

uthash User Guide

=====

Troy D. Hanson, Arthur O'Dwyer
v2.3.0, February 2021

To download uthash, follow this link back to the
<https://github.com/troydhanson/uthash>[GitHub project page].
Back to my <http://troydhanson.github.io/>[other projects].

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, hashes or "dictionaries" are used all the time. In C, hashes don't exist in the language itself. This software provides a hash table for C structures.

What can it do?

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

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

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 1000 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 `<<hash_functions,built-in hash functions>>`.

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.

C/C++ and platforms

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

- * Linux
- * Windows using Visual Studio 2008 and 2010
- * Solaris
- * OpenBSD
- * FreeBSD
- * Android

Test suite

AAAAAAAAAA

To run the test suite, enter the ``tests`` directory. Then,

- * on Unix platforms, run ``make``

* on Windows, run the "do_tests_win32.cmd" batch file. (You may edit the batch file if your Visual Studio is installed in a non-standard location).

BSD licensed

This software is made available under the [link:license.html](#)[revised BSD license].
It is free and open source.

Download uthash

Follow the links on <https://github.com/troydhanson/uthash> to clone uthash or get a zip file.

Getting help

Please use the <https://groups.google.com/d/forum/uthash>[uthash Google Group] to ask questions. You can email it at uthash@googlegroups.com.

Contributing

You may submit pull requests through GitHub. However, the maintainers of uthash value keeping it unchanged, rather than adding bells and whistles.

Extras included

Three "extras" come with uthash. These provide lists, dynamic arrays and strings:

- * [link:utlist.html](#)[utlist.h] provides linked list macros for C structures.
- * [link:utarray.html](#)[utarray.h] implements dynamic arrays using macros.
- * [link:utstring.html](#)[utstring.h] implements a basic dynamic string.

History

I wrote uthash in 2004-2006 for my own purposes. Originally it was hosted on SourceForge. Uthash was downloaded around 30,000 times between 2006-2013 then transitioned to GitHub. It's been incorporated into commercial software, academic research, and into other open-source software. It has also been added to the native package repositories for a number of Unix-y distros.

When uthash was written, there were fewer options for doing generic hash tables in C than exist today. There are faster hash tables, more memory-efficient hash tables, with very different API's today. But, like driving a minivan, uthash is convenient, and gets the job done for many purposes.

As of July 2016, uthash is maintained by Arthur O'Dwyer.

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 pointer itself is the value.

.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 */
};
```

Note that, in uthash, your structure will never be moved or copied into another location when you add it into a hash table. This means that you can keep other data structures that safely point to your structure-- regardless of whether you add or delete it from a hash table during your program's lifetime.

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.

Unique keys

AAAAAAAAAAAA

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``.

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.

A word about memory

Overhead

AAAAAAAAAA

The hash handle consumes about 32 bytes per item on a 32-bit system, or 56 bytes per item on a 64-bit system. The other overhead costs-- the buckets and the table-- are negligible in comparison. You can use ``HASH_OVERHEAD`` to get the overhead size, in bytes, for a hash table. See <<Macro_reference,Macro Reference>>.

How clean up occurs

AAAAAAAAAAAAAAAAAAAA

Some have asked how uthash cleans up its internal memory. The answer is simple: 'when you delete the final item' from a hash table, uthash releases all the internal memory associated with that hash table, and sets its pointer to NULL.

Hash operations

This section introduces the uthash macros by example. For a more succinct listing, see <<Macro_reference,Macro Reference>>.

.Convenience vs. general macros:

The uthash macros fall into two categories. The 'convenience' macros can be used with integer, pointer 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.

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 */
```

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``).

.Add an item to a hash

```
-----  
void 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.

[[validc]]

.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.

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.

Checking uniqueness

^^

In the example above, we didn't check to see if ``user_id`` was already a key of some existing item in the hash. *If there's any chance that duplicate keys could be generated by your program, you must explicitly check the uniqueness* before adding the key to the hash. If the key is already in the hash, you can simply modify the existing structure in the hash rather than adding the item. 'It is an error to add two items with the same key to the hash table'.

Let's rewrite the ``add_user`` function to check whether the id is in the hash. Only if the id is not present in the hash, do we create the item and add it. Otherwise we just modify the structure that already exists.


```

-----
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.

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).

.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.

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. For example when using `HASH_REPLACE` macros, a replaced output argument is returned back, in order to make it possible for the user to de-allocate it.

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).

Iterative deletion

^^

The `HASH_ITER` macro is a deletion-safe iteration construct which expands to a simple 'for' loop.

