Oracle Berkeley DB

Getting Started with Berkeley DB for C++

Release 4.7

[illegible]

Table of Contents

ef ce a v i
n n U n h kB v o Ce o ti se d is t io v i
F r m m r n I M o o e fo o ati v
r n I 1 . kB r l DB o t dyo tio e e u 1
h n l A T M o ts i a u u 2
B r l DB n p y e e e o Ce s t 2
h A M ccess e o ts d 4
In h A e eS M ti cgess e o ts d 4
h n b n B rn IH w ooS T ieg et 5e ad s a
h n b n n n R w ooS ieg et eQ adc o u u
D m L n Pbr l atse a sy it ad ta i it 6
E nm n v o i e s t 6
Ep n n H n l x ce o ti ad i g 7
Err r n R r o e ts u 8
n n n U DB e G tti g ad i g 8
. 2 atsesa 9
n n D eO i g atsesa 9
n l D osC i g atsesa 1 0
D p n Fl atse a eO s ag 11
n r h A v Md s i i t etie o ts d 11
Err p R r n F n o e o ti gc o ti u 1
n D n n E nm n M a agivg atsesa i 5 i e s t 1
D nEp l x atse a ae 1 7
R r atse3aeco s d 2 0
U D R r s i g atse aeco s d 2 0
n R n nr D R r W e d i g ad ti g atse aeco s d 1 2
nr R r h D W tiego s do t e t atse a 1 2
n R r m r h D e G ttiego s flo e t atse a 22
D n l R r e e tiego s d 2
D P r n atass e icte 2
D U nEp l x atse as e ag ae 4 2
r r . 4 s i gs6 s u
n n n l r r eO i g ados C i gs6 s u
n R r n U h r r e G ttiego s ds i gt s6 u 4
h m r R r e S a fog eco s d5
k n r h p D l R r W w o i g it c i ateco sud 8
P h R r n U r r ttiego s ds i gs6 s u 4 0
D n l R r n U r r e e tiego s ds i gs6 s u 4 2
R l R r n U r r e c a iego s ds i gs6 s u 4
r r nEp l x s6 ae u 44
r D . y ecoS d a atsesa 4 9
n n n n l n r D eO i g adys C iego S 5 d a atsesa 0
m l n n I K E r r y ex e tieg cto5st 1
k n r h p l l K W w M i y it e5 e s u 2
n R n r D e y d iego S d a at5esa
D n l n r D R r e y tiego S d a at5 aeco s d 4

U r r h n r D ws i gsC sy .... itecoSu 5d a ..... atsesa 6
D n .... atse ao s i 5 7
nU n r r .... sJ iog i s6 s 5 u 7
n r D mEp l y .... ecoS xd a atse a 5e 9
n r D h mp l b y l ecoS wd a x _atsesa ..... _et ..... ae ..... d atse ao d 0 6
n r D h mp l b y r ecoS wd a x _atsesa ..... _et ..... ae ..... d atse ae d 4 6
n . m .... 6 atse ao fC ig oati u 6 8
n h P .... ze S tti gt e ag eSi 6 8
r l P .... v w .... eOo esag 6 8
kLn .... oc i g 9 6
E n I .... y fOc e ici 0 7
P n .... z A v e ag Si i g de i 0 7
ln h h .... e eSz ti gt c Ga eSi 1 7
B r n m T .... ee o fC ig oati u 1 7
lh pD l R r A w .... o i g c e ateco sud 2 7
r pD l .... o S td c esat u 2 7
n U r pD l .... so e td c esat u 2 7
n n r D pp rpD l o fC ig i g a atseuao t So ..... t c esat u 7
n mp mr n F n .... e S ttiog C soa i c o ti u ... 4 7
m mp mr n F n .... e C atiog C Soa i c o ti ..... u 7

Preface

[illegible]

Conventions Used in this Book

[illegible]

[illegible] `method names` [illegible] `Db::open()` [illegible] `Db` [illegible]

[illegible] *DB_INSTALL* [illegible]

[illegible]

```
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;
```

[illegible]

```
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
    char sales_rep[MAXFIELD];        // Name of sales representative
    char sales_rep_phone[MAXFIELD];  // Sales rep's phone number
} VENDOR;
```

[illegible] ☞ [illegible]

For More Information

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

- Getting Started with Transaction Processing for C++ [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_txn/CXX/index.html]
- Berkeley DB Getting Started with Replicated Applications for C++ [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_db_rep/CXX/index.html]
- Berkeley DB Programmer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html]
- Berkeley DB C++ API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_cxx/frame.html]

Chapter 1. Introduction to Berkeley DB

[illegible]

[illegible]

[illegible]

About This Manual

[illegible]

[illegible]

[illegible]

[illegible]

```
DB_INSTALL/examples_cxx/getting_started
```

[illegible] *DB_INSTALL* [illegible]

[illegible]

Berkeley DB Concepts

[illegible]

[illegible] *record's key* [illegible] *record's data* [illegible]

[illegible]

[illegible]

[illegible]

I r n , l DB l n y r n h ey se af e eq b t a cu r io at si e a e sd e ig sd t at a e ig d at se sa o et s a

mr r mr m r *environment*, E nm n a p o (o e f) iv . oatisee o i e s t e ag 6

b lk r ll b T h kl e se m *methods* me a h e a rd y ec sea e t oo sou c e i e a ou t o d a C

br H . h T m h c ss a e y n a re a rr t o at se o t c e ss u es te e o t s d o fie *handle* e d s a a

r r mr b m v m e l et i i g e a o f d b b a d at se a s *getting the record* d ec sea e t

m m ll b eco s d s *putting the record* d y eputs( ) a o se e tu ue u o t d t o d

r p r , r , r W by n e h n o s ol e t o r, u t e a o rd t u a d at se as is g it a du e t e co s ds io e t d

I pp r p l r rh n , fn a d at se b a h n s t d k e y a t e co s d e t o c a e a d u at se a v a d o e e t e i

n I n b h n l n , l r a do b iti o s r i g a y at s e p u a a d l o c w a s o a e a d a d e it d at a i s g e c i a

m ll r r r n . ll *cursors* a c e a rd a k s6 b y e e s se u t y a e it o s at t w at c v a e o t u a e u

pDB r, p k nl y b v ll i a o e s d s e c i a o d r se *database* a y ec o S d a

[illegible]

Access Methods

[illegible]

[illegible]

[illegible]

[illegible]

h	A M ccess e o d i i estc o	p
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Selecting Access Methods

[illegible]

h m h n m y, h v b ceO on he a reyB rsut eichsHoiior ,ws c tbose eu Teut e eit ee o s ao
rn Rh n . b r n eQe o ec o u su eidsoixs eist e idt t

Choosing between BTree and Hash

llk n r h n rnlmwm o sh aon, i ghsetsbyt tfateityeti ie o etes oi wfeteie ce e eet
lH hB p ll m.r T llh ehe wnachs a o jlt .eiwfon h, s sta s aeutb et y stsi itoatio uigt
ll B r n r, h j r rnwh n h smTstae rse uDPe ffoinuoo e t e soaj y t aet o a oft ac ioatise
. T ee

h m n n n hr r rk n r oe nt t aet,nrcayice w e ens oi o y.i gdsetatoc ot e Myti dsetau t a
n m n n l rm n n m r n b ac ioati ai tai eagoa soltl foiy ioati o t ee d utcessoes a
h h n rn h o owof nyt atd athay it fawh eyc w o S uat aotcto s eaids iet d atat at
llm l n h ,m y wl hll oy m n oi beti use pp uetsn uo tofa aetydata aeagd o ac ioatiu

r ,rk n r rwv h p yn ohwe rs oann o w igdsutata oesw o tyeto iteeo c oa f t ititu aoit
h n n, k m r hyn h yre o etom he obtwe taouecleyae,coos iog cceass e yotudeCSfc ii a
: c oose

