QISH introduction

Basile STARYNKEVITCH
basile@starynkevitch.net
http://www.starynkevitch.net/Basile/
8, rue de la Faencerie, 92340 Bourg La Reine, France

May 8, 2005

$Id: qishintro.tex 19 2005-05-08 17:35:50Z basile $

Contents

Please be nice to send me an email if you use this information and this Qish software.

Qish is available from `http://www.starynkevitch.net/Basile/qish-1.0pre1.tar.gz` (as a gnuzipped source tarball) and the latest snapshot is on `http://www.starynkevitch.net/Basile/qish-snapshot.tar.gz` and this document is on `http://www.starynkevitch.net/Basile/qishintro.html`. See also my home page on `http://www.starynkevitch.net/Basile/` or Qish page `http://freshmeat.net/projects/qish/` on Freshmeat for announcement of newer versions.

Documentation should be rewritten. Multithreading is not fully working yet in version 0.9. Stay tuned.

A mailing list (not yet archived) is available as `qish@lists.apinc.org` send an email to `qish-subscribe@lists.apinc.org` or to me at `basile@starynkevitch.net` for subscription.

Qish is developed on a PC/x86 running Linux. It could be portable to other Unixes machines[1]. Qish could be ported to x86 under Windows, but I don't want to do that.

To compile Qish on a strictly conforming ISO C 1999 compiler use the `-DSTRICT_C99` compile flag[2]. But Qish requires that successful pointer arguments are located consecutively (upwards or downwards) in memory (which is not guaranteed or even meaningful in general for ISO C 1999 compilers.).

The machine dependent parts of Qish are carefully coded, but not tested elsewhere. If you have access to other machine architecture, please tell me if you succeeded in compiling, porting, and running this software. Qish requires that successive pointer arguments and pointer fields are consecutive in memory.

Boehm's conservative garbage collector on `http://www.hpl.hp.com/personal/Hans_Boehm/gc/` should be easier to use[3], but does not compact memory, as any copying GC (like Qish) does. See the section 4 for a small (and not very significant) benchmark running both Boehm's and Qish GC (and explicit malloc and free).

This GC is not mostly copying[4]. It copies every movable object (even if it is on the call stack - in that case the variable or argument should be known to our GC thru the `BEGIN_LOCAL_FRAME` or equivalent macro), and mark fixed objects (which have finalizers).

This GC uses classical algorithms (inspired by some A.Appel's papers and R.Lins

[1] On some machines you'll need to flush the register stack in `qish_garbagecollect` - for example, Sparc machines might need `ta 3` to flush registers

[2] With this flag, Qish compiles without warnings both on GNU `gcc-3.2` with `-pedantic -std=c99` and on `tcc` - see `http://tinycc.org/`

[3] Boehm's GC has an interface compatible with `malloc`, is compatible with threads and finalization, and is used in GCC-3 for the Java runtime

[4] Unlike Bartlett's GC -1990- US patent 4,907,151 - thanks to Gerd Moellmann for the reference

& R.Jones' book). But I know of no other (classical) copying generational GC, usable for C, which can be customized to arbitrary data structures (while following stringent coding rules). This GC is probably best suited for generated C code, since the generator could be designed to follow every required coding rules. A code generator might use `C--` (and `QuickC--` see `http://www.cminusminus.org/`) when it will be available.

This GC is a *copying precise* (or exact) garbage collector. This means that pointers are changed by the GC (which may be called at every allocation point). The GC may change any (GC-managed) pointer on the call stack and in the heap. So you have to particularily be careful to tell the GC where are each pointer on the stack[5] and inside each object in the heap[6], and the GC might change any of them (to the address of a fresh copy of the pointed object). And the compiler should not optimize[7] too much (eg by putting a local variable only in a register where the GC can't change it).

Qish should be portable to any Unix, provided that the C compiler understands the **volatile** keyword as meaning that the declared volatile pointer may be changed by the GC. Purists says that Qish does not target the full C language[8], but only a subset; this is true in the sense that specific coding style is required, and that the compiler should understand the **volatile** keyword as above.

An alternative compiler on Linux is `tcc`[9] on `http://tinycc.org` which compiles extremely quickly (and is very suitable for dynamically generated C code using Qish and following Qish coding conventions). Copying collectors **condemn** a region of memory then push all objects out of this region by copying them, forwarding the pointer to them, and changing all pointers to the new copy.

This GC is a *generational* garbage collector. This means that the garbage collector focus work on newer objects, assuming that they are mostly temporary and will die soon. This suggests or favors a programming style with lots of (usually temporary) object allocations. So object allocation is quite fast on the average. But the garbage collector has to notice and scan explicitly any old object which has been updated by the application to point to a newer object. This require a write barrier, i.e. the notification of object updates thru `qish_write_notify`.

[5] You tell the GC where are the stack pointers with the `BEGIN_LOCAL_FRAME` or equivalent macro, which tells the GC the number of local pointers, the first of them, and the number of argument pointers and the first such argument.

[6] You tell the GC where are pointers inside heap objects by explicitly providing *mandatory routines* (see section 3.4) to scan and copy such heap objects.

[7] This is why the **volatile** keyword is required for arguments and for the `_locals_` structure.

[8] But every GC, even conservative, for C requires some coding style, usually much less restrictive than for Qish...

[9] Tcc don't optimise much and ignores the **volatile** keyword. The generated code is about 30% slower than GCC with optimisation, but the compilation time is sometimes ten times faster than with `gcc`.

Any precise garbage collector requires some coding conventions in C (to know about pointers on the call stack, or for the write barrier,...): examples include the Ocaml runtime primitives coding conventions (see `http://caml.inria.fr/ocaml/htmlman/manual032.html` section 18.5) or the Xemacs primitives coding conventions (see `http://www.xemacs.org/Documentation/21.5/html/internals_12.html`).

1 Short overview

Qish contains a reusable (i.e. rather generic) generational copying garbage collector usable from C. This garbage collector requires a particular (and low-level) coding style.

