Quire Documentation

Release 2.0-beta1

Paul Colomiets

Apr 05, 2017

Contents

Quire is a configuration parser-generator for the C language with the following most prominent features:

- Structured Yaml configuration files
- Generated header with C structures with apropriate typing
- Rich set of tools for end users (includes, yaml anchors, variables...)

Sources: http://github.com/tailhook/quire

Contents:

CHAPTER 1

Quire Tutorial

This tutorial assumes that reader is proficient with C and build tools, and have setup building as described in *Developer Guide*.

It is also assumed that you are familiar with YAML. There is quick *cheat sheet* in user guide.

Minimal Config

Let's make minimal configuration definition file, which is by coincidence is also a YAML file. By convention it's called `config.yaml`:

```
__meta__:
  program-name: frog
  default-config: /etc/frog.yaml
  description: The test program that is called frog just because it
    rhymes with prog (i.e. abbreviated "program")
```

Now let's build them to see what we have (we use `quire-tool` like it's installed in the system, you can use any *build tool* to build the files):

```
quire-tool --source config.yaml --c-header config.h --c-header config.c
```

Let's take a look at what we have in `config.h`:

```
/*  Main configuration structure  */
struct cfg_main {
    qu_config_head head;
};

/*  API  */
int cfg_load(struct cfg_main *cfg, int argc, char **argv);
void cfg_free(struct cfg_main *cfg);
```

Disclaimer

Generated code pieces are shown stripped and reformatted for teaching purposes

We don't see anything useful here yet. But let's make it work anyway. We need a `main.c`:

```
#include "config.h"

static void run_program(struct cfg_main *cfg) {
    printf("The test program is doing nothing right now!\n");
}

int main(int argc, char **argv) {
    int rc;
    struct cfg_main cfg;

    rc = cfg_load(&cfg, argc, argv);
    if(rc == 0) {
        run_program(&cfg);
    }
    cfg_free(&cfg);

    if(rc > 0) {
        /*  rc > 0 means we had some configuration error  */
        return rc;
    } else {
        /*  rc == 0 means we have run successfully  */
        /*  rc < 0 means we've done some config action successfully  */
        return 0;
    }
}
```

As you can see there is a tiny bit of boilerplate with handling error codes and freeing memory. Let's build it:

```
gcc main.c config.c -o frog -lquire -g
```

Let's see what batteries we have out of the box:

```
$ ./prog
Error parsing file /etc/frog.yaml: No such file or directory
```

Hm, we don't have a configuration file, yet. And we don't want to put configuration into `/etc` yet. Let's see what we can do:

```
$ ./prog --help
Usage:
    frog [-c CONFIG_PATH] [options]

The test program that is called frog just because it rhymes with prog (i.e.
abbreviated "program")

Configuration Options:
  -h,--help          Print this help
  -c,--config PATH   Configuration file name [default: /etc/frog.yaml]
  -D,--config-var NAME=VALUE
                     Set value of configuration variable NAME to VALUE
  -C,--config-check
```

```
                        Check configuration and exit
  -P                    Print configuration after reading, then exit. The
                        configuration printed by this option includes values
                        overriden from command-line. Double flag `-PP` prints
                        comments.
  --config-print TYPE
                        Print configuration file after reading. TYPE maybe
                        "current", "details", "example", "all", "full"
```

You can change path to configuration file, you can play with configuration checking and printing, you can put some variables into configuration (more below). And you get all of this for free.

So to run the command now, execute:

```
$ touch frog.yaml
$ ./frog -c frog.yaml
The test program is doing nothing right now!
```

Let's make it easier to test by picking up configuration file from current directory:

```
__meta__:
  ...
  default-config: frog.yaml
  ...
```

```
$ ./frog
The test program is doing nothing right now!
```

Adding Useful Stuff

Let's add some integer knob to our config:

```
jumps: !Int 3
```

After building we have the following header:

```
struct cfg_main {
    qu_config_head head;
    long jumps;
};
```

And we can now make advantage of this variable:

```
void run_program(struct cfg_main *cfg) {
    int i;
    for(i = 0; i < cfg->jumps; ++i) {
        printf("jump\n");
    }
}
```

Let's run and play with it a little bit:

```
$ ./frog
jump
jump
```

```
jump
$ echo "jumps: 4" > frog.yaml
$ ./frog
jump
jump
jump
jump
```

Note: I'm editing the file by shell command. It's probably too freaky way to do that. You can just edit the file, and see how changes are reflected.

