uthash User Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.5	April 2009		TDH

Contents

1 A hash in C

This document is written for C programmers. Since you're reading this, chances are that you know a hash is used for looking up items using a key. In scripting languages like Perl, hashes are used all the time. In C, hashes don't exist in the language itself. This software provides a hash table for C structures.

1.1 What can it do?

This software supports these basic operations on items in a hash table:

1. add
2. find
3. delete
4. count
5. iterate
6. sort

1.2 Is it fast?

Add, find and delete are normally constant-time operations. This is influenced by your key domain and the hash function.

This hash aims to be minimalistic and efficient. It's around 600 lines of C. It inlines automatically because it's implemented as macros. It's fast as long as the hash function is suited to your keys. You can use the default hash function, or easily compare performance and choose from among several other built-in hash functions.

1.3 Is it a library?

No, it's just a single header file: `uthash.h`. All you need to do is copy the header file into your project, and:

```
#include "uthash.h"
```

Since uthash is a header file only, there is no library code to link against.

1.4 C/C++ and platforms

This software can be used in C and C++ programs. It has been tested on:

- Linux
- Mac OS X
- Solaris
- OpenBSD
- Cygwin
- MinGW

1.4.1 Non-GNU C++ compilers

When compiling C++ code it may be necessary to use the GNU compiler, g++, which supports the `typeof` extension. Alternatively some users of non-GNU C++ compilers simply hardcode the `TYPEOF` macro in `uthash.h` to the data type they want to use.

1.4.2 Test suite

You can run the test suite on these platforms, or any prospective Unix-like platform, in this way:

```
cd tests/
make
```

1.5 BSD licensed

This software is made available under the revised BSD license. It is free and open source.

1.6 Obtaining uthash

Please follow the link to download on the uthash website at http://uthash.sourceforge.net.

1.7 Getting help

Feel free to email the author at thanson@users.sourceforge.net.

1.8 Resources

News
: The author has a news feed for software updates (RSS).

Linked list macros
: uthash includes a separate, standalone header called utlist.h which has *linked list macros* for C structures, similar in style to the hash macros

1.9 Who's using it?

Since releasing uthash in 2006, it has been downloaded thousands of times, incorporated into commerical software, academic research, and into other open-source software.

2 Your structure

In uthash, a hash table is comprised of structures. Each structure represents a key-value association. One or more of the structure fields constitute the key; the structure itself is the value.

Example 2.1 Defining a structure that can be hashed

```
#include "uthash.h"

struct my_struct {
    int id;                    /* key */
    char name[10];
    UT_hash_handle hh;         /* makes this structure hashable */
};
```

2.1 The key

There are no restrictions on the data type or name of the key field. The key can also comprise multiple contiguous fields, having any names and data types.

> Any data type... really? Yes, your key and structure can have any data type. Unlike function calls with fixed prototypes, uthash consists of macros-- whose arguments are untyped-- and thus able to work with any type of structure or key.

2.1.1 Unique keys

As with any hash, every item must have a unique key. Your application must enforce key uniqueness. Before you add an item to the hash table, you must first know (if in doubt, check!) that the key is not already in use. You can check whether a key already exists in the hash table using `HASH_FIND`.

2.2 The hash handle

The `UT_hash_handle` field must be present in your structure. It is used for the internal bookkeeping that makes the hash work. It does not require initialization. It can be named anything, but you can simplify matters by naming it `hh`. This allows you to use the easier "convenience" macros to add, find and delete items.

3 Add, find, delete, count, sort, iterate

This section introduces the uthash macros by example. For a more succinct listing, see Appendix F: Macro Reference.

> Convenience vs. general macros: The uthash macros fall into two categories. The *convenience* macros can be used with integer or string keys (and require that you chose the conventional name `hh` for the `UT_hash_handle` field). The convenience macros take fewer arguments than the general macros, making their usage a bit simpler for these common types of keys.
>
> The *general* macros can be used for any types of keys, or for multi-field keys, or when the `UT_hash_handle` has been named something other than `hh`. These macros take more arguments and offer greater flexibility in return. But if the convenience macros suit your needs, use them-- your code will be more readable.

3.1 Declare the hash

Your hash must be declared as a `NULL`-initialized pointer to your structure.

```
struct my_struct *users = NULL;     /* important! initialize to NULL */
```

3.2 Add item

Allocate and initialize your structure as you see fit. The only aspect of this that matters to uthash is that your key must be initialized to a unique value. Then call `HASH_ADD`. (Here we use the convenience macro `HASH_ADD_INT`, which offers simplified usage for keys of type `int`).

