NAME

IO::Prompter – Prompt for input, read it, clean it, return it.

VERSION

This document describes IO::Prompter version 0.004015

SYNOPSIS

```
use IO::Prompter;

while (prompt -num, 'Enter a number') {
    say "You entered: $_";
}

my $passwd
    = prompt 'Enter your password', -echo=>'*';

my $selection
    = prompt 'Choose wisely...', -menu => {
            wealth => [ 'moderate', 'vast', 'incalculable' ],
            health => [ 'hale', 'hearty', 'rude' ],
            wisdom => [ 'cosmic', 'folk' ],
        }, '>';
```

CAVEATS

1. Several features of this module are known to have problems under Windows. If using that platform, you may have more success (and less distress) by trying IO::Prompt::Tiny, IO::Prompt::Simple, or IO::Prompt::Hooked first.
2. By default the `prompt()` subroutine does not return a string; it returns an object with overloaded string and boolean conversions. This object ***always*** evaluates true in boolean contexts, unless the read operation actually failed. This means that the object evaluates true *even when the input value is zero or an empty string.* See "Returning raw data" to turn off this (occasionally counter-intuitive) behaviour.

DESCRIPTION

IO::Prompter exports a single subroutine, `prompt`, that prints a prompt (but only if the program's selected input and output streams are connected to a terminal), then reads some input, then chomps it, and finally returns an object representing that text.

The `prompt()` subroutine expects zero-or-more arguments.

Any argument that starts with a hyphen (`-`) is treated as a named option (many of which require an associated value, that may be passed as the next argument). See "Summary of options" and "Options reference" for details of the available options.

Any other argument that is a string is treated as (part of) the prompt to be displayed. All such arguments are concatenated together before the prompt is issued. If no prompt string is provided, the string `'> '` is used instead.

Normally, when `prompt()` is called in either list or scalar context, it returns an opaque object that autoconverts to a string. In scalar boolean contexts this return object evaluates true if the input operation succeeded. In list contexts, if the input operation fails `prompt()` returns an empty list instead of a return object. This allows failures in list context to behave correctly (i.e. be false).

If you particularly need a list-context call to `prompt()` to always return a value (i.e. even on failure), prefix the call with `scalar`:

```
# Only produces as many elements
# as there were successful inputs...
my @data = (
    prompt(' Name:'),
    prompt('  Age:'),
    prompt('Score:'),
);

# Always produces exactly three elements
# (some of which may be failure objects)...
my @data = (
    scalar prompt(' Name:'),
    scalar prompt('  Age:'),
    scalar prompt('Score:'),
);
```

In void contexts, `prompt()` still requests input, but also issues a warning about the general uselessness of performing an I/O operation whose results are then immediately thrown away. See "Useful useless uses of `prompt()`" for an exception to this.

The `prompt()` function also sets `$_` if it is called in a boolean context but its return value is not assigned to a variable. Hence, it is designed to be a drop-in replacement for `readline` or <>.

INTERFACE

All the options for `prompt()` start with a hyphen (`-`). Most have both a short and long form. The short form is always the first letter of the long form.

Most options have some associated value. For short-form options, this value is specified as a string appended to the option itself. The associated value for long-form options is always specified as a separated argument, immediately following the option (typically separated from it by a =>).

Note that this implies that short-form options may not be able to specify every possible associated value (for example, the short-form `-d` option cannot specify defaults with values `'efault'` or `'$%^!'`). In such cases, just use the long form of the option (for example: `-def=>'efault'` or `-default=>'$%^!'`).

Summary of options

Note: For options preceded by an asterisk, the short form is actually a Perl file operator, and hence cannot be used by itself. Either use the long form of these options, or bundle them with another option, or add a “no-op” to them.

```
  Short   Long
  form    form               Effect
  =====   =============      ======================================

  -a      -argv              Prompt for @ARGV data if !@ARGV

          -comp[lete]=>SPEC  Complete input on <TAB>, as specified

  -dSTR   -def[ault]=>STR    What to return if only <ENTER> typed
          -DEF[AULT]=>STR    (as above, but skip any -must checking)

* -e[STR] -echo=>STR         Echo string for each character typed

          -echostyle=>SPEC   What colour/style to echo input in

* -f      -filenames         Input should be name of a readable file

          -fail=>VALUE       Return failure if input smartmatches value
```

```
           -guar[antee]=>SPEC Only allow the specified words to be entered

   -h[STR] -hist[ory][=>SPEC] Specify the history set this call belongs to

           -in=>HANDLE        Read from specified handle

   -i      -integer[=>SPEC]   Accept only valid integers (that smartmatch SPEC)

   -k      -keyletters        Accept only keyletters (as specified in prompt)

 * -l      -line              Don't autochomp

           -menu=>SPEC        Specify a menu of responses to be displayed

           -must=>HASHREF     Specify requirements/constraints on input

   -n      -num[ber][=>SPEC]  Accept only valid numbers (that smartmatch SPEC)

           -out=>HANDLE       Prompt to specified handle

           -prompt=>STR       Specify prompt explicitly

 * -rSTR   -ret[urn]=>STR     After input, echo this string instead of <CR>

 * -s -1   -sing[le]          Return immediately after first key pressed

           -stdio             Use STDIN and STDOUT for prompting

           -style=>SPEC       What colour/style to display the prompt text in

   -tNUM   -time[out]=>NUM    Specify a timeout on the input operation

   -v      -verb[atim]        Return the input string (no context sensitivity)

           -void              Don't complain in void context

 * -w      -wipe              Clear screen
           -wipefirst         Clear screen on first prompt() call only

 * -y      -yes    [=> NUM]   Return true if [yY] entered, false otherwise
   -yn     -yesno  [=> NUM]   Return true if [yY] entered, false if [nN]
   -Y      -Yes    [=> NUM]   Return true if Y entered, false otherwise
   -YN     -YesNo  [=> NUM]   Return true if Y entered, false if N

 * -_                         No-op (handy for bundling ambiguous short forms)
```

Automatic options

Any of the options listed above (and described in detail below) can be automatically applied to every call to `prompt()` in the current lexical scope, by passing them (via an array reference) as the arguments to a `use IO::Prompter` statement.

