# 4 Logic, loops and flow control

## 4.1 Syntax of C Flow of control

We can can use the following C constructs to control program execution.

When we can count our way through a sequence or series:

```
for( initial value; keep on until ; incremental change )
        { do this; and this; and this; }
```

When we are waiting for some condition to change:

```
while( this is true )
        { do this; and this; and this; }
```

or if we want to do something at least once then test:

```
do { do this; and this; and this; }
        while( this is true )
```

When we have a single option to test:

```
if( this is true )
        { do this; and this; and this; }
else
        { do this; and this; and this; }
```

When we have more options to test:

```
if( this is true )
        { do this; and this; and this; }
else if ( this is true )
         { do this; and this; and this; }
else
        { do this; and this; and this; }
```

When we have more options to test based on an integer or single character value:

```
switch( on an integer or character value )
{
        case 0: do this; and this; and this; break;
        case n: do this; and this; and this; break;
        default:do this; and this; and this; break;
}
```

## 4.2 Controlling what happens and in which order

This part is all about **if**, and **then**, and **else** and **true** and **false** – the nuts and bolts of how we express and control the execution of a program. This can be very dry and dusty material so to make it more understandable we are going to solve a problem you are going to need to solve to do any interactive web work of any complexity.

We will build something we can use in order to provide something like the functionality that can be obtained from typical **getParameter("ITEM1")** method in Java servlets or **$_REQUEST["ITEM1"]** function in PHP.

In Chapter 1 we saw that environment variables can be accessed by the implicit argument to the main function. We can also use the library function **getenv()** to request the value of any named environment variable.

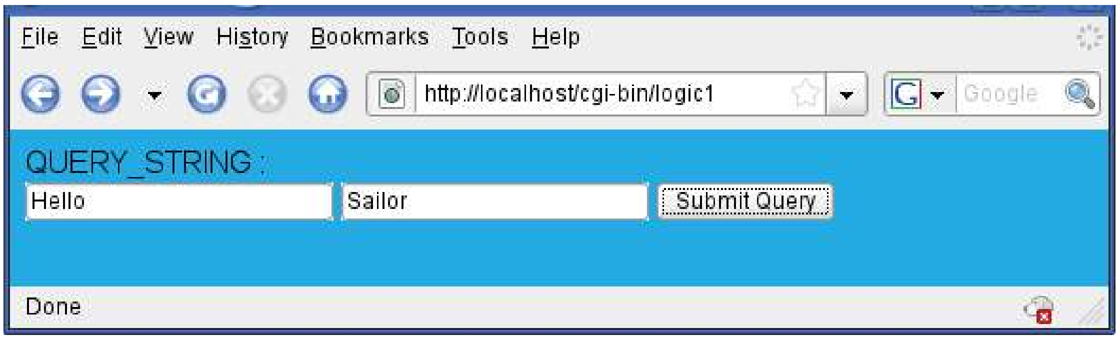
```
/******************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter4_1.c                                 *
******************************************************************/
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[], char *env[])
{
        printf("Content-type:text/html\n\n<html><body bgcolor=#23abe2>\n");
        char value[256] = "";
    strncpy(value,(char *) getenv("QUERY_STRING"),255);
        printf("QUERY_STRING : %s<BR>\n", value );
        printf("<form>\n");
        printf("<input type=\"TEXT\" name=\"ITEM1\">\n");
        printf("<input type=\"TEXT\" name=\"ITEM2\">\n");
        printf("<input type=\"SUBMIT\">");
        printf("</form></body></html>\n");
  return 0;
}
```

Here we display the **QUERY_STRING** which is what the program gets as the entire contents of an HTML form which contains NAME=VALUE pairs for all the named form elements.

An HTML form by default uses the **GET** method which transmits all form data back to the program or page that contains the form unless otherwise specified in an action attribute. This data is contained in the QUERY_STRING as a series of variable = value pairs separated by the & character.

Note that in HTML values of things are enclosed in quotation marks, so to embed these inside a C string we have to "escape" the character with a special sign \ like this **"\"ITEM1\" ".** Also we are using "\n" or explicit new line characters at the end of each piece of HTML output, so that when we select "view source" in the browser we get some reasonably formatted text to view rather than the whole page appearing as one long single line.