.Delete all items from a hash

```

-----
void delete_all() {
    struct my_struct *current_user, *tmp;

    HASH_ITER(hh, users, current_user, tmp) {
        HASH_DEL(users, current_user); /* delete; users advances to next */
        free(current_user);           /* optional- if you want to free */
    }
}
-----

```

```
}
```

All-at-once deletion

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

If you only want to delete all the items, but not free them or do any per-element clean up, you can do this more efficiently in a single operation:

```
HASH_CLEAR(hh, users);
```

Afterward, the list head (here, `users`) will be set to `NULL`.

Count items

~~~~~

The number of items in the hash table can be obtained using `HASH\_COUNT`:

.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 if the list head (here, `users`) is `NULL`, in which case the count is 0.

Iterating and sorting

~~~~~

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

.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.

[[deletesafe]]

Deletion-safe iteration

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

In the example above, it would not be safe to delete and free `s` in the body of the 'for' loop, (because `s` is dereferenced each time the loop iterates). This is easy to rewrite correctly (by copying the `s->hh.next` pointer to a temporary variable 'before' freeing `s`), but it comes up often enough that a deletion-safe iteration macro, `HASH_ITER`, is included. It expands to a `for`-loop header. Here is how it could be used to rewrite the last example:

```
struct my_struct *s, *tmp;  
  
HASH_ITER(hh, users, s, tmp) {  
    printf("user id %d: name %s\n", s->id, s->name);  
    /* ... it is safe to delete and free s here */  
}
```

.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 = static_cast<my_struct*>(s->hh.next)`.

Sorting

^^^^^^

The items in the hash are visited in "insertion order" when you follow the `hh.next` pointer. You can sort the items into a new order using `HASH_SORT`.

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

The second argument is a pointer to a comparison function. It must accept two pointer arguments (the items to compare), and must return an `int` which is less than zero, zero, or greater than zero, if the first item sorts before, equal to, or after the second item, respectively. (This is the same convention used by `strcmp` or `qsort` in the standard C library).

```
int sort_function(void *a, void *b) {
    /* compare a to b (cast a and b appropriately)
     * return (int) -1 if (a < b)
     * return (int)  0 if (a == b)
     * return (int)  1 if (a > b)
     */
}
```

Below, `name_sort` and `id_sort` are two examples of sort functions.

.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`.

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.

.A complete program

```
-----
#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;

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

    HASH_FIND_INT(users, &user_id, s); /* id already in the hash? */
    if (s == NULL) {
        s = (struct my_struct *)malloc(sizeof *s);
        s->id = user_id;
        HASH_ADD_INT(users, id, s); /* id: name of key field */
    }
    strcpy(s->name, name);
}

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, *tmp;

    HASH_ITER(hh, users, current_user, tmp) {
        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 = (struct my_struct*)(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);
}

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, running = 1;
    struct my_struct *s;
    unsigned num_users;

    while (running) {
        printf(" 1. add user\n");
        printf(" 2. add/rename user by id\n");
        printf(" 3. find user\n");
        printf(" 4. delete user\n");
        printf(" 5. delete all users\n");
        printf(" 6. sort items by name\n");
        printf(" 7. sort items by id\n");
        printf(" 8. print users\n");
        printf(" 9. count users\n");
        printf("10. quit\n");
        gets(in);
        switch(atoi(in)) {
            case 1:
                printf("name?\n");
                add_user(id++, gets(in));
                break;
            case 2:
                printf("id?\n");
                gets(in); id = atoi(in);
                printf("name?\n");
                add_user(id, gets(in));
                break;
            case 3:
                printf("id?\n");
                s = find_user(atoi(gets(in)));
                printf("user: %s\n", s ? s->name : "unknown");
                break;
            case 4:
                printf("id?\n");
                s = find_user(atoi(gets(in)));
                if (s) delete_user(s);
                else printf("id unknown\n");
                break;
            case 5:
                delete_all();
                break;
            case 6:
                sort_by_name();
                break;
            case 7:
                sort_by_id();
                break;
        }
    }
}

```

```

        case 8:
            print_users();
            break;
        case 9:
            num_users = HASH_COUNT(users);
            printf("there are %u users\n", num_users);
            break;
        case 10:
            running = 0;
            break;
    }
}

delete_all(); /* free any structures */
return 0;
}

```

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

Standard key types

This section goes into specifics of how to work with different kinds of keys. You can use nearly any type of key-- integers, strings, pointers, structures, etc.

[NOTE]

.A note about float

=====

You can use floating point keys. This comes with the same caveats as with any program that tests floating point equality. In other words, even the tiniest difference in two floating point numbers makes them distinct keys.

=====

Integer keys

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

String keys

If your structure has a string key, the operations to use depend on whether your structure 'points to' the key (`char *`) or the string resides 'within' the structure (`char a[10]`). **This distinction is important**. As we'll see below, you need to use `HASH_ADD_KEYPTR` when your structure 'points' to a key (that is, the key itself is 'outside' of the structure); in contrast, use `HASH_ADD_STR` for a string key that is contained **within** your structure.

