Command Line Processing and Guarded System Calls

Gene Myers

July 28, 2009

In the course of writing any complete Unix program in C one must use the standard library and process command line arguments. In this appendix we present a library of routines for these core functions in the module `utilities`. It consists of a collection of routines that guard the return results of standard library calls for memory allocation and file handling, and a collection of routines to facilitate the processing of command line arguments.

1 Guarded System Calls

When writing a standard Unix pipe, a C programmer uses standard library routines such as `malloc` and `fopen`, for allocating memory, opening files, and other system level functions. These routines can fail for a number of reasons in which case one should detect and handle the failure. In interactive applications involving a GUI one should inform the user and recover. But for the common case that one is building a batch-oriented UNIX command, we supply the routines below that simply stop execution after reporting to `stderr` that an exception occured in routine `routine`.

1. `void *Guarded_Malloc(int size, char *routine);`

   A guarded version of `malloc`.

2. `void *Guarded_Realloc(void *array, int size, char *routine);`

   A guarded version of `realloc`.

3. `char *Guarded_Strdup(char *string, char *routine);`

   A guarded version of `strdup`.

4. `FILE *Guarded_Fopen(char *file_name, char *options, char *routine);`

   A guarded version of `fopen`. On exception, report the name of the file.

2 Command Line Processing

All UNIX programs take arguments from the command line as an array `argv` of `argc` strings. It becomes quite tedious to copy and retailor code for processing the command line each time you write a new program. To alleviate this the core library contains the routine `Process_Arguments` that interprets the command line arguments from a specification, where the specification can be a complex pattern that also serves as the usage statement shown to a user of the program in the event a user invokes the program improperly. `Process_Arguments` stores the results of its command-line interpretation in a permanent table so that you can then get the value of any command-line argument at any time during the subsequent computation of your program with a number of simple access routines.

We assume that the reader is familiar with UNIX command line conventions and syntax, but we quickly review the range of possibilities to set the stage. A typical UNIX program takes a sequence of required arguments that must occur in a given order and a number of optional arguments that begin with '`-`' and can occur any where in the command list. Some optional arguments can take additional arguments which must immediately follow them, e.g. `-f myfile`, and others serve simply as boolean indicators that we call *pure options*, e.g. `-catenate`. In many cases a single letter is used for a boolean option and these are called *flags*. The special feature of flags is that any number may be strung together in a single argument, e.g. `ps -axl`. Often one can iterate or repeat an argument, e.g. `mv file1 file2 ... target`. Finally, in some cases the number and type of the arguments determine the nature of the interpretation.

Our design goal for this utility was to encompass as many of the features and styles of command line interfaces and conventions as possible. To do this required a quite flexible and powerful specification language whose semantics are compicated enough that (a) it takes a bit of effort to understand, and (b) you can create some pretty surprising patterns if you wish. Since designing it I have not encountered a situation where I could not specify the command-line syntax that I desired and I have created some rather interesting ones in less than a couple of minutes of coding effort. I hope you find it useful too.

2.1 Units: Basic Matchable Items

Conceptually we consider a command-line specification to be a sequence of units of one of three types. A *required unit* consists of a name and a type between angle-brackets separated by a colon, e.g. `<query:string>`. The name can be any string that is descriptive of the intended meaning of the argument, and the type can be one of `int`, `double`, or `string`. On the command line, this unit matches an argument of the given type and within your program you can retrieve the matched value by its name. To further enhance expression, you may actually give any descriptive string as a type, e.g. `fasta`, in which case the type will be considered to be `string`.

