libHX 2.6
Documentation

March 28, 2009

Contents

1 Introduction

libHX collects many useful day-to-day functions, intended to reduce the amount of otherwise repeatedly open-coded instructions.

2 Overview

- Red-black binary tree with key-value pair extension (section 10)
  Originally created to provide a data structure like Perl's associative arrays. Uses an rbtree as underlying engine for somewhat quick insertion and deletion with a small memory footprint for ordered traversal. (Hashes would require gathering all keys first and sorting them.)

- Deques (section 11)
  Double-ended queues, implemented as a doubly-linked list with sentinels, are suitable for both providing stack and queue functionality.

- Inline doubly-linked list, uncounted and counted (sections 12 and 13)
  Light-weight linked lists as used in the Linux kernel.

- Common string operations (section 14)
  basename, chomp, dirname, getl(ine), split, strlcat/strlcpy, strlower/-upper, str*trim, strsep, etc.

- Memory containers, auto-sizing string operations (section 15)
  Scripting-like invocation for string handling — automatically doing (re)allocations as needed.

- String formatter (section 16)
  HXfmt is a small template system for by-name variable expansion. It can be used to substitute placeholders in format strings supplied by the user by appropriate expanded values defined by the program.

- Directory creation, traversal, removal, and file copying (sections 17, 18 and 19)

- Option parsing (section 20)
  Table-/callback-based option parser that works similar to Perl's `Getopt::Long` — no open-coding but a single "atomic" invocation.

- Shell-style config parser (section 21)
  Configuration file reader for Shell-style "configuration" files with key-value pairs, as usually foudn in `/etc/sysconfig`.

- Random number gathering (section 22)
  Convenient wrapper that uses kernel-provided RNG devices when available.

- External process invocation (section 23)
  Setting up pipes for the standard file descriptors for sending/capturing data to/from a program.

- *a bit more beyond that ... Miscellaneous*

3 Resources

As of this writing, the repository is located at

- `git://libhx.git.sf.net/gitroot/libhx` — clone URL
- `http://libhx.git.sf.net/` — gitweb interface
- `http://libhx.sf.net/` — home page (and link to tarballs)
- `http://freshmeat.net/projects/libHX/` — Freshmeat page (useful for subscription, i. e. automatic notification of releases)

4 Installation

libHX uses GNU autotools as a build environment, which means that all you have to run as a end-user is the `configure` with any options that you need, plus the usual `make` and `make install` as desired.

Pay attention to bi-arch Linux distributions where you most likely need to specify "lib64" instead of "lib":

```
$ ./configure --libdir='${prefix}/lib64'
```

Other libdir naming besides "`lib`" and "`lib64`" may be in use by different distributions, but are rarely seen (like "`lib32`").

4.1 Requirements

- GNU C Compiler 3.3.5 or newer. Other compilers (non-GCC) have not been tested in months — use at your own risk.
- approximately 80 KB of disk space on Linux for the shared library and header files; somewhat more for *BSD.

A C++ compiler is only needed if you want to build the test programs that come with libHX, in C++ mode. This is done by default so it may occur that you get prompted for a C++ compiler, but it is not strictly required, as this is just for the test programs.

- No external libraries are needed for compilation of libHX. Helper files, like `libxml_helper.h`, may reference their include files, but they are not used during compilation.

5 Portability notice

libHX runs on contemporary versions of Linux, Solaris and the three BSD distributions. It might even work on Microsoft Windows, but this is not tested very often, if at all. Overly old systems, especially Unices, are not within focus. While AIX 5.3 might still classify as contemporary, strangelets like "Ultrix" or "Dynix" you can find in the autotools-related file `config.guess` are some that are definitely not.

Furthermore, a compiler that understands the C99 or GNU89 standard is required. The integer type "int" should at best have 32 bits at least. There is no ultra-portable version as of this writing, but feel free to start one akin to the "p" variants of OpenBSD software such as OpenSSH.

6 History

The origins of libHX trace back, even crossing a language boundary, to when the author started on using Perl in 1999. Some tasks were just too damn useful to be open-coded every time. Two such examples are what is these days known as `HX_basename` and `HX_mkdir`. The name does not relate to anyone's initials; it is a result of a truncation of the author's nick used years ago.

Around the beginning of 2003, the author also started on the C programming language and soon the small library was converted from Perl to C. The libHX library as of today is the result of working with C ever since, and naturally grew from there to support whatever the author was in need of.

The "correct" name for libHX is with an uppercase "H" and uppercase "X", and the same is used for filenames, such as "libHX.so"[1].

[1]Software projects may choose to entirely lowercase the project name for use in filenames, such as the Linux kernel which is released as `linux-${`*version*`}.tar.bz2`, or the project may choose to keep the name for filenames, like Mesa and SDL do. libHX is of the latter.

Part I
General

Many functions are prefixed with "HX_" or "HXsubsys_", as are structures (sometimes without underscores, be sure to check the syntax and names), to avoid name clashes with possibly existing files. Functions that are not tied to a specific data structure such as most of the string functions (see chapter 14) use the subsystem-less prefix, "HX_". Functions from a clearly-defined subsystem, such as ARBtree or deque, augment the base prefix by a suffix, forming e. g. "HXbtree_".

7 Type-checking casts

The C++ language provides so-called "new-style casts", referring to the four template-looking invocations static_cast<>, const_cast<>, reinterpret_cast<> and dynamic_cast<>. No such blessing was given to the C language, but still, even using macros that expand to the olde cast make it much easier to find casts in source code and annotate why something was casted, which is already an improvement. — Actually, it *is* possible to do a some type checking, using some GCC extensions, which augments these macros from their documentary nature to an actual safety measure.

7.1 reinterpret_cast

reinterpret_cast() maps directly to the old-style typecast, (type)(expr), and causes the bit pattern for the expr rvalue to be "reinterpreted" as a new type. You will notice that "reinterpret" is the longest of all the *_cast names, and can easily cause your line to grow to 80 columns (the good maximum in many style guides). As a side effect, it is a good indicator that something potentially dangerous might be going on, for example converting intergers from/to pointer.

```
#include <libHX/defs.h>

int i;
/* Tree with numeric keys */
tree = HXbtree_init(HXBT_ICMP);
for (i = 0; i < 6; ++i)
        HXbtree_add(tree, reinterpret_cast(void *,
                    static_cast(long, i)), my_data);
```

7.2 signed_cast

This tag is for annotating that the cast was solely done to change the signedness of pointers to char — and only those. No integers etc. The intention is to facilitate working with libraries that use unsigned char * pointers, such as libcrypto and libssl (from the OpenSSL project) or libxml2, for example. See table 1 for the allowed conversions. C++ does *not* actually have a signed_cast<>, and one would have to use reinterpret_cast<> to do the conversion, because static_cast<> does not allow conversion from const char * to const unsigned char *, for example. (libHX's static_cast() would also throw at least a compiler warning about the different signedness.) libHX does provide a signed_cast<> for C++ though. This is where signed_cast comes in.

From \ To	c*	sc*	uc*	Cc*	Csc*	Cuc*
`char *`	✓	✓	✓	✓	✓	✓
`signed char *`	✓	✓	✓	✓	✓	✓
`unsigned char *`	✓	✓	✓	✓	✓	✓
`const char *`	–	–	–	✓	✓	✓
`const signed char *`	–	–	–	✓	✓	✓
`const unsigned char *`	–	–	–	✓	✓	✓

Table 1: Accepted conversions for `signed_cast()`

7.3 `static_cast`

Just like C++'s `static_cast<>`, libHX's `static_cast()` verifies that `expr` can be implicitly converted to the new type (by a simple `b = a`). Such is mainly useful for forcing a specific type, as is needed in varargs functions such as `printf`, and where the conversion actually incurs other side effects, such as truncation or promotion:

```
/* Convert to a type printf knows about */
uint64_t x = something;
printf("%llu\n", static_cast(unsigned long long, x));
```

Because there is no format specifier for `uint64_t` for `printf`, a conversion to an accepted type is necessary to not cause undefined behavior. The author has seen code that did, for example, `printf("%u")` on a "long", which only works on architectures where `sizeof(unsigned int)` happens to equal `sizeof(unsigned long)`, such as x86_32. On x86_64, an `unsigned long` is usually twice as big as an `unsigned int`, so that 8 bytes are pushed onto the stack, but `printf` only unshifts 4 bytes because the developer used "`%u`", leading to misreading the next variable on the stack.

```
/* Force promotion */
double a_quarter = static_cast(double, 1) / 4;
```

Were "1" not promoted to double, the result in `q` would be zero because 1/4 is just an integer division, yielding zero. By making one of the operands a floating-point quantity, the compiler will instruct the FPU to compute the result. Of course, one could have also written "`1.0`" instead of `static_cast(double, 1)`, but this is left for the programmer to decide which style s/he prefers.

```
/* Force truncation before invoking second sqrt */
double f = sqrt(static_cast(int, 10 * sqrt(3.0 / 4)));
```

And here, the conversion from `double` to `int` incurs a (wanted) truncation of the decimal fraction, that is, rounding down for positive numbers, and rounding up for negative numbers.

7.3.1 Allowed conversions

- **Numbers**
  Conversion between numeric types, such as `char`, `short`, `int`, `long`, `long long`, `intN_t`, both their signed and unsigned variants, `float` and `double`.

- **Generic Pointer**
  Conversion from `type *` to and from `void *`. (Where `type` may very well be a type with further indirection.)

- **Generic Pointer (const)**
  Conversion from `const type *` to and from `const void *`.

7.4 `const_cast`

`const_cast` allows to add or remove "const" qualifiers from the type a pointer is pointing to. Due to technical limitations, it could not be implemented to support arbitrary indirection. Instead, `const_cast` comes in three variants, to be used for indirection levels of 1 to 3:

- **`const_cast1(type *,`** `expr)` with **`typeof(expr) = type *`**. (Similarly for any combinations of const.)
- **`const_cast2(type **,`** `expr)` `with` **`typeof(expr) = type **`** (and all combinations of const in all possible locations).
- **`const_cast3(type ***,`** `expr)` `with` **`typeof(expr) = type ***`** (and all combinations...).

As indirection levels above 3 are really unlikely, having only these three type-checking cast macros was deemed sufficient. The only place where libHX even uses a level-3 indirection is in the option parser.

`int **`	`int *const *`
`const int **`	`const int *const *`

Table 2: Accepted expr/target types for `const_cast2`; example for the "int" type
Conversion is permitted when expression and target type are from the table.

8 Macros

All macros in this section are available through `#include <libHX/defs.h>`.

8.1 Preprocessor

```
#define HX_STRINGIFY(s)
```

Transforms the expansion of the argument `s` into a C string.

8.2 Locators

```
long offsetof(type, member);
output_type *containerof(input_type *ptr, output_type, member);
```

In case `offsetof` and `containerof` have not already defined by inclusion of another header file, libHX's defs.h will define these accessors. `offsetof` is defined in `stddef.h` (for C) or `cstddef` (C++), but inclusion of these is not necessary if you have included `defs.h`. defs.h will use GCC's `__builtin_offsetof` if available, which does some extra sanity checks in C++ mode.

`offsetof` calculates the offset of the specified member in the type, which needs to be a struct or union.