The tutorial gives you an overview of what quire is able to parse and generate, for full list of types supported see *Developer Guide*.

Nested Structures

Now the interesting begins. You can make hierarchical config, configuration sections of arbitrary depth:

```
jumping:
  number: !Int 3
  distance: !Float 1
```

Yields:

```
struct cfg_main {
    qu_config_head head;
    struct {
        long number;
        double distance;
    } jumping;
};
```

In config it looks like:

```
jumping:
  number: 5
  distance: 2
```

Note: The presence of nested structures in quire doesn't mean that nesting too deep is encouraged. Probably the example above is better written as:

```
jumping-number: !Int 3
jumping-distance: !Float 1
```

Particularly, flat structure is more convenient for *merging* maps. So use nested structures sparingly.

Command-line Arguments

Many values can be controlled from the command-line. Let's return to the simpler example:

```
jumps: !Int 3
```

Command-line is enabled easily. First we should reformat our declaration, to equivalent one with mapping syntax:

```
jumps: !Int
  default: 3
```

Now we can add a command-line option:

```
jumps: !Int
  default: 3
  command-line: [-j, --jumps]
```

Let's see:

```
$ ./frog --help
Usage:
    frog [-c CONFIG_PATH] [options]

The test program that is called frog just because it rhymes with prog (i.e.
abbreviated "program")

Configuration Options:
  -h,--help         Print this help
  -c,--config PATH  Configuration file name [default: /etc/frog.yaml]
  -D,--config-var NAME=VALUE
                    Set value of configuration variable NAME to VALUE
  -C,--config-check
                    Check configuration and exit
  -P                Print configuration after reading, then exit. The
                    configuration printed by this option includes values
                    overriden from command-line. Double flag `-PP` prints
                    comments.
  --config-print TYPE
                    Print configuration file after reading. TYPE maybe
                    "current", "details", "example", "all", "full"

Options:
  -j,--jumps INT    Set "jumps"
$ ./frog
jump
jump
jump
$ ./frog -j 1
jump
$ ./frog --jumps=2
jump
jump
$ ./frog --ju 1
jump
```

For integer types there are increment and decrement arguments:

```
jumps: !Int
  default: 3
  command-line: [-j, --jumps]
  command-line-incr: --jump-incr
  command-line-decr: [-J,--jump-decr]
```

This works as following:

```
$ ./frog
jump
jump
jump
$ ./frog --jump-decr
jump
jump
$ ./frog -JJ
jump
$ ./frog -JJJ
$ ./frog --jump
Option error "--jump": Ambiguous option abbreviation
```

Note: Making command-line arguments is easy. However, too many command-line options makes `--help` output too long. There is another mechanism to expose configuration variables to the command-line: *variables*. Variables in quire are even more powerful, but somewhat less easy to use. At the end of the day, declare command-line arguments for options that either useful for almost every user, or should only be specified in the command-line.

Arrays

So far we have only declared simple options, that every configuration library, supports. But here is where the power of the quire comes. The arrays are declared like the following:

```
sounds: !Array
  element: !String
```

Here we declared array of strings. Here is how it looks like in C structure:

```
struct cfg_a_str {
    struct cfg_a_str *next;
    const char *val;
    int val_len;
};

struct cfg_main {
    qu_config_head head;
    struct cfg_a_str *sounds;
    struct cfg_a_str **sounds_tail;
    int sounds_len;
};
```

It's looks too ugly at the first glance. But the rules are:

1. The array is a linked list
2. The type of list element is named `cfg_a_TYPENAME`
3. The head of the linked list is named as variable in yaml
4. The tail may be ignored unless you want to insert another element
5. There is `_len`-suffixed element for the number of elements in array
6. The element of linked list is named `val` (suffixes work here too)

Ok, let's see how to use it in code:

```
struct cfg_a_str *el;
for(el = cfg->sounds; el; el = el->next) {
    printf("%s\n", el->val);
}
```

Now if we write following config:

```
sounds:
- croak
- ribbit
```

We can have a frog that can cry with both USA and UK slang :)

```
$ ./frog -c flog.yaml
croak
ribbit
```

You can also create nested arrays, and arrays of structures.

Mappings

We can also declare a mapping:

```
sounds: !Mapping
  key-element: !String
  value-element: !String
```

Here we declared mapping of string to string. Here is how it looks like in C structure:

```
struct cfg_m_str_str {
    struct cfg_m_str_str *next;
    const char *key;
    int key_len;
    const char *val;
    int val_len;
};

struct cfg_main {
    qu_config_head head;
    struct cfg_m_str_str *sounds;
    struct cfg_m_str_str **sounds_tail;
    int sounds_len;
};
```

The structure is very similar to array's one, but the element type is named `cfg_m_KEYTYPE_VALUETYPE`.

