Setting up your System

This book presumes you are using the Linux operating system with either a KDE or Gnome desktop. Specific set-up instructions for common architectures are at http://www.ubiubi.org/CProgrammingInLinux/

If you are using the KDE desktop you will have Konqueror or Dolphin as the File Manager and kate or kedit for an editor

In Gnome you would probably use Nautilus and gedit

You need to be familiar with the idea of doing some things as "super user" so that you have access permission to copy or edit certain files. This is normally done by prefacing the Linux command with "sudo" and providing the password, as in this example:

```
"sudo cp hello3 /srv/www/cgi-bin/hello3"
```

which copies the file "hello3" to the area where the Apache server locates common gateway interface or cgi programs.

In KDE “kdesu konqueror” would open a file manager as super user.

In Gnome “gnomesu nautilus” would open a file manager as super user.

You will need to have installed the following packages:

package	Ubuntu	Open Suse
C development libraries	build-essential	Base Development (pattern)
Apache web server	apache2	Web and LAMP Server (pattern)
Apache development libraries	apache2-prefork-dev	apache2-devel
MySQL server, client and development libraries	mysql-server libmysqlclient-dev	libmysqlclient-devel
GD and development libraries	libgd2-xpm libgd2-xpm-dev	gd gd-devel

Throughout the text you will see references to the folder **cgi-bin**. The location of this will vary between Linux distributions. By default this folder used for web programs is:

OpenSuse:	**/srv/www/cgi-bin**
Ubuntu:	**/usr/lib/cgi-bin**

To place programs there you need superuser rights, so it may be better to create a folder inside your own home/*****/public_html/cgi-bin directory and change the **ScriptAlias** and associated **Directory** references inside the Apache configuration files (OpenSuse) **/etc/apache2/default-server.conf** or (Ubuntu) **/etc/apache2/sites-available/default.**

1 Hello World

1.1 Hello Program 1

Using the File Manager (in KDE, Konqueror or in Gnome, Nautilus) create a new directory somewhere in your home directory called something appropriate for all the examples in this book, perhaps "Programming_In_Linux" without any spaces in the name.

Open an editor (in KDE, kate, or in Gnome, gedit) and type in (or copy from the supplied source code zip bundle) the following:

```
/*****************************************************************
C Programming in Linux (c) David Haskins 2008
chapter1_1.c
*****************************************************************/
#include <stdio.h>


int main(int argc, char *argv[])
{
	printf("Hello, you are learning C!!\n");
	return 0;
}
```

Save the text as **chapter1_1.c** in the new folder you created in your home directory.

Open a terminal window and type: **gcc -o hello chapter1_1.c**
to compile the program into a form that can be executed.

Now type "ls -l" to list the details of all the files in this directory. You should see that chapter1_2.c is there and a file called "hello" which is the compiled C program you have just written.

Now type: **./hello**
to execute, or run the program and it should return the text:

"Hello you are learning C!!".

If this worked, congratulations, you are now a programmer!

Anatomy of the program:

The part inside /*** ***/ is a comment and is not compiled but just for information and reference.

The "#include..." part tells the compiler which system libraries are needed and which header files are being referenced by this program. In our case "printf" is used and this is defined in the stdio.h header.

The "int main(int argc, char *argv[])" part is the start of the actual program. This is an entry-point and most C programs have a main function.

The "int argc" is an argument to the function "main" which is an integer count of the number of character string arguments passed in "char *argv[]" (a list of pointers to character strings) that might be passed at the command line when we run it.

A pointer to some thing is a name given to a memory address for this kind of data type. We can have a pointer to an integer: int *iptr, or a floating point number: float *fPtr. Any list of things is described by [], and if we know exactly how big this list is we might declare it as [200]. In this case we know that the second argument is a list of pointers to character strings.

Everything else in the curly brackets is the main function and in this case the entire program expressed as lines.

Each line or statement end with a semi-colon ";".

We have function calls like "printf(...)" which is a call to the standard input / output library defined in the header file stdio.h.

At the end of the program "return 0" ends the program by returning a zero to the system.

Return values are often used to indicate the success or status should the program not run correctly.

1.2 Hello Program 2

Taking this example a stage further, examine the start of the program at the declaration of the entry point function: int main(int argc, char *argv[])

In plain English this means:

The function called "main", which returns an integer, takes two arguments, an integer called "argc" which is a count of the number of command arguments then *argv[] which is a list or array of pointers to strings which are the actual arguments typed in when you run the program from the command line.

Some Definitions:

function: a block of program code with a **return data type**, a name, some arguments of varying data types separated by commas, enclosed in **brackets**, then the body of the function enclosed in **curly brackets**, each statement ending with a **semi-colon**.
integer symbol **int** : a counting number like 0,1,2,3,4,5.
list, array symbol **[]:** a sequence of things of the same kind in a numbered order.
pointer symbol * : a memory address locating the start of piece of data of a certain type.
string or **char** * : a pointer to a sequence of characters like 'c' ,'a', 't' making up "cat". A character string ends with s special character NULL or '\0' ascii value 0 or hex 00

Let's rewrite the program to see what all this means before we start to panic.

```
/*****************************************************************
C Programming in Linux (c) David Haskins 2008
chapter1_2.c
*****************************************************************/
#include <stdio.h>


int main(int argc, char *argv[])
{
        int i=0;
    printf("Hello, you are learning C!!\n");
    printf("Number of arguments to the main function:%d\n", argc);
        for(i=0; i<argc; i++)
        {
        printf("argument number %d is %s\n", i, argv[i]);
        }
    return 0;
}
```

Save the text as **chapter1_2.c** in the same folder.

Open a terminal window and type:
gcc -o hello2 chapter1_2.c to compile the program into a form that can be executed.

Now type **ls -l** to list the details of all the files in this directory. You should see that chapter1_2.c is there and a file called **hello2** which is the compiled C program you have just written.

Now type **./hello2** to execute, or run the program and it should return the text:

```
Hello, you are still learning C!!
Number of arguments to the main function:1
argument number 0 is ./hello2
```

We can see that the name of the program itself is counted as a command line argument and that the counting of things in the list or array of arguments starts at zero not at one.

Now type **./hello2 my name is David** to execute the program and it should return the text:

```
Hello, you are still learning C!!
Number of arguments to the main function:5
argument number 0 is ./hello2
argument number 1 is my
argument number 2 is name
argument number 3 is is
argument number 4 is David
```

So, what is happening here? It seems we are reading back each of the character strings (words) that were typed in to run the program.

Anatomy of the program:

printf("Hello, you are learning C!!\n");

the library function printf is called with one argument, a character string ending with a \n or new line character.

printf("Number of arguments to the main function:%d\n", argc);

the library function printf is called with two arguments, a character string ending with a \n that includes %d as a placeholder for the second argument argc which is an int.

for(i=0; i<argc; i++)

is a "for loop" in which we do something repeatedly using a counter integer i which is incremented (by the expression i++) at each iteration or looping which continues while i stays less than the value of argc

printf("argument number %d is %s\n", i, argv[i]);

the library function printf is called with three arguments, a character string ending with a \n that includes %d as a placeholder for the second argument argc which is an int, and %s which is a placeholder for the third argument argv[i], the i-th member of the array of pointers to character strings called argv[].

