Declarative Command-line Interfaces

Damian Conway

School of Computer Science and Software Engineering
Monash University
Clayton 3168, Australia

mailto:damian@csse.monash.edu.au
http://www.csse.monash.edu.au/~damian

Abstract

This paper describes a new approach to generating command-line argument parsers in Perl [1]. The system presented takes a standard "usage" description and reverse-engineers a parser which satisfies that description. This ability to specify complex parsers declaratively also proves useful in other contexts, such as comma-separated-value processing, simple input parsing, and string interpolation.

Introduction

Non sunt multiplicanda entia praeter necessitatem.

- William of Occam

In apparent defiance of Occam's Razor, command-line argument parsing libraries multiply beyond all reasonable necessity. A 1994 survey [2] compares a dozen libraries for C/C++ alone, whilst the Comprehensive Perl Archive Network catalogues nine distinct Perl packages for the same purpose. Worse still, this paper describes `Getopt::Declare` - *yet another* command-line argument parser for Perl.

Command-line processing packages multiply because, despite its apparent simplicity, unrestricted command-line processing is a complex and specialized parsing task. Solutions may be optimized for execution speed, library size, flexibility, expressive power, level of automation, ease of use, or conciseness of specification, but not for all of these at once.

Thus the CPAN offers: small and fast (but unsophisticated) `Getopt::` packages; large and powerful (but harder-to-use) `Getopt::` packages; and middle-sized and easy-to-use (but restrictive) `Getopt::` packages. `Getopt::Declare` is targeted at still another niche in this multidimensional manifold, offering a large, powerful, flexible, easy-to-use, unrestrictive, highly-automated (but less concisely-specified and somewhat slower) package.

More significantly, `Getopt::Declare` represents a quite different *approach* to specifying the nature and meaning of command-line parameters. Most `Getopt::` packages take a list of the allowed parameters in some form, possibly annotated with corresponding parameter descriptions, or lists of subarguments, or other flags which control the command-line processing. In contrast, to use `Getopt::Declare`, the programmer simply specifies the complete "usage" string they wish to have implemented. `Getopt::Declare` then parses this specification and builds a command-line processor to match.

Thus, using the standard **`Getopt::Long`**, one might write:

```
GetOptions( 'foo|f=s',  \$foo,
            'bar=i',  \&proc,
            'ar=s@',  \@ar)
    or die;
print "foo = $foo\n";
print "ar  = ", @ar;
```

whereas, using `Getopt::Declare`, one would write:

```
$args = new Getopt::Declare q{
    -foo <str>       Peeking option
    -f <str>         [ditto]

    -bar <num:i>     Drinking option
            { proc($_PARAM_,$num) }

    -ar <str>...     Pirate option
                     [repeatable]
};
print "foo = $args->{-foo}\n";
print "ar  = ", @{$args->{-ar}};
```

which is considerably more verbose, but also much clearer and easier to get right. Note that the `Getopt::Declare` version also provides full automatic usage and version enquiry parameters (**`-h`** and **`-v`**, respectively) and detailed error messages.

Declarative command-line parsing.

As illustrated above, `Getopt::Declare` takes a declarative approach to command-line parsing. To parse the command-line in **`@ARGV`**, one simply creates a

Getopt::Declare object, by passing **Getopt::Declare::new()** a specification of the various parameters that may be encountered.

The specification is a single string in which the syntax of each parameter is declared, along with a description and (optionally) one or more actions to be performed when the parameter is encountered. The specification string may also include other usage formatting information (such as group headings or separators) as well as standard Perl comments (which are ignored).

Calling **Getopt::Declare::new()** parses the contents of the array **@ARGV**, extracting any arguments that match the parameters defined in the specification string, and storing the parsed values as hash elements within the new Getopt::Declare object being created.

Other features of the Getopt::Declare package include:

- The use of full Perl regular expressions to constrain matching of parameter components.
- Automatic generation of error, usage and version information.
- Optional conditional execution of embedded actions (i.e. only on successful parsing of the entire command-line)
- Strict or non-strict parsing (unrecognized command-line elements may either trigger an error or may simply be left in **@ARGV**)
- Declarative specification of various inter-parameter relationships (for example, two parameters may be declared mutually exclusive and this relationship will then be automatically enforced).
- Intelligent clustering of adjacent flags (for example: the command-line sequence: –a –b –c may be abbreviated to: –abc, unless there is also a **–abc** flag declared).
- Selective or global case-insensitivity of parameters.
- A number of pre-defined grammars (implementing, for example, **perl –s** and **awk** styles of command-line processing).
- The ability to parse files (especially configuration files) or strings, instead of the command-line.

Terminology

The terminology of command-line processing is often confusing, with various terms (such as "argument", "parameter", "option", "flag", "switch", etc.) frequently being used interchangeably and inconsistently in the documentation of the various Getopt:: packages available. In this paper the following terms will be used consistently:

"parameter" (or **"parameter specification"**)

A specification of a single entity which may appear in the command-line. Always includes at least one syntax (called a *parameter definition*) for the entity. Optionally may include other syntaxes (or *variants*), one or more descriptions of the entity, and/or an action to be performed when the entity is encountered. In this paper, parameters are presented in a **bold fixed-width** font.