Ok, let's see how to use it in code:

```
struct cfg_a_str *el;
for(el = cfg->sounds; el; el = el->next) {
    printf("%s -- %s\n", el->key, el->val);
}
```

Now if we write following config:

```
sounds:
 gb: croak
 usa: ribbit
```

We can have a frog that can display both the slang and the text:

```
$ ./frog -c flog.yaml
gb -- croak
usa -- ribbit
```

Note: The mapping is represented by a linked list too. There is no hash table or other mapping structures that makes access by key fast. There are few reasons for this decision, the most imporant one is that most programs will copy the mapping into their own hash table implementation anyway.

Warning: The order of the elements in the linked list is preserved. But this shouldn't be relied upon, as the YAML spec doesn't guarantee that. For example some tool may rewrite yaml file and get keys reordered.

The `key-element` can be any scalar type (string, int, float...).

The `value-element` can be any type supported by quire, including nested arrays and mappings.

Custom Types

CHAPTER 2

Developer Guide

Build Process

This section discusses how to run quire-gen and how to use quire as the part of your applications, using different build systems.

Note: The ABI for the library is not stable so the recommended way of using quire is by using it as `git submodule` of your application.

Raw Process

Whole configuration parser generation is based on single YAML file. By convention it's called "config.yaml" and is put near the sources of the project (e.g. "src/" folder).

If you have installed quire to system, to make parser generator run:

> quire-gen –source config.yaml –c-header config.h –c-header config.c

Then you may use the files as normal C sources. But be careful to update them when yaml file changes. If you add them as a part of build process you may need the mark as "generated" or equal, so that build system would not error if they are absent. See below for instructions for specific build systems. You do *not* need to bundle original yaml file with distribution of your application.

This is it. See *tutorial* for examples of the yaml itself and how to use it in your own code.

Using Make

To use make for configuration file generation you might write something along the lines of:

```
config.h config.c: config.yaml
    quire-gen --source $^ --c-header config.h --c-source config.c
```

Using CMake and Git Submodule

If you are using cmake for building your project, you are lucky, because the developers of quire use cmake too. So the whole process is easy.

Let's add submodule first:

```
git submodule add git@github.com:tailhook/quire quire
```

Now we should add the following to the `CMakeLists.txt`:

```
# Assuming you have "exe_name" executable
# Add "config.c" that's will be generated to list of sources
ADD_EXECUTABLE(exe_name
    main.c
    other.c
    config.c)
# Builds quire itself
ADD_SUBDIRECTORY(quire)
# Get's the full path of quire-gen executable just built
GET_TARGET_PROPERTY(QUIRE_GEN quire-gen LOCATION)
# Adds target to build C files and headers
# You may need to adjust source and/or directory
ADD_CUSTOM_COMMAND(
    OUTPUT config.h config.c
    COMMAND ${QUIRE_GEN}
        --source ${CMAKE_CURRENT_SOURCE_DIR}/config.yaml
        --c-header config.h
        --c-source config.c
    DEPENDS config.yaml quire-gen
    )
# Marks files as generated so make/cmake doesn't complain they are absent
SET_SOURCE_FILES_PROPERTIES(
    config.h config.c
    PROPERTIES GENERATED 1
    )
# Add include search path for the files that include "config.h"
SET_SOURCE_FILES_PROPERTIES(
    main.c other.c  # These files need to be adjusted
    COMPILE_FLAGS "-I${CMAKE_CURRENT_BINARY_DIR}")
# Add include search path for quire.h (overriding system one if exists)
INCLUDE_DIRECTORIES(BEFORE SYSTEM quire/include)
# Add linkage, adjust "exe_name" to name of your executable
TARGET_LINK_LIBRARIES(exe_name quire)
```

Now just run `cmake && make` like you always do with cmake.

You also need to include folder `quire` to your source distributions, even if they have C files generated. You also need to add instructions to run `git submodule update --init` for building from git.

Variable Types

All variable declarations start with yaml tag (an string starting with exclamation mark). Almost any type can be declared in it's short form, as a (tagged) scalar:

```
val1: !Int 0
val2: !String hello
```

```
val3: !Bool yes
val4: !Float 1.5
val5: !Type some_type
```

And any type can be written in equivalent long form as a mapping:

```
val1: !Int
  default: 0
val2: !String
  default: hello
val3: !Bool
  default: yes
val4: !Float
  default: 1.5
val5: !Type
  type: some_type
```

Using the latter form adds more features to the type definition. Next section describes properties that can be used in any type, and following sections describe each type in detail.