r k r h m • l l T ryn r hyeevf oih es, yesonlle oc aoT efe e ceyswsatfi ietyso et aod
p h r r n k x klll bct ea bsc b t tvatae yfowr aegiue wie iy e fo oye d ae fo o e of u
hb r . s ie iogs

r mr• l l r Fnyr n s .afmexi h dyetamBtiu,me teynv ea go a egiccess e ot d s t aitai u
m m n n n rn l m r n H heam.tpaioa nofm ein hafovi oatio ee et oa oft foi ouati t at
B m n n rBrm h r hm rHs hTt aitfahu ees icregeat Ttufa s a e es stitsatau
r h n n, rn l m r n n mw dhsetaobs hstpienit h foir oatc aod eatetc cewo teto itee
r r l l lpl l pp l n v yetes eir ebtiB.reibb ceaefrfotA ac ioati dTtsa esa teec aeufoce d
m r k m h m I/ r rn hl n o teH nschi mOcmyorefewequ t tvao u ds a egiuets ea oa t
. of d ata

r r ,r b m M lvrh y DBeoHelfoi mr ohsetaetoues sow ea g t at yi oas cet tai e ao t
m r k I/ n r m r h n eHo ydlni oOst pfati maoade eepv tet syau eifd eiito etfo u
b h n r n r lT r hee eehw hsea isfeaBu eit eaco sdo.se c a ot T g taeds uee

Choosing between Queue and Recno

rn R r h n h pp l n n eQe o eedo ulerasenrmbe wturac ioati saot te o c giecou d esfo u
r r b k L lr nrmby retrnyiattatseaehr onc giecod d e y eassseu tia eietysg t at ej
h b r rh n b y h. m r ebif tiert dTatyseaeo bd e lc ae e eit w eao f e iduee a tea
mb m h m h h n b ecor dr er so ie rt at igF eas ad atseaeco sdesaoetxod eeee td e id
lr nrmb n r h rn r r l b c giebov o e smee cru eaweg gads of atd atse ae oati e a
r . e fo e d

n b n n Wn Rh , ew eed di ieg eet eQe adec o uc woose

pp l n •r h r h yr neQe frroi uapuioatieu.qesi igdeegyofcoc evc eQe uo esd u u
r l - kln pp h(p vl - kechcoh ehe ob irsgoaoshevod tet e aege oc igt aeto et cceass e otsd
n h , n) r nl n n l hr ebp adsntciulms ynmtri fig iiuafts eat oty g fot iscocu e t u
l n . ac ioati

Database Limits and Portability

Environments

[illegible] *must* [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] *Berkeley DB Getting Started with Transaction Processing* [illegible] *Berkeley DB Getting Started with Replicated Applications* [illegible]

Exception Handling

[illegible] C++ [illegible]

[illegible] `DbException` [illegible] `DbException` [illegible] try [illegible]

```
#include <db_cxx.h>
    ...
try
{
    // DB and other code goes here
}
catch(DbException &e)
{
  // DB error handling goes here
}
catch(std::exception &e)
{
    // All other error handling goes here
}
```

[illegible] DbException [illegible] DbException::get_errno() [illegible] DbException::what()

[illegible] DbException [illegible] DB_CXX_NO_EXCEPTIONS [illegible] DbException [illegible]

Error Returns

[illegible]

[illegible]

[illegible] DB_NOTFOUND [illegible]

Getting and Using DB

[illegible]

[illegible] DB_INSTALL/docs/index.html [illegible] DB_INSTALL [illegible]

[illegible] *Berkeley DB Programmer's Reference Guide* [illegible]

Chapter 2. Databases

[illegible]

[illegible]

Opening Databases

[illegible]

[illegible] `DB_CREATE` [illegible] `open()` [illegible]

[illegible]

```
#include <db_cxx.h>

...

Db db(NULL, 0);               // Instantiate the Db object

u_int32_t oFlags = DB_CREATE; // Open flags;

try {
    // Open the database
    db.open(NULL,                // Transaction pointer
            "my_db.db",          // Database file name
            NULL,                // Optional logical database name
            DB_BTREE,            // Database access method
            oFlags,              // Open flags
            0);                  // File mode (using defaults)
// DbException is not subclassed from std::exception, so
// need to catch both of these.
} catch(DbException &e) {
    // Error handling code goes here
} catch(std::exception &e) {
    // Error handling code goes here
}
```

Closing Databases

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>

...

Db db(NULL, 0);

 // Database open and access operations happen here.

try {
    // Close the database
    db.close(0);
// DbException is not subclassed from std::exception, so
// need to catch both of these.
} catch(DbException &e) {
    // Error handling code goes here
} catch(std::exception &e) {
    // Error handling code goes here
}
```

Database Open Flags

[illegible] *Berkeley DB C++ API guide.*

[illegible] `Db::open()` [illegible]

```
u_int32_t open_flags = DB_CREATE | DB_EXCL;
```

- `DB_CREATE`

[illegible]

- `DB_EXCL`

[illegible] `DB_CREATE` [illegible]

- `DB_RDONLY`

[illegible]

- `DB_TRUNCATE`

[illegible]

Administrative Methods

[illegible]

- `Db::get_open_flags()`

[illegible]

```
#include <db_cxx.h>
...
Db db(NULL, 0);
u_int32_t open_flags;

// Database open and subsequent operations omitted for clarity

db.get_open_flags(&open_flags);
```

- `Db::remove()`

  `Db::remove()` removes the specified database. If no value is given for the *database* parameter, then the entire file referenced by this method is removed.

  Never remove a database that has handles opened for it. Never remove a file that contains databases with opened handles.

```
#include <db_cxx.h>
...
Db db(NULL, 0);

// Database handle creation omitted for clarity

db.remove("mydb.db",             // Database file to remove
          NULL,                  // Database to remove. This is
                                 // NULL so the entire file is
                                 // removed.
         0);                     // Flags. None used.
```

- `Db::rename()`

  `Db::rename()` renames the specified database. If no value is given for the *database* parameter, then the entire file referenced by this method is renamed.

  Never rename a database that has handles opened for it. Never rename a file that contains databases with opened handles.

```
#include <db_cxx.h>
...
Db db(NULL, 0);

// Database handle creation omitted for clarity

db.rename("mydb.db",             // Database file to rename
          NULL,                  // Database to rename. This is
                                 // NULL so the entire file is
                                 // renamed.
         "newdb.db",             // New database file name
         0);                     // Flags. None used.
```

Error Reporting Functions

[illegible]

- `set_error_stream()`

[illegible] `ostream` [illegible]

- `set_errcall()`

[illegible]

- `set_errfile()`

[illegible] `FILE *` [illegible]

- `set_errpfx()`

[illegible]

- `err()`

[illegible] `set_errcall` [illegible] `set_errfile()` [illegible] `set_error_stream()` [illegible]

[illegible] `set_errpfx()` [illegible] `printf` [illegible]

- `errx()`

[illegible] `err()` [illegible]

[illegible] `db_strerror()` [illegible]

[illegible]

```
/*
 * Function called to handle any database error messages
 * issued by DB.
 */
void
my_error_handler(const char *error_prefix, char *msg)
{
  /*
   * Put your code to handle the error prefix and error
   * message here. Note that one or both of these parameters
   * may be NULL depending on how the error message is issued
   * and how the DB handle is configured.
   */
}
```

n h n r hr b lk A ll de:t e segitetc w ac asfoa o s

```
#include <db_cxx.h>
...