"argument"

A substring of the command-line which matches some variant of a single parameter. Unlike some other Getopt:: packages, in Getopt::Declare an argument may be a single complete element of **@ARGV**, or just part of a single **@ARGV** element, or the concatenation of contiguous parts of several adjacent **@ARGV** elements. In this paper, arguments are presented in a sans-serif font.

"parameter flag" (or just **"flag"**)

A sequence of non-space characters which introduces a parameter. Traditionally a parameter flag begins with a *flag prefix* such as **–** or **––**, but Getopt::Declare allows any sequence of characters to be used as a flag.

"parameter variable"

A place-holder (within a parameter) for some value that will appear in any argument matching that parameter. For example, in the parameter: **–window <h> x <w>**, the components **<h>** and **<w>** are parameter variables.

"parameter punctuator" (or just **"punctuator"**)

A literal sequence of characters (within a parameter specification) which will appear in any argument matching that parameter. In the previous example, the literal **x** is a punctuator.

The command-line parsing process

Whenever a Getopt::Declare object is created, the current command-line is parsed sequentially, by attempting to match each parameter in the object's specification string against the current elements in the **@ARGV** array (but see "Parsing from other sources" below). The order in which parameters are tried against `@ARGV` is determined by three rules:

- Parameters with longer flags are tried first. Hence the command-line argument –quiet would be parsed as matching the parameter `-quiet` rather than the parameter `-q <string>`, even if the `-q` parameter was defined first.
- Parameter variants with the most components are tried first. Hence the argument –rand 12345 would be parsed as matching the parameter variant `-rand <seed>`, rather than the variant `-rand`, even if the "shorter" `-rand` variant was defined first.
- Otherwise, parameters are tried in the order they are defined in the specification.

Elements of `@ARGV` which do not match any defined parameter are collected during parsing and are eventually put back into `@ARGV` (but see "Strict and non-strict command-line parsing").

Specifying command-line parameters

Parameter definitions

In a `Getopt::Declare` specification, each parameter consists of three parts: the parameter definition, a textual description, and any actions to be performed when the parameter is matched.

The parameter definition consists of a leading flag or parameter variable, followed by any number of parameter variables or punctuators, optionally separated by spaces. The parameter definition is terminated by one or more tabs (at least one trailing tab *must* be present).

For example, all of the following are valid `Getopt::Declare` parameter definitions:

```
-v
in=<infile>
+range  <from>..<to>
--lines  <start>  -  <stop>
ignore  bad  lines
<outfile>
```

Note that each of the above examples has at least one trailing tab (even if you can't see it). Note too that this hodge-podge of parameter styles is certainly not recommended within a single program, but is shown so as to illustrate some of the range of parameter syntax conventions that `Getopt::Declare` supports.

The spaces between components of the parameter definition are optional, but significant. If two components are separated by a space in the definition, then there may be optional spaces at the same point in a matching argument. If there is no space between two components, then there may not be any space at the same point in a matching argument. Hence, as specified above, the `--lines` parameter would match any of the following:

```
--lines1-10      --lines 1-10      --lines 1 -10
--lines1 - 10    --lines 1- 10
```

whereas the `+range` parameter would match only +range1..10 or +range 1..10, since its definition implies that spaces are not permitted either side of the `..` punctuator.

Types of parameter variables

By default, a parameter variable will match a single blank-terminated or quote-delimited string. For example, the parameter definition: `-val <str>` would match any of the following the arguments:

```
-value              # $str = "ue";
-val  abcd          # $str = "abcd";
-val  "a value"     # $str = "a value";
```

It is also possible to restrict the types of values which may be matched by a given parameter variable. For example:

```
-limit  <l:n>     Set threshold to a
                  (numerical)  value

-count  <N:i>     Set count to <N>
                  (must  be  integer)
```

See "Parameter variable types" for details of this mechanism.

Parameter variables are treated as scalars by default, but this too can be altered. Any parameter variable immediately followed by an ellipsis (`...`) is treated as a list variable, and matches its specified type sequentially as many times as possible. For example, the parameter specification:

```
-pages  <pages:i>...
```

would match either of the following arguments:

```
-pages 1
-pages 1 2 7 20
```

Note that both scalar and list parameter variables "respect" the flags of other parameters, as well as their own trailing punctuators. For example, given the specifications:

```
-a
-b  <b_list>...
-c  <c_list>...  ;
```

The following argument lists will be parsed as indicated:

```
–b –d –e –a     # @b_list = ("-d","-e")
–b –d ;         # @b_list = ("-d",";")
–c –d ;         # @c_list = ("-d")
```

Optional parameter components

Except for the leading flag, any part of a parameter definition may be made optional by placing it in square brackets. For example:

```
+range <from> [..] [<to>]
```

which now matches any of:

```
+range 1..10        +range 1..
+range 1 10         +range 1
```

List parameter variables may also be made optional (the ellipsis must follow the parameter variable name immediately, so it goes *inside* the square brackets):

```
-list [<pages>...]
```

Two or more parameter components may be made jointly optional, by specifying them in the same pair of brackets. Optional components may also be nested. For example:

```
-range <from> [.. [<to>] ]
```

Scalar optional parameter variables (such as `[<to>]`) are given undefined values if they are skipped during a successful parameter match. List optional parameter variables (such as `[<page>...]`) are assigned an empty list if unmatched.