For example:

```
use IO::Prompter;
```

```
# This call has no automatically added options...
my $assent = prompt "Do you wish to take the test?", -yn;

{
    use IO::Prompter [-yesno, -single, -style=>'bold'];

    # These three calls all have: -yesno, -single, -style=>'bold' options
    my $ready = prompt 'Are you ready to begin?';
    my $prev  = prompt 'Have you taken this test before?';
    my $hints = prompt 'Do you want hints as we go?';
}

# This call has no automatically added options...
scalar prompt 'Type any key to start...', -single;
```

The current scope's lexical options are always *prepended* to the argument list of any call to `prompt()` in that scope.

To turn off any existing automatic options for the rest of the current scope, use:

```
use IO::Prompter [];
```

Prebound options

You can also ask IO::Prompter to export modified versions of `prompt()` with zero or more options prebound. For example, you can request an `ask()` subroutine that acts exactly like `prompt()` but has the `- yn` option pre-specified, or a `pause()` subroutine that is `prompt()` with a "canned" prompt and the `-echo`, `-single`, and `-void` options.

To specify such subroutines, pass a single hash reference when loading the module:

```
use IO::Prompter {
    ask     => [-yn],
    pause   => [-prompt=>'(Press any key to continue)', -echo, -single, -void
}
```

Each key will be used as the name of a separate subroutine to be exported, and each value must be an array reference, containing the arguments that are to be automatically supplied.

The resulting subroutines are simply lexically scoped wrappers around `prompt()`, with the specified arguments prepended to the normal argument list, equivalent to something like:

```
my sub ask {
    return prompt(-yn, @_);
}

my sub pause {
    return prompt(-prompt=>'(Press any key to continue)', -echo, -single, -vo
}
```

Note that these subroutines are lexically scoped, so if you want to use them throughtout a source file, they should be declared in the outermost scope of your program.

Options reference

Specifying what to prompt

`-prompt => STRING`

`-pSTRING`

By default, any argument passed to `prompt()` that does not begin with a hyphen is taken to be part of the prompt string to be displayed before the input operation. Moreover, if no such string is specified in the argument list, the function supplies a default prompt (`'> '`) automatically.

The `-prompt` option allows you to specify a prompt explicitly, thereby enabling you to use a prompt that starts with a hyphen:

```
my $input
    = prompt -prompt=>'-echo';
```

or to disable prompting entirely:

```
my $input
    = prompt -prompt => "";
```

Note that the use of the `-prompt` option doesn't override other string arguments, it merely adds its argument to the collective prompt.

Prompt prettification

If the specified prompt ends in a non-whitespace character, `prompt()` adds a single space after it, to better format the output. On the other hand, if the prompt ends in a newline, `prompt()` removes that character, to keep the input position on the same line as the prompt.

You can use that second feature to override the first, if necessary. For example, if you wanted your prompt to look like:

```
Load /usr/share/dict/_
```

(where the _ represents the input cursor), then a call like:

```
$filename = prompt 'Load /usr/share/dict/';
```

would not work because it would automatically add a space, producing:

```
Load /usr/share/dict/ _
```

But since a terminal newline is removed, you could achieve the desired effect with:

```
$filename = prompt "Load /usr/share/dict/\n";
```

If for some reason you *do* want a newline at the end of the prompt (i.e. with the input starting on the next line) just put two newlines at the end of the prompt. Only the very last one will be removed.

Specifying how the prompt looks

```
-style  => SPECIFICATION
```

If the `Term::ANSIColor` module is available, this option can be used to specify the colour and styling (e.g. bold, inverse, underlined, etc.) in which the prompt is displayed.

You can can specify that styling as a single string:

```
prompt 'next:' -style=>'bold red on yellow';
```

or an array of styles:

```
prompt 'next:' -style=>['bold', 'red', 'on_yellow'];
```

The range of styles and colour names that the option understands is quite extensive. All of the following work as expected:

```
prompt 'next:' -style=>'bold red on yellow';

prompt 'next:' -style=>'strong crimson on gold';

prompt 'next:' -style=>'highlighted vermilion, background of cadmium';

prompt 'next:' -style=>'vivid russet over amber';

prompt 'next:' -style=>'gules fort on a field or';
```

However, because `Term::ANSIColor` maps everything back to the standard eight ANSI text colours and seven ANSI text styles, all of the above will also be rendered identically. See that module's documentation for details.

If `Term::ANSIColor` is not available, this option is silently ignored.

Please bear in mind that up to 10% of people using your interface will have some form of colour vision impairment, so its always a good idea to differentiate information by style *and* colour, rather than by colour alone. For example:

```
if ($dangerous_action) {
    prompt 'Really proceed?', -style=>'bold red underlined';
}
else {
    prompt 'Proceed?', -style=>'green';
}
```

Also bear in mind that (even though `-style` does support the `'blink'` style) up to 99% of people using your interface will have Flashing Text Tolerance Deficiency. Just say "no".

Specifying where to prompt

`-out => FILEHANDLE`

`-in => FILEHANDLE`

`-stdio`

The `-out` option (which has no short form) is used to specify where the prompt should be written to. If this option is not specified, prompts are written to the currently `select`−ed filehandle. The most common usage is:

```
prompt(out => *STDERR)
```

The `-in` option (which also has no short form) specifies where the input should be read from. If this option is not specified, input is read from the `*ARGV` filehandle. The most common usage is:

```
prompt(in => *STDIN)
```

in those cases where `*ARGV` has been opened to a file, but you still wish to interact with the terminal (assuming `*STDIN` is opened to that terminal).

The `-stdio` option (which again has no short form) is simply a shorthand for: `-in => *STDIN, -out => *STDOUT`. This is particularly useful when there are arguments on the commandline, but you don't want prompt to treat those arguments as filenames for magic `*ARGV` reads.

Specifying how long to wait for input

`-timeout =>` *N*

`-t`*N*

Normally, the `prompt()` function simply waits for input. However, you can use this option to specify a timeout on the read operation. If no input is received within the specified *N* seconds, the call to `prompt()` either returns the value specified by the `-default` option (if any), or else an object indicating the read failed.

