## NAME

**Cdt –** container data types

## SYNOPSIS

```
#include <cdt.h>
```

### DICTIONARY TYPES

```
Dt_t;
Dtdisc_t;
Dtmethod_t;
Dtlink_t;
Dtstat_t;
```

### DICTIONARY CONTROL

```
Dt_t*       dtopen(const Dtdisc_t* disc, const Dtmethod_t* meth);
int         dtclose(Dt_t* dt);
void        dtclear(dt);
Dtmethod_t* dtmethod(Dt_t* dt, const Dtmethod_t* meth);
Dtdisc_t*   dtdisc(Dt_t* dt, const Dtdisc_t* disc, int type);
Dt_t*       dtview(Dt_t* dt, Dt_t* view);
int         dttreeset(Dt_t* dt, int minp, int balance);
```

### STORAGE METHODS

```
Dtmethod_t* Dtset;
Dtmethod_t* Dtbag;
Dtmethod_t* Dtoset;
Dtmethod_t* Dtobag;
Dtmethod_t* Dtlist;
Dtmethod_t* Dtstack;
Dtmethod_t* Dtqueue;
Dtmethod_t* Dtdeque;
```

### DISCIPLINE

```
#define DTOFFSET(struct_s,member)
#define DTDISC(disc,key,size,link,makef,freef,comparf,hashf,memoryf,eventf)
typedef void*      (*Dtmake_f)(Dt_t*, void*, Dtdisc_t*);
typedef void         (*Dtfree_f)(Dt_t*, void*, Dtdisc_t*);
typedef int          (*Dtcompar_f)(Dt_t*, void*, void*, Dtdisc_t*);
typedef unsigned int (*Dthash_f)(Dt_t*, void*, Dtdisc_t*);
typedef void*      (*Dtmemory_f)(Dt_t*, void*, size_t, Dtdisc_t*);
typedef int          (*Dtevent_f)(Dt_t*, int, void*, Dtdisc_t*);
```

### OBJECT OPERATIONS

```
void*   dtinsert(Dt_t* dt, void* obj);
void*   dtappend(Dt_t* dt, void* obj);
void*   dtdelete(Dt_t* dt, void* obj);
void*   dtattach(Dt_t* dt, void* obj);
void*   dtdetach(Dt_t* dt, void* obj);
void*   dtsearch(Dt_t* dt, void* obj);
void*   dtmatch(Dt_t* dt, void* key);
void*   dtfirst(Dt_t* dt);
void*   dtnext(Dt_t* dt, void* obj);
void*   dtlast(Dt_t* dt);
void*   dtprev(Dt_t* dt, void* obj);
void*   dtfinger(Dt_t* dt);
void*   dtrenew(Dt_t* dt, void* obj);
int       dtwalk(Dt_t* dt, int (*userf)(Dt_t*, void*, void*), void*);
Dtlink_t* dtflatten(Dt_t* dt);
```

```
Dtlink_t* dtlink(Dt_t*, Dtlink_t* link);
void*   dtobj(Dt_t* dt, Dtlink_t* link);
Dtlink_t* dtextract(Dt_t* dt);
int       dtrestore(Dt_t* dt, Dtlink_t* link);

#define   DTTREESEARCH(Dt_t* dt, void* obj, action)
#define   DTTREEMATCH(Dt_t* dt, void* key, action)
```

### DICTIONARY STATUS

```
Dt_t*     dtvnext(Dt_t* dt);
int       dtvcount(Dt_t* dt);
Dt_t*     dtvhere(Dt_t* dt);
int       dtsize(Dt_t* dt);
int       dtstat(Dt_t* dt, Dtstat_t*, int all);
```

### HASH FUNCTIONS

```
unsigned int dtstrhash(unsigned int h, char* str, int n);
unsigned int dtcharhash(unsigned int h, unsigned char c);
```

## DESCRIPTION

*Cdt* manages run-time dictionaries using standard container data types: unordered set/multiset, ordered set/multiset, list, stack, and queue.

### DICTIONARY TYPES

**Dt_t**

This is the type of a dictionary handle.

