Table of Contents

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Chapter 2. User Interaction

Hacks 12-18

Without users, there'd be few reasons to write programs. Without users—and this includes *you*—there'd be few bugs reported for weird error messages, strange behaviors, and classic "What were you thinking and why did it do that?" moments.

Your programs don't have to be that way. You can make your users happy, make your code work where it has to work, and even make pretty graphics with Perl, all by mastering a few tricks and tips. When your program has to interact with a real person somewhere, do it with style. People may not notice when your code just stays out of their way, but you'll know by their happy glows of productivity.

Hack 12. Use $EDITOR As Your UI

Nothing beats your favorite editor for editing text.

If you live on the command line and have a reputation for turning your favorite beverage[1] into code, you're likely pretty handy on the keyboard. If you're a relentless automator, you probably have dozens of little programs and aliases to make your life easier.

[1] Your author recommends peppermint tea.

Sometimes they need arguments. Yet beyond a certain point, prompting for arguments every time or inventing more and more command-line options just doesn't work anymore. Before you resign yourself to the fate of writing a little GUI or a web frontend, consider using a more comfortable user interface instead—your preferred text editor.

The Hack

Suppose you have a series of little programs for updating your web site. Your workflow is to create a small YAML file with a new posting, then run that data through a template, update the index, and copy those pages to your server. Instead of copying a blank YAML file (or trying to recreate the necessary fields and formatting by hand), just launch an editor.

For example, a simple news site might have entries that need only a title, the date of posting, and a multiline block of text to run through some formatter. Easy:

```
use YAML 'DumpFile';
use POSIX 'strftime';

local $YAML::UseBlock = 1;

exit 1 unless -d 'posts';
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
my @posts = <posts/*.yaml>;
my $file  = 'posts/' . ( @posts + 1 ) . '.yaml';

my $fields =
{
    title => '',
    date  => strftime( '%d %B %Y', localtime( ) ),
    text  => "\\n\\n",
};

DumpFile( $file, $fields );

system( $ENV{EDITOR}, $file ) = = 0
    or die "Error launching $ENV{EDITOR}: $!\\n";
```

Assuming you have the `EDITOR` environment variable set to your preferred editor, this program creates a new blank post in the *posts/* subdirectory with the appropriate id (monotonically increasing, of course), then drops you in your editor to edit the YAML file. It has already populated the date field with the current date in the proper format. Additionally, setting `$YAML::UseBlock` to a true value makes `YAML` treat the multiline `text` string as a YAML heredoc, making it much easier to edit.

Running the Hack

From the proper directory, just run the program. It will launch a new editor on the file. When you've finished editing, save and quit, and the program will continue.

This may work very differently on non-Unix systems.

Hacking the Hack

You don't have to give up on error checking even without a formal GUI. If you can't read in the YAML file or don't have all of the right fields filled in, you can rewrite the file with as much or as little information as you like, prompting the user to try again. You can even add comments or special fields to the file explaining the error.

To read in the file, just call `LoadFile` with the filename—then continue as normal, as if the user hadn't had to create the file.

Hack 13. Interact Correctly on the Command Line

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Be kind to other programs.

Command-line programs that expect input from the keyboard are easy, right? Certainly they're easier than writing good GUI applications, right? Not necessarily. The Unix command line is flexible and powerful, but that flexibility can break naively written programs.

Prompting for interactive input in Perl typically looks like:

```
print "> ";
while (my $next_cmd = <>)
{
    chomp $next_cmd;
    process($next_cmd);
    print "> ";
}
```

If your program needs to handle noninteractive situations as well, things get a whole lot more complicated. The usual solution is something like:

```
print "> " if -t *ARGV && -t select;
while (my $next_cmd = <>)
{
    chomp $next_cmd;
    process($next_cmd);
    print "> " if -t *ARGV && -t select;
}
```

The `-t` test checks whether its filehandle argument is connected to a terminal. To handle interactive cases correctly, you need to check both that you're reading from a terminal (`-t *ARGV`) and that you're writing to one (`-t select`). It's a common mistake to mess those tests up, and write instead:

```
print "> " if -t *STDIN && -t *STDOUT;
```

The problem is that the <> operator doesn't read from `STDIN`; it reads from `ARGV`. If there are filenames specified on the command line, those two filehandles aren't the same. Likewise, although `print` usually writes to `STDOUT`, it won't if you've explicitly `select`-ed some other destination. You need to call `select` with no arguments to get the filehandle which each `print` will currently target.