One important use for optional punctuators is to provide abbreviated versions of specific flags. For example:

```
-num[eric]

-lexic[ographic]al

-b[ells+]w[histles]
```

Note that the actual flags for these three parameters are **-num**, **-lexic** and **-b**, respectively.

Parameter descriptions

Providing a textual description for each parameter (or parameter variant) is optional, but strongly recommended. Apart from providing internal documentation, parameter descriptions form the basis of the automatically-generated usage information provided by `Getopt::Declare`.

Descriptions may be placed after the tab(s) following the parameter definition and may be continued on subsequent lines, so long as those lines do not contain any tabs after the first non-whitespace character (because any such line will instead be treated as a new parameter specification). The description is terminated by a blank line, an action specification (see "Actions") or another parameter specification.

For example:

```
-v                      Verbose mode

in=<infile>             Input file

+range <from>..<to>     Lines to be
                        considered

ignore bad lines        Ignore bad
                        lines :-)

<outfile>               Output file
```

The parameter description may also contain special directives which alter the way in which the parameter is parsed. These are described in later sections.

Actions

Embedded Actions

Each parameter specification may also include one or more blocks of Perl code, specified in a pair of curly brackets (which *must* start on a new line). For example:

```
-v          Verbose mode
                  { $::verbose = 1; }
-q          Quiet mode
                  { $::verbose = 0; }
```

Each action is executed as soon as the corresponding parameter is successfully matched in the command-line (but see "Deferred actions" for a means of delaying this response). Actions are executed (as "strict" **do** blocks) in the package in which the `Getopt::Declare` object containing them was created. In addition, each parameter variable belonging to the corresponding parameter is made available as a (block-scoped) Perl variable with the same name. For example:

```
-list <sect>          Sections to list
                    { setsect($sect) }

-list <page:i>...     Pages to list
                    { foreach (@page)
                          { list($_) }
                    }
```

Note that scalar parameter variables become scalar Perl variables, and list parameter variables become Perl arrays.

Termination and rejection

It is sometimes useful to be able to terminate command-line processing before all arguments have been parsed. To this end, `Getopt::Declare` provides a special local operator (**finish**) which may be used within actions. The **finish** operator takes an optional single argument. If that argument is true (or omitted), command-line processing is terminated at once (although the current parameter is still marked as having been successfully matched). For example:

```
--          Usual arg terminator
        { finish }

##          Alternate terminator
            (only valid Wednesdays)
        { finish((localtime)[6]==3) }
```

It is also possible to reject a successful parameter match from within its associated action (and then continue trying other candidates), by using the **reject** operator. This allows actions to be used to perform more sophisticated tests on the value of a parameter variable, or to implement complicated parameter interdependencies. The **reject** operator takes an optional argument and causes the current parameter to be immediately rejected if that argument is true (or omitted). For example:

```
-ar <R:n>       Set aspect ratio
             { $::sawaspect++;
               reject($R<=0 || $R>1);
               setaspect($R); }
```

Note that any actions performed *before* the call to **reject** will still have effect (for example, the variable **$::sawaspect** remains incremented even if the aspect ratio parameter is subsequently rejected).

The **reject** operator may also take a second parameter, which is used as an error message if the rejected argument subsequently fails to match any other parameter. For example:

```
-q          Quiet mode
        { reject $verbose => "Loud!";
          $::verbose = 0;  }
```

Deferred actions

It is often desirable or necessary to defer actions taken in response to particular arguments until the entire command-line has been parsed. The most obvious case is where command-line switches must be able to be specified *after* the arguments they modify.

To support this, `Getopt::Declare` provides a local operator (**defer**) which delays the execution of a particular action until the command-line processing is finished. The **defer** operator takes a single block, the execution of which is deferred until the command-line is fully and successfully parsed (the block is converted to a closure, which is stored and executed only when parsing is finished). If command-line processing *fails* for some reason, deferred blocks are never executed.

For example:

```
$args = Getopt::Declare q{
    <files>...  Files to process
           { defer {
                  foreach (@files) {
                         proc($_);
                  }
             }
           }
    -rev[erse]  Reverse order
    -rand[om]   Random order
};
```

With the above specification, the **-rev** and/or **-rand** flags can be specified *after* the list of files, but still affect the processing of those files (assuming that **proc()** consults **$args->{'-rev'}** and **$args->{'-rand'}**).