`containerof` will return a pointer to the struct in which ptr is contained as the given member.

```
struct foo {
        int bar;
        int baz;
};

static void test(int *ptr)
{
        struct foo *self = containerof(baz, struct foo, baz);
}
```

8.3 Array size

```
size_t ARRAY_SIZE(type array[]); /* implemented as a macro */
```

Returns the number of elements in `array`. This only works with true arrays (`type[]`), and will not output a meaningful value when used with a pointer-to-element (`type *`), which is often used for array access too.

8.4 Compile-time build checks

```
void BUILD_BUG_ON(bool condition); /* implemented as a macro */
```

Causes the compiler to fail when `condition` evaluates to true. If not implemented for a compiler, it will be a no-op.

8.5 UNIX file modes

```
#define S_IRUGO (S_IRUSR | S_IRGRP | S_IROTH)
#define S_IWUGO (S_IWUSR | S_IWGRP | S_IWOTH)
#define S_IXUGO (S_IXUSR | S_IXGRP | S_IXOTH)
#define S_IRWXUGO (S_IRUGO | S_IWUGO | S_IXUGO)
```

The defines make it vastly easier to specify permissions for large group of users. For example, if one wanted to create a file with the permissions `rw-r--r--` (ignoring the umask in this description), `S_IRUSR | S_IWUSR` can now be used instead of the longer `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH`.

9 Miscellaneous functions

```
#include <libHX/misc.h>

int HX_ffs(unsigned long z);
int HX_fls(unsigned long z);
void HX_hexdump(FILE *fp, const void *ptr, unsigned int len);
int HX_time_compare(const struct stat *a, const struct stat *b, int mode);
void HX_zvecfree(char **);
int HX_zveclen(const char *const *);
```

HX_ffs Finds the first (lowest-significant) bit in a value and returns its position, or -1 to indicate failure.

HX_fls Finds the last (most-significant) bit in a value and returns its position, or -1 to indicate failure.

HX_hexdump Outputs a nice pretty-printed hex and ASCII dump to the filedescriptor `fp`. `ptr` is the memory area, of which `len` bytes will be dumped.

HX_time_compare Compares the timestamps from two `struct stat`s. `mode` indicates which field is compared, which can either be `'a'` for the access time, `'c'` for the inode change time, `'m'` for the modification time, or `'o'` for the creation time (where available). Returns a negative number if the time in `a` is less than `b`, zero when they are equal, or a positive number greater than zero if `a` is greater than `b`.

HX_zvecfree Frees the supplied Z-vector array. (Frees all array elements from the first element to (excluding) the first `NULL` element.)

HX_zveclen Counts the number of array elements until the first `NULL` array element is seen, and returns this number.

Part II
Data structures

10 ARBtree

ARBtree stands for "associative red-black tree" and implements a structure that can be used in a multitude of scenarios, ranging from a poor man's sorting mechanism, a sparse bitmap, over to a key-value map. Using a red-black tree as ADT allows for somewhat quick insertion and deletion and small memory footprint for ordered traversals. (Using a hash would have meant to gather all keys first and sort them instead.)

10.1 Structural definition

```
#include <libHX/arbtree.h>

struct HXbtree {
        void *uptr;
        /* public readonly: */
        unsigned int items;
        /* Unlisted members are "private". */
};

struct HXbtree_node {
        /* public readonly/readwrite (see notes in section 10.2): */
        union {
                void *key;
                const char *const skey;
        };
        union {
                void *data;
                char *sdata;
        };
        /* Unlisted members are "private". */
};
```

Member descriptions for `struct HXbtree`:

uptr A custom user-supplied pointer, usually to aid associating the tree with something else when needed

items The number of items in the tree. This field tracks the number of items in the tree and is used to report the number of elements to the user, and is updated whenever an element is inserted or removed from the tree. This is faster than deep-walking the tree everytime. The field must not be changed by user.

Member descriptions for `struct HXbtree_node`:

key The key for this node. It uniquely identifies the object (node), depending on the implementation of the chosen key comparison function.

data Any associated data, in case of a `HXBT_MAP` tree.

10.2 Storage models

10.2.1 The (entire) data is the key

To begin the introduction, in this model, the tree serves to store "basic" elements, e. g. integers or strings.

without doing a deep comparison, e. g. integers or strings or any quantity that could be considered atomic. While C programmers may not see a string, essentially an array of char, as atomic, the special-casing of strings in HXbtree makes them rather undividable.

```
struct HXbtree *b = HXbtree_init(HXBT_ICMP);
HXbtree_add(b, reinterpret_cast(void *, static_cast(long, rand())));
```

Not necessarily pretty, but then again a tree was not originally designed to support non-pointer data. As it stands, it works on many platforms as shown.

Figure 1: Code sample for storing integers

```
struct HXbtree *b = HXbtree_init(HXBT_SCMP);
HXbtree_add(b, "Hello");
HXbtree_add(b, "World");
```

Figure 2: Code sample for storing strings (pointers)

```
struct HXbtree *b = HXbtree_init(HXBT_CDATA | HXBT_SCMP);
char c[10];
unsigned int i;

for (i = 0; i < 15; ++i) {
        snprintf(c, sizeof(c), "%u", i);
        HXbtree_add(b, c);
}
```

Figure 3: Storing strings with automatic duplication

Storing data this way effectively turns the data structure into a bitmap, as all the tree will return when you search for an element is the element itself, or nothing.

String duplication is needed if the data provided by the pointer during the call to `HXbtree_add` may change during the lifetime of the tree, as shown in figure 3. Because the buffer will be overwritten in every iteration, having the tree only store a pointer would infer problems because the tree would effectively be externally changed without updating the tree metadata. This is why `HXBT_CDATA` needs to be specified so that HXbtree will copy the string to a new memory block first before putting the (then new) pointer into the tree. Only if you can be sure that the object will not be modified can you omit `HXBT_CDATA`. What it boils down to is that the primary key (as SQL people call it) may not be modified directly, because this would invalidate the tree structure and metadata. The next storage models will elaborate on this.

10.2.2 Data with embedded key

Since it is possible to store any kind of data, non-atomic types such as structs are also eligible. The only component needed is a comparison function that can tell the order or equivalence

of two structures. For atomic, integer types, this function is internally provided by and for HXbtree automatically as is for strings; `HXBT_ICMP` and `HXBT_SCMP` select these, respectively. A very simplistic approach could be:

```
struct point_with_data {
        int x, y;
        struct timeval timestamp;
        unsigned int flags;
};
```

Figure 4: Storing non-atomic data

10.2.3 Detached key

It is possible to detach the key from its data struct. The reason to want to do is to associate multiple keys with the same data while being memory efficient and to prevent against change anomalies. That is, if multiple keys are supposed to point to the same data even if the data will be changed (thus all keys associating to it will return the new data).

```
struct point {
        int x, y;
        struct timeval timestamp;
        unsigned int flags;
};
```

to

```
struct point {
        int x, y;
};
struct point_data {
        struct timeval timestamp;
        unsigned int flags;
};
```

When you make use of the feature of associating the same data with multiple keys, `HXBT_CDATA` cannot be used, and it must be ensured that the data is valid throughout the lifetime of the nodes in the tree that refer to it.

10.3 Constructors

10.3.1 HXbtree_init

```
struct HXbtree *HXbtree_init(unsigned int flags, ...);
```

`HXbtree_init` initializes a new binary tree. The bitfield `flags` can contain the following options:

`HXBT_MAP` This changes the tree semantics to behave like an associative array (key-value pairs). One of `HXBT_CMPFN`, `HXBT_SCMP` or `HXBT_ICMP` must be specified when `HXBT_MAP` is selected.

HXBT_CKEY Duplicate the key used for `HXbtree_add` before inserting it into the tree. This must only be used when the object to be added is a C-style string.

HXBT_CDATA Duplicate the data used for `HXbtree_add` before inserting it into the tree. This must only be used when the object to be added is a C-style string.

HXBT_CMPFN Selects the function that defines the order of elements in the tree. This pulls one argument from the varargs stack, which must be a function pointer of type `int (*)(const void *, const void *, size_t)`. It will acts as a key comparison function for sorting, searching and traversal. `HXBT_CMPFN` takes precedence over `HXBT_SCMP`, `HXBT_ICMP` in regard to the selection of a comparison function.

HXBT_SCMP The relation between keys is given by their string sorting order. It is an abbreviation for using `HXBT_CMPFN` with `strcmp`.

HXBT_ICMP Use a by-value comparison for the keys ("ICMP" = "integer compare"). This is useful if you plan on using numbers as keys, which must not be dereferenced by HXbtree.

HXBT_CID This flag instructs the traverser[2] to copy not the pointer value, but the string pointed to by it, to its internal state, for re-lookup in case the tree changes during traversal.
Details: When the tree is changed during traversal, whereby change here does not mean adding or deleting a node, but causing a rebalance in the red-black tree as part of an addition or deletion, the traverser state, which keeps a path to the currently-visited node, may become invalid. The traverser must therefore re-establish this path by walking down to the last known node before it can find the node's successor. If the node has been deleted and was re-inserted, the address for the key string may have changed though the key itself has not. In this case, the traverser needs to compare by string key, not value. The `HXBT_CID` flag can unfortunately not be implicit for `HXBT_SCMP`, since one can also specify a comparison function of which we do not know its characteristics, using `HXBT_CMPFN`. Also, if you can assure that no tree modifications happen during traversal, not specifying `HXBT_CID` will be faster.

10.3.2 HXbtree_init2

```
struct HXbtree *HXbtree_init2(unsigned int flags,
    int (*k_compare)(const void *, const void *, size_t),
    void *(*k_clone)(const void *, size_t), void (*k_free)(const void *),
    void *(*d_clone)(const void *, size_t), void (*d_free)(const void *));
```

`HXbtree_init2` is a newer variant that extends `HXbtree_init` by allowing to specify the size of the key and data struct, so that HXbtree can duplicate the key and/or data if `HXBT_CKEY` and `HXBT_CDATA` is specified. By default, it will be using `HX_memdup` and `free`(3). Previously, with `HXbtree_init`, this had to be done by the user as it only supported duplicating strings. The function pointers `k_clone`, `d_clone`, `k_free` and `d_free` may be used to override the defaults if they are non-`NULL`. The following flags are supported:

HXBT_MAP (as described above)

HXBT_CKEY (as described above)

HXBT_CDATA (as described above)

[2]Most people would call it iterator, but traverser stuck with the code.

HXBT_SKEY Select string operations `HX_strdup` and `free` for the key cloning and release by default. The functions can still be overridden by the parameters in the argument list.

HXBT_SDATA Select string operations for data cloning and release by default. The functions can still be overridden by the parameters in the argument list.

HXBT_CID (as described above)

Any other flags, such as `HXBT_SCMP` and `HXBT_ICMP`, are not allowed.

10.4 Destruction

```
void HXbtree_free(struct HXbtree *tree);
```

The `HXbtree_free` function will delete the tree and any associated objects. If `HXBT_MAP | HXBT_CKEY` had been specified during the constructing call to `HXbtree_init`, all keys are freed too, since they were initially duplicated and are owned by the tree. Conversely, if `HXBT_CDATA` was specified, `free` is called on all data pointers. It is therefore important that, by the time `HXbtree_del` is called, the tree only contains nodes with key and data pointers that it actually owns.