[NOTE]

.char[] vs. char*

=====

The string is 'within' the structure in the first example below-- `name` is a `char[10]` field. In the second example, the key is 'outside' of the structure-- `name` is a `char *`. So the first example uses `HASH_ADD_STR` but the second example uses `HASH_ADD_KEYPTR`. For information on this macro, see the <<Macro_reference,Macro reference>>.

=====

String 'within' structure

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

.A string-keyed hash (string within structure)

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

struct my_struct {
    char name[10];          /* key (string is WITHIN the structure) */
    int id;
    UT_hash_handle hh;     /* makes this structure hashable */
};

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

    for (int i = 0; names[i]; ++i) {
        s = (struct my_struct *)malloc(sizeof *s);
        strcpy(s->name, names[i]);
        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);

    /* free the hash table contents */
    HASH_ITER(hh, users, s, tmp) {
        HASH_DEL(users, s);
        free(s);
    }
    return 0;
}
-----
```

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

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

String 'pointer' in structure
^^

Now, here is the same example but using a `char *` key instead of `char []`:

.A string-keyed hash (structure points to string)

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

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

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

```

```

for (int i = 0; names[i]; ++i) {
    s = (struct my_struct *)malloc(sizeof *s);
    s->name = names[i];
    s->id = i;
    HASH_ADD_KEYPTR(hh, users, s->name, strlen(s->name), s);
}

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

/* free the hash table contents */
HASH_ITER(hh, users, s, tmp) {
    HASH_DEL(users, s);
    free(s);
}
return 0;
}

```

This example is included in `tests/test40.c`.

Pointer keys

~~~~~

Your key can be a pointer. To be very clear, this means the 'pointer itself' can be the key (in contrast, if the thing 'pointed to' is the key, this is a different use case handled by `HASH\_ADD\_KEYPTR`).

Here is a simple example where a structure has a pointer member, called `key`.

#### .A pointer key

```

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

typedef struct {
    void *key;
    int i;
    UT_hash_handle hh;
} el_t;

el_t *hash = NULL;
char *someaddr = NULL;

int main() {
    el_t *d;
    el_t *e = (el_t *)malloc(sizeof *e);
    if (!e) return -1;
    e->key = (void*)someaddr;
    e->i = 1;
    HASH_ADD_PTR(hash, key, e);
    HASH_FIND_PTR(hash, &someaddr, d);
    if (d) printf("found\n");

    /* release memory */
    HASH_DEL(hash, e);
    free(e);
    return 0;
}

```

-----

This example is included in `tests/test57.c`. Note that the end of the program deletes the element out of the hash, (and since no more elements remain in the

hash), uthash releases its internal memory.

## Structure keys

Your key field can have any data type. To uthash, it is just a sequence of bytes. Therefore, even a nested structure can be used as a key. We'll use the general macros `HASH_ADD` and `HASH_FIND` to demonstrate.

NOTE: Structures contain padding (wasted internal space used to fulfill alignment requirements for the members of the structure). These padding bytes 'must be zeroed' before adding an item to the hash or looking up an item. Therefore always zero the whole structure before setting the members of interest. The example below does this-- see the two calls to `memset`.

.A key which is a structure

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

typedef struct {
    char a;
    int b;
} record_key_t;

typedef struct {
    record_key_t key;
    /* ... other data ... */
    UT_hash_handle hh;
} record_t;

int main(int argc, char *argv[]) {
    record_t l, *p, *r, *tmp, *records = NULL;

    r = (record_t *)malloc(sizeof *r);
    memset(r, 0, sizeof *r);
    r->key.a = 'a';
    r->key.b = 1;
    HASH_ADD(hh, records, key, sizeof(record_key_t), r);

    memset(&l, 0, sizeof(record_t));
    l.key.a = 'a';
    l.key.b = 1;
    HASH_FIND(hh, records, &l.key, sizeof(record_key_t), p);

    if (p) printf("found %c %d\n", p->key.a, p->key.b);

    HASH_ITER(hh, records, p, tmp) {
        HASH_DEL(records, p);
        free(p);
    }
    return 0;
}
-----
```

This usage is nearly the same as use of a compound key explained below.

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 <<Macro\_reference,Macro Reference>>.

## Advanced Topics

## Compound keys

Your key can even comprise multiple contiguous fields.