Worse, still, even the correct version:

```
print "> " if -t *ARGV && -t select;
```

doesn't always work correctly. That's because the `ARGV` filehandle is magically self-opening, but only magically self-opens during the first read operation on it. If you haven't already done at least one <> before you start prompting for input, then the `ARGV` handle won't be open yet, so the first `-t *ARGV` test (the one before the `while` loop) won't be true, and the first prompt won't print.

To accurately test if an application is running interactively in all possible circumstances, you need an elaborate nightmare:

```
use Scalar::Util qw( openhandle );

sub is_interactive
{
    # Not interactive if output is not to terminal...
    return 0 if not -t select;

    # If *ARGV is opened, we're interactive if...
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
    if (openhandle *ARGV)
    {
        # ...it's currently opened to the magic '-' file
        #    and the standard input is interactive...
        return -t *STDIN if defined $ARGV && $ARGV eq '-';

        # ...or it's at end-of-file and the next file
        #    is the magic '-' file...
        return @ARGV>0 && $ARGV[0] eq '-' && -t *STDIN if eof *ARGV;

        # ...or it's directly attached to the terminal
        return -t *ARGV;
    }

    # If *ARGV isn't opened, it will be interactive if *STDIN is
    # attached to a terminal and either there are no files specified
    # on the command line or if there are files and the first is the
    # magic '-' file...
    else
    {
        return -t *STDIN && (@ARGV= =0 || $ARGV[0] eq '-');
    }
}
```

The Hack

Of course, no one wants to reinvent *that* for each project, so there's a CPAN module that does it for you:

```
use IO::Interactive qw( is_interactive );

print "> " if is_interactive;
while (my $next_cmd = <>)
{
    chomp $next_cmd;
    process($next_cmd);
    print "> " if is_interactive;
}
```

The Hack

The module has a second interface that's even Lazier. Instead of an explicit interactivity test, it can provide you with a writable filehandle that implicitly tests for interactivity:

```
use IO::Interactive qw( interactive );

print {interactive} "> ";
while (my $next_cmd = <>)
{
    chomp $next_cmd;
    process($next_cmd);
    print {interactive} "> ";
}
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Hack 14. Simplify Your Terminal Interactions

Read data from users correctly, effectively, and without thinking about it.

Even when you know the right way to handle interactive I/O [Hack #13], the resulting code can still be frustratingly messy:

```
my $offset;
print "Enter an offset: " if is_interactive;
GET_OFFSET:
while (<>)
{
    chomp;
    if (m/\\A [+-] \\d+ \\z/x)
    {
        $offset = $_;
        last GET_OFFSET;
    }
    print "Enter an offset (please enter an integer): "
        if is_interactive;
}
```

You can achieve exactly the same effect (and much more) with the `prompt( )` subroutine provided by the `IO::Prompt` CPAN module. Instead of all the above infrastructure code, just write:

```
use IO::Prompt;

my $offset = prompt( "Enter an offset: ", -integer );
```

`prompt( )` prints the string you give it, reads a line from standard input, chomps it, and then tests the input value against any constraint you specify (for example, `-integer`). If the constraint is not satisfied, the prompt repeats, along with a clarification of what was wrong. When the user finally enters an acceptable value, `prompt( )` returns it.

Most importantly, `prompt( )` is smart enough not to bother writing out any prompts if the application isn't running interactively, so you don't have to code explicitly for that case.

Infrastructure code is code that doesn't actually contribute to solving your problem, but merely exists to hold your program together. Typically this kind of code implements standard low-level tasks that probably ought to have built-ins dedicated to them. Many modules in the standard library and on CPAN exist solely to provide cleaner alternatives to continually rewriting your own infrastructure code. Discovering and using them can significantly decrease both the size and cruftiness of your code.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Train -req

`prompt( )` has a general mechanism for telling it what kind of input you need and how to ask for that input. For example:

```
my $hex_num = prompt( "Enter a hex number> ",
          -req => { "A *hex* number please!> " => qr/^[0-9A-F]+$/i }
          );

print "That's ", hex($hex_num), " in base 10\\n";
```

When this code executes, you will see something like:

```
Enter a hex number> 2B|!2B
A *hex* number please!> C3P0
A *hex* number please!> 124C1
That's 74945 in base 10
```

The `-req` argument takes a hash reference, in which each value is something to test the input against, and each key is a secondary prompt to print when the test fails. The tests can be regexes (which the input must match) or subroutines (which receive the input as `$_` and should return true if that input satisfies the constraint). For example:

```
my $factor = prompt( "Enter a prime: ",
                     -req => { "Try again: " => sub { is_prime($_) } }
                   );
```

Yea or Nay

One particularly useful constraint that `prompt( )` supports is a mode that accepts only the letters `y` or `n` as input:

```
if (prompt -YESNO, "Quit? ")
{
    save_changes($changes)
        if $changes && prompt -yes, "Save changes? ";
    print "Changes: $changes\\n";
    exit;
}
```

The first call to `prompt( )` requires the user to type a word beginning with `Y` or `N`. It will ignore anything else and return the prompt with an explanation. If the input is `Y`, the call will return true; if `N`, it will return false. On the other hand, the second call (with the `-yes` argument) actually accepts any input. If that string starts with a `y` or `Y`, `prompt( )` returns true; for any other input, it returns false. For example:

```
Quit? q
Quit? (Please enter 'Y' or 'N') Y
Save changes? n
Changes: not saved
```

These different combinations of `-YES`/`-yes`/`-no`/`-NO` allow for varying degrees of punctiliousness in obtaining the user's consent. In particular, using `-YESNO` forces users to hit Shift and one of only two possible keys, which often provides enough of a pause to prevent unthinking responses that they'll deeply regret about 0.1 seconds after hitting Enter.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

At the Touch of a Button

On the other hand, sometimes it's immensely annoying to have to press Enter at all. Sometimes you want to hit a single key and just let the application get on with things. Thus `prompt( )` provides a single character mode:

```
for my $file (@matching_files)
{
    next unless prompt -one_char, -yes, "Copy $file? ";
    copy($file, "$backup_dir/$file");
}
```

With `-one_char` in effect, the first typed character completes the entire input operation. In this case, `prompt( )` returns true only if that character was `y` or `Y`.

Of course, single character mode can accept more than just `y` and `n`. For example, the following call allows the user to select a drive instantly, simply by typing its single character name (in upper- or lowercase):

```
my $drive = uc prompt "Select a drive: ",
                      -one_char,
                      -req => { "Please select A-F: " => qr/[A-F]/i };
```

Engage Cloaking Device

You can tell `prompt( )` not to echo input (good for passwords):

```
my $passwd = prompt( "First password: ", -echo=>"" );
```

or to echo something different in place of what you actually type (also good for passwords):

```
my $passwd = prompt( "Second password: ", -echo=>"*" );
```

This allows you to produce interfaces like:

```
First password:
Second password: ********
```

What's On the Menu?

Often you can't rely on users to type in the right responses; it's easier to list them and ask the user to choose. This is menu-driven interaction, and `prompt( )` supports various forms of it. The simplest is just to give the subroutine a list of possible responses in an array:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
my $device = prompt 'Activate which device?',
                    -menu =>
                    [
                        'Sharks with "laser" beams',
                        'Disinhibiter gas grenades',
                        'Death ray',
                        'Mirror ball',
                    ];

print "Activating $device in 10:00 and counting...\\n";
```

This produces the request:

```
Activate which device?
  a. Sharks with "laser" beams
  b. Disinhibiter gas grenades
  c. Death ray
  d. Mirror ball
> q
(Please enter a-d) > d
Activating Mirror ball in 10:00 and counting...
```

The menu call to `prompt` only accepts characters in the range displayed, and returns the value corresponding to the character entered.

You can also pass the `-menu` option a hash reference:

```
my $device = prompt 'Initiate which master plan?',
                    -menu =>
                    {
                        Cousteau => 'Sharks with "laser" beams',
                        Libido   => 'Disinhibiter gas grenades',
                        Friar    => 'Death ray',
                        Shiny    => 'Mirror ball',
                    };

print "Activating $device in 10:00 and counting...\\n";
```

in which case it will show the list of keys and return the value corresponding to the key selected:

```
Initiate which master plan?
 a. Cousteau
 b. Friar
 c. Libido
 d. Shiny
> d
Activating Mirror ball in 10:00 and counting...
```

You can even nest hashes and arrays:

```
my $device = prompt 'Select your platform:',
                    -menu =>
                    {
                        Windows => [ 'WinCE', 'WinME', 'WinNT' ],
                        MacOS   => {
                                     'MacOS 9' => 'Mac (Classic)',
                                     'MacOS X' => 'Mac (New Age)',
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
                        },
            Linux   => 'Linux',
        };
```

to create hierarchical menus:

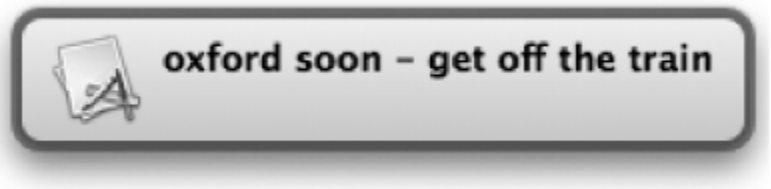
```
Select your platform:
 a. Linux
 b. MacOS
 c. Windows
> b

MacOS:
 a. MacOS 9
 b. Mac OS X
> b

Compiling for Mac (New Age)...
```

Hack 15. Alert Your Mac

Schedule GUI alerts from the command line.