An *optional unit* always begins with a '`-`' and thereafter is a string that (a)

may have imbedded within it *value* templates that are one of the three types between angle-brackets, e.g. `-r<int>:<int>`, and (b) may be followed by a sequence of value templates separated by white space, e.g. `-range <int> <int>`. In the second example the optional unit will match three arguments on the command line and we say it has a *span* of 3, e.g. it will match `-range 20 50` on the command line, where `-range`, `20`, and `50` are each an argument with respect to the parsing of the command line into the array `argv` by the operating system. The first example has a span of 1 and will, for example, match `-r20:50` on the command line. Note that one can obviously give ambiguous templates such as `-a<string>:<string>` which can match `-afoo:boo:too` in two different ways. We assume as the designer that you will avoid such ambiguous specifications. Our software simply returns the first match it finds if more than one match is possible.

An optional unit that consists of a single letter, e.g. `-r`, is a *flag*. As a special shorthand, a `!` followed by a sequence of letters is a *flag unit* that matches any combination of the letters except the empty set, e.g. `!axl` denotes the flags `-a`, `-x`, and `-l`, and it matches any combination such as `-al`, `-x`, and `-lax`.

Required units must be found on the command line in the order in which they occur in the specification whereas optional and flag units may occur in any order. Flags in a flag unit may also be set in separate command-line arguments. For example, consider the specification:

```
!axl -r <int> -v:<int> <a:int> <b:string>
```

This specification matches any of the following sample command lines:

```
-lax -r 10 -v:20 30 i_match_b
30 -la -v:20 i_match_b -x -r 10
-xl 30 -r 10 -xa -v:20 i_match_b
```

All three command lines match the specification with the same values and are equivalent in every way. Note that for the option `-r` that has a span of 2, the option can appear anywhere but its follow-on argument must always occur immediately after it, that is, the components of an option with a span of more than one must occur consecutively and in order on the command line. The

following grammar gives a formal definition of units and their syntax:

<unit> ← <required> | <optional> | <flag>

<required> ← <named>

<optional> ← <dash>(<value><text>)*<value>?(<white><value>)*

<flag> ← <list>

<named> ← '<'<name>':'<type>'>'

<dash> ← '-'<text>

<list> ← '!'<text>

<text> ← any string not containing '(', ')', '[', ']', '{', '}', '|', '@, '', or '...' save those escaped by ' (right quote)

<value> ← '<'<type>'>'

<white> ← any string of spaces, tabs, or new-lines

<name> ← any string not containing ':'

<type> ← 'int' | 'double' | 'string' | any string not containing '>'

The restriction on the characters that can occur in a <text> item will become apparent momentarily when the special meaning of the restricted characters is introduced.

2.2 Specifications

A specification is actually not a sequence of unit definitions, but a regular expression of the *atoms*: <named>, <dash>, <list>, <text>, <value>, and <white> introduced in the grammar for units in the previous section. These atoms may be combined with the operators: `|` for alternation, juxtaposition for concatenation, surrounding square brackets `[]` for optional, and `...` as a unary suffix operator for 1-or-more repetition. Specifically, the grammar defining a specification is as follows:

<specification> → <or>

<or> → <con> ('|' <con>)*

<con> → <rep> <rep> *

<rep> → <fact> '...'

<fact> → '(' <or> ')' | '[' <or> ']' | '{' <or> '}' | <atom>

<atom> → <named> | <dash> | <list> | <text> | <value> | <white>

We'll defer giving an explanation of the meaning of curly braces until the next subsection. As an example consider:

```
[-matrix <string>] [-thresh'(<int>')] <query:fasta> <target:fasta> ...
```

In this case the options `-matrix` and `-thresh` are optional, and the required argument `target` can be repeated. Note that the parentheses need to be escaped

for the option `-thresh`, which will match, for example, `-thresh(40)` on the command line. As a more powerful example consider:

```
[!Ccu] <1:string> | [!CuU] -s <1:string> <2:string>
```

This specification allows the command line to have one or two required arguments and different flags available depending on whether or not the `-s` flag is set. For example,

```
foo                                  foo boo
-s foo boo                           -s foo
-uc foo                              -U foo
-Us foo boo                          -cs foo boo
```

all the command lines at left match the specification, whereas all those on the right do not.