If this stuff is useful to you, be nice to send me an email to `basile@starynkevitch.net`. The license of this software is the GNU Lesser General Public License [10] (i.e. LGPL - see the COPYING file); so this is a free or opensource software.(if this license annoys you, send me an email).

Why the name Qish? Qish was the father of Sal, see 1 Samuel, chap 9 (the Bible). I'm bored of finding useful acronyms (especially pleasant in several European languages), so I am using names from the Holy Bible. At least I hope that Qish is not offensive[11].

2 Requirements

To use this package, you need the following stuff:

- Ruby (for scripting) see `http://www.ruby-lang.org/`. I don't know much Perl and prefer Ruby for scripting tasks
- a Glibc system such as GNU/Linux, etc... I am using a Linux Debian/Sid system with a 2.4.22 kernel.
- The GNU make utility, since I depend upon GNU make extensions - see `http://www.gnu.org/software/make/`. I am using make 3.80.
- The GNU GCC compiler version 3.3 or later (I recommend against using 3.2, 3.1, 3.0 or 2.x versions) - see `http://www.gnu.org/software/gcc/`.
- the Tiny CC compiler (by Fabrice Bellard) on `http://tinycc.org` is very useful to compile (perhaps generated) C code using Qish (but you may compile the runtime with GCC using optimisations). `tcc 0.9.14` compiles very quickly (sometimes 10 times faster than gcc 3.2) while producing code no more than 30% slower than `gcc -O3`.
- This documentation is processed with LaTeX and HeVeA. HeVeA is a good LaTeX to HTML translator. See http://para.inria.fr/ maranget/hevea/ It is written in Ocaml. See `http://www.ocaml.org/`

[10] it used to be GPL only.

[11] In the past I experimented on a reflexive system that I also named Qish - they have no much in common except the name and the author

- I am using PRCS for version control. See `http://prcs.sourceforge.net/`. But it is only needed in the `scripts/qish_snap` Ruby script. You should not need it otherwise.

3 The garbage collector

A garbage collector (or GC) manage resources (mostly memory). If you are not familiar with garbage collection, see R.Jones's GC page on `http://www.cs.ukc.ac.uk/people/staff/rej/gc.html` and also `http://www.memorymanagement.org`. In this document, an object is just a memory zone managed by the GC; it may be or not an object visible to your application. All pointers to this object points to the start of the object and are managed by the GC. A word may contain a pointer or some kind of integer. On x86, words are 4 byte long.

A *copying* GC moves objects. The advantage of moving objects is that freeing dead objects is easy; alive objects are moved outside a big zone, and then this big zone is freed at once. So dead objects are not reclaimed one by one. Also, a copying GC do compact the used memory (but need temporarily twice as much as memory), thus avoiding fragmentation. So, *pointers are changed* by the copying GC. Therefore, all object pointers are `volatile` for the C compiler, and the garbage collector needs to know *about every (garbage collected) pointer*.

A *generational* GC favors young objects. It separate objects in old and new regions, and do two kinds of garbage collections (minor and full). Objects are linearly allocated in a birth zone. When this zone is full, a *minor garbage collection* is trigerred, which copy alive born object into the *old* region. Once in a while a *full garbage collection* is done, by condemning the previous old region and copying alive old objects into a fresher old region (and also working on the birth zone like the minor GC does). Care should be taken about pointers from object in the old region to the new birth one. So every object modification (when a pointer field in an object is changed) should be explicitly notified to the GC. Allocation in our GC is much faster (in the usual case, when no garbage collection is needed) than a cheap call to a `malloc` like routine.

Our GC also provides *finalized fixed objects*. Such objects are not moved, and are explicitly destroyed (one by one) by the GC, which calls a finalization routine. Finalized objects are more costly than copied ones. They are intended to manage external system resources like files or windows. The finalization routine should not use or change garbage collected pointers or fields (except by clearing them). Our finalized objects are not like Java's in that respect: the finalizer routine cannot allocate any GC-ed object!

Our GC also supports tagged integers. Any pointer word ending with a set LSB bit is assumed to be such an integer. Use the `qish_is_tagged_int` inlined function to test if a pointer is such a tagged integer. If it is one, you can convert it to `int` or `long` with the `QISH_TAGGED2INT` and `QISH_TAGGED2LONG` macros. To make a tagged integer (ie to encode integers in a GC-ed pointer) use the `QISH_INT2TAGGED` or `QISH_LONG2TAGGED` macros.

Qish provides global constant pointers (up to 65536 pointers). Use `QISH_GLOBCONST(`*N*`)` to get the *N*-th constant, and `QISH_SET_GLOBCONST(`*N,V*`)` to set the *N*-th constant to the pointer value *V*.

Qish also provides module constant garbage collected pointers (one pointer per module).

3.1 Introductory examples

Notice: examples talk about Ruko which is obsolete.

To give a concrete feeling about Qish runtime, here are some illustrative examples from Ruko. Ruko provides (among other types) vectors and tuples, which are sequences of garbage-collected object pointers. Vectors are mutable (the components `vect->tab` can be changed) but tuples are immutable (the components are set at creation time and then are read-only).

The file `ruko/ruko.h` declare the following structure (common to vectors and tuples):

```
struct ktuple_t {
unsigned header; // first word is a common discriminating
header
void* tab[0]; // next words are GC-ed object pointers
};
```

The `header` word encodes both a kind (which is `KIND_TUPLE` for a tuple and `KIND_VECTOR` for a vector) and a size - which for vectors and tuples is the number of components, i.e. the real length of the `vect->tab` field. But other uses of Qish have their first word containing a garbage-collected *non-null* pointer to a class object.