Example 3.1 Add an item to a hash

```
int add_user(int user_id, char *name) {
    struct my_struct *s;

    s = malloc(sizeof(struct my_struct));
    s->id = user_id;
    strcpy(s->name, name);
    HASH_ADD_INT( users, id, s );  /* id: name of key field */
}
```

The first parameter to HASH_ADD_INT is the hash table, and the second parameter is the *name* of the key field. Here, this is id. The last parameter is a pointer to the structure being added.

Wait.. the field name is a parameter? If you find it strange that id, which is the *name of a field* in the structure, can be passed as a parameter, welcome to the world of macros. Don't worry- the C preprocessor expands this to valid C code.

3.2.1 Key must not be modified while in-use

Once a structure has been added to the hash, do not change the value of its key. Instead, delete the item from the hash, change the key, and then re-add it.

3.3 Find item

To look up a structure in a hash, you need its key. Then call HASH_FIND. (Here we use the convenience macro HASH_FIND_INT for keys of type int).

Example 3.2 Find a structure using its key

```
struct my_struct *find_user(int user_id) {
    struct my_struct *s;

    HASH_FIND_INT( users, &user_id, s );  /* s: output pointer */
    return s;
}
```

Here, the hash table is users, and &user_id points to the key (an integer in this case). Last, s is the *output* variable of HASH_FIND_INT. The final result is that s points to the structure with the given key, or is NULL if the key wasn't found in the hash.

Note
The middle argument is a *pointer* to the key. You can't pass a literal key value to HASH_FIND. Instead assign the literal value to a variable, and pass a pointer to the variable.

3.4 Delete item

To delete a structure from a hash, you must have a pointer to it. (If you only have the key, first do a HASH_FIND to get the structure pointer).

Example 3.3 Delete an item from a hash

```
void delete_user(struct my_struct *user) {
    HASH_DEL( users, user);  /* user: pointer to deletee */
    free(user);              /* optional; it's up to you! */
}
```

Here again, `users` is the hash table, and `user` is a pointer to the structure we want to remove from the hash.

3.4.1 uthash never frees your structure

Deleting a structure just removes it from the hash table-- it doesn't `free` it. The choice of when to free your structure is entirely up to you; uthash will never free your structure.

3.4.2 Delete can change the pointer

The hash table pointer (which initially points to the first item added to the hash) can change in response to `HASH_DEL` (i.e. if you delete the first item in the hash table).

3.5 Delete all items

To delete all the structures in a hash table, you need to iteratively delete. It's easy: just keep deleting the first item. If you plan to free it, copy the pointer beforehand since the delete will advance the "first item" to the next.

Example 3.4 Delete all items from a hash

```
void delete_all() {
  user_struct *current_user;

  while(users) {
    current_user = users;          /* copy pointer to first item     */
    HASH_DEL(users,current_user);  /* delete; users advances to next */
    free(current_user);            /* optional- if you want to free  */
  }
}
```

If you only want to delete all the items, but not free them, you can write this even more concisely as

```
while (users) HASH_DEL(users,users);
```

3.6 Count items

The number of items in the hash table can be obtained using `HASH_COUNT`:

Example 3.5 Count of items in the hash table

```
unsigned int num_users;
num_users = HASH_COUNT(users);
printf("there are %u users\n", num_users);
```

Incidentally, this works even the list (`users`, here) is `NULL`, in which case the count is 0.

3.7 Iterating and sorting

You can loop over the items in the hash by starting from the beginning and following the `hh.next` pointer.

Example 3.6 Iterating over all the items in a hash

```
void print_users() {
    struct my_struct *s;

    for(s=users; s != NULL; s=s->hh.next) {
        printf("user id %d: name %s\n", s->id, s->name);
    }
}
```

There is also an `hh.prev` pointer you could use to iterate backwards through the hash, starting from any known item.

A hash is also a doubly-linked list. Iterating backward and forward through the items in the hash is possible because of the `hh.prev` and `hh.next` fields. All the items in the hash can be reached by repeatedly following these pointers, thus the hash is also a doubly-linked list.

If you're using uthash in a C++ program, you need an extra cast on the `for` iterator, e.g., `s=(struct my_struct*)s→hh.next`.

3.7.1 Sorted iteration

The items in the hash are, by default, traversed in the order they were added ("insertion order") when you follow the `hh.next` pointer. But you can sort the items into a new order using `HASH_SORT`. E.g.,

```
HASH_SORT( users, name_sort );
```

The second argument is a pointer to a comparison function. It must accept two arguments which are pointers to two items to compare. Its return value should be less than zero, zero, or greater than zero, if the first item sorts before, equal to, or after the second item, respectively. (Just like `strcmp`).