.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;

typedef struct {
    char encoding;
    int text[];
} lookup_key_t;

int main(int argc, char *argv[]) {
    unsigned keylen;
    msg_t *msg, *tmp, *msgs = NULL;
    lookup_key_t *lookup_key;

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

    /* allocate and initialize our structure */
    msg = (msg_t *)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 = (lookup_key_t *)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);

    HASH_ITER(hh, msgs, msg, tmp) {
        HASH_DEL(msgs, msg);
        free(msg);
    }
}
```

```

    return 0;
}
-----

```

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).

[[multifield\_note]]

.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 ``. 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.

Alternatively, you can customize the global <<hash\_keycompare,key comparison function>> and <<hash\_functions,key hashing function>> to ignore the padding in your key. See <<hash\_keycompare,Specifying an alternate key comparison function>>.

[[multilevel]]

Multi-level hash tables

-----

A multi-level hash table arises when each element of a hash table contains its own secondary hash table. There can be any number of levels. In a scripting language you might see:

```

$items{bob}{age}=37

```

The C program below builds this example in uthash: the hash table is called `items`. It contains one element (`bob`) whose own hash table contains one element (`age`) with value 37. No special functions are necessary to build a multi-level hash table.

While this example represents both levels (`bob` and `age`) using the same structure, it would also be fine to use two different structure definitions. It would also be fine if there were three or more levels instead of two.

.Multi-level hash table

-----

```

#include <stdio.h>

```



```

#include <string.h>
#include <stdlib.h>
#include "uthash.h"

/* hash of hashes */
typedef struct item {
    char name[10];
    struct item *sub;
    int val;
    UT_hash_handle hh;
} item_t;

item_t *items = NULL;

int main(int argc, char *argvp[]) {
    item_t *item1, *item2, *tmp1, *tmp2;

    /* make initial element */
    item_t *i = malloc(sizeof(*i));
    strcpy(i->name, "bob");
    i->sub = NULL;
    i->val = 0;
    HASH_ADD_STR(items, name, i);

    /* add a sub hash table off this element */
    item_t *s = malloc(sizeof(*s));
    strcpy(s->name, "age");
    s->sub = NULL;
    s->val = 37;
    HASH_ADD_STR(i->sub, name, s);

    /* iterate over hash elements */
    HASH_ITER(hh, items, item1, tmp1) {
        HASH_ITER(hh, item1->sub, item2, tmp2) {
            printf("$items{%s}{%s} = %d\n", item1->name, item2->name, item2->val);
        }
    }

    /* clean up both hash tables */
    HASH_ITER(hh, items, item1, tmp1) {
        HASH_ITER(hh, item1->sub, item2, tmp2) {
            HASH_DEL(item1->sub, item2);
            free(item2);
        }
        HASH_DEL(items, item1);
        free(item1);
    }

    return 0;
}

```

-----  
The example above is included in `tests/test59.c`.

[[multihash]]  
Items in several hash tables

~~~~~  
A structure can be added to more than one hash table. A few reasons you might do this include:

- each hash table may use a different 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;
```

Items with multiple keys

~~~~~

You might create a hash table keyed on an ID field, and another hash table keyed on username (if usernames are unique). You can add the same user structure to both hash tables (without duplication of the structure), allowing lookup of a user structure by their name or ID. The way to achieve this is to have a separate `UT\_hash\_handle` for each hash to which the structure may be added.

.A structure with two different keys

```
-----  
struct my_struct {  
    int id;                /* first key */  
    char username[10];     /* second key */  
    UT_hash_handle hh1;    /* handle for first hash table */  
    UT_hash_handle hh2;    /* handle for second hash table */  
};  
-----
```

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).

.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);  
  
    /* find 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);  
  
    /* find 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);  
-----
```

Sorted insertion of new items

~~~~~

If you would like to maintain a sorted hash you have two options. The first option is to use the HASH_SRT() macro, which will sort any unordered list in 'O(n log(n))'. This is the best strategy if you're just filling up a hash

table with items in random order with a single final `HASH_SRT()` operation when all is done. Obviously, this won't do what you want if you need the list to be in an ordered state at times between insertion of items. You can use `HASH_SRT()` after every insertion operation, but that will yield a computational complexity of $O(n^2 \log n)$.

The second route you can take is via the in-order add and replace macros. The ``HASH_ADD_INORDER*`` macros work just like their ``HASH_ADD*`` counterparts, but with an additional comparison-function argument:

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

HASH_ADD_KEYPTR_INORDER(hh, items, &item->name, strlen(item->name), item,
name_sort);
```

New items are sorted at insertion time in $O(n)$, thus resulting in a total computational complexity of $O(n^2)$ for the creation of the hash table with all items.

For in-order add to work, the list must be in an ordered state before insertion of the new item.

Several sort orders

~~~~~  
It comes as no surprise that two hash tables can have different sort orders, but this fact can also be used advantageously to sort the 'same items' in several ways. This is based on the ability to store a structure in several hash tables.

Extending the previous example, suppose we have many users. We have added each user structure to the ``users_by_id`` hash table and the ``users_by_name`` hash table.

(To reiterate, this is done without the need to have two copies of each structure.)