Parameter Variable Types

Specifying other parameter variable types

As was mentioned in "Type of parameter variables", parameter variables can be restricted to matching only numbers or only integers by using the type specifiers **:n** and **:i**. `Getopt::Declare` provides eight other inbuilt type specifiers, as well as two mechanisms for defining new restrictions on parameter variables. The other inbuilt type specifiers are:

- **:+i** which restricts a parameter variable to matching positive, non-zero integers.
- **:+n** which restricts a parameter variable to matching positive, non-zero numbers (integer or floating point).
- **:0+i** which restricts a parameter variable to matching non-negative integers.
- **:0+n** which restricts a parameter variable to matching non-negative numbers.
- **:id** which requires a parameter variable to match an identifier (that is, a string matching **/[A-Za-z_]\w*/**)
- **:s** which allows a parameter variable to match any quote-delimited or whitespace-terminated string. Note that this is the default behaviour.
- **:if** which is used to match input file names, and requires that the matched argument be either **-** (indicating standard input) or the name of a readable file.

:of which is used to match output file names. It is exactly like type **:if** except that it requires that the string be either **-** (indicating standard output) or the name of a file that is either writable or non-existent.

For example:

```
-repeat  <n:+i>       Once or more
-scale  <f:0+n>       Positive factor
-o  <file:of>         Writable file
```

Alternatively, parameter variables can be restricted to matching a specific regular expression, by providing the required pattern explicitly (in matched **'/'** delimiters after the colon). For example:

```
-parity  <p:/even|odd|both/>
-file  <name:/\w*\.[A-Z]{3}/>
```

Defining new parameter variable types

Explicit regular expressions are very powerful, but also cumbersome to use (or reuse) in some situations. `Getopt::Declare` provides a general "parameter variable type definition" mechanism to simplify such cases.

To declare a new parameter variable type, the **[type:...]** directive is used. A **[type...]** directive specifies the name, matching pattern, and action for the new parameter variable type (though both the pattern and action are optional).

The name string may be *any* whitespace-terminated sequence of characters which does not include a "**>**". The name may also be specified within a pair of quotation marks (single or double) or within any Perl quotelike operation. The pattern is used in initial matching of the parameter variable. Patterns are normally specified as a **'/'**-delimited Perl regular expression:

```
[type: num        /\d+/          ]
[type: q{nbr}      /\d+(\.\d*)/  ]
[type: "a num"    /[+-]?\d+/     ]
```

This definitions mean that the type **:num** now matches any string of digits, that the type **:nbr** matches a string of digits optionally followed by a decimal expansion, and the type **:a num** matches a signed string of digits.

Alternatively the pattern associated with a new type may be specified as a "**:**" followed by the name of another parameter variable type. In this case the new type matches the same pattern (and action! - see below) as the named type. For example:

```
[type: posnum   :+i ]
<v:posnum>
```

is the same as:

```
<v:+i>
```

As a third alternative, the pattern may be omitted altogether, in which case the new type matches whatever the inbuilt pattern **:s** matches.

The optional action which may be included in any **[type:...]** directive is executed *after* the corresponding parameter variable matches the command line but *before* any actions belonging to the enclosing parameter are executed. Typically, such type actions will call the **reject** operator (see "Termination and rejection") to test extra conditions, but any valid Perl code is acceptable. For example:

```
[type: num        /\d+/
    {  reject  {(localtime)[6]==3}}  ]
[type: 'a num'   :n
    { print "a num!" }               ]
[type: q{nbr}    :'a num'
    { reject {$::no_nbr} }           ]
```

If a new type is defined in terms of another (for example, **:a num** and **:nbr** above), any action specified by that new type is *prepended* to the action of that other type. Hence:

- the new type **:num** matches any string of digits, but then (silently) rejects the match on a Wednesday.
- the new type **:a num** matches any string of digits (like its parent type **:num)**, *then* prints out "a num!", *and then* rejects the match if it's Wednesday (like its parent type **:num**).
- the new type **:nbr** matches any string of digits (like its parent type **:a num**), but then rejects the match if the global **$::no_nbr** variable is true. Otherwise it next prints out "a num!" (like its parent type **:a num**), and finally rejects the match if it's Wednesday (like its grandparent type **:num**).

Parsing from other sources

`Getopt::Declare` normally parses the contents of **@ARGV**, but can be made to parse from other text sources. To accommodate this extra functionality, **Getopt::Declare::new()** takes an optional second parameter, which specifies the source to be parsed. The parameter may be either:

A FileHandle reference

in which case **Getopt::Declare::new()** reads the filehandle until end-of-file, and parses the resulting text (even if it is an empty string).

If the input is not successfully parsed, **undef** is returned.

The array reference `['-STDIN']`

in which case **Getopt::Declare::new()** parses data from the standard input stream.

The array reference `['-CONFIG']`

in which case **Getopt::Declare::new()** finds the files $ENV{HOME}/.${progname}rc and $ENV{PWD}/.${progname}rc, concatenates their contents, and parses that. If neither file is found (or if both are inaccessible) **Getopt::Declare::new()** immediately returns zero. If a file is found but the parse subsequently fails, **undef** is returned.