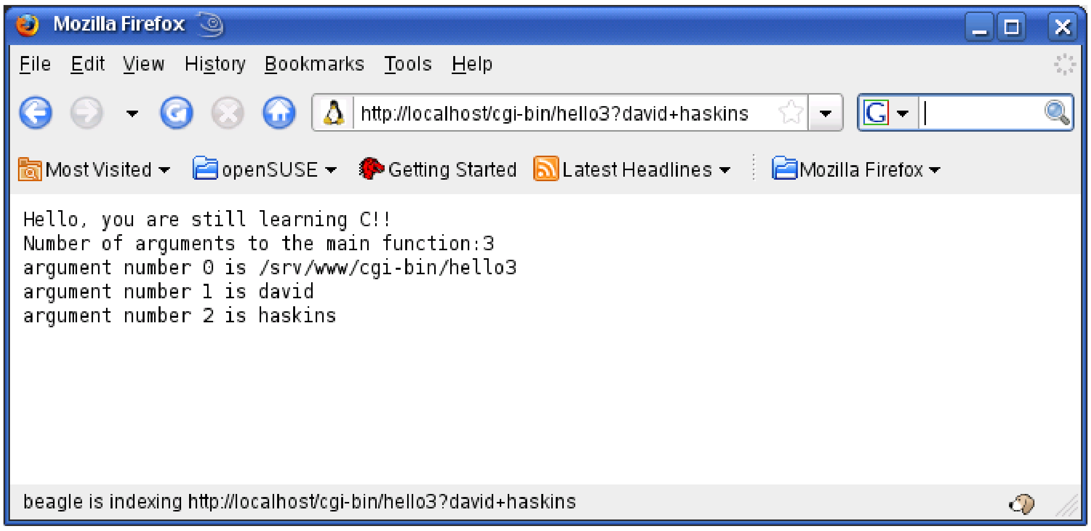
1.3 Hello Program 3

Lets get real and run this in a web page. Make the extra change adding the first output printf statement "Content-type:text/plain\n\n" which tells our server what kind of MIME type is going to be transmitted.

Compile using **gcc -o hello3 chapter1_3.c** and copy the compiled file hello3 to your public_html/cgi-bin directory (or on your own machine as superuser copy the program to /srv/www/cgi-bin (OpenSuse) or /usr/lib/cgi-bin (Ubuntu)).

```
/*******************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter1_3.c                                          *
*******************************************************************/
#include <stdio.h>


int main(int argc, char *argv[])
{
        int i=0;

    printf("Content-type:text/plain\n\n");
    printf("Hello, you are still learning C!!\n");
    printf("Number of arguments to the main function:%d\n", argc);
        for(i=0;i<argc;i++)
        {
        printf("argument number %d is %s\n", i, argv[i]);
        }
     return 0;
}
```

Open a web browser and type in the URL http://localhost/cgi-bin/hello3?david+haskins and you should see that web content can be generated by a C program.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/hello3?david+haskins
Most Visited
openSUSE
Getting Started
Latest Headlines
Mozilla Firefox

```
Hello, you are still learning C!!
Number of arguments to the main function:3
argument number 0 is /srv/www/cgi-bin/hello3
argument number 1 is david
argument number 2 is haskins
```

```
beagle is indexing http://localhost/cgi-bin/hello3?david+haskins
```

1.4 Hello Program 4

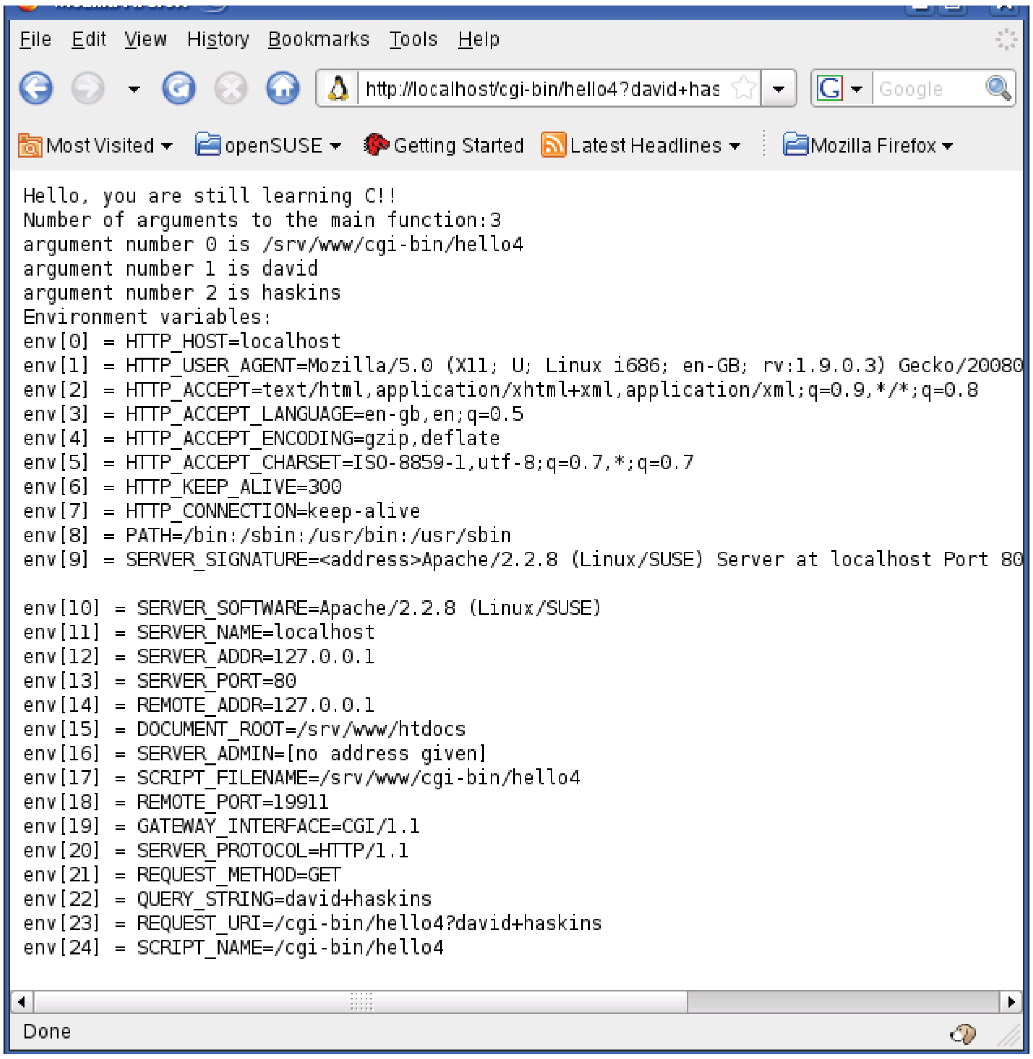
A seldom documented feature of the function signature for "main" is that it can take **three** arguments and the last one we will now look at is char *env[ ] which is also a list of pointers to strings, but in this case these are the **system environment variables** available to the program at the time it is run

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter1_4.c                                   *
*****************************************************************/
#include <stdio.h>


int main(int argc, char *argv[], char *env[])
{
        int i=0;

    printf("Content-type:text/plain\n\n");
    printf("Hello, you are still learning C!!\n");
    printf("Number of arguments to the main function:%d\n", argc);
        for(i=0;i<argc;i++)
        {
        printf("argument number %d is %s\n", i, argv[i]);
        }
        i = 0;
        printf("Environment variables:\n");
        while(env[i])
        {
                printf("env[%d] = %s\n", i, env[i]);
                i++;
        }
    return 0;
}
```

Compile with **gcc -o hello4 chapter1_4.c** and as superuser copy the program to /srv/www/cgi-bin (OpenSuse) or /usr/lib/cgi-bin (Ubuntu). You can run this from the terminal where you compiled it with **./hello4** and you will see a long list of environment variables. In the browser when you enter http://localhost/cgi-bin/hello4 you will a different set altogether.

Wikipedia defines **environment variables** like this:

“In all Unix and Unix-like systems, each process has its own private set of environment variables. By default, when a process is created it inherits a duplicate environment of its parent process, except for explicit changes made by the parent when it creates the child........ All Unix operating system flavors as well as DOS and Microsoft Windows have environment variables; however, they do not all use the same variable names. Running programs can access the values of environment variables for configuration purposes. Examples of environment variables include...... PATH, HOME... “


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/hello4?david+has
Google
Most Visited
openSUSE
Getting Started
Latest Headlines
Mozilla Firefox
Done