**Dtdisc_t**

This defines the type of a discipline structure which describes object lay-out and manipulation functions.

**Dtmethod_t**

This defines the type of a container method.

**Dtlink_t**

This is the type of a dictionary object holder (see `dtdisc()`.)

**Dtstat_t**

This is the type of a structure to return dictionary statistics (see `dtstat()`.)

### DICTIONARY CONTROL

**Dt_t* dtopen(const Dtdisc_t* disc, const Dtmethod_t* meth)**

This creates a new dictionary. `disc` is a discipline structure to describe object format. `meth` specifies a manipulation method. `dtopen()` returns the new dictionary or `NULL` on error. See also the events `DT_OPEN` and `DT_ENDOPEN` below.

**int dtclose(Dt_t* dt)**

This deletes `dt` and its objects. Note that `dtclose()` fails if `dt` is being viewed by some other dictionaries (see `dtview()`). `dtclose()` returns `0` on success and `-1` on error. See also the events `DT_CLOSE` and `DT_ENDCLOSE` below.

**void dtclear(Dt_t* dt)**

This deletes all objects in `dt` without closing `dt`.

**Dtmethod_t dtmethod(Dt_t* dt, const Dtmethod_t* meth)**

If `meth` is `NULL`, `dtmethod()` returns the current method. Otherwise, it changes the storage method of `dt` to `meth`. Object order remains the same during a method switch among `Dtlist`, `Dtstack`, `Dtqueue` and `Dtdeque`. Switching to and from `Dtset/Dtbag` and `Doset/Dtobag` may cause objects to be rehashed, reordered, or removed as the case requires. `dtmethod()` returns the previous method or `NULL` on error.

**Dtdisc_t* dtdisc(Dt_t* dt, const Dtdisc_t* disc, int type)**

If `disc` is `NULL`, `dtdisc()` returns the current discipline. Otherwise, it changes the discipline of `dt` to `disc`. Objects may be rehashed, reordered, or removed as appropriate. `type` can be any bit combination of `DT_SAMECMP` and `DT_SAMEHASH`. `DT_SAMECMP` means that objects will compare exactly the same as before thus obviating the need for reordering or removing new duplicates. `DT_SAMEHASH` means that hash values of objects remain the same thus obviating the need to rehash. `dtdisc()` returns the previous discipline on success and `NULL` on error.

**Dt_t* dtview(Dt_t* dt, Dt_t* view)**

A viewpath allows a search or walk starting from a dictionary to continue to another. `dtview()` first terminates any current view from `dt` to another dictionary. Then, if `view` is `NULL`, `dtview` returns the terminated view dictionary. If `view` is not `NULL`, a viewpath from `dt` to `view` is established. `dtview()` returns `dt` on success and `NULL` on error.

It is an error to have dictionaries on a viewpath with different storage methods. In addition, dictionaries on the same view path should treat objects in a consistent manner with respect to comparison or hashing. If not, undefined behaviors may result.

**int dttreeset(Dt_t* dt, int minp, int balance)**

This function only applies to dictionaries operated under the method `Dtoset` which uses top-down splay trees (see below). It returns 0 on success and -1 on error.

`minp`: This parameter defines the minimum path length before a search path is adjusted. For example, `minp` equal 0 would mean that search paths are always adjusted. If `minp` is negative, the minimum search path is internally computed based on a function of the current dictionary size. This computed value is such that if the tree is balanced, it will never require adjusting.

`balance`:
If this is non-zero, the tree will be made balanced.

## STORAGE METHODS

Storage methods are of type `Dtmethod_t*`. *Cdt* supports the following methods:

**Dtoset**
**Dtobag**

Objects are ordered by comparisons. `Dtoset` keeps unique objects. `Dtobag` allows repeatable objects.

**Dtset**
**Dtbag**

Objects are unordered. `Dtset` keeps unique objects. `Dtbag` allows repeatable objects and always keeps them together (note the effect on dictionary walking.) These methods use a hash table with chaining to manage the objects. See also the event `DT_HASHSIZE` below on how to manage hash table resizing when objects are inserted.