The array reference `['-BUILD']`

in which case **Getopt::Declare::new()** builds a parser from the supplied grammar and returns a reference to it, but does not attempt to parse anything at that point. See "The **Getopt::Declare::parse()** method".

The array reference `['-SKIP']` or `[]`

in which case **Getopt::Declare::new()** immediately returns zero.

A reference to any other array of strings

(for example: **["data1", "data2"]** or **[glob('data*')]**, in which case **Getopt::Declare::new()** treats each string in the array as a filename, concatenates the contents of those files, and parses the resultant string. In the case where none of the strings supplied denotes an accessible file, **Getopt::Declare::new()** immediately returns zero. If matching files are found, but not successfully parsed, **undef** is returned.

A subroutine reference

in which case the subroutine is called to generate text to be parsed. Each time the subroutine returns a defined value, that value is stringified and parsed. When the subroutine eventually returns an **undef**, parsing ceases.

A string

in which case **Getopt::Declare::new()** parses the string directly, returning **undef** if the parse fails.

Note that if any specified source corresponds to an interactive TTY (for example: ***STDIN** or **['-']** or **[-STDIN]** or **new IO::File('<-')**), then data from that source is read in (and parsed) line-by-line, after the processing of any other source files (see "Simple input handling" for an example).

Using `Getopt::Declare` objects

After command-line processing is completed, the object returned by **Getopt::Declare::new()** can be used to access parsed parameter data, or to issue usage or version information, or to do further processing.

Parameter data

For each successfully matched parameter, the `Getopt::Declare` object will contain a hash element. The key of that element will be the leading flag or parameter variable of the parameter. The value of the element will be a reference to another hash which contains the names and values of each distinct parameter variable and/or punctuator which was matched by the parameter. Punctuators generate string values containing the actual text matched. Scalar parameter variables generate scalar values. List parameter variables generate array references.

As a special case, if a parameter consists of a single parameter variable (optionally preceded by a flag), then the value for the corresponding hash key is not a hash reference, but the actual value matched.

For example, given the following specification:

```
$args = new Getopt::Declare q{
    -find <value> [exact]    Search
    <infile>                 Input
    -o <outfiles>...         Output
};
```

the object **$args** would have the following members (assuming that all parameters were matched):

`$args->{'-v'}{'<value>'}`

The argument matched by the **<value>** parameter variable of the **-v** parameter.

`$args->{'-v'}{'exact'}`

The argument (if any) matched by the optional **[exact]** punctuator of the **-v** parameter.

`$args->{'<infile>'}`

The argument matched by the **<infile>** parameter.

`$args->{'-o'}`

The argument matched by the **<outfile>** parameter variable of the **-o** parameter.

The values which are assigned to the various hash elements are copied from the corresponding blocked-scoped variables which are available within actions. In particular, if the value of any of those block-scoped variables is changed within an action, that changed value is saved in the hash. For example, given the specification:

```
$args = new Getopt::Declare q{
    ar = <R:n>  Set aspect ratio
                 { $R = 0 if $R<0;
                   $R = 1 if $R>1; }
};
```

then the value of `$args->{'ar'}{'<R>'}` will always be between zero and one.

The `@ARGV` array

In its default "non-strict" mode (see "Strict and non-strict command-line parsing"), once a `Getopt::Declare` object has completed its command-line processing, it pushes any unrecognized arguments back into the (now-emptied) command-line array **`@ARGV`.** Note that these remaining arguments will be in sequential elements (starting at **`$ARGV[0]`**), *not* in their original positions in **`@ARGV`**.

The `Getopt::Declare::usage()` method

Once a `Getopt::Declare` object is created, its **`usage()`** method may be called to explicitly print out usage information corresponding to the specification with which it was built. See "Usage information" for more details. If the **`usage()`** method is called with an argument, that argument is passed to **`exit`** after the usage information is printed (whereas the no-argument version of **`usage()`** simply returns at that point).

The `Getopt::Declare::version()` method

Another useful method of a `Getopt::Declare` object is **`version()`**, which prints out the name of the enclosing program, the last time it was modified, and the value of **`$::VERSION`** (if it is defined). Note that this implies that *all* `Getopt::Declare` objects in a single program will print out identical version information. Like the **`usage()`** method, if **`version()`** is passed an argument, it will exit with that value after printing.

The `Getopt::Declare::parse()` method

It is possible to separate the construction of a `Getopt::Declare` parser from the actual parsing it performs. If **`Getopt::Declare::new()`** is called with the second parameter **`['-BUILD']`** (see "Parsing from other sources", it constructs and returns a parser, without parsing anything. The resulting parser object can then be used to parse multiple sources, by calling its **`parse()`** method.

`Getopt::Declare::parse()` takes an optional parameter which specifies the source of the text to be parsed (it parses **`@ARGV`** if the parameter is omitted). This parameter takes the same set of values as the second parameter of **`Getopt::Declare::new()`**.

`Getopt::Declare::parse()` returns true if the source is located and parsed successfully. It returns a defined false (zero) if the source is not located. An **`undef`** is returned if the source is located, but not successfully parsed.