```
Hello, you are still learning C!!
Number of arguments to the main function:3
argument number 0 is /srv/www/cgi-bin/hello4
argument number 1 is david
argument number 2 is haskins
Environment variables:
env[0] = HTTP_HOST=localhost
env[1] = HTTP_USER_AGENT=Mozilla/5.0 (X11; U; Linux i686; en-GB; rv:1.9.0.3) Gecko/20080
env[2] = HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
env[3] = HTTP_ACCEPT_LANGUAGE=en-gb,en;q=0.5
env[4] = HTTP_ACCEPT_ENCODING=gzip,deflate
env[5] = HTTP_ACCEPT_CHARSET=ISO-8859-1,utf-8;q=0.7,*;q=0.7
env[6] = HTTP_KEEP_ALIVE=300
env[7] = HTTP_CONNECTION=keep-alive
env[8] = PATH=/bin:/sbin:/usr/bin:/usr/sbin
env[9] = SERVER_SIGNATURE=<address>Apache/2.2.8 (Linux/SUSE) Server at localhost Port 80

env[10] = SERVER_SOFTWARE=Apache/2.2.8 (Linux/SUSE)
env[11] = SERVER_NAME=localhost
env[12] = SERVER_ADDR=127.0.0.1
env[13] = SERVER_PORT=80
env[14] = REMOTE_ADDR=127.0.0.1
env[15] = DOCUMENT_ROOT=/srv/www/htdocs
env[16] = SERVER_ADMIN=[no address given]
env[17] = SCRIPT_FILENAME=/srv/www/cgi-bin/hello4
env[18] = REMOTE_PORT=19911
env[19] = GATEWAY_INTERFACE=CGI/1.1
env[20] = SERVER_PROTOCOL=HTTP/1.1
env[21] = REQUEST_METHOD=GET
env[22] = QUERY_STRING=david+haskins
env[23] = REQUEST_URI=/cgi-bin/hello4?david+haskins
env[24] = SCRIPT_NAME=/cgi-bin/hello4
```

We will soon find out that **QUERY_STRING** is an important environment variable for us in communicating with our program and in this case we see it has a value of "david+haskins" or everything after the "?" in the URL we typed. It is a valid way to send information to a **common gateway interface** (CGI) program like hello4 but we should restrict this to just one string. In our case we have used a "+" to join up two strings. If we typed: "david haskins" the browser would translate this so we would see:

```
QUERY_STRING=david%20haskins
```

We will learn later how complex sets of input values can be transmitted to our programs.

1.5 Hello World conclusion

We have seen that a simple program with a tiny bit of input and some output is in fact extremely powerful in that it reveals and exposes the inner workings of a great deal of our computer.

Even though we have just begun we have encountered many of the key concepts we will use over and over again:

- functions and arguments
- Numbers (integers) and character strings as data types
- Lists or arrays
- Loops using “for” and “while”

We have made a deliberate big leap from writing a program that runs simply in a “terminal screen” to one which will be visible over the internet in a browser.

The reason for this is that the process of writing programs that interact with users in windowing systems like Windows, Gnome or KDE is extremely complex and not something you will be asked very often to do .

The internet browser has become the de facto interface mode for almost everything we do these days so we might as well understand using it from the start.

In all the successive chapters we will follow this model: starting off with some basic technique then applying it to a web-based system.

In practice there is not much real-world C common gateway interface programming going on but there is a great deal of C and C++ based code running as Apache modules and Microsoft IIS ISAPI Dlls. Perhaps not many know that much of Ebay is written in C / C++.

Why? It is as fast as things get and their business with the bargain snipers in a global real-time market needs this lightning fast core, so there is no other way to get that performance.

2 Data and Memory

2.1 Simple data types?

When we write programs we have to make decisions or assertions about the nature of the world as we declare and describe variables to represent the kinds of things we want to include in our information processing.

This process is deeply philosophical; we make **ontological** assertions that this or that thing exists and we make **epistemological** assertions when we select particular data types or collections of data types to use to describe the attributes of these things. Heavy stuff with a great responsibility and not to be lightly undertaken.

As a practical example we might declare something that looks like the beginnings of a database record for geography.

```
/*************************************************************** ***
C Programming in Linux (c) David Haskins 2008
chapter2_1.c
*****************************************************************/
#include <stdio.h>
#DEFINE STRINGSIZE 256

int main(int argc, char *argv[])
{
        char town[STRINGSIZE] = "Guildford";
        char county[STRINGSIZE] = "Surrey";
        char country[STRINGSIZE] = "Great Britain";
        int population = 66773;
        float latitude = 51.238599;
        float longitude = -0.566257;
        printf("Town name: %s  population:%d\n",town,population);
        printf("County: %s\n",county);
        printf("Country: %s\n",country);
        printf("Location latitude: %f longitude: %f\n",latitude,longitude);
        printf("char = %d byte int = %d bytes float = %d bytes\n",
        sizeof(char),sizeof(int),sizeof(float) );
        printf("memory used:%d bytes\n",
                ((STRINGSIZE * 3) * sizeof(char)) + sizeof(int) + (2 * sizeof(float)));
        return 0;
}
```

Here we are doing the following:

- asserting that all the character strings we will ever encounter in this application will be 255 limited to characters so we **define** this with a **preprocessor** statement – these start with #.
- assert that towns are associated with counties, and counties are associated with countries some hierarchical manner.
- assert that the population is counted in whole numbers – no half-people.
- assert the location is to be recorded in a particular variant (WGS84) of the convention of describing spots on the surface of the world in latitude and longitude that uses a decimal fraction for degrees, minutes, and seconds.

Each of these statements allocates **memory** within the **scope** of the function in which it is declared. Each **data declaration** will occupy an amount of memory in **bytes** and give that bit of memory a label which is the **variable name**. Each data type has a specified size and the **sizeof()** library function will return this as an integer. In this case 3 × 256 characters, one integer, and two floats. The exact size is machine dependent but probably it is 780 bytes.

Outside the function in which the data has been declared this data is inaccessible – this is the **scope** of declaration. If we had declared outside the main() function it would be **global** in scope and other functions could access it. C lets you do this kind of dangerous stuff if you want to, so be careful.

Generally we keep a close eye on the scope of data, and pass either **read-only copies**, or **labelled memory addresses** to our data to parts of the programs that might need to do work on it and even change it. These labelled memory addresses are called **pointers**.

We are using for output the **printf** family of library functions (**sprintf** for creating strings, **fprintf** for writing to files etc.) which all use a common **format string** argument to specify how the data is to be represented.

- %c character
- %s string
- %d integer
- %f floating point number etc.

The remaining series of variables in the arguments are placed in sequence into the format string as specified.

In C it is a good idea to **intialise** any data you declare as the contents of the memory allocated for them is not cleared but may contain any old rubbish.

Compile with: **gcc -o data1 chapter2_1.c -lc**
Output of the program when called with : **./data1**

```
Town name: Guildford population:66773
County: Surrey
Country: Great Britain
Location latitude: 51.238598 longitude: -0.566257
char = 1 byte int = 4 bytes float = 4 bytes
memory used:780 bytes
```

A note on *make* a helpful utility

By now you are probably getting bored typing in all these compiler commands and for this reason there is a utility called **make** that runs on a file called **Makefile** in the folder where your code is stored. Here is the Makefile for the examples so far:

```
#Makefile
all:chap1 chap2
chap1: 1-1 1-2 1-3 1-4
1-1:
	gcc -o hello1 chapter1_1.c  -lc
1-2:
	gcc -o hello2 chapter1_2.c  -lc
1-3:
	gcc -o hello3 chapter1_3.c  -lc
1-4:
	gcc -o hello4 chapter1_4.c  -lc
chap2: 2-1 2-2
2-1:
	gcc -o data1 chapter2_1.c  -lc
2-2:
	gcc -o data2 chapter2_2.c  -lc
clean:
	rm hello* data* *~
```

to compile everything type **make all**

to compile target 2-1 for chapter2_1.c type **make 2-1**

the tab after each make target is vital to the syntax of make

In the code bundle there is a Makefile for the whole book.

2.2 What is a string?

Some programming languages like Java and C++ have a **string data type** that hides some of the complexity underneath what might seem a simple thing.

An essential attribute of a character string is that it is a series of individual character elements of indeterminate length.

Most of the individual characters we can type into a keyboard are represented by simple numerical ASCII codes and the C data type **char** is used to store character data.

Strings are stored as **arrays** of characters ending with a NULL so an array must be large enough to hold the sequence of characters plus one. Remember array members are always counted from zero.

In this example we can see 5 individual characters declared and initialised with values, and an empty character array set to "".

Take care to notice the difference between single quote marks ' used around characters and double quote marks " used around character strings.

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_2.c                                      *
*****************************************************************/
#include <stdio.h>


int main(int argc, char *argv[], char *env[])
{
        char c1 = 'd';
        char c2 = 'a';
        char c3 = 'v';
        char c4 = 'i';
        char c5 = 'd';
        char name[6] = "";

        sprintf(name,"%c%c%c%c%c",c1,c2,c3,c4,c5);
        printf("%s\n",name);
   return 0;
}
```

Compile with: **gcc -o data2 chapter2_2.c -lc**
Output of the program when called with : **./data2**

```
david
```

2.3 What can a string "mean"

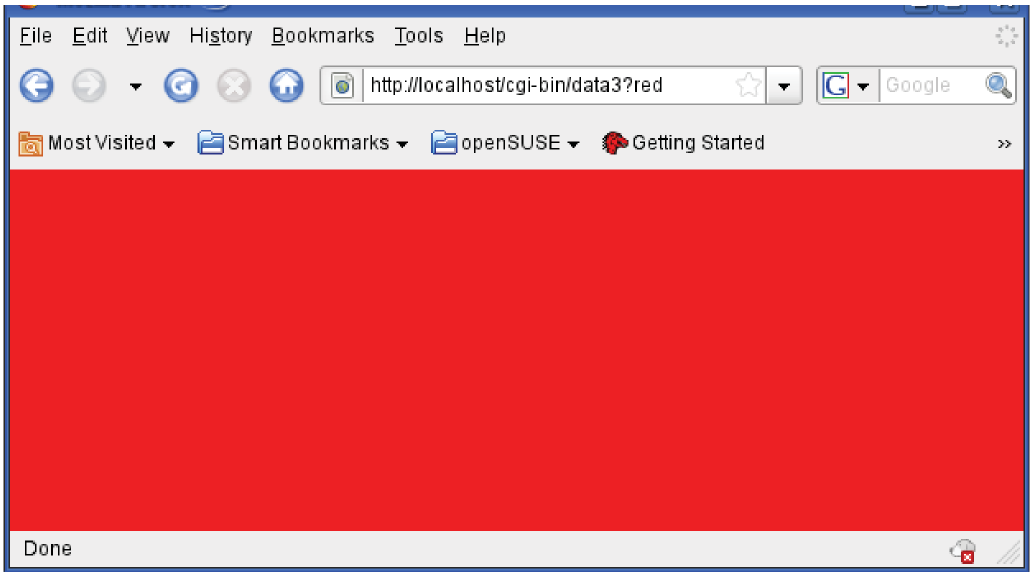
Anything at all – **name** given to a **variable** and its **meaning** or its **use** is entirely in the mind of the beholder. Try this

