NAME

getargs(), getlargs(), getvargs() - parse arguments until a non-flag is reached

SYNOPSIS

```
#include <schily/getargs.h>

int getargs(pac, pav, fmt, a1, ..., an)
        int *pac;       /* pointer to arg count */
        char *(*pav)[]; /* pointer to address of arg vector */
        char *fmt;      /* format string */
        type *a1;       /* pointer to result 1 (corresponding */
                        /* to the first descriptor in fmt) */
        type *an;       /* pointer to result n (corresponding */
                        /* to the nth descriptor in fmt) */

int getlargs(pac, pav, props, fmt, a1, ..., an)
        int *pac;       /* pointer to arg count */
        char *(*pav)[]; /* pointer to address of arg vector */
        struct ga_props *props; /* control properties */
        char *fmt;      /* format string */
        type *a1;       /* pointer to result 1 (corresponding */
                        /* to the first descriptor in fmt) */
        type *an;       /* pointer to result n (corresponding */
                        /* to the nth descriptor in fmt) */

int getvargs(pac, pav, props, vfmt)
        int *pac;       /* pointer to arg count */
        char *(*pav)[]; /* pointer to address of arg vector */
        struct ga_props *props; /* control properties */
        struct ga_flags *vfmt;  /* array of formats and args */
```

DESCRIPTION

getargs() is part of the advanced option parsing interface together with the **getallargs**() and **getfiles**() family.

getargs() looks at each argument that begins with '-', '+', or has an '=' in it and tries to find a matching description in **fmt**. If a match is found, the corresponding value pointed to by **a1** to **an** is set to the value according to the conversion specification.

If a match is not found, **getargs**() returns the error code **-1** (**BADFLAG**), with ***pav[0]** pointing to the bad argument. If an argument that does not begin with '**-**' or '**+**' or does not contain an '**=**' is found, **getargs**() returns **+1** (**NOTAFLAG**), again with **pav[0]** pointing to the non-flag argument. If the argument "**--**" is found, **getargs**() returns **+2** (**FLAGDELIM**) and **pav[0]** points to the argument after the argument "**--**".

getlargs() is similar to **getargs**() but implements an additional **ga_props** parameter that must be initialized with **getarginit**() before it is passed. Instead of using an explicit structure parameter, the special parameter **GA_NO_PROPS** may be used to enforce default behavior and the special parameter **GA_POSIX_PROPS** may be used to enforce **POSIX** compliant behavior.

getvargs() is similar to **getlargs**() but uses a structure **ga_flags** instead of a format string and a variable argument list with pointers. The array of structures **ga_flags**:

```
struct ga_flags {
  const char  *ga_format; /* Comma separated list for one flag */
  void        *ga_arg;    /* Ptr. to variable to fill for flag */
  getpargfun  ga_funcp;   /* Ptr. for function to call (&/~)   */
};
```

is terminated by an element with **ga_format == NULL**. For a **ga_format** that does not expect a function pointer, **ga_funcp** is **NULL**.

With **getargs**(), each normal format takes one address argument from **a1** to **an** and each function type format takes two address arguments from **a1** to **an**.

In the description, it is assumed that **pac = &ac** and **pav = &av**, where **ac** and **av** are the two arguments passed to **main**(). The pointers are necessary so that **getargs**() can update **ac** and **av** as it verifies each argument and reflects the *current* position back to the user.

The format string is a series of one or more option descriptors. Each option descriptor starts with the **option-name** which is composed of **characters**, **numbers**, the **underscore character '-'**, **minus** or **plus**, which must match the option parameter on the command line. The **plus** sign (+) must be escaped via \\ in the format string to distinguish it from the **+** format character. Each **option-name** is followed by the optional **format descriptor** and an optional **size modifier**.

Legal conversions and their meanings are:

**Integer**

The remainder of the current argument, or, if it is empty, the next existing argument is converted to an int value. An error in conversion results in a **BADFLAG** situation.

+ **Increment** *sized* **integer**

The value of the related argument pointer is incremented, assuming a *size* that depends on the optional **size modifier** after the **+**. See the integer conversions above for a list of valid **size modifiers**.

empty
BOOLean **TRUE**

If the **option-name** is not followed by a format descriptor, the value of the related argument pointer is interpreted as an integer and set to **TRUE** (**+1**).

%0 .. **%9**
Set *sized* **integer** to value in the range **0**..**9**.

The value of the related argument pointer is either set to the single digit value that follows the **%** character, assuming a *size* that depends on the optional **size modifier** after **%0** .. **%9**. See the integer conversions above for a list of valid **size modifiers**.

? **Character**

The next character in the current argument is the result. If there is no next char, the result is '**\0**'.

***** **String**

A pointer to the remainder of the current argument is returned in the related argument pointer. If there are no more data in the argument the next argument is used, and if there is no next argument, a **BADFLAG** situation is returned.

& **Call function**

This format takes two parameters in the argument list of **getargs**(). The first argument is a pointer to a function to call. The second argument is a pointer to a variable that is passed to the function as

second argument.

