An oversimplified introduction to HTML and CGI

October 3, 2015

Contents

Chapter 1

HTML for document formatting

Chapter 2

CGI for building dynamic web pages

2.1 The need for CGI

We often need customized web pages. For instance, whenever we use a search engine, the page that we see is created specific to the query we submitted - it is just not possible that there is already present a static page specific to our query - all possible queries cannot be anticipated, and even if that were possible, how could we create the huge number of pages and where could we store them? Thus, the page has to be created by a programme, a CGI (Common Gateway Interface - CHECK!) programme. A search engine is a typical example of a CGI programme interfacing with a data-base.

Another very important application of CGI programmes, of particular interest to us, is for controlling experimental set-ups from remote locations. Thus, from the comfort of your home, it is possible to check experimental parameters and modify settings if necessary. If you have a collaborator in Australia, she can control the experiments running in you lab, and vice versa.

If you develop a software for some computation work, CGI programmes make it possible to share it over the Internet. The user computer may run any operating system and need not have the libraries and utilities necessary for running the programme. All that is needed is a web browser. If the CGI programme is not loaded with too many complex constructs for visual effects, any browser can access it successfully. It is also possible to have the CGI programme determine the kind of browser the user has, and tailor the page

to take advantage of the specific capabilities. All this is possible because the CGI programme runs completely on the server. The client computer needs only to run the browser.

2.2 Packages needed on the server side

2.2.1 For the CGI programme

A CGI programme is a standard executable programme written in any language, even shell-script if necessary. Thus, depending on our coding preferences, we may need compilers like gcc (C), g++ (C++), gfortran (FORTRAN), ghc (haskell) or interpreters like Ruby, Python, Tcl/TK etc. A common practice is to use Perl, because many commercial service providers do not allow loading of compiled binaries in their servers. Again, interpreters are notoriously slow. Perl treads the proverbial middle path - saves the source code, compiles it once during execution and uses the compiled executable through all the loops inside the programme (unlike true interpreters which compile instruction by instruction as many times as required) and then deletes the executable.

The other necessary package is an http *daemon*.

2.2.1.1 Daemon - help!

A daemon is not a demon! In Greek mythology a daemon is a minor God who controls only a particular aspect of the lives of humans, much like our *Sitala, Manasha, Ola-bibi,* etc! In computer parlance a daemon programme runs in user-space[1] and looks after a particular functionality[2]. Client programmes connect with the daemons through IPC (Inter Process Communication) and send requests for various services.

The daemon that concerns us here is *httpd*, the hypertext transfer protocol daemon. The packages for this are *apache2*, *lighthttpd*, and many other

[1] A *process* is a programme which has been loaded into memory and is either running or waiting - this region of memory is called the user-space. In contrast the operating system has its own sphere, the kernel-space. A daemon programme is just an ordinary programme owned by root and started either during the boot sequence (remember all those 'starting' messages?) or due to some external event - USB connection, etc.

[2] e.g. lpd or cupsd for handling printers, crond for handling timed events, sshd for remote incoming connections, etc.

similar packages. You have to install one of these and enable CGI execution in some cases (recent versions of Ubuntu do not have CGI execution enabled by default - instructions are available on the Internet).

An HTML file created in your user area can be opened directly from the browser's menu. However, if you want other computers to have access to your web pages, or even if you want to access your own pages using the url *http://localhost* on your browser, you need the daemon. In the case of CGI programmes, the daemon is a must, because the daemon runs the programme on the host computer and sends the output to the client browser.

Not all directories of the server are accessible to the outside, even via the httpd daemon. That would be a huge security risk. There are certain specified directories (specified through entries in configuration files for httpd, usually found in */etc*) which are accessible to the daemon and the files have to be placed there. In Debian, the html files have to be placed in */var/www* and its subdirectories and the CGI programmes in */usr/lib/cgi-bin*. [**Note:** Both these locations are owned by root and cannot be written to by non-root users. To enable our students to make suitable entries, subdirectories named *chanapona* have been created with read-write-execute permissions under these directories.]

