nuqneH

1 Introduction

A common enough task when writing a programs, both large and small, is to parse arguments given on the command line. While not very hard, the code tends to be repetitive and tedious to write[1]. The common solution to this problem is either to have very rudimentary argument parsing or to use a library like getopt. The first solution, while simple, leaves a lot to be desired in terms of functionality and the second, while effective, still requires quite a bit of work.

1.1 The solution

So, what is this about then? What is the supposed magic bullet which will miraculously solve all command-line parameter problems? The magic bullet is the program and programming language[2] **nuqneH**. Well — in reality it does not solve all possible problems, but it does offer more flexibility with less effort than the other solutions mentioned.

The approach taken by nuqneH is to generate the source code for a parser to parse the arguments described in a nuqneH specification. This combines the ease of specifying the parameters with a template with the convenience of having every argument directly available without extra functions or type conversions.

1.2 Other benefits

In addition to creating argument parsers, nuqneH also attempts to encourage documentation of parameters in two ways. The first is by having an easy way to write parameter descriptions — the benefit of descriptions must be greater than the effort going into writing then. The second way nuqneH encourages documentation is by generation of (optional) external documentation in the form of LaTeXor `HTML`.

Recording of parameters (to a file) is built in and the generated file is human readable. This allows for easy logging of parameters when running a program multiple times and to recreate a run, the parameters can also be read back for use. If a similar, but not identical, invocation is desired, any parameters specified on the command line will override the ones specified in the file.

2 Specification

A nuqneH specification describes a number of command line arguments by their name, type, description and constraints. It takes the form of a list of sections followed by the keyword “finished”, which denotes the end of the specification.

Each section begins with the keyword “section” followed by a section type and is concluded by the keyword end. Except for the “args” section, which defines the parameters and has to come first, the order of the sections is arbitrary. Sections can also be specified several times, in which case nuqneH will use them

[1] Also known as “boilerplate” code.

[2] Much in the same way that **make** is both a program and a language.

all (so you can specify the same attribute for different parameters in different sections of the same type). The available section types are:

options A list of global options for the generated parser.

args A list of arguments and their types.

alias A list of argument aliases (alternate names, e.g. "f" instead of "file").

desc A list of descriptions for the arguments.

default A list of default values for the arguments.

required A list of all required arguments.

validate A list of tests for value-validation of parameters.

2.1 Sample Specification

This simple parser specification is for a program which takes two arguments. One argument which is a filename and one which is a type. The supposed program prints out the length of a file in different units.

```
section args
   file=string
   units={bytes,kb,Mb,Gb}
end

section default
   units=kb
end

section desc
   file="A filename."
   units="Units to display the file size in."
end

section required
   file
end

finished
```

The parser will define new variables and macros in the program which uses it. The sample parser will define four variables[3]; *file*, *file_test*, *units* and *units_test*.

The *file_test* and *units_test* variables can be used to test if an argument was specified on the command line and the *file* and *units* variables contain either the values specified or the default values if nothing was specified. The types of the variables depend on their type in the parser specification and e.g. `string` translates to `char *`. The units variable has enumeration type and the enumeration values have the names `OPT_xxx` where `xxx` is one of `bytes`, `kb`, `Mb` or `Gb` (e.g. `OPT_Mb`).

[3]These are actually macros, but you can usually treat them as variables.

The **desc** section is used when generating the usage (help) message for the program. Running the program with the argument "-h" or "–help" will display the following:

```
-file <string>
     (Required argument)
   A filename.
-units <x> where <x>={bytes|kb|Mb|Gb}
     (Default: kb)
   Units to display the file size in.
```

As can be seen, nuqneH attempts to generate a helpful and readable message with the information it has available. This both simplifies for the programmer who will not have to specify things twice just to get the help message to display them and for the user who will have access to a description of every argument.

3 Generating a parser

When generating a parser for our program, the first thing is to write a specification. The specification can either be written in a separate file like the previous example or it can be included directly in the source-code of the program using the parser.

Including a parser in a program is done by using a "trick" with comments. The following program to print "Hello" and a string illustrates how this works.

```
/* A simple example. nuqneH sees this as a comment. */
#include <stdio.h>
#include <nuqneH.h> /* nuqneH treats # as a one-line comment.

section args
   what=string
end

section default
   what="world"
end

finished

*/

int main(int cnt,char *arg[])
{
#include "muvtay.c"

printf("Hello %s!\n",what);

return(0);
}
```

Having our parser specification, we simply run the program nuqneH on the file and like this:

```
nuqneH hello.c
```

nuqneH will generate a file named "muvtay.c" which contains the argument parser. If the parser uses enumerations (like the first examples), a file named "muvtay.h" containing the `OPT_` definitions will also be generated.

If you find the name "muvtay" to be confusing, the output filename can be set with the "-o" option (the corresponding header file will be named after the source-file but with a ".h" suffix).:

```
nuqneH -o hello_parser.c hello.c
```

4 Using the parser in a program

While it is nice and well to have a parser, it is of no use unless it can be incorporated into our program. As the hello example in the previous section showed, the parser is simply included into the `main()` function of our program. This, however, is not quite enough — two more things need to be done. The first is to include the nuqneH header-file (`#include <nuqneH.h>`) and the second is to name the parameters to `main()` *cnt* and *arg*.

The first is to define support functions and macros (which made writing nuqneH a lot easier) and the second is to enable the parser to know where the arguments are stored. People who now complain that they do not want to name the arguments to `main()` *cnt* and *arg* have no valid reason to do so.