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_3.c                                      *
*****************************************************************/
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[], char *env[])
{
        printf("Content-type:text/html\n\n");
        printf("<html>\n");
        printf("<body bgcolor=\"%s\">\n",argv[1]);
        printf("</body>\n");
        printf("</html>\n");
   return 0;
}
```

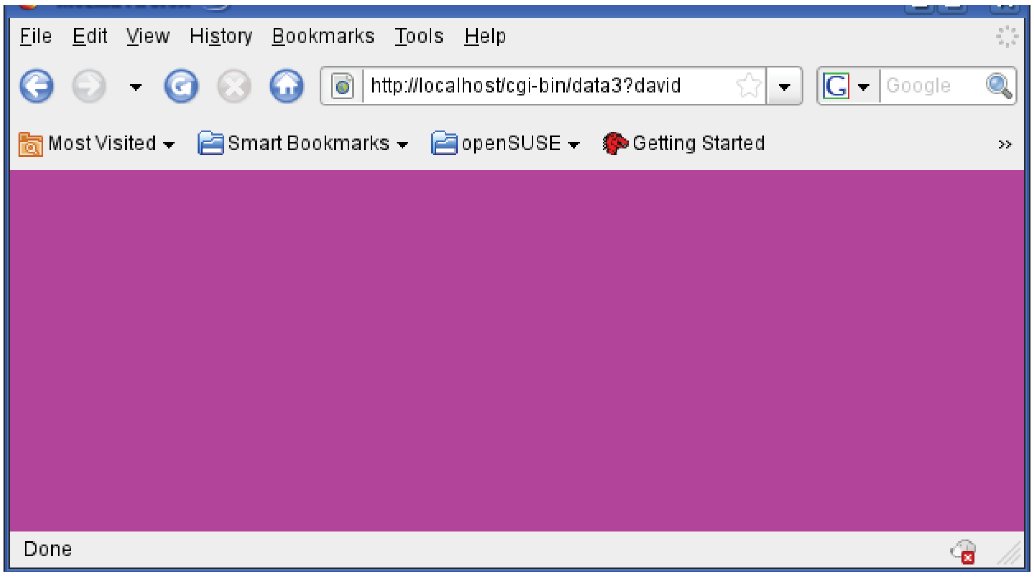
Compile with: **gcc -o data3 chapter2_3.c -lc**

As superuser copy the program to your public_html/cgi-bin directory (or /srv/www/cgi-bin (OpenSuse) or /usr/lib/cgi-bin (Ubuntu)).

In the browser enter: http://localhost/cgi-bin/data3?red
what you should see is this:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/data3?red
Google
Most Visited
Smart Bookmarks
openSUSE
Getting Started
Done

Or if send a parameter of anything at all you will get surprising results:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/data3?david
Google
Most Visited
Smart Bookmarks
openSUSE
Getting Started
Done

What we are doing here is **using** the string parameter **argv[1]** as a background colour code inside an **HTML** body tag. We change the Content-type specification to text/html and miraculously now our program is generating HTML content. A language being expressed inside another language. Web browsers understand a limited set of colour terms and colours can be also defined hexadecimal codes such as #FFFFFF (white) #FF0000 (red) #00FF00 (green) #0000FF (blue).

This fun exercise is not just a lightweight trick, the idea that one program can generate another in another language is very powerful and behind the whole power of the internet. When we generate HTML (or XML or anything else) from a **common gateway interface program** like this we are creating **dynamic content** that can be linked to live, changing data rather than **static** pre-edited web pages. In practice most web sites have a mix of dynamic and static content, but here we see just how this is done at a very simple level.

Throughout this book we will use the browser as the preferred interface to our programs hence we will be generating HTML and binary image stream web content purely as a means to make immediate the power of our programs. Writing code that you peer at in a terminal screen is not too impressive, and writing window-type applications is not nearly so straightforward.

In practice most of the software you may be asked to write will be running on the web so we might as well start with this idea straight away. Most web applications involve **multiple languages** too such as CSS, (X)HTML, XML, JavaScript, PHP, JAVA, JSP, ASP, .NET, SQL. If this sounds frightening, don't panic. A knowledge of C will show you that many of these languages, which all perform different functions, have a basis of C in their syntax.

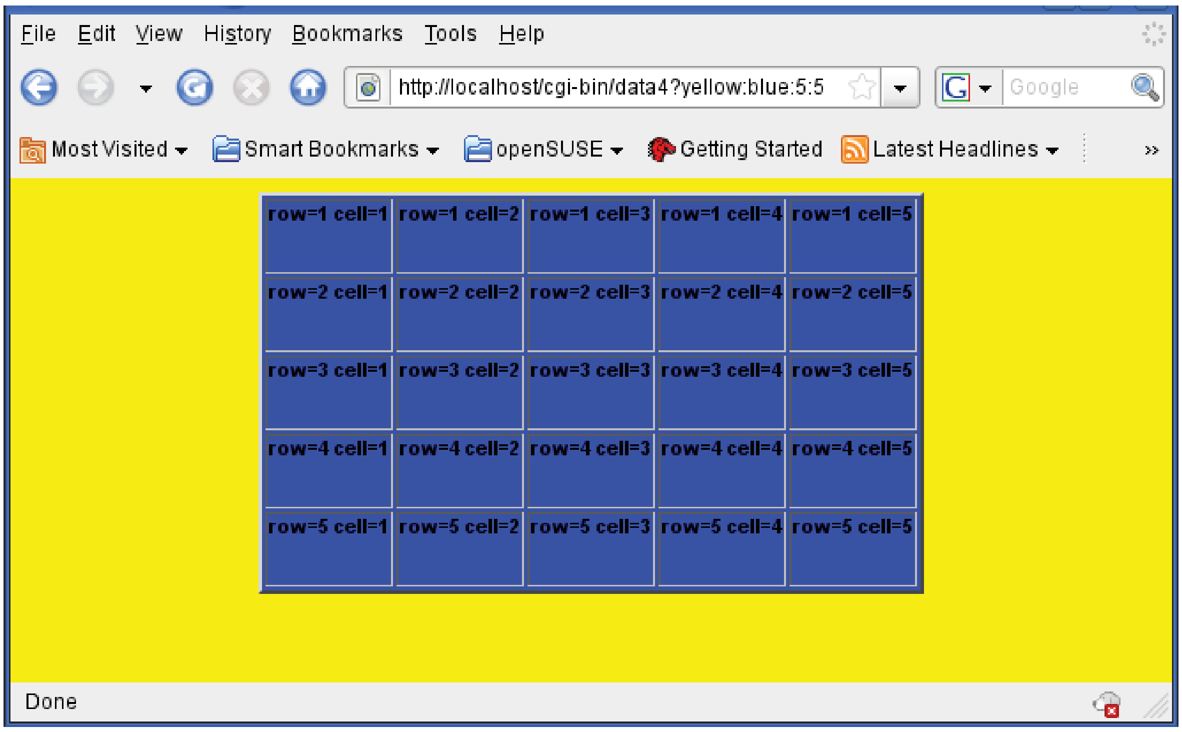
2.4 Parsing a string

The work involved in extracting meaning or valuable information from some kind of input string is called "parsing". We will now build another fun internet-callable CGI program to demonstrate the power in our hands.

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_4.c                                          *
*****************************************************************/
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[], char *env[])
{
        char *token = NULL;
        char colour1[256] = "";
        char colour2[256] = "";
        int wide = 0;
        int high = 0;
        int columns = 0;
        int rows = 0;

        token = (char *) strtok(argv[1],":");
        strcpy(colour1,token);
        token = (char *) strtok(NULL,":");
        strcpy(colour2,token);
        token = (char *) strtok(NULL,":");
        wide = atoi(token);
        token = (char *) strtok(NULL,":");
        high = atoi(token);
        printf("Content-type:text/html\n\n");
        printf("<html>\n");
        printf("<body bgcolor=\"%s\">\n",colour1);
        printf("<center>\n");
        printf("<table bgcolor=\"%s\" border=2>\n",colour2);
        for(rows=1;rows<=high;rows++)
        {
                printf("<tr>\n");
                for(columns=1;columns<=wide;columns++)
                {
                        printf("<td><h6>row=%d cell=%d</h6></td>\n",rows,columns);
                }
                printf("</tr>\n");
        }
        printf("</table>\n");
        printf("</body>\n");
        printf("</html>\n");
   return 0;
}
```

Compile with: **gcc -o data4 chapter2_4.c -lc**
As superuser copy the program to /srv/www/cgi-bin (OpenSuse) or /usr/lib/cgi-bin (Ubuntu).

In the browser enter:
http://localhost/cgi-bin/data4?red:blue:5:5:
what you should see is this:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/data4?yellow:blue:5:5
Google
Most Visited
Smart Bookmarks
openSUSE
Getting Started
Latest Headlines
Done

row=1 cell=1	row=1 cell=2	row=1 cell=3	row=1 cell=4	row=1 cell=5
row=2 cell=1	row=2 cell=2	row=2 cell=3	row=2 cell=4	row=2 cell=5
row=3 cell=1	row=3 cell=2	row=3 cell=3	row=3 cell=4	row=3 cell=5
row=4 cell=1	row=4 cell=2	row=4 cell=3	row=4 cell=4	row=4 cell=5
row=5 cell=1	row=5 cell=2	row=5 cell=3	row=5 cell=4	row=5 cell=5

In this program we take argv[1] which here is **yellow:blue:5:5:** and **parse** it using the library function **strtok** which chops the string into tokens separated by an arbitrary character ':' and use these tokens as strings to specify colours and integer numbers to specify the row and cell counts of a table.

The function **atoi** converts an string representation of a integer to an integer ("1" to 1).

The function **strtok** is a little odd in that the first time you call it with the string name you want to parse, then on subsequent calls the first parameter is changed to NULL.

The **for(…)** loop mechanism was used to do something a set number of times.

The HTML terms introduced were:

<html> <body> <table> <tr> table row **<td>** table data cell

2.5 Data and Memory – conclusion

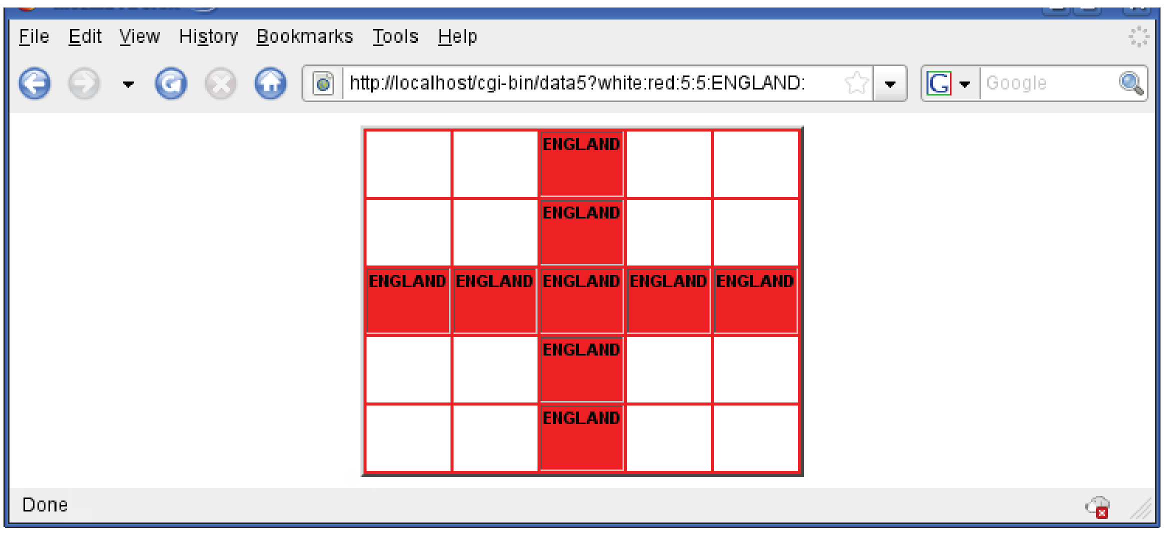
We have used some simple data types to represent some information and transmit input to a program and to organise and display some visual output.