Growl (http://www.growl.info/) is a small utility for Mac OS X that allows any application to send notifications to the user. The notifications pop up as a small box in a corner of the screen, overlayed on the current active window (as shown in Figure 2-1).

Figure 2-1. A simple Growl notification



The Hack

You can send Growl notifications from Perl, thanks to Chris Nandor's `Mac::Growl`. The first thing you have to do is tell Growl that your script wants to send notifications. The following code registers a script named `growlalert` and tells Growl that it sends `alert` notifications:

```
use Mac::Growl;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
Mac::Growl::RegisterNotifications(
    'growlalert', # application name
    [ 'alert' ],  # notifications this app sends
    [ 'alert' ],  # enable these notifications
);
```

Growl displays a notification to let you know the script has registered successfully (Figure 2-2). You need only register an application once on each machine that uses it.

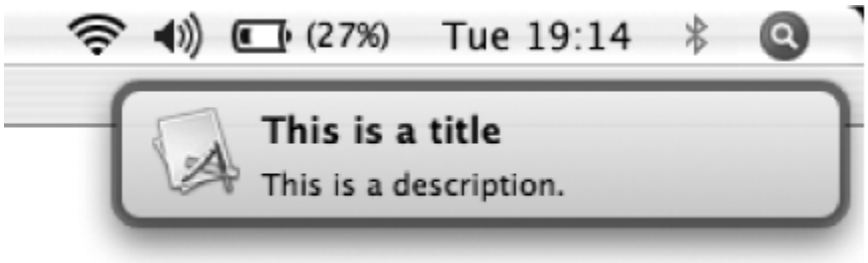
Figure 2-2. A newly registered application

When you want to send a notification, call `PostNotification( )`, passing the name of the script, the kind of notification to send, a title, and a description:

```
Mac::Growl::PostNotification(
    'growlalert', # application name
    'alert',      # type of notification
    "This is a title",
    "This is a description.",
);
```

This will pop up a notification window (Figure 2-3) and fade it out again after a few seconds.

Figure 2-3. Notification with title and description



Running the Hack

You might want a small script that sends you an alert after a time delay. The following command-line utility takes a time period to delay and a message to display in the alert. It calculates the time when the alert should appear, then forks to return control of the terminal window to the user. The forked child sleeps the requested amount of time, and then posts a Growl notification with the message as the title and no description.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
my %seconds_per =
(
    's'   => 1,
    'm'   => 60,
    'h'   => 60*60,
);

my ( $period, @message ) = @ARGV;
my ( $number, $unit )    = ( $period =~ m/^([\\.\\d]+)(.*)$/ )
      or die "usage: ga number[smh] message\\n";
$unit ||= 's';

my $growl_time = $number * $seconds_per{$unit};

my $pid        = fork;
die "fork failed ($!)\\n" unless defined $pid;

unless ( $pid )
{
    require Mac::Growl;
    sleep $growl_time;

    Mac::Growl::PostNotification(
        'growlalert', # application name
        'alert',      # type of notification
        "@message",   # title
        "",           # no description
        1,            # notification is sticky
    );
}
```

The additional argument passed to `PostNotification` tells Growl that the notification should stay on the screen until the user clicks it, instead of fading after a few seconds.

Some common uses of this program are:

```
$ ga 5m coffee
$ ga 2.5h 'oxford soon - get off the train'
```

Hacking the Hack

For Unix systems running the X Window system, use the `xmessage` command instead of `Mac::Growl`:

```
unless ( $pid )
{
    sleep $growl_time;
    system( 'xmessage', @message );
}
```

You can get creative there, popping up the window near the cursor or automatically fading out after a specified period of time.

On Windows systems running the messenger service, the proper invocation is something like:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
unless ( $pid )
{
    sleep $growl_time;
    system( qw( cmd net send localhost ), @message );
}
```

Hack 16. Interactive Graphical Apps

Paint pretty pictures with Perl.

People often see Perl as a general-purpose language: you start by using it to write short scripts, do administrative tasks, or text processing. If you happen to appreciate it, you end up enjoying its flexibility and power to perform almost anything that doesn't require the speed of compiled binaries.

Consider instead the number one requirement of games. Unless you're exclusively a fan of card games, you'd say "CPU power." Fortunately a crazy guy named David J. Goehrig had the mysterious idea to bind the functions of the C low-level graphical library SDL for the Perl language. The result is an object-oriented approach to SDL called *sdlperl*.

Blitting with SDL Perl

With SDL you will manipulate *surfaces*. These are rectangular images, and the most common operation is to copy one onto another; this is *blitting*.[2] To implement a basic image loader with SDL Perl in just four non-comment lines of code, write:

[2] A blit is the action of copying a series of bits between memory addresses. The main goal of this operation is to perform the copy as fast as possible.

```
use SDL::App;

# open a 640x480 window for your application
our $app = SDL::App->new(-width => 640, -height => 480);

# create a surface out of an image file specified on the command-line
our $img = SDL::Surface->new( -name => $ARGV[0] );

# blit the surface onto the window of your application
$img->blit( undef, $app, undef );

# flush all pending screen updates
$app->flip( );

# sleep for 3 seconds to let the user view the image
sleep 3;
```

You might wonder how to perform positioning and cropping during a blit. In the previous code, replace the two `undef` parameter values with instances of `SDL::Rect`, the first one specifying the *rectangle* to copy from the source *surface*, and the second specifying the *rectangle* where to blit on the destination *surface*. When you use `undef` instead, SDL uses top-left positioning and full sizing. Here's a blit replacement that specifies a 100x100 area in the source *surface* at a horizontal offset of 200 pixels:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
$img->blit( SDL::Rect->new(
    -width => 100, -height => 100, -x => 200, -y => 0
), $app, undef);
```

Animating with SDL Perl

You're already closer than you think to being able to write a full-fledged game with SDL Perl. The previous example opened the application, created a *surface* from an image file, and showed it onscreen. Add sound and input handling, and you're (mostly) done! Sound is too easy to use to show here; input handling requires the proper monitoring of *events* reported by an instance of `SDL::Event`.

The simplest option for input handling is to loop waiting for new events. This is fine if you have no animated sprites on screen (movement will block while you wait on the loop), but if you need to handle animations, you need a main loop that performs more steps:

- Erase all sprites at their current position.
- Check *events* to see if user interaction changes anything in the game.
- Move sprites and update game states.
- Draw all sprites at their new position.
- Tell SDL Perl to display any changes on screen.
- Synchronize the animation by sleeping for a short amount of time, corresponding to the target animation speed.

Iterate forever through these steps, and you have the core of a game engine.

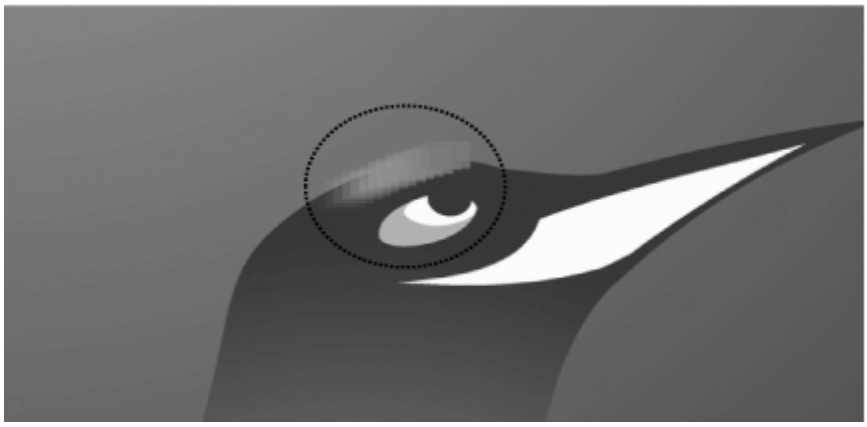
You might wonder if any visual artifacts (*flickers*) are visible between the moment you erase all sprites and the moment you draw them at their new positions. This will not happen, because windowing systems use back buffers, synchronizing them only when the program explicitely asks for a screen update.

A Working Animation

To illustrate everything, here's a short example program animating a colored rectangle and its fading tail (as shown in Figure 2-4). It first creates the needed series of *surfaces*, with a fading color and transparency, then implements the above main loop, animating the sprites along a periodic path. Monitoring events allows the user to temporarily stop the animation by pressing any key and to exit by hitting the escape key or by closing the application window. Read through this example for extensive details about its implementation.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Figure 2-4. An animation with SDL Perl



```
use SDL;
use SDL::App;
use strict;

# specify the target animation speed here, in milliseconds between two
# frames; for 50 frames per second, this is 20 ms
our $TARGET_ANIM_SPEED = 20;

# define an array where to store all the rectangles changed between two
# frames; this allows faster screen updates than using SDL::App#flip
our @update_rects;

# initialize the background surface to an image file if user specified one
# on the commandline, to blank otherwise

our $background = SDL::Surface->new(-f $ARGV[0] ? (-name   => $ARGV[0])
                                                : (-width  => 640,
                                                   -height => 480     ));
# open a 640x480 window for the application
our $app = SDL::App->new(-width => 640, -height => 480);

# copy the whole background surface to the application window
$background->blit(undef, $app, undef);

# update the application window
$app->flip;

# define an array where to store all the surfaces representing
# the colored sprites with all the levels of color and transparency
our @imgs = map
{
    # create a 30x20 surface for one sprite
    my $surface = SDL::Surface->new(
        -width => 30, -height => 20, -depth => 32
    );

    # fill the surface with a solid color; let it fade from
    # blue to white while the mapped int value is iterated over
    $surface->fill(undef,
        SDL::Color->new(-r => 128+$_*255/45, -g => 128+$_*255/45, -b => 255)
    );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
    # set the transparency of the surface (more and more transparent)
    $surface->set_alpha(SDL_SRCALPHA, (15-$_)*255/15);

    # convert the surface to the display format, to allow faster blits
    # to the application window
    $surface->display_format( );

} (1..15);

# define a helper function to blit a surface at a given position on the
# application window, adding the rectangle involved to the array of needed
# updates

sub blit_at
{
    my ($surface, $x, $y) = @_;
    my $dest_rect = SDL::Rect->new(
        -width => $surface->width( ), -height => $surface->height( ),
        -x => $x, '-y' => $y
    );
    $surface->blit(undef, $app, $dest_rect);
    push @update_rects, $dest_rect;
}

# define a helper function to blit the portion of background similar to the
# area of a surface at a given position on the application window, adding
# the rectangle involved to the array of needed updates; this actually
# "erases" the surface previously blitted there
sub erase_at
{
    my ($surface, $x, $y) = @_;
    my $dest_rect = SDL::Rect->new(
        -width => $surface->width( ), -height => $surface->height( ),
        -x => $x, '-y' => $y
    );
    $background->blit($dest_rect, $app, $dest_rect);
    push @update_rects, $dest_rect;
}

# define an array to store the positions of the sprites, a counter to
# calculate new positions of the sprite while it's animated, and a boolean
# to know if the animation has stopped or not
our (@pos, $counter, $stopped);

# define an instance of SDL::Event for event monitoring
our $event = SDL::Event->new( );

# start the main loop here
while (1)
{
    # store the current value of the sdlperl milliseconds counter; the end
    # of the mainloop uses it for animation synchronization
    my $synchro_ticks = $app->ticks( );

    # erase all sprites at their current positions (stored by @pos)
    for (my $i = 0; $i < @pos; $i++)
    {
        erase_at($imgs[$i], $pos[$i]{'x'}, $pos[$i]{'y'});
    }

    # ask for new events
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
    $event->pump( );

    if ($event->poll != 0)
    {
        # if the event is a key press, stop the animation
        if ($event->type( ) = = SDL_KEYDOWN)
        {
            $stopped = 1;
        }

        # if the event is a key release, resume the animation
        if ($event->type( ) = = SDL_KEYUP)
        {
            $stopped = 0;
        }

        # if we receive a "QUIT" event (user clicked the "close" icon of the
        # application window) or the user hit the Escape key, exit program
        if ($event->type = = SDL_QUIT ||
            $event->type = = SDL_KEYDOWN && $event->key_sym = = SDLK_ESCAPE)
        {
            die "quit\\n";
        }
    }

    # if the animation is not stopped, increase the counter
    $stopped or $counter++;

    # insert a new position in top of @pos; let positions be a sine-based
    # smooth curve
    unshift @pos,
    {
        'x' => 320 + 200 * sin($counter/30),
        'y' => 240 +  80 * cos($counter/25),
    };

    # remove the superfluous positions
    @pos > 15 and pop @pos;

    # draw all sprites at their new positions
    for (my $i = @pos - 1; $i >= 0; $i--)
    {
        blit_at($imgs[$i], $pos[$i]{'x'}, $pos[$i]{'y'});
    }

    # tell sdlperl to flush all updates in the specified rectangles
    $app->update(@update_rects);

    # empty the array of rectangles needing an update
    @update_rects = ( );

    # wait the time necessary for this frame to last the target number of
    # milliseconds of a frame.  This allows the animation to look smooth
    my $to_wait = $TARGET_ANIM_SPEED - ($app->ticks - $synchro_ticks);


    $to_wait > 0 and $app->delay($to_wait);
}
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Hack 17. Collect Configuration Information

Save and re-use configuration information.

Some code you write needs configuration information when you build and install it. For example, consider a program that can use any of several optional and conflicting plug-ins. The user must decide what to use when she builds the module, especially if some of the dependencies themselves have dependencies.

When you run your tests and the code in general, having this information available in one spot is very valuable—you can avoid expensive and tricky checks if you hide everything behind a single, consistent interface.

How do you collect and store this information? Ask the user, and then write it into a simple configuration module!

The Hack

Both `Module::Build` and `ExtUtils::MakeMaker` provide user prompting features to ask questions and get answers. The benefit of this is that they silently accept the defaults during automated installations. Users at the keyboard can still answer a prompt, while users who just want the software to install won't launch the installer, turn away, and return an hour later to find that another prompt has halted the process in the meantime.

`Module::Build` is easier to extend, so here's a simple subclass that allows you to specify questions, default values, and configuration keys before writing out a standard module containing this information:

```
package Module::Build::Configurator;

use strict;
use warnings;

use base 'Module::Build';

use SUPER;
use File::Path;
use Data::Dumper;
use File::Spec::Functions;

sub new
{
    my ($class, %args) = @_;
    my $self           = super( );
    my $config         = $self->notes( 'config_data' ) || { };

    for my $question ( @{ $args{config_questions} } )
    {
        my ($q, $name, $default) = map { defined $_ ? $_ : '' } @$question;
        $config->{$name}         = $self->prompt( $q, $default );
    }

    $self->notes( 'config_module', $args{config_module} );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
    $self->notes( 'config_data',   $config );
    return $self;
}

sub ACTION_build
{
    $_[0]->write_config( );
    super( );
}

sub write_config
{
    my $self      = shift;
    my $file      = $self->notes( 'config_module' );
    my $data      = $self->notes( 'config_data' );
    my $dump      = Data::Dumper->new( [ $data ], [ 'config_data' ] )->Dump;
    my $file_path = catfile( 'lib', split( /::/, $file . '.pm' ) );

    my $path      = ( splitpath( $file_path ) )[1];
    mkpath( $path ) unless -d $path;

    my $package   = <<END_MODULE;
    package $file;

    my $dump

    sub get_value
    {
        my (\\$class, \\$key) = \\@_;

        return unless exists \\$config_data->{ \\$key };
        return               \\$config_data->{ \\$key };
    }

    1;
END_MODULE

    $package =~ s/^\\t//gm;

    open( my $fh, '>', $file_path )
        or die "Cannot write config file '$path': $!\\n";
    print $fh $package;
    close $fh;
}

1;
```

The module itself is a straightforward subclass of `Module::Build`. It overrides `new( )` to collect the `config_module` argument (containing the name of the configuration module to write) and to loop over every configuration question (specified with `config_questions`). The latter argument is an array reference of array references containing the name of the question to ask, the name of the value as stored in the configuration module, and the default value of the question, if any. In an unattended installation, the `prompt ( )` method will return the default value rather than interrupt the process.

The module also overrides `ACTION_build( )`, the method that `perl ./Build` runs, to write the configuration file. The `write_config ( )` method takes the hash of configuration data created in `new( )` (and stored with `Module::Build`'s handy `notes( )` mechanism), serializes it with `Data::Dumper`, and writes it and a module skeleton to the necessary path under *lib/*.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Note the use of `File::Spec::Functions` and `File::Path` to improve file handling and to make sure that the destination directory exists for the configuration module.

Using the Hack

To use the hack, write a *Build.PL* file as normal:

```
use Module::Build::Configurator;

my $build = Module::Build::Configurator->new(
    module_name      => 'User::IrisScan',
    config_module    => 'User::IrisScan::Config',
    config_questions =>
    [
        [ 'What is your name?',                      'name', 'Anouska' ],
        [ 'Rate yourself as a spy from 1 to 10.',    'rating', '10'      ],
        [ 'What is your eye color?',             'eye_color', 'blue'    ],
    ],
);

$build->create_build_script( );
```

This file builds a distribution for the `User::IrisScan` module. Run it to see the prompts:

```
$ perl Build.PL
What is your name? [Anouska] Faye
Rate yourself as a spy from 1 to 10. [10] 8
What is your eye color? [blue] blue
Deleting Build
Removed previous script 'Build'
Creating new 'Build' script for 'User-IrisScan' version '1.28'
$
```

Now look at *lib/User/IrisScan/Config.pm*:

```
package User::IrisScan::Config;

my $config_data = {
             'eye_color' => 'blue',
             'name' => 'Faye',
             'rating' => '8'
           };


sub get_value
{
    my ($class, $key) = @_;

    return unless exists $config_data->{ $key };
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
    return                $config_data->{ $key };
}

1;
```

You can use this configuration module within your tests or within the program now as normal. If you're clever, you can even check for this module when *upgrading* your software. If it exists, use the configured values there for the defaults. Your users will love you.

Hack 18. Rewrite the Web

Use the power of Perl to rewrite the web.

The Greasemonkey extension for Mozilla Firefox and related browsers is a powerful way to modify web pages to your liking. In fact, the Mozilla family projects are customizable in many ways—as long as you like writing C++, JavaScript, or XUL.

If your network doesn't run only Firefox, or if you just prefer to customize the Web with Perl instead of any other language, `HTTP::Proxy` can help.

The Hack

For whatever reason (registrar greed, mostly), plenty of useful sites such as Perl Monks have *.com* and *.org* domain names. One visitor might use http://www.perlmonks.com/, while the truly blessed saints prefer http://perlmonks.org/. That's all well and good except for the cases where you have logged in to the site through one domain name but not the others. Your HTTP cookie uses the specific domain name for identification.

Thus you may follow a link from somewhere that leads to the correct site with the incorrect domain name. How annoying!

Fixing this with `HTTP::Proxy` is easy though:

```
use strict;
use warnings;

use HTTP::Proxy ':log';
use HTTP::Proxy::HeaderFilter::simple;

# start the proxy with the given command-line parameters
my $proxy = HTTP::Proxy->new( @ARGV );

for my $redirect (<DATA>)
{
    chomp $redirect;

    my ($pattern, $destination) = split( /\|/, $redirect );
    my $filter                  = get_filter( $destination );

    $proxy->push_filter( host => $pattern, request => $filter );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

```
}

$proxy->start( );

my %filters;

sub get_filter
{
    my $site = shift;

    return $filters{ $site } ||= HTTP::Proxy::HeaderFilter::simple->new(
        sub
        {
            my ( $self, $headers, $message ) = @_;

            # modify the host part of the request only
            $message->uri( )->host( $site );

            # create a new redirect response
            my $res = HTTP::Response->new(
                301,
                "Moved to $site",
                [ Location => $message->uri( ) ]
            );

            # and make the proxy send it back to the client
            $self->proxy( )->response( $res );
        }
    );
}

__DATA__
perlmonks.com|perlmonks.org
www.perlmonks.org|perlmonks.org
```

The program creates a new `HTTP::Proxy` object, then reads all of the data at the end of the program to create header filters. When a request comes in, the proxy runs all header filters that match the request. These filters can manipulate the request as appropriate.

In this example, if the host of a request matches `perlmonks.com`, the filter sends back an HTTP 301 status code redirecting the request to `perlmonks.org`. A well-behaved client will repeat the request with the new host (this time, sending along the proper cookie).

The use of the `%filters` lexical is the Orcish Maneuver. Read the line in `get_filter( )` as "return the cached object or cache a new one".

Running the Hack

Run the program from the command line. If necessary, pass in arguments, perhaps to run on a different port:

```
$ perl memoryproxy.pl port 5000
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006

Configure your browser to use this proxy and try to visit http://perlmonks.com/). You'll end up at http://perlmonks.org/.

Hacking the Hack

There are countless uses for `HTTP::Proxy`, even beyond rewriting both request and response headers and bodies. Try:

- Restricting a browsing session to no more than ten minutes at a time during working hours.
- Maintaining a list (or graph or tree) of relationships between sites.
- Forbidding yourself from wasting time reading certain sites during working hours.
- Creating shortcuts for URLs [Hack #1] across multiple browsers without manipulating local DNS records.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe
ISBN: 0596526741 Publisher: O'Reilly Print Publication Date: 5/1/2006