Here is the commented code (from file `ruko/ruko.c`) of the `ruk_dup_vector` function, which duplicate a source tuple or vector as a fresh newly allocated vector, initialized with the same components as those of the source. This function returns a garbage collected pointer:

```
struct ktuple_t*
ruk_dup_vector(struct ktuple_t* volatile vec) { //
volatile arguments!
```

It is mandatory that all garbage collected argument pointers should be declared volatile as above, because the GC may move (i.e. change) such pointers. Also, all garbage collected argument pointers should be consecutive.

```
struct ktuple_t* volatile tup=0;
// local variable tup is also volatile
int kd=0; int len=0; int i=0;
```

We have one single local garbage collected pointer `tup` (which will hold the result of the function). Since it is alone we just declare it volatile and initialize it to 0. If we had several local GC-ed pointers (or even one of them) we would pack them inside a volatile structure, conventionally named `_locals_` and also initialized to all 0. It is important that all local GC-ed pointers variables are volatile, consecutive, and initialized to 0 (or a valid GC-ed pointer). It is a good habit to initialize every local variables, even the plain `int` ones.

```
// mandatory start of GC-ed frame
BEGIN_SIMPLE_FRAME(1, vec, 1, tup);
// we have 1 garbage collected argument starting at vec
// we have 1 GC-ed local pointer starting at tup
```

The call to the (deprecated `BEGIN_SIMPLE_FRAME` macro, or) `BEGIN_LOCAL_FRAME` when using the `_locals_` structure) is mandatory. It indicates to the GC the number of garbage collected pointer arguments, and the first such argument, and the number of garbage collected local pointers, and the first such pointer. The expanded C code registers a GC frame (in a linked-list of frames handled by the GC) and executes in a small constant time. The `BEGIN_SIMPLE_FRAME` macro call should be the first executable statement (after initialization of variables to constants such as pointer 0) of the body of any function using the GC.

Actually it is strongly suggested to always use a `_locals_` structure declared `volatile struct` for local garbage collected pointers. Actually do not use the deprecated `BEGIN_SIMPLE_FRAME` macro but only use the recommended `BEGIN_LOCAL_FRAME` (when you have pointers both in arguments and in `_locals_`), or `BEGIN_LOCAL_FRAME_WITHOUT_ARGS` (when you have pointers only in `_locals_`) or `BEGIN_FRAME_WITHOUT_LOCALS` (when you have only pointers in arguments).

Then we compute the kind of the source vector. We only duplicate vectors and tuples. To duplicate, we compute the length of the source (taken from its header).

```
kd = ruk_kind(vec);
if (kd == KIND_VECTOR || kd == KIND_TUPLE) {
len = HEADERSIZE(vec->header);
```

Once the length is computed, we allocate the resulting `tup` with a call to `qish_allocate`. This call may trigger a garbage collection, which may change (by moving them) some or even all garbage collected pointers, including the current frame local pointers or argument pointers. With compilers without inlining, you might consider using the `QISH_ALLOCATE` macro (invoke it without any side-effects in arguments, eg `QISH_ALLOCATE(--p,sz++);` is incorrect).

More generally, you inhibit all small inline functions by compiling your application with the `-DNO_QISH_INLINE` flag. But then you have to use macros instead of functions.

```
tup = qish_allocate(sizeof(*tup) + len*sizeof(void*));
//a garbage collection may occur above, changing many pointers
```

The `qish_allocate` function returns a zeroed chunk of memory. We have to initialize it by filling all relevant fields:

```
tup->header = MAKEHEADER(KIND_VECTOR, len);
for (i=0; i<len; i++) tup->tab[i] = vec->tab[i];
} // end if kind is KIND_VECTOR or KIND_TUPLE
```

Each function should end with the macro call to `EXIT_FRAME`, which pops the frame registered by (the deprecated `BEGIN_SIMPLE_FRAME`) or `BEGIN_LOCAL_FRAME`. This macro should only be followed by a simple `return` statement, which returns a constant or a simple variable:

```
EXIT_FRAME();
return tup;
} // end of ruk_dup_vector
```

The preferred coding style is to have a `_locals_` structure, which should be **volatile** and initially cleared!

```
// preferred coding
void* foo(struct ktuple_t* volatile tup,
           void* volatile val) {
// declare and clear a volatile _locals_ structure for local pointers
volatile struct { void* ptr; void* res; } _locals_
= {0,0};
BEGIN_LOCAL_FRAME(2,tup);
//... use _locals_fields as the only GC pointer local variables
if (some_condition(tup))
_locals_.res = val;
```

```
//...
EXIT_FRAME();
return _locals_.res;
}
```

This idiom is so common that a tiny Ruby script `scripts/gen_locals` exist to generate macro definitions like

```
// generated by gen_locals script
#define l_res _locals_.res
```

The script generate such definitions for every different occurrence of `l_*` names in the source. So just code `l_res`, run the script (with the `.c` source as first argument and the generated `.h` as second argument), and include its output near the start of your file.

A function which changes a garbage collected object by updating a pointer field in it should notify the garbage collector, for example:

```
// this function set the component of a vector returning its previous
value
void*
ruk_vector_set(struct ktuple_t* volatile tup, void*
volatile val, int rk) {
int sz=0;
void* oldval=0;
BEGIN_SIMPLE_FRAME (2, tup, 1, oldval);
if (ruk_kind(tup) != KIND_VECTOR) goto end;
sz = HEADERSIZE(tup->header);
if (rk<0 || rk>=sz) goto end;
oldval = tup->tab[rk];
tup->tab[rk] = val;
qish_write_notify(tup); // notification of a changed ob-
ject
end:
EXIT_FRAME();
return oldval;
}
```

Note that every function which may use directly or indirectly the GC should follow the same coding rules (detailed below). Using the GC means either allocating new objects or calling functions which may use the GC. Following these rules is ok even for other functions, when in doubt always follow them.

The macro `QISH_WRITE_NOTIFY` is equivalent to the `qish_write_notify` function, provided this macro is invoked without side-effects.

The application has to follow some (very liberal) rules regarding data structures.