Calling this program in a browser we see a form and can enter some data in the boxes:

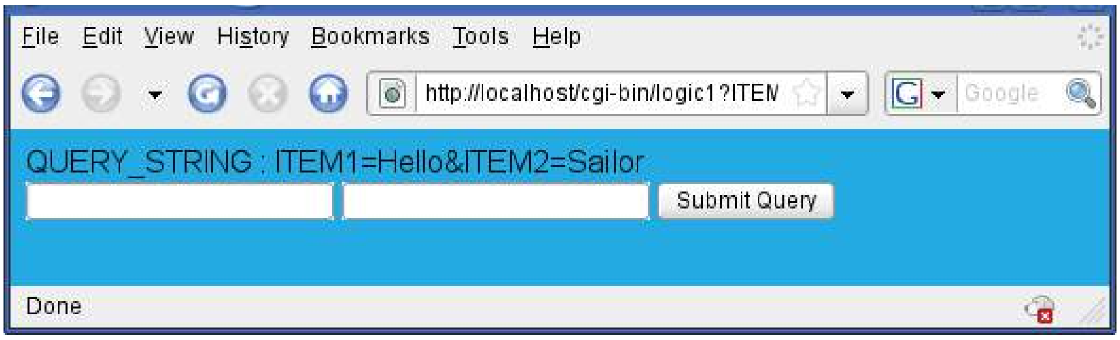

File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic1
Google
Submit Query
Done

QUERY_STRING :

Hello

Sailor

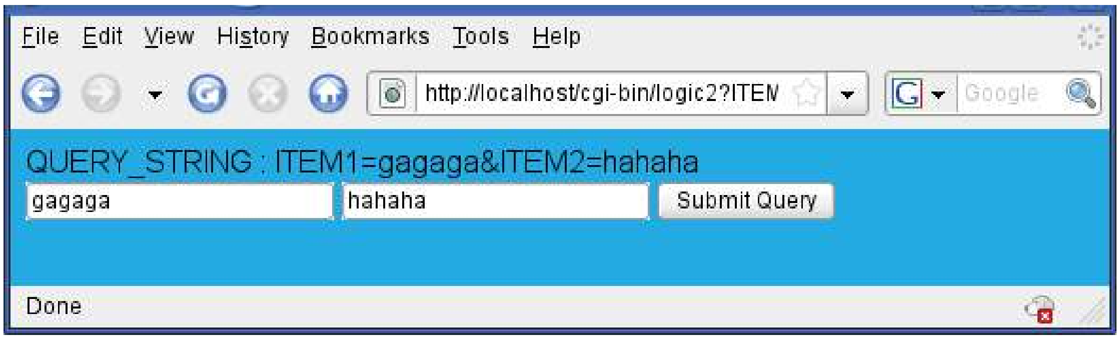
And after submitting the form we see:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic1?ITEM
Google
Submit Query
Done

```
QUERY_STRING : ITEM1=Hello&ITEM2=Sailor
```

To make much sense of the QUERY_STRING and find a particular value in it, we are going to have to **parse** it, to chop it up into its constituent pieces and for this we will need some **conditional logic** (if, else etc.) and some **loop** to count through the characters in the variable. A basic function to do this would ideally be created as this is a task you might need to do do again and again so it makes sense to have a chunk of code that can be called over again.

In the next example we add this function and the noticeable difference in the output is that we can insert the extracted values into the HTML boxes after we have parsed them. We seem to have successfully created something like a java getParameter() function – or have we?


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic2?ITEM
Google
Submit Query
Done

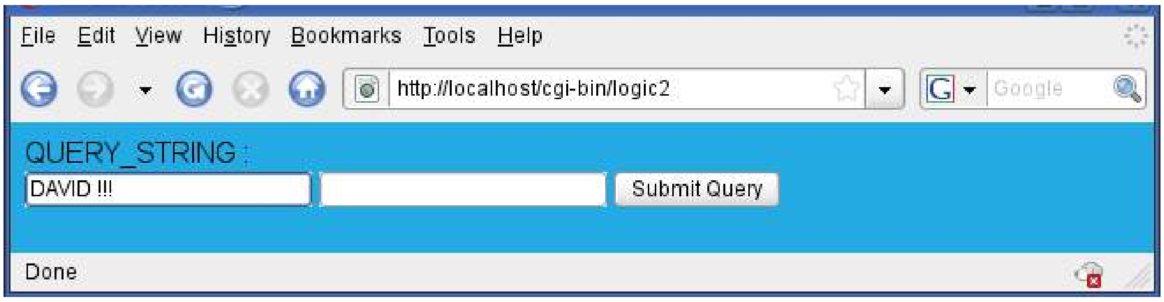
QUERY_STRING : ITEM1=gagaga&ITEM2=hahaha