Note that, if the short form is used, *N* must be an integer. If the long form is used, *N* may be an integer or floating point value.

You can determine whether an input operation timed out, even if a default value was returned, by calling the `timedout()` method on the object returned by `prompt()`:

```
if (prompt('Continue?', -y1, -timeout=>60) && !$_->timedout) {
    ...
}
```

If a time-out occurred, the return value of `timedout()` is a string describing the timeout, such as:

```
    "timed out after 60 seconds"
```

Providing a menu of responses

`-menu => SPECIFICATION`

You can limit the allowable responses to a prompt, by providing a menu.

A menu is specified using the `-menu` option, and the menu choices are specified as an argument to the option, either as a reference to an array, hash, or string, or else as a literal string.

If the menu is specified in a hash, `prompt()` displays the keys of the hash, sorted alphabetically, and with each alternative marked with a single alphabetic character (its "selector key").

For example, given:

```
    prompt 'Choose...',
           -menu=>{ 'live free'=>1, 'die'=>0, 'transcend'=>-1 },
           '>';
```

`prompt()` will display:

```
    Choose...
        a. die
        b. live free
        c. transcend
    > _
```

It will then only permit the user to enter a valid selector key (in the previous example: 'a', 'b', or 'c'). Once one of the alternatives is selected, `prompt()` will return the corresponding value from the hash (0, 1, or −1, respectively, in this case).

Note that the use of alphabetics as selector keys inherently limits the number of usable menu items to 52. See "Numeric menus" for a way to overcome this limitation.

A menu is treated like a special kind of prompt, so that any other prompt strings in the `prompt()` call will appear either before or after the menu of choices, depending on whether they appear before or after the menu specification in the call to `prompt()`.

If an array is used to specify the choices:

```
    prompt 'Choose...',
           -menu=>[ 'live free', 'die', 'transcend' ],
           '>';
```

then each array element is displayed (in the original array order) with a selector key:

```
    Choose...
        a. live free
        b. die
        c. transcend
    > _
```

and `prompt()` returns the element corresponding to the selection (i.e. it returns 'live free' if 'a' is entered, 'die' if 'b' is entered, or 'transcend' if 'c' is entered).

Hence, the difference between using an array and a hash is that the array allows you to control the order of items in the menu, whereas a hash allows you to show one thing (i.e. keys) but have something related (i.e. values) returned instead.

If the argument after `-menu` is a string or a reference to a string, the option splits the string on newlines, and treats the resulting list as if it were an array of choices. This is useful, for example, to request the user select a filename:

```
    my $files = `ls`;
    prompt 'Select a file...', -menu=>$files, '>';
```

Numbered menus

As the previous examples indicate, each menu item is given a unique alphabetic selector key. However, if the `-number` or `-integer` option is specified as well:

```
prompt 'Choose...',
       -number,
       -menu=>{ 'live free'=>1, 'die'=>0, 'transcend'=>-1 },
       '>';
```

`prompt()` will number each menu item instead, using consecutive integers as the selector keys:

```
Choose...
    1. die
    2. live free
    3. transcend
> _
```

This allows for an unlimited number of alternatives in a single menu, but prevents the use of `-single` for one-key selection from menus if the menu has more than nine items.

Hierarchical menus

If you use a hash to specify a menu, the values of the hash do not have to be strings. Instead, they can be references to nested hashes or arrays.

This allows you to create hierarchical menus, where a selection at the top level may lead to a secondary menu, etc. until an actual choice is possible. For example, the following call to prompt:

```
my $choices = {
    animates => {
        animals => {
            felines => [qw<cat lion lynx>],
            canines => [qw<dog fox wolf>],
            bovines => [qw<cow ox buffalo>],
        },
        fish => [qw<shark carp trout bream>],
    },
    inanimates => {
        rocks     => [qw<igneous metamorphic sedimentary>],
        languages => [qw<Perl Python Ruby Tcl>],
    },
};

my $result = prompt -1, 'Select a species...', -menu=>$choices, '> ';
```

might result in an interaction like this:

```
Select a species...
a.  animates
b.  inanimates
> a

Select from animates:
a.  animals
b.  fish
> b

Select from fish:
a.  shark
b.  carp
```

```
    c.  trout
    d.  bream
    > c
```

At which point, `prompt()` would return the string `'trout'`.

Note that you can nest an arbitrary number of hashes, but that each "bottom" level choice has to be either a single string, or an array of strings.

Navigating hierarchical menus

Within a hierarchical menu, the user must either select a valid option (by entering the corresponding letter), or else may request that they be taken back up a level in the hierarchy, by entering `<ESC>`. Pressing `<ESC>` at the top level of a menu causes the call to `prompt()` to immediately return with failure.

Simulating a command-line

```
-argv
```

```
-a
```

The `prompt()` subroutine can be used to request that the user provide command-line arguments interactively. When requested, the input operation is only carried out if `@ARGV` is empty.

Whatever the user enters is broken into a list and assigned to `@ARGV`.

The input is first `glob`bed for file expansions, and has any environment variables (of the form `$VARNAME` interpolated). The resulting string is then broken into individual words, except where parts of it contain single or double quotes, the contents of which are always treated as a single string.

This feature is most useful during development, to allow a program to be run from within an editor, and yet pass it a variety of command-lines. The typical usage is (at the start of a program):

```
    use IO::Prompter;
    BEGIN { prompt -argv }
```

However, because this pattern is so typical, there is a shortcut:

```
    use IO::Prompter -argv;
```

You can also specify the name with which the program args, are to be prompted, in the usual way (i.e. by providing a prompt):

```
    use IO::Prompter -argv, 'demo.pl';
```

Note, however, the critical difference between that shortcut (which calls `prompt -argv` when the module is loaded) and:

```
    use IO::Prompter [-argv];
```

(which sets `-argv` as an automatic option for every subsequent call to `prompt()` in the current lexical scope).

Note too that the `-argv` option also implies `-complete=`'filenames'>.

