# A Simple Shopping Cart using CGI

*Professor Don Colton, BYU Hawaii*

March 5, 2004

In this section of the course, we learn to use CGI and Perl to create a simple web-based shopping cart program. We assume you have written some small CGI programs already, and are familiar with directories, permissions, and running from the command line.

## 1 Program Overview

Business Purpose: We will create a simple web-based business to sell a limited variety of products.

Shopping Experience: We desire the customer to first see a Welcome screen when they visit our web site. From the Welcome screen, the customer will enter the store. While in the store, the customer will select items to add to his shopping cart. Occasionally the customer will check his cart to see what is in it. Finally, the customer will check out and pay for his purchase.

### 1.1 Screens

We visualize the web site as consisting of the following screens.

**Welcome:** This screen will introduce the business in a friendly manner and provide access to the shopping screen.

**Shop:** This screen will list the entire product catalog, and allow items to be added to the cart.

**Cart:** This screen will list the contents of the shopping cart, and will allow the customer to remove items from the cart.

**Checkout:** This screen will list the contents of the shopping cart, and give a total price. It will allow the customer to enter a credit card number.

### 1.2 Subroutines

We visualize the program as consisting of the following subroutines.

**Main:** Not really a subroutine, reads the CGI input and decides which other subroutines should be called.

**Welcome:** Paints the Welcome screen.

**Shop:** Paints the Shop screen. Handles the adding of items into the shopping cart.

**Cart:** Paints the Cart screen. Handles the removal of items from the shopping cart.

**Checkout:** Paints the Checkout screen. Calculates the total price. Handles the entry of the credit card number.

**Init:** Initializes the inventory database.

# 2 Step One

Our first step is to construct a highly simplified version of the final program. This will be the skeleton to which we will later add functionality.

```
#!/usr/bin/perl -Tw
chomp ( $in = <STDIN> );
init();
welcome(); exit;

sub welcome {
  print "Content-type: text/html\n\n
debug: $in=($in)
<h1>Welcome</h1>
<form method=post action=''>
<input type=submit name=go value=Shop>
<input type=submit name=go value=Cart>
<input type=submit name=go value=Checkout>
</form>
"; }
sub shop {}
sub cart {}
sub checkout {}
sub init {}
```

The first line, (`#!/usr/bin/perl -Tw`) tells the computer that this is a perl program, and that (T) taint checking should be turned on, and that (w) warning messages should be given in case perl can tell our program is not correct.

`chomp ( $in = <STDIN> );` reads in the one line of input that our program will receive. It places it into the variable `$in` and removes the trailing newline character.

`init();` will be used later to provide inventory for sale. You can leave it out until you develop the init subroutine later, but there is no harm putting it in now so you don't forget.

`welcome(); exit;` calls your subroutine to paint the welcome screen. This would be the case the first time the program is run. When the welcome subroutine completes its work, we `exit` directly to stop the program.

`sub welcome { ... }` defines `welcome` to be a subroutine consisting of all the activities between the curly braces. More on that soon.

`sub shop {}` defines `shop` to be a subroutine with no content. Such a subroutine is called a **stub**. Later we will add content to it. For now it is just a blank chapter in the book. The same is true for `cart`, `checkout`, and `init`.

## 2.1 Welcome code

The `welcome` subroutine consists of exactly one statement. It is a `print` statement that creates an entire web page.

`Content-type: text/html\n\n` is printed first. This tells the browser that the remaining lines will be html code. The two newlines are used to create a blank line between the headers (content type and other such lines) and the web page.

`debug: $in=($in)` gives us some visibility into what is happening. When we run our program on the web, it shows us the input that our program was working with. This will prove to be very handy during debugging, but we will remove it before the program goes into actual use.

`<h1>Welcome</h1>` provides the cheerful introduction to our business. In real life we would be a bit more verbose.

`<form method=post action=''>` specifies the start of the form. `post` causes our inputs to not be displayed on the URL line of the web browser. Action appears to be blank. It is in fact a relative reference telling what we should do if this form is used. What we do is nothing. Nothing other than run this same program again. If we wanted to run a different program we would specify it here.

`<input type=submit name=go value=Shop>` creates a `[Shop]` button and provides that `go=Shop` will be sent to our program if the customer presses it. Similarly the links to Cart and Checkout are specified.

`</form>` designates the end of the form.

Finally, the print statement ends with `";` and the subroutine ends with `}`.

# 3 Command-Line Testing

We have written enough that we can begin testing. Our programming method is called **top-down programming** using **stepwise refinement**. By top down, we mean that we are looking at the big picture first, the top view as seen by an airplane flying over the countryside. We identify the major features (the main program and the main subroutines). We do not specify too much detail at first to reduce complexity and confusion.

Type `emacs ~/public_cgi/mystore` to start the text editor. Key in the program. Save the program. Rather than fully exit (C-x C-c) just shell out (C-z). This will allow you to resume your edit (`fg`) after each mistake is announced.

Type `chmod 711 ~/public_cgi/mystore` to set the proper permissions. This allows your program to be executed by the web server.

Type `~/public_cgi/mystore` to run the program. You should be presented with a blank line while your program waits for input. Press enter. You should receive output like this:

```
Content-type: text/html


debug: $in=()
<h1>Welcome</h1>
<form method=post action=''>
<input type=submit name=go value=Shop>
<input type=submit name=go value=Cart>
<input type=submit name=go value=Checkout>
</form>
```

These are the lines that were specified in the `welcome` subroutine. If you receive anything unexpected, fix it. The most common problems are these.

Unterminated string. This means that you have opening quote marks someplace but the closing quote marks cannot be found. Find the line where this was reported. The problem is on that line or above.

(more to be added)

# 4 Web Testing

It does ver little good to perform web testing until you know that your program works correctly from the command line. The error messages on the web are not very helpful. The error messages on the command line are much more helpful.

Start your browser and type in the URL for your program. It should be something like this.

```
http://cgi.cs.byuh.edu/~aa999/mystore
```

Of course, replace `aa999` with your login name. Replace `mystore` with the name of your program if you used a different name.

You should see your first screen. It should consist of the following words.

```
Welcome
[Shop] [Cart] [Checkout]
```

View the page source. It should consist of the following words.

```
debug: $in=()
<h1>Welcome</h1>
<form method=post action=''>
<input type=submit name=go value=Shop>
<input type=submit name=go value=Cart>
<input type=submit name=go value=Checkout>
</form>
```

These are the same words you included in your `welcome` subroutine shown above.

If you press on any of the buttons, `[Shop]` for instance, your program will run, but the browser will think that it crashed because your program did not create any output.

# 5 Step Two

We are ready to flesh out the stubs a tiny bit. In this step, we will create working (but almost empty) web pages for `shop`, `cart`, and `checkout`. Change the code for each. Before:

```
sub cart {}
```

After:

```
sub cart {
print "Content-type: text/html\n\n
debug: in=($in)
<h1>Shopping Cart</h1>" }
```

We have not done much. All we have done is print a viable web page in each case, and we have included the debug information we want from the `$in` variable.

```
if ( $in =~ /go=Shop/ ) { shop(); exit }
if ( $in =~ /go=Cart/ ) { cart(); exit }
if ( $in =~ /go=Checkout/ ) {
  checkout(); exit }
welcome(); exit;
```

The last line should already be in your program. The other three activate the subroutines that you just wrote, but only when the right words appear in the input string.

Test your revised program. Using your browser, press each of the buttons on the welcome screen. Use the back button on your browser to return to the welcome screen.

# 6 Step Three: Inventory

Next we will create some inventory to be sold. We will use a business model for selling fishing supplies. Our company will be called Fish Bait, Inc. Replace the `init` subroutine with the following:

```
# populate the product catalog
sub init {
  push @inv, "worms, 100=1.00";
  push @inv, "flies, 50=2.00";
  push @inv, "squid, small=1.50";
  push @inv, "chum, bucket=5.00";
  push @inv, "minnow=0.25";
  push @inv, "hooks, large=2.75";
}
```

In this subroutine, we are adding items to an array named `@inv`. Each item is specified by "name=price". For your own project, you should think of a different theme (not fish bait) and a list of appropriate products. You can do something general like Ebay or a garage sale, or you can do something focused like cosmetics or car parts. Make sure you have a call to init in your main program.

We are now ready to write the `shop()` subroutine. Change it to say the following.

```
# paint the products screen
sub shop {
print "Content-type: text/html\n\n
debug: in=($in)<br>
<h1>Fish Bait Products</h1>
<form method=post action=''>
<table><tr><td>Item<td>Price<td>Qty
<td>Buy It\n";
for ( $i = 0; $i < @inv; $i++ ) {
  ( $item, $price ) = split /=/, $inv[$i];
  print "<tr><td>$item<td>$price";
  print "<td><input type=text";
  print " name=qty$i value=1>\n";
  print "<td><input type=submit";
  print " name=add$i value='add'>\n";
}
print "</table></form>\n"; }
```

The for loop will walk through the inventory and produce one line of output (three print statements) for each product in the inventory.

Test your program. See whether it creates a nice table of inventory. If you are brave, press one of the `[add]` buttons to see what happens. You should end up back at the welcome screen, but with an interesting input line to ponder.

# 7 Step Four: The Cart

We can recognize that an `[add]` button has been pressed as follows. This line should be inserted in the main program, somewhere after init and before welcome.

```
if ( $in =~ /add\d+=add/ ) { shop(); exit }
```

Once we get to the `shop()` subroutine, we must identify the item and its quantity.

```
  if ( $in =~ /add(\d+)=add/ ) {
    $new = $1;
    $in =~ /qty$new=(\d+)/;
    $qty = $1;
    $cart .= " $new.$qty";
  }
```

We are finding the product number (`$new`) and its quantity (`$qty`). We are placing them into the shopping cart (`$cart`) that we have newly created.

So now we will need a way to carry the shopping cart from screen to screen. We will do this with a hidden field:

```
<input type=hidden name=cart value='$cart'>
```

We will put that line right after the `<form>` line. We will also need to capture the cart information before we can add to it.

```
if ( $in =~ /cart=([^&]*)/ ) {
  $cart = $1 } else { $cart = "" }
$cart =~ s/[+]/ /g;
$cart =~ s/%(..)/pack('C',hex($1))/eg;
```

To help us see the effects of this new shopping cart, we will add another debug line to the shop subroutine.

```
<br>debug: cart=($cart)
```

This line goes right after the `debug: in` line.

If we did everything right (or after we fix everything that we broke trying to do everything right), our program should behave as follows. Each time an `[add]` button is pressed, the shopping cart should have more items inside it.