3.2 Data structure of objects

Moved objects should all start with a common prefix (i.e. a word). *Their first word should never be zero*[12], so it can be a non-nil pointer [13] (garbage collected or not) or a header word. All objects should be at least two words long. The GC do not need any additional word for moved objects.

Objects should know their size (the GC do not manage by itself the objects' size) and their data type.

For example, one could start every object with an `unsigned header` whose topmost byte is a non-zero kind number and whost 3 lower bytes encode some sizing information (dependent on the kind).

Your application has to define all its objects types and data representations, provided they start with a never-zero word or pointer.

As a simple toy example (nearly meaningless, only for illustrative purposes), suppose you are coding an integer calculator with formal variables. Then each object can start with two half-words, a kind and a size or code. Objects can be binary-operations, or variables (or tagged integer).

```
enum {
  KIND_NONE=0 /*unused*/,
  KIND_BINOP,
  KIND_VARIABLE
};
```

Binary operations in your calculator will be represented like:

```
  enum { OP_NONE, OP_ADD, OP_SUB, OP_MULT, OP_DIV };

  struct binop_st {
    short kind; /*always KIND_BINOP*/
    short opcode;
    void* left;
    void* right;
  };
```

[12] A zeroed first word indicates forwarded objects to the GC

[13] The pointer could be some kind of descriptor, or even a C++ vtable pointer, if you have a tree of classes sharing a common root class, with only single inheritance, and virtual methods.

Variables have a value (either a tagged integer or an binary operation) and a name; the size is the length of the name, and variables are objects of various size.

```
struct variable_st {
  short kind; /*always KIND_VARIABLE*/
  short namelen; /*length of name*/
  void* value; /*value of variable*/
  char name[1]; /*actually [namelen] bytes + final '\0' */
};
```

3.3 required coding practices

Our GC is not conservative[14], but exact. It has to know about every garbage collected pointer, and usually modify them (when copying alive objects).

3.3.1 No interior pointers

It is not allowed to have pointers to the inside of any garbage collected object. Every pointer should point to the start of such objects. (therefore, your C++ application cannot have multiple inheritance)

3.3.2 No multi-threading

Our GC does not support threading. If you dare use Posix threads, be careful that only one thread should use the GC.

3.3.3 Limited global data

You should have almost no global pointers (I believe having lots of global data is a bad practice). The only permissible exceptions are:

1. you can use the small (fixed size) [15] array of global pointers `qish_roots` as you wish, reading and writing in it any pointer to a GC-ed object (or the nil pointer, or a tagged integer).

[14]Conservative GCs [e.g. Boehm's] are much easier to use, but they might leak, may be slower on some applications, and do not compact memory. But Boehm's GC is not disruptive like Qish, and has an API compatible with (or similar to) `malloc`.

[15] The global roots is an array of `QISH_NB_ROOTS` pointers defined as 64 in `qish.h`. You could if needed change it to a rather small value (at most a few hundreds): at every garbage collection, the whole roots array is scanned and updated, so having lots (e.g. thousands) of such roots will not be reasonable.

2. Each module is described by an entry in `qish_moduletab`. Each such entry contain a rather constant pointer. You can set or change the constant of module to pointer p by calling `qish_changeconstant(`i`,` p`)`, and you can get this constant by `qish_constant(`i`)` or even accessing directly the `km_constant` field in the entry of `qish_moduletab`. In practice, you could make this constant point to a structure of GC-ed pointers.

3.3.4 Pointers are volatile

All pointers are volatile (because the GC silently moves them), in particular arguments should be volatile. You should compile with the `-fvolatile` and `-fvolatile-global` flags to the `gcc` compiler. Most importantly, you should explicitly declare volatile your formal arguments. So the following is incorrect

```
#error formal argument not declared volatile
void* foo(struct variable_st* var) { /*body*/ }
```

You should definitely code instead like this - notice that the `volatile` qualifier goes *after* the `*` indicating a pointer):

```
/* explicit volatile argument */
void* foo(struct variable_st* volatile var) { /*body*/ }
```

Omitting the `volatile` qualifier does produce hard to find bugs.

3.3.5 Indicate arguments and locals to the GC

Each function body [16] should start with a prologue and end with an epilogue. The prologue mark the current call frame (remembering the first argument, the first local pointer, and the number of arguments and of local pointers), and the epilogue reset the previous call frame. The prologue is
`BEGIN_SIMPLE_FRAME(`$nbparam$`,`$firstparam$`,`$nblocal$`,`$firstlocal$`);`
the epilogue is `EXIT_FRAME();`. The cpu time cost of the prologue or epilogue small, nearly constant, and independent of the numbers of locals or arguments. Actually, this `BEGIN_SIMPLE_FRAME` macro is deprecated. Use the `BEGIN_LOCAL_FRAME` macro (when you have parameters and a `_local_` struct), or `BEGIN_FRAME_WITHOUT_LOCALS` macro (when you have parameters but no local pointer), or `BEGIN_LOCAL_FRAME_WITHOUT_ARGS` macro (when you have a `_local_` struct without any pointer parameters).

[16] At least each function using directly or not the GC, i.e. either allocating memory with `qish_allocate` or calling -directly or indirectly- any function which itself does allocation

All local pointers should be explicitly initialized to 0 (or a simple value).

It is not permissible to return from a function without going thru `EXIT_FRAME();` the suggested coding convention is the have one single exit point, ie a single `EXIT_FRAME();` at the end followed by a `return` statement. If a function returns a garbage-collected value, it should be a local pointer.

For convenience, there is also a `BEGIN_LOCAL_FRAME(`$nbparam$`,`$firstparam$`)` macro, which assumes that a local variable named `_locals_` is defined as a structure containing only garbage-collected pointers. A ***Ruby*** script `gen_locals` is provided to generate (in a separate file to be included) for each variable named like `l_*`, e.g. `l_foo` a macro `#define l_foo _locals_.foo`; in practice, local GC-ed pointers eg `bar` should be declared as a pointer field in `_locals_` and referred as `l_bar`.