Input autocompletion

```
-comp[lete] => SPECIFICATION
```

When this option is specified, the `prompt()` subroutine will complete input using the specified collection of strings. By default, when completion is active, word completion is requested using the `<TAB>` key, but this can be changed by setting the `$IO_PROMPTER_COMPLETE_KEY` environment variable. Once completion has been initiated, you can use the completion key or else `<CTRL-N>` to advance to the next completion candidate. You can also use `<CTRL-P>` to back up to the previous candidate.

The specific completion mechanism can be defined either using a subroutine, an array reference, a hash reference, or a special string:

```
    Specification       Possible completions supplied by...

      sub {...}         ...whatever non-subroutine specification
                        (as listed below) is returned when the
                        subroutine is called. The subroutine is passed
                        the words of the current input text, split on
                        whitespace, as its argument list.

        [...]           ...the elements of the array

        {...}           ...the keys of the hash

     'filenames'        ...the list of files supplied by globbing the
                        last whitespace-separated word of the input text

     'dirnames'         ...the list of directories supplied by globbing the
                        last whitespace-separated word of the input text
```

If an array or hash is used, only those elements or keys that begin with the last whitespace-separated word of the current input are offered as completions.

For example:

```
    # Complete with the possible commands...
    my $next_cmd
        = prompt -complete => \%cmds;

    # Complete with valid usernames...
    my $user
        = prompt -complete => \@usernames;

    # Complete with valid directory names...
    my $file
        = prompt -complete => 'dirnames';

    # Complete with cmds on the first word, and filenames on the rest...
    my $cmdline
        = prompt -complete => sub { @_ <= 1 ? \%cmds : 'filenames' };
```

Completing from your own input history

The `prompt()` subroutine also tracks previous input and allows you to complete with that instead. No special option is required, as the feature is enabled by default.

At the start of a prompted input, the user can cycle backwards through previous inputs by pressing `<CTRL-R>` (this can be changed externally by setting the `$IO_PROMPTER_HISTORY_KEY` environment variable, or internally by assigning a new keyname to `$ENV{IO_PROMPTER_HISTORY_KEY}`). After the first `<CTRL-R>`, subsequent `<CTRL-R>`'s will recall earlier inputs. You can also use `<CTRL-N>` and `<CTRL-P>` (as in user-specified completions) to move back and forth through your input history.

If the user has already typed some input, the completion mechanism will only show previous inputs that begin with that partial input.

History sets

`-h[NAME]`
`-hist[ory] [=> NAME]`

By default, IO::Prompter tracks every call to `prompt()` within a program, and accumulates a single set of history completions for all of them. That means that, at any prompt, `<CTRL-R>` will take the user back through *every* previous input, regardless of which call to `prompt()` originally retrieved it.

Sometimes that's useful, but sometimes you might prefer that different calls to `prompt()` retained distinct memories. For example, consider the following input loop:

```
while (my $name = prompt 'Name:') {
    my $grade   = prompt 'Grade:', -integer;
    my $comment = prompt 'Comment:';
    ...
}
```

If you're entering a name, there's no point in `prompt()` offering to complete it with previous grades or comments. In fact, that's just annoying.

IO::Prompter allows you to specify that a particular call to `prompt()` belongs to a particular "history set". Then it completes input history using only the history of those calls belonging to the same history set.

So the previous example could be improved like so:

```
while (my $name = prompt 'Name:', -hNAME) {
    my $grade   = prompt 'Grade:', -hGRADE, -integer;
    my $comment = prompt 'Comment:', -hOTHER;
    ...
}
```

Now, when prompting for a name, only those inputs in the `'NAME'` history set will be offered as history completions. Likewise only previous grades will be recalled when prompting for grades and earlier only comments when requesting comments.

If you specify the `-h` or `-history` option without providing the name of the required history set, `prompt()` uses the prompt text itself as the name of the call's history set. So the previous example would work equally well if written:

```
while (my $name = prompt 'Name:', -h) {
    my $grade   = prompt 'Grade:', -h, -integer;
    my $comment = prompt 'Comment:', -h;
    ...
}
```

though now the names of the respective history sets would now be `'Name: '`, `'Grade: '`, and `'Comment: '`. This is by far the more common method of specifying history sets, with explicitly named sets generally only being used when two or more separate calls to `prompt()` have to share a common history despite using distinct prompts. For example:

```
for my $n (1..3) {
    $address .= prompt "Address (line $n):", -hADDR;
}
```

If you specify `'NONE'` as the history set, the input is not recorded in the history. This is useful when inputting passwords.

Configuring the autocompletion interaction

By default, when user-defined autocompletion is requested, the `prompt()` subroutine determines the list of possible completions, displays it above the prompt, and completes to the longest common prefix. If the completion key is pressed again immediately, the subroutine then proceeds to complete with each possible completion in a cyclic sequence. This is known as "list+longest full" mode.

On the other hand, when historical completion is requested, `prompt()` just immediately cycles through previous full inputs. This is known as "full" mode.

You can change these behaviours by setting the `$IO_PROMPTER_COMPLETE_MODES` and `$IO_PROMPTER_HISTORY_MODES` environment variables *before the module is loaded* (either in your shell, or in a `BEGIN` block before the module is imported).

Specifically, you can set the individual string values of either of these variables to a whitespace-separated sequence containing any of the following:

```
list         List all options above the input line

longest      Complete to the longest common prefix

full         Complete with each full match in turn
```

For example:

```
# Just list options without actually completing...
BEGIN{ $ENV{IO_PROMPTER_COMPLETE_MODES} = 'list'; }

# Just cycle full alternatives on each <TAB>...
BEGIN{ $ENV{IO_PROMPTER_COMPLETE_MODES} = 'full'; }

# For history completion, always start with the
# longest common prefix on the first <CTRL-R>,
# then just list the alternatives on a subsequent press...
BEGIN{ $ENV{IO_PROMPTER_HISTORY_MODES} = 'longest list'; }
```

Specifying what to return by default

`-DEF[AULT] => `*`STRING`*

`-def[ault] => `*`STRING`*

`-d`*`STRING`*

If a default value is specified, that value will be returned if the user enters an empty string at the prompt (i.e. if they just hit `<ENTER>/<RETURN>` immediately) or if the input operation times out under the `timeout` option.