Common Properties

The following properties can be used for any type, given the it's written in it's long form (in form of mapping). Here is a list (string is for the sake of example, any type could be used):

```
val: !String
  description: This value is something that is set in config
  default: nothing-relevant
  example: something-cool
  only-command-line: no
  command-line:
    names: [-v, --val-set]
    group: Options
    metavar: STR
    descr: This option sets val
```

Let's take a closer look.

```
val: !String
  description: This value is something that is set in config
```

The description is displayed in the output of `--config-print` and `-PP` command-line options. It's reformatted to the 80 characters in width, on output. If set it's also used in command-line option description (`--help`) if not overriden in `command-line` section.

```
val: !String
  default: nothing-relevant
```

Set's default value for the property. It should be the same type as the target value.

```
val: !String
  example: something-cool
```

Set's the example value for the configuration variable. It's only output in `--config-print=example` and may be any piece of yaml. However it's recommended to obey same structure as a target value, as it may be enforced in the future. See description of `--config-print` for more information.

```
val: !String
  only-command-line: yes
```

This flag marks an option to be accepted from the command-line only. It is neither parsed in yaml file, nor printed using `--config-print`, but otherwise it is placed in the same place in configuration structure and respect same rules. If there is no `command-line` (see below) for this option, then a member of the structure is generated and default is set anyway.

The command-line may be specified in several ways. The simplest is:

```
val: !String
  command-line: -v
```

This adds single command-line option. Several options can be used too, mostly useful for having short and long options, but may be used for aliases too:

```
val: !String
  command-line: [-v, --val]
```

And full command-line specification is a mapping. Each property in a mapping is described in detail below.

```
val1: !String
  command-line:
    name: -v
    names: [-v, --val]
```

Either `name` or `names` may be specified, for the single option and multiple options respectively.

```
val1: !String
  command-line:
    group: Options
```

The group of the options in the `--help`. Doesn't have any semantic meaning just keeps list of options nice. By default all options are listed under group `Options`.

```
val1: !String
  command-line:
    metavar: STR
```

The metavar that's used in command-line description, e.g. `--val STR`. By default reasonably good type-specific name is used.

```
val1: !String
  command-line:
    descr: This option sets val
```

The description used in `--help`. If not set, the `description` in the option definition is used, if the latter is absent, some text similar to `Set "val"` is used instead.

There are also type-specific command-line actions:

```
intval: !Int
  command-line-incr: --incr
  command-line-decr: --decr
boolval: !Bool
  command-line-enable: --enable
  command-line-disable: --disable
```

They all obey pattern `command-line-ACTION`. Every such option may be specified by any ways that `command-line` can. However, they have the following difference:

- they inherit `group` from the `command-line` if specified
- they often have `metavar` useless
- they don't inherit `description` as it's usually misleading

String Type

String is most primitive data type. It accepts any YAML scalar and stores it's value as `const char *` along with it's length.

The simplest config:

```
val: !String
```

If you supply scalar, is stands for the default value:

```
val: !String default_value
```

Maximum specification for string is something like the following:

```
val: !String
  description: This value is something that is set in config
  default: default_value
  example: some example
  command-line:
    names: [-v, --val-set]
    group: Options
    metavar: STR
    descr: This option sets val
```

The fields in C structure look like the following:

```
const char *val;
int val_len;
```

Note that the string is both nul-terminated and has length in the structure.

> **Warning:** Technically it's possible that the string contain embedded nulls. In most cases this fact may be ignored. But do not rely on `val_len` be the length of the string after `strdup` or similar operation.

Integer Type

Unlike in C there is only one integer type in quire. And it's represented by `long` value in C.

The simplest config:

```
val: !Int
```

If you supply scalar, is stands for the default value:

```
val: !Int 10
```

The comprehensive specification for integer is something like the following:

```
val: !Int
  default: 1
  min: 0
  max: 10
  description: This value is something that is set in config
  example: 100
  command-line:
    names: [-v, --val-set]
    group: Options
    metavar: NUM
    descr: This option sets val
  command-line-incr:
    name: --incr
    group: Options
    descr: This option increments val
  command-line-decr:
    name: --decr
    group: Options
    descr: This option decrements val
```

The field in C structure look like the following:

```
long val;
```

The additinal keys represent minimum and maximum value for the integer:

```
val: !Int
  min: 0
  max: 10
```

Both values are inclusive. If user specifies bigger or smaller value either in configuration file or on command-line, error is printed and configuration rejected. If value overflows by using increments by command-line arguments (see below), the value is simply adjusted to the maximum or minimum value as appropriate.