Example 3.7 Sorting the items in the hash

```
int name_sort(struct my_struct *a, struct my_struct *b) {
    return strcmp(a->name,b->name);
}

int id_sort(struct my_struct *a, struct my_struct *b) {
    return (a->id - b->id);
}

void sort_by_name() {
    HASH_SORT(users, name_sort);
}

void sort_by_id() {
    HASH_SORT(users, id_sort);
}
```

When the items in the hash are sorted, the first item may change position. In the example above, `users` may point to a different structure after calling `HASH_SORT`.

3.8 A complete example

We'll repeat all the code and embellish it with a `main()` function to form a working example.

If this code was placed in a file called `example.c` in the same directory as `uthash.h`, it could be compiled and run like this:

```
cc -o example example.c
./example
```

Follow the prompts to try the program, and type `Ctrl-C` when done.

Example 3.8 A complete program (part 1 of 2)

```
#include <stdio.h>   /* gets */
#include <stdlib.h>  /* atoi, malloc */
#include <string.h>  /* strcpy */
#include "uthash.h"

struct my_struct {
    int id;                    /* key */
    char name[10];
    UT_hash_handle hh;         /* makes this structure hashable */
};

struct my_struct *users = NULL;

int add_user(int user_id, char *name) {
    struct my_struct *s;

    s = malloc(sizeof(struct my_struct));
    s->id = user_id;
    strcpy(s->name, name);
    HASH_ADD_INT( users, id, s );  /* id: name of key field */
}

struct my_struct *find_user(int user_id) {
    struct my_struct *s;

    HASH_FIND_INT( users, &user_id, s );  /* s: output pointer */
    return s;
}

void delete_user(struct my_struct *user) {
    HASH_DEL( users, user);  /* user: pointer to deletee */
    free(user);
}

void delete_all() {
  struct my_struct *current_user;

  while(users) {
    current_user = users;          /* grab pointer to first item */
    HASH_DEL(users,current_user);  /* delete it (users advances to next) */
    free(current_user);            /* free it */
  }
}

void print_users() {
    struct my_struct *s;

    for(s=users; s != NULL; s=s->hh.next) {
        printf("user id %d: name %s\n", s->id, s->name);
    }
}

int name_sort(struct my_struct *a, struct my_struct *b) {
    return strcmp(a->name,b->name);
}

int id_sort(struct my_struct *a, struct my_struct *b) {
    return (a->id - b->id);
}
```

Example 3.9 A complete program (part 2 of 2)

```
void sort_by_name() {
    HASH_SORT(users, name_sort);
}

void sort_by_id() {
    HASH_SORT(users, id_sort);
}

int main(int argc, char *argv[]) {
    char in[10];
    int id=1;
    struct my_struct *s;
    unsigned num_users;

    while (1) {
        printf("1. add user\n");
        printf("2. find user\n");
        printf("3. delete user\n");
        printf("4. delete all users\n");
        printf("5. sort items by name\n");
        printf("6. sort items by id\n");
        printf("7. print users\n");
        printf("8. count users\n");
        gets(in);
        switch(atoi(in)) {
            case 1:
                printf("name?\n");
                add_user(id++, gets(in));
                break;
            case 2:
                printf("id?\n");
                s = find_user(atoi(gets(in)));
                printf("user: %s\n", s ? s->name : "unknown");
                break;
            case 3:
                printf("id?\n");
                s = find_user(atoi(gets(in)));
                if (s) delete_user(s);
                else printf("id unknown\n");
                break;
            case 4:
                delete_all();
                break;
            case 5:
                sort_by_name();
                break;
            case 6:
                sort_by_id();
                break;
            case 7:
                print_users();
                break;
            case 8:
                num_users=HASH_COUNT(users);
                printf("there are %u users\n", num_users);
                break;
        }
    }
}
```

This program is included in the distribution in `tests/example.c`. You can run `make example` in that directory to compile it easily.

4 Kinds of keys

4.1 Integer keys

The preceding examples demonstrated use of integer keys. To recap, use the convenience macros `HASH_ADD_INT` and `HASH_FIND_IN` for structures with integer keys. (The other operations such as `HASH_DELETE` and `HASH_SORT` are the same for all types of keys).

4.2 String keys

String keys are handled almost the same as integer keys. The convenience macros for dealing with string keys are called `HASH_ADD_STR` and `HASH_FIND_STR`.

char[ ] vs. char*
The string is *within* the structure in the next example-- `name` is a `char[10]` field. If instead our structure merely *pointed* to the key (i.e., `name` was declared `char *`), we'd use `HASH_ADD_KEYPTR`, described in Appendix F.