2.3 Headers

When you open a file from the 'file' menu of a browser, the browser knows, from the extension to the file name, what kind to content to expect. However, when the content comes from a httpd server on a remote machine, it is just a stream of 0s and 1s - you do not even know whether the contents should be treated as a binary data stream or the contents of an ascii file. This difficulty is handled by the server (httpd) which sends a particular header which uniquely specifies the type of the data that it is sending. Thus, for instance, when an HTML file is being sent, the data stream is furnished with the header *Content-type: text/html* with a blank line following it. On receiving it, the browser knows that the rest of the data has to be be handled as an ascii file with HTML content and renders it accordingly. How does the daemon know what type of data it is sending? From the file name extension! Thus when it sends a file with an extension of *html* or *htm* it automatically adds the header shown above and then sends the contents of the file bit by bit (or byte by byte - if you like). In the case of a CGI programme, the

extension is just *cgi* and the daemon runs it and sends its output to the requesting browser, but how does it know of what kind is the data that the programme produces. If it is of type HTML, then httpd would have to add *Content-type: text/html (blank line)* as a header, for a *jpeg* image the header would be *Content-type: image/jpeg (blank line)* but unlike in the case of static files with different extensions for different types, all these programmes have the same extension *cgi*. So httpd does not sent a header at all! *The header has to be supplied by the CGI programme itself.* This is the most important thing to keep in mind while writing a CGI proramme - the syntax of the header has to be exact - *Content-type* and not *content-type*, a single blank after the colon and the mandatory blank line. Most mistakes occur here.

2.4 Our first CGI programme

A CGI programme is just a series of writes to standard output which are captured by the http daemon and sent to the browser which renders them according to the *Content-type* specification in the header. Since it is a programme, it can do other things also - maybe turn on a light or change the temperature settings of a furnace - but on the side of the server only.

2.4.1 The code

```
#include <iostream>

using namespace std;

int main(void)
{
cout << "Content-type:  text/html\n" << endl; // The header
cout << "<html><body bgcolor=black text=orange>" << endl;
// The endl above is optional, but use it for good readability
// in case you want to run the programme directly from the
// terminal for debugging.
cout << "You are a <b> bhoot!  </b>" << endl;
cout << "</body></html>" << endl;
```

```
return(0);
}
```

2.4.2 Where to put our code

Compile the code above and put a *cgi* extension to the executable. You can do this by either compiling first and then renaming the executable (the *mv* command) or by running something like (assuming that the file is named *who_am_I.C*) - *g++ -O -Wall who_am_I.C -o who_am_I.cgi*.

Now copy the executable to the target directory (somewhere under /usr/lib/cgi-bin in Debian, in our case /usr/lib/cgi-bin/chanapona). That's all!

2.4.3 Running the CGI programme

Launch your browser (there is a very fast and beautiful text-mode browser called *lynx*). Type in the url - *http://localhost/cgi-bin/chanapona/who_am_I.cgi*. The browser will readily furnish you with some self-realisation!

The *directory* path in the url is very interesting. You see, the root directory for CGI programmes is already */usr/lib/cgi-bin*. Then why do we type in the *cgi-bin* explicitly in the url? This should have been the case if there was an extra directory called *cgi-bin* under */usr/lib/cgi-bin*? This is how httpd ensures that the client browser does not have access to the CGI file at all. There is no CGI programme where the url points! However the daemon (which is playing on *our* side!) knows that the *cgi-bin* in the url is not a path but actually a signal that this time it needs to run a programme. It runs the programme from the directory apparently above what the url states, and simply sends the output of the programme. There is no way for the client to get at the code (Remember that *perl* and interpreted programmes do not differentiate between source code and executable, so what would happen if a hacker could get the raw code and find out exactly how a programme in a bank works - with database passwords and all?).