Because the argument just after the function address argument is passed as a second argument to the function, common routines can have their results in different places depending on which switch is invoked.

The function is called with five arguments:

1) A pointer to the option argument, taken from the matching element from the command line from ***pav**.

2) A pointer to the variable that should be set by the function.

3) The current value of **pac**.

4) The current value of **pav**.

5) A pointer to the matching part of the format string.

The function must return one of these values:

FLAGDELIM = **+2** Pretend that "**--**" stopped flag processing.

FLAGPARSED = **+1** Option processing was successful.

NOARGS = **0** Pretend that all arguments have been examined.

BADFLAG = **-1** The current flag argument or parameter is not understood.

BADFMT = **-2** An unspecified error occurred.

NOTAFILE = **-3** Probably another flag type argument. Tell the calling function (**getargs**()) to continue to check for other flag type arguments in the format string for a possible match.

Note: If a flag is found multiple times, the function is called each time.

~ **Call function** for **BOOL**ean flag

This is a variant of the **&**-format, but as a boolean flag is assumed, no option argument is assumed and if the related option is a single char option, it may be combined with other single char options. The called function may reset other options at the same time.

As boolean flags take no arguments, the first argument of the called function points to an empty string.

The conversion types:

Integer conversion

+ Increment integer

%[0-9] Integer assignment

may have a size modifier:

c or **C**
The assignment is made to an character sized object.

s or **S**
The assignment is made to a short int sized object.

empty
i or **I**
The assignment is made to an int sized object.

l or **L**
The assignment is made to a long int sized object.

ll or **LL**
The assignment is made to a long long int sized object.

Flag (option) descriptors are separated by a '**,**' (without whitespace) in the format string. They correspond in order to the resultant pointers, **a1...an**. Note that function type formats take two arguments from resultant pointers, **a1...an**.

It is an error to expect more than one conversion from a single match (e.g., **"x#*"** to attempt to get both the numerical value and the actual string for the **x** flag); a **-2** (**BADFMT**) error will result if this is

attempted.

Although flags must appear exactly as they do in the format string, the format string does not contain the leading '-'. If the flag should start with a '+', the '+' needs to be in the format string. If a long flag should start with a '--', and a long flag with a single dash should not be permitted, a single '-' needs to be in front of the flag name in the format string.

Flags, where conversion is to take place, may appear either as:

-f*value*
f=*value*
f= *value*
-f=*value*
-f= *value*

where **f** is the matching flag string. No additional effort is required to get these different ways of specifying values.

Long flags, where conversion is to take place, may appear either as:

-flag*value*
--flag*value*
flag=*value*
flag= *value*
-flag=*value*
--flag=*value*
-flag= *value*
--flag= *value*

where **flag** is the matching flag string. All the above variants are accepted by the function.

For flags of type *****, **?**, **&** and **#**, when the format character is immediately followed by a space or underscore character, the permitted option calling variants are limited:

- The underscore character enforces that **option-name** and **option-argument** need to be written as a single argument. This allows implementing options with optional arguments.

- The space character enforces that **option-name** and **option-argument** need to be written as separate arguments.

RETURNS

FLAGDELIM 2 The command line argument "**--**" stopped flag processing.

NOTAFLAG 1 The argument ***pav** does not appear to be a flag.

NOARGS 0 All arguments have been successfully examined.

BADFLAG -1 A bad flag (option) argument was supplied to the program. The argument ***pav** contains the offending command line argument.

BADFMT -2 A bad format descriptor string has been detected. This means an error in the calling program, not a user input data error.

General rules for the return code:

> 0 A file type argument was found.

0 All arguments have been parsed.

< 0 An error occurred or not a file type argument.

Flag and file arg processing should be terminated after getting a return code <= 0.

SEE ALSO

getarginit(3), **getallargs**(3), **getargerror**(3), **getfiles**(3), **getlallargs**(3), **getlargs**(3), **getlfiles**(3), **getvallargs**(3), **getvargs**(3), **getvfiles**(3).

NOTES

Users might find it surprising that given a format string like *"foo*,bar*"* and called with the command line *"foo= bar=baz"* the **getargs**(3) family of functions will consider *"bar=baz"* as the argument to the "foo=" flag. Pay special attention to this in shell scripts where e.g. "foo=$bar" will consume the next argument if "$bar" is empty. To avoid this, write *foo= "$bar"* instead.

getargs() assumes the first argument is at **av[0]**. Commands are invoked by the system with the command name in **av[0]** and the first argument in **av[1]**, so they must increment **av** and decrement **ac** before calling **getargs**().

getargs() should only be used when the position of the switches influences how an argument is processed, or when all switches must be before all file type arguments. In other cases, use **getallargs**().

BUGS

None currently known.

Mail bugs and suggestions to **schilytools@mlists.in-berlin.de** or open a ticket at **https://codeberg.org/schilytools/schilytools/issues**.

The mailing list archive may be found at:

https://mlists.in-berlin.de/mailman/listinfo/schilytools-mlists.in-berlin.de.

AUTHOR

Joerg Schilling and the schilytools project authors.