One thing to keep in mind is also that versions of C before ISO/IEC 9899:1999 (the C99 standard) does not allow variable declarations after code statements so the parser should be included *after* all variable declarations but *before* any code.

5 Reference

The following is a short reference listing the different sections, types and options.

5.1 Sections

Sections are the principal method of ordering data in nuqneH and all types of sections except the "options" type contain a list of arguments. Arguments are first declared with their type in the args section and attributes are added in the following sections.

5.1.1 options

A list of global options for the generated parser. See options below.

5.1.2 args

A list of arguments and their types (see types below). An argument marked with an asterisk ('*') at the end of the type-name is considered a default argument and the first value without an argument specifier will be assumed to be the value of this argument.

5.1.3 alias

A list of argument aliases. E.g.:

```
section alias
   file=f
   foo=bar
   frobnicate=q
end
```

Will make "f" and alias for "file", "bar" an alias for "foo" and "q" an alias for "frobnicate". Not all arguments need have aliases and only one alias per argument is allowed (For clarity. If you really need more aliases, create more arguments and write the glue code yourself or modify the generated parser.).

5.1.4 desc

A list of descriptions for the arguments. E.g.:

```
section desc
   file="The file to process."
   foo="foobar the file before processing."
   frobnicate="Frobnicate the file after processing."
end
```

To include an quote ('"') character, use a double double quote inside the string (e.g. `"""is quoted"" is quoted"`) The desc section can also be named "help".

5.1.5 default

A list of default values for the arguments. The default values are of the same type as specified in the args section.

5.1.6 required

A list of all required arguments. If any argument specified in a required section is left out of the progam invocation, the parser will terminate the program with a message specifying the missing arguments.

5.1.7 validate

A list of tests for value-validation of parameters. The tests will only be done if the parameters are actually specified. The test is a code fragment suitable for evaluation in an if-statement (in C). Example:

```
section validate
   port="port>0 && port<65536"
   count="count<=16"
end
```

In the test, the name of the argument being tested can be written as '$' — so `port="port>0 && port<65536"` could be written as `port="$>0 && $<65536"`. This is simply a convenience feature for the sake of brevity.

5.2 Types

When defining the argument to a program, there are several defined types to choose from. The set of types has been chosen to reflect the most common requirements, but could conceivably grow in the future. The types and their C-equivalents are:

5.2.1 int

Standard integer type. This corresponds to a "long int" in C and will typically be 32 bits.

5.2.2 maxint

Maximum length integer type. This corresponds to a "long long int" in C, but take care to note that this might not be supported by older C compilers. The ISO/IEC 9899:1999 (C99) standard defines this to be *at least* 64 bits.

5.2.3 float

A floating point type. This corresponds to a C "double".

5.2.4 string

A string, which corresponds to a "char *" in C.

5.2.5 bool

The bool type converts to the "int" type in C. This is for better compatibility with older compilers (in lieu of the boolean type introduced in C99).

5.2.6 enumerations

Enumerations are lists of items, the names of which are enclosed in curly braces ('{' and '}'). The enumeration values are converted into a C enum declaration but the prefix OPT_ is prepended to every name. Example:

```
section args
   colour={red,green,yellow}
end
```

Will result in an argument named colour of type enum. The values red, green and yellow are named `OPT_red`, `OPT_green` and `OPT_yellow`.

5.2.7 lists

A list, in nuqneH, is a comma-separated sequence of values. The values must all have the same type, but any type except bool or a list is valid as a base for lists. To define a list of some type, enclose the type in square brackets ('[' and ']'). Example:

```
section args
   ilist=[int]
   flist=[float]
   elist=[{green,orange,apple}]
end
```

In the example, ilist is a list of integers, which on the commandlist would be used like "`-ilist 3,1,4,1,5`". The C type is a pointer to the base type (so `[int]` will become long int * and `[string]` will become char **) and for every list, the parser will define the value XXX_count to hold the number of elements in XXX (e.g. `ilist_count`).

5.2.8 rest

The rest type is a specialised variation on the [string] type which uses space (' ') as a separator instead of comma (','). Its intended use is for shell-expanded patterns. Example:

```
section args
   files=rest
end
```

A program using this parser can be invoked as "prog -files *", whereupon the shell will expand the wildcard ('*') to a space-separated list of file-names. Note that no other arguments can come after a rest argument.

5.3 Options

The options section of a parser contains directives affecting the working of the generated parser. First, a small example.

```
section opts
   key
   singlefile
end
```

The currently supported options are:

5.3.1 key

The generated parser will parse options on the form <key>=<value> instead of -<key> <value> (e.g. "count=5" instead of "-count 5"). The parser will still recognise the built-in "-h" and "–help" options.

5.3.2 switch

The opposite of key (and the default).

5.3.3 ignore

Instructs the parser to ignore unrecognised arguments.

5.3.4 indirect

Do not define macros for the arguments. All arguments will have to be accessed as ARGS.VALUE._XXX where XXX is the name of the argument.

5.3.5 defaults

automatically generate the "read" and "write" arguments to enable loading and saving of argument sets.

5.3.6 selfmux

Prevent arguments from being specified more than once on the commandline.

5.3.7 singlefile

Do not generate a header file for enumeration values.

5.3.8 nohelp

Do not generate a help/usage text.

5.4 Leftovers

This is where all features I forgot to write about should have been.

6 The name

Finally, there is one more thing to consider. Where does the name "nuqneH" come from and why is it capitalised like it is? To make a short story even shorter, the word "nuqneH" means "What do you want?" in Klingon[4].

[4]The program itself can also be compiled to display all error messages in Klingon.