Thus, the following code first constructs parsers for a series of alternate configuration files and for the command line, and then parses them:

```
# BUILD PARSERS
my $cfg =
  Getopt::Declare::new($cfg_spec,
                       [-BUILD]);
my $args =
  Getopt::Declare::new($cmd_spec,
                       [-BUILD]);

# TRY STANDARD CONFIG FILES
$cfg->parse([-CONFIG])

# TRY GLOBAL CONFIG
or  $cfg->parse(['/usr/.cfgrc'])

# TRY OPENING A FILEHANDLE
or  $config->parse(
        new FileHandle (".cfgrc"));

# NOW PARSE THE COMMAND LINE
$args->parse()  or die;
```

Miscellaneous Features

Case-insensitive parameter matching

By default, a `Getopt::Declare` object parses the command-line in a *case-sensitive* manner. However, if a **`[nocase]`** directive is included in the description of a specific parameter variant, then that variant (only) will be matched without regard for case. If a **`[nocase]`** directive appears anywhere *outside* a parameter description, then the entire specification is declared case-insensitive and all parameters defined in that specification are matched without regard to case.

Undocumented parameters

If a parameter description is omitted, or consists entirely of whitespace, or contains the special directive **`[undocumented]`**, then the parameter is still parsed as normal, but will not appear in the automatically generated usage information (see "Usage information").

Apart from allowing for "secret" parameters (a dubious benefit), this feature enables the programmer to specify some (undocumented) action which is to be taken on encountering an otherwise unknown argument. For example:

```
<huh>     [undocumented] last resort
            { handle_unknown($huh); }
```

"Dittoed" parameters

Sometimes it is desirable to provide two or more alternate flags for the same behaviour (typically, a short form and a long form). To reduce the burden of specifying such pairs, the special directive **[ditto]** is provided. If the description of a parameter *begins* with a **[ditto]** directive, that directive is replaced with the description for the immediately preceding parameter (including any other directives). For example:

```
-v            Verbose mode
--verbose     [ditto] (long form )
```

Furthermore, if the "dittoed" parameter has no action(s) specified, the actions of the preceding parameter are reused. For example, the specification:

```
-v            Verbose mode
                 { $::verbose = 1; }
--verbose     [ditto]
```

would result in the **--verbose** option setting **$::verbose** just like the **-v** option. On the other hand, the specification:

```
-v            Verbose mode
                 { $::verbose = 1; }
--verbose     [ditto]
                 { $::verbose = 2; }
```

would give separate actions to each flag.

Flag clustering

Like various other `Getopt::` packages, `Getopt::Declare` allows parameter flags to be "clustered" or "bundled". That is, if two or more flags have the same *flag prefix* (one or more leading non-whitespace and non-alphanumeric characters), those flags may be concatenated behind a single copy of that prefix.

`Getopt::Declare` allows flag clustering at *any point* where the remainder of the command-line being processed starts with a non-whitespace character and where the remaining substring would not otherwise immediately match a parameter flag. This means that multiple-character flags can be clustered, as can flags with parameter variables and punctuators.

If the idea of such unconstrained flag clustering is too libertarian for a particular application, the feature may be restricted (or removed entirely), by including a **[cluster:<*option*>]** directive anywhere in the specification string. The clustering options are:

any	The **[cluster:any]** directive allows any suitable flags to be clustered (that is, it simply makes explicit the default behaviour).
flags	The **[cluster:flags]** directive restricts clustering to parameters which are "pure flags" (that is, those which have no parameter variables or punctuators - not even optional ones).
singles	The **[cluster:singles]** directive restricts clustering to parameters which are "pure flags", and which consist of a flag prefix followed by a single alphanumeric character.
none	The **[cluster:none]** directive turns off clustering completely.

Strict and non-strict command-line parsing

"Strictness" in `Getopt::Declare` refers to the way in which unrecognized command-line arguments are handled. By default, `Getopt::Declare` is "non-strict", in that it simply skips silently over any unrecognized command-line argument, leaving it in **@ARGV** at the conclusion of command-line processing.

However, if a new `Getopt::Declare` object is created with a specification string containing the **[strict]** directive (at any point in the specification):

```
$args = new Getopt::Declare
                        <<'EOSPEC';
    [strict]
    -a    Append mode
    -b    Back-up mode
    -c    Copy mode
EOSPEC
```

then the command-line is parsed "strictly". In this case, any unrecognized command-line argument (such as –q) will cause an error message to be written to **STDERR**, and command-line processing to fail (after the entire command-line has been parsed). On such a failure, the call to **Getopt::Declare::new()** returns **undef** instead of the usual hash reference.

The only concession that "strict" mode makes to the unknown is that, if command-line processing is prematurely terminated via the **finish** operator, any command-line arguments which have not yet been examined are left in **@ARGV** and do not cause the parse to fail (of course, if any unknown arguments are encountered *before* the **finish** was executed, those earlier arguments *will* cause command-line processing to fail).