We have used HTML embedded in output strings to make output visible in a web browser.

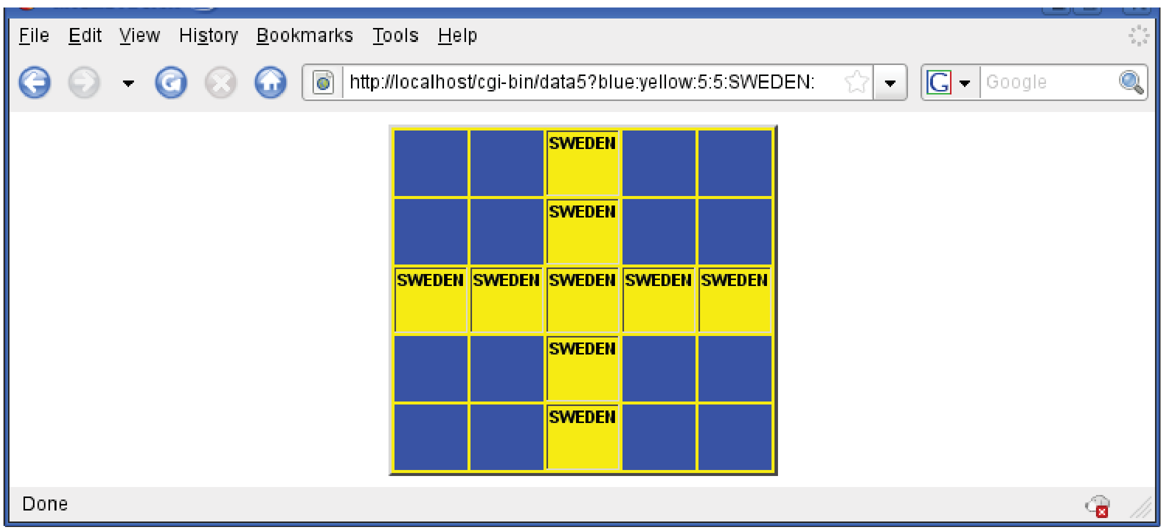
As an exercise try this:

Write a program to put into your **/public_html/cgi-bin** folder which can be called in a browser with the **name** of a **sports team** or a **country** and a series of **colours** specified perhaps as hexadecimals e.g. ff0000 = red (rrggbb) used for the team colours or map colours, and which displays something sensible. My version looks like this:

ie business school
#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013
#gobeyond
MASTER IN MANAGEMENT
Because achieving your dreams is your greatest challenge. IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.
Choose your area of specialization.
Customize your master through the different options offered.
Global Immersion Weeks in locations such as Rio de Janeiro, Shanghai or San Francisco.
Because you change, we change with you.
www.ie.edu/master-management
mim.admissions@ie.edu



		ENGLAND		
		ENGLAND		
ENGLAND	ENGLAND	ENGLAND	ENGLAND	ENGLAND
		ENGLAND		
		ENGLAND		



		SWEDEN		
		SWEDEN		
SWEDEN	SWEDEN	SWEDEN	SWEDEN	SWEDEN
		SWEDEN		
		SWEDEN		

3 Functions, pointers and structures

3.1 Functions

The entry point into all our programs is called main() and this is a **function**, or a **piece of code** that does something, usually returning some value. We structure programs into functions to stop them become long unreadable blocks of code than cannot be seen in one screen or page and also to ensure that we do not have repeated identical chunks of code all over the place. We can call **library functions** like printf or strtok which are part of the C language and we can call our own or other peoples functions and libraries of functions. We have to ensure that the appropriate header file exists and can be read by the preprocessor and that the source code or compiled library exists too and is accessible.

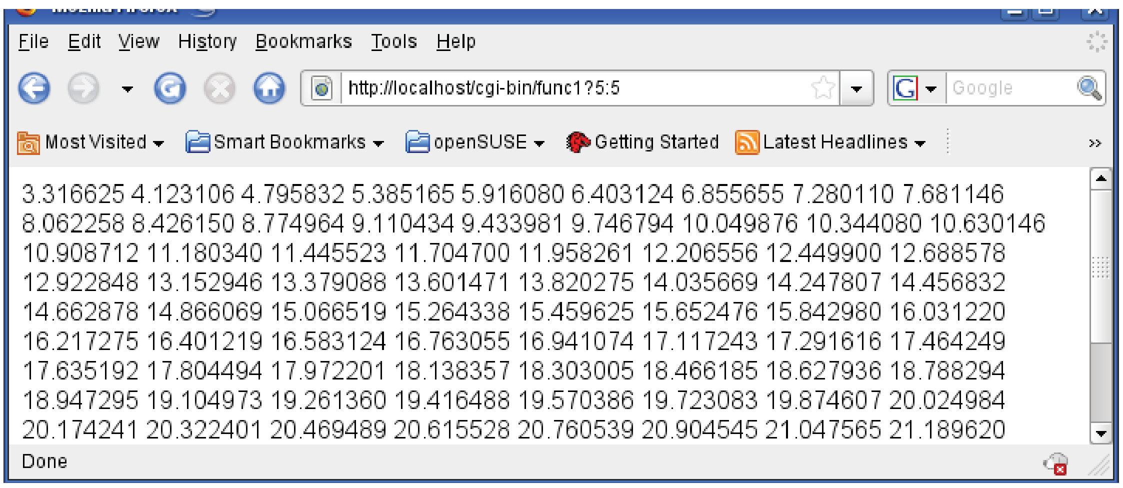
As we learned before, the **scope** of data is restricted to the **function** in which is was declared, so we use **pointers** to data and blocks of data to pass to functions that we wish to do some work on our data. We have seen already that strings are handled as pointers to arrays of single characters terminated with a NULL character.

```
/****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter3_1.c                                        *
****************************************************************/
#include <stdio.h>
#include <string.h>
#include <math.h>

double doit(int number1, int number2)
{
        return sqrt((double)(number1 + number2) );
}

int main(int argc, char *argv[], char *env[])
{
        int n1 = 0, n2 = 0, i=0;
        n1 = atoi((char *) strtok(argv[1],":"));
        n2 = atoi((char *) strtok(NULL,":"));
        printf("Content-type:text/html\n\n<html><body>\n");
        for(i=1;i<=100;i++)     printf("%f ",doit(n1+i,n2*i));
        printf("</body></html>\n");
        return 0;
}
```