Now we can 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 iterating over the items in ``users_by_id`` will traverse them in id-order while, naturally, iterating over ``users_by_name`` will traverse them in name-order. The items are fully forward-and-backward linked in each order. So even for one set of users, we might store them in two hash tables to provide easy iteration in two different sort orders.

#### Bloom filter (faster misses)

~~~~~  
Programs that generate a fair miss rate (``HASH_FIND`` that result in ``NULL``) may benefit from the built-in Bloom filter support. This is disabled by default, because programs that generate only hits would incur a slight penalty from it. Also, programs that do deletes should not use the Bloom filter. While the program would operate correctly, deletes diminish the benefit of the filter. To enable the Bloom filter, simply compile with ``-DHASH_BLOOM=n`` like:

-DHASH_BLOOM=27

where the number can be any value up to 32 which determines the amount of memory used by the filter, as shown below. Using more memory makes the filter more accurate and has the potential to speed up your program by making misses bail out faster.

```
.Bloom filter sizes for selected values of n
[width="50%",cols="10m,30",grid="none",options="header"]
|=====
| n   | Bloom filter size (per hash table)
| 16  | 8 kilobytes
| 20  | 128 kilobytes
| 24  | 2 megabytes
| 28  | 32 megabytes
| 32  | 512 megabytes
|=====
```

Bloom filters are only a performance feature; they do not change the results of hash operations in any way. The only way to gauge whether or not a Bloom filter is right for your program is to test it. Reasonable values for the size of the Bloom filter are 16-32 bits.

Select

~~~~~

An experimental 'select' operation is provided that inserts those items from a source hash that satisfy a given condition into a destination hash. This insertion is done with somewhat more efficiency than if this were using ``HASH_ADD``, namely because the hash function is not recalculated for keys of the selected items. This operation does not remove any items from the source hash. Rather the selected items obtain dual presence in both hashes. The destination hash may already have items in it; the selected items are added to it. In order for a structure to be usable with ``HASH_SELECT``, it must have two or more hash handles. (As described <<multihash,here>>, a structure can exist in many hash tables at the same time; it must have a separate hash handle for each one).

```
user_t *users = NULL; /* hash table of users */
user_t *admins = NULL; /* hash table of admins */

typedef struct {
    int id;
    UT_hash_handle hh; /* handle for users hash */
    UT_hash_handle ah; /* handle for admins hash */
} user_t;
```

Now suppose we have added some users, and want to select just the administrator users who have id's less than 1024.

```
#define is_admin(x) (((user_t*)x)->id < 1024)
HASH_SELECT(ah, admins, hh, users, is_admin);
```

The first two parameters are the 'destination' hash handle and hash table, the second two parameters are the 'source' hash handle and hash table, and the last parameter is the 'select condition'. Here we used a macro ``is_admin(x)`` but we could just as well have used a function.

```
int is_admin(const void *user) {
    user_t *user = (const user_t*)user;
    return (user->id < 1024) ? 1 : 0;
}
```

If the select condition always evaluates to true, this operation is essentially a 'merge' of the source hash into the destination hash.

`HASH\_SELECT` adds items to the destination without removing them from the source; the source hash table remains unchanged. The destination hash table must not be the same as the source hash table.

An example of using `HASH\_SELECT` is included in `tests/test36.c`.

```
[[hash_keycompare]]  
Specifying an alternate key comparison function  
-----
```

When you call `HASH\_FIND(hh, head, intfield, sizeof(int), out)`, uthash will first call <<hash\_functions,`HASH\_FUNCTION`>>`(intfield, sizeof(int), hashvalue)` to determine the bucket `b` in which to search, and then, for each element `elt` of bucket `b`, uthash will evaluate  
`elt->hh.hashv == hashvalue && elt->hh.keylen == sizeof(int) && HASH\_KEYCMP(intfield, elt->hh.key, sizeof(int)) == 0`.  
`HASH\_KEYCMP` should return `0` to indicate that `elt` is a match and should be returned, and any non-zero value to indicate that the search for a matching element should continue.

By default, uthash defines `HASH\_KEYCMP` as an alias for `memcmp`. On platforms that do not provide `memcmp`, you can substitute your own implementation.

```
-----  
#undef HASH_KEYCMP  
#define HASH_KEYCMP(a,b,len) bcmp(a, b, len)  
-----
```

Another reason to substitute your own key comparison function is if your "key" is not trivially comparable. In this case you will also need to substitute your own `HASH\_FUNCTION`.

```
-----  
struct Key {  
    short s;  
    /* 2 bytes of padding */  
    float f;  
};  
/* do not compare the padding bytes; do not use memcmp on floats */  
unsigned key_hash(struct Key *s) { return s + (unsigned)f; }  
bool key_equal(struct Key *a, struct Key *b) { return a.s == b.s && a.f == b.f; }  
  
#define HASH_FUNCTION(s,len,hashv) (hashv) = key_hash((struct Key *)s)  
#define HASH_KEYCMP(a,b,len) (!key_equal((struct Key *)a, (struct Key *)b))  
-----
```

Another reason to substitute your own key comparison function is to trade off correctness for raw speed. During its linear search of a bucket, uthash always compares the 32-bit `hashv` first, and calls `HASH\_KEYCMP` only if the `hashv` compares equal. This means that `HASH\_KEYCMP` is called at least once per successful find. Given a good hash function, we expect the `hashv` comparison to produce a "false positive" equality only once in four billion times. Therefore, we expect `HASH\_KEYCMP` to produce `0` most of the time. If we expect many successful finds, and our application doesn't mind the occasional false positive, we might substitute a no-op comparison function:

```
-----  
#undef HASH_KEYCMP  
#define HASH_KEYCMP(a,b,len) 0 /* occasionally wrong, but very fast */  
-----
```

Note: The global equality-comparison function `HASH\_KEYCMP` has no relationship at all to the less-than-comparison function passed as a parameter to `HASH\_ADD\_INORDER`.

[[hash\_functions]]

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, currently Jenkins.

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
```

.Built-in hash functions

[width="50%",cols="^5m,20",grid="none",options="header"]

```
=====
```

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

```
=====
```

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):

.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 |
|-----|--------|--------|----------|------|----|----------|-----------|---------|------|
| SFH | 91.6%  | 1219   | 256      | 0%   | ok | 92       | 131       |         | 25   |
| FNV | 90.3%  | 1219   | 512      | 0%   | ok | 107      | 97        |         | 31   |
| SAX | 88.7%  | 1219   | 512      | 0%   | ok | 111      | 109       |         | 32   |
| OAT | 87.2%  | 1219   | 256      | 0%   | ok | 99       | 138       |         | 26   |
| JEN | 86.7%  | 1219   | 256      | 0%   | ok | 87       | 130       |         | 27   |
| BER | 86.2%  | 1219   | 256      | 0%   | ok | 121      | 129       |         | 27   |

```
-----
```

[NOTE]

The number 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 ``SFH``. 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.

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 `<<expansion,Expansion internals>>`. 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

[[ideal]]

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.

\*\*\*\*\*

[[hashscan]]

hashscan

~~~~~

NOTE: This utility is only available on Linux, and on FreeBSD (8.1 and up).

A utility called ``hashscan`` is included in the ``tests/`` directory. It is built automatically when you run ``make`` in that directory. This tool examines a running process and reports on the uthash tables that it finds in that program's memory. It can also save the keys from each table in a format that can be fed into ``keystats``.

Here is an example of using `hashscan`. First ensure that it is built:

```
cd tests/  
make
```

Since `hashscan` needs a running program to inspect, we'll start up a simple program that makes a hash table and then sleeps as our test subject:

```
./test_sleep &  
pid: 9711
```

Now that we have a test program, let's run `hashscan` on it:

```
./hashscan 9711  
Address          ideal    items  buckets mc fl bloom/sat fcn keys saved to  
-----  
0x862e038        81%    10000   4096 11 ok 16    14% JEN
```

If we wanted to copy out all its keys for external analysis using `keystats`, add the `-k` flag:

```
./hashscan -k 9711  
Address          ideal    items  buckets mc fl bloom/sat fcn keys saved to  
-----  
0x862e038        81%    10000   4096 11 ok 16    14% JEN /tmp/9711-0.key
```

Now we could run `./keystats /tmp/9711-0.key` to analyze which hash function has the best characteristics on this set of keys.

hashscan column reference

^^

Address::

virtual address of the hash table

ideal::

The percentage of items in the table which can be looked up within an ideal number of steps. See <<ideal>> in the `keystats` section.

items::

number of items in the hash table

buckets::

number of buckets in the hash table

mc::

the maximum chain length found in the hash table (uthash usually tries to keep fewer than 10 items in each bucket, or in some cases a multiple of 10)

fl::

flags (either `ok`, or `NX` if the expansion-inhibited flag is set)

bloom/sat::

if the hash table uses a Bloom filter, this is the size (as a power of two) of the filter (e.g. 16 means the filter is 2¹⁶ bits in size). The second number is the "saturation" of the bits expressed as a percentage. The lower the percentage, the more potential benefit to identify cache misses quickly.

fcn::

symbolic name of hash function

keys saved to::

file to which keys were saved, if any

.How hashscan works

When hashscan runs, it attaches itself to the target process, which suspends the target process momentarily. During this brief suspension, it scans the target's virtual memory for the signature of a uthash hash table. It then checks if a valid hash table structure accompanies the signature and reports what it finds. When it detaches, the target process resumes running normally. The hashscan is performed "read-only"-- the target process is not modified.