Db db(NULL, 0);
std::string dbFileName("my_db.db");

try
{
    // Set up error handling for this database
    db.set_errcall(my_error_handler);
    db.set_errpfx("my_example_program");
```

n n mr A : d sts e ea o essu e ag

```
    // Open the database
    db.open(NULL, dbFileName.c_str(), NULL, DB_BTREE, DB_CREATE, 0);
}
    // Must catch both DbException and std::exception
    catch(DbException &e)
    {
        db.err(e.get_errno(), "Database open failed %s",
            dbFileName.c_str());
        throw e;
    }
    catch(std::exception &e)
    {
        // No DB error number available, so use errx
        db.errx("Error opening database: %s", e.what());
        throw e;
    }
```

Managing Databases in Environments

[illegible]

[illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>
...
u_int32_t env_flags = DB_CREATE |     // If the environment does not
                                      // exist, create it.
                      DB_INIT_MPOOL; // Initialize the in-memory cache.

std::string envHome("/export1/testEnv");
DbEnv myEnv(0);

try {
    myEnv.open(envHome.c_str(), env_flags, 0);
} catch(DbException &e) {
    std::cerr << "Error opening database environment: "
              << envHome << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error opening database environment: "
              << envHome << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

[illegible]

```
#include <db_cxx.h>
...
u_int32_t env_flags = DB_CREATE |  // If the environment does not
                                  // exist, create it.
                      DB_INIT_MPOOL; // Initialize the in-memory cache.
std::string envHome("/export1/testEnv");
```

```
u_int32_t db_flags = DB_CREATE;   // If the database does not
                                  // exist, create it.
std::string dbName("mydb.db");
DbEnv myEnv(0);
Db *myDb;

try {
    myEnv.open(envHome.c_str(), env_flags, 0);
    myDb = new Db(&myEnv, 0);
    myDb->open(NULL,
               dbName.c_str(),
               NULL,
               DB_BTREE,
               db_flags,
               0);
} catch(DbException &e) {
    std::cerr << "Error opening database environment: "
              << envHome
              << " and database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error opening database environment: "
              << envHome
              << " and database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

h n rn h n n nmWh y m , e lwo eBavde utnyea o i e d s cydse uefoue o cose a
n nmn m k , r l n vp n be o i ey t e a ye. o cose ua e u d d atsesa

```
try {
    if (myDb != NULL) {
        myDb->close(0);
    }
    myEnv.close(0);

} catch(DbException &e) {
    std::cerr << "Error closing database environment: "
              << envHome
              << " or database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error closing database environment: "
```

```
                << envHome
                << " or database "
                << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

Database Example

Throughout this book we will build a couple of applications that load and retrieve inventory data from DB databases. While we are not yet ready to begin reading from or writing to our databases, we can at least create some important structures and functions that we will use to manage our databases.

Note that subsequent examples in this book will build on this code to perform the more interesting work of writing to and reading from the databases.

Note that you can find the complete implementation of these functions in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 2.1. MyDb Class

To manage our database open and close activities, we encapsulate them in the `MyDb` class. There are several good reasons to do this, the most important being that we can ensure our databases are closed by putting that activity in the `MyDb` class destructor.

To begin, we create our class definition:

```
// File: MyDb.hpp
#include <db_cxx.h>

class MyDb
{
public:
    // Constructor requires a path to the database,
    // and a database name.
    MyDb(std::string &path, std::string &dbName);

    // Our destructor just calls our private close method.
    ~MyDb() { close(); }

    inline Db &getDb() {return db_;}

private:
    Db db_;
    std::string dbFileName_;
    u_int32_t cFlags_;
    // Make sure the default constructor is private
    // We don't want it used.
    MyDb() : db_(NULL, 0) {}

    // We put our database close activity here.
    // This is called from our destructor. In
    // a more complicated example, we might want
    // to make this method public, but a private
    // method is more appropriate for this example.
    void close();
};
```

N n h mp m ln n bxr wn re ret ee de t :ie e oatfio etos cto t u

```
// File: MyDb.cpp
#include "MyDb.hpp"

// Class constructor. Requires a path to the location
// where the database is located, and a database name
MyDb::MyDb(std::string &path, std::string &dbName)
    : db_(NULL, 0),               // Instantiate Db object
      dbFileName_(path + dbName), // Database file name
      cFlags_(DB_CREATE)          // If the database doesn't yet exist,
                                  // allow it to be created.
{
    try
    {
        // Redirect debugging information to std::cerr
        db_.set_error_stream(&std::cerr);

        // Open the database
        db_.open(NULL, dbFileName_.c_str(), NULL, DB_BTREE, cFlags_, 0);
    }
    // DbException is not a subclass of std::exception, so we
    // need to catch them both.
    catch(DbException &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

n h n n h mp m ln A n h r w mdeht e ee de t ie `close()` oatfio e t e o t d

```
// Private member used to close a database. Called from the class
// destructor.
void
MyDb::close()
{
    // Close the db
    try
    {
        db_.close(0);
        std::cout << "Database " << dbFileName_
                  << " is closed." << std::endl;
    }
    catch(DbException &e)
    {
        std::cerr << "Error closing database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error closing database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

Chapter 3. Database Records

[illegible]

[illegible]

[illegible]

Using Database Records

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

float money = 122.45;
char *description = "Grocery bill.";

Dbt key(&money, sizeof(float));
Dbt data(description, strlen(description)+1);
```

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

Dbt key, data;
float money;
char *description;

key.set_data(&money);
key.set_ulen(sizeof(float));
       key.set_flags(DB_DBT_USERMEM);

       // Database retrieval code goes here

       // Money is set into the memory that we supplied.
       description = (char *)data.get_data();
```

Reading and Writing Database Records

[illegible]

[illegible]

[illegible]

[illegible] `Db::put()` [illegible] `Db::get()` [illegible]

[illegible]

Writing Records to the Database

[illegible]

[illegible]

[illegible] `Db::put()` [illegible]

[illegible] `DB_NOOVERWRITE` [illegible] `DB_KEYEXIST` [illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

char *description = "Grocery bill.";
float money = 122.45;

Db my_database(NULL, 0);
// Database open omitted for clarity

Dbt key(&money, sizeof(float));
Dbt data(description, strlen(description) + 1);

int ret = my_database.put(NULL, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    my_database.err(ret, "Put failed because key %f already exists", money);
}
```

Getting Records from the Database

You can use the `Db::get()` method to retrieve database records. Note that if your database supports duplicate records, then by default this method will only return the first record in a duplicate set. For this reason, if your database supports duplicates, the common solution is to use a cursor to retrieve records from it. Cursors are discussed in Using Cursors (page 33).

(You can also retrieve a set of duplicate records using a bulk get. To do this, you use the `DB_MULTIPLE` flag on the call to `Db::get()`. For more information, see the DB Programmer's Reference Guide).

By default, `Db::get()` returns the first record found whose key matches the key provided on the call to `Db::get()`. If your database supports duplicate records, you can change this behavior slightly by supplying the `DB_GET_BOTH` flag. This flag causes `Db::get()` to return the first record that matches the provided key and data.

If the specified key and/or data does not exist in the database, this method returns `DB_NOTFOUND`.

```
#include <db_cxx.h>
#include <string.h>

...

float money;
char description[DESCRIPTION_SIZE + 1];

Db my_database(NULL, 0);
// Database open omitted for clarity
```

```
money = 122.45;

Dbt key, data;

key.set_data(&money);
key.set_size(sizeof(float));

data.set_data(description);
data.set_ulen(DESCRIPTION_SIZE + 1);
data.set_flags(DB_DBT_USERMEM);

my_database.get(NULL, &key, &data, 0);

// Description is set into the memory that we supplied.
```

[illegible]

Deleting Records

[illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
// Database open omitted for clarity

float money = 122.45;
Dbt key(&money, sizeof(float));

my_database.del(NULL, &key, 0);
```

Data Persistence

[illegible]

[illegible]

[illegible]

[illegible] *Berkeley DB Getting Started with Transaction Processing* [illegible]

[illegible]

[illegible] `Db::sync()` [illegible]

[illegible] `DB_NOSYNC` [illegible] `Db::close()` [illegible] `Db::sync()` [illegible]

[illegible]

Database Usage Example

[illegible]

[illegible]

`DB_INSTALL/examples_cxx/getting_started`

[illegible] *DB_INSTALL* [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
// File: gettingStartedCommon.hpp
#define MAXFIELD 20
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
    char sales_rep[MAXFIELD];        // Name of sales representative
    char sales_rep_phone[MAXFIELD];  // Sales rep's phone number
} VENDOR;
```

[illegible]

[illegible]

[illegible]

[illegible] `gettingStartedCommon.hpp` [illegible] `VENDOR` [illegible]

[illegible]

```
class InventoryData
{
public:
    inline void setPrice(double price) {price_ = price;}
    inline void setQuantity(long quantity) {quantity_ = quantity;}
    inline void setCategory(std::string &category) {category_ = category;}
    inline void setName(std::string &name) {name_ = name;}
    inline void setVendor(std::string &vendor) {vendor_ = vendor;}
    inline void setSKU(std::string &sku) {sku_ = sku;}

    inline double& getPrice() {return(price_);}
    inline long& getQuantity() {return(quantity_);}
    inline std::string& getCategory() {return(category_);}
    inline std::string& getName() {return(name_);}
    inline std::string& getVendor() {return(vendor_);}
    inline std::string& getSKU() {return(sku_);}

    // Initialize our data members
    void clear()
    {
        price_ = 0.0;
        quantity_ = 0;
        category_.clear();
        name_.clear();
        vendor_.clear();
        sku_.clear();
    }
```

[illegible]

```
    // Default constructor
    InventoryData() { clear(); }

    // Constructor from a void *
    // For use with the data returned from a bdb get
    InventoryData(void *buffer)
    {
        char *buf = (char *)buffer;

        price_ = *((double *)buf);
        bufLen_ = sizeof(double);

        quantity_ = *((long *)(buf + bufLen_));
        bufLen_ += sizeof(long);

        name_ = buf + bufLen_;
        bufLen_ += name_.size() + 1;

        sku_ = buf + bufLen_;
        bufLen_ += sku_.size() + 1;

        category_ = buf + bufLen_;
        bufLen_ += category_.size() + 1;

        vendor_ = buf + bufLen_;
        bufLen_ += vendor_.size() + 1;
    }
```

[illegible]

```
    // Marshalls this classes data members into a single
    // contiguous memory location for the purpose of storing
    // the data in a database.
    char *
    getBuffer()
    {
        // Zero out the buffer
        memset(databuf_, 0, 500);
        // Now pack the data into a single contiguous memory location for
        // storage.
        bufLen_ = 0;
        int dataLen = 0;

        dataLen = sizeof(double);
        memcpy(databuf_, &price_, dataLen);
        bufLen_ += dataLen;

        dataLen = sizeof(long);
        memcpy(databuf_ + bufLen_, &quantity_, dataLen);
        bufLen_ += dataLen;

        packString(databuf_, name_);
        packString(databuf_, sku_);
        packString(databuf_, category_);
        packString(databuf_, vendor_);

        return (databuf_);
    }

    // Returns the size of the buffer. Used for storing
    // the buffer in a database.
    inline int getBufferSize() { return (bufLen_); }
```

[illegible]

```
     // Utility function used to show the contents of this class
    void
    show() {
        std::cout << "\nName:         " << name_ << std::endl;
        std::cout << "    SKU:        " << sku_ << std::endl;
        std::cout << "    Price:      " << price_ << std::endl;
        std::cout << "    Quantity:   " << quantity_ << std::endl;
        std::cout << "    Category:   " << category_ << std::endl;
        std::cout << "    Vendor:     " << vendor_ << std::endl;
    }
```

[illegible]

```
private:

    // Utility function that appends a char * to the end of
    // the buffer.
    void
    packString(char *buffer, std::string &theString)
    {
        int string_size = theString.size() + 1;
        memcpy(buffer+bufLen_, theString.c_str(), string_size);
        bufLen_ += string_size;
    }

    // Data members
    std::string category_, name_, vendor_, sku_;
    double price_;
    long quantity_;
    int bufLen_;
    char databuf_[500];
};
```

3.3 [illegible]

[illegible]

```
DB_INSTALL/examples_cxx/getting_started
```

[illegible] *DB_INSTALL* [illegible] DB [illegible]

[illegible]

```
// File: example_database_load.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
void loadVendorDB(MyDb&, std::string&);
void loadInventoryDB(MyDb&, std::string&);
```

[illegible] `main()` [illegible]

```
// Loads the contents of vendors.txt and inventory.txt into
// Berkeley DB databases.
int
main(int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string basename("./");
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");

    // Parse the command line arguments here and determine
    // the location of the flat text files containing the
    // inventory data here. This step is omitted for clarity.

    //  Identify the full name for our input files, which should
    //  also include some path information.
    std::string inventoryFile = basename + "inventory.txt";
    std::string vendorFile = basename + "vendors.txt";

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);

        // Load the vendor database
        loadVendorDB(vendorDB, vendorFile);

        // Load the inventory database
        loadInventoryDB(inventoryDB, inventoryFile);
    } catch(DbException &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

[illegible]

[illegible]

[illegible] `loadVendorDB()` [illegible]

[illegible]

```
// Loads the contents of the vendors.txt file into a database
void
loadVendorDB(MyDb &vendorDB, std::string &vendorFile)
{
    std::ifstream inFile(vendorFile.c_str(), std::ios::in);
    if ( !inFile )
    {
        std::cerr << "Could not open file '" << vendorFile
                  << "'. Giving up." << std::endl;
        throw std::exception();
    }

    VENDOR my_vendor;
    while (!inFile.eof())
    {
        std::string stringBuf;
        std::getline(inFile, stringBuf);
        memset(&my_vendor, 0, sizeof(VENDOR));

        // Scan the line into the structure.
        // Convenient, but not particularly safe.
        // In a real program, there would be a lot more
        // defensive code here.
        sscanf(stringBuf.c_str(),
          "%20[^#]#%20[^#]#%20[^#]#%3[^#]#%6[^#]#%13[^#]#%20[^#]#%20[^\n]",
          my_vendor.name, my_vendor.street,
          my_vendor.city, my_vendor.state,
          my_vendor.zipcode, my_vendor.phone_number,
          my_vendor.sales_rep, my_vendor.sales_rep_phone);

        Dbt key(my_vendor.name, strlen(my_vendor.name) + 1);
        Dbt data(&my_vendor, sizeof(VENDOR));

        vendorDB.getDb().put(NULL, &key, &data, 0);
    }
    inFile.close();
}
```

[illegible] loadInventoryDB() [illegible]

[illegible]

[illegible] END [illegible] InventoryData [illegible]

[illegible]

```
// Used to locate the first pound sign (a field delimiter)
// in the input string.
int
getNextPound(std::string &theString, std::string &substring)
{
    int pos = theString.find("#");
    substring.assign(theString, 0, pos);
    theString.assign(theString, pos + 1, theString.size());
    return (pos);
}


// Loads the contents of the inventory.txt file into a database
void
loadInventoryDB(MyDb &inventoryDB, std::string &inventoryFile)
{
    InventoryData inventoryData;
    std::string substring;
    int nextPound;

    std::ifstream inFile(inventoryFile.c_str(), std::ios::in);
    if (!inFile)
    {
        std::cerr << "Could not open file '" << inventoryFile
                  << "'. Giving up." << std::endl;
        throw std::exception();
    }

    while (!inFile.eof())
    {
        inventoryData.clear();
        std::string stringBuf;
        std::getline(inFile, stringBuf);

        // Now parse the line
        if (!stringBuf.empty())
        {
            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setName(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setSKU(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setPrice(strtod(substring.c_str(), 0));

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setQuantity(strtol(substring.c_str(), 0, 10));

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setCategory(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setVendor(substring);

            void *buff = (void *)inventoryData.getSKU().c_str();
            int size = inventoryData.getSKU().size()+1;
            Dbt key(buff, size);

            buff = inventoryData.getBuffer();
            size = inventoryData.getBufferSize();
            Dbt data(buff, size);

            inventoryDB.getDb().put(NULL, &key, &data, 0);
        }
    }
    inFile.close();
}
```

h n hl p rp r n mp hlx h hetwe cv tr hea txe n o emi ea w æ w tsat s o v o ey adet ei o t
n r b .v e o d d atsesa

Chapter 4. Using Cursors

[illegible]

[illegible]

Opening and Closing Cursors

[illegible] `Db::cursor()` [illegible]

[illegible]

```
#include <db_cxx.h>

...

Dbc *cursorp;
Db my_database(NULL, 0);

// Database open omitted for clarity

// Get a cursor
my_database.cursor(NULL, &cursorp, 0);
```

[illegible] `Dbc::close()` [illegible]

```
#include <db_cxx.h>

...

Dbc *cursorp;
Db my_database(NULL, 0);

// Database and cursor open omitted for clarity

if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Getting Records Using the Cursor

r rb r r mTh , rv ro rehiteoatl mpd atse,aeho sdf or ref s ieco dytets at ioe ets
h hn h mh Nh Dbc::adet()sep eh u e lyot doe t toay DB_NEXT t uet uf
m h Frmp l . : sx e ot doe æ

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    Dbt key, data;
    int ret;

    // Iterate over the database, retrieving each record in turn.
    while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0) {
        // Do interesting things with the Dbts here.
    }
    if (ret != DB_NOTFOUND) {
        // ret should be DB_NOTFOUND upon exiting the loop.
        // Dbc::get() will by default throw an exception if any
        // significant errors occur, so by default this if block
        // can never be reached.
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

[illegible] DB_PREV [illegible]: DB_NEXT

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    Dbt key, data;
    int ret;
    // Iterate over the database, retrieving each record in turn.
    while ((ret = cursorp->get(&key, &data, DB_PREV)) == 0) {
        // Do interesting things with the Dbts here.
    }
    if (ret != DB_NOTFOUND) {
        // ret should be DB_NOTFOUND upon exiting the loop.
        // Dbc::get() will by default throw an exception if any
        // significant errors occur, so by default this if block
        // can never be reached.
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Searching for Records

[illegible]

[illegible] DB_NOTFOUND [illegible]

r r h r r r Tr b D, o shencasa:()phsetumc afu eao q,We y et g tue o sey st ie outudo
n p r h lh l v : c aow edi e foo o fi gs ag

N n h lh lh h r r l ho kketi ie foo o h n hsg i t t aetr so fy wSEE e uewtze o ud e et so
m n h k p nr h x r rn jh e yahs(s ,erter o lu toif reterco s d istc ise a et so ssei ottuteco
h k m h h pb r w h y r ose evr h)aev h e, t ear or he d d Mtuetz sow o eo e u e et so ses e
k p r h n h , r pr ny wGETb h he k o dch e t et slp s mis o ieanyd b te te v etv d ata ea o ed d
h r r . et so u

R r l h k r r reh y geds oy hre t e ,or krd nse owe g teeou d itc aso y e t so u's'e adud
r ll h h rDbt hn r h rw s r ehf a he hi d viter t pd aeraneteif fcb we tedo .d t c i et so s ois oeieo d

- DB_SET

h M o es e t

```
Alabama/Athens
Alabama/Florence
Alaska/Anchorage
Alaska/Fairbanks
Arizona/Avondale
Arizona/Florence
```

[illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Search criteria
    char *search_key = "Alaska";
    char *search_data = "Fa";

    // Set up our DBTs
    Dbt key(search_key, strlen(search_key) + 1);
    Dbt data(search_data, strlen(search_data) + 1);

    // Position the cursor to the first record in the database whose
    // key matches the search key and whose data begins with the search
    // data.
    int ret = cursorp->get(&key, &data, DB_GET_BOTH_RANGE);
    if (!ret) {
        // Do something with the data
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Close the cursor
if (cursorp != NULL)
    cursorp->close();

// Close the database
my_database.close(0);
```

Working with Duplicate Records

[illegible]

[illegible]

[illegible]

- `DB_NEXT`, `DB_PREV`

[illegible]

- `DB_GET_BOTH_RANGE`

[illegible]

- `DB_NEXT_NODUP`, `DB_PREV_NODUP`

[illegible] `DB_PREV_NODUP` [illegible]

```
Alabama/Athens
Alabama/Florence
Alaska/Anchorage
Alaska/Fairbanks
```

```
Arizona/Avondale
Arizona/Florence
```

[illegible] Alaska/Fairbanks [illegible] `Dbc::get()` [illegible] `DB_PREV_NODUP` [illegible] `Dbc::get()` [illegible] `DB_NEXT_NODUP` [illegible]

[illegible] `DB_NOTFOUND` [illegible]

- `DB_NEXT_DUP`

[illegible] `Dbc::get()` [illegible] `DB_NEXT_DUP` [illegible] `DB_NOTFOUND` [illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

char *search_key = "Al";

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(search_key, strlen(search_key) + 1);
    Dbt data;

    // Position the cursor to the first record in the database whose
    // key and data begin with the correct strings.
    int ret = cursorp->get(&key, &data, DB_SET);
    while (ret != DB_NOTFOUND) {
        std::cout << "key: " << (char *)key.get_data()
                  << "data: " << (char *)data.get_data()<< std::endl;
        ret = cursorp->get(&key, &data, DB_NEXT_DUP);
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Close the cursor
if (cursorp != NULL)
    cursorp->close();

// Close the database
my_database.close(0);
```

Putting Records Using Cursors

You can use cursors to put records into the database. DB's behavior when putting records into the database differs depending on the flags that you use when writing the record, on the access method that you are using, and on whether your database supports sorted duplicates.

Note that when putting records to the database using a cursor, the cursor is positioned at the record you inserted.

You use `Dbc::put()` to put (write) records to the database. You can use the following flags with this method:

- DB_NODUPDATA

  If the provided key already exists in the database, then this method returns DB_KEYEXIST.

  If the key does not exist, then the order that the record is put into the database is determined by the insertion order in use by the database. If a comparison function has been provided to the database using the DB_DUPSORT flag, then the order is determined by that comparison function. Otherwise, the order is determined by the order of insertion.

  This flag can only be used for the BTree and Hash access methods, and only if the database has been configured to support sorted duplicate data items (DB_DUPSORT was specified at database creation time).

  This flag cannot be used with the Queue or Recno access methods.

  For more information on duplicate records, see Allowing Duplicate Records (page 62).

- DB_KEYFIRST

  For databases that do not support duplicates, this method behaves exactly the same as if a default insertion was performed. If the database supports duplicate records, and a duplicate sort function has been specified, the inserted data item is added in the position appropriate to the sort function. If the key specified does not already exist in the database

[illegible]

- DB_KEYLAST

[illegible] DB_KEYFIRST [illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

char *key1str = "My first string";
char *data1str = "My first data";
char *key2str = "A second string";
char *data2str = "My second data";
char *data3str = "My third data";


Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Set up our DBTs
    Dbt key1(key1str, strlen(key1str) + 1);
    Dbt data1(data1str, strlen(data1str) + 1);

    Dbt key2(key2str, strlen(key2str) + 1);
    Dbt data2(data2str, strlen(data2str) + 1);
    Dbt data3(data3str, strlen(data3str) + 1);

    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Assuming an empty database, this first put places
    // "My first string"/"My first data" in the first
    // position in the database
    int ret = cursorp->put(&key1, &data1, DB_KEYFIRST);

    // This put places "A second string"/"My second data" in the
    // the database according to its key sorts against the key
    // used for the currently existing database record. Most likely
    // this record would appear first in the database.
    ret = cursorp->put(&key2, &data2,
            DB_KEYFIRST); /* Added according to sort order */

    // If duplicates are not allowed, the currently existing record that
    // uses "key2" is overwritten with the data provided on this put.
    // That is, the record "A second string"/"My second data" becomes
    // "A second string"/"My third data"
    //
    // If duplicates are allowed, then "My third data" is placed in the
    // duplicates list according to how it sorts against "My second data".
    ret = cursorp->put(&key2, &data3,
            DB_KEYFIRST); // If duplicates are not allowed, record
                          // is overwritten with new data. Otherwise,
                          // the record is added to the beginning of
                          // the duplicates list.
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Deleting Records Using Cursors