2.4.4 Too bland - something more interesting?

Our first CGI programme simply wastes some CPU power - it could have been implemented using a static HTML page. So what is the point? Try the following code:

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <cstring>

using namespace std;

int main(void)
{
srand(time(NULL)); // seed the random number generator

cout << "Content-type:  text/html\n" << endl;
cout << "<html><body bgcolor=black text=PaleGoldenRod>" << endl;

char bhoot_type[8];

if(rand() <= RAND_MAX/2)
{ strcpy(bhoot_type," pocha "); }
else
{ strcpy(bhoot_type," bomba "); }

cout << "You are a <b>" << bhoot_type << "bhoot</b>" << endl;
cout << "</body></html>" << endl;
return(0);
}
```

Better now? Try to use random numbers for the background and text colours also?

2.5 The other way round - sending parameters to a CGI programme

2.5.1 Environmental variables

What happens if you type into a terminal: *x=bhoot*? You will see that no error is generated - the shell accepts the input happily. Actually an *environmental variable* called x is created. These variables stay in the environment

and can hold only strings, of immense length if necessary[3]. The interesting thing is that environmental variables can be accessed by other programmes. In C and C++, the *getenv* function is used to get the value of an environmental variable. Try the following code:

```
#include <iostream>
#include <stdlib>

using namespace std;

int main(void)
{
cout << "The variable sankhchunni contains:  " << getenv("sankhchunni")
<< endl;

return(0);
}
```

Compile this programme (say, to en.x) and run it using the command - *sankhchunni=petni ./ en.x*. Since the executable is run in a temporary shell of its own *sankhchunni* needs to be initialised inside the command itself. If you want to initialise *sankhchunni* and run the programme separately, you would have to export the variable first using *export sankhchunni=petni*. You could then run the executable subsequently.

How is this relevant to our current topic? If you type a question mark after a 'normal' url and type a string after it (do not leave blanks - though some browsers can handle blanks), the http daemon stores that string into an environmental variable called *QUERY_STRING* (the name is fixed) and then runs the CGI programme. The CGI programme can now use *getenv* to access this variable and its contents. This is one method of passing variables from the browser end to a CGI programme. Consider the following CGI source code:

```
#include <iostream>
#include <cstdlib>
```

[3]You can use *echo* to read environmental variables. If you type *echo x* then you will get *x*, but if you type *echo $x* then the value of the environmental variable will be shown. Try it.

```
using namespace std;

int main(void)
{
cout << "Content-type:  text/html\n" << endl;
cout << "<html><body><h2>You have typed:  ";
cout << getenv("QUERY_STRING");
cout << "</h2></body></html>";

return(0);
}
```

Run this code with the following url - http://localhost/cgi-bin/<chanapona>/<name.cgi>?<somet
and check what you get.

2.5.2 HTML forms

The method discussed above, of coding input into the url directly, is called 'get'. Instead of forming your own QUERY_STRING, *HTML forms* allow you to use a friendly interface and have the browser form the QUERY_STRING for you. HTML forms are defined between *<form>* and *</form>* tags. In between, any normal HTML code can be written. In addition, special tags may be used to create fields which create widgets which accept inputs. Consider the following html page (we assume that the CGI programme discussed in the previous subsection is named *boka.cgi*):