Because a specification is over atoms as opposed to units, one can write quite complex patterns. For example, for a program that displays alignments between pairs of potentially clipped DNA sequences, one might use the specification `-a[c[:<file>]]` that matches either the flag `-a`, the pure option `-ac`, or the option `-ac:<file>`. The meaning in this example, is that if `-a` is set then one is requesting to see the alignments between sequence pairs. If `-ac` is set then one wishes to see the alignments of the clipped sequences where a default clipping is used. And lastly, if `-ac:<file>` is given then one wishes to see the alignments of the sequences clipped according to coordinates given in `<file>`. What is aesthetic here is giving the file makes no sense without the clipping option set, and giving the clipping option makes no sense unless the user wishes to view the alignments.

As another exmaple, `-a(i <int>|d <double>)` can match `-ai <int>`, or `-ad <double>`. But not every regular expression of atoms is necesarily meaningful as it may match a sequence of atoms that cannot be interpreted as a sequence of units. For example, `-a(i <int>|d <double>)x` is not valid because `-ai <int>x` is not a unit. Formally, we say that a specification is *valid* if and only if:

> *Every sequence of atoms matched by the specification can be parsed as a sequence of white-space separated units.*

The routine `Process_Arguments` checks a specification to make sure that it is valid and issues a hopefully meaningful error if not.

We also place some restrictions on the nature of repetitions. Although none of these is absolutely necessary, the restrictions do little to limit your expressive power yet make specifications easier to understand and process. Specifically,

> *Repetitions (a) cannot contain a fractional part of a unit, (b) cannot contain flags or pure options, and (c) cannot nest.*

As an example of (a), `[-a(b -c)...d]` is actually valid (`-ab`, `-cb`, and `-cd` are all units), but it is difficult to see this at first, and one could just as well have written `[-ab [-bc...] -cd]` where units are wholly contained within repetitions. With regard to restriction (b), there is no value in being able to repeat a pure option or flag: once it is set, it is redundant to set it again. Finally, as we will see shortly, in order to access the value of an option or required unit within a repetition you have to specify from which iteration you want the value. Nesting complicates the indexing of values and I've yet to see a program that employs it. Moreover, very few specifications involving nested repetitions are unambiguous.

If a specification matches a command line in more than one way, but one match involves more required unit matches to integers and doubles, than that match is taken as it is more specific than one that uses string matches. For example `( <s:string> | <b:int> <e:int> )...` matches `foo 1 2 3 4 boo` in five ways, but there is only one match involving four numbers, namely, `s` matching `foo` and `boo`, `b` matching `1` and `3`, and `e` matching `2` and `4`. As a designer, you should strive to create specifications that always have a unique, most-specific match to any given command line possibility. If a specification does have two or more equally specific matches, then `Process_Arguments` simply uses the first of these that it finds and prints a warning message.

2.3 Synonyms and Default Values

Curly braces, {}, can be used within the textual part of an option to create synonymous option names. For example `-{m|matrix}` specifies *an* option whose name is either `-m` or `-matrix`, where the value of having two names for the same thing is that the user can type in either at the command line, but you can choose one name to access the value throughout your program. Note in this example, that one form, `-m` is a flag and can be set as such, e.g. `-xmu` where `-x` and `-u` are presumed to be other flags. We expect that the most typical use of this feature is to introduce long and short forms of options as is conventional in a number of popular UNIX programs.

Curly braces are restricted to enclosing just the non-value parts of an option specification as they don't make sense anywhere else. But you can still get quite tricky if you'd like, for example, `-{c|circle[_cent(er|re)]}` establishes `-c`, `-circle`, `-circle_center`, and `-circle_centre` as synonyms.

A final specification convenience is that one can declare a default value for a given option value by placing the value in parenthesis after the type indication. For example, `-matrix <file(PAM120)>`, declares that the default scoring matrix will come from the file `PAM120` if the option is not matched on the command line. As a more complex example, `-center'(<int(0)>,<int(0)>')`, declares the default value(s) of `-center` to be `(0,0)`. To recapitulate, the default value is the value returned when one requests a value for an option that has not been matched on the command line.