[illegible] Dbc::del() [illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

char *key1str = "My first string";
Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(key1str, strlen(key1str) + 1);
    Dbt data;

    // Iterate over the database, deleting each record in turn.
    int ret;
    while ((ret = cursorp->get(&key, &data,
                              DB_SET)) == 0) {
        cursorp->del(0);
    }

} catch(DbException &e) {
    my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
    my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Replacing Records Using Cursors

l h r b Y r br n o e ce a h th d fo ta ay at Dbc::put() d s w i g DB_CURRENT ite t f ag

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

int ret;
char *key1str = "My first string";
char *replacement_data = "replace me";

try {
    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(key1str, strlen(key1str) + 1);
    Dbt data;

    // Position the cursor */
    ret = cursorp->get(&key, &data, DB_SET);
    if (ret == 0) {
        data.set_data(replacement_data);
        data.set_size(strlen(replacement_data) + 1);
        cursorp->put(&key, &data, DB_CURRENT);
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

[illegible]

[illegible]

[illegible]

Cursor Example

[illegible]

[illegible] `example_database_read` [illegible]

[illegible] `example_database_load` [illegible]

[illegible]

[illegible]

```
DB_INSTALL/examples_cxx/getting_started
```

[illegible] *DB_INSTALL* [illegible]

[illegible]

[illegible]

```
// File: example_database_read.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
int show_all_records(MyDb &inventoryDB, MyDb &vendorDB);
int show_vendor(MyDb &vendorDB, const char *vendor);
```

[illegible] `main()` [illegible]

```
// Displays all inventory items and the associated vendor record.
int
main (int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");

    // Parse the command line arguments
    // Omitted for brevity

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);

        show_all_records(inventoryDB, vendorDB);
    } catch(DbException &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

[illegible] `show_all_records()` [illegible]

```
// Shows all the records in the inventory database.
// For each inventory record shown, the appropriate
// vendor record is also displayed.
int
show_all_records(MyDb &inventoryDB, MyDb &vendorDB)
{
    // Get a cursor to the inventory db
    Dbc *cursorp;
    try {
        inventoryDB.getDb().cursor(NULL, &cursorp, 0);

        // Iterate over the inventory database, from the first record
        // to the last, displaying each in turn
        Dbt key, data;
        int ret;
        while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0 )
        {
            InventoryData inventoryItem(data.get_data());
            inventoryItem.show();

            show_vendor(vendorDB, inventoryItem.getVendor().c_str());
        }
    } catch(DbException &e) {
        inventoryDB.getDb().err(e.get_errno(), "Error in show_all_records");
        cursorp->close();
        throw e;
    } catch(std::exception &e) {
        cursorp->close();
        throw e;
    }

    cursorp->close();
    return (0);
}
```

[illegible]

[illegible]

```
// Shows a vendor record. Each vendor record is an instance of
// a vendor structure. See loadVendorDB() in
// example_database_load for how this structure was originally
// put into the database.
int
show_vendor(MyDb &vendorDB, const char *vendor)
{
    Dbt data;
    VENDOR my_vendor;

    try {
        // Set the search key to the vendor's name
        // vendor is explicitly cast to char * to stop a compiler
        // complaint.
        Dbt key((char *)vendor, strlen(vendor) + 1);

        // Make sure we use the memory we set aside for the VENDOR
        // structure rather than the memory that DB allocates.
        // Some systems may require structures to be aligned in memory
        // in a specific way, and DB may not get it right.

        data.set_data(&my_vendor);
        data.set_ulen(sizeof(VENDOR));
        data.set_flags(DB_DBT_USERMEM);

        // Get the record
        vendorDB.getDb().get(NULL, &key, &data, 0);
        std::cout << "        " << my_vendor.street << "\n"
                  << "        " << my_vendor.city << ", "
                  << my_vendor.state << "\n"
                  << "        " << my_vendor.zipcode << "\n"
                  << "        " << my_vendor.phone_number << "\n"
```

```
                     << "        Contact: " << my_vendor.sales_rep << "\n"
                     << "                 " << my_vendor.sales_rep_phone
                     << std::endl;

    } catch(DbException &e) {
        vendorDB.getDb().err(e.get_errno(), "Error in show_vendor");
        throw e;
    } catch(std::exception &e) {
        throw e;
    }
    return (0);
}
```

[illegible] `example_database_read()` [illegible]

Chapter 5. Secondary Databases

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Opening and Closing Secondary Databases

[illegible]

- [illegible]

- [illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>

...

Db my_database(NULL, 0); // Primary
Db my_index(NULL, 0);    // Secondary

// Open the primary
my_database.open(NULL,       // Transaction pointer
                  "my_db.db", // On-disk file that holds the database.
                 NULL,        // Optional logical database name
                 DB_BTREE,    // Database access method
                 DB_CREATE,   // Open flags
                 0);          // File mode (using defaults)

// Setup the secondary to use sorted duplicates.
// This is often desirable for secondary databases.
my_index.set_flags(DB_DUPSORT);

// Open the secondary
my_index.open(NULL,               // Transaction pointer
              "my_secondary.db", // On-disk file that holds the database.
              NULL,              // Optional logical database name
              DB_BTREE,          // Database access method
              DB_CREATE,         // Open flags.
              0);                // File mode (using defaults)


// Now associate the primary and the secondary
my_database.associate(NULL,          // Txn id
                      &my_index,     // Associated secondary database
                      get_sales_rep, // Callback used for key extraction.
                                     // This is described in the next
```

```
                                    // section.
                     0);            // Flags
```

[illegible]

```
// Close the secondary before the primary
my_index.close(0);
my_database.close(0);
```

Implementing Key Extractors

[illegible]

[illegible]

[illegible]

[illegible]

```
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 */
    char sales_rep[MAXFIELD];        /* Name of sales representative */
    char sales_rep_phone[MAXFIELD];  /* Sales rep's phone number */
} VENDOR;
```

[illegible]

```
#include <db_cxx.h>

...

int
get_sales_rep(Db *sdbp,          // secondary db handle
              const Dbt *pkey,   // primary db record's key
              const Dbt *pdata,  // primary db record's data
              Dbt *skey)         // secondary db record's key
```

```
{
    VENDOR *vendor;

    // First, extract the structure contained in the primary's data
    vendor = (VENDOR *)pdata->get_data();

    // Now set the secondary key's data to be the representative's name
    skey->set_data(vendor->sales_rep);
    skey->set_size(strlen(vendor->sales_rep) + 1);

    // Return 0 to indicate that the record can be created/updated.
    return (0);
}
```

[illegible]

```
db.associate(NULL,            // TXN id
             &sdb,            // Secondary database
             get_sales_rep,       // Callback used for key creation.
             0);                  // Flags
```

Working with Multiple Keys

[illegible]

[illegible]

[illegible] DB_DBT_MULTIPLE [illegible]

[illegible]

```
int
my_callback(Db *dbp, const Dbt *pkey, const Dbt *pdata, Dbt *skey)
{
    Dbt *tmpdbt;
    char *tmpdata1, tmpdata2;

    // This example skips the step of extracting the data you
    // want to use for building your secondary keys from the
    // pkey or pdata Dbt.

    // Assume for the purpose of this example that the data
    // is temporarily stored in two variables,
    // tmpdata1 and tmpdata2.

    // Create an array of Dbts that is large enough for the
    // number of keys that you want to return. In this case,
    // we go with an array of size two.

    tmpdbt = malloc(sizeof(Dbt) * 2);
    memset(tmpdbt, 0, sizeof(Dbt) * 2);

    // Now assign secondary keys to each element of the array.
    tmpdbt[0].set_data(tmpdata1);
    tmpdbt[0].set_size((u_int32_t)strlen(tmpdbt[0].data) + 1);
    tmpdbt[1].set_data(tmpdata2);
    tmpdbt[1].set_size((u_int32_t)strlen(tmpdbt[1].data) + 1);

    // Now we set flags for the returned Dbt. DB_DBT_MULTIPLE is
    // required in order for DB to know that the Dbt references an
    // array. In addition, we set DB_DBT_APPMALLOC because we
    // dynamically allocated memory for the Dbt's data field.
    // DB_DBT_APPMALLOC causes DB to release that memory once it
    // is done with the returned Dbt.
    skey->set_flags(DB_DBT_MULTIPLE | DB_DBT_APPMALLOC);

    // Point the results data field to the arrays of Dbts
    skey->set_data(tmpdbt);

    // Indicate the returned array is of size 2
    skey->size = 2;

    return (0);
}
```

Reading Secondary Databases

[illegible] `Db::get()` [illegible] `Db::pget()` [illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...