The additional command-line actions:

```
command-line-incr: --incr
command-line-decr: --decr
```

May be used to increment the value in the configuration. They are applied after parsing the configuration file, and *set*-style options (regardless of the order of the command-line options). Mostly useful for log-level or similar things. The value printed using `--config-print` option includes all incr/decr arguments applied.

All integer values support parsing different *bases* (e.g. `0xA1` for hexadecimal 161) and *units* (e.g. `1M` for one million)

Boolean Type

The simplest boolean:

```
val: !Bool
```

If you supply scalar, is stands for the default value:

```
val: !Bool yes
```

The comprehensive specification for boolean is something like the following:

```
val: !Bool
  default: no
  description: This value is something that is set in config
  example: true
  command-line:
    names: [-v, --val-set]
    group: Options
    metavar: BOOL
    descr: This option sets val
  command-line-enable:
    name: --yes
    group: Options
    descr: This option sets val to true
  command-line-disable:
    name: --no
    group: Options
    descr: This option sets val to false
```

The field in C structure look like the following:

```
int val;
```

The value of `val` is always either `0` or `1` which stands for boolean false and true respectively.

The additional command-line actions:

```
command-line-enable: --yes
command-line-disable: --no
```

May be used to enable/disable the value in the configuration. They are applied after parsing the configuration file, and after *set*-style options. If multiple enable/disable options used, the last one wins. The value printed using `--config-print` option includes all enable/disable arguments applied.

The following values may be used as booleans, both on the command-line and in configuration file. The values are case insensitive:

False	True
false	true
no	yes
n	y
~	
empty string	

Floating Point Type

The simplest config:

```
val: !Float
```

If you supply scalar, is stands for the default value:

```
val: !Float 1.5
```

The comprehensive specification for floating point is something like the following:

```
val: !Float
  default: 1.5
  description: This value is something that is set in config
  example: 2.5
  command-line:
    names: [-v, --val-set]
    group: Options
    metavar: FLOAT
    descr: This option sets val
```

The field in C structure look like the following:

```
double val;
```

All floating point values support parsing decimal numbers, optionally followed by `e` and a decimal exponent. Floating point values also support *units* (e.g. `1M` for one million). Note that fractional units are not supported yet.

Array Type

The array type has no short form, and is always written as a mapping. The only key required in the mapping is an `element` which denotes the type of item in each array element.

```
arr: !Array
  element: !Int
```

Any quire type may be the element of the array. Including array itself. More comprehensive example below:

```
arr: !Array
  description: Array of strings
  element: !String hello
  example: [hello, world]
```

Note: Command-line argument parsing is not supported neither for the array itself nor for any child of it. This may be improved in future. But look at *variables*, if you need some command-line customization.

The C structure for the array is a linked list:

```
struct cfg_a_str {
    struct cfg_a_str *next;
    const char *val;
    int val_len;
};

struct cfg_main {
    qu_config_head head;
    struct cfg_a_str *arr;
    struct cfg_a_str **arr_tail;
    int arr_len;
};
```

The example of array usage is given in *tutorial*.

Mapping Type

The mapping type has no short form, and is always written as a mapping. The two properties required in the mapping are `key-element` and `value-element` which denote the type of key and value for the mapping.

```
arr: !Mapping
  key-element: !Int
  value-element: !String
```

Any quire type may be the value element of the array. Including array itself. A key may be any *scalar* type. More comprehensive example below:

```
map: !Mappings
  description: A mapping of string to structure
  key-element: !String
  value-element: !String
  example:
     apple: fruit
     carrot vegetable
```

Note: Command-line argument parsing is not supported neither for the mapping itself nor for any child of it. This may be improved in future. But look at *variables*, if you need some command-line customization.

The C structure for the mapping is a linked list:

```
struct cfg_m_str_str {
    struct cfg_m_str_str *next;
    const char *key;
    int key_len;
    const char *val;
    int val_len;
};

struct cfg_main {
    qu_config_head head;
    struct cfg_m_str_str *map;
    struct cfg_m_str_str **map_tail;
    int map_len;
};
```

The example of mapping usage is given in *tutorial*.