10.5 Adding nodes

```
struct HXbtree_node *HXbtree_add(struct HXbtree *tree, const void *data);
struct HXbtree_node *HXbtree_add(struct HXbtree *tree,
        const void *key, const void *data);
```

`HXbtree_add` adds a new node to the tree using the given key and/or data. When an object is in the tree, only parts may be modified that would not change the order of elements. If you need to change the key (which may be packed into `data`, see example in section 10.10.4), you will have to delete the object from the tree and re-insert it.

On success, a pointer to the newly added node is returned if the insertion was successful, or `NULL` otherwise. On error, `errno` will be set appropriately.

10.6 Search

```
struct HXbtree_node *HXbtree_find(struct HXbtree *tree, const void *key);
void *HXbtree_get(struct HXbtree *tree, const void *key);
```

`HXbtree_find` will find the node for the given key. The key can be read from the node using `node->key` or node->skey (convenience alias for `key`, but with a type of `const char *`), and the data by using `node->data` or `node->sdata`. `HXbtree_get` will directly return `node->data` instead of the node itself. Since `HXbtree_get` may legitimately return `NULL` if `NULL` was stored in the tree as the data for a given key, only `errno` will really tell whether the node was found or not; in the latter case, `errno` is set to `ENOENT`.

10.7 Deletion

```
void *HXbtree_del(struct HXbtree *tree, const void *key);
```

Delete the node given by `key` from the tree and return the associated data if the tree does not own the data (because if it does own the data, it has to free it, at which point it cannot be returned), or `NULL` otherwise.

10.8 Traversal

```
void *HXbtrav_init(struct HXbtree *tree);
struct HXbtree_node *HXbtraverse(void *trav);
void HXbtrav_free(void *trav);
```

HXbtrav_init Initializes a B-tree traverser on the given tree. Traversal starts at the left-most node.

HXbtraverse Returns the next inorder element. The tree may be modified during traversal and the traverser will relookup the nodes in the tree to restore its state. However, for repickup to work, the HXBT_CID flag to work with non-integer keys.

HXbtrav_free Frees the storage that the traverser used.

10.9 Limitations

The implementation has a theoretical minimum on the maximum number of nodes, $2^{24} = 16{,}777{,}216$. A worst-case tree with this many elements already has a height of 48 (BT_MAXDEP). The larger the height is that arbtree is supposed to handle, the more memory (linear increase) it needs. All functions that build or keep a path reserve memory for BT_MAXDEP nodes; on x86_64 this is 9 bytes per ⟨node, direction⟩ pair, amounting to 432 bytes for path tracking alone. It may not sound like a lot to many, but given that kernel people can limit their stack usage to 4096 bytes is impressive alone[3].

10.10 Examples

10.10.1 Case-insensitive ordering

This one is easy:

```
b = HXbtree_init(HXBT_MAP | HXBT_CMPFN, strcasecmp);
```

10.10.2 Reverse sorting order ($Z \rightarrow A$)

Any function that behaves like strcmp can be used. It merely has to return negative when $a < b$, zero on $a = b$, and positive non-zero when $a > b$.

```
static int strcmp_rev(const void *a, const void *b)
{
        return strcmp(b, a);
}

static int strcmp_rev3(const void *a, const void *b, size_t z)
{
        /* z is provided for cases when things are raw memory blocks. */
        return strcmp(b, a);
}

b = HXbtree_init(HXBT_MAP | HXBT_CMPFN, strcmp_rev);
b = HXbtree_init2(HXBT_MAP | HXBT_SKEY, strcmp3_rev, NULL, NULL, NULL, NULL);
```

[3]Not always of course. Linux kernels are often configured to use an 8K stack because some components still use a lot of stack space, but eve 8K is still damn good.

10.10.3 Data-only tree

It is unbeknown if this usage is really that often used. I mean, you can abuse a self-balancing tree in a number of ways, for example to sort elements.

```
b = HXbtree_init(HXBT_SCMP);
HXbtree_add(b, "cheese");
HXbtree_add(b, "cake");
HXbtree_add(b, "fruit");
HXbtree_add(b, "cake");
```

Now you have the elements in the tree, and traversing it will return them in ordered fashion. It is my gut feeling though, that inserting the elements into a HXdeque instead, converting that to a zvec and then running `qsort` is faster.

Another application that comes to mind is a very sparse bitmap:

```
b = HXbtree_init(HXBT_ICMP);
/* Grab six random numbers from 1..49 */
for (i = 0; i < 6; ++i)
        HXbtree_add(b, (const void *)HX_irand(1, 50));
```

10.10.4 Non-associative tree

Keys can be stored together with their actual data, especially when they are not just composed of a single integer or string, bundling them with their data may make sense:

```
struct package {
        char *name;
        unsigned int major_version;
        unsigned int minor_version;
        char notes[64];
};

static int package_cmp(const void *a, const void *b)
{
        const struct package *p = a, *q = b;
        int ret;
        ret = strcmp(p->name, q->name);
        if (ret != 0)
                return ret;
        ret = p->major_version - q->major_version;
        if (ret != 0)
                return ret;
        ret = p->minor_version - q->minor_version;
        if (ret != 0)
                return ret;
        return 0;
}

HXbtree_init(HXBT_CMPFN, myobject_cmp);
```

In this case, the key consists of ⟨package name, major version, minor version⟩.

11 Doubly-linked list

HXdeque is a data structure for a doubly-linked non-circular NULL-sentineled list. Despite being named a deque, which is short for double-ended queue, and which may be implemented using an array, HXdeque is in fact using a linked list to provide its deque functionality. Furthermore, a dedicated root structure and decidated node structures with indirect data referencing are used.

11.1 Structural definition

```
#include <libHX/deque.h>

struct HXdeque {
        struct HXdeque_node *first, *last;
        unsigned int items;
        void *ptr;
};

struct HXdeque_node {
        struct HXdeque_node *next, *prev;
        struct HXdeque *parent;
        void *ptr;
};
```

The ptr member in struct HXdeque provides room for an arbitrary custom user-supplied pointer. items will reflect the number of elements in the list, and must not be modified. first and last provide entrypoints to the list's ends.

ptr within struct HXdeque_node is the pointer to the user's data. It may be modified and used at will by the user. See example section .

11.2 Constructor, destructors

```
struct HXdeque *HXdeque_init(void);
void HXdeque_free(struct HXdeque *dq);
void HXdeque_genocide(struct HXdeque *dq);
void **HXdeque_to_vec(struct HXdeque *dq, unsigned int *num);
```

To allocate a new empty list, use HXdeque_init. HXdeque_free will free the list (including all nodes owned by the list), but not the data pointers.

HXdeque_genocide is a variant that will not only destroy the list, but also calls free() on all stored data pointers. This puts a number of restrictions on the characteristics of the list: all data pointers must have been obtained with malloc , calloc or realloc before, and no data pointer must exist twice in the list. The function is more efficient than an open-coded loop over all nodes calling HXdeque_del.

To convert a linked list to a NULL-terminated array, HXdeque_to_vec can be used. If num is not NULL, the number of elements excluding the NULL sentinel, is stored in *num.

11.3 Addition and removal

```
struct HXdeque_node *HXdeque_push(struct HXdeque *dq, void *ptr);
struct HXdeque_node *HXdeque_unshift(struct HXdeque *dq, void *ptr);
```

```
void *HXdeque_pop(struct HXdeque *dq);
void *HXdeque_shift(struct HXdeque *dq);
struct HXdeque *HXdeque_move(struct HXdeque_node *target,
                             struct HXdeque_node *node);
void *HXdeque_del(struct HXdeque_node *node);
```

`HXdeque_push` and `HXdeque_unshift` add the data item in a new node at the end ("push") or as the new first element ("unshift" as Perl calls it), respectively. The functions will return the new node on success, or `NULL` on failure and `errno` will be set. The node is owned by the list.

`HXdeque_pop` and `HXdeque_shift` remove the last ("pop") or first ("shift") node, respectively, and return the data pointer that was stored in the data.

`HXdeque_move` will unlink a node from its list, and reinsert it after the given target node, which may be in a different list.

Deleting a node is accomplished by calling `HXdeque_del` on it. The data pointer stored in the node is not freed, but returned.

11.4 Iteration

Iterating over a HXdeque linked list is done manually and without additional overhead of function calls:

```
const struct HXdeque_node *node;
for (node = dq->first; node != NULL; node = node->next)
        do_something(node->ptr);
```

11.5 Searching

```
struct HXdeque_node *HXdeque_find(struct HXdeque *dq, const void *ptr);
void *HXdeque_get(struct HXdeque *dq, void *ptr);
```

`HXdeque_find` searches for the node which contains `ptr`, and does so by beginning at the start of the list. If no node is found, `NULL` is returned. If a pointer is more than once in the list, any node may be returned.

`HXdeque_get` will further return the data pointer stored in the node — however, since that is just what the `ptr` argument is, the function practically only checks for existence of `ptr` in the list.

11.6 Examples

In this example, all usernames are obtained from NSS, and put into a list. `HX_strdup` is used, because `getpwent` will overwrite the buffer it uses to store its results. The list is then converted to an array, and the list is freed (because it is not need it anymore). `HXdeque_genocide` must not be used here, because it would free all the data pointers (strings here) that were just inserted into the list. Finally, the list is sorted using the well-known `qsort` function. Because `strcmp` takes two `const char *` arguments, but `qsort` mandates a function taking two `const void *`, a cast can be used to silence the compiler. This only works because we know that the array consists of a bunch of `char *` pointers, so `strcmp` will work.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libHX/defs.h>
#include <libHX/deque.h>
#include <libHX/string.h>
#include <pwd.h>

int main(void)
{
        struct HXdeque *dq = HXdeque_init();
        struct passwd *pw;
        unsigned int elem;
        char **users;

        setpwent();
        while ((pw = getpwent()) != NULL)
                HXdeque_push(dq, HX_strdup(pw->pw_name));
        endpwent();

        users = reinterpret_cast(char **, HXdeque_to_vec(dq, &elem));
        HXdeque_free(dq);

        qsort(users, elem, sizeof(*users), static_cast(void *, strcmp));
        return 0;
}
```

Figure 5: Example use of HXdeque to store and sort a list

12 Inline doubly-linked list

Classical linked-list implementations, such as HXdeque, either store the actual data within a node, or indirectly through a pointer, but the "inline doubly-linked list" instead does it reverse and has the list head within the data structure.

```
struct package_desc {
        char *package_name;
        int version;
};
struct classic_direct_node {
        struct classic_direct_node *next, *prev;
        struct package_desc direct_data;
};
struct classic_indirect_node {
        struct classic_indirect_node *next, *prev;
        void *indirect_data;
};
```

Figure 6: Classic linked-list implementations with direct/indirect data blocks.

```
struct package_desc {
        struct HXlist_head list;
        char *package_name;
        int version;
};
```

Figure 7: List head (next,prev pointers) inlined into the data block

At first glance, an inline list does not look much different from `struct classic_direct_data`, it is mostly a viewpoint decision which struct is in the foreground.

12.1 Synopsis

```
#include <libHX/list.h>

struct HXlist_head {
        /* All fields considered private */
}

HXLIST_HEAD_INIT(name);
HXLIST_HEAD(name);
void HXlist_init(struct HXlist_head *list);
void HXlist_add(struct HXlist_head *list, struct HXlist_head *elem);
void HXlist_add_tail(struct HXlist_head *list, struct HXlist_head *elem);
void HXlist_del(struct HXlist_head *element);
```

HXLIST_HEAD_INIT This macro expands to the static initializer for a list head.

HXLIST_HEAD This macro expands to the definition of a list head (i. e. `struct HXlist_head name = HXLIST_HEAD_INIT;`)

HXlist_init Initializes the list head. This function is generally used when the list head is on the heap where the static initializer cannot be used.

HXlist_add Adds `elem` to the front of the list.

HXlist_add_tail Adds `elem` to the end of the list.

HXlist_del Deletes the given element from the list.

12.2 When to use HXdeque/HXlist

The choice whether to use HXdeque or HXlist/HXclist depends on whether one wants the list head handling on the developer or on the library. Especially for "atomic" and "small" data, it might be easier to just let HXdeque do the management. Compare the following two code examples to store strings:

```
int main(int argc, const char **argv)
{
        struct HXdeque *dq = HXdeque_init();
        while (--argc)
                 HXdeque_push(dq, ++argv);
        return 0;
}
```

Figure 8: Storing strings in a HXdeque

```
struct element {
        struct HXlist_head list;
        char *data;
};

int main(int main, const char **argv)
{
        HXLIST_HEAD(lh);
        while (--argc) {
                struct element *e = malloc(sizeof(*e));
                e->data = *++argv;
                HXlist_init(&e->list);
                HXlist_add_tail(&e->list);
        }
        return 0;
}
```

Figure 9: Storing strings in a HXlist

These examples assume that `argv` is persistent, which, for the sample, is true.

```
struct point p = {15, 30};
HXdeque_push(dq, &p);
HXdeque_push(dq, &p);
```

Figure 10: Data can be added multiple times in a HXdeque without ill effects

With HXlist, one needs to have a struct with a HXlist_head in it, and if one does not already have such a struct —e. g. by means of wanting to store more than just one value — one will need to create it first, as shown, and this may lead to an expansion of code.

This however does not mean that HXlist is the better solution over HXdeque for data already available in a struct. As each struct has a list_head that is unique to the node, it is not possible to share this data. Trying to add a HXlist_head to another list is not going to end well, while HXdeque has no problem with this as list heads are detached from the actual data in HXdeque.

To support this, an extra allocation is needed on the other hand. In a HXlist, to store n elements of compound data (e. g. `struct point`), n allocations are needed, assuming the list head is a stack object, and the points are not. HXdeque will need at least $2n + 1$ allocations, n for the nodes, n for the points and another for the head.

13 Counted inline doubly-linked list

clist is the inline doubly-linked list from chapter 12, extended by a counter to retrieve the number of elements in the list in $\mathcal{O}(1)$ time. This is also why all operations always require the list head. For traversal of clists, use the corresponding HXlist macro.

13.1 Synopsis

```
#include <libHX/clist.h>

struct HXclist_head {
        /* public readonly: */
        unsigned int items;
        /* Undocumented fields are considered "private" */
};

HXCLIST_HEAD_INIT(name);
HXCLIST_HEAD(name);
void HXclist_init(struct HXclist_head *head);
void HXclist_unshift(struct HXclist_head *head, struct HXlist_head *new_node);
void HXclist_push(struct HXclist_head *head, struct HXlist_head *new_node);
type HXclist_pop(struct HXclist_head *head, type, member);
type HXclist_shift(struct HXclist_head *head, type, member);
void HXclist_del(struct HXclist_head *head, struct HXlist_chead *node);
```

HXCLIST_HEAD_INIT Macro that expands to the static initializer for a clist.

HXCLIST_HEAD Macro that expands to the definition of a clist head, with initialization.

HXclist_init Initializes a clist. This function is generally used when the head has been allocated from the heap.

HXclist_unshift Adds the node to the front of the list.

HXclist_push Adds the node to the end of the list.

HXclist_pop Removes the last node in the list and returns it.

HXclist_shift Removes the first node in the list and returns it.

HXclist_del Deletes the node from the list.

The list count in the clist head is updated whenever a modification is done on the clist through these functions.

Part III
Strings and memory

14 String operations

Some string functions are merely present in libHX because they are otherwise unportable; some are only in the C libraries of the BSDs, some only in GNU libc.

14.1 Locating chars

```
#include <libHX/string.h>

char *HX_strbchr(const char *start, const char *now, char delimiter);
size_t HX_strrcspn(const char *s, const char *reject);
```

HX_strbchr Searches the character specified by `delimiter` in the range from `now` to `start`. It works like `strrchr`(3), but begins at `now` rather than the end of the string.

HX_strrcspn Works like `strcspn`(3), but processes the string from end to start.

14.2 Extraction

```
#include <libHX/string.h>

char *HX_basename(const char *s);
char *HX_dirname(const char *s);
char *HX_strmid(const char *s, long offset, long length);
```

HX_basename Returns a pointer to the basename portion of the supplied path `s`. The result must not be freed. The input string must not have any trailing slashes[4].

HX_dirname Returns a pointer to a new string that contains the directory name portion (everything except basename). When done using the string, it must be freed to avoid memory leaks.

HX_strmid Extract a substring of `length` characters from `s`, beginning at `offset`. If `offset` is negative, counting beings from the end of the string; −1 is the last character (not the `'\0'` byte). If `length` is negative, it will leave out that many characters off the end. The function returns a pointer to a new string, and the user has to free it.

14.3 In-place transformations

```
#include <libHX/string.h>

char *HX_chomp(char *s);
size_t HX_strltrim(char *s);
char *HX_strlower(char *s);
```

[4]This was a "design" choice — stripping slashes at the library level would require an allocation, and having slashes does not seem to happen that often. Even if there are slashes in a string, the caller often has more knowledge about the exact string and can just replace them by NULs.

```
char *HX_strrev(char *s);
size_t HX_strrtrim(char *s);
char *HX_strupper(char *s);
```

HX_chomp Removes the characters '\r' and '\n' from the right edge of the string. Returns the original argument.

HX_strltrim Trim all whitespace (characters on which isspace(3) return true) on the left edge of the string. Returns the number of characters that were stripped.

HX_strlower Transforms all characters in the string s into lowercase using tolower(3). Returns the original argument.

HX_strrev Reverse the string in-place. Returns the original argument.

HX_strrtrim Trim all whitespace on the right edge of the string. Returns the number of characters that were stripped.

HX_strupper Transforms all characters in the string s into uppercase using toupper(3). Returns the original argument.

14.4 Tokenizing

```
#include <libHX/string.h>

char **HX_split(const char *s, const char *delimiters, size_t *fields, int max);
char **HX_split4(char *s, const char *delimiters, int *fields, int max);
int HX_split5(char *s, const char *delimiters, int max, char **stack);
char *HX_strsep(char **sp, const char *delimiters);
char *HX_strsep2(char **sp, const char *dstr);
```

HX_split Split the string s on any characters from the "delimiters" string. Both the substrings and the array holding the pointers to these substrings will be allocated as required; the original string is not modified. If max is larger than zero, produces no more than max fields. If fields is not NULL, the number of elements produced will be stored in *fields. The result is a NULL-terminated array of char *, and the user needs to free it when done with it, using HX_zvecfree or equivalent.

HX_split4 Split the string s in-place on any characters from the "delimiters" string. The array that will be holding the pointers to the substrings will be allocated and needs to be freed by the user, using free(3). The fields and max arguments work as with HX_split.

HX_split5 Split the string s in-place on any characters from the "delimiters" string. The array for the substring pointers must be provided by the user through the stack argument. max must be the number of elements in the array or less. The array will not be NULL-terminated[5]. The number of fields produced is returned.

HX_strsep Extract tokens from a string.
This implementation of strsep has been added since the function is non-standard (according to the manpage, conforms to BSD4.4 only) and may not be available on every operating system.

[5] An implementation may however decide to put NULL in the unassigned fields, but this is implementation and situation-specific. Do not rely on it.

This function extracts tokens, separated by one of the characters in `delimiters`. The string is modified in-place and thus must be writable. The delimiters in the string are then overwritten with `'\0'`, `*sp` is advanced to the character after the delimiter, and the original pointer is returned. After the final token, `strsep` will return `NULL`.

HX_strsep2 Like `HX_strsep`, but `dstr` is not an array of delimiting characters, but an entire substring that acts as a delimiter.

14.5 Size-bounded string ops

```
#include <libHX/string.h>

char *HX_strlcat(char *dest, const char *src, size_t length);
char *HX_strlcpy(char *dest, const char *src, size_t length);
char *HX_strlncat(char *dest, const char *src, size_t dlen, size_t slen);
```

`HX_strlcat` and `HX_strlcpy` provide implementations of the BSD-originating `strlcat`(3) and `strlcpy`(3). `strlcat` and `strlcpy` are less error-prone variants for `strncat` and `strncpy` as they always take the length of the entire buffer specified by `dest`, instead of just the length that is to be written. The functions guarantee that the buffer is `'\0'`-terminated.

14.6 Allocation-related

```
#include <libHX/string.h>

void *HX_memdup(const void *ptr, size_t length);
char *HX_strdup(const char *str);
 char *HX_strclone(char **pa, const char *pb);

#ifdef __cplusplus
template<typename type> type HX_memdup(const void *ptr, size_t length);
#endif
```

HX_memdup Duplicates `length` bytes from the memory area pointed to by `ptr` and returns a pointer to the new memory block.

HX_strdup Duplicates the string. The function is equivalent to `strdup`, but the latter may not be available on all platforms.

HX_strclone Copies the string pointed to by `pb` into `*pa`. If `*pa` was not `NULL` by the time `HX_strclone` was called, the string is freed before a new one is allocated. The function returns `NULL` and sets `errno` to `EINVAL` if `pb` is `NULL` (this way it can be freed), or, if `malloc` fails, returns `NULL` and leaves `errno` at what `malloc` set it to.
The use of this function is deprecated, albeit no replacement is proposed.

14.7 Examples

14.7.1 Using HX_split5

`HX_split5`, where the "5" should be interpreted (with a bit of imagination and the knowledge of leetspeak) as an "S" for stack, as `HX_split5` is often used only with on-stack variables and where the field count of interest is fixed, as the example for parsing `/etc/passwd` shows:

```
#include <stdio.h>
#include <libHX/string.h>

char *field[8];
hxmc_t *line = NULL;

while (HX_getl(&line, fp) != NULL) {
        if (HX_split5(line, ":", ARRAY_SIZE(field), field) < 7) {
                fprintf(stderr, "That does not look like a valid line.\n");
                continue;
        }
        printf("Username: %s\n", field[0]);
}
```

14.7.2 Using HX_split4

Where the number of fields is not previously known and/or estimatable, but the string can be modified in place, one uses `HX_split4` as follows:

```
#include <errno.h>
#include <stdio.h>
#include <libHX/string.h>

while (HX_getl(&line, fp) != NULL) {
        char **field = HX_split4(line, ":", NULL, 0);
        if (field == NULL) {
                fprintf(stderr, "Badness! %s\n", strerror(errno));
                break;
        }
        printf("Username: %s\n", field[0]);
        free(field);
}
```

14.7.3 Using HX_split

Where the string is not modifiable in-place, one has to resort to using the full-fledged `HX_split` that allocates space for each substring.

```
#include <errno.h>
#include <stdio.h>
#include <libHX/string.h>

while (HX_getl(&line, fp) != NULL) {
        char **field = HX_split(line, ":", NULL, 0);
        if (field == NULL) {
                fprintf(stderr, "Badness. %s\n", strerror(errno));
                break;
        }
        printf("Username: %s\n", field[0]);
        /* Suppose "callme" needs the original string */
        callme(line);
```

```
        HX_zvecfree(field);
}
```

14.7.4 Using HX_strsep

`HX_strsep` provides for thread- and reentrant-safe tokenizing a string where strtok from the C standard would otherwise fail.

```
#include <stdio.h>
#include <libHX/string.h>

char line[] = "root:x:0:0:root:/root:/bin/bash";
char *wp, *p;

wp = line;
while ((p = HX_strsep(&wp, ":")) != NULL)
        printf("%s\n", p)
```

15 Memory containers

The HXmc series of functions provide scripting-like semantics for strings, especially automatically resizing the buffer on demand. They can also be used to store a binary block of data together with its length. (Hence the name: mc = memory container.)

The benefit of using the HXmc functions is that one does not have to meticulously watch buffer and string sizes anymore.

```
/* Step 1 */
    char buf[whatever was believed to be long enough] = "helloworld";
    if (strlen(buf) + strlen(".txt") < sizeof(buf))
            strcat(s, ".txt");

/* Step 2 */

    char buf[long_enough] = "helloworld";
    strlcat(s, ".txt", sizeof(buf));

/* Step 3 */

    hxmc_t *buf = HXmc_strinit("helloworld");
    HXmc_strcat(&s, ".txt");
```

Figure 11: Improvement of string safety over time

This makes it quite similar to the string operations (and append seems to be the most commonly used one to me) supported in scripting languages that also do without a size argument. The essential part of such memory containers is that their internal (hidden) metadata structure contains the length of the memory block in the container. For binary data this may be the norm, but for C-style strings, the stored and auto-updated length field serves as an accelerator cache. For more details, see `HXmc_length`.

Of course, the automatic management of memory comes with a bit of overhead as the string expands beyond its preallocated region. Such may be mitigated by doing explicit (re)sizing.

15.1 Structural overview

HXmc functions do not actually return a pointer to the memory container (e. g. `struct`) itself, but a pointer to the data block. Conversely, input parameters to HXmc functions will be the data block pointer. It is of type `hxmc_t *`, which is typedef'ed to `char *` and inherits all properties and privileges of `char *`. Pointer arithmetic is thus supported. It also means you can just pass it to functions that take a `char *` without having to do a member access like `s.c_str`. The drawback is that many functions operating on the memory container need a `hxmc_t **` (a level-two indirection), because not only does the memory block move, but also the memory container itself. This is due to the implementation of the container metadata which immediately and always precedes the writable memory block.

HXmc ensures that the data block is terminated by a NUL (`'\0'`) byte (unless you trash it), so you do not have to, and of course, to be on the safe side. But, the automatic NUL byte is not part of the region allocated by the user. That is, when one uses the classic approach with `malloc(4096)`, the user will have control of 4096 bytes and has to stuff the NUL byte in there somehow on his own; for strings this means the maximum string length is 4095. Requesting

space for a 4096-byte sized HXmc container gives you the possibility to use all 4096 bytes for the string, because HXmc provides a NUL byte.

By the way, `hxmc_t` is the *only* typedef in this entire library, to distinguish it from regular `char *` that does not have a backing memory cointainer.

15.2 Constructors, destructors

```
#include <libHX/string.h>

hxmc_t *HXmc_strinit(const char *s);
hxmc_t *HXmc_meminit(const void *ptr, size_t size);
```

HXmc_strinit Creates a new hxmc_t object from the supplied string and returns it.

HXmc_meminit Creates a new hxmc_t object from the supplied memory buffer of the given size and returns it. `HXmc_meminit(NULL, len)` may be used to obtain an empty container with a preallocated region of `len` bytes (zero is accepted for `len`).

15.3 Data manipulation

15.3.1 Binary-based

```
hxmc_t *HXmc_trunc(hxmc_t **mc, size_t len);
hxmc_t *HXmc_setlen(hxmc_t **mc, size_t len);
hxmc_t *HXmc_memcpy(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_memcat(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_mempcat(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_memins(hxmc_t **mc, size_t pos, const void *ptr, size_t len);
hxmc_t *HXmc_memdel(hxmc_t **mc, size_t pos, size_t len);
```

When `ptr` is `NULL`, each call behaves as if `len` would be zero. Specifically, no undefined behavior will result of the use of `NULL`.

HXmc_trunc Truncates the container's data to `len` size. If `len` is greater than the current data size of the container, the length is in fact *not* updated, but a reallocation may be triggered, which can be used to do explicit allocation.

HXmc_setlen Set the data length, doing a reallocation of the memory container if needed. The newly available bytes are uninitialized. Make use of this function when letting 3rd party functions write to the buffer, but it should not be used with `HXmc_str*`(),

HXmc_memcpy Truncates the container's data and copies `len` bytes from the memory area pointed to by `ptr` to the container.

HXmc_memcat Concatenates (appends) `len` bytes from the memory area pointed to by `ptr` to the container's data.

HXmc_mempcat Prepends `len` bytes from the memory area pointed to by `ptr` to the container's data.

HXmc_memins Prepends `len` bytes from the memory area pointed to by `ptr` to the `pos`'th byte of the container's data.

HXmc_memdel Deletes `len` bytes from the container beginning at position `pos`.

In case of a memory allocation failure, the `HXmc_*` functions will return `NULL`.

15.3.2 String-based

The string-based functions correspond to their binary-based equivalents with a `len` argument of `strlen(s)`.

```
hxmc_t *HXmc_strcpy(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strcat(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strpcat(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strins(hxmc_t **mc, size_t pos, const char *s);
```

15.3.3 From auxiliary sources

```
hxmc_t *HX_getl(hxmc_t **mc, FILE *fp);
```

HX_getl Read the next line from `fp` and store the result in the container. Returns `NULL` on error, or when end of file occurs while no characters have been read.

15.4 Container properties

```
size_t HXmc_length(const hxmc_t **mc);
```

HXmc_length Returns the length of the memory container. This is not always equal to the actual string length. For example, if `HX_chomp` was used on an MC-backed string, `strlen` will return less than `HXmc_length` if newline control characters (`'\r'` and `'\n'`) were removed.

16 Format templates

HXfmt is a small template system for by-name variable expansion. It can be used to substitute placeholders in format strings supplied by the user by appropriate expanded values defined by the program. Such can be used to allow for flexible configuration files that define key-value mappings such as

```
detect_peer = ping6 -c1 %(ADDR)
#detect_peer = nmap -sP %(ADDR) | grep -Eq "appears to be up"
```

Consider for example a monitoring daemon that allows the administrator to specify a program of his choice with which to detect whether a peer is alive or not. The user can choose any program that is desired, but evidently needs to pass the address to be tested to the program. This is where the daemon will do a substitution of the string "`ping -c1 %(ADDR)`" it read from the config file, and put the actual address in it before finally executing the command.

```
printf("%s has %u files\n", user, num);
printf("%2$u files belong to %1$s\n", num, user);
```

"%s" (or "%1$s" here) specifies how large "user" is — `sizeof(const char *)` in this case. If that is missing, there is no way to know the offset of "`num`" relative to "`user`", making varargs retrieval impossible.

Figure 12: `printf` positional parameters

`printf`, at least from GNU libc, has something vaguely similar: positional parameters. They have inherent drawbacks, though. One is of course the question of portability, but there is a bigger issue. All parameters must be specified, otherwise there is no way to determine the location of all following objects following the missing one on the stack in a varargs-function like `printf`., which makes it unsuitable to be used with templates where omitting some placeholders is allowed.

16.1 Initialization, use and deallocation

```
#include <libHX/option.h>

struct HXbtree *HXformat_init(void);
void HXformat_free(struct HXbtree *table);
int HXformat_add(struct HXbtree *table, const char *key,
                 const void *ptr, unsigned int ptr_type);
```

`HXformat_init` will allocate and set up a simple string-to-string map HXbtree that is used for the underlying storage, and returns it.

To release the substitution table and memory associated with it, call `HXformat_free`.

`HXbtree_add` is used to add substitution entries. Even though a string data-based HXbtree is used, one can specify other types such as numeral types. `ptr_type` describes the type behind `ptr` and are constants from `option.h` (cf. section 20.2) — not all constants can be used, though, and their meaning also differs from what `HX_getopt` or `HX_shconfig` use them for — the two could be seen as "read" operations, while HXformat is a write operation.

16.1.1 Immediate types

"Immediate types" are resolved when `HXformat_add` is called, that is, they are copied and inserted into the tree, and are subsequently independent from any changes to variables in the program. Because the HXopt-originating type name, that is, `HXTYPE_*`, is also used for deferred types, the constant `HXFORMAT_IMMED` needs to be specified on some types to denote an immediate value.

- `HXTYPE_STRING` — `ptr` is a `const char *`.
- `HXTYPE_{U,}{CHAR,SHORT,INT,LONG,LLONG} | HXFORMAT_IMMED` — mapping to the standard types

16.1.2 Deferred types

"Deferred types" are resolved on every invocation of a formatter function (`HXformat_*printf`). The expansions may be changed by modifying the underlying variable pointed to, but the pointer must remain valid and its pointee not go out of scope. Figure 13 shows the difference in a code sample.

- `HXTYPE_STRP` — `ptr` is a `const char *const *`; the pointer resolution is deferred until the formatter is called with one of the `HXformat_*printf` functions. Deferred in the sense it is always resolved anew.
- `HXTYPE_BOOL` — `ptr` is a `const int *`.
- `HXTYPE_{U,}{CHAR,SHORT,INT,LONG,LLONG}` — mapping to the standard types with one indirection (e. g. `int *`)
- `HXTYPE_{FLOAT,DOUBLE}` — mapping to the two floating-point types with one indirection (e. g. `double *`)

16.2 Invoking the formatter

```
int HXformat_aprintf(struct HXbtree *table, hxmc_t **dest, const char *template);
int HXformat_sprintf(struct HXbtree *table, char *dest, size_t size, const char *t
int HXformat_fprintf(struct HXbtree *table, FILE *filp, const char *template);
```

`HXformat_aprintf` Substitute placeholders in `template` using the given table. This will produce a string in a HX memory container (hxmc_t), and the pointer is put into `*dest`. The caller will be responsible for freeing it later when it is done using the result.

`HXformat_sprintf` Do substitution and store the expanded result in the buffer `dest` which is of size `size`.

`HXformat_fprintf` Do substituion and directly output the expansion to the given stdio stream.

On success, the length of the expanded string is returned, excluding the trailing `'\0'`. While `HXformat_sprintf` will not write more than `size` bytes (including the `'\0'`), the length it would have taken is returned, similar to what `sprintf` does. On error, negative errno is returned.

16.3 Examples

