NAME

Term::ANSIColor - Color screen output using ANSI escape sequences

SYNOPSIS

```
use Term::ANSIColor;
print color 'bold blue';
print "This text is bold blue.\n";
print color 'reset';
print "This text is normal.\n";
print colored ("Yellow on magenta.", 'yellow on_magenta'), "\n";
print "This text is normal.\n";
print colored ['yellow on_magenta'], 'Yellow on magenta.';
print "\n";

use Term::ANSIColor qw(uncolor);
print uncolor ('01;31'), "\n";

use Term::ANSIColor qw(colorstrip);
print colorstrip '\e[1mThis is bold\e[0m', "\n";

use Term::ANSIColor qw(colorvalid);
my $valid = colorvalid ('blue bold', 'on_magenta');
print "Color string is ", $valid ? "valid\n" : "invalid\n";

use Term::ANSIColor qw(:constants);
print BOLD, BLUE, "This text is in bold blue.\n", RESET;

use Term::ANSIColor qw(:constants);
{
    local $Term::ANSIColor::AUTORESET = 1;
    print BOLD BLUE "This text is in bold blue.\n";
    print "This text is normal.\n";
}

use Term::ANSIColor qw(:pushpop);
print PUSHCOLOR RED ON_GREEN "This text is red on green.\n";
print PUSHCOLOR BLUE "This text is blue on green.\n";
print RESET BLUE "This text is just blue.\n";
print POPCOLOR "Back to red on green.\n";
print LOCALCOLOR GREEN ON_BLUE "This text is green on blue.\n";
print "This text is red on green.\n";
{
    local $Term::ANSIColor::AUTOLOCAL = 1;
    print ON_BLUE "This text is red on blue.\n";
    print "This text is red on green.\n";
}
print POPCOLOR "Back to whatever we started as.\n";
```

DESCRIPTION

This module has two interfaces, one through color() and colored() and the other through constants. It also offers the utility functions uncolor(), colorstrip(), and colorvalid(), which have to be explicitly imported to be used (see *SYNOPSIS*).

Function Interface

color() takes any number of strings as arguments and considers them to be space-separated lists of attributes. It then forms and returns the escape sequence to set those attributes. It doesn't print it out, just returns it, so you'll have to print it yourself if you want to (this is so that you can save it as a string, pass it to something else, send it to a file handle, or do anything else with it that you might care to). color() throws an exception if given an invalid attribute, so you can also use it to check attribute names for validity (see *EXAMPLES*).

uncolor() performs the opposite translation, turning escape sequences into a list of strings.

colorstrip() removes all color escape sequences from the provided strings, returning the modified strings separately in array context or joined together in scalar context. Its arguments are not modified.

colorvalid() takes attribute strings the same as color() and returns true if all attributes are known and false otherwise.

The recognized non-color attributes are clear, reset, bold, dark, faint, underline, underscore, blink, reverse, and concealed. Clear and reset (reset to default attributes), dark and faint (dim and saturated), and underline and underscore are equivalent, so use whichever is the most intuitive to you. The recognized foreground color attributes are black, red, green, yellow, blue, magenta, cyan, and white. The recognized background color attributes are on_black, on_red, on_green, on_yellow, on_blue, on_magenta, on_cyan, and on_white. Case is not significant.

Note that not all attributes are supported by all terminal types, and some terminals may not support any of these sequences. Dark and faint, blink, and concealed in particular are frequently not implemented.

Attributes, once set, last until they are unset (by sending the attribute `clear` or `reset`). Be careful to do this, or otherwise your attribute will last after your script is done running, and people get very annoyed at having their prompt and typing changed to weird colors.

As an aid to help with this, colored() takes a scalar as the first argument and any number of attribute strings as the second argument and returns the scalar wrapped in escape codes so that the attributes will be set as requested before the string and reset to normal after the string. Alternately, you can pass a reference to an array as the first argument, and then the contents of that array will be taken as attributes and color codes and the remainder of the arguments as text to colorize.