**Dtlist**

Objects are kept in a list. The call `dtinsert()` inserts a new object in front of *the current object* (see `dtfinger()`) if it is defined or at list front if no current object is defined. Similarly, the call `dtappend()` appends a new object after *the current object* (see `dtfinger()`) if it is defined or at list end if no current object is defined.

**Dtdeque**

Objects are kept in a deque. This is similar to `Dtlist` except that objects are always inserted at the front and appended at the tail of the list.

**Dtstack**

Objects are kept in a stack, i.e., in reverse order of insertion. Thus, the last object inserted is at stack top and will be the first to be deleted.

**Dtqueue**

Objects are kept in a queue, i.e., in order of insertion. Thus, the first object inserted is at queue head and will be the first to be deleted.

## DISCIPLINE

Object format and associated management functions are defined in the type `Dtdisc_t`:

```
typedef struct
{ int         key, size;
  int         link;
  Dtmake_f    makef;
  Dtfree_f    freef;
  Dtcompar_f  comparf;
  Dthash_f    hashf;
  Dtmemory_f  memoryf;
  Dtevent_f   eventf;
} Dtdisc_t;
```

**int key, size**

Each object `obj` is identified by a key used for object comparison or hashing. `key` should be non-negative and defines an offset into `obj`. If `size` is negative, the key is a null-terminated string with starting address `*(void**)((char*)obj+key)`. If `size` is zero, the key is a null-terminated string with starting address `(void*)((char*)obj+key)`. Finally, if `size` is positive, the key is a byte array of length `size` starting at `(void*)((char*)obj+key)`.

**int link**

Let `obj` be an object to be inserted into `dt` as discussed below. If `link` is negative, an internally allocated object holder is used to hold `obj`. Otherwise, `obj` should have a `Dtlink_t` structure embedded `link` bytes into it, i.e., at address `(Dtlink_t*)((char*)obj+link)`.

**void* (*makef)(Dt_t* dt, void* obj, Dtdisc_t* disc)**

If `makef` is not `NULL`, `dtinsert(dt,obj)` or `dtappend()` will call it to make a copy of `obj` suitable for insertion into `dt`. If `makef` is `NULL`, `obj` itself will be inserted into `dt`.

**void (*freef)(Dt_t* dt, void* obj, Dtdisc_t* disc)**

If not `NULL`, `freef` is used to destroy data associated with `obj`.

**int (*comparf)(Dt_t* dt, void* key1, void* key2, Dtdisc_t* disc)**

If not `NULL`, `comparf` is used to compare two keys. Its return value should be `<0`, `=0`, or `>0` to indicate whether `key1` is smaller, equal to, or larger than `key2`. All three values are significant for method `Dtoset` and `Dtobag`. For other methods, a zero value indicates equality and a non-zero value indicates inequality. If `(*comparf)()` is `NULL`, an internal function is used to compare the keys as defined by the `Dtdisc_t.size` field.

**unsigned int (*hashf)(Dt_t* dt, void* key, Dtdisc_t* disc)**

If not `NULL`, `hashf` is used to compute the hash value of `key`. It is required that keys compared equal will also have same hash values. If `hashf` is `NULL`, an internal function is used to hash the key as defined by the `Dtdisc_t.size` field.

**void* (*memoryf)(Dt_t* dt, void* addr, size_t size, Dtdisc_t* disc)**

If not `NULL`, `memoryf` is used to allocate and free memory. When `addr` is `NULL`, a memory segment of size `size` is requested. If `addr` is not `NULL` and `size` is zero, `addr` is to be freed. If `addr` is not `NULL` and `size` is positive, `addr` is to be resized to the given size. If `memoryf` is `NULL`, *malloc(3)* is used.

**int (*eventf)(Dt_t* dt, int type, void* data, Dtdisc_t* disc)**

If not `NULL`, `eventf` announces various events. Each event may have particular handling of the return values from `eventf`. But a negative return value typically means failure. Following are the events:

`DT_OPEN`:

`dt` is being opened. If `eventf` returns negative, the opening process terminates with failure. If `eventf` returns zero, the opening process proceeds in a default manner. A positive return value indicates special treatment of memory as follows. If `*(void**)data` is set to point to some memory segment as discussed in `memoryf`, that segment of memory is used to start the dictionary. If `*(void**)data` is `NULL`, all memory including that of the dictionary handle itself will be allocated via `memoryf`.

`DT_ENDOPEN`:
This event announces that `dtopen()` has successfully opened a dictionary and is about to return. The `data` argument of `eventf` should be the new dictionary handle itself.

`DT_CLOSE`:
`dt` is about to be closed. If `eventf` returns negative, the closing process stops immediately and `dtclose()` returns -1. Objects in the dictionary are deleted only if `eventf` returns zero. The dictionary handle itself is processed as follows. If it was allocated via `malloc()`, it will be freed. If it was allocated via `memoryf` (see `dtopen()`) and `eventf` returns 0, a call to `memoryf` will be issued to attempt freeing the handle. Otherwise, nothing will be done to its memory.

As should be clear from their description, the events `DT_OPEN` and `DT_CLOSE` are designed to be used along with `memoryf` to manage the allocation and deallocation of dictionary and object memory across dictionaries. In fact, they can be used to manage dictionaries based on shared and/or persistent memory.

`DT_ENDCLOSE`:
This event announces that `dtclose()` has successfully closed a dictionary and is about to return.

`DT_DISC`:
The discipline of `dt` is being changed to a new one given in `(Dtdisc_t*)data`.

`DT_METH`:
The method of `dt` is being changed to a new one given in `(Dtmethod_t*)data`.

`DT_HASHSIZE`:
The hash table (for `Dtset` and `Dtbag`) is being resized. In this case, `*(int*)data` has the current size of the table. The application can set the new table size by first changing `*(int*)data` to the desired size, then return a positive value. The application can also fix the table size at the current value forever by setting `*(int*)data` to a negative value, then again return a positive value. A non-positive return value from the event handling function means that Cdt will be responsible for choosing the hash table size.

**#define DTOFFSET(struct_s,member)**
This macro function computes the offset of `member` from the start of structure `struct_s`. It is useful for getting the offset of a `Dtlink_t` embedded inside an object.

**#define DTDISC(disc,key,size,link,makef,freef,comparf,hashf,memoryf,eventf)**
This macro function initializes the discipline pointed to by `disc` with the given values.

## OBJECT OPERATIONS

**void* dtinsert(Dt_t* dt, void* obj)**
**void* dtappend(Dt_t* dt, void* obj)**
These functions add an object prototyped by `obj` into `dt`. `dtinsert()` and `dtappend()` perform the same function for all methods except for `Dtlist`. See `Dtlist` for details. If there is an existing object in `dt` matching `obj` and the storage method is `Dtset` or `Dtoset`, `dtinsert()` and `dtappend()` will simply return the matching object. Otherwise, a new object is inserted according to the method in use. See `Dtdisc_t.makef` for object construction. The new object or a matching object as noted will be returned on success while `NULL` is returned on error.

**void* dtdelete(Dt_t* dt, void* obj)**
If `obj` is `NULL`, methods `Dtstack` and `Dtqueue` delete respectively stack top or queue head while other methods do nothing. If `obj` is not `NULL`, there are two cases. If the method in use is not `Dtbag` or `Dtobag`, the first object matching `obj` is deleted. On the other hand, if the method in use is `Dtbag` or `Dtobag`, the library check to see if `obj` is in the dictionary and delete it. If `obj` is not in the dictionary, some object matching it will be deleted. See `Dtdisc_t.freef` for object destruction. `dtdelete()` returns the deleted object (even if it was deallocated) or `NULL` on error.

**void* dtattach(Dt_t* dt, void* obj)**

This function is similar to `dtinsert()` but `obj` itself will be inserted into `dt` even if a discipline function `makef` is defined.

**void* dtdetach(Dt_t* dt, void* obj)**

This function is similar to `dtdelete()` but the object to be deleted from `dt` will not be freed (via the discipline `freef` function).

**void* dtsearch(Dt_t* dt, void* obj)**
**void* dtmatch(Dt_t* dt, void* key)**

These functions find an object matching `obj` or `key` either from `dt` or from some dictionary accessible from `dt` via a viewpath (see `dtview()`.) `dtsearch()` and `dtmatch()` return the matching object or `NULL` on failure.

**void* dtfirst(Dt_t* dt)**
**void* dtnext(Dt_t* dt, void* obj)**

`dtfirst()` returns the first object in `dt`. `dtnext()` returns the object following `obj`. Objects are ordered based on the storage method in use. For `Dtoset` and `Dtobag`, objects are ordered by object comparisons. For `Dtstack`, objects are ordered in reverse order of insertion. For `Dtqueue`, objects are ordered in order of insertion. For `Dtlist`, objects are ordered by list position. For `Dtset` and `Dtbag`, objects are ordered by some internal order (more below). Thus, objects in a dictionary or a viewpath can be walked using a `for(;;)` loop as below.

```
for(obj = dtfirst(dt); obj; obj = dtnext(dt,obj))
```

When a dictionary uses `Dtset` or `Dtbag`, the object order is determined upon a call to `dtfirst()`/`dtlast()`. This order is frozen until a call `dtnext()`/`dtprev()` returns `NULL` or when these same functions are called with a `NULL` object argument. It is important that a `dtfirst()/dtlast()` call be balanced by a `dtnext()/dtprev()` call as described. Nested loops will require multiple balancing, once per loop. If loop balancing is not done carefully, either performance is degraded or unexpected behaviors may result.

**void* dtlast(Dt_t* dt)**
**void* dtprev(Dt_t* dt, void* obj)**

`dtlast()` and `dtprev()` are like `dtfirst()` and `dtnext()` but work in reverse order. Note that dictionaries on a viewpath are still walked in order but objects in each dictionary are walked in reverse order.

**void* dtfinger(Dt_t* dt)**

This function returns the *current object* of `dt`, if any. The current object is defined after a successful call to one of `dtsearch()`, `dtmatch()`, `dtinsert()`, `dtfirst()`, `dtnext()`, `dtlast()`, or `dtprev()`. As a side effect of this implementation of *Cdt*, when a dictionary is based on `Dtoset` and `Dtobag`, the current object is always defined and is the root of the tree.

**void* dtrenew(Dt_t* dt, void* obj)**

This function repositions and perhaps rehashes an object `obj` after its key has been changed. `dtrenew()` only works if `obj` is the current object (see `dtfinger()`).

**dtwalk(Dt_t* dt, int (*userf)(Dt_t*, void*, void*), void* data)**

This function calls `(*userf)(walk,obj,data)` on each object in `dt` and other dictionaries viewable from it. `walk` is the dictionary containing `obj`. If `userf()` returns a `<0` value, `dtwalk()` terminates and returns the same value. `dtwalk()` returns `0` on completion.

**Dtlink_t* dtflatten(Dt_t* dt)**
**Dtlink_t* dtlink(Dt_t* dt, Dtlink_t* link)**
**void* dtobj(Dt_t* dt, Dtlink_t* link)**

Using `dtfirst()/dtnext()` or `dtlast()/dtprev()` to walk a single dictionary can incur significant cost due to function calls. For efficient walking of a single directory (i.e., no viewpathing), `dtflatten()` and `dtlink()` can be used. Objects in `dt` are made into a linked list and walked as follows:

```
for(link = dtflatten(dt); link; link = dtlink(dt,link) )
```

Note that `dtflatten()` returns a list of type `Dtlink_t*`, not `void*`. That is, it returns a dictionary holder pointer, not a user object pointer (although both are the same if the discipline field `link` is zero.) The macro function `dtlink()` returns the dictionary holder object following `link`. The macro function `dtobj(dt,link)` returns the user object associated with `link`, Beware that the flattened object list is unflattened on any dictionary operations other than `dtlink()`.

**Dtlink_t* dtextract(Dt_t* dt)**
**int dtrestore(Dt_t* dt, Dtlink_t* link)**

`dtextract()` extracts all objects from `dt` and makes it appear empty. `dtrestore()` repopulates `dt` with objects previously obtained via `dtextract()`. `dtrestore()` will fail if `dt` is not empty. These functions can be used to share a same `dt` handle among many sets of objects. They are useful to reduce dictionary overhead in an application that creates many concurrent dictionaries. It is important that the same discipline and method are in use at both extraction and restoration. Otherwise, undefined behaviors may result.

**#define DTTREESEARCH(Dt_t* dt, void* obj, action)**
**#define DTTREEMATCH(Dt_t* dt, void* key, action)**

These macro functions are analogues of `dtsearch()` and `dtmatch()` but they can only be used on a dictionary based on a binary search tree, i.e., `Dtoset` or `Dtobag`.

`obj` or `key`:
These are used to find a matching object. If there is no match, the result is `NULL`.

`action`:
The matching object `o` (which may be `NULL`) will be processed as follow:

```
action (o);
```

Since `action` is used verbatim, it can be any C code fragment combinable with `(o)` to form a syntactically correct C statement. For example, suppose that the matching object is an integer, the below code accumulates the integer value in a variable `total`:

```
DTTREEMATCH(dt, key, total += (int));
```

## DICTIONARY INFORMATION

**Dt_t* dtvnext(Dt_t* dt)**

This returns the dictionary that `dt` is viewing, if any.

**int dtvcount(Dt_t* dt)**

This returns the number of dictionaries that view `dt`.

**Dt_t* dtvhere(Dt_t* dt)**

This returns the dictionary `v` viewable from `dt` where an object was found from the most recent search or walk operation.

**int dtsize(Dt_t* dt)**

This function returns the number of objects stored in `dt`.

**int dtstat(Dt_t *dt, Dtstat_t* st, int all)**

This function reports dictionary statistics. If `all` is non-zero, all fields of `st` are filled. Otherwise, only the `dt_type` and `dt_size` fields are filled. It returns `0` on success and `-1` on error.

`Dtstat_t` contains the below fields:

`int dt_type`:
This is one of `DT_SET`, `DT_BAG`, `DT_OSET`, `DT_OBAG`, `DT_LIST`, `DT_STACK`, and `DT_QUEUE`.

`int dt_size`:
This contains the number of objects in the dictionary.

`int dt_n`:

For `Dtset` and `Dtbag`, this is the number of non-empty chains in the hash table. For `Dtoset` and `Dtobag`, this is the deepest level in the tree (counting from zero.) Each level in the tree contains all nodes of equal distance from the root node. `dt_n` and the below two fields are undefined for other methods.

`int dt_max`:

For `Dtbag` and `Dtset`, this is the size of a largest chain. For `Dtoset` and `Dtobag`, this is the size of a largest level.

`int* dt_count`:

For `Dtset` and `Dtbag`, this is the list of counts for chains of particular sizes. For example, `dt_count[1]` is the number of chains of size `1`. For `Dtoset` and `Dtobag`, this is the list of sizes of the levels. For example, `dt_count[1]` is the size of level `1`.

### HASH FUNCTIONS

**unsigned int dtcharhash(unsigned int h, char c)**

**unsigned int dtstrhash(unsigned int h, char* str, int n)**

These functions compute hash values from bytes or strings. `dtcharhash()` computes a new hash value from byte `c` and seed value `h`. `dtstrhash()` computes a new hash value from string `str` and seed value `h`. If `n` is positive, `str` is a byte array of length `n`; otherwise, `str` is a null-terminated string.

## IMPLEMENTATION NOTES

`Dtset` and `Dtbag` are based on hash tables with move-to-front collision chains. `Dtoset` and `Dtobag` are based on top-down splay trees. `Dtlist`, `Dtstack` and `Dtqueue` are based on doubly linked list.

## AUTHOR

Kiem-Phong Vo, kpv@research.att.com