Custom Type

Sometimes you want to reuse a part of the config in multiple places. You can do this with yaml aliases. But it's better to be done by declaring a custom type. Here we will describe only how to refer to a custom type. See *custom types* for a way to declare a type.

The simplest type reference is:

```
val: !Type type_name
```

As with most types, declaration may be expanded to a mapping:

```
val: !Type
  description: My Value
  type: type_name
  example: some data
```

Note: Neither command-line, nor default are supported for type reference for now. But this is expected to be improved in future

Special Keys

Types

The `__types__` defines the custom types that can be used in multiple places inside the configuration. It can also be used to define recursive types. Any type defined inside `__types__` can be referred by `!Type name_of_the_type`. See *custom types* for more info.

Conditionals

There is a common use case where you have several utilities sharing mostly same config with some deviations. The most typical use case is a daemon process and a command-line interface to it, with a different set of command-line argumemnts. Here is how it looks like:

```
__if__:defined CLIENT:
 query: !String
   only-command-line: yes
   command-line: --query
```

When compiling utility you should *define* the `CLIENT` macro:

```
gcc ... -DCLIENT
```

And you will get additional command-line arguments for this binary. In code it looks like:

```
struct cfg_main_t {
    int val1;
#if defined CLIENT
    const char *query;
    int query_len;
#endif  /* defined CLIENT */
}
```

The rule is: if expression is evaluated to true, you get the configuration with all the contents of conditional merged inside the mapping (i.e. conditional replaced by `<<:`). In case expression is evaluated to false, you should get the all the configuration structures and semantics as the key and all its contents doesn't exist at all.

You can use any expression that C preprocessor is able to evaluate instead of `defined CLIENT`

Warning: You must define the macro consistently across all C files that use configuration header (`config.h`). In particular you can't share `config.o` generated for the two executables having different definitions. CMake handles this case automatically but some other build systems don't.

Include

There is `__include__` special key, which allows to add `#include` directive to the generated configuration file header. This key can be present at any place and will add the preprocessor directive at the top of the file.

For example:

```
__include__: "types.h"
```

Will result into the following line in the `config.h` file:

```
#include "types.h"
```

Note: There is no way to include a system header (`#include <filename>`), you can include some intermediate file, which includes the system header, if you really need the functionality. But most of the time double-quoted name will be searched for in system folders if not found in the project itself.

Set Flags

The flag `__set_flags__` can be used to generate `xx_set` field for each of the structure field. This flag may be used to find out whether field is set by user or the default value is provided. For example:

```
data:
  ? __set_flags__
  a: !Int 1
  b: !Int 2
```

Note: The syntax `? __set_flags__` is YAML shortcut to `__set_flags__: null`. We use and recommend this syntax for structure flags as it's not only shorter, but also stand out from structure field definitions.

Will turn into the following structure:

```
struct cfg_main {
  qu_config_head head;
  struct {
     unsigned int a_set:1;
     unsigned int b_set:1;
     long a;
     long b;
  } data;
};
```

Note: The syntax `int yy:1;` is a syntax for bit field. I.e. the field that is only one bit in width. Given it is unsigned it can have one of the two values `0` and `1`.

The values of `a` and `b` fields will always be intitialized (to 1 and 2 respectively), but the `a_set` and `b_set` will be non-zero only when user specified them in configuration file.

The `__set_flags__` property can be specified in any structure, including the root structure and `!Struct` custom type or its descendent. The flag is propagated to the nested structures but not to the `!Type` fields.

Structure Name

Usually nested mappings that do not denoted by `!Type` are represented by anonymous structures. But you can set `__name__` for the structure to have a name.

```
data:
  __name__: data
  a: !Int 1
  b: !Int 2
```

Will name the internal structure:

```
struct cfg_main {
  qu_config_head head;
  struct cfg_data {
     unsigned int a_set:1;
     unsigned int b_set:1;
     long a;
     long b;
  } data;
};
```

This is occasionally useful to use the structures in code.

Note: Author of config is responsible to set unique name of the structure otherwise the C compiler will throw an error.

Custom Types

Structure Type

Choice Type

Enumeration Type

Tagged Scalar Type

Field Type

Field type allows to wrap any other type into yet another C structure. It is sometimes useful, especially with non-scalar types. For example:

```
__types__:
  string_list: !Field
    field: !Array
      element: !String
```

Results into the following C definitions:

```
struct cfg_string_list {
    struct cfg_a_str *val;
    struct cfg_a_str **val_tail;
```

```
    int val_len;
};
struct cfg_main {
    qu_config_head head;
    struct cfg_string_list arr1;
};
```

C Fields

> **Warning:** The functionality described in this section is currently discouraged and is subject to removing/adjusting at any time.