Normally, colored() just puts attribute codes at the beginning and end of the string, but if you set $Term::ANSIColor::EACHLINE to some string, that string will be considered the line delimiter and the attribute will be set at the beginning of each line of the passed string and reset at the end of each line. This is often desirable if the output contains newlines and you're using background colors, since a background color that persists across a newline is often interpreted by the terminal as providing the default background color for the next line. Programs like pagers can also be confused by attributes that span lines. Normally you'll want to set $Term::ANSIColor::EACHLINE to `"\n"` to use this feature.

Constant Interface

Alternately, if you import `:constants`, you can use the constants CLEAR, RESET, BOLD, DARK, FAINT, UNDERLINE, UNDERSCORE, BLINK, REVERSE, CONCEALED, BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, ON_BLACK, ON_RED, ON_GREEN, ON_YELLOW, ON_BLUE, ON_MAGENTA, ON_CYAN, and ON_WHITE directly. These are the same as color('attribute') and can be used if you prefer typing:

```
print BOLD BLUE ON_WHITE "Text", RESET, "\n";
```

to

```
print colored ("Text", 'bold blue on_white'), "\n";
```

(Note that the newline is kept separate to avoid confusing the terminal as described above since a background color is being used.)

When using the constants, if you don't want to have to remember to add the `, RESET` at the end of each print line, you can set $Term::ANSIColor::AUTORESET to a true value. Then, the display mode will automatically be reset if there is no comma after the constant. In other words, with that variable set:

```
print BOLD BLUE "Text\n";
```

will reset the display mode afterward, whereas:

```
print BOLD, BLUE, "Text\n";
```

will not. If you are using background colors, you will probably want to print the newline with a separate print statement to avoid confusing the terminal.

The subroutine interface has the advantage over the constants interface in that only two subroutines are exported into your namespace, versus twenty-two in the constants interface. On the flip side, the constants interface has the advantage of better compile time error checking, since misspelled names of colors or attributes in calls to color() and colored() won't be caught until runtime whereas misspelled names of constants will be caught at compile time. So, pollute your namespace with almost two dozen subroutines that you may not even use that often, or risk a silly bug by mistyping an attribute. Your choice, TMTOWTDI after all.

The Color Stack

As of Term::ANSIColor 2.0, you can import `:pushpop` and maintain a stack of colors using PUSHCOLOR, POPCOLOR, and LOCALCOLOR. PUSHCOLOR takes the attribute string that starts its argument and pushes it onto a stack of attributes. POPCOLOR removes the top of the stack and restores the previous attributes set by the argument of a prior PUSHCOLOR. LOCALCOLOR surrounds its argument in a PUSHCOLOR and POPCOLOR so that the color resets afterward.

When using PUSHCOLOR, POPCOLOR, and LOCALCOLOR, it's particularly important to not put commas between the constants.

```
print PUSHCOLOR BLUE "Text\n";
```

will correctly push BLUE onto the top of the stack.

```
print PUSHCOLOR, BLUE, "Text\n";    # wrong!
```

will not, and a subsequent pop won't restore the correct attributes. PUSHCOLOR pushes the attributes set by its argument, which is normally a string of color constants. It can't ask the terminal what the current attributes are.

DIAGNOSTICS

Bad escape sequence %s

(F) You passed an invalid ANSI escape sequence to uncolor().

Bareword "%s" not allowed while "strict subs" in use

(F) You probably mistyped a constant color name such as:

```
$Foobar = FOOBAR . "This line should be blue\n";
```

or:

```
@Foobar = FOOBAR, "This line should be blue\n";
```

This will only show up under use strict (another good reason to run under use strict).

Invalid attribute name %s

(F) You passed an invalid attribute name to either color() or colored().

Name "%s" used only once: possible typo

(W) You probably mistyped a constant color name such as:

```
print FOOBAR "This text is color FOOBAR\n";
```

It's probably better to always use commas after constant names in order to force the next error.

No comma allowed after filehandle

(F) You probably mistyped a constant color name such as:

```
print FOOBAR, "This text is color FOOBAR\n";
```

Generating this fatal compile error is one of the main advantages of using the constants interface, since you'll immediately know if you mistype a color name.