Parameter dependencies

`Getopt::Declare` provides several other directives which modify the behaviour of the command-line parser in some way. One or more of these directives may be included in any parameter description. In addition, the **[mutex:...]** and **[repeatable]** directives may also appear in any usage "decoration".

Each directive specifies a particular set of conditions that a command-line must fulfil. If any such condition is violated, an appropriate error message is printed. Furthermore, once the command-line is completely parsed, if any condition was violated, the call to **Getopt::Declare::new()** dies.

The directives are:

[required]

The **[required]** directive specifies that an argument matching at least one variant of the corresponding parameter *must* be specified somewhere in the command-line. That is, if two or more required parameters share the same flag, it suffices that *any one* of them matches an argument (recall that `Getopt::Declare` considers all parameter specifications with the same flag merely to be variant forms of a single "underlying" parameter).

[repeatable]

By default, `Getopt::Declare` objects allow each of their parameters to be matched only once (that is, once any variant of a particular parameter matches an argument, *all* variants of that same parameter are subsequently excluded from further consideration when parsing the rest of the command-line).

However, it is often useful to allow a particular parameter to match more than once. Any parameter whose description includes the directive **[repeatable]** is *never* excluded as a potential argument match, no matter how many times it has matched previously:

```
    -nice      Increment nice value
               [repeatable]
                     { $::nice++; }
```

If the **[repeatable]** directive appears outside the description of any parameter (usually at the start of a specification), then *all* parameters are marked as repeatable.

[mutex:*<flaglist>*]

The **[mutex:...]** directive specifies a set of parameters which are to be treated as mutually exclusive. That is, no two or more of them may appear in the same command-line. For example:

```
    -case      set to all lower case
    -CASE      SET TO ALL UPPER CASE
    -CaSe      seT tO All RaNsoM cASe
        [mutex: -case -CASE -CaSe]
```

The interaction of the **[mutex:...]** and **[required]** directives is potentially awkward in the case where two "required" arguments are also mutually exclusive (since the **[required]** directives insist that both parameters must appear in the command-line, whilst the **[mutex:...]** directive expressly forbids this).

`Getopt::Declare` resolves such contradictory constraints by relaxing the meaning of "required" slightly, so that an argument that matches any flag in a **[mutex...]** set is implicitly considered to have matched all the flag's mutually exclusive alternatives as well. Hence the specifications:

```
    -case      set to all lower case
                     [required]
    -CASE      SET TO ALL UPPER CASE
                     [required]
    -CaSe      seT tO All RaNsoM cASe
                     [required]
        [mutex: -case -CASE -CaSe]
```

mean that *exactly one* of these three flags must appear on the command-line, but that the presence of any of them will suffice to satisfy the "requiredness" of all.

[requires:*<condition>*]

The **[requires:...]** directive specifies a set of flags which must also appear (or not appear) in order for a particular flag to be permitted in the command-line. The condition is a boolean expression, in which the terms are the flags of various parameters, and the operations are **&&**, **||**, **!**, and bracketing. For example, the specifications:

```
    -num              Numeric sort order
    -len              Sort by length
    -field <N:+i>     Sort on field <N>
    -rev              Reverse sort order
                      [requires: -num ||
                        (-len && !-field)]
```

means that the **-rev** flag is valid only if the **-num** parameter has matched, or if the **-len** parameter has been found but not the **-field** parameter. Note that the operators **&&**, **||** and **!** retain their normal Perl precedences.

Predefined grammars

`Getopt::Declare` currently provides two predefined grammars which may be specified using a single keyword instead of a full usage specification. For example:

```
$args = new Getopt::Declare(-PERL);
```

declares a command-line parser which is functionally equivalent to the one installed by the **perl -s** option (except, of course, the `Getopt::Declare` version also provides automated usage and version information, and allows flags to appear anywhere on the command-line). Likewise:

```
$args = new Getopt::Declare(-AWK);
```

allows the program to use *awk*-like arguments (of the form: var=val) to create run-time variables (of the form: **$::var = 'val'**).

It is also possible to specify both predefined grammars together , by concatenating the keywords:

```
$args = new Getopt::Declare
                         ('-AWK-PERL');
```

Autogenerated Features

Usage information

The original specification string that is passed to **Getopt::Declare::new()** is used (almost verbatim) as a "usage" display whenever usage information is requested. However, the following changes are made to the original specification before it is displayed:

- All actions and comments are deleted,
- Any **[ditto]** directive is converted to an appropriate set of "ditto" marks,
- Any text in matching unescaped square brackets (including any directive) is deleted,
- Any parameter variable type specifier (**:i**, **:n**, **:/pat/**, etc.) is deleted.

Otherwise, the usage information displayed retains all the formatting present in the original specification.

In addition to this information, `Getopt::Declare` displays three sample command-lines: one indicating the normal usage (including any required parameter variables), one indicating how to invoke help, and one indicating how to determine the current version of the program.

Help parameters

By default, `Getopt::Declare` automatically defines six case-insensitive parameters: three "help" parameters (**-h**, **-help**, and **--help**) and three "version" parameters (**-v**, **-version**, and **--version**). Hence, most attempts by the user to request help will be successful.