```
<html>
<body>
<h1>To see what QUERY_STRING looks like</h1>
<form action=/cgi-bin/chanapona/boka.cgi method=get>
Type something:
<input type=text length=50 name=mamdo>
<input type=submit name=show value=show>
</body>
</html>
```

Save this page in */var/www/chanapona* and open it using the url *http://localhost/chanapona/<nam*

You will find that a box for typing input will be created followed by a button which you can click to submit the form. The *action* will be executed when you press any button of type *submit*. Since the *method* is *get*, all the inputs will be collected together, encoded into a string without blanks, and automatically appended to the url following a question mark (check the url which appears in the browser on pressing the *show* button which is of type *submit*). This will subsequently be converted into QUERY_STRING and placed in the environment by httpd running at the server end and then *boka.cgi* will be executed.

Notice how QUERY_STRING is formed. There are two inputs named *mamdo* and *show*. An ampersand (&) is used as a spacer between the two fields to form *mamdo=<your input>&show=show*. The submit button has a fixed value, as declared by the *value=* directive. The input of type *text* will assume the value that you type in, but you can also put in a *value=* directive, in which case that value will be pre-loaded into the text field to be overwritten or modified by the user.

Try typing different things into the input. Notice the following:

- A blank in the input gets converted into a '+'.
- If you type in things like a real '+' or an '&', the browser, since it uses these characters to signify a space or a spacer, has to resort to a different stratagem. It puts in a '%' (another reserved character!) and then places two characters which form the ascii representation of the desired character in hexadecimal.

Thus, if you want your CGI code to handle multiple and complex inputs (many text boxes, blanks in inputs etc.), you will have to separate the different fields by looking at the '&'s, replace the '+' characters with blanks and then convert all the '%ab' to the correct characters. This makes the programme complex.

2.5.3 *uncgi* to the rescue

There is a free programme called *uncgi* which can be used both as a stand-alone programme and as a library to handle these kinds of inputs. We will use the stand-alone version.

2.5.3.1 Where *uncgi* resides

uncgi may be installed in different places, but we prefer to put it inside the home *cgi-bin* directory where it just acts like another CGI programme.

2.5.3.2 How *uncgi* is incorporated into the HTML form

Just put in *uncgi* into the *action=* directive just after *cgi-bin*. Thus, if the original directive was *action=/cgi-bin/chanapona/halum.cgi*, replace it with *action=/cgi-bin/uncgi/chanapona/halum.cgi* as if *uncgi* were an extra directory. Note, however, that this is not the case and the CGI programmes remain where they were previously.

2.5.3.3 What *uncgi* does

The http daemon parses the *action=* directive and descends the directory tree below the home CGI directory till it hits *uncgi*. There is no directory with this name but an executable is there, so it is run. *uncgi* now studies QUERY_STRING and creates *a series of environmental variables*. If an input field is given a name *bhoot* with the *name=* directive, the corresponding variable created is named *WWW_bhoot* (notice the capital letters - the commonest source of mistakes!). The value of the field is now placed inside this variable, with all the necessary conversions aleady made. After the whole of QUERY_STRING is processed in this way, *uncgi* now, in its turn, runs the actual CGI programme! So your CGI programme only has to read in the necessary environmental variables.

2.6 Learning with examples

The possibilities are limitless. We have limited time! Therefore we try to get a feel of the system through some examples. Remember that the HTML code resides (in our case) in */var/www/chanapona* and the CGI code in */usr/lib/cgi-bin/chanapona*.

2.6.1 A small calculator

This is a small calculator which can only add, subtract, multiply and divide! Notice that at the user end, only a browser is needed - the calculation is all

done at the server end.

2.6.1.1 HTML

onko.html

```
<html>
<body bgcolor=black text=PaleGoldenRod>
<form action=/cgi-bin/uncgi/chanapona/onko.cgi method=get
target=uttor>
<input type=text name=prothom>
<input type=submit name=bhombol value=+>
<input type=submit name=bhombol value=->
<input type=submit name=bhombol value=*>
<input type=submit name=bhombol value=/>
<input type=text name=dwitiya>
=
<iframe height=100 width=200 name=uttor>
</form>
</body>
</html>
```

2.6.1.2 CGI

onko.C - source for onko.cgi

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void)
{
float a,b,c;
char action

a = atof(getenv(“WWW_prothom”)); // get the input values
b = atof(getenv(“WWW_dwitiya”));
```

```
action = getenv(“WWW_bhombol”)[0]; // get the desired action

cout << “Content-type:  text/html\n” << endl; // Header
cout << “<html><body bgcolor=PalegGoldenRod text=red>

if(action == ’+’)
{ c = a+b; }
else if(action == ’-’)
{ c = a-b; }
else if(action == ’*’)
{ c = a*b; }
else // We know that we need to divide ...  but division by zero?
...
{
if(b == 0)
{
cout << “Division by 0!  @#%!”;
cout << “</body></html>”;
return(0);
}
c = a/b; // We come here only if everything is o.k.
}

cout << c; // the result
cout << “</body></html>”;
return(0);
}
```

2.6.1.3 Discussions

With *atof* we convert the ascii string returned by *getenv* into a *float*. Note that we have four *submit* buttons, each named *bhombol*. Any one will work because they are all of type *submit* but the value associated with *bhombol* will be different in each case (determined by the *value=* directive). This value is a string (C string, i.e. a character array with a NULL terminator) and thus if we want to compare the first character (which contains the action in this case) we need to isolate it with *getenv(“WWW_bhombol”)[0]* which

we can compare with another character using ==.