Note that the default value is not added to the prompt, unless you do so yourself. A typical usage might therefore be:

```
my $frequency
    = prompt "Enter polling frequency [default: $DEF_FREQ]",
             -num, -def=>$DEF_FREQ;
```

You can determine if the default value was autoselected (as opposed to the same value being typed in explicitly) by calling the `defaulted()` method on the object returned by `prompt()`, like so:

```
if ($frequency->defaulted) {
    say "Using default frequency";
}
```

If you use the `-must` option any default value must also satisfy all the constraints you specify, unless you use the `-DEFAULT` form, which skips constraint checking when the default value is selected.

If you use the `-menu` option, the specified default value will be returned immediately `<ENTER>/<RETURN>` is pressed, regardless of the depth you are within the menu. Note that the default value specifies the value to be returned, not the selector key to be entered. The default value does not even have to be one of the menu choices.

Specifying what to echo on input

`-echo => `*`STR`*

`-e`*`STR`*

When this option is specified, the `prompt()` subroutine will echo the specified string once for each character that is entered. Typically this would be used to shroud a password entry, like so:

```
    # Enter password silently:
    my $passwd
        = prompt 'Password:', -echo=>"";

    # Echo password showing only asterisks:
    my $passwd
        = prompt 'Password:', -echo=>"*";
```

As a special case, if the `−echo` value contains a slash (/) and the any of the <−yesno> options is also specified, the substring before the slash is taken as the string to echo for a 'yes' input, and the substring after the slash is echoed for a 'no' input.

Note that this option is only available when the Term::ReadKey module is installed. If it is used when that module is not available, a warning will be issued.

Specifying how to echo on input

`−echostyle =>` *`SPECIFICATION`*

The `−echostyle` option works for the text the user types in the same way that the `−style` option works for the prompt. That is, you can specify the style and colour in which the user's input will be rendered like so:

```
    # Echo password showing only black asterisks on a red background:
    my $passwd
        = prompt 'Password:', -echo=>"*", -echostyle=>'black on red';
```

Note that `−echostyle` is completely independent of `−echo`:

```
    # Echo user's name input in bold white:
    my $passwd
        = prompt 'Name:', -echostyle=>'bold white';
```

The `−echostyle` option requires `Term::ANSIColor`, and will be silently ignored if that module is not available.

Input editing

When the Term::ReadKey module is available, `prompt()` also honours a subset of the usual input cursor motion commands:

`CTRL−B`
: Move the cursor back one character

`CTRL−F`
: Move the cursor forward one character

`CTRL−A`
: Move the cursor to the start of the input

`CTRL−E`
: Move the cursor to the end of the input

Specifying when input should fail

`−fail =>` *`VALUE`*

`−f`*`STRING`*

If this option is specified, the final input value is compared with the associated string or value, by smartmatching just before the call to `prompt()` returns. If the two match, `prompt()` returns a failure value. This means that instead of writing:

```
    while (my $cmd = prompt '>') {
        last if $cmd eq 'quit';
        ...
    }
```

you can just write:

```
    while (my $cmd = prompt '>', -fail=>'quit') {
        ...
    }
```

Constraining what can be typed

`-guar[antee] => SPEC`

This option allows you to control what input users can provide. The specification can be a regex or a reference to an array or a hash.

If the specification is a regex, that regex is matched against the input so far, every time an extra character is input. If the regex ever fails to match, the guarantee fails.

If the specification is an array, the input so far is matched against the same number of characters from the start of each of the (string) elements of the array. If none of these substrings match the input, the guarantee fails.

If the specification is a hash, the input so far is matched against the same number of characters from the start of each key of the hash. If none of these substrings match the input, the guarantee fails.

If the guarantee fails, the input is rejected (just as the `-must` option does). However, unlike `-must`, `-guarantee` rejects the input character-by-character as it typed, and *before* it is even echoed. For example, if your call to `prompt()` is:

```
    my $animal = prompt -guarantee=>['cat','dog','cow'];
```

then at the prompt:

```
    > _
```

you will only be able to type in ’c’ or ’d’. If you typed ’c’, then you would only be able to type ’a’ or ’o’. If you then typed ’o’, you would only be able to type ’w’.

In other words, `-guarantee` ensures that you can only type in a valid input, and simply ignores any typing that would not lead to such an input.

To help users get the input right, specifying `-guarantee` as an array or hash reference also automatically specifies a `-complete` option with the array or hash as its completion list as well. So, whenever a `-guarantee` is in effect, the user can usually autocomplete the acceptable inputs.

Note, however, that `-guarantee` can only reject (or autocomplete) input as it is typed if the Term::ReadKey module is available. If that module cannot be loaded, `-guarantee` only applies its test after the `<ENTER>/<RETURN>` key is pressed, and there will be no autocompletion available.

Constraining input to numbers

```
-i
-integer [=> SPEC]
-n
-num[ber] [=> SPEC]
```

If any of these options are specified, `prompt()` will only accept a valid integer or number as input, and will reprompt until one is entered.

If you need to restrict the kind of number further (say, to positive integers), you can supply an extra constraint as an argument to the long-form option. Any number entered must satisfy this constraint by successfully smart-matching it. For example:

```
$rep_count = prompt 'How many reps?', -integer => sub{ $_ > 0 };

$die_roll = prompt 'What did you roll?', -integer => [1..6];

$factor = prompt 'Prime factor:', -integer => \&is_prime;

$score = prompt 'Enter score:', -number => sub{ 0 <= $_ && $_ <= 100 };
```

If the constraint is specified as a subroutine, the entered number will be passed to it both as its single argument and in `$_`.

You cannot pass a scalar value directly as a constraint, except those strings listed below. If you want a scalar value as a constraint, use a regex or array reference instead:

```
# Wrong...
$answer = prompt "What's the ultimate answer?",
                  -integer => 42;

# Use this instead...
$answer = prompt "What's the ultimate answer?",
                 -integer => qr/^42$/;

# Or this...
$answer = prompt "What's the ultimate answer?",
                 -integer => [42];
```

Only the following strings may be passed directly as scalar value constraints. They do mot match exactly, but instead act as specifiers for one or more built-in constraints. You can also pass a string that contains two or more of them, separated by whitespace, in which case they must all be satisfied. The specifiers are:

`'pos'` or `'positive'`
: The number must be greater than zero

`'neg'` or `'negative'`
: The number must be less than zero

`'zero'`
: The number must be equal to zero

`'even'` or `'odd'`
: The number must have the correct parity

You can also prepend `"non"` to any of the above to reverse their meaning.