For convenience, there is a `BEGIN_FRAME_WITHOUT_LOCALS(`$nbparam$`,`$firstparam$`)` macro to be used when you don't have any local garbage-collected pointer (but only parameters). Symetrically, a `BEGIN_LOCAL_FRAME_WITHOUT_ARGS()` macro is provided, when you only have local garbage collected pointers inside your usual `_locals_` structure.

If a frame (therefore a function body) have no local pointers (or no pointer arguments) it can pass `qish_nil` (or any unused adress) as the appropriate argument to the `BEGIN_SIMPLE_FRAME` or `BEGIN_LOCAL_FRAME` macro. For example, a function without arguments and with `_locals_` encapsulated pointer variable should start with a `BEGIN_LOCAL_FRAME(0, qish_nil)`. Internally `qish_nil` is a pseudo-data whose address is the nil pointer.

Every local GC pointer should be explicitly initialized to a simple value (usually the null pointer). It is best to clear any pointers heavily used in a loop when exiting out of the loop.

You cannot use `longjmp` without special measures. If you want exceptions, use the `BEGIN_EXCEPT_BLOCK CATCH_EXCEPT_BLOCK END_EXCEPT_BLOCK THROW_EXCEPTION` macros from file `qish.h`

3.3.6 No composition of function calls

Since each pointer should be known to the GC (either as a argument or a local explicited thru `BEGIN_SIMPLE_FRAME` or `BEGIN_LOCAL_FRAME`, or as a global root or constant, or as a field in a garbage collected object) it is not permissible to call several functions, e.g.

```
#error no function composition
l_res = f(g(y),l_z->ptrfield);
```

but you should code thru a temporary value

```
/* use a temporary variable */
l_tmp = g(y);
l_res = f(l_tmp, l_z->ptrfield);
}
```

3.3.7 Complete allocation

Every allocation of a garbage collected object is done thru a call to one of the following functions (which may trigger a garbage collection):

- `qish_allocate(`*bytesize*`)` to allocate an ordinary (movable) object with a natural (word) alignement. This is the most often used allocation function; usually the *bytesize* is some `sizeof(type)`.
- `qish_allocate_aligned(`*bytesize*`,`*alignment*`)` to allocate a movable object with an explicit *alignment* expressed in bytes (which must be a small power of 2 in words).
- `qish_fixed_alloc(`*bytesize*`,`*finalizer*`)` to allocate a fixed finalized object (aligned to at least the `sizeof(double)`), with an optional finalising routine (called by the GC with the adress of the fixed object).

Once an object is allocated it can be (and should be) filled. The allocated object should be filled to become valid (for marking and scanning routines) before the next allocation. Memory provided by the above allocation routines is cleared to all-zero bytes.

It should be noted that in the usual case `qish_allocate` and `_allocate_aligned` are very quick and inlined routines (basically a pointer increment and a compare to the birth region limit) which occasionally triggers a garbage collection. Adventurous expert users could even allocate several object at one with a single call to `qish_allocate`, giving it the cumulated total size of all allocated objects.

Since object allocation is very quick (much faster than a call to `malloc`) it is expected that the application makes frequent allocation to short lived objects.

The macro `QISH_ALLOCATE` does the same as `qish_allocate` provided its invocation has no side-effects.

3.3.8 Notification of updates (write barrier)

Since the garbage collector has to track pointers from old to new generation, it should be aware of any updates of allocated objects. This is done by calling `qish_write_notify(`*objptr*`)` after changing the GC-ed pointers in the object *objptr* and before any further allocation or call.

It is not required to notify the GC after any non-pointer updates (e.g. adding bytes into a garbage collected string). It is better to code some useless calls to `qish_write_notify` than to forgot one.

A garbage collection can be triggered at each `qish_write_notify` points.

Explicit update notification favors a functional programming style (where updates are rare).

Assignement to local GC pointers, to arguments, and root variables do not require any notification.

3.3.9 Optional explicit garbage collection

The garbage collector can be explicitly called by the application with `qish_garbagecollect(`$size$`,`$fullflag$`)` where the $size$ is an estimation (which can be left as 0) of the needed size -in bytes- of future objects and the $fullflag$ is non-zero to force a full garbage collection (otherwise, a minor collection will be done, unless the old generation has grown significantly).

To be sure that at least $size$ bytes are allocatable without GC, you can call `qish_reserve(`$size$`)` which calls the garbage collector unless $size$ bytes are available in the birth zone. This is particularily useful in applications having garbage-collected type descriptors, which have to bootstrap and fill the type descriptors' descriptor without any garbage collection.

It could be interesting to trigger an explicit garbage collection once in a while in an idle loop, or before an important processing requiring lots of allocation or recursion. This is always an optimisation, and the GC will work without a single explicit call to `qish_garbagecollect` in the application code.

3.3.10 Exception handling

Exception handling has to cooperate with the GC, because of the frame linking mechanism. You cannot simply call `longjmp` or throw C++ exceptions[17]. Instead you have first to declare (in your application) one (or very few) `void*` global (or static) pointer variable e.g. `exv`. You also need a local integer variable `cod` holding a non-zero error or exception code (usable as you want, but never 0 if exception thrown, it is the result of `setjmp`) and a local pointer `_locals_.exobj` holding any exception (garbage-collected) object.

Then you surround any potentially exception-raising code (after the usual `BEGIN_LOCAL_FRAME`... or similar macro) with:

```
BEGIN_EXCEPT_BLOCK(exv);
// your code here may directly or indirectly throw "exceptions"
```

[17] If you use C++ exceptions, ensure (by coding tricky appropriate constructors and destructors) that the `EXIT_FRAME` is called on exceptional frame unwinding. You are on your own.