Ocasionally there is a need to put custom C field into generated structure. You can do that with the following syntax:

```
_field-name: !CDecl struct some_c_struct
```

Where `_field-name` may be arbitrary but must start with underscore. And at the right of the `!CDecl` may be any C type that compiler is able to understand. It's written as is, so may potentially produce broken header if some garbage is written instead of the type name.

If you need to add some header for type to be known to the compiler use `__include__` special key:

```
__include__: "types.h"
_field-name: !CDecl struct some_c_struct
```

Note all files are added with `#include "filename"` syntax, *not* the `#include <filename>`.

CHAPTER 3

User Guide

Yaml Cheat Sheet

Usually YAML structure is denoted by indentation.

Quire Tricks

Underscore Names

Integers

Integers can be of base 10, just like everybody used to. It can also start with `0x` to be interpreted as base 16, and if it starts with zero it is interpreted as an octal number.

Units

A lot of integer values in configuration files are quite big, e.g. should be expressed in megabytes or gigabytes. Instead of common case of making default units of megabytes or any other arbitrary choice, quire allows to specify order of magnitude units for every integer and floating point value. E.g:

```
int1: 1M
int2: 2k
int3: 2ki
```

Results into the following, after parsing:

```
int1: 1000000
int2: 2000
int3: 2048
```

Note that there is a difference between prefixes for powers of 1024 and powers of the 1000.

The following table summarizes all units supported:

Unit	Value
k	1000
ki	1024
M	1000000
Mi	1048576
G	1000000000
Gi	1073741824
T	1000000000000
Ti	1099511627776
P	1000000000000000
Pi	1125899906842624
E	1000000000000000000
Ei	1152921504606846976

Variables

YAML has a notion of anchors. You can anchor the node with ampersand `&`, and then alias it's value with star `*`. Here is an example:

```
var1: &amp some_value
var2: *amp
```

When encountering the code above, the parser sees:

```
var1: some_value
var2: some_value
```

It's very powerful and very useful thing. You can even anchor entire hierarchy:

```
map1: &a
  key1: value1
  key2: value2
map2: *a
```

Yields:

```
map1:
  key1: value1
  key2: value2
map2:
  key1: value1
  key2: value2
```

This is powerful for keeping yourself from writing too much code. But it only allows to substitute the whole yaml node. So there is more powerful scalar expansion:

```
var1: &var some_value
var2: $var
```

Note we replaced the aliasing using star `*` with dollar sign `$`. This doesn't look more powerful. But now we can override the value from the command line:

```
./myprog -Dvar=another_value
```

Which yields:

```
var1: some_value
var2: another_value
```

You can also substitute a part of the string:

```
_target: &target world
var1: hello $target
```

Let's play with it a bit:

```
$ ./myprog -f test.yaml -P
var1: hello world
$ ./myprog -f test.yaml -Dtarget=foo -P
var1: hello foo
```

There are two things interesting above:

1. Anchors and scalar variables are somewhat interchangable
2. Command-line variables override anchors. So latter may be used as default values

Note using underscored names for declaring variables. It's described in *quire tricks*.

There is even more powerful form of variable expansion:

```
_n: &n 100
int1: ${2*n}k
```

This leverages several features. Let's see the result:

```
int1: 200000
```

Few comments:

1. The `${...}` expands an expression not just single variable
2. The variable is referenced without dollar `$` inside the expression
3. The result of substitution is parsed using same rules as plain scalar, so may use *units* as well.

Note: You can't use variables when declaring mapping key. The only case where you can is inside a *Template*.

Templates

Even more powerful construction in a combination with variables is a template. Template is basically an anchored node which has some variable references, and may be used with different variable values in different contexts. For example:

```
_tpl: &price !NoVars
  chair: $x dollars
  table: ${x*4} dollars
```

```yaml
shops:
  cheap: !Template:price
    x: 50
  expensive: !Template:price
    x: 150
```

The example above will be expanded as the following:

```yaml
shops:
  cheap:
    chair: 50 dollars
    table: 200 dollars
  expensive:
    chair: 150 dollars
    table: 600 dollars
```

The templates may be arbitrarily complex. There are few limitations:

1. Template-scoped variables may only be scalar
2. The anchored node is expanded too, you may either use `!NoVars` like in example, or define all the variables to get rid of warnings of `Undefined variable`
3. Variables in tags are not supported

Note, the limitation #1, doesn't limit you to use anchor or templates inside a template (the anchored node), just the scoped variables inside the template invocation (the items of a mapping tagged `!Template`) must be scalar. And anchors are never scoped.