```
const char *b = "Hello World";
char c[] = "Hello World";
struct HXbtree *table = HXformat_init();
HXformat_add(table, "%(GREETING1)", b, HXTYPE_STRING);
HXformat_add(table, "%(GREETING2)", &c, HXTYPE_STRP);
b = NULL;
snprintf(c, sizeof(c), "Hello Home");
HXformat_aprintf(...);
```

Upon calling `HXformat_*printf`, `%(GREETING1)` will expand to "Hello World" whereas `%(GREETING2)` will expand to "Hello Home".

Figure 13: Immediate and deferred resolution

Part IV
Filesystem operations

17 Directory traversal

libHX provides a minimal readdir-style wrapper for cross-platform directory traversal. This is needed because platforms such as do not have readdir (e. g. Win32), or to work around peculiarities in the lower implementation. Solaris's `struct dirent` for example is "too small", that is, `readdir` will cause a buffer overrun when Linux code is directly ported to it without anticipating for this scenario. libHX's `dir.c` mitigates this.

17.1 Synopsis

```
#include <libHX/misc.h>

void *HXdir_open(const char *directory);
const char *HXdir_read(void *handle);
void HXdir_close(void *handle);
```

`HXdir_open` returns a pointer to its private data area, or `NULL` upon failure, in which case `errno` is preserved from the underlying system calls. `HXdir_read` causes the next entry from the directory to be fetched. The pointer returned by `HXdir_read` must not be freed, and the data is overwritten in subsequent calls to the same handle. If you want to keep it around, you will have to duplicate it yourself. `HXdir_close` will close the directory and free the private data it held.

17.2 Example

```
#include <errno.h>
#include <stdio.h>
#include <libHX/misc.h>

void *dh;
if ((dh = HXdir_open(".")) == NULL) {
        fprintf(stderr, "Could not open directory: %s\n", strerror(errno));
        return;
}
while ((dentry = HXdir_read(dh)) != NULL)
        printf("%s\n", dentry);
HXdir_close(dh);
```

This sample will open the current directory, and print out all entries as it iterates over them.

18 Directory operations

18.1 Synopsis

```
#include <libHX/misc.h>
```

```
int HX_mkdir(const char *path);
int HX_rrmdir(const char *path);
```

HX_mkdir will create the directory given by path, and all its parents that do not exist yet. It is equivalent to the 'mkdir -p' shell command. It will return >0 for success, or -errno on error.

HX_rrmdir also maps to an operation commonly done on the shell, 'rm -Rf', deleting the directory given by path, including all files within it and its subdirectories. Errors during deletion are ignored, but if there was any, the errno value of the first one is returned negated.

19 File operations

19.1 Synopsis

```
#include <libHX/misc.h>

int HX_copy_file(const char *src, const char *dest, unsigned int flags, ...);
int HX_copy_dir(const char *src, const char *dest, unsigned int flags, ...);
```

Possible flags that can be used with the functions:

HXF_KEEP Do not overwrite existing files.

HXF_UID Change the new file's owner to the UID given in the varargs section (...). HXF_UID is processed before HXF_GID.

HXF_GID Change the new file's group owner to the GID given in the varargs section. This is processed after HXF_UID.

Error checking is flakey.

HX_copy_file will return >0 on success, or -errno on failure. Errors can arise from the use of the syscalls open, read and write. The return value of fchmod, which is used to set the UID and GID, is actually ignored, which means verifying that the owner has been set cannot be detected with HX_copy_file alone (historic negligience?).

Part V
Options and Configuration Files

20 Option parsing

libHX uses a table-based approach like libpopt[6]. It provides for both long and short options and the different styles associated with them, such as absence or presence of an equals sign for long options (`--foo=bar` and `--foo bar`), bundling (writing `-abc` for non-argument taking options `-a -b -c`), squashing (writing `-fbar` for an argument-requiring option `-f bar`). The "lone dash" that is often used to indicate standard input or standard output, is correctly handled[7], as in `-f -`.

A table-based approach allows for the parser to run as one atomic block of code (callbacks are, by definition, "special" exceptions), making it more opaque than an open-coded `getopt`(3) loop. You give it your argument vector and the table, snip the finger (call the parser function once), and it is done. In getopt on the other hand, the `getopt` function returns for every argument it parsed and needs to be called repeatedly.

20.1 Synopsis

```
#include <libHX/option.h>

struct HXoption {
        const char *ln;
        char sh;
        unsigned int type;
        int ival;
        const char *sval;
        void *ptr, *uptr;
        void (*cb)(const struct HXoptcb *);
        const char *help, *htyp;
};

int HX_getopt(const struct HXoption *options_table, int *argc,
              const char ***argv, unsigned int flags);
```

The various fields of `struct HXoption` are:

`ln` The long option name, if any. May be `NULL` if none is to be assigned for this entry.

`sh` The short option name/character, if any. May be `'\0'` if none is to be assigned for this entry.

`type` The type of the entry, essentially denoting the type of the target variable.

`ival` An integer value to be stored into `*(int *)ptr` when `HXTYPE_IVAL` is used.

`sval` A string whose address will be stored into `*(const char **)ptr` when `HXTYPE_SVAL` is used.

[6]The alternative would be an iterative, open-coded approach like `getopt`(3) requires.

[7]popt failed to do this for a long time.

ptr A pointer to the variable so that the option parser can store the requested data in it. The pointer may be `NULL` in which case no data is stored (but `cb` is still called if defined, with the data).

uptr A user-supplied pointer. Its value is passed verbatim to the callback, and may be used for any purpose the user wishes.

cb If not `NULL`, call out to the referenced function after the option has been parsed (and the results possibly be stored in `ptr`)

help A help string that is shown for the option when the option table is dumped by request (e. g. `yourprgram --help`)

htyp String containing a keyword to aid the user in understanding the available options during dump. See examples.

Due to the amount of fields, it is advised to use C99 named initializers to populate a struct, as they allow to omit unspecified fields, and assume no specific order of the members:

```
struct HXoption e = {.sh = 'f', .help = "Force"};
```

It is a sad fact that C++ has not gotten around to implement these yet. It is advised to put the option parsing code into a separate `.c` file that can then be compiled in C99 rather than C++ mode.

20.2 Type map

HXTYPE_NONE The option does not take any argument, but the presence of the option may be record by setting the `*(int *)ptr` to 1. Other rules apply when `HXOPT_INC` or `HXOPT_DEC` are specified as flags (see section 20.3).

HXTYPE_VAL Use the integer value specified by `ival` and store it in `*(int *)ptr`.

HXTYPE_SVAL Use the memory location specified by `sval` and store it in `*(const char **)ptr`.

HXTYPE_BOOL Interpret the supplied argument as a boolean descriptive (must be "yes", "no", "on", "off", "true", "false", "0" or "1") and store the result in `*(int *)ptr`.

HXTYPE_STRING The argument string is duplicated to a new memory region and the resulting pointer stored into `*(char **)ptr`. This incurs an allocation so that subsequently modifying the original argument string in any way will not falsely propagate.

HXTYPE_STRDQ The argument string is duplicated to a new memory region and the resulting pointer is added to the given HXdeque. Note that you often need to use deferred initialization of the options table to avoid putting `NULL` into the entry. See section 20.6.1.

The following table lists the types that map to the common integral. Signed and unsigned integeral types are processed using `strtol` and `strtoul`, respectively. `strtol` and `strtoul` will be called with automatic base detection. This usually means that a leading "0" indicates the string is given in octal (8) base, a leading "0x" indicates hexadecimal (16) base, and decimal (10) otherwise. `HXTYPE_LLONG`, `HXTYPE_ULLONG`, `HXTYPE_INT64` and `HXTYPE_UINT64` use `strtoll` and/or `strtoull`, which may not be available on all platforms.

type	Type of pointee	type	Type of pointee
HXTYPE_CHAR	char	HXTYPE_INT8	int8_t
HXTYPE_UCHAR	unsigned char	HXTYPE_UINT8	uint8_t
HXTYPE_SHORT	short	HXTYPE_INT16	int16_t
HXTYPE_USHORT	unsigned short	HXTYPE_UINT16	uint16_t
HXTYPE_INT	int	HXTYPE_INT32	int32_t
HXTYPE_UINT	unsigned int	HXTYPE_UINT32	uint32_t
HXTYPE_LONG	long	HXTYPE_INT64	int64_t
HXTYPE_ULONG	unsigned long	HXTYPE_UINT64	uint64_t
HXTYPE_LLONG	long long	HXTYPE_FLOAT	float
HXTYPE_ULLONG	unsigned long long	HXTYPE_DOUBLE	double

Table 3: Integral and floating-point types for the libHX option parser

HXTYPE_FLOAT and HXTYPE_DOUBLE make use of strtod (strtof is not used). A corresponding type for the "long double" format is not specified, but may be implemented on behalf of the user via a callback (see section 20.8.4).

20.3 Flags

Flags can be combined into the type parameter by OR'ing them. It is valid to not specify any flags at all, but most flags collide with one another.

HXOPT_INC Perform an increment on the memory location specified by the *(int *)ptr pointer. Make sure the referenced variable is initialized before!

HXOPT_DEC Perform a decrement on the pointee.

Only one of HXOPT_INC and HXOPT_DEC may be specified at a time, and they require that the base type is HXTYPE_NONE, or they will have no effect. An example may be found in section 20.8.2.

HXOPT_NOT Binary negation of the argument directly after reading it from the command line into memory. Any of the three following operations are executed with the already-negated value.

HXOPT_OR Binary "OR"s the pointee with the specified/transformed value.

HXOPT_AND Binary "AND"s the pointee with the specified/transformed value.

HXOPT_XOR Binary "XOR"s the pointee with the specified/transformed value.

Only one of (HXOPT_OR, HXOPT_AND, HXOPT_XOR) may be specified at a time, but they can be used with any integral type (HXTYPE_UINT, HXTYPE_ULONG, etc.). An example can be found in section 20.8.3.

HXOPT_OPTIONAL This flag allows for an option to take zero or one argument. Needless to say that this can be confusing to the user. *iptables*'s "-L" option for example is one of this kind (though it does not use the libHX option parser). When this flag is used, "-f -b" is interpreted as -f without an argument, as is "-f --bar". "-f -" of course is not, because "-" is not an option, but serves to indicate standard input/output.

20.4 Special entries

HXopt provides two special entries via macros:

HXOPT_AUTOHELP Adds entries to recognize "-?" and "--help" that will display the (long-format) help screen, and "--usage" that will display the short option syntax overview. All three options will exit the program afterwards.

HXOPT_TABLEEND This sentinel marks the end of the table and is required on all tables. (See examples for details.)

20.5 Invoking the parser

```
int HX_getopt(const struct HXoption *options_table, int *argc,
              const char ***argv, unsigned int flags);
```

HX_getopt is the actual parsing function. It takes the option table, and a pointer to your argc and argv variables that you get from the main function. The parser will, unlike GNU getopt, literally "eats" all options and their arguments, leaving only non-options in argv, and argc updated, when finished. This is similar to how Perl's "Getopt::Long" module works. Additional flags can control the exact behavior of HX_getopt:

HXOPT_PTHRU "Passthrough mode". Any unknown options are not "eaten" and are instead passed back into the resulting argv array.

HXOPT_QUIET Do not print any diagnostics when encountering errors in the user's input.

HXOPT_HELPONERR Display the (long-format) help when an error, such as an unknown option or a violation of syntax, is encountered.

HXOPT_USAGEONERR Display the short-format usage syntax when an error is encountered.

The return value can be one of the following:

-HXOPT_ERR_UNKN An unknown option was encountered.

-HXOPT_ERR_VOID An argument was given for an option which does not allow one. In practice this only happens with "--foo=bar" when --foo is of type HXTYPE_NONE, HXTYPE_VAL or HXTYPE_SVAL. This does not affect "--foo bar", because this can be unambiguously interpreted as "bar" being a remaining argument to the program.

-HXOPT_ERR_MIS Missing argument for an option that requires one.

positive non-zero Success.

20.6 Pitfalls

20.6.1 Staticness of tables

The following is an example of a trap regarding HXTYPE_STRDQ:

```
static struct HXdeque *dq;

static bool get_options(int *argc, const char ***argv)
{
```

```
        static const struct HXoption options_table[] = {
                {.sh = 'N', .type = HXTYPE_STRDQ, .q_strdq = dq,
                 .help = "Add name"},
                HXOPT_TABLEEND,
        };
        return HX_getopt(options_table, argc, argv, HXOPT_USAGEONERR) > 0;
}

int main(int argc, const char **argv)
{
        dq = HXdeque_init();
        get_options(&argc, &argv);
        return 0;
}
```

The problem here is that `options_table` is, due to the `static` keyword, initialized at compile-time where `dq` is still `NULL`. To counter this problem and have it doing the right thing, you must remove the `static` qualifier on the options table when used with `HXTYPE_STRDQ`, so that it will be evaluated when it is first executed.

It was not deemed worthwhile to have `HXTYPE_STRDQ` take an indirect HXdeque (`struct HXdeque **`) instead just to bypass this issue. (Live with it.)

20.7 Limitations

The HX option parser has been influenced by both popt and Getopt::Long, but eventually, there are differences:

- Long options with a single dash ("`-foo bar`"). This unsupported syntax clashes very easily with support for option bundling or squashing. In case of bundling, "`-foo`" might actually be "`-f -o -o`", or "`-f oo`" in case of squashing. It also introduces redundant ways to specify options, which is not in the spirit of the author.

- Options using a "`+`" as a prefix, as in "`+foo`". Xterm for example uses it as a way to negate an option. In the author's opinion, using one character to specify options is enough — by GNU standards, a negator is named "`--no-foo`". Even Microsoft stuck to a single option introducing character (that would be "`/`").

- Table nesting like implemented in popt. HXopt has no provision for nested tables, as the need has not come up yet. It does however support chained processing (see section 20.8.5). You cannot do nested tables even with callbacks, as the new `argv` array is only put in place shortly before `HX_getopt` returns.

20.8 Examples

20.8.1 Basic example

The following code snippet should provide an equivalent of the GNU getopt sample[8].

```
#include <stdio.h>
#include <stdilb.h>
#include <libHX/option.h>
```

[8] `http://www.gnu.org/software/libtool/manual/libc/Example-of-Getopt.html#Example-of-Getopt`

```
int main(int argc, const char **argv)
{
        int aflag = 0;
        int bflag = 0;
        char *cflag = NULL;

        struct HXoption options_table[] = {
                {.sh = 'a', .type = HXTYPE_NONE, .ptr = &aflag},
                {.sh = 'b', .type = HXTYPE_NONE, .ptr = &bflag},
                {.sh = 'c', .type = HXTYPE_STRING, .ptr = &cflag},
                HXOPT_AUTOHELP,
                HXOPT_TABLEEND,
        };

        if (HX_getopt(options_table, &argc, &argv, HXOPT_USAGEONERR) <= 0)
                return EXIT_FAILURE;

        printf("aflag = %d, bflag = %d, cvalue = %s\n",
               aflag, bflag, cvalue);

        while (*++argv != NULL)
                printf("Non-option argument %s\n", *argv);

        return EXIT_SUCCESS;
}
```

20.8.2 Verbosity levels

```
static int verbosity = 1; /* somewhat silent by default */
static const struct HXoption options_table[] = {
        {.sh = 'q', .type = HXTYPE_NONE | HXOPT_DEC, .q_int = &verbosity,
         .help = "Reduce verbosity"},
        {.sh = 'v', .type = HXTYPE_NONE | HXOPT_INC, .q_int = &verbosity,
         .help = "Increase verbosity"},
        HXOPT_TABLEEND,
};
```

This sample option table makes it possible to turn the verbosity of the program up or down, depending on whether the user specified `-q` or `-v`. By passing multiple `-v` flags, the verbosity can be turned up even more. The range depends on the "`int`" data type for your particular platform and compiler; if you want to have the verbosity capped at a specific level, you will need to use an extra callback:

```
static int verbosity = 1;

static void v_check(const struct HXoptcb *cbi)
{
        if (verbosity < 0)
                verbosity = 0;
        else if (verbosity > 4)
```

```
                verbosity = 4;
}

static const struct HXoption options_table[] = {
        {.sh = 'q', .type = HXTYPE_NONE | HXOPT_DEC, .q_int = &verbosity,
         .cb = v_check, .help = "Lower verbosity"},
        {.sh = 'v', .type = HXTYPE_NONE | HXOPT_INC, .q_int = &verbosity,
         .cb = v_check, .help = "Raise verbosity"},
        HXOPT_TABLEEND,
};
```

20.8.3 Mask operations

```
/* run on all CPU cores by default */
static unsigned int cpu_mask = ~0U;
/* use no network connections by default */
static unsigned int net_mask = 0;
static struct HXoption options_table[] = {
        {.sh = 'c', .type = HXTYPE_UINT | HXOPT_NOT | HXOPT_AND,
         .q_uint = &cpu_mask,
         .help = "Mask of cores to exclude", .htyp = "cpu_mask"},
        {.sh = 'n', .type = HXTYPE_UINT | HXOPT_OR, .q_uint = &net_mask,
         .help = "Mask of network channels to additionally use",
         .htyp = "channel_mask"},
        HXOPT_TABLEEND,
};
```

What this options table does is `cpu_mask &= ~x` and `net_mask |= y`, the classic operations of clearing and setting bits.

20.8.4 Support for non-standard actions

Supporting additional types or custom storage formats is easy, by simply using `HXTYPE_STRING`, `NULL` as the data pointer (usually by not specifying it at all), the pointer to your data in the user-specified pointer `uptr`, and the callback function in `cb`.

```
struct fixed_point {
        int integral;
        unsigned int fraction;
};

static struct fixed_point number;

static void fixed_point_parse(const struct HXoptcb *cbi)
{
        char *end;

        number.integral = strtol(cbi->data, &end, 0);
        if (*end == '\0')
                number.fraction = 0;
        else if (*end == '.')
```

```
                number.fraction = strtoul(end + 1, NULL, 0);
        else
                fprintf(stderr, "Illegal input.\n");
}

static const struct HXoption options_table[] = {
        {.sh = 'n', .type = HXTYPE_STRING, .cb = fixed_point_parse,
         .uptr = &number, .help = "Do this or that",
        HXOPT_TABLEEND,
};
```

20.8.5 Chained argument processing

On the first run, only `--cake` and `--fruit` is considered, which is then used to select the next set of accepted options. Note that `HXOPT_DESTROY_OLD` is used here, which causes the argv that is produced by the first invocation of `HX_getopt` in the `get_options` function to be freed as it gets replaced by a new argv again by `HX_getopt` in `get_cakes`/`get_fruit`. `HXOPT_DESTROY_OLD` is however *not* specified in the first invocation, because the initial argv resides on the stack and cannot be freed.

```
static bool get_cakes(int *argc, const char ***argv)
{
        struct HXoption option_table[] = {
                ...
        };
        return HX_getopt(cake_table, argc, argv,
               HXOPT_USAGEONERR | HXOPT_DESTROY_OLD) > 0;
}

static bool get_fruit(int *argc, const char ***argv)
{
        struct HXoption fruit_table[] = {
                ...
        };
        return HX_getopt(fruit_table, argc, argv,
               HXOPT_USAGEONERR | HXOPT_DESTROY_OLD) > 0;
}

static bool get_options(int *argc, const char ***argv)
{
        int cake = 0, fruit = 0;
        struct HXoption option_table[] = {
                {.ln = "cake",  .type = HXTYPE_NONE, .ptr = &cake},
                {.ln = "fruit", .type = HXTYPE_NONE, .ptr = &fruit},
                HXOPT_TABLEEND,
        };
        if (HX_getopt(option_table, argc, argv, HXOPT_PTHRU) <= 0)
                return false;
        if (cake)
                return get_cakes(argc, argv);
        else if (fruit)
```

```
                return get_fruit(argc, argv);
        return false;
}
```

21 Shell-style configuration file parser

libHX provides functions to read shell-style configuration files. Such files are common, for example, in `/etc/sysconfig` on Linux systems. The format is pretty basic; it only knows about "`key=value`" pairs and does not even have sections like INI files. Not relying on any features however makes them quite interchangable as the syntax is accepted by Unix Shells.

Lines beginning with a hash mark (`#`) are ignored, as are empty lines and unrecognized keys.

```
# Minimum / maximum values for automatic UID selection
UID_MIN=100
UID_MAX=65000

# Home directory base
HOME="/home"
#HOME="/export/home"
```

Any form of variable or parameter substitution or expansion is highly implementation specific, and is not supported in libHX's reader. Even Shell users should not rely on it as you never know in which context the configuration files are evaluated. Still, you will have to escape specific sequences like you would need to in Shell. The use of single quotes is acceptable. That means:

```
AMOUNT="US\$5"
AMOUNT='US$5'
```

21.1 Synopsis

```
#include <libHX/option.h>

int HX_shconfig(const char *file, const struct HXoption *table);
int HX_shconfig_pv(const char **path_vec, const char *file,
                   const struct HXoption *table, unsigned int flags);
```

The shconfig parser reuses `struct HXoption` that fits very well in specifying name-pointer associations. `HX_shconfig` will read the given file using the key-to-pointer mappings from the table to store the variable contents. Of `struct HXoption`, described in section 20.1, only the "`ln`", "`type`" and "`ptr`" fields are used. The list of accepted types is described in section 20.2.

To parse a file, call `HX_shconfig` function with the corresponding parameters. If you want to read configuration files from different paths, i. e. to build up on default values, you can use `HX_shconfig_pv`[9], which is a variant for reading a file from multiple locations. Its purpose is to facilitate reading system-wide settings which are then overriden by a file in the users home directory, for example (per-setting-override). It is also possible to do per-file-override, that is, a file in the home directory has higher precedence than a system-wide one in such a way that the system-wide configuration file is not even read. This is accomplished by traversing the paths in the "other" direction (actually you have to turn the array around) and stopping at the first existing file by use of the `SHCONF_ONE` flag.

`SHCONF_ONE` Parsing files will stop after one file has been successfully parsed. This allows for a "personal overrides system config" style.

The call to `HX_shconfig` will either return >0 for success, 0 for no success (actually, this is never returned) and `-errno` for an error.

[9] pv = path vector

21.2 Example

21.2.1 Per-setting-override