Example 4.1 A string-keyed hash

```
#include <string.h>  /* strcpy */
#include <stdlib.h>  /* malloc */
#include <stdio.h>   /* printf */
#include "uthash.h"

struct my_struct {
    char name[10];             /* key */
    int id;
    UT_hash_handle hh;         /* makes this structure hashable */
};


int main(int argc, char *argv[]) {
    char **n, *names[] = { "joe", "bob", "betty", NULL };
    struct my_struct *s, *users = NULL;
    int i=0;

    for (n = names; *n != NULL; n++) {
        s = malloc(sizeof(struct my_struct));
        strcpy(s->name, *n);
        s->id = i++;
        HASH_ADD_STR( users, name, s );
    }

    HASH_FIND_STR( users, "betty", s);
    if (s) printf("betty's id is %d\n", s->id);
}
```

This example is included in the distribution in `tests/test15.c`. It prints:

```
betty's id is 2
```

4.3 Binary keys

We're using the term "binary" here to simply mean an arbitrary byte sequence. Your key field can have any data type. To uthash, it is just a sequence of bytes. We'll use the general macros `HASH_ADD` and `HASH_FIND` to demonstrate usage of a floating point key of type `double`.

Example 4.2 A key of type double

```
#include <stdlib.h>
#include <stdio.h>
#include "uthash.h"

typedef struct {
    double veloc;
    /* ... other data ... */
    UT_hash_handle hh;
} veloc_t;

int main(int argc, char *argv[]) {
    veloc_t *v, *v2, *veloc_table = NULL;
    double x = 1/3.0;

    v = malloc( sizeof(*v) );
    v->veloc = x;
    HASH_ADD(hh, veloc_table, veloc, sizeof(double), v);
    HASH_FIND(hh, veloc_table, &x, sizeof(double), v2 );

    if (v2) printf("found (%.2f)\n", v2->veloc);
}
```

Note that the general macros require the name of the `UT_hash_handle` to be passed as the first argument (here, this is `hh`). The general macros are documented in Appendix F: Macro Reference.

4.4 Multi-field keys

Your key can even comprise multiple contiguous fields.

Example 4.3 A multi-field key

```
#include <stdlib.h>    /* malloc       */
#include <stddef.h>    /* offsetof     */
#include <stdio.h>     /* printf       */
#include <string.h>    /* memset       */
#include "uthash.h"

#define UTF32 1

typedef struct {
  UT_hash_handle hh;
  int len;
  char encoding;      /* these two fields */
  int text[];         /* comprise the key */
} msg_t;

int main(int argc, char *argv[]) {
    int keylen;
    msg_t *msg, *msgs = NULL;
    struct { char encoding; int text[]; } *lookup_key;

    int beijing[] = {0x5317, 0x4eac};   /* UTF-32LE for &#x5317;&#x4eac; */

    /* allocate and initialize our structure */
    msg = malloc( sizeof(msg_t) + sizeof(beijing) );
    memset(msg, 0, sizeof(msg_t)+sizeof(beijing)); /* zero fill */
    msg->len = sizeof(beijing);
    msg->encoding = UTF32;
    memcpy(msg->text, beijing, sizeof(beijing));

    /* calculate the key length including padding, using formula */
    keylen =   offsetof(msg_t, text)       /* offset of last key field */
             + sizeof(beijing)             /* size of last key field */
             - offsetof(msg_t, encoding);  /* offset of first key field */

    /* add our structure to the hash table */
    HASH_ADD( hh, msgs, encoding, keylen, msg);

    /* look it up to prove that it worked :-) */
    msg=NULL;

    lookup_key = malloc(sizeof(*lookup_key) + sizeof(beijing));
    memset(lookup_key, 0, sizeof(*lookup_key) + sizeof(beijing));
    lookup_key->encoding = UTF32;
    memcpy(lookup_key->text, beijing, sizeof(beijing));
    HASH_FIND( hh, msgs, &lookup_key->encoding, keylen, msg );
    if (msg) printf("found \n");
    free(lookup_key);
}
```

This example is included in the distribution in `tests/test22.c`.

If you use multi-field keys, recognize that the compiler pads adjacent fields (by inserting unused space between them) in order to fulfill the alignment requirement of each field. For example a structure containing a `char` followed by an `int` will normally have 3 "wasted" bytes of padding after the char, in order to make the `int` field start on a multiple-of-4 address (4 is the length of the int).