Then you code as usual, you can call some other routines which do allocation and may indirectly throw an "exception" (using the `THROW_EXCEPTION` macro detailed below). The `exv` argument to the `BEGIN_EXCEPT_BLOCK(exv);` macro (vaguely similar to the `try` keyword of Java or Ocaml) holds the adress of the exception-catching frame, and this `BEGIN_EXCEPT_BLOCK(exv);` opens a brace (so starts a C code block). Then you catch exceptions with

```
CATCH_EXCEPT_BLOCK(cod,_locals_.exobj);
// your code handle here exception of error cod ...
// ... with _locals_.exobj set to the exception object
```

So this `CATCH_EXCEPT_BLOCK` is vaguely similar to the `with` keyword of Java or Ocaml exception handlers. At last you have to end the exception hanling code with

```
// this ends the exception handling code
END_EXCEPT_BLOCK(exv);
```

The `void*` variable argument to `END_EXCEPT_BLOCK(exv);` should always be the same as the argument of the matching previous `BEGIN_EXCEPT_BLOCK(exv);`.

Inside the normal block enclosed with `BEGIN_EXCEPT_BLOCK` and `CATCH_EXCEPT_BLOCK` you can call functions (or even directly) which throws (directly or indirectly) an "exception" with `THROW_EXCEPTION(exv,`Cod `,` $Exob$ `).` The Cod should be a non-zero integer (an error code, passed as the second argument to `longjmp`) and the $Exob$ is the (garbage-collected) error object. If this "exception" throwing happens, control jumps to the `CATCH_EXCEPT_BLOCK`.

Of course, the `BEGIN_EXCEPT_BLOCK`, `CATCH_EXCEPT_BLOCK` and `END_EXCEPT_BLOCK` have to be in the same C code block (so the same function).

3.3.11 Utility routines

The Qish runtime provide some utility functions, in particular:

- `qish_strhash(`$string$`,`$length$`)` compute an hashcode of a given $string$ with an explicit $length$; if this length is negative, the $string$ is supposed null-terminated, as if $length == strlen(string)$
- `qish_sigexecvp(`$file$`,`$argv$`)` spawn a process to execute $file$ with the given program $argv$ null-terminated arguments and wait for its completion.
- `qish_prime_after(`i`)` returns a prime number bigger than i provided that $0 < i < 10000000 = 10^7$ which can be useful for hashtables, etc.

- `qish_parameter(`$name$`)` retrieves the string-value of a runtime parameter of a given $name$.
- `qish_put_parameter(`$name$`,`val`)` put into the runtime paramter named $name$ the string value val
- `qish_parse_configfile(`$filename$`)` parse a simple configuration file and set parameters appropriately
- `qishgc_init();` should be called once to initialize the garbage collector, before any allocation!
- `qish_load_module(`$modulename$`,`$rank$`)` loads (with `dlopen`) a shared object module at a given rank. It returns 0 iff ok. A previous module at the same rank is closed latter with `qish_postponed_dlclose`
- `qish_get_symbol(`$name$`,`$modrank$`)` gets the address of the symbol of given $name$ in a module of given $modrank$ or in any modules if $modrank < 0$
- `qish_postponed_dlclose()` close any previous module with `dlclose` and should be called when the call stack is very low (i.e. in your event loop)
- `qish_panic` is a printf-like macro which aborts after displaying a panic message.

3.4 Mandatory routines to the GC

Your application should provide four mandatory routines to the GC and store their address in global function pointer variables before any GC call. All such functions should be provided (give a dummy function if not needed). Since these functions are called by the GC, they should not follow the above coding guidelines (no `BEGIN_SIMPLE_FRAME` etc...) and should of course never allocate objects or call the GC.

3.4.1 Copying function `qish_gc_copy_p`

The function pointer `qish_gc_copy_p` should be set by your application to a routine which copy an object (into an address provided by the GC). Its prototype is `void*` gc_{copy} `(void**padr, void* dst, const void* src).` It should set `*padr` to the new adress of the copy (usually `dst` or some aligned word after) and should return the first word after the copied object.

For simple cases (word aligned structures) the copy routine can be as simple as e.g. a switch of cases like

```
/*after determining the dynamic object type*/
*((object_type*)dst) = *((object_type*)src);
 return ((object_type*)dst)+1;
```

after having determined the dynamic object type (e.g. thru its header).
The first word of the copied object should not be zero.

3.4.2 Minor scan of a movable object `qish_minor_scan_p`

The function pointer `qish_minor_scan_p` should be set by your application to a routine which scans for the minor garbage collection and object by updating each of its pointer fields with the `QISHGC_MINOR_UPDATE` macro (called with the pointer field). This routine should return the next word after the scanned object. Its prototype is `void*` $minor_{scan}$ `(void*ptr);` and it should return the first word after the scanned object at address `ptr`.

For simple cases (word aligned structures) the minor scan routine can be as simple as e.g. a switch of cases like

```
/*after determining the dynamic object type*/
QISHGC_MINOR_UPDATE(((object_type*)src)->ptrfield1);
QISHGC_MINOR_UPDATE(((object_type*)src)->ptrfield2);
/* etc for every field*/
 return ((object_type*)src)+1;
```

after having determined the dynamic object type (e.g. thru its header).

When you are sure that the pointer is a true pointer (ie is never a tagged integer) you can use `QISHGC_MINOR_PTR_UPDATE` instead of `QISHGC_MINOR_UPDATE`.

Qish accepts not only null pointers, but also any address inside the first page of address space (so on x86 any address below 0x1000). For instance you could mark emptied slots in hashtable specially by such an address (eg `(void*)16`).

3.4.3 Full scan of a movable object `qish_full_scan_p`

The function pointer `qish_full_scan_p` should be set by your application to a routine which scans for the full garbage collection and object by updating each of its pointer fields with the `QISHGC_FULL_UPDATE` macro (called with the pointer field). This routine should return the next word after the scanned object. Its prototype is `void*` $full_{scan}$ `(void*ptr);` and it should return the first word after the scanned object at address `ptr`.