No name for escape sequence %s

(F) The ANSI escape sequence passed to uncolor() contains escapes which aren't recognized and can't be translated to names.

ENVIRONMENT

ANSI_COLORS_DISABLED

If this environment variable is set, all of the functions defined by this module (color(), colored(), and all of the constants not previously used in the program) will not output any escape sequences and instead will just return the empty string or pass through the original text as appropriate. This is intended to support easy use of scripts using this module on platforms that don't support ANSI escape sequences.

For it to have its proper effect, this environment variable must be set before any color constants are used in the program.

RESTRICTIONS

It would be nice if one could leave off the commas around the constants entirely and just say:

```
print BOLD BLUE ON_WHITE "Text\n" RESET;
```

but the syntax of Perl doesn't allow this. You need a comma after the string. (Of course, you may consider it a bug that commas between all the constants aren't required, in which case you may feel free to insert commas unless you're using $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR.)

For easier debugging, you may prefer to always use the commas when not setting $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR so that you'll get a fatal compile error rather than a warning.

NOTES

The codes generated by this module are standard terminal control codes, complying with ECMA-048 and ISO 6429 (generally referred to as "ANSI color" for the color codes). The non-color control codes (bold, dark, italic, underline, and reverse) are part of the earlier ANSI X3.64 standard for control sequences for video terminals and peripherals.

Note that not all displays are ISO 6429-compliant, or even X3.64-compliant (or are even attempting to be so). This module will not work as expected on displays that do not honor these escape sequences,

such as cmd.exe, 4nt.exe, and command.com under either Windows NT or Windows 2000. They may just be ignored, or they may display as an ESC character followed by some apparent garbage.

Jean Delvare provided the following table of different common terminal emulators and their support for the various attributes and others have helped me flesh it out:

	clear	bold	faint	under	blink	reverse	conceal
xterm	yes	yes	no	yes	yes	yes	yes
linux	yes	yes	yes	bold	yes	yes	no
rxvt	yes	yes	no	yes	bold/black	yes	no
dtterm	yes	yes	yes	yes	reverse	yes	yes
teraterm	yes	reverse	no	yes	rev/red	yes	no
aixterm	kinda	normal	no	yes	no	yes	yes
PuTTY	yes	color	no	yes	no	yes	no
Windows	yes	no	no	no	no	yes	no
Cygwin SSH	yes	yes	no	color	color	color	yes
Mac Terminal	yes	yes	no	yes	yes	yes	yes

Windows is Windows telnet, Cygwin SSH is the OpenSSH implementation under Cygwin on Windows NT, and Mac Terminal is the Terminal application in Mac OS X. Where the entry is other than yes or no, that emulator displays the given attribute as something else instead. Note that on an aixterm, clear doesn't reset colors; you have to explicitly set the colors back to what you want. More entries in this table are welcome.

Note that codes 3 (italic), 6 (rapid blink), and 9 (strike-through) are specified in ANSI X3.64 and ECMA-048 but are not commonly supported by most displays and emulators and therefore aren't supported by this module at the present time. ECMA-048 also specifies a large number of other attributes, including a sequence of attributes for font changes, Fraktur characters, double-underlining, framing, circling, and overlining. As none of these attributes are widely supported or useful, they also aren't currently supported by this module.

SEE ALSO

ECMA-048 is available on-line (at least at the time of this writing) at *http://www.ecma-international.org/publications/standards/ECMA-048.HTM*.

ISO 6429 is available from ISO for a charge; the author of this module does not own a copy of it. Since the source material for ISO 6429 was ECMA-048 and the latter is available for free, there seems little reason to obtain the ISO standard.

The current version of this module is always available from its web site at *http://www.eyrie.org/~eagle/software/ansicolor/*. It is also part of the Perl core distribution as of 5.6.0.

AUTHORS

Original idea (using constants) by Zenin, reimplemented using subs by Russ Allbery <rra@stanford.edu>, and then combined with the original idea by Russ with input from Zenin. Russ Allbery now maintains this module.

COPYRIGHT AND LICENSE

PUSHCOLOR, POPCOLOR, and LOCALCOLOR were contributed by openmethods.com voice solutions.