Calculating the length of a multi-field key: To determine the key length when using a multi-field key, you must include any intervening structure padding the compiler adds for alignment purposes.

An easy way to calculate the key length is to use the `offsetof` macro from `<stddef.h>`. The formula is:

```
key length =   offsetof(last_key_field)
             + sizeof(last_key_field)
             - offsetof(first_key_field)
```

In the example above, the `keylen` variable is set using this formula.

When dealing with a multi-field key, you must zero-fill your structure before `HASH_ADD`'ing it to a hash table, or using its fields in a `HASH_FIND` key.

In the previous example, `memset` is used to initialize the structure by zero-filling it. This zeroes out any padding between the key fields. If we didn't zero-fill the structure, this padding would contain random values. The random values would lead to `HASH_FIND` failures; as two "identical" keys will appear to mismatch if there are any differences within their padding.

5 Structures in multiple hash tables

A structure can be added to multiple hash tables. A few reasons you might do this include:

- each hash table may use an alternative key;
- each hash table may have its own sort order;
- or you might simply use multiple hash tables for grouping purposes. E.g., you could have users in an `admin_users` and a `users` hash table.

Your structure needs to have a `UT_hash_handle` field for each hash table to which it might be added. You can name them anything. E.g.,

```
UT_hash_handle hh1, hh2;
```

5.1 Alternative keys on the same structure

You might create a hash table keyed on an ID field, and another hash table keyed on username (if they are unique). You can add the same user structure to both hash tables, then look up a user by either their unique ID or username.

Example 5.1 A structure with two alternative keys

```
struct my_struct {
    int id;                    /* key 1 */
    char username[10];         /* key 2 */
    UT_hash_handle hh1,hh2;    /* makes this structure hashable */
};
```

In the example above, the structure can now be added to two separate hash tables. In one hash, `id` is its key, while in the other hash, `username` is its key. (There is no requirement that the two hashes have different key fields. They could both use the same key, such as `id`).

Notice the structure has two hash handles (`hh1` and `hh2`). In the code below, notice that each hash handle is used exclusively with a particular hash table. (`hh1` is always used with the `users_by_id` hash, while `hh2` is always used with the `users_by_name` hash table).

Example 5.2 Two keys on a structure

```
    struct my_struct *users_by_id = NULL, *users_by_name = NULL, *s;
    int i;
    char *name;

    s = malloc(sizeof(struct my_struct));
    s->id = 1;
    strcpy(s->username, "thanson");

    /* add the structure to both hash tables */
    HASH_ADD(hh1, users_by_id, id, sizeof(int), s);
    HASH_ADD(hh2, users_by_name, username, strlen(s->username), s);

    /* lookup user by ID in the "users_by_id" hash table */
    i=1;
    HASH_FIND(hh1, users_by_id, &i, sizeof(int), s);
    if (s) printf("found id %d: %s\n", i, s->username);

    /* lookup user by username in the "users_by_name" hash table */
    name = "thanson";
    HASH_FIND(hh2, users_by_name, name, strlen(name), s);
    if (s) printf("found user %s: %d\n", name, s->id);
```

5.2 Multiple sort orders

Extending the previous example, suppose we have many users in our `users_by_id` and our `users_by_name` hash, and that we want to sort each hash so that we can print the keys in order. We'd define two sort functions, then use `HASH_SRT`:

```
int sort_by_id(struct my_struct *a, struct my_struct *b) {
  if (a->id == b->id) return 0;
  return (a->id < b->id) ? -1 : 1;
}

int sort_by_name(struct my_struct *a, struct my_struct *b) {
  return strcmp(a->username,b->username);
}

HASH_SRT(hh1, users_by_id, sort_by_id);
HASH_SRT(hh2, users_by_name, sort_by_name);

/* now iterate over users_by_id and users_by_name in sorted order */
```

A Built-in hash functions

Internally, a hash function transforms a key into a bucket number. You don't have to take any action to use the default hash function, Jenkin's hash.

Some programs may benefit from using another of the built-in hash functions. There is a simple analysis utility included with uthash to help you determine if another hash function will give you better performance.

You can use a different hash function by compiling your program with `-DHASH_FUNCTION=HASH_xyz` where `xyz` is one of the symbolic names listed below. E.g.,

```
cc -DHASH_FUNCTION=HASH_BER -o program program.c
```

Table 1: Built-in hash functions

Symbol	Name
JEN	Jenkins (default)
BER	Bernstein
SAX	Shift-Add-Xor
OAT	One-at-a-time
FNV	Fowler/Noll/Vo

A.1 Which hash function is best?