Note: The variable expansion in mapping keys work *only* for template, but doesn't work in all other cases.

Includes

All includes have common structure. They are denoted by tagged scalar, with the special tag. With the scalar being the path/filename to include. After parsing the yaml but before converting the data into configuration file structure, the node is replaced by the actual contents of the file(s).

Few more properties that are common for all include directives:

- All paths are relative to the configuration file name which contains the include directive (in fact relative to the name under which file is opened in case it symlinked into multiple places)
- Include directives can be arbirarily nested (up to the memory limit)
- File inclusion is logical not textual, so (a) each file must be a full valid YAML file (with the anchors exception described below), and (b) the included data is contained at the place where directive is (unlike many other configuration systems where inclusion usually occurs at the top level of the config), but you can include at the top level of the config too
- Variable references are not parsed in include file names yet, but it's on todo list, so do not rely on include paths that contain dollar signs
- There is a common namespace for anchors and variables between parent and include files, but this behavior may be changed in future

Include Raw File Data

The `!FromFile` tag includes the contents of the file as a scalar value. For example if `somefile.txt` has the following contents:

```
line1
: line2
```

The following yaml:

```
text: !FromFile "somefile.txt"
```

Is equivalent to:

```
text: "line1\n: line2"
```

The context of the file is not parsed. And it's the only way to include binary data in configuration at the moment.

Include Yaml

The `!Include` tag includes the contents of the file replaceing the node that contains tag. For example:

```
# config.yaml
items: !Include items.yaml
```

```
# items.yaml
- apple
- cherry
- banana
```

Is equivalent of:

```
items:
- apple
- cherry
- banana
```

Include Sequence of Yamls

The `!GlobSeq` tag includes the files matching a glob-like pattern, so that each file represents an entry in the sequence. Each included file is a valid YAML file.

The pattern is not full glob pattern (yet). It may contain only a single star and an arbitrary prefix and suffix.

For example:

```
# config.yaml
items: !GlobSeq fruits/*.yaml
```

```
# fruits/apple.yaml
name: apple
price: 1
```

```
# fruits/pear.yaml
name: pear
price: 2
```

Is equivalent of:

```
items:
- name: apple
  price: 1
- name: pear
  price: 2
```

Note: The entries are unsorted, so you should not use the `!GlobSeq` in places sensitive to positions for items. You should use plain sequence with `!Include` for each item instead

This construction is particularly powerful with *merge key* `<<`. For example:

```
# config.yaml
<<: !GlobSeq config/*.yaml
```

```
# config/basics.yaml
firstname: John
lastname: Smith
```

```
# config/location.yaml
country: UK
city: London
```

Is equivalent of:

```
firstname: John
lastname: Smith
country: UK
city: London
```

Multiple sets of files might be concatenated using *unpack operator*.

Include Mapping From Set of Files

The `!GlobMap` tag includes the files matching a glob-like pattern, so that each file represents an entry in the mapping. The key in the mapping is extracted from the part of the filename that is enclosed in parenthesis. Each included file is a valid YAML file.

The pattern is not full glob pattern (yet). It may contain only a single star and an arbitrary prefix and suffix. It must contain parenthesis and the star character must be between the parenthesis.

```
# config.yaml
items: !GlobSeq fruits/(*).yaml
```

```
# fruits/apple.yaml
title: Russian Apples
price: 1
```

```yaml
# fruits/pear.yaml
title: Sweet Pears
price: 2
```

Is equivalent of:

```yaml
items:
  apple:
    title: Russian Apples
    price: 1
  pear:
    title: Sweet Pears
    price: 2
```

You can also merge mappings from the multiple directories and do other crazy things using *merge operator* `<<`.

Merging Mappings

We use standard YAML way for merging mappings. It's achieved using `<<` key and either mapping or a list of mappings for the value.

The most useful merging is with aliases. Example:

```yaml
fruits: &fruits
  apple: yes
  banana: yes
food:
  bread: yes
  milk: yes
  <<: *fruits
```

Will be parsed as:

```yaml
fruits:
  apple: yes
  banana: yes
food:
  bread: yes
  milk: yes
  apple: yes
  banana: yes
```

Merging Sequences