Note however that, if a parameter with any of these flags is explicitly specified in the string passed to **Getopt::Declare::new()**, that flag (only) is removed from the list of possible help flags. For example:

```
-h  <pixels:+i>      Height in pixels
```

would cause the **-h** help parameter to be removed (although help would still be accessible by specifying any of the arguments: –H, –help, –Help, ––HELP, etc).

Other applications

The wide range of features and ease-of-specification of `Getopt::Declare` parsers, combined with their ability to parse from sources other than **@ARGV**, make them adaptable to a number of other common parsing applications. This section examines three such uses: parsing comma-separated values, processing text templates, and implementing simple command languages.

CSV parsing

Alan Citterman's excellent **Text::CSV** package provides a simple mechanism for parsing comma-separated values:

```
my $parser = new Text::CSV;
open CSV_FILE, $file or die;
while (defined($line = <CSV_FILE>))
{
    if ($csv->parse($line))
    {
        my ($ID, $name, $score) =
                 $csv->fields();
        process($ID,$name,$score)
        && next
           if $ID=~/^[A-Z]\d{7}$/
              && $score eq 0+$score;
    }
    print "Invalid: $line\n";
}
```

`Getopt::Declare` can mimic this behaviour, somewhat more compactly:

```
my $fmt =
q{
  [repeatable]
  [type: id /[A-Z]\d{7}/ ]

  <ID:id> , <name:qs> , <score:n>
     { process($ID,$name,$score) }

  <line:/.*/>
     { print "Invalid: $line\n"; }
};
new Getopt::Declare($fmt, [$file])
   or die;
```

More importantly, `Getopt::Declare` makes it simple to handle variant formats of comma-separated values in the same input stream:

```
my $fmt =
q{
  [repeatable]
  [type: id /[A-Z]\d{7}/ ]

  <ID:id> , <name:s> , <score:n>
     { process($ID,$name,$score) }

  <name:s> , <ID:id> , <score:n>
     { process($ID,$name,$score) }

  <ID:id> , <score:n>
     { process($ID,'???',$score) }

  <line:/.*/>
     { print "Invalid: $line\n"; }
};
new Getopt::Declare($fmt, [$file])
   or die;
```

Text processing

`Getopt::Declare` can also be used to implement simple text parsing and interpolation. For example, here is a simple text template processor subroutine (**`interpolate()`**) that interpolates any Perl code appearing between doubled curly brackets:

```
my $interpolator =
    new Getopt::Declare
        (<<'EOINTERP',[-BUILD]);

  [repeatable] [cluster:none]
  [type: NOTDELIM /(?:(?!}}).)+/]
  [type: WS /\s*/ ]

  \{{ <cmd:NOTDELIM> }}[<ows:WS>]
       { $self->{text} .=
           eval "no strict; $cmd";
         $self->{text} .= $ows
              if $ows; }

  <other>[<ows:WS>]
       { $self->{text} .= $other;
         $self->{text} .= $ows
              if $ows; }
EOINTERP

sub interpolate($)
{
  $interpolator->{text} = '';
  $interpolator->parse($_[0]);
  return $interpolator->{text};
}

print interpolate
   'mean: {{ sum(@a)/@a }}';
```

Simple input handling

Because `Getopt::Declare` handles input from TTY devices on a line-by-line basis, it can also be used to implement simple command processors in a straightforward manner:

```
my $cmds = q{
    [type: ID /[A-Z]\d{7}/ ]
    [repeatable]

    f[ind] <id:ID>        Find by ID
       { print $marks->find($id) }

    f[ind] <name:/.*/>    Find name
       { print $marks->find($name) }

    d[elete] <id:ID>      Remove ID
       { $marks->find($id)->del() }

    h[elp]
       { $self->usage(); }
};

new Getopt::Declare($cmds, ['-']);
```

Prompting is also easy to incorporate, by using a subroutine reference as a source:

```
my $get =
     sub { print "> " if -t; <> };

new Getopt::Declare ($cmds, $get);
```

Conclusion

The `Getopt::Declare` package fills yet another niche in the multi-dimensional `Getopt::` solution space. Its approach of constructing command-line recognizers by reverse-engineering a "usage" specification has proved to be a simple and power means of argument parsing, and one which encourages better documentation of a program's code and interface.

Moreover, the approach is easily generalized to provide declarative solutions to a range of similar parsing tasks, where the full power of a recursive parser is not required.

`Getopt::Declare` is freely available from the author at:

```
http://www.csse.monash.edu.au/~damian/
                    CPAN/Getopt-Declare.tar.gz
```

References

[1] Wall, L., Christiansen, T., & Schwartz, R.L., *Programming Perl*, 2nd Edition, O'Reilly & Associates, 1996.

[2] Conway, D.M., *Autogenerating Documented Command Line Interfaces*, in "Springer's Lecture Notes of Computer Science: Human-Computer Interaction", ed. Blumenthal, Gornostaev & Unger, vol. 876., pp. 77-94, Springer-Verlag, Berlin, 1994.