// The string to search for
char *search_name = "John Doe";

// Instantiate our Dbt's
Dbt key(search_name, strlen(search_name) + 1);
Dbt pkey, pdata; // Primary key and data


Db my_secondary_database(NULL, 0);
// Primary and secondary database opens omitted for brevity

// Returns the key from the secondary database, and the data from the
// associated primary database entry.
my_secondary_database.get(NULL, &key, &pdata, 0);

// Returns the key from the secondary database, and the key and data
// from the associated primary database entry.
my_secondary_database.pget(NULL, &key, &pkey, &pdata, 0);
```

[illegible]

Deleting Secondary Database Records

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0); // Primary
Db my_index(NULL, 0);    // Secondary

// Open the primary
my_database.open(NULL,       // Transaction pointer
                 "my_db.db", // On-disk file that holds the database.
                NULL,        // Optional logical database name
                DB_BTREE,    // Database access method
                DB_CREATE,   // Open flags
                0);          // File mode (using defaults)

// Setup the secondary to use sorted duplicates.
// This is often desireable for secondary databases.
my_index.set_flags(DB_DUPSORT);

// Open the secondary
my_index.open(NULL,              // Transaction pointer
              "my_secondary.db", // On-disk file that holds the database.
              NULL,              // Optional logical database name
              DB_BTREE,          // Database access method
              DB_CREATE,         // Open flags.
              0);                // File mode (using defaults)


// Now associate the primary and the secondary
my_database.associate(NULL,          // Txn id
                      &my_index,     // Associated secondary database
                      get_sales_rep, // Callback used for key extraction.
                      0);            // Flags

// Name to delete
char *search_name = "John Doe";

// Get a search key
Dbt key(search_name, strlen(search_name) + 1);

// Now delete the secondary record. This causes the associated primary
// record to be deleted. If any other secondary databases have secondary
// records referring to the deleted primary record, then those secondary
// records are also deleted.
my_index.del(NULL, &key, 0);
```

Using Cursors with Secondary Databases

[illegible]

[illegible]

- [illegible]

- [illegible] `DB_GET_BOTH` [illegible]

[illegible]

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Db my_index(NULL, 0);


// Get a cursor on the secondary database
Dbc *cursorp;
my_index.cursor(NULL, &cursorp, 0);


// Name to delete
char *search_name = "John Doe";

// Instantiate Dbts as normal
Dbt key(search_name, strlen(search_name) + 1);
Dbt data;


// Position the cursor
while (cursorp->get(&key, &data, DB_SET) == 0)
    cursorp->del(0);
```

Database Joins

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Using Join Cursors

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] Db::join() [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

// Exception handling omitted

int ret;

Db automotiveDB(NULL, 0);
Db automotiveColorDB(NULL, 0);
Db automotiveMakeDB(NULL, 0);
Db automotiveTypeDB(NULL, 0);

// Database and secondary database opens omitted for brevity.
// Assume a primary database:
//   automotiveDB
// Assume 3 secondary databases:
//   automotiveColorDB  -- secondary database based on automobile color
//   automotiveMakeDB  -- secondary database based on the manufacturer
//   automotiveTypeDB  -- secondary database based on automobile type

// Position the cursors
Dbc *color_curs;
automotiveColorDB.cursor(NULL, &color_curs, 0);
char *the_color = "red";
Dbt key(the_color, strlen(the_color) + 1);
Dbt data;
if ((ret = color_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

Dbc *make_curs;
automotiveMakeDB.cursor(NULL, &make_curs, 0);
char *the_make = "Toyota";
key.set_data(the_make);
key.set_size(strlen(the_make) + 1);
if ((ret = make_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

Dbc *type_curs;
automotiveTypeDB.cursor(NULL, &type_curs, 0);
char *the_type = "minivan";
key.set_data(the_type);
key.set_size(strlen(the_type) + 1);
if ((ret = type_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

// Set up the cursor array
Dbc *carray[4];
carray[0] = color_curs;
carray[1] = make_curs;
carray[2] = type_curs;
carray[3] = NULL;

// Create the join
Dbc *join_curs;
if ((ret = automotiveDB.join(carray, &join_curs, 0)) != 0) {
    // Error handling goes here
}

// Iterate using the join cursor
while ((ret = join_curs->get(&key, &data, 0)) == 0) {
    // Do interesting things with the key and data
}

// If we exited the loop because we ran out of records,
// then it has completed successfully.
if (ret == DB_NOTFOUND) {
     // Close all our cursors and databases as is appropriate,  and
     // then exit with a normal exit status (0).
}
```

Secondary Database Example

[illegible]

[illegible]

[illegible]

Secondary Databases with example_database_load

[illegible] example_database_load [illegible]

[illegible]

[illegible]

[illegible]

[illegible] example_database_load

[illegible]

```
DB_INSTALL/examples_cxx/getting_started
```

[illegible] *DB_INSTALL* [illegible]

[illegible] gettingStartedCommon.hpp [illegible]

```
// File: gettingStartedCommon.hpp
// Forward declarations
class Db;
class Dbt;

// Used to extract an inventory item's name from an
// inventory database record. This function is used to create
// keys for secondary database records.
int
get_item_name(Db *dbp, const Dbt *pkey, const Dbt *pdata, Dbt *skey)
{
    // Obtain the buffer location where the we placed the item's name. In
    // this example, the item's name is located in the primary data. It is
    // the first string in the buffer after the price (a double) and
    // the quantity (a long).
    size_t offset = sizeof(double) + sizeof(long);
    char * itemname = (char *)pdata->get_data() + offset;

    // unused
```

```
(void)pkey;

// If the offset is beyond the end of the data, then there is a
// problem with the buffer contained in pdata, or there's a
// programming error in how the buffer is marshalled/unmarshalled.
// This should never happen!
if ((u_int32_t)id.getBufferSize() != pdata->get_size()) {
    // Make sure the default constructor is private
    // We don't want it used.
    MyDb() : db_(0, 0) {}

    // We put our database close activity here.
    // This is called from our destructor. In
    // a more complicated example, we might want
    // to make this method public, but a private
    // method is more appropriate for this example.
    void close();
};
```

[illegible]

```
// File: MyDb.cpp
#include "MyDb.hpp"

// Class constructor. Requires a path to the location
// where the database is located, and a database name
MyDb::MyDb(std::string &path, std::string &dbName,
           bool isSecondary)
    : db_(NULL, 0),               // Instantiate Db object
      dbFileName_(path + dbName), // Database file name
      cFlags_(DB_CREATE)          // If the database doesn't yet exist,
                                  // allow it to be created.
{
    try
    {
        // Redirect debugging information to std::cerr
        db_.set_error_stream(&std::cerr);

        // If this is a secondary database, support
        // sorted duplicates
        if (isSecondary)
            db_.set_flags(DB_DUPSORT);

        // Open the database
        db_.open(NULL, dbFileName_.c_str(), NULL, DB_BTREE, cFlags_, 0);
    }
    // DbException is not a subclass of std::exception, so we
    // need to catch them both.
    catch(DbException &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

nn, p T w pode example_database_load u wo ote o y e seco dua d atse a h n n r b adsoa.v eiat yoit te t ei o t d atse a

n , h h n T mp mln wsn e a hepawe rodo s on ee etl i; e dafo st o g æ e s e t d h n n h wh , w j main() r s ent w f ucwo th c iu. ie e oa o fd T oatocc o see e t u mp mln n h r mm n es, of t e t b ie e l patfo stco _(ade)_e. æ d atse ao d 2 ag 8

```
// Loads the contents of vendors.txt and inventory.txt into
// Berkeley DB databases.
int
main(int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string basename("./");
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");
    std::string itemSDbName("itemname.sdb");

    // Parse the command line arguments here and determine
    // the location of the flat text files containing the
    // inventory data here. This step is omitted for clarity.

    //  Identify the full name for our input files, which should
    //  also include some path information.
    std::string inventoryFile = basename + "inventory.txt";
    std::string vendorFile = basename + "vendors.txt";

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);
        MyDb itemnameSDB(databaseHome, itemSDbName, true);

        // Associate the primary and the secondary
        inventoryDB.getDb().associate(NULL,
                                      &(itemnameSDB.getDb()),
                                      get_item_name,
                                      0);

        // Load the vendor database
        loadVendorDB(vendorDB, vendorFile);

        // Load the inventory database
        loadInventoryDB(inventoryDB, inventoryFile);
    } catch(DbException &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

[illegible]

[illegible]

Secondary Databases with example_database_read

[illegible]

[illegible]

[illegible]

```
// File: example_database_read.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```
#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
int show_all_records(MyDb &inventoryDB, MyDb &vendorDB);
int show_item(MyDb &itemnameSDB, MyDb &vendorDB, std::string &itemName);
int show_vendor(MyDb &vendorDB, const char *vendor);
```

[illegible]

[illegible]

```
// Displays all inventory items and the associated vendor record.
int
main (int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string databaseHome("./");
    std::string itemName;

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");
    std::string itemSDbName("itemname.sdb");

    // Parse the command line arguments
    // Omitted for brevity

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);
        MyDb itemnameSDB(databaseHome, itemSDbName, true);

        // Associate the secondary to the primary
        inventoryDB.getDb().associate(NULL,
                                      &(itemnameSDB.getDb()),
                                      get_item_name,
                                      0);

        if (itemName.empty())
        {
            show_all_records(inventoryDB, vendorDB);
        } else {
            show_item(itemnameSDB, vendorDB, itemName);
        }
    } catch(DbException &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

[illegible] `show_item()` [illegible]

[illegible]

`DB_INSTALL/examples_cxx/getting_started`

[illegible] *DB_INSTALL* [illegible]

```
// Shows the records in the inventory database that
// have a specific item name. For each inventory record
// shown, the appropriate vendor record is also displayed.
int
show_item(MyDb &itemnameSDB, MyDb &vendorDB, std::string &itemName)
{
    // Get a cursor to the itemname secondary db
    Dbc *cursorp;

    try {
        itemnameSDB.getDb().cursor(NULL, &cursorp, 0);

        // Get the search key. This is the name on the inventory
        // record that we want to examine.
        std::cout << "Looking for " << itemName << std::endl;
        Dbt key((void *)itemName.c_str(), itemName.length() + 1);
        Dbt data;

        // Position the cursor to the first record in the secondary
        // database that has the appropriate key.
        int ret = cursorp->get(&key, &data, DB_SET);
        if (!ret) {
            do {
                InventoryData inventoryItem(data.get_data());
                inventoryItem.show();
                show_vendor(vendorDB, inventoryItem.getVendor().c_str());

            } while(cursorp->get(&key, &data, DB_NEXT_DUP) == 0);
        } else {
            std::cerr << "No records found for '" << itemName
                      << "'" << std::endl;
        }
    } catch(DbException &e) {
        itemnameSDB.getDb().err(e.get_errno(), "Error in show_item");
        cursorp->close();
        throw e;
    } catch(std::exception &e) {
        itemnameSDB.getDb().errx("Error in show_item: %s", e.what());
        cursorp->close();
        throw e;
    }

    cursorp->close();
    return (0);
}
```

o l pr T scoinU heexample_invedetnaorty_reuadu s y i sgt i dwe ato c uan u nr h nll n rm h m h sep cwafor vnl mdoy Faremipo tl e.is t at c zat a catxi a ea o eu æ

```
    example_inventory_read -i "Zulu Nut"
```

Chapter 6. Database Configuration

[illegible]

[illegible] *Berkeley DB Programmer's Reference Guide*.

Setting the Page Size

[illegible]

[illegible]

[illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]

[illegible]

Overflow Pages

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] `DbEnv::stat()` [illegible] `db_stat` [illegible]

Locking

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] `DbEnv::lock_stat()` [illegible] `Environment.getLockStats()` [illegible] `db_stat` [illegible] `st_lock_wait` [illegible]

IO Efficiency

[illegible]

Page Sizing Advice

[illegible]

[illegible]

Selecting the Cache Size

[illegible]

[illegible] `DbEnv::set_cachesize()` [illegible]

[illegible]

[illegible] `db_stat` [illegible]

BTree Configuration

[illegible]

- [illegible]
- [illegible]

Allowing Duplicate Records

[illegible]

[illegible] `Db::set_bt_compare()` [illegible]

[illegible]

[illegible]

[illegible]

Sorted Duplicates

[illegible] `DB_DUPSORT` [illegible]

[illegible] `Db::set_dup_compare()` [illegible]

Unsorted Duplicates

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

- DB_AFTER

[illegible]

[illegible]

[illegible]

- DB_BEFORE

[illegible]

- DB_KEYFIRST

[illegible]

- DB_KEYLAST

[illegible]

Configuring a Database to Support Duplicates

[illegible]

[illegible]

- DB_DUP

[illegible]

- DB_DUPSORT

[illegible]

[illegible]

```
#include <db_cxx.h>
...

Db db(NULL, 0);
const char *file_name = "myd.db";

try {
    // Configure the database for sorted duplicates
    db.set_flags(DB_DUPSORT);

    // Now open the database
    db.open(NULL,       // Txn pointer
            file_name,  // File name
            NULL,       // Logical db name (unneeded)
            DB_BTREE,   // Database type (using btree)
            DB_CREATE,  // Open flags
            0);         // File mode. Using defaults
} catch(DbException &e) {
    db.err(e.get_errno(), "Database '%s' open failed.", file_name);
} catch(std::exception &e) {
    db.errx("Error opening database: %s : %s\n", file_name, e.what());
}

...

try {
    db.close(0);
} catch(DbException &e) {
    db.err(e.get_errno(), "Database '%s' close failed.", file_name);
} catch(std::exception &e) {
    db.errx("Error closing database: %s : %s\n", file_name, e.what());
}
```

Setting Comparison Functions

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Creating Comparison Functions

[illegible] `Db::set_bt_compare()` [illegible] `Db::set_dup_compare()` [illegible]

[illegible]

[illegible] `set_bt_compare()` [illegible]

```
int (*function)(Db *db, const Dbt *key1, const Dbt *key2)
```

[illegible]

[illegible] `set_dup_compare()` [illegible]

[illegible]

```
int
compare_int(Db *dbp, const Dbt *a, const Dbt *b)
{
    int ai, bi;

    // Returns:
    // < 0 if a < b
    // = 0 if a = b
    // > 0 if a > b
    memcpy(&ai, a->get_data(), sizeof(int));
    memcpy(&bi, b->get_data(), sizeof(int));
    return (ai - bi);
}
```

[illegible]

[illegible]

```
#include <db_cxx.h>
#include <string.h>

...

Db db(NULL, 0);

// Set up the btree comparison function for this database
db.set_bt_compare(compare_int);

// Database open call follows sometime after this.
```