For example:

```
$rep_count = prompt 'How much do you bid?', -number => 'positive';

$step_value = prompt 'Next step:', -integer => 'even nonzero';
```

Constraining input to filenames

`-f`
`-filenames`

You can tell `prompt()` to accept only valid filenames, using the `-filenames` option (or its shortcut: `-f`).

This option is equivalent to the options:

```
-must => {
    'File must exist'       => sub { -e },
    'File must be readable' => sub { -r },
},
-complete => 'filenames',
```

In other words `-filenames` requires `prompt()` to accept only the name of an existing, readable file, and it also activates filename completion.

Constraining input to "keyletters"

`-k`
`-key[let[ter]][s]`

A common interaction is to offer the user a range of actions, each of which is specified by keying a unique letter, like so:

```
INPUT:
given (prompt '[S]ave, (R)evert, or (D)iscard:', -default=>'S') {
    when (/R/i) { revert_file()  }
    when (/D/i) { discard_file() }
    when (/S/i) { save_file()    }
    default     { goto INPUT;    }
}
```

This can be cleaned up (very slightly) by using a guarantee:

```
given (prompt '[S]ave, (R)evert, or (D)iscard:', -default=>'S',
              -guarantee=>qr/[SRD]/i
) {
    when (/R/i) { revert_file()  }
    when (/D/i) { discard_file() }
    default     { save_file()    }
}
```

However, it's still annoying to have to specify the three key letters twice (and the default choice three times) within the call to `prompt()`. So IO::Prompter provides an option that extracts this information directly from the prompt itself:

```
given (prompt '[S]ave, (R)evert, or (D)iscard:', -keyletters) {
    when (/R/i) { revert_file()  }
    when (/D/i) { discard_file() }
    default     { save_file()    }
}
```

This option scans the prompt string and extracts any purely alphanumeric character sequences that are enclosed in balanced brackets of any kind (square, angle, round, or curly). It then makes each of these character sequences a valid input (by implicitly setting the `-guarantee` option), and adds the first option in square brackets (if any) as the `-default` value of the prompt.

Note that the key letters don't have to be at the start of a word, don't have to be a single character, and can be either upper or lower case. For example:

```
my $action = prompt -k, '(S)ave, Save(a)ll, (Ex)it without saving';
```

Multi-character key letters are often a good choice for options with serious or irreversible consequences.

A common idiom with key letters is to use the `-single` option as well, so that pressing any key letter immediately completes the input, without the user having to also press `<ENTER>/<RETURN>`:

```
given (prompt -k1, '[S]ave, (R)evert, or (D)iscard:') {
    when (/R/i) { revert_file()   }
    when (/D/i) { discard_file()  }
    default     { save_file()     }
}
```

Preserving terminal newlines

`-l`
`-line`

The (encapsulated) string returned by `prompt()` is automatically chomped by default. To prevent that chomping, specify this option.

Constraining what can be returned

`-must => HASHREF`

This option allows you to specify requirements and constraints on the input string that is returned by `prompt()`. These limitations are specified as the values of a hash.

If the `-must` option is specified, once input is complete every value in the specified hash is smartmatched against the input text. If any of them fail to match, the input is discarded, the corresponding hash key is printed as an error message, and the prompt is repeated.

Note that the values of the constraint hash cannot be single strings or numbers, except for certain strings (such as `'pos'`, `'nonzero'`, or `'even'`, as described in "Constraining input to numbers").

If you want to constrain the input to a single string or number (a very unusual requirement), just place the value in an array, or match it with a regex:

```
# This doesn't work...
my $magic_word = prompt "What's the magic word?",
                        -must => { 'be polite' => 'please' };

# Use this instead...
my $magic_word = prompt "What's the magic word?",
                        -must => { 'be polite' => ['please'] };

# Or, better still...
my $magic_word = prompt "What's the magic word?",
                        -must => { 'be polite' => qr/please/i };
```

The `-must` option allows you to test inputs against multiple conditions and have the appropriate error messages for each displayed. It also ensures that, when `prompt()` eventually returns, you are guaranteed that the input meets all the specified conditions.

For example, suppose the user is required to enter a positive odd prime number less than 100. You could enforce that with:

```
my $opnlt100 = prompt 'Enter your guess:',
                      -integer,
                      -must => { 'be odd'                 => 'odd',
                                 'be in range'            => [1..100],
                                 'It must also be prime:' => \&isprime,
                               };
```

Note that, if the error message begins with anything except an uppercase character, the prompt is reissued followed by the error message in parentheses with the word "must" prepended (where appropriate). Otherwise, if the error message does start with an uppercase character, the prompt is not reissued and the error message is printed verbatim. So a typical input sequence for the previous example might look like:

```
    Enter your guess: 101
    Enter your guess: (must be in range) 42
    It must also be prime: 2
    Enter your guess: (must be odd) 7
```

at which point, the call to `prompt()` would accept the input and return.

See also the `-guarantee` option, which allows you to constrain inputs as they are typed, rather than after they are entered.

Changing how returns are echoed

`-r[STR]`
`-ret[urn] [=> STR]`

When `<ENTER>/<RETURN>` is pressed, `prompt()` usually echoes a carriage return. However, if this option is given, `prompt()` echoes the specified string instead. If the string is omitted, it defaults to `"\n"`.

For example:

```
    while (1) {
        my $expr = prompt 'Calculate:', -ret=>' = ';
        say evaluate($expr);
    }
```

would prompt for something like this:

```
    Calculate: 2*3+4^5_
```

and when the `<ENTER>/<RETURN>` key is pressed, respond with:

```
    Calculate: 2*3+4^5 = 1030
    Calculate: _
```

The string specified with `-return` is also automatically echoed if the `-single` option is used. So if you don't want the automatic carriage return that `-single` mode supplies, specify `-return=>""`.