You can easily determine the best hash function for your key domain. To do so, you'll need to run your program once in a data-collection pass, and then run the collected data through an included analysis utility.

First you must build the analysis utility. From the top-level directory,

```
cd tests/
make
```

We'll use `test14.c` to demonstrate the data-collection and analysis steps (here using `sh` syntax to redirect file descriptor 3 to a file):

Example A.1 Using keystats

```
% cc -DHASH_EMIT_KEYS=3 -I../src -o test14 test14.c
% ./test14 3>test14.keys
% ./keystats test14.keys
fcn  ideal%     #items   #buckets  dup%  fl   add_usec  find_usec  del-all usec
---  ------ ---------- ---------- -----  -- ---------- ----------  ------------
FNV   90.3%       1219        512    0%  ok        244        136            44
SAX   88.7%       1219        512    0%  ok        201        145            46
OAT   87.2%       1219        256    0%  ok        166        214            40
JEN   86.7%       1219        256    0%  ok        266        221            40
BER   86.2%       1219        256    0%  ok        171        155            45
```

Note
The value 3 in `-DHASH_EMIT_KEYS=3` is a file descriptor. Any file descriptor that your program doesn't use for its own purposes can be used instead of 3. The data-collection mode enabled by `-DHASH_EMIT_KEYS=x` should not be used in production code.

Usually, you should just pick the first hash function that is listed. Here, this is `FNV`. This is the function that provides the most even distribution for your keys. If several have the same `ideal%`, then choose the fastest one according to the `find_usec` column.

A.2 keystats column reference

fcn
symbolic name of hash function

ideal%
The percentage of items in the hash table which can be looked up within an ideal number of steps. (Further explained below).

#items
the number of keys that were read in from the emitted key file

#buckets
the number of buckets in the hash after all the keys were added

dup%
the percent of duplicate keys encountered in the emitted key file. Duplicates keys are filtered out to maintain key uniqueness. (Duplicates are normal. For example, if the application adds an item to a hash, deletes it, then re-adds it, the key is written twice to the emitted file.)

flags
this is either `ok`, or `nx` (noexpand) if the expansion inhibited flag is set, described in Appendix B. It is not recommended to use a hash function that has the `noexpand` flag set.

add_usec
the clock time in microseconds required to add all the keys to a hash

find_usec
the clock time in microseconds required to look up every key in the hash

del-all usec
the clock time in microseconds required to delete every item in the hash

A.3 ideal%

What is ideal%? The *n* items in a hash are distributed into *k* buckets. Ideally each bucket would contain an equal share *(n/k)* of the items. In other words, the maximum linear position of any item in a bucket chain would be *n/k* if every bucket is equally used. If some buckets are overused and others are underused, the overused buckets will contain items whose linear position surpasses *n/k*. Such items are considered non-ideal.

As you might guess, `ideal%` is the percentage of ideal items in the hash. These items have favorable linear positions in their bucket chains. As `ideal%` approaches 100%, the hash table approaches constant-time lookup performance.

B Expansion internals

Internally this hash manages the number of buckets, with the goal of having enough buckets so that each one contains only a small number of items.

Why does the number of buckets matter? When looking up an item by its key, this hash scans linearly through the items in the appropriate bucket. In order for the linear scan to run in constant time, the number of items in each bucket must be bounded. This is accomplished by increasing the number of buckets as needed.

B.1 Normal expansion

This hash attempts to keep fewer than 10 items in each bucket. When an item is added that would cause a bucket to exceed this number, the number of buckets in the hash is doubled and the items are redistributed into the new buckets. In an ideal world, each bucket will then contain half as many items as it did before.

Bucket expansion occurs automatically and invisibly as needed. There is no need for the application to know when it occurs.

B.1.1 Per-bucket expansion threshold

Normally all buckets share the same threshold (10 items) at which point bucket expansion is triggered. During the process of bucket expansion, uthash can adjust this expansion-trigger threshold on a per-bucket basis if it sees that certain buckets are over-utilized.

When this threshold is adjusted, it goes from 10 to a multiple of 10 (for that particular bucket). The multiple is based on how many times greater the actual chain length is than the ideal length. It is a practical measure to reduce excess bucket expansion in the case where a hash function over-utilizes a few buckets but has good overall distribution. However, if the overall distribution gets too bad, uthash changes tactics.

B.2 Inhibited expansion

You usually don't need to know or worry about this, particularly if you used the `keystats` utility during development to select a good hash for your keys.