For simple cases (word aligned structures) the full scan routine can be as simple as e.g. a switch of cases like

```
/*after determining the dynamic object type*/
QISHGC_FULL_UPDATE(((object_type*)src)->ptrfield1);
QISHGC_FULL_UPDATE(((object_type*)src)->ptrfield2);
/* etc for every field*/
 return ((object_type*)src)+1;
```

after having determined the dynamic object type (e.g. thru its header).

When you are sure that the pointer is a true pointer (ie is never a tagged integer) you can use `QISHGC_FULL_PTR_UPDATE` instead of `QISHGC_FULL_UPDATE`.

Usually the full scanner has the same code as the minor scanner, except for the update macros `QISHGC_FULL_UPDATE` instead of `QISHGC_MINOR_UPDATE`.

If the first word of your object is not a header but a garbage collected pointer -for instance the class pointer in a ObjVlisp-like object language (i.e. single inheritance language with classes reified as objects)-, be careful in your scanning GC routines to check that it is a pointer (you could use the `QISH_IS_MOVING_PTR(`ptr`)` macro), then update the pointer first (with `QISHGC_FULL_UPDATE` or `QISHGC_MINOR_UPDATE`), then use it appropriately (your reified class could contain a description of its fields, or scanning routine pointers, ...).

3.4.4 Full scan of a fixed object `qish_fixed_scan_p`

The function pointer `qish_fixed_scan_p` should be set by your application to a routine which scans for the full garbage collection and object by updating each of its pointer fields with the `QISHGC_FULL_UPDATE` macro (called with the pointer field). This routine returns void and knows about the fixed object size. Its prototype is `void` $fixed_{scan}$ `(void*ptr, int size);` where `ptr` is the address of the object and `size` is its size in bytes.

For simple cases (word aligned structures) the fixed scan routine can be as simple as e.g. a switch of cases like

```
/*after determining the dynamic object type*/
QISHGC_FULL_UPDATE(((object_type*)src)->ptrfield1);
QISHGC_FULL_UPDATE(((object_type*)src)->ptrfield2);
/* etc for every field*/
 return;
```

after having determined the dynamic object type (e.g. thru its header).

Even an application with only moving objects (and no finalized, fixed objects) should provide a dummy fixed scanner.

For the simple examples in section 3.2, the routines could be:

```
void* gc_copy(void**padr, void* dst, const void* src) {
  switch (*(short*)src) {
  case KIND_BINOP:
    *((struct binop_st*)dst) = *((struct binop_st*)src);
    return ((struct binop_st*)dst)+1;
  case KIND_VARIABLE:
    {  struct variable_st* srcvar = src;
       int srcnamlen = srcvar->namelen;
       memcpy(dst, src, sizeof(struct variable_st)+srcnamlen);
       if (srcnamlen & (sizeof(void*)-1)) {
         /*round up name length to word*/
         srcnamlen |= (sizeof(void*)-1); srcnamelen++;
       }
       return ((struct variable_st*)dst)->name + srcnamlen;
    }
  }
}

void* minor_scan(void*ptr) {
  switch (*(short*)ptr) {
  case KIND_BINOP:
    QISHGC_MINOR_UPDATE(((struct binop_st*)ptr)->left);
    QISHGC_MINOR_UPDATE(((struct binop_st*)ptr)->right);
    return ((struct binop_st*)ptr)+1;
  case KIND_VARIABLE:
    {  struct variable_st* var = ptr;
       int namlen = var->namelen;
       QISHGC_MINOR_UPDATE(var->value);
       if (namlen & (sizeof(void*)-1)) {
         /*round up name length to word*/
         namlen |= (sizeof(void*)-1); namelen++;
       }
       return var->name + srcnamlen;
    }
  }
}
```

The `full_scan` routine would be similar (using `QISHGC_FULL_UPDATE`).

Since there are no fixed object you have to provide a dummy fixed scanning routine which just calls `abort`

3.5 Using Qish in C++ code

I just give very few hints on using Qish in C++ code:

1. no implicit or explicit use of `this`. Since it is not possible to declare the argument `this` to be a volatile pointer, you should not use it (either explicitly as in `this->method(foo)` or `this->_field` or implicitly as in `method(foo)` or `_field`). Instead, copy `this` to a local pointer, using it explicitly. This stylistical constraint is annoying.

2. no multiple inheritance. Since Qish does not support interior pointer, you cannot use multiple inheritance.

3. Have a tree and not a forest of classes by defining a common superclass to all your GC-ed objects. The virtual table pointer is usually the first word of such objects.

I would suggest to avoid using C++ with Qish on new code. Instead consider using other programming languages like Ocaml, CommonLisp, Java.

3.6 Advanced application explicit forwarding.

Advanced applications using Qish (and understanding quite well its internal processing) may explicitly forward pointers in the application code. Use this (experimental) feature with caution.

Explicit forwarding could be useful when you want every pointer to a (garbage-collected and allocated) address α to be replaced by a fixed β. A typical application could be to grow existing values, dynamically change the class (hence the size) of an object, etc...

An application can for such an explicit forwarding with the macro call `QISH_EXPLICIT_FORWARD(`$alpha$`,`$beta$`)`. Then the garbage collector will replace any[18] pointer to $alpha$ than its scans with $beta$.

If an application use this explicit forwarding, it has to follow (explicitly in application code) any potential such pointer with the `QISH_FOLLOW_FORWARD` macro which should be applied to every pointer (either argument, or local after assignment). If the pointer is never a tagged integer, you may call `QISH_FOLLOW_FORWARD_PTR` directly. If you statically know that only some pointers may have been explicitly forwarded by `QISH_EXPLICIT_FORWARD` you mayy (at your own risk) call `QISH_FOLLOW_FORWARD` only for such pointers.

Refer to the `include/qish.h` file for definitions of these macros.

[18] So all occurrences are replaced only after a major full garbage collection; after a minor collection only some pointers are replaced!

Explicit application forwarding is an experimental untested feature. Use it at your own risk! If you happens to use it, be kind enough to explain me why.