Single-character input

`-s`
`-1`
`-sing[le]`

This option causes `prompt()` to return immediately once any single character is input. The user does not have to push the `<ENTER>/<RETURN>` key to complete the input operation. `-single` mode input is only available if the Term::ReadKey module can be loaded.

By default, `prompt()` echoes the single character that is entered. Use the `-echo` option to change or prevent that.

```
    # Let user navigate through maze by single, silent keypresses...
    while ($nextdir = prompt "\n", -single, -echo, -guarantee=>qr/[nsew]/) {
        move_player($nextdir);
    }
```

Unless echoing has been disabled, by default `prompt()` also supplies a carriage return after the input character. Use the `-return` option to change that behaviour. For example, this:

```
    my $question = <<END_QUESTION;
    Bast is the goddess of: (a) dogs  (b) cats  (c) cooking  (d) war?
    Your answer:
    END_QUESTION

    my $response = prompt $question, -1, -return=>' is ', -g=>['a'..'d'];
    say $response eq $answer ? 'CORRECT' : 'incorrect';
```

prompts like this:

```
    Bast is the goddess of: (a) dogs  (b) cats  (c) cooking  (d) war?
    Your answer: _
```

accepts a single character, like so:

```
    Bast is the goddess of: (a) dogs  (b) cats  (c) cooking  (d) war?
    Your answer: b_
```

and completes the line thus:

```
    Bast is the goddess of: (a) dogs  (b) cats  (c) cooking  (d) war?
    Your answer: b is CORRECT
    _
```

Returning raw data

`-v`
`-verb[atim]`

Normally, `prompt()` returns a special object that contains the text input, the success value, and other information such as whether the default was selected and whether the input operation timed out.

However, if you prefer to have `prompt()` just return the input text string directly, you can specify this option.

Note however that, under `-verbatim`, the input is still autochomped (unless you also specify the `-line` option.

Prompting on a clear screen

`-w`
`-wipe[first]`

If this option is present, `prompt()` prints 1000 newlines before printing its prompt, effectively wiping the screen clear of other text.

If the `-wipefirst` variant is used, the wipe will only occur if the particular call to `prompt()` is the first such call anywhere in your program. This is useful if you'd like the screen cleared at the start of input only, but you're not sure which call to `prompt()` will happen first: just use `-wipefirst` on all possible initial calls and only the actual first call will wipe the screen.

Requesting confirmations

`-y[n]` or `-Y[N]`
`-yes[no]` or `-Yes[No]`
`-yes[no] => COUNT` or `-Yes[No] => COUNT`

This option invokes a special mode that can be used to confirm (or deny) something. If one of these options is specified, `prompt` still returns the user's input, but the success or failure of the object returned now depends on what the user types in.

A true result is returned if `'y'` is the first character entered. If the flag includes an `n` or `N`, a false result is returned if `'n'` is the first character entered (and any other input causes the prompt to be reissued). If the option doesn't contain an `n` or `N`, any input except `'y'` is treated as a "no" and a false value is returned.

If the option is capitalized (`-Y` or `-YN`), the first letter of the input must be likewise a capital (this is a handy means of slowing down automatic unthinking `y`..."Oh no!" responses to potentially serious decisions).

This option is most often used in conjunction with the `-single` option, like so:

```
    $continue = prompt("Continue? ", -yn1);
```

so that the user can just hit `y` or `n` to continue, without having to hit `<ENTER>/<RETURN>` as well.

If the optional *COUNT* argument is supplied, the prompting is repeated that many times, with increasingly insistent requests for confirmation. The answer must be "yes" in each case for the final result to be true. For example:

```
$rm_star = prompt("Do you want to delete all files? ", -Yes=>3 );
```

might prompt:

```
Do you want to delete all files?  Y
Really?  Y
Are you sure?  Y
```

Bundling short-form options

You can bundle together any number of short-form options, including those that take string arguments. For example, instead of writing:

```
if (prompt "Continue? ", -yes, -1, -t10, -dn) {
```

you could just write:

```
if (prompt "Continue? ", -y1t10dn) {...}
```

This often does *not* improve readability (as the preceding example demonstrates), but is handy for common usages such as `-y1` ("ask for confirmation, don't require an `<ENTER>/<RETURN>`) or `-vl` ("Return a verbatim and unchomped string").

Escaping otherwise-magic options

```
-_
```

The `-_` option exists only to be an explicit no-op. It allows you to specify short-form options that would otherwise be interpreted as Perl file operators or other special constructs, simply by prepending or appending a _ to them. For example:

```
my $input
    = prompt -l_;  # option -l, not the -l file operator.
```

The following single-letter options require an underscore to chaperone them when they're on their own: `-e`, `-l`, `-r`, `-s`, `-w`, and `-y`. However, an underscore is not required if two or more are bundled together.

Useful useless uses of `prompt()`

Normally, in a void context, a call to `prompt()` issues a warning that you are doing an input operation whose input is immediately thrown away.

There is, however, one situation where this useless use of `prompt()` in a void context is actually useful:

```
say $data;
prompt('END OF DATA. Press any key to exit', -echo, -single);
exit;
```

Here, we're using prompt simply to pause the application after the data is printed. It doesn't matter what the user types in; the typing itself is the message (and the message is "move along").

In such cases, the "useless use..." warning can be suppressed using the `-void` option:

```
say $data;
prompt('END OF DATA. Press any key to exit', -echo, -single, -void);
exit;
```

Simulating input

IO::Prompter provides a mechanism with which you can "script" a sequence of inputs to an application. This is particularly useful when demonstrating software during a presentation, as you do not have to remember what to type, or concentrate on typing at all.

If you pass a string as an argument to `use IO::Prompter`, the individual lines of that string are used as successive input lines to any call to `prompt()`. So for example, you could specify several sets of input data, like so:

```
use IO::Prompter <<END_DATA
Leslie
45
165
Jessie
28
178
Dana
12
120
END_DATA
```

and then read this data in an input loop:

```
while (my $name   = prompt 'Name:') {
       my $age    = prompt 'Age:';
       my $height = prompt 'Height:';

       process($name, $age, $height);
}
```

Because the `use IO::Prompter` supplies input data, the three calls to `prompt()` will no longer read data from `*ARGV`. Instead they will read it from the supplied input data.