The *iframe* is a very useful widget. It allows us to include another page inside one web page. We use it here to display the answer. Note the use of the *name* directive inside the *iframe* and *target=* inside the *form* to allow the result of an action initiated in one frame (the main page in this case) to be displayed inside another one.

2.6.2 Inputting a matrix

The first form will allow us to set the number of rows and columns. Then the *show_form.cgi* programme generates *another form* which allows us to input the elements of the matrix. These elements are accessed by the second CGI programme - *matrix.cgi* which simply prints out the elements. You may try to do something more useful with the data - why not try to read-in two matrices and multiply them?

2.6.2.1 HTML

matrix.html

```
<html>
<body bgcolor=white text=DarkOliveGreen>
<form action=/cgi-bin/uncgi/chanapona/show_form.cgi method=get>
<h1>Welcome to our matrix entry page!</h1>
<b><i>No.  of rows:  <b></i><input type=text
name=num_rows>  
<b><i>No.  of cols:  <b><i><input type=text name=num_cols>
<input type=submit name=halum value=create>
</form>
</body>
</html>
```

2.6.2.2 CGI

show_form.C - source for show_form.cgi

```
#include <iostream>
#include <cstdlib>
```

```
#include <cstdio>

using namespace std;

int main(void)
{
int num_rows,num_cols;
// Get the dimensions of the matrix
num_rows = atoi(getenv("WWW_num_rows"));
num_cols = atoi(getenv("WWW_num_cols"));
cout << "Content-type:  text/html\n" << endl;
cout << "<html><body bgcolor=black text=cyan>" << endl;
int i,j;
char puchkay[20];
// Create the form and store num_rows and num_cols as hidden variables
//for the
// Next CGI programme
cout << "<h2>Type in the elements below:</h2>\n";
cout << "<form action=/cgi-bin/uncgi/chanapona/matrix.cgi method=get>"
<< endl;
cout << "<input type=hidden name=num_rows value="<<num_rows<<">"<<endl;
cout << "<input type=hidden name=num_cols value="<<num_cols<<">"<<endl;
// Create the table, create names for the inputs
cout << "<table noborder width=30%>" << endl;
for(i=0;i<num_rows;++i)
{
cout << "<tr>\n";
for(j=0;j<num_cols;++j)
{
sprintf(puchkay,"ele_%d_%d",i,j);
cout << "<td><input type=text size=7 name="<<puchkay<<"></td>\n";
}
cout << "</tr>\n";
}
cout << "<tr><td align=center colspan="<<num_cols<<">";
cout << "<input type=submit name=bhombol value=\"Show Matrix\"></td></tr>\n";
cout << "</table></form></body></html>\n";
return(0);
```

```
}
```