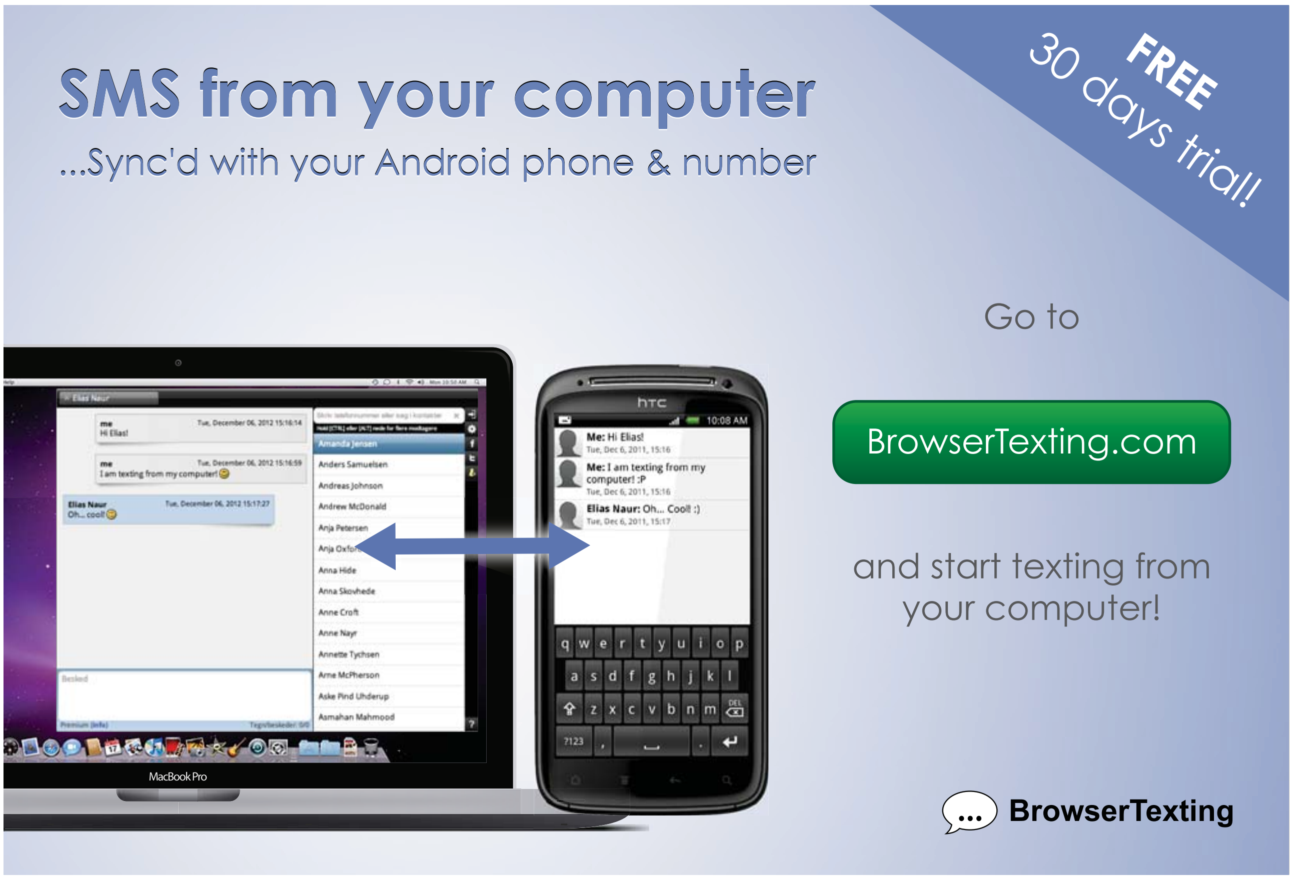
In this example we can repeatedly call the function "doit" that takes two integer arguments and reurns the result of some mathematical calculation.

Compile: gcc -o func1 chapter3_1.c -lm
Copy to cgi-bin: cp func1 /home/david/public_html/cgi-bin/func1

(You should be using the Makefile supplied or be maintaining a Makefile as you progress, adding targets to compile examples as you go.)


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/func1?5:5
Google
Most Visited
Smart Bookmarks
openSUSE
Getting Started
Latest Headlines
Done

```
3.316625 4.123106 4.795832 5.385165 5.916080 6.403124 6.855655 7.280110 7.681146
8.062258 8.426150 8.774964 9.110434 9.433981 9.746794 10.049876 10.344080 10.630146
10.908712 11.180340 11.445523 11.704700 11.958261 12.206556 12.449900 12.688578
12.922848 13.152946 13.379088 13.601471 13.820275 14.035669 14.247807 14.456832
14.662878 14.866069 15.066519 15.264338 15.459625 15.652476 15.842980 16.031220
16.217275 16.401219 16.583124 16.763055 16.941074 17.117243 17.291616 17.464249
17.635192 17.804494 17.972201 18.138357 18.303005 18.466185 18.627936 18.788294
18.947295 19.104973 19.261360 19.416488 19.570386 19.723083 19.874607 20.024984
20.174241 20.322401 20.469489 20.615528 20.760539 20.904545 21.047565 21.189620
```


SMS from your computer
...Sync'd with your Android phone & number
FREE
30 days trial!
Go to
BrowserTexting.com
and start texting from your computer!
BrowserTexting

The result in a browser looks like this called with "func1?5:5".

In this case the arguments to our function are sent as **copies** and are not modified in the function but used.

If we want to actual modify a variable we would have to send its pointer to a function.

```
/****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter3_2.c                                    *
****************************************************************/
#include <stdio.h>
#include <string.h>
#include <math.h>
void doit(int number1, int number2, double *result)
{
        *result =  sqrt((double)(number1 + number2) );
}
int main(int argc, char *argv[], char *env[])
{
        int n1 = 0, n2 = 0, i=0;
        double result = 0;
        n1 = atoi((char *) strtok(argv[1],":"));
        n2 = atoi((char *) strtok(NULL,":"));
        printf("Content-type:text/html\n\n<html><body>\n");
        for(i=1;i<=100;i++)
        {
                doit(n1+i,n2*i,&result);
                printf("%f ", result);
        }
        printf("</body></html>\n");
  return 0;
}
```

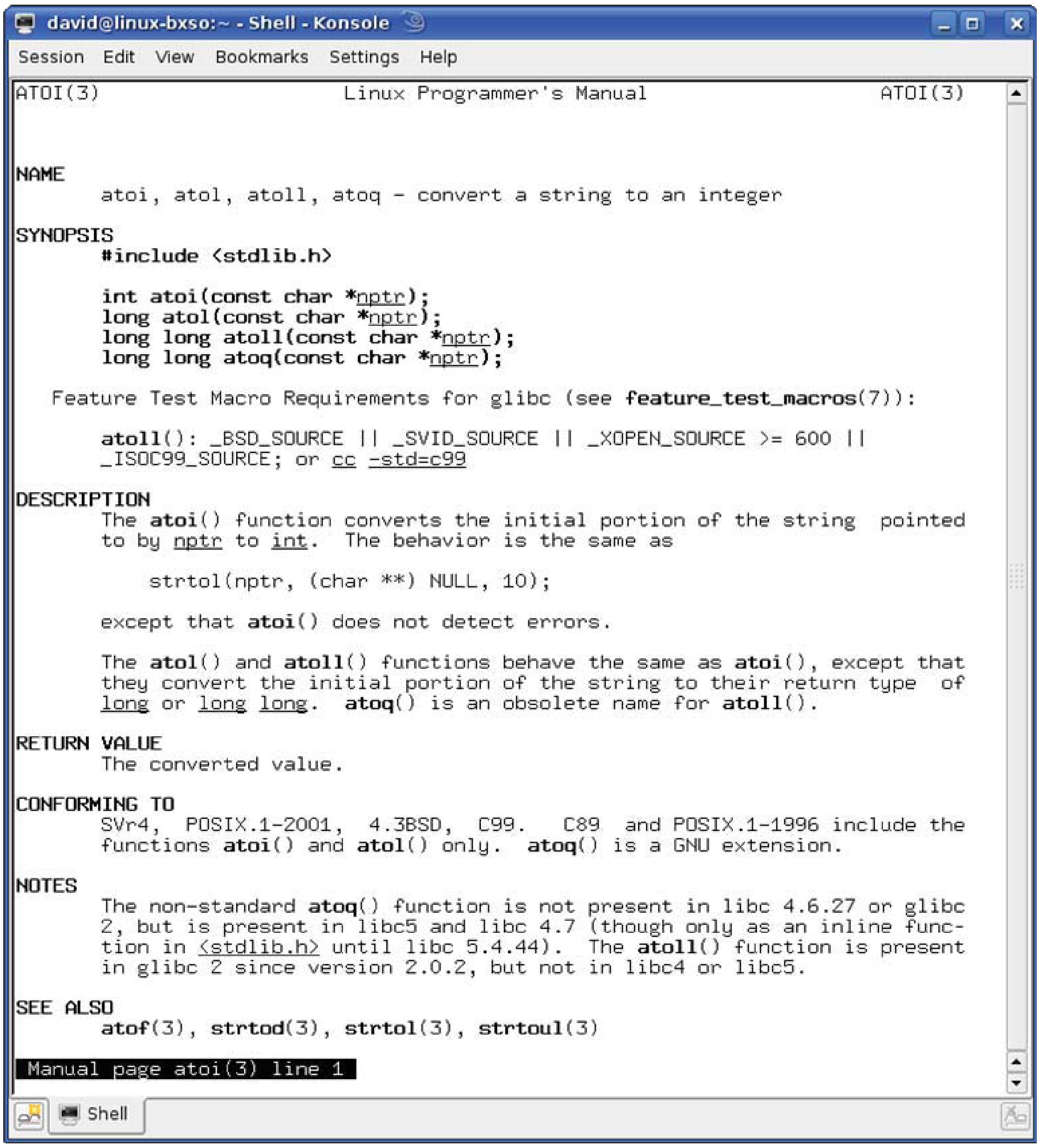
We send the address of the variable 'result' with **&result**, and in the function doit we ***de-reference*** the pointer with ***result** to get at the float and change its value, outside its scope inside main. This gives identical output to chapter3_1.c.