A hash function may yield an uneven distribution of items across the buckets. In moderation this is not a problem. Normal bucket expansion takes place as the chain lengths grow. But when significant imbalance occurs (because the hash function is not well suited to the key domain), bucket expansion may be ineffective at reducing the chain lengths.

Imagine a very bad hash function which always puts every item in bucket 0. No matter how many times the number of buckets is doubled, the chain length of bucket 0 stays the same. In a situation like this, the best behavior is to stop expanding, and accept O(n) lookup performance. This is what uthash does. It degrades gracefully if the hash function is ill-suited to the keys.

If two consecutive bucket expansions yield `ideal%` values below 50%, uthash inhibits expansion for that hash table. Once set, the *bucket expansion inhibited* flag remains in effect as long as the hash has items in it. Inhibited expansion may cause `HASH_FIND` to exhibit worse than constant-time performance.

C Hooks

You don't need to use these hooks- they are only here if you want to modify the behavior of uthash. Hooks can be used to change how uthash allocates memory, and to run code in response to certain internal events.

C.1 malloc/free

By default this hash implementation uses `malloc` and `free` to manage memory. If your application uses its own custom allocator, this hash can use them too.

Example C.1 Specifying alternate memory management functions

```
#include "uthash.h"


/* undefine the defaults */
#undef uthash_bkt_malloc
#undef uthash_bkt_free
#undef uthash_tbl_malloc
#undef uthash_tbl_free

/* re-define, specifying alternate functions */
#define uthash_bkt_malloc(sz) my_malloc(sz)  /* for UT_hash_bucket */
#define uthash_bkt_free(ptr) my_free(ptr)
#define uthash_tbl_malloc(sz) my_malloc(sz)  /* for UT_hash_table  */
#define uthash_tbl_free(ptr) my_free(ptr)

...
```

C.1.1 Why are there two pairs of malloc/free functions?

One deals with `UT_hash_bucket` structures, the other with `UT_hash_table` structures. While the two structures don't *need* to have their own allocation and free functions (indeed, the default is just to use `malloc` and `free` for both), they exist separately for each structure for convenient integration with pool or "slab" type allocators. This type of allocator provides a separate pool for each structure.

C.2 Out of memory

If memory allocation fails (i.e., the malloc function returned `NULL`), the default behavior is to terminate the process by calling `exit(-1)`. This can be modified by re-defining the `uthash_fatal` macro.

```
#undef uthash_fatal
#define uthash_fatal(msg) my_fatal_function(msg);
```

The fatal function should terminate the process; uthash does not support "returning a failure" if memory cannot be allocated.

C.3 Internal events

There is no need for the application to set these hooks or take action in response to these events. They are mainly for diagnostic purposes.

These two hooks are "notification" hooks which get executed if uthash is expanding buckets, or setting the *bucket expansion inhibited* flag. Normally both of these hooks are undefined and thus compile away to nothing.

C.3.1 Expansion notification

There is a hook for the bucket expansion event.

Example C.2 Bucket expansion hook

```
#include "uthash.h"

#undef uthash_expand_fyi
#define uthash_expand_fyi(tbl) printf("expanded to %d buckets\n", tbl->num_buckets)

...
```

C.3.2 Expansion-inhibited notification

This hook can be defined to code to execute in the event that uthash decides to set the *bucket expansion inhibited* flag.

Example C.3 Bucket expansion inhibited hook

```
#include "uthash.h"

#undef uthash_noexpand_fyi
#define uthash_noexpand_fyi printf("warning: bucket expansion inhibited\n");

...
```

D Debug mode

If a program that uses this hash is compiled with `-DHASH_DEBUG=1`, a special internal consistency-checking mode is activated. In this mode, the integrity of the whole hash is checked following every add or delete operation. This is for debugging the uthash software only, not for use in production code.

In the `tests/` directory, running `make debug` will run all the tests in this mode.

In this mode, any internal errors in the hash data structure will cause a message to be printed to `stderr` and the program to exit.

The `UT_hash_handle` data structure includes `next`, `prev`, `hh_next` and `hh_prev` fields. The former two fields determine the "application" ordering (that is, insertion order-- the order the items were added). The latter two fields determine the "bucket chain" order. These link the `UT_hash_handles` together in a doubly-linked list that is a bucket chain.

Checks performed in `-DHASH_DEBUG=1` mode:

- the hash is walked in its entirety twice: once in *bucket* order and a second time in *application* order
- the total number of items encountered in both walks is checked against the stored number
- during the walk in *bucket* order, each item's `hh_prev` pointer is compared for equality with the last visited item
- during the walk in *application* order, each item's `prev` pointer is compared for equality with the last visited item

Macro debugging: Sometimes it's difficult to interpret a compiler warning on a line which contains a macro call. In the case of uthash, one macro can expand to dozens of lines. In this case, it is helpful to expand the macros and then recompile. By doing so, the warning message will refer to the exact line within the macro.