```
long uid_min, uid_max;
char *passwd_file;
struct HXoption options_table[] = {
        {.ln = "UID_MIN",  .type = HXTYPE_LONG,   .ptr = &uid_min},
        {.ln = "UID_MAX",  .type = HXTYPE_LONG,   .ptr = &uid_max},
        {.ln = "PWD_FILE", .type = HXTYPE_STRING, .ptr = &passwd_file},
        HXOPT_TABLEEND,
};
const char *home = getenv("HOME");
const char *paths[] = {"/etc", home, NULL};
HX_shconfig(paths, "test.cf", options_table, 0);
```

21.2.2 Per-file-override

```
const char *home = getenv("HOME");
const char *paths[] = {home, "/usr/local/etc", "/etc", NULL};
HX_shconfig_pv(paths, "test.cf", options_table, SHCONF_ONE);
```

Part VI
Systems-related components

22 Random numbers

22.1 Function overview

```
#include <libHX/misc.h>

int HX_rand(void);
unsigned int HX_irand(unsigned int min, unsigned int max);
```

HX_rand Retrieve the next random number, with $0 \leq n < 2^{b-1}$ where b is the number of bits in an "int". This is different than libc's rand which returns a number $0 \leq n \leq$ RAND_MAX.

HX_irand Retrieve the next random number and fold it so that $min \leq n < max$.

22.2 Implementation information

On systems that provide operating system-level random number generators, predominantly Linux and Unix-alikes such as BSD and Solaris, these will be used when they are available and random numbers are requested through HX_rand or HX_irand.

On Linux, Solaris and the BSDs, this is /dev/urandom.

If no random number generating device is available (and libHX configured to use it), it will fall back to using the libc's rand function. If libc is selected for random number generation, srand will be called on library initialization with what is believed to be good defaults — usually this will be before a program's main function with normal linking, but may actually happen later when used with dlopen. The initial seed would be the current microtime when gettimeofday is available, or just the seconds with time. To counter the problem of different programs potentially using the same seed within a time window of a second due to the limited granularity of standard time, the seed is augmented by process ID and parent process ID where available.

/dev/random is not used on Linux because it may block during read, and /dev/urandom is just as good when there is entropy available. If you need definitive PRNG security, perhaps use one from a crypto suite such as OpenSSL.

23 Process management

The process code is experimental at this stage (just moved from the pam_mount codebase). As it also relies on the POSIX functions `fork`, `execv`, `execvp` and `pipe`(2), so it may not be available everywhere. Where this is the case, the functions will return `-ENOSYS`.

23.1 Process metadata structure

```
#include <libHX/proc.h>

struct HXproc {
        const struct HXproc_ops *p_ops;
        void *p_data;
        unsigned int p_flags;

        /* Following members should only be read */
        int p_stdin, p_stdout, p_stderr;
        int p_pid;
        char p_status;
        bool p_exited, p_terminated;
};
```

When creating a new process with the intent of running it asynchronously (using `HXproc_run_async`), the first three fields must be filled in by the user.

p_ops A table of callbacks, generally used for setting and/or restoring signals before/after execution. This member may be `NULL`.

p_data Free pointer for the user to supply. Will be passed to the callback functions when they are invoked.

p_flags Process creation flags, see below.

After the subprocess has been started, `HXproc_run_async` will have filled in some fields:

p_stdin If `HXPROC_STDIN` was specified in `p_flags`, `p_stdin` will be assigned the write side file descriptor of the subprocess's to-be stdin. The subprocess will get the read side file descriptor in this member. This is so that the correct fd is used in when `p_ops->p_postfork` is called.

p_stdout If `HXPROC_STDOUT` is specified in `p_flags`, `p_stdout` will be assigned the read side file descriptor of the subprocess's to-be stdout. The subprocess will get the write side file descriptor in this member.

p_stderr If `HXPROC_STDERR` is specified in `p_flags`, `p_stderr` will be assigned the read side file descriptor of the subprocess's to-be stderr, and the subprocess will get the write side fd.

p_pid The process ID of the spawned process.

Upon calling `HXproc_wait`, further fields will have been filled when the function returns:

p_exited Whether the process exited normally (cf. signalled/terminated).

p_terminated Whether the process was terminated (signalled).

p_status The exit status of the process or the termination signal.

23.1.1 Flags

Possible values for the `p_flags` member are:

HXPROC_STDIN The subprocess's stdin file descriptor shall be connected to the master program, that is, not inherit the stdin of the master.

HXPROC_STDOUT Connect the stdout file descriptor of the subprocess with the master.

HXPROC_STDERR Connect the stderr file descriptor of the subprocess with the master.

HXPROC_VERBOSE Have the subprocess print an error message to stderr if exec'ing returned an error.

HXPROC_A0 `argv[0]` refers to program file, while `argv[1]` to the program invocation name, with `argv[2]` being the arguments. Without this flag, `argv[0]` will be both the program file and program invocation name, and arguments begin at `argv[1]`.

HXPROC_EXECV Normally, `execvp`(3) will be used which scans `$PATH` for the program. Use this flag to use `execv`(3) instead, which will not do such thing.

23.2 Callbacks

```
#include <libHX/proc.h>

struct HXproc_ops {
        void (*p_prefork)(void *);
        void (*p_postfork)(void *);
        void (*p_complete)(void *);
};
```

`struct HXproc_ops` provides a way to run user-specified functions just before the fork, after, and when the process has been waited for. They can be used to set and/or restore signals as needed, for example. The function pointers can be `NULL`. The `p_data` member is passed as an argument.

p_prefork Run immediately before calling `fork`(2). This is useful, for taking any action regarding signals, like setting `SIGCHLD` to `SIG_DFL`, or `SIGPIPE` to `SIG_IGN`, for example.

p_postfork Run in the subprocess (and only there) after forking. Useful to do a `setuid`(2) or other change in privilege level.

p_complete Run in `HXproc_wait` when the process has been waited for. Useful to restore the signal handler(s).

23.3 Process control

```
#include <libHX/proc.h>

int HXproc_run_async(const char *const *argv, struct HXproc *proc);
int HXproc_run_sync(const char *const *argv, unsigned int flags);
int HXproc_wait(struct HXproc *proc);
```

HXproc_run_async Start a subprocess according to the parameters in `proc`. Returns a negative errno code if something went wrong, or positive non-zero on success.

HXproc_run_sync Start a subprocess synchronously, similar to calling system(3), but with the luxury of being able to specify arguments as separate strings (via argv) rather than one big command line that is run through the shell. flags is a value composed of the HXproc flags mentioned above in section 23.1.1. HXPROC_STDIN, HXPROC_STDOUT and HXPROC_STDERR are ignored because there would be no one in a synchronous execution that could supply data to these file descriptors or read from them[10].

HXproc_wait Wait for a subprocess to terminate, if it has not already. It will also retrieve the exit status of the process and store it in the struct HXproc.

24 Helper headers

24.1 ctype helpers

Functions from the <ctype.h> header, including, but not limited to, isalpha, tolower, and so forth, are defined to take an "int" as first argument. Strings used in C programs are usually "char *", without any "signed" or "unsigned" qualifier. By a high-level view, which also matches daily common sense, characters (a. k. a. letters) have no notion of signedness — there is no "positive" or "negative" "A" in at least the Latin alphabet that is mapped into the ASCII set. In fact, char * could either be signed char * or unsigned char *, depending on the compiler settings. Only when you start interpreting and using characters as a number does such become important.

There come the problems. Characters are in the same class as numbers in C, that is, can be implicitly converted from or to a "number" (in this case, their ASCII code point) without causing a compiler warning. That may be practical in some cases, but is also a bit "unfortunate". Characters, when interpreted as the 8-bit signed numeric quantity they are implicitly convertable to, run from 0 to 127 and -128 to -1. Since the isalpha function and others from ctype.h take a (signed) int as argument means that values fed to isalpha are sign-extended, preserving negative values.

```
/* "hyvää yötä", UTF-8 encoded */
const char h[] = {'h', 'y', 'v', 0xc3, 0xa4, 0xc3, 0xa4, ' ',
                  'y', 0xc3, 0xb6, 't', 0xc3, 0xa4};
```

When you now pass h[3] to isalpha for example (regardless of whether doing so actually produces a meaningful result), the CPU is instructed to copy "0xc3" into a register and sign-extend it (because "char" is often "signed char", see above), producing 0xffffffc3 (-61). But passing -61 is not what was intended.

libHX's ctype_helper.h therefore provides wrappers with a different function signature that uses zero extension (not sign extension) by means of using an unsigned quantity. Currently this is unsigned char, because isalpha's domain only goes from 0–255. The implication is that you cannot pass EOF to HX_isalpha.

```
#include <libHX/ctype_helper.h>

bool HX_isalnum(unsigned char c);
bool HX_isalpha(unsigned char c);
bool HX_isdigit(unsigned char c);
bool HX_islower(unsigned char c);
```

[10]Even for threads, please just use the async model.

```
bool HX_isprint(unsigned char c);
bool HX_isspace(unsigned char c);
bool HX_isupper(unsigned char c);
bool HX_isxdigit(unsigned char c);
unsigned char HX_tolower(unsigned char c);
unsigned char HX_toupper(unsigned char c);
```

The `is*` functions also differ from ctype's in that they return `bool` instead of `int`. Not all functions from `ctype.h` are present either; `isascii`, `isblank`, `iscntrl`, `isgraph`, `ispunct` and `isxdigit` have been omitted as the author has never needed them so far.

24.2 libxml2 helpers

libxml2 uses an "`xmlChar`" type as an underlying type for the strings that it reads and outputs. `xmlChar` is typedef'ed to `unsigned char` by libxml2, causing compiler warnings related to differing signedness whenever interacting with strings from the outside world, which are usually just a pointer to `char`. Because casting would be a real chore, `libxml_helper.h` will do it by providing some wrappers with better argument types.

```
#include <libHX/libxml_helper.h>

int xml_strcmp(const xmlChar *a, const char *b);
int xml_strcasecmp(const xmlChar *a, const char *b);
char *xml_getprop(xmlNode *node, const char *attr);
xmlAttr *xml_newprop(xmlNode *node, const char *attr);
xmlNode *xml_newnode(xmlNs *ns, const char *name);
xmlAttr *xml_setprop(xmlNode *node, const char *name, const char *value);
```

The functions map to `strcmp`(3), `strcasecmp`(3), `xmlGetProp`, `xmlNewProp`, `xmlNewNode` and `xmlSetProp`, respectively.

24.3 wxWidgets

```
#include <libHX/wx_helper.hpp>
```

24.3.1 Shortcut macros

`wxACV` Expands to `wxALIGN_CENTER_VERTICAL`

`wxCFF` Expands to a set of "common frame flags" for dialogs.

`wxDPOS` Expands to `wxDefaultPosition`.

`wxDSIZE` Expands to `wxDefaultSize`.

24.3.2 String conversion

```
wxString wxfu8(const char *);
wxString wxfv8(const char *);
const char *wxtu8(const wxString &);
```

`wxfu8` Converts an UTF-8 string to a `wxString` object.

wxfv8 Converts an UTF-8 string to an entity usable by `wxPrintf`.

wxtu8 Converts a wxString to a pointer to char usable by `printf`. Note that the validity of the pointer is very limited and usually does not extend the statement in which it is used. Hence storing the pointer in a variable ("`const char *p = wxtu8(s);`") will make `p` pointing to an invalid region as soon as the assignment is done.

Part VII
Appendix

Index

L