2.4 Looking Up Argument Values

We now turn to how one gets values from the command line once `Process_Arguments` has matched the command line arguments to the specification. Matters are complicated by (a) the presence of union and option which mean that a particular unit may or may not be matched in an overall command-line match to the specification, (b) the fact that an option can have multiple values, and (c) the fact that units in loops can be matched multiple times.

But first a more basic question: what is the name of a unit? For a required unit, you reference it by the name you gave in the first half of the angle-bracket item defining it, e.g. the name of `<query:string>` is "`query`". For a flag unit, the name is a dash followed by the flag symbol in question, e.g. "`-a`". For an option, the name is obtained by (a) replacing each value with an '`@`'-sign, (b) compressing any white space to a single blank, and (c) removing any escaping quotes. For example, the name of `-mat <int>` is "`-mat @`", and the name of `-def<string>'(<int>')` is "`-def@(@)`". Since you designed the specification, you know the set of all possible unit names. So if a prefix of a name unambiguously identifies a given unit then you may also use the prefix as the name argument in the access routines. For example, if "`-def`" or "`-mat`" uniquely identify the option examples above, then they can be used instead, thus avoiding the somewhat cumbersome syntax of the full option name. In looking up a name, a complete match is taken over a prefix match in the event that a given name happens to be a prefix of another. For example, if one had a flag `-m` and an option `-mango`, then the name "`-m`" refers to the flag and not the option. Moreover, if two prefix matches are synonyms of each other, there is no ambiguity since they identify the same option value(s).

There is an access routine for each type of value: `Get_Int_Arg`, `Get_Double_Arg`, and `Get_String_Arg`. Each takes the name of a unit, and optionally the iteration of the unit desired if it is in a repetition, and optionally the index of the value desired if the unit is a multi-value option. For example, consider the specification:

```
[!abc] [-r <double> [<double>]] ...
<db:string> ( <list:string> | <beg:int> <end:int> ) ...
```

and the command line:

```
<program> -ac data file1 5 6 7 8 file2 -r 1. 2. -r 3.
```

The call `Get_String_Arg("db")` returns "`data`". Flags and pure options are considered to be 0/1 integers, so the call `Get_Int_Arg("-b")` returns `0`. Some of the required arguments are in a repetition and in each iteration either a string (`list`) is matched or a pair of integers (`beg` and `end`) are matched. In our sample command-line, the most specific match matches `list` to `file1` and `file2` in the first and fourth iteration of the repetition, and `beg`,`end` to `5`,`6` and `7`,`8` in the second and third iterations. So the call `Get_String_Arg("list",4)` would return `file2`. Similarly, 7 is returned by the call `Get_String_Arg("beg",3)`. The

second value of the first match to the loop containing the option `-r` is returned by the call `Get_Double_Arg("-r @ @",1,2)`. Similarly, the first value, `3.`, in the second match to the option is returned by the call `Get_Double_Arg("-r @",2)`. Note, in these last examples that one has to use the full option name including `@`-signs in order to unambiguously specify the unit, and that one includes or does not include an option argument strictly on the bases of whether or not the unit is in a repetition and whether or not the unit has multiple values.

The example above also serves to illustrate that you need to be able to ask how many times a loop has been repeated and whether or not a particular item has been matched, e.g. how does one know that there were two matches to the `-r` option, and that the second match has one value as opposed to two (`"-r @"` versus `"-r @ @"`)? `Get_Repeat_Count(name)` returns 0 if the unit was not in a repetition that was used to match the comand line, and otherwise it returns the number of times that repetition was repeated in order to make the match. For example, `Get_Repeat_Count("beg")` would return 4 above even though `beg` itself is only matched in the second and third iterations. Indeed, the call would return 4 even if `list` was matched in all four iterations – all that matters is that the loop *containing* the unit was traversed four times to make the match. To determine things like whether or not `list` was matched in the third loop iteration versus `beg` and `end`, we supply the routine `Is_Arg_Matched(name[,no])` which returns a non-zero value only if the named unit was matched (in the `no`th iteration if the unit is in a repetition). As an example, `Is_Arg_Matched("-r @",1)` returns 0, where as `Is_Arg_Matched("-r @ @",1)` returns 1.