Moreover, each call to `prompt()` will simulate the typing-in process automatically. That is, `prompt()` uses a special input mode where, each time you press a keyboard letter, it echoes not that character, but rather the next character from the specified input. The effect is that you can just type on the keyboard at random, but have the correct input appear. This greatly increases the convincingness of the simulation.

If at any point, you hit `<ENTER>`/`<RETURN>` on the keyboard, `prompt()` finishes typing in the input for you (using a realistic typing speed), and returns the input string. So you can also just hit `<ENTER>`/`<RETURN>` when the prompt first appears, to have the entire line of input typed for you.

Alternatively, if you hit `<ESC>` at any point, `prompt()` escapes from the simulated input mode for that particular call to `prompt()`, and allows you to (temporarily) type text in directly. If you enter only a single `<ESC>`, then `prompt()` throws away the current line of simulated input; if you enter two `<ESC>`'s, the simulated input is merely deferred to the next call to `prompt()`.

All these keyboard behaviours require the Term::ReadKey module to be available. If it isn't, `prompt()` falls back on a simpler simulation, where it just autotypes each entire line for you and pauses at the end of the line, waiting for you to hit `<ENTER>`/`<RETURN>` manually.

Note that any line of the simulated input that begins with a <CTRL–D> or <CTRL–Z> is treated as an input failure (just as if you'd typed that character as input).

DIAGNOSTICS

All non-fatal diagnostics can be disabled using a `no warnings` with the appropriate category.

`prompt(): Can't open *ARGV: %s`

(F) By default, `prompt()` attempts to read input from the `*ARGV` filehandle. However, it failed to open that filehandle. The reason is specified at the end of the message.

`prompt(): Missing value for %s (expected %s)`

(F) A named option that requires an argument was specified, but no argument was provided after the option. See "Summary of options".

`prompt(): Invalid value for %s (expected %s)`

(F) The named option specified expects an particular type of argument, but found one of an incompatible type instead. See "Summary of options".

`prompt(): Unknown option %s ignored`
: (W misc) `prompt()` was passed a string starting with a hyphen, but could not parse that string as a valid option. The option may have been misspelt. Alternatively, if the string was supposed to be (part of) the prompt, it will be necessary to use the `-prompt` option to specify it.

`prompt(): Unexpected argument (% ref) ignored`
: (W reserved) `prompt()` was passed a reference to an array or hash or subroutine in a position where an option flag or a prompt string was expected. This may indicate that a string variable in the argument list didn't contain what was expected, or a reference variable was not properly dereferenced. Alternatively, the argument may have been intended as the argument to an option, but has become separated from it somehow, or perhaps the option was deleted without removing the argument as well.

`Useless use of prompt() in void context`
: (W void) `prompt()` was called but its return value was not stored or used in any way. Since the subroutine has no side effects in void context, calling it this way achieves nothing. Either make use of the return value directly or, if the usage is deliberate, put a `scalar` in front of the call to remove the void context.

`prompt(): -default value does not satisfy -must constraints`
: (W misc) The `-must` flag was used to specify one or more input constraints. The `-default` flag was also specified. Unfortunately, the default value provided did not satisfy the requirements specified by the `-must` flag. The call to `prompt()` will still go ahead (after issuing the warning), but the default value will never be returned, since the constraint check will reject it. It is probably better simply to include the default value in the list of constraints.

`prompt(): -keyletters found too many defaults`
: (W ambiguous) The `-keyletters` option was specified, but analysis of the prompt revealed two or more character sequences enclosed in square brackets. Since such sequences are taken to indicate a default value, having two or more makes the default ambiguous. The prompt should be rewritten with no more than one set of square brackets.

`Warning: next input will be in plaintext`
: (W bareword) The `prompt()` subroutine was called with the `-echo` flag, but the Term::ReadKey

module was not available to implement this
feature. The input will proceed as normal, but
this warning is issued to ensure that the user
doesn't type in something secret, expecting it
to remain hidden, which it won't.

`prompt(): Too many menu items. Ignoring the final %d`
(W misc) A `-menu` was specified with more than 52 choices.
Because, by default, menus use upper and lower–
case alphabetic characters as their selectors,
there were no available selectors for the extra
items after the first 52. Either reduce the number
of choices to 52 or less, or else add the
`-number` option to use numeric selectors instead.

CONFIGURATION AND ENVIRONMENT

IO::Prompter can be configured by setting any of the following environment variables:

`$IO_PROMPTER_COMPLETE_KEY`
Specifies the key used to initiate user-specified completions. Defaults to <TAB>

`$IO_PROMPTER_HISTORY_KEY`
Specifies the key used to initiate history completions. Defaults to <CTRL–R>

`$IO_PROMPTER_COMPLETE_MODES`
Specifies the response sequence for user-defined completions. Defaults to `'list+longest full'`

`$IO_PROMPTER_HISTORY_MODES`
Specifies the response sequence for history completions. Defaults to `'full'`.

DEPENDENCIES

Requires the Contextual::Return module.

The module also works much better if Term::ReadKey is available (though this is not essential).

INCOMPATIBILITIES

This module does not play well with Moose (or more specifically, with Moose::Exporter) because both of them try to play sneaky games with Scalar::Util::blessed.

The current solution is to make sure that you load Moose before loading IO::Prompter. Even just doing this:

```
    use Moose ();
    use IO::Prompter;
```

is sufficient.

BUGS AND LIMITATIONS

No unresolved bugs have been reported.

Please report any bugs or feature requests to `bug-io-prompter@rt.cpan.org`, or through the web interface at <http://rt.cpan.org>.

AUTHOR

Damian Conway `<DCONWAY@CPAN.org>`

LICENCE AND COPYRIGHT

Copyright (c) 2009, Damian Conway `<DCONWAY@CPAN.org>`. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DISCLAIMER OF WARRANTY

BECAUSE THIS SOFTWARE IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED