator< twin_itr_t,  
iterator > &ritr)
```

Copy constructor.

```
db_reverse_iterator()
```

Default constructor.

### Class

[db\\_reverse\\_iterator](#)

## operator=

### Function Details

```
const self& operator=(const self &ri)
```

Assignment operator.

#### Parameters

**ri**

The iterator to assign with.

#### Return Value

The iterator ri.

#### See Also

[db\\_base\\_iterator::operator=\(const self&\)](#)

### Class

[db\\_reverse\\_iterator](#)

## **operator[]**

### **Function Details**

```
value_type_wrap operator[](difference_type Off) const
```

Return the reference of the element which can be reached by moving this reverse iterator by Off times backward.

If Off is negative, the movement will be forward.

### **Class**

[db\\_reverse\\_iterator](#)

---

## Chapter 21. Dbstl Helper Classes

Classes of this module help to achieve various features of dbstl.

### Public Members

Member	Description
<a href="#">BulkRetrievalOption</a>	BulkRetrievalOption
<a href="#">ReadModifyWriteOption</a>	ReadModifyWriteOption
<a href="#">DbstlElemTraits</a>	DbstlElemTraits
<a href="#">DbstlDbt</a>	DbstlDbt
<a href="#">ElementRef and ElementHolder wrappers.</a>	ElementRef and ElementHolder wrappers.

### Group

None

---

## Chapter 22. ElementRef and ElementHolder Wappers

An [ElementRef](#) and [ElementHolder](#) object represents the reference to the data element referenced by an iterator.

Each iterator object has an [ElementRef](#) or [ElementHolder](#) object that stores the data element that the iterator points to.

The [ElementHolder](#) class is used to store primitive types into STL containers.

The [ElementRef](#) class is used to store other types into STL containers.

The [ElementRef](#) and [ElementHolder](#) classes have identical interfaces, and are treated the same by other STL classes. Since the [ElementRef](#) class inherits from the template data class, all methods have a `_DB_STL_` prefix to avoid name clashes.

An [ElementRef](#) or [ElementHolder](#) class corresponds to a single iterator instance. An Element object is generally owned by an iterator object. The ownership relationship is swapped in some specific situations, specifically for the dereference and array index operator.

### Public Members

Member	Description
<a href="#">ElementRef</a>	ElementRef
<a href="#">ElementHolder</a>	ElementHolder

### Group

[Dbstl Helper Classes \(page 289\)](#)

---

## Chapter 23. ElementHolder

A wrapper class for primitive types.

It has identical usage and public interface to the [ElementRef](#) class.

### See Also

[ElementRef](#) .

### Public Members

Member	Description
<a href="#">ElementHolder</a> (page 292)	Constructor.
<a href="#">~ElementHolder</a> (page 293)	Destructor.
<a href="#">operator+=</a> (page 294)	
<a href="#">operator-=</a> (page 295)	
<a href="#">operator *=</a> (page 296)	
<a href="#">operator/=</a> (page 297)	
<a href="#">operator%=</a> (page 298)	
<a href="#">operator &amp;=</a> (page 299)	
<a href="#">operator =</a> (page 300)	
<a href="#">operator^=</a> (page 301)	
<a href="#">operator&gt;&gt;=</a> (page 302)	
<a href="#">operator&lt;&lt;=</a> (page 303)	
<a href="#">operator++</a> (page 304)	
<a href="#">operator--</a> (page 305)	
<a href="#">operator=</a> (page 306)	
<a href="#">operator ptype</a> (page 307)	This operator is a type converter.
<a href="#">_DB_STL_value</a> (page 308)	Returns the data element this wrapper object wraps;.
<a href="#">_DB_STL_StoreElement</a> (page 309)	Function to store the data element.

### Group

[ElementRef and ElementHolder Wappers](#) (page 290)

# ElementHolder

## Function Details

```
ElementHolder(iterator_type *pitr=NULL)
```

Constructor.

If the `pitr` parameter is `NULL` or the default value is used, the object created is a simple wrapper and not connected to a container. If a valid iterator parameter is passed in, the wrapped element will be associated with the matching key/data pair in the underlying container.

### Parameters

**pitr**

The iterator owning this object.

```
ElementHolder(const ptype &dt)
```

Constructor.

Initializes an [ElementRef](#) wrapper without an iterator. It can only be used to wrap a data element in memory, it can't access an unerlying database.

### Parameters

**dt**

The base class object to initialize this object.

```
ElementHolder(const self &other)
```

Copy constructor.

The constructor takes a "deep" copy. The created object will be identical to, but independent from the original object.

### Parameters

**other**

The object to clone from.

## Class

[ElementHolder](#)

## ~ElementHolder

### Function Details

```
~ElementHolder()
```

Destructor.

### Class

[ElementHolder](#)

## operator+=

### Function Details

```
const self& operator+=(const ElementHolder< T2 > &p2)
```

```
const self& operator+=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator-=

### Function Details

```
const self& operator-=(const ElementHolder< T2 > &p2)
```

```
const self& operator-=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator \*=

### Function Details

```
const self& operator *=(const ElementHolder< T2 > &p2)
```

```
const self& operator *=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator/=

### Function Details

```
const self& operator/=(const ElementHolder< T2 > &p2)
```

```
const self& operator/=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator%=

### Function Details

```
const self& operator%=(const ElementHolder< T2 > &p2)
```

```
const self& operator%=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator &=

### Function Details

```
const self& operator &=(const ElementHolder< T2 > &p2)
```

```
const self& operator &=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator|=

### Function Details

```
const self& operator|=(const ElementHolder< T2 > &p2)
```

```
const self& operator|=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator^=

### Function Details

```
const self& operator^=(const ElementHolder< T2 > &p2)
```

```
const self& operator^=(const self &p2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator>>=

### Function Details

```
const self& operator>>=(size_t n)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, -- These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator<<=

### Function Details

```
const self& operator<<=(size_t n)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, -- These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator++

### Function Details

```
self& operator++()
```

```
self operator++(int)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator--

### Function Details

```
self& operator--()
```

```
self operator--(int)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator=

### Function Details

```
const ptype& operator=(const ptype &dt2)
```

```
const self& operator=(const self &dt2)
```

### Group: Math operators.

[ElementHolder](#) class templates also have all C/C++ self mutating operators for numeric primitive types, including: +=, -=, \*=, /=, =, <<=, >>=, &=, |=, ^=, ++, --. These operators should not be used when ddt is a sequence pointer type like char\* or wchar\_t\* or T\*, otherwise the behavior is undefined.

These methods exist only to override default behavior to store the new updated value, otherwise, the type convert operator could have done all the job. As you know, some of them are not applicable to float or double types or [ElementHolder](#) wrapper types for float/double types. These operators not only modifies the cached data element, but also stores new value to database if it associates a database key/data pair.

### Class

[ElementHolder](#)

## operator ptype

### Function Details

```
operator ptype() const
```

This operator is a type converter.

Where an automatic type conversion is needed, this function is called to convert this object into the primitive type it wraps.

### Class

[ElementHolder](#)

## **\_DB\_STL\_value**

### **Function Details**

```
const ptype& _DB_STL_value() const
```

Returns the data element this wrapper object wraps;.

```
ptype& _DB_STL_value()
```

Returns the data element this wrapper object wraps;.

### **Class**

[ElementHolder](#)

## **\_DB\_STL\_StoreElement**

### **Function Details**

```
void _DB_STL_StoreElement()
```

Function to store the data element.

The user needs to call this method after modifying the underlying object, so that the version stored in the container can be updated.

When `db_base_iterator`'s `directdb_get_member` is true, this function must be called after modifying the data member and before any subsequent container iterator dereference operations. If this step is not carried out any changes will be lost.

If the data element is changed via `ElementHolder<>::operator=()`, you don't need to call this function.

### **Class**

[ElementHolder](#)

---

## Chapter 24. ElementRef

[ElementRef](#) element wrapper for classes and structures.

### See Also

[ElementHolder](#)

### Public Members

Member	Description
<a href="#">~ElementRef (page 311)</a>	Destructor.
<a href="#">ElementRef (page 312)</a>	Constructor.
<a href="#">operator= (page 313)</a>	Assignment Operator.
<a href="#">_DB_STL_StoreElement (page 314)</a>	Function to store the data element.
<a href="#">_DB_STL_value (page 315)</a>	Returns the data element this wrapper object wraps.

### Group

[ElementRef and ElementHolder Wappers \(page 290\)](#)

## ~ElementRef

### Function Details

```
~ElementRef()
```

Destructor.

### Class

[ElementRef](#)

# ElementRef

## Function Details

```
ElementRef(iterator_type *pitr=NULL)
```

Constructor.

If the `pitr` parameter is `NULL` or the default value is used, the object created is a simple wrapper and not connected to a container. If a valid iterator parameter is passed in, the wrapped element will be associated with the matching key/data pair in the underlying container.

### Parameters

**pitr**

The iterator owning this object.

```
ElementRef(const ddt &dt)
```

Constructor.

Initializes an [ElementRef](#) wrapper without an iterator. It can only be used to wrap a data element in memory, it can't access an underlying database.

### Parameters

**dt**

The base class object to initialize this object.

```
ElementRef(const self &other)
```

Copy constructor.

The constructor takes a "deep" copy. The created object will be identical to, but independent from the original object.

### Parameters

**other**

The object to clone from.

## Class

[ElementRef](#)

## operator=

### Function Details

```
const ddt& operator=(const ddt &dt2)
```

Assignment Operator.

#### Parameters

**dt2**

The data value to assign with.

#### Return Value

The object dt2's reference.

```
const self& operator=(const self &me)
```

Assignment Operator.

#### Parameters

**me**

The object to assign with.

#### Return Value

The object me's reference.

### Group: Assignment operators.

The assignment operators are used to store right-values into the wrapped object, and also to store values into an underlying container.

### Class

[ElementRef](#)

## **\_DB\_STL\_StoreElement**

### **Function Details**

```
void _DB_STL_StoreElement()
```

Function to store the data element.

The user needs to call this method after modifying the underlying object, so that the version stored in the container can be updated.

When `db_base_iterator`'s `directdb_get_member` is true, this function must be called after modifying the data member and before any subsequent container iterator dereference operations. If this step is not carried out any changes will be lost.

If the data element is changed via `ElementHolder<>::operator=()`, you don't need to call this function.

### **Class**

[ElementRef](#)

## **\_DB\_STL\_value**

### **Function Details**

```
const ddt& _DB_STL_value() const
```

Returns the data element this wrapper object wraps.

```
ddt& _DB_STL_value()
```

Returns the data element this wrapper object wraps.

### **Class**

[ElementRef](#)

---

## Chapter 25. DbstlDbt

You can persist all bytes in a chunk of contiguous memory by constructing an [DbstlDbt](#) object A (use malloc to allocate the required number of bytes for A.data and copy the bytes to be stored into A.data, set other fields as necessary) and store A into a container, e.g.

`db_vector<DbstlDbt>`, this stores the bytes rather than the object A into the underlying database. The [DbstlDbt](#) class can help you avoid memory leaks, so it is strongly recommended that you use [DbstlDbt](#) rather than `Dbt` class.

[DbstlDbt](#) derives from `Dbt` class, and it does a deep copy on copy construction and assignment --by calling malloc to allocate its own memory and then copying the bytes to it; Conversely the destructor will free the memory on destruction if the data pointer is non-NULL. The destructor assumes the memory is allocated via malloc, hence why you are required to call malloc to allocate memory in order to use [DbstlDbt](#) .

[DbstlDbt](#) simply inherits all methods from `Dbt` with no extra new methods except the constructors/destructor and assignment operator, so it is easy to use.

In practice you rarely need to use [DbstlDbt](#) or `Dbt` because `dbstl` enables you to store any complex objects or primitive data. Only when you need to store raw bytes, e.g. a bitmap, do you need to use [DbstlDbt](#) .

Hence, [DbstlDbt](#) is the right class to use to store any object into Berkeley DB via `dbstl` without memory leaks.

Don't free the memory referenced by [DbstlDbt](#) objects, it will be freed when the [DbstlDbt](#) object is destructed.

Please refer to the two examples using [DbstlDbt](#) in `TestAssoc::test_arbitrary_object_storage` and `TestAssoc::test_char_star_string_storage` member functions, which illustrate how to correctly use [DbstlDbt](#) in order to store raw bytes.

This class handles the task of allocating and de-allocating memory internally. Although it can be used to store data which cannot be handled by the [DbstlElemTraits](#) class, in practice, it is usually more convenient to register callbacks in the [DbstlElemTraits](#) class for the type you are storing/retrieving using `dbstl`.

### Public Members

Member	Description
<a href="#">DbstlDbt</a> (page 318)	Construct an object with an existing chunk of memory of size1 bytes, referred by data1,.
<code>~DbstlDbt</code> (page 319)	The memory will be free'ed by the destructor.
<code>operator=</code> (page 320)	The memory will be reallocated if necessary.

### Group

[Dbstl Helper Classes](#) (page 289)



## DbstlDbt

### Function Details

```
DbstlDbt(void *data1,  
         u_int32_t size1)
```

Construct an object with an existing chunk of memory of size1 bytes, referred by data1,.

```
DbstlDbt()
```

```
DbstlDbt(const DbstlDbt &d)
```

This copy constructor does a deep copy.

### Class

[DbstlDbt](#)

## ~DbstlDbt

### Function Details

```
~DbstlDbt()
```

The memory will be freed by the destructor.

### Class

[DbstlDbt](#)

## **operator=**

### **Function Details**

```
const DbstlDbt& operator=(const DbstlDbt &d)
```

The memory will be reallocated if necessary.

### **Class**

[DbstlDbt](#)

---

## Chapter 26. DbstlElemTraits

This class is used to register callbacks to manipulate an object of a complex type.

These callbacks are used by dbstl at runtime to manipulate the object.

A complex type is a type whose members are not located in a contiguous chunk of memory. For example, the following class A is a complex type because for any instance a of class A, a.b\_ points to another object of type B, and dbstl treats the object that a.b\_ points to as part of the data of the instance a. Hence, if the user needs to store a.b\_ into a dbstl container, the user needs to register an appropriate callback to de-reference and store the object referenced by a.b. Similarly, the user also needs to register callbacks to marshall an array as well as to count the number of elements in such an array.

```
class A { int m; B *p_; }; class B { int n; };
```

The user also needs to register callbacks for i). returning an object; ii) s size in bytes; iii). Copying a complex object and assigning an object to another object of the same type; iv). Element comparison. v). Compare two sequences of any type of objects; Measuring the length of an object sequence and copy an object sequence.

Several elements located in a contiguous chunk of memory form a sequence. An element of a sequence may be a simple object located at a contiguous memory chunk, or a complex object, i.e. some of its members may contain references (pointers) to another region of memory. It is not necessary to store a special object to denote the end of the sequence. The callback to traverse the constituent elements of the sequence needs to be able to determine the end of the sequence.

Marshalling means packing the object's data members into a contiguous chunk of memory; unmarshalling is the opposite of marshalling. In other words, when you unmarshall an object, its data members are populated with values from a previously marshalled version of the object.

The callbacks need not be set to every type explicitly. dbstl will check if a needed callback function of this type is provided. If one is available, dbstl will use the registered callback. If the appropriate callback is not provided, dbstl will use reasonable defaults to do the job.

For returning the size of an object, the default behavior is to use the sizeof() operator; For marshalling and unmarshalling, dbstl uses memcpy, so the default behavior is sufficient for simple types whose data reside in a contiguous chunk of memory; Dbstl uses >, == and < for comparison operations; For char\* and wchar\_t \* strings, dbstl already provides the appropriate callbacks, so you do not need to register them. In general, if the default behavior is adequate, you don't need to register the corresponding callback.

If you have registered proper callbacks, the DbstlElemTraits<T> can also be used as the char\_traits<T> class for std::basic\_string<T, char\_traits<T>, >, and you can enable your class T to form a basic\_string<T, DbstlElemTraits<T>, >, and use basic\_string's functionality and the algorithms to manipulate it.

**Public Members**

Member	Description
<a href="#">assign (page 324)</a>	Assign one object to another.
<a href="#">eq (page 325)</a>	Check for equality of two objects.
<a href="#">lt (page 326)</a>	Less than comparison.
<a href="#">compare (page 327)</a>	Sequence comparison.
<a href="#">length (page 328)</a>	Returns the number of elements in sequence seq1.
<a href="#">copy (page 329)</a>	Copy first cnt number of elements from seq2 to seq1.
<a href="#">find (page 330)</a>	Find within the first cnt elements of sequence seq the position of element equal to elem.
<a href="#">move (page 331)</a>	Sequence movement.
<a href="#">to_char_type (page 332)</a>	
<a href="#">to_int_type (page 333)</a>	
<a href="#">eq_int_type (page 334)</a>	
<a href="#">eof (page 335)</a>	
<a href="#">not_eof (page 336)</a>	
<a href="#">set_restore_function (page 337)</a>	
<a href="#">get_restore_function (page 338)</a>	
<a href="#">set_assign_function (page 339)</a>	
<a href="#">get_assign_function (page 340)</a>	
<a href="#">get_size_function (page 341)</a>	
<a href="#">set_size_function (page 342)</a>	
<a href="#">get_copy_function (page 343)</a>	
<a href="#">set_copy_function (page 344)</a>	
<a href="#">set_sequence_len_function (page 345)</a>	
<a href="#">get_sequence_len_function (page 346)</a>	
<a href="#">get_sequence_copy_function (page 347)</a>	
<a href="#">set_sequence_copy_function (page 348)</a>	
<a href="#">set_compare_function (page 349)</a>	
<a href="#">get_compare_function (page 350)</a>	
<a href="#">set_sequence_compare_function (page 351)</a>	
<a href="#">get_sequence_compare_function (page 352)</a>	
<a href="#">set_sequence_n_compare_function (page 353)</a>	

---

Member	Description
<a href="#">get_sequence_n_compare_function (page 354)</a>	
<a href="#">instance (page 355)</a>	Factory method to create a singleton instance of this class.
<a href="#">~DbstlElemTraits (page 356)</a>	
<a href="#">DbstlElemTraits (page 357)</a>	

**Group**

[Dbstl Helper Classes \(page 289\)](#)

## assign

### Function Details

```
static void assign(T &left,  
                  const T &right)
```

Assign one object to another.

```
static T* assign(T *seq, size_t cnt,  
                T elem)
```

Assign first cnt number of elements of sequence seq with the value of elem.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the C++ STL algorithms.

### Class

[DbstElemTraits](#)

## eq

### Function Details

```
static bool eq(const T &left,  
              const T &right)
```

Check for equality of two objects.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## lt

### Function Details

```
static bool lt(const T &left,  
              const T &right)
```

Less than comparison.

Returns if object left is less than object right.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## compare

### Function Details

```
static int compare(const T *seq1, const T *seq2,  
                 size_t cnt)
```

Sequence comparison.

Compares the first cnt number of elements in the two sequences seq1 and seq2, returns negative/0/positive if seq1 is less/equal/greater than seq2.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## length

### Function Details

```
static size_t length(const T *seq)
```

Returns the number of elements in sequence seq1.

Note that seq1 may or may not end with a trailing ", it is completely user's responsibility for this decision, though seq[0], seq[1],... seq[length - 1] are all sequence seq's memory.

### Group: Interface compatible with std::string's char\_traits.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in std::basic\_string template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## copy

### Function Details

```
static T* copy(T *seq1, const T *seq2,  
              size_t cnt)
```

Copy first cnt number of elements from seq2 to seq1.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## find

### Function Details

```
static const T* find(const T *seq, size_t cnt,  
                    const T &elem)
```

Find within the first cnt elements of sequence seq the position of element equal to elem.

### Group: Interface compatible with `std::string's char_traits`.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## move

### Function Details

```
static T* move(T *seq1, const T *seq2,  
              size_t cnt)
```

Sequence movement.

Move first cnt number of elements from seq2 to seq1, seq1 and seq2 may or may not overlap.

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## to\_char\_type

### Function Details

```
static T to_char_type(const int_type &meta_elem)
```

### Group: Interface compatible with std::string's char\_traits.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in std::basic\_string template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## to\_int\_type

### Function Details

```
static int_type to_int_type(const T &elem)
```

### Group: Interface compatible with std::string's char\_traits.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in std::basic\_string template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## eq\_int\_type

### Function Details

```
static bool eq_int_type(const int_type &left,  
                        const int_type &right)
```

### Group: Interface compatible with std::string's char\_traits.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in std::basic\_string template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## eof

### Function Details

```
static int_type eof()
```

### Group: Interface compatible with `std::string`'s `char_traits`.

Following are `char_traits` functions, which make this class `char_traits` compatible, so that it can be used in `std::basic_string` template, and be manipulated by the C++ STL algorithms.

### Class

[DbstElemTraits](#)

## not\_eof

### Function Details

```
static int_type not_eof(const int_type &meta_elem)
```

### Group: Interface compatible with std::string's char\_traits.

Following are char\_traits functions, which make this class char\_traits compatible, so that it can be used in std::basic\_string template, and be manipulated by the c++ stl algorithms.

### Class

[DbstElemTraits](#)

## set\_restore\_function

### Function Details

```
void set_restore_function(ElemRstoreFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_restore\_function

### Function Details

```
ElemRstoreFunct get_restore_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstElemTraits](#)

## set\_assign\_function

### Function Details

```
void set_assign_function(ElemAssignFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_assign\_function

### Function Details

```
ElemAssignFunct get_assign_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstElemTraits](#)

## get\_size\_function

### Function Details

```
ElemSizeFunct get_size_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_size\_function

### Function Details

```
void set_size_function(ElemSizeFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_copy\_function

### Function Details

```
ElemCopyFunct get_copy_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_copy\_function

### Function Details

```
void set_copy_function(ElemCopyFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_sequence\_len\_function

### Function Details

```
void set_sequence_len_function(SequenceLenFunc f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_sequence\_len\_function

### Function Details

```
SequenceLenFunct get_sequence_len_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_sequence\_copy\_function

### Function Details

```
SequenceCopyFunc get_sequence_copy_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_sequence\_copy\_function

### Function Details

```
void set_sequence_copy_function(SequenceCopyFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_compare\_function

### Function Details

```
void set_compare_function(ElemCompareFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_compare\_function

### Function Details

```
ElemCompareFunct get_compare_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstElemTraits](#)

## set\_sequence\_compare\_function

### Function Details

```
void set_sequence_compare_function(SequenceCompareFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstElemTraits](#)

## get\_sequence\_compare\_function

### Function Details

```
SequenceCompareFunct get_sequence_compare_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## set\_sequence\_n\_compare\_function

### Function Details

```
void set_sequence_n_compare_function(SequenceNCompareFunct f)
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstlElemTraits](#)

## get\_sequence\_n\_compare\_function

### Function Details

```
SequenceNCompareFunct get_sequence_n_compare_function()
```

### Group: Set/get functions for callback function pointers.

These are the setters and getters for each callback function pointers.

### Class

[DbstElemTraits](#)

## instance

### Function Details

```
static DbstElemTraits* instance()
```

Factory method to create a singleton instance of this class.

The created object will be deleted by dbstl upon process exit.

### Class

[DbstElemTraits](#)

## ~DbstlElemTraits

### Function Details

```
~DbstlElemTraits()
```

### Class

[DbstlElemTraits](#)

## **DbstElemTraits**

### **Function Details**

```
DbstElemTraits()
```

### **Class**

[DbstElemTraits](#)

---

## Chapter 27. BulkRetrievalOption

Bulk retrieval configuration helper class.

Used by the `begin()` function of a container.

### Public Members

Member	Description
<a href="#">BulkRetrievalOption (page 359)</a>	
<a href="#">operator== (page 360)</a>	Equality comparison.
<a href="#">operator= (page 361)</a>	Assignment operator.
<a href="#">bulk_buf_size (page 362)</a>	Return the buffer size set to this object.
<a href="#">bulk_retrieval (page 363)</a>	This function indicates that you need a bulk retrieval iterator, and it can be also used to optionally set the bulk read buffer size.
<a href="#">no_bulk_retrieval (page 364)</a>	This function indicates that you do not need a bulk retrieval iterator.

### Group

[Dbstl Helper Classes \(page 289\)](#)

## BulkRetrievalOption

### Function Details

```
BulkRetrievalOption(Option bulk_retrieve1,  
    u_int32_t bulk_buf_sz=DBSTL_BULK_BUF_SIZE)
```

### Class

[BulkRetrievalOption](#)

## operator==

### Function Details

```
bool operator==(const BulkRetrievalOption &bro) const
```

Equality comparison.

### Class

[BulkRetrievalOption](#)

## **operator=**

### **Function Details**

```
void operator=(BulkRetrievalOption::Option opt)
```

Assignment operator.

### **Class**

[BulkRetrievalOption](#)

## bulk\_buf\_size

### Function Details

```
u_int32_t bulk_buf_size()
```

Return the buffer size set to this object.

### Class

[BulkRetrievalOption](#)

## bulk\_retrieval

### Function Details

```
static BulkRetrievalOption bulk_retrieval(u_int32_t bulk_buf_sz=  
DBSTL_BULK_BUF_SIZE)
```

This function indicates that you need a bulk retrieval iterator, and it can be also used to optionally set the bulk read buffer size.

### Class

[BulkRetrievalOption](#)

## no\_bulk\_retrieval

### Function Details

```
static BulkRetrievalOption no_bulk_retrieval()
```

This function indicates that you do not need a bulk retrieval iterator.

### Class

[BulkRetrievalOption](#)

---

## Chapter 28. ReadModifyWriteOption

Read-modify-write cursor configuration helper class.

Used by each `begin()` function of all containers.

### Public Members

Member	Description
<a href="#">operator= (page 366)</a>	Assignment operator.
<a href="#">operator== (page 367)</a>	Equality comparison.
<a href="#">read_modify_write (page 368)</a>	Call this function to tell the container's <code>begin()</code> function that you need a read-modify-write iterator.
<a href="#">no_read_modify_write (page 369)</a>	Call this function to tell the container's <code>begin()</code> function that you do not need a read-modify-write iterator.

### Group

[Dbstl Helper Classes \(page 289\)](#)

## operator=

### Function Details

```
void operator=(ReadModifyWriteOption::Option rmw1)
```

Assignment operator.

### Class

[ReadModifyWriteOption](#)

## **operator==**

### **Function Details**

```
bool operator==(const ReadModifyWriteOption &rmw1) const
```

Equality comparison.

### **Class**

[ReadModifyWriteOption](#)

## read\_modify\_write

### Function Details

```
static ReadModifyWriteOption read_modify_write()
```

Call this function to tell the container's begin() function that you need a read-modify-write iterator.

### Class

[ReadModifyWriteOption](#)

## no\_read\_modify\_write

### Function Details

```
static ReadModifyWriteOption no_read_modify_write()
```

Call this function to tell the container's begin() function that you do not need a read-modify-write iterator.

This is the default value for the parameter of any container's begin() function.

### Class

[ReadModifyWriteOption](#)

---

## Chapter 29. Dbstl Exception Classes

dbstl throws several types of exceptions on several kinds of errors, the exception classes form a class hierarchy.

First, there is the [DbstlException](#), which is the base class for all types of dbstl specific concrete exception classes. [DbstlException](#) inherits from the class DbException of Berkeley DB C++ API. Since DbException class inherits from C++ STL exception base class `std::exception`, you can make use of all Berkeley DB C++ and dbstl API exceptions in the same way you use the C++ `std::exception` class.

Besides exceptions of [DbstlException](#) and its subclasses, dbstl may also throw exceptions of DbException and its subclasses, which happens when a Berkeley DB call failed. So you should use the same way you catch Berkeley DB C++ API exceptions when you want to catch exceptions throw by Berkeley DB operations.

When an exception occurs, dbstl initialize an local exception object on the stack and throws the exception object, so you should catch an exception like this:

```
try { dbstl operations } catch(DbstlException ex){ Exception handling throw ex; // Optionally throw ex again }
```

### Public Members

Member	Description
<a href="#">DbstlException</a>	DbstlException
<a href="#">NotEnoughMemoryException</a>	NotEnoughMemoryException
<a href="#">InvalidIteratorException</a>	InvalidIteratorException
<a href="#">InvalidCursorException</a>	InvalidCursorException
<a href="#">InvalidDbtException</a>	InvalidDbtException
<a href="#">FailedAssertionException</a>	FailedAssertionException
<a href="#">NoSuchKeyException</a>	NoSuchKeyException
<a href="#">InvalidArgumentException</a>	InvalidArgumentException
<a href="#">NotSupportedException</a>	NotSupportedException
<a href="#">InvalidFunctionCall</a>	InvalidFunctionCall

### Group

None

---

## Chapter 30. DbstlException

Base class of all dbstl exception classes.

It is derived from Berkeley DB C++ API DbException class to maintain consistency with all Berkeley DB exceptions.

### Public Members

Member	Description
<a href="#">DbstlException (page 372)</a>	
<a href="#">operator= (page 373)</a>	
<a href="#">~DbstlException (page 374)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## DbstlException

### Function Details

```
DbstlException(const char *msg)
```

```
DbstlException(const char *msg,  
               int err)
```

```
DbstlException(const DbstlException &ex)
```

```
DbstlException(int err)
```

```
DbstlException(const char *prefix, const char *msg,  
               int err)
```

### Class

[DbstlException](#)

## **operator=**

### **Function Details**

```
const DbstlException& operator=(const DbstlException &exobj)
```

### **Class**

[DbstlException](#)

## ~DbstlException

### Function Details

```
virtual ~DbstlException()
```

### Class

[DbstlException](#)

---

## Chapter 31. InvalidDbtException

The Dbt object has inconsistent status or has no valid data, it is unable to be used any more.

### Public Members

Member	Description
<a href="#">InvalidDbtException (page 376)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## **InvalidDbtException**

### **Function Details**

```
InvalidDbtException()
```

```
InvalidDbtException(int error_code)
```

### **Class**

[InvalidDbtException](#)

---

## Chapter 32. FailedAssertionException

The assertions inside dbstl failed.

The code file name and line number will be passed to the exception object of this class.

### Public Members

Member	Description
<a href="#">what (page 378)</a>	
<a href="#">FailedAssertionException (page 379)</a>	
<a href="#">~FailedAssertionException (page 380)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## what

### Function Details

```
virtual const char* what() const
```

### Class

[FailedAssertionException](#)

## FailedAssertionException

### Function Details

```
FailedAssertionException(const char *fname, size_t lineno,  
                        const char *msg)
```

```
FailedAssertionException(const FailedAssertionException &ex)
```

### Class

[FailedAssertionException](#)

## ~FailedAssertionException

### Function Details

```
virtual ~FailedAssertionException()
```

### Class

[FailedAssertionException](#)

---

## Chapter 33. InvalidCursorException

The cursor has inconsistent status, it is unable to be used any more.

### Public Members

Member	Description
<a href="#">InvalidCursorException (page 382)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## InvalidCursorException

### Function Details

```
InvalidCursorException()
```

```
InvalidCursorException(int error_code)
```

### Class

[InvalidCursorException](#)

---

## Chapter 34. NoSuchKeyException

There is no such key in the database.

The key can't not be passed into the exception instance because this class has to be a class template for that to work.

### Public Members

Member	Description
<a href="#">NoSuchKeyException (page 384)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## **NoSuchKeyException**

### **Function Details**

```
NoSuchKeyException()
```

### **Class**

[NoSuchKeyException](#)

---

## Chapter 35. NotEnoughMemoryException

Failed to allocate memory because memory is not enough.

### Public Members

Member	Description
<a href="#">NotEnoughMemoryException (page 386)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## **NotEnoughMemoryException**

### **Function Details**

```
NotEnoughMemoryException(const char *msg,  
                          size_t sz)
```

```
NotEnoughMemoryException(const NotEnoughMemoryException &ex)
```

### **Class**

[NotEnoughMemoryException](#)

---

## Chapter 36. NotSupportedException

The function called is not supported in this class.

### Public Members

Member	Description
<a href="#">NotSupportedException (page 388)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## **NotSupportedException**

### **Function Details**

```
NotSupportedException(const char *str)
```

### **Class**

[NotSupportedException](#)

---

## Chapter 37. InvalidIteratorException

The iterator has inconsistent status, it is unable to be used any more.

### Public Members

Member	Description
<a href="#">InvalidIteratorException (page 390)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## InvalidIteratorException

### Function Details

```
InvalidIteratorException()
```

```
InvalidIteratorException(int error_code)
```

### Class

[InvalidIteratorException](#)

---

## Chapter 38. InvalidFunctionCall

The function can not be called in this context or in current configurations.

### Public Members

Member	Description
<a href="#">InvalidFunctionCall (page 392)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

# InvalidFunctionCall

## Function Details

```
InvalidFunctionCall(const char *str)
```

## Class

[InvalidFunctionCall](#)

---

## Chapter 39. InvalidArgumentException

Some argument of a function is invalid.

### Public Members

Member	Description
<a href="#">InvalidArgumentException (page 394)</a>	

### Group

[Dbstl Exception Classes \(page 370\)](#)

## **InvalidArgumentException**

### **Function Details**

```
InvalidArgumentException(const char *errmsg)
```

```
InvalidArgumentException(const char *argtype,  
const char *arg)
```

### **Class**

[InvalidArgumentException](#)