matrix.C - source code for matrix.cgi

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
using namespace std;
int main(void)
{
int num_rows,num_cols;
num_rows = atoi(getenv("WWW_num_rows"));
num_cols = atoi(getenv("WWW_num_cols"));
int i,j;
float element[num_rows][num_cols];
char puchkay[30];
for(i=0;i<num_rows;++i)
{
for(j=0;j<num_cols;++j)
{
sprintf(puchkay,"WWW_ele_%d_%d",i,j);
element[i][j] = atof(getenv(puchkay));
}
}
cout << "Content-type:  text/html\n" << endl;
cout << "<html><body bgcolor=PaleGoldenRod text=brown>" << endl;
cout << "<h1>The matrix elements are:</h1>";
for(i=0;i<num_rows;++i)
{
for(j=0;j<num_cols;++j)
{
cout << "element["<<i<<"]["<<j<<"] = "<< element[i][j] << "<p>";
}
}
cout <<"</body></html>";
return(0);
}
```

2.6.3 Discussions

This is a two-stage programme. The points to note are:

- The first HTML form reads in the number of rows and columns. *atoi* and *getenv* are used to get these values from the first CGI programme.
- The first CGI programme, *show_form.cgi*, does quite a few things:
  - It creates a new form which will be processed by the next CGI programme, *matrix.cgi*.
  - It uses two hidden inputs to store the number of rows and columns inside the new form. Hidden inputs are not really 'inputs'; they are not shown and their values are fixed by the *value=* directive. They can be accessed just like any other input field, however and in this way they can be used by the previous form to send values through the next form to the next CGI programme. We do this because the next CGI programme, *matrix.cgi*, needs to know the number of rows and columns.
  - A table is used to display the input widgets for the matrix elements in a nice way.
  - The *input* widgets need to have unique names (using *name=*) so that their contents can be accessed by the next CGI programme. We have used a C function, *sprintf*, to create names inside a small array and used the content of that array in the *name=* directive. Check this out - it is very interesting and useful! When we print something into a character array *puchkay* using *sprintf*, the ascii characters get loaded into the array one by one. Thus for the element corresponding to row 3 and column 5, the pattern *ele_3_5* gets written into 'puchkay' (*puchkay[0]=e, puchkay[1]=l,* etc.). The next programme will get this value by accessing *WWW_ele_3_5* through *getenv*. Thus in *matrix.cgi* *sprintf* is used to print *WWW_ele_3_5* into its own 'puchkay', which is fed to *getenv*.
- The second CGI programme, *matrix.cgi*, simply displays the values of the elements of the matrix. For this purpose, it creates names like

WWW_ele_3_5 using *sprintf. atof(getenv(puchkay))* allows us to access the values entered into the form. What we do with these values is up to us!

2.7 A list of form elements

text The most obvious entry element, it takes text input. The *size* attribute may be used to specify the size of the box. However, the actual entry may be as big as you like, the input scrolling over the box. In order to prevent malicious people/robots from submitting impossibly large inputs, the *maxsize* attribute may be used, but since the source code may be changed, it is absolutely necessary for the CGI programme to have some kind of a limiting scheme (use *fgets* instead of *gets*, for instance). The *value* attribute preloads a string into the widget, which may be overwritten.

password Just like *text* but the input does not appear - only stars are seen! If the *get* method is used, as in our case, the password is actually shown as plain text in the url! For real security, the *post* method should be used, where the values are actually sent after the connection is established, by connecting to the *stdin* of the CGI programme. *uncgi* can handle this very nicely, so just put *method=post* and you are done. However, the communication between the two machines is normally not encrypted. Use *https* in the url to ensure encryption. This requires setting up the *ssl* module in the server - an extra complexity :)

hidden Not shown in the form, but can be used to park values inside the form which the next CGI programme can use. The other alternative would be to use *magic cookies.*

submit The actual submit button which causes execution of the *action=* directive in the *form* tag. These buttons have names and can be attached to a string through the *value* tag. A very good use is to have multiple buttons having the same name but different values. By reading this value via *getenv*, our calculator programme is able to find out the action/operation desired.

radio Typical radiobuttons - you make more than one using sets which have the name in common and different values within a set. If you press one

within a set (as determined from *name=*) the others pop out. They do not normally cause a form submission (CHECK - is the action= attribute valid for radiobuttons?) but allow one to choose one of various choices. Basically multiple-choice options, they do not display the value (unlike the submit buttons) and so the legend for a particular member of a set of radiobuttons should be shown separately. The *checked* (just *checked*, not *checked=*) directive can be used to pre-select a default option.

textarea This is ideal for large text entry. The syntax is like `<textarea rows=10 cols=20 name=halum>This is nice</textarea>`. A pane for input is opened in the browser with *This is nice* preloaded. Scroll bars appear automatically if the text exceeds the given pane size.

select This is a drop-down menu. The syntax is:

```
<select name=goofy size=3>
<option value=one>First Class</option>
<option value=two>Third Class</option>
<option value=three>Second Class</option>
</select>
```

This wlll create a menu entry with a scroll bar which will display *size* number of options directly. The text of each option is what is written between the *<option>* and *</option>* tags. The value sent on submission, however, will be the one set using the *value=* directive.