Since hashscan is analyzing a momentary snapshot of a running process, it may return different results from one run to another.

[[expansion]] 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.

Normal expansion

AAAAAAAAAAAAAAAAAAAA

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.

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.

Inhibited expansion

AAAAAAAAAAAAAAAAAAAA

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.

Diagnostic hooks

^^^^^^^^^^^^^^^^^^^^

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

The ``uthash_expand_fyi`` hook can be defined to execute code whenever uthash performs a bucket expansion.

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

The ``uthash_noexpand_fyi`` hook can be defined to execute code whenever uthash sets the 'bucket expansion inhibited' flag.

```
-----  
#undef uthash_noexpand_fyi  
#define uthash_noexpand_fyi(tbl) printf("warning: bucket expansion inhibited\n")  
-----
```

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 replace standard library functions that might be unavailable on some platforms, to change how uthash allocates memory, or to run code in response to certain internal events.

The ``uthash.h`` header will define these hooks to default values, unless they are already defined. It is safe either to ``#undef`` and redefine them after including ``uthash.h``, or to define them before inclusion; for example, by passing ``-Duthash_malloc=my_malloc`` on the command line.

### Specifying alternate memory management functions

^^^^^^^^^^^^^^^^^^^^

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

```
-----  
#include "uthash.h"  
  
/* undefine the defaults */  
#undef uthash_malloc  
#undef uthash_free  
  
/* re-define, specifying alternate functions */  
#define uthash_malloc(sz) my_malloc(sz)  
#define uthash_free(ptr, sz) my_free(ptr)  
  
...  
-----
```

Notice that ``uthash_free`` receives two parameters. The ``sz`` parameter is for convenience on embedded platforms that manage their own memory.

### Specifying alternate standard library functions

^^^^^^^^^^^^^^^^^^^^

Uthash also uses ``strlen`` (in the ``HASH_FIND_STR`` convenience macro, for

example) and `memset` (used only for zeroing memory). On platforms that do not provide these functions, you can substitute your own implementations.

```
-----  
#undef uthash_bzero  
#define uthash_bzero(a, len) my_bzero(a, len)  
  
#undef uthash_strlen  
#define uthash_strlen(s) my_strlen(s)  
-----
```

Out of memory  
^^^^^^^^^^^^^^^^

If memory allocation fails (i.e., the `uthash_malloc` function returns `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 or `longjmp` back to a safe place. Note that an allocation failure may leave allocated memory that cannot be recovered. After `uthash_fatal`, the hash table object should be considered unusable; it might not be safe even to run `HASH_CLEAR` on the hash table when it is in this state.

To enable "returning a failure" if memory cannot be allocated, define the macro `HASH_NONFATAL_OOM` before including the `uthash.h` header file. In this case, `uthash_fatal` is not used; instead, each allocation failure results in a single call to `uthash_nonfatal_oom(elt)` where `elt` is the address of the element whose insertion triggered the failure. The default behavior of `uthash_nonfatal_oom` is a no-op.

```
-----  
#undef uthash_nonfatal_oom  
#define uthash_nonfatal_oom(elt) perhaps_recover((element_t *) elt)  
-----
```

Before the call to `uthash_nonfatal_oom`, the hash table is rolled back to the state it was in prior to the problematic insertion; no memory is leaked. It is safe to `throw` or `longjmp` out of the `uthash_nonfatal_oom` handler.

The `elt` argument will be of the correct pointer-to-element type, unless `uthash_nonfatal_oom` is invoked from `HASH_SELECT`, in which case it will be of `void*` type and must be cast before using. In any case, `elt->hh.tbl` will be `NULL`.

Allocation failure is possible only when adding elements to the hash table (including the `ADD`, `REPLACE`, and `SELECT` operations). `uthash_free` is not allowed to fail.

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`.

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``.

## [[Macro\_reference]]

### Macro reference

-----

#### 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[]`` or pointer

.Convenience macros

```
[width="90%",cols="10m,30m",grid="none",options="header"]
```

```
=====
```

macro	arguments
HASH_ADD_INT	(head, keyfield_name, item_ptr)
HASH_REPLACE_INT	(head, keyfield_name, item_ptr, replaced_item_ptr)
HASH_FIND_INT	(head, key_ptr, item_ptr)
HASH_ADD_STR	(head, keyfield_name, item_ptr)
HASH_REPLACE_STR	(head, keyfield_name, item_ptr, replaced_item_ptr)
HASH_FIND_STR	(head, key_ptr, item_ptr)
HASH_ADD_PTR	(head, keyfield_name, item_ptr)
HASH_REPLACE_PTR	(head, keyfield_name, item_ptr, replaced_item_ptr)
HASH_FIND_PTR	(head, key_ptr, item_ptr)
HASH_DEL	(head, item_ptr)
HASH_SORT	(head, cmp)
HASH_COUNT	(head)

```
=====
```

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[]``.

#### .General macros

```
[width="90%",cols="10m,30m",grid="none",options="header"]
```

```
=====
```

| macro                                                 | arguments                                                        |
|-------------------------------------------------------|------------------------------------------------------------------|
| HASH_ADD<br>item_ptr)                                 | (hh_name, head, keyfield_name, key_len,<br>item_ptr)             |
| HASH_ADD_BYHASHVALUE<br>hashv, item_ptr)              | (hh_name, head, keyfield_name, key_len,<br>hashv, item_ptr)      |
| HASH_ADD_KEYPTR<br>item_ptr)                          | (hh_name, head, key_ptr, key_len,<br>item_ptr)                   |
| HASH_ADD_KEYPTR_BYHASHVALUE<br>item_ptr)              | (hh_name, head, key_ptr, key_len, hashv,<br>item_ptr)            |
| HASH_ADD_INORDER<br>item_ptr, cmp)                    | (hh_name, head, keyfield_name, key_len,<br>item_ptr, cmp)        |
| HASH_ADD_BYHASHVALUE_INORDER<br>hashv, item_ptr, cmp) | (hh_name, head, keyfield_name, key_len,<br>hashv, item_ptr, cmp) |
| HASH_ADD_KEYPTR_INORDER<br>item_ptr, cmp)             | (hh_name, head, key_ptr, key_len,<br>item_ptr, cmp)              |
| HASH_ADD_KEYPTR_BYHASHVALUE_INORDER                   | (hh_name, head, key_ptr, key_len, hashv,                         |

```
=====
```

```

item_ptr, cmp)
|HASH_REPLACE                | (hh_name, head, keyfield_name, key_len,
item_ptr, replaced_item_ptr)
|HASH_REPLACE_BYHASHVALUE    | (hh_name, head, keyfield_name, key_len,
hashv, item_ptr, replaced_item_ptr)
|HASH_REPLACE_INORDER        | (hh_name, head, keyfield_name, key_len,
item_ptr, replaced_item_ptr, cmp)
|HASH_REPLACE_BYHASHVALUE_INORDER | (hh_name, head, keyfield_name, key_len,
hashv, item_ptr, replaced_item_ptr, cmp)
|HASH_FIND                    | (hh_name, head, key_ptr, key_len,
item_ptr)
|HASH_FIND_BYHASHVALUE        | (hh_name, head, key_ptr, key_len, hashv,
item_ptr)
|HASH_DELETE                  | (hh_name, head, item_ptr)
|HASH_VALUE                   | (key_ptr, key_len, hashv)
|HASH_SRT                     | (hh_name, head, cmp)
|HASH_CNT                     | (hh_name, head)
|HASH_CLEAR                   | (hh_name, head)
|HASH_SELECT                  | (dst_hh_name, dst_head, src_hh_name,
src_head, condition)
|HASH_ITER                    | (hh_name, head, item_ptr, tmp_item_ptr)
|HASH_OVERHEAD                | (hh_name, head)
|=====

```

[NOTE]

`HASH\_ADD\_KEYPTR` is used when the structure contains a pointer to the key, rather than the key itself.

The `HASH\_VALUE` and `...\_BYHASHVALUE` macros are a performance mechanism mainly for the special case of having different structures, in different hash tables, having identical keys. It allows the hash value to be obtained once and then passed in to the `...\_BYHASHVALUE` macros, saving the expense of re-computing the hash value.

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 <<validc,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 <<multifield\_note,this note>>.)

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.

hashv::

the hash value of the provided key. This is an input parameter for the `...\_BYHASHVALUE` macros, and an output parameter for `HASH\_VALUE`. Reusing a cached hash value can be a performance optimization if you're going to do repeated lookups for the same key.

item\_ptr::

pointer to the structure being added, deleted, replaced, or looked up, or the current pointer during iteration. This is an input parameter for the ``HASH_ADD``, ``HASH_DELETE``, and ``HASH_REPLACE`` macros, and an output parameter for ``HASH_FIND`` and ``HASH_ITER``. (When using ``HASH_ITER`` to iterate, ``tmp_item_ptr`` is another variable of the same type as ``item_ptr``, used internally).

`replaced_item_ptr::`  
used in ``HASH_REPLACE`` macros. This is an output parameter that is set to point to the replaced item (if no item is replaced it is set to NULL).

`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``).

`condition::`  
a function or macro which accepts a single argument (a void pointer to a structure, which needs to be cast to the appropriate structure type). The function or macro should evaluate to a non-zero value if the structure should be "selected" for addition to the destination hash.

```
// vim: set tw=80 wm=2 syntax=asciidoc:
```