gagaga

hahaha

Have a good long look at chapter4_2.c and try it out with characters other than A-Z a-z or numerals and you will see something is not quite right. There is some kind of encoding going on here!

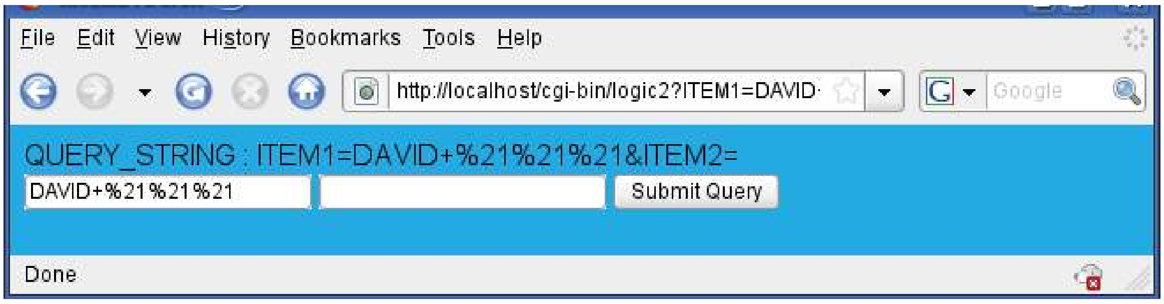
If I were tp type **DAVID !!!** into the first field:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic2
Google
Submit Query
Done

QUERY_STRING :

DAVID !!!

I get this result:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic2?ITEM1=DAVID
Google
Submit Query
Done

QUERY_STRING : ITEM1=DAVID+%21%21%21&ITEM2=

DAVID+%21%21%21

A **space** character has become a + and **!** has become **%21.**

This encoding occurs because certain characters are explicitly used in the transmission protocol itself. The **&** for example is used to separate portions of the QUERY_STRING and the space cannot be sent at all as it is.

Any program wishing to use information from the HTML form must be able to decode all this stuff which will now attempt to do.

The program chapter4_2.c accomplishes what we see so far. It has a main function and a decode_value function all in the same file.

The decode_value function takes three arguments:

> the name of the value we are looking for "ITEM1=" or "ITEM2=".
> the address of the variable into which we are going to put the value if found
> the maximum number of characters to copy

The function looks for the start and end positions in the QUERY_STRING of the value and then copies the characters found one by one to the value variable, adding a NULL charcter to terminate the string.

```
/*************************************************************** ****
* C Programming in Linux (c) David Haskins 2008
* chapter4_2.c                                                    *
******************************************************************/
#include <stdio.h>
#include <string.h>

void decode_value(const char *key, char *value, int size)
{
        int length = 0, i = 0, j = 0;
        char *pos1 = '\0', *pos2 = '\0';
        //if the string key is in the query string
        if( ( pos1 = strstr((char *) getenv("QUERY_STRING"), key)) != NULL )
        {
                //find start of value for this key
                for(i=0; i<strlen(key); i++) pos1++;
                //find length of the value
                if( (pos2 = strstr(pos1,"&")) != NULL )
                        length = pos2 - pos1;
                else length = strlen(pos1);
                //character by character, copy value from query string
                for(i = 0, j = 0; i <  length ; i++, j++)
                {
                        if(j < size) value[j] = pos1[i];
                }
                //add NULL character to end of the value
                if(j < size) value[j] = '\0';
                else value[size-1] = '\0';
        }
}
int main(int argc, char *argv[], char *env[])
{
        printf("Content-type:text/html\n\n<html><body bgcolor=#23abe2>\n");
        char value[255] = "";
        strncpy(value,(char *) getenv("QUERY_STRING"),255);
        printf("QUERY_STRING : %s<BR>\n", value );
        printf("<form>\n");
        //call the decode_value function to get value of "ITEM1"
        decode_value( "ITEM1=", (char *) &value, 255);
        if(strlen(value) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM1\" value=\"%s\">\n",value);
        else
                printf("<input type=\"TEXT\" name=\"ITEM1\">\n");
        //call the decode_value function to get value of "ITEM2"
        decode_value( "ITEM2=", (char *) &value, 255);
        if(strlen(value) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM2\" value=\"%s\">\n",value);
        else
                printf("<input type=\"TEXT\" name=\"ITEM2\">\n");
        printf("<input type=\"SUBMIT\">");
        printf("</form></body></html>\n");
  return 0;
}
```