4 A tiny benchmark

A tiny benchmark (adapted from the GCbench.c by H.Boehm, J.Ellics, P.Kovac, W.Clinger, et al) is ported to qish.

4.1 benchmark results

It is the file `GCBench.c` (where we changed the allocate sized to 4 times the original) in our `lib/` subdirectory (where you can run all 3 benches with `make OPTIMFLAGS='-O2 -DNDEBUG' clean lib bench`. The same file `GCBench.c` compiles with Qish GC, with Boehm's GC, and with manual malloc and free according to the setting of preprocessor flags `QISH` for Qish, `GC` for Boehm, and no flags for malloc and free. (Times have been measured with release 0.3 of Qish).

- Qish GC (standard birth size of 8Mbytes): CPU 9.160 user + 4.510 system = 13.670 total time (sec); done 131 minor and 16 full garbage collections
- Boehm's GC: CPU 18.570 user + 0.310 system = 18.880 total time (sec); Completed 42 collections
- explicit malloc and free: CPU 24.810 user + 2.810 system = 27.620 total time (sec)
- Qish GC with the birth size reduced to 4 Mbytes made with `make OPTIMFLAGS='-O2 -DNDEBUG -DMIN_BIRTH_SIZE=4194304' clean lib benchqish`: CPU 13.810 user + 6.350 system = 20.160 total time (sec); Qish done 257 minor and 39 full garbage collections
- Qish GC with the birth size increased to 16Mbytes: CPU 6.900 user + 4.350 system = 11.250 total time (sec) Qish done 67 minor and 6 full garbage collections
- Qish GC with the birth size increased to 32Mbytes (probably not significant, since it is similar to the live object size): CPU 5.920 user + 3.700 system = 9.620 total time (sec) Qish done 34 minor and 2 full garbage collections
- Boehm's GC with holes `make OPTIMFLAGS='-O2 -DNDEBUG -DHOLES' clean lib bench`, so for every used node allocated, an extra useless (dead) node is also allocated: CPU 43.220 user + 0.380 system = 43.600 total time (sec) Completed 72 collections

- Qish GC with holes (and standard birth size of 8Mbytes): CPU 12.490 user + 7.380 system = 19.870 total time (sec) Qish done 197 minor and 23 full garbage collections
- Qish GC with holes and increased birth size of 16Mbytes: CPU 9.560 user + 6.460 system = 16.020 total time (sec) Qish done 99 minor and 10 full garbage collections
- Qish GC with holes and small birth size of 4Mbytes: CPU 17.530 user + 8.500 system = 26.030 total time (sec) Qish done 389 minor and 52 full garbage collections

Obviously, this small benchmark does not prove much. But Qish is not too bad, even w.r.t. the famous Boehm's (et al.) conservative (and quite mature) garbage collector.

Explicit bug-prone manual memory management with the infamous `free` routine is not only harder to code, but seems even slower than all the other automatic memory management techniques.

Qish is highly sensitive to the birth size. This is expected (since a GC is trigerred only when the birth region is full). Qish requires lot of system calls, because it does `mmap`-ing at every GC. Perhaps we could improve it by caching memory zones... (but the GC requires zero-ed memory). Qish tuning can be done by carefully changing some compile-time constants (notably `MIN_BIRTH_SIZE MAX_BIRTH_SIZE FULL_GC_PERIOD`) at the start of file `lib/qigc.c`.

Qish works well with holes (because it compact them) and is designed with allocation of small short lived temporary objects in mind (which may favor some "functional" style of coding).

It would be very interesting to port Qish to an existing major GC-ed application (like guile, some application using Boehm's GC, or even emacs) but I have not enough time for this. I am willing to help any person which wants to do so.

Qish is designed to be used for C code generators.

4.2 Tuning Qish

To optimize Qish for your needs you could :

- use the usual trick of allocating several objects simultanously (in one single `qish_allocate` for all of them).
- explicitly invoke `qish_garbagecollect` when needed (e.g. at start of a topmost loop...).

- set the number `QISH_NB_ROOTS` of variable roots (ie the size of the `qish_roots` global array) in `include/qish.h` to your needs. Leave it at most to a few hundred or dozen.
- change the `MIN_BIRTH_SIZE` and `MAX_BIRTH_SIZE` and `FULL_GC_THRESHOLD` in `lib/qigc.c`. The minimal birth size is a sensitive number (8 megabytes by default). I believe it should be at least half a megabyte (and at most a fraction, e.g. the tenth, of your available RAM). The maximal birth size limits the maximal size of allocated objects.
- change the `QISH_MAXNBCONST` number (65536) to a power of two (at least 1024). It is the maximal number of global constant pointers (for `QISH_GLOBCONST`)
- if your plateform has enough registers (this is false for x86 with 6 usable registers only) you could (with GCC) reserve global registers for `qishgc_birth_cur` and `qishgc_birth_storeptr` global variables (in `include/qish.h`).

5 ToDo list (multi-threading?)

Some people expressed the wish of making Qish multi-threaded (using Posix threads ie `<pthread.h>`), in the sense of having a few threads[19] concurrently allocating garbage collected objects. This could be doable with a "stop the world" strategy: when a thread requires a (major) collection it has to stop all other threads, but minor collections eremains local to threads (and sending objects between threads requiring some special precautions). This approach works only for a few (at most a dozen) threads.

The problem with this approach is that the current allocation pointer `qishgc_birth_cur` has to become a thread-local variable. And there is no standard mechanism providing them very quickly: I believe that the standard `pthread_getspecific` function (which would have to be called at each allocation with `qish_allocate` - even those which do not trigger any collection) would significantly slow up this runtime.

I am not very fluent with multi-threading applications, and I don't have any biprocessor machine at home yet (experimentally a biprocessor machine is almost required to test multithreading applications).

People having access to 64 bits machines could try to compile Qish on these. Contributions are welcome!

Comments on this are welcome.

[19] If you need multi-threaded capable garbage collection, I suggest using Hans Boehm's collector.