We conclude with some subtleties that would have overburdened the descriptions above. First note that we have been very careful in our language about when a unit is in a loop: for the specification — `<a:int> | (-r <a:int>)...` — "`a`" may or may not be in a repetition in a given command-line match depending on whether or not the `-r` option is present. Second, even if an option has a default for every value in it, `Is_Arg_Matched` returns a non-zero value only if the option was matched in the command line. However, calling the appropriate `Get`-routine on a given option will return the default value regardless of its match status.

2.5 The Programming Interface

We conclude with a description of the routines in the library that process the command line and return argument values to the user.

1. `void Process_Arguments(int argc, char *argv[], char *spec[], int no_escapes);`

   This should be one of the very first if not the first call in the main routine of the program. The `argc` arguments in `argv` from the command line are parsed and interpreted according to `spec`. Rather than giving the specification as a single string, one gives it as an array of strings whose last element is `NULL` and where the concatenation of the strings in the

array is the specification. The reason for this is to allow you to control exactly what the usage statement will look like, as each string will be output as a line of the specification, offset by the program name. For example, consider the specification:

```
static char *Spec[] = { "[-m <int>] [!xyz] [-c'(<int>')]",
                        "  <arg1:int>  <arg2:int>",
                        NULL;
                      };
```

and suppose that the compiled program is called, say `my_program`. Then the usage statement that will be printed by `Print_Argument_Usage` (see below) is:

```
Usage: my_program [-m <int>] [-xyz] [-c'(<int>')]
                    <arg1:int>  <arg2:int>
```

provided that the argument `no_escapes` is zero. Otherwise, the escaping quotes before the parens in the option `-c` will not be printed.

Note carefully that the routine catches two classes of errors – those made by the programmer and those made by the user. The author of the program should be certain to thoroughly debug the specification and the access calls so that the only errors a user sees are those due to their giving the program invalid input on the command line. Messages to the user always begin with the program name and messages to the implementor always begin with "`Error in <routine name>:`". Messages to the user always end with a usage statement for the program, thus the necessity of having `no_escapes` as a parameter to `Process_Arguments`.

Once the arguments have been processed by this routine, all the following routines can be used to fetch information at any time.

2. `char *Program_Name();`

   Return a pointer to the name of the program.

3. `int Get_Repeat_Count(char *name);`

   Return 0 if the named unit is not in a repetition or if the repetition it is in was not used in making the match to the command line. Otherwise return the number of times the repetition containing the named unit was iterated to match the command line. Note carefully that the name should not contain any escaping quotes that might have been necessary in the specification.

4. `int Is_Arg_Matched(char *name [, int no]);`

   For a given unit name, return non-zero if the unit instance has been matched.

5. `int Get_Int_Arg(char *name [, int no] [, int an]);`

   Return the (integer) value of the argument whose name is `name`. If the argument is in a repetition then you must supply the parameter `no`, and the routine returns the `no`th instance. If the argument is one of several in a multi-value option, then you must also supply the parameter `an`, and the routine returns the `an`th value in the option.

6. `double Get_Double_Arg(char *name [, int no] [, int an]);`

   Exactly like `Get_Int_Arg` save that the returned value is double.

7. `char *Get_String_Arg(char *name [, int no] [, int an]);`

   Exactly like `Get_Int_Arg` save that the returned value is string.

8. `void Print_Argument_Usage(FILE *file, int no_escapes);`

   Print a usage message for the program to stream `file`. The message is taken directly from the specification as discussed under the description of `Process_Arguments`. If the argument `no_escapes` is non-zero then any escape characters are removed from the display of the specification.