It looks like we are going to have to do some serious work on this decode_value package so as this is work we can expect to do over and over again it makes sense to write a **function** that can be **reused**.

First off we can put this function into a separate file called **decode_value.c** and create a file for all the functions we may write called **c_in_linux.h** and compile all this into a **library**. In the Make file we can add:

```
SRC_CIL = decode_value.c
OBJ_CIL =  decode_value.o
#CIL_INCLUDES = -I/usr/include/apache2 -I. -I/usr/include/apache2 -I/usr/include/apr-1
#CIL_LIBS = -L/usr/lib/mysql -lmysqlclient -L/usr/lib -lgd -
L/home/david/public_html/Ventus/code


all: lib_cil 4-4
lib_cil:
	gcc -c $(SRC_CIL)
	ar rcs c_in_linux.a $(OBJ_CIL)
	$(RM) *.o
4-4:
	gcc -o logic4 chapter4_3.c c_in_linux.a -lc
	cp logic4 /home/david/public_html/cgi-bin/logic4
```

This looks horrible and complex but all it means is this:
typing "**make all**" will:

compile all the *.c files listed in the list OBJ_SRC and into object files *.o
compile all the object files into a library archive called lib_c_in_linux.a
compile 4-4 using this new archive.

This is the model we will use to keep our files as small as possible and the share-ability of code at its maximum.

We can now have a simpler "main" function file, and files for stuff we might want to write as call-able functions from anywhere really which we do not yet know about. All this is organised into a **library file** (***.a** for **archive**) – these can also be compiled as dynamically loadable **shared objects *.so** whch are much like Windows DLLs. This exactly how all Linux software is written and delivered.

For example the MySQL C Application Programmers Interface (API) comprises:

all the header files in /usr/include/mysql
the library file /usr/lib/mysql/libmysqlclient.a

What we are doing really is how all of Linux is put together – we are simply adding to it in the same way.

Our **main** file now looks like this:

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter4_3.c                                          *
*****************************************************************/
#include <stdio.h>
#include <string.h>
#include "c_in_linux.h"

int main(int argc, char *argv[], char *env[])
{
        printf("Content-type:text/html\n\n<html><body bgcolor=#23abe2>\n");
        char value[255] = "";
        strncpy(value,(char *) getenv("QUERY_STRING"),255);
        printf("QUERY_STRING : %s<BR>\n", value );
        printf("<form>\n");
        //call the decode_value function to get value of "ITEM1"
        decode_value( "ITEM1=", (char *) &value, 255);
        if(strlen(value) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM1\"
value=\"%s\">\n",value);
        else
                printf("<input type=\"TEXT\" name=\"ITEM1\">\n");
        //call the decode_value function to get value of "ITEM2"
        decode_value( "ITEM2=", (char *) &value, 255);
        if(strlen(value) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM2\"
value=\"%s\">\n",value);
        else
                printf("<input type=\"TEXT\" name=\"ITEM2\">\n");
        printf("<input type=\"SUBMIT\">");
        printf("</form></body></html>\n");
   return 0;
}
```

This code calls the function decode_value in the same way but because the library, **c_in_linux.a** was linked in when it was compiled and as it has access to the header file **c_in_linux.h** that lists all the functions in the library it all works properly.

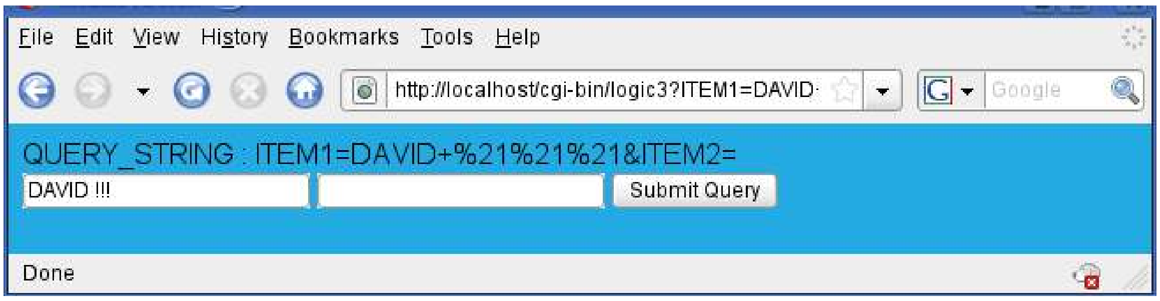
Try to describe the process in **pseudocode** of decoding this QUERY STRING:

```
get the QUERY_STRING
find the search string "ITEM1=" inside it
look for the end of the value of "ITEM1="
copy the value to our "value" variable, translating funny codes such as:
    %21 is ! %23 is #
```

These special codes are generated by the browser so that whatever you put in an HTML form will get safely transmitted and not mess about with the HTTP protocol. There are lot of them and the task for this chapter is to **finish this task off** so that EVERY key on your keyboard works as you think it should!!

Program chapter4_3.c calls this unfinished function decode_value which this far can only cope with the **space** character and **!** – it uses **if** and **else** and **for** and the library function **getenv**, **strcpy**, **strlen**, **ststr** in a piece of **conditional logic** in which a string is analysed to find a specific item and this thing then copied into a piece of memory called **value** which has been passed to it.

The result shows the decoded value pasted into the first field;


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic3?ITEM1=DAVID
Google
Submit Query
Done

```
QUERY_STRING : ITEM1=DAVID+%21%21%21&ITEM2=
DAVID !!!
```

```
/****************************************************************
* program: decode_value.c                                   *
* version: 0.1                                     *
* author: david haskins February 2008                      *
****************************************************************/
#include <stdlib.h>
#include <string.h>
void decode_value(const char *key, char *value, int size)
{
	unsigned int length = 0;
	unsigned int i = 0;
	int j = 0;
	char *pos1 = '\0',*pos2 = '\0', code1 = '\0',code2 = '\0';

	strcpy(value,"");
	if( ( pos1 = strstr(getenv("QUERY_STRING"), key)) != NULL )
	{
		for(i=0; i<strlen(key); i++) pos1++;
		if( (pos2 = strstr(pos1,"&")) != NULL )
		{
			length = pos2 - pos1;
		}
		else length = strlen(pos1);
		for(i = 0, j = 0; i <  length ; i++, j++)
		{
			if(j < size)
			{
				if(pos1[i] == '%')
				{
					i++;	code1 = pos1[i];
					i++;	code2 = pos1[i];
					if(code1 == '2' && code2== '0')
						value[j] = ' ';//0x20
					else if(code1 == '2' && code2== '1')
						value[j] = '!';//0x21
				}
				else value[j] = pos1[i];
			}
		}
		if(j < size)
		{
			value[j] = '\0';
		}
		else value[size-1] = '\0';
	}
}
```

## 4.3 Logic, loops and flow conclusion

The most important part of controlling the flow of your program is to have a clear idea about what it is you are trying to do. We have also learned to break our code up into manageable lumps, and started to build and use a library and header file of our own.

Being able to express a process in normal words or **pseudocode** is useful and helps you to break the code into steps.

Use **for loops** to explicitly count through things you know have an ending point.
Use **while** and **do…while** loops to do things until some condition changes.
Use **switch** statements to when integers or single characters determine what happens next.
Use **if** and **else if** and **else** when mutually exclusive things can be tested in a sequence.
Complex sets of **if** and **else** and not (**!**) conditionals can end up unreadable.
Use braces ({ }) to break it all up into chunks.

**Exercise:**

A useful **task** now would be to **complete the function decode_value** so you have a useful tool to grab web content from HTML forms decoding all the non alpha-numeric keys on your keyboard.
You will use this exercise again and again so it is worth getting it right.