Here is an example of how to expand the macros and then recompile. This uses the `test1.c` program in the `tests/` subdirectory.

```
gcc -E -I../src test1.c > /tmp/a.c
egrep -v '^#' /tmp/a.c > /tmp/b.c
indent /tmp/b.c
gcc -o /tmp/b /tmp/b.c
```

The last line compiles the original program (test1.c) with all macros expanded. If there was a warning, the referenced line number can be checked in `/tmp/b.c`.

E Thread safety

You can use uthash in a threaded program. But you must do the locking. Use a read-write lock to protect against concurrent writes. It is ok to have concurrent readers (since uthash 1.5).

For example using pthreads you can create an rwlock like this:

```
pthread_rwlock_t lock;
if (pthread_rwlock_init(&lock,NULL) != 0) fatal("can't create rwlock");
```

Then, readers must acquire the read lock before doing any `HASH_FIND` calls or before iterating over the hash elements:

```
if (pthread_rwlock_rdlock(&lock) != 0) fatal("can't get rdlock");
HASH_FIND_INT(elts, &i, e);
pthread_rwlock_unlock(&lock);
```

Writers must acquire the exclusive write lock before doing any update. Add, delete, and sort are all updates that must be locked.

```
if (pthread_rwlock_wrlock(&lock) != 0) fatal("can't get wrlock");
HASH_DEL(elts, e);
pthread_rwlock_unlock(&lock);
```

If you prefer, you can use a mutex instead of a read-write lock, but this will reduce reader concurrency to a single thread at a time.

An example program using uthash with a read-write lock is included in `tests/threads/test1.c`.

F Macro reference

F.1 Convenience macros

The convenience macros do the same thing as the generalized macros, but require fewer arguments.

In order to use the convenience macros,

1. the structure's `UT_hash_handle` field must be named `hh`, and
2. for add or find, the key field must be of type `int` or `char[]`

Table 2: Convenience macros

macro	arguments
HASH_ADD_INT	(head, keyfield_name, item_ptr)
HASH_FIND_INT	(head, key_ptr, item_ptr)
HASH_ADD_STR	(head, keyfield_name, item_ptr)
HASH_FIND_STR	(head, key_ptr, item_ptr)
HASH_DEL	(head, item_ptr)
HASH_SORT	(head, cmp)
HASH_COUNT	(head)

F.2 General macros

These macros add, find, delete and sort the items in a hash. You need to use the general macros if your `UT_hash_handle` is named something other than `hh`, or if your key's data type isn't `int` or `char[]`.

Table 3: General macros

macro	arguments
HASH_ADD	(hh_name, head, keyfield_name, key_len, item_ptr)
HASH_ADD_KEYPTR	(hh_name, head, key_ptr, key_len, item_ptr)
HASH_FIND	(hh_name, head, key_ptr, key_len, item_ptr)
HASH_DELETE	(hh_name, head, item_ptr)
HASH_SRT	(hh_name, head, cmp)
HASH_CNT	(hh_name, head)

Note
`HASH_ADD_KEYPTR` is used when the structure contains a pointer to the key, rather than the key itself.

F.2.1 Argument descriptions

hh_name
name of the `UT_hash_handle` field in the structure. Conventionally called `hh`.

head
the structure pointer variable which acts as the "head" of the hash. So named because it initially points to the first item that is added to the hash.

keyfield_name
the name of the key field in the structure. (In the case of a multi-field key, this is the first field of the key). If you're new to macros, it might seem strange to pass the name of a field as a parameter. See note.

key_len
the length of the key field in bytes. E.g. for an integer key, this is `sizeof(int)`, while for a string key it's `strlen(key)`. (For a multi-field key, see the notes in this guide on calculating key length).

key_ptr
for `HASH_FIND`, this is a pointer to the key to look up in the hash (since it's a pointer, you can't directly pass a literal value here). For `HASH_ADD_KEYPTR`, this is the address of the key of the item being added.

item_ptr
pointer to the structure being added, deleted or looked up. This is an input parameter for HASH_ADD and HASH_DELETE macros, and an output parameter for HASH_FIND.

cmp
pointer to comparison function which accepts two arguments (pointers to items to compare) and returns an int specifying whether the first item should sort before, equal to, or after the second item (like `strcmp`).