3.2 Library Functions

C contains a number of built-in functions for doing commonly used tasks. So far we have used **atoi**, **printf**, **sizeof**, **strtok**, and **sqrt.** To get full details of any built-in library function all we have to do is type for example:

```
man 3 atoi
```

and we will see all this:


david@linux-bxso:~ - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
ATOI(3)                    Linux Programmer's Manual                   ATOI(3)


NAME
       atoi, atol, atoll, atoq - convert a string to an integer

SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);
       long atol(const char *nptr);
       long long atoll(const char *nptr);
       long long atoq(const char *nptr);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       atoll(): _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600 ||
       _ISOC99_SOURCE; or cc -std=c99

DESCRIPTION
       The atoi() function converts the initial portion of the string  pointed
       to by nptr to int.  The behavior is the same as

           strtol(nptr, (char **) NULL, 10);

       except that atoi() does not detect errors.

       The atol() and atoll() functions behave the same as atoi(), except that
       they convert the initial portion of the string to their return type  of
       long or long long.  atoq() is an obsolete name for atoll().

RETURN VALUE
       The converted value.

CONFORMING TO
       SVr4,  POSIX.1-2001,  4.3BSD,  C99.   C89  and POSIX.1-1996 include the
       functions atoi() and atol() only.  atoq() is a GNU extension.

NOTES
       The non-standard atoq() function is not present in libc 4.6.27 or glibc
       2, but is present in libc5 and libc 4.7 (though only as an inline func-
       tion in <stdlib.h> until libc 5.4.44).  The atoll() function is present
       in glibc 2 since version 2.0.2, but not in libc4 or libc5.

SEE ALSO
       atof(3), strtod(3), strtol(3), strtoul(3)

 Manual page atoi(3) line 1
```

Which pretty-well tells you everything you need to know about this function and how to use it and variants of it. Most importantly it tells you which **header file** to include.

3.3 A short library function reference

Full details of all the functions available can be found at:
http://www.gnu.org/software/libc/manual/

```
Common Library Functions by Header File:

        math.h
                Trigonometric Functions e.g.:
                        cos sin tan
                Exponential, Logarithmic, and Power Functions e.g.:
                        exp log pow sqrt
                Other Math Functions e.g.:
                        ceil fabs floor  fmod

        stdio.h
                File Functions e.g.:
                        fclose  feof  fgetpos  fopen fread fseek
                Formatted I/O Functions e.g.:
                        printf scanf Functions
                Character I/O Functions e.g.:
                        fgetc fgets  fputc fputs getc getchar gets
                        putc putchar puts

        stdlib.h
                String Functions e.g.:
                        atof atoi atol
                Memory Functions e.g.:
                        calloc free malloc
                Environment Functions e.g.:
                        abort exit  getenv system
                Math Functions  e.g.:
                        abs div rand

        string.h
                String Functions e.g.:
                        strcat strchr strcmp strncmp  strcpy
                        strncpy strcspn strlen strstr strtok

        time.h
                Time Functions e.g.:
                        asctime clock difftime time
```

There is no point in learning about library functions until you find you need to do something which then leads you to look for a function or a library of functions that has been written for this purpose. You will need to understand the function signature – or what the argument list means and how to use it and what will be returned by the function or done to variables passed as pointers to functions.

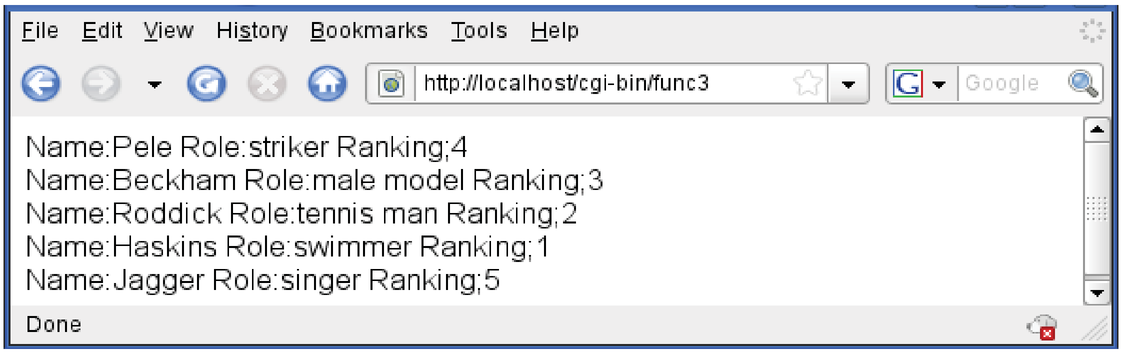
3.4 Data Structures

Sometimes we wish to manage a set of variable as a group, perhaps taking all the values from a database record and passing the whole record around our program to process it. To do this we can group data into structures.

This program uses a struct to define a set of properties for something called a player. The main function contains a declaration and instantiation of an array of 5 players. We pass a pointer to each array member in turn to a function to rank each one. This uses a switch statement to examine the first letter of each player name to make an arbitrary ranking. Then we pass a pointer to each array member in turn to a function that prints out the details.

```
/****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter3_3.c                                      *
****************************************************************/
#include <stdio.h>
#include <string.h>
struct player
{
                char name[255];
                char role[255];
                int ranking;
};
void rank(struct player *p)
{
        switch(p->name[0])
        {
                case 'P': p->ranking = 4;break;
                case 'H': p->ranking = 1;break;
                case 'R': p->ranking = 2;break;
                case 'J': p->ranking = 5;break;
                case 'B': p->ranking = 3;break;
        }
}
void show(struct player p)
{
        printf("Name:%s Role:%s Ranking;%d<br>\n",
                p.name,p.role,p.ranking);
}
int main(int argc, char *argv[], char *env[])
{
        struct player myteam[5] = { { "Pele","striker",0 },

                        { "Beckham","male model",0 },

                        { "Roddick","tennis man",0 },

                        { "Haskins","swimmer",0 },

                        { "Jagger","singer",0 } };
        int i=0;
        printf("Content-type:text/html\n\n");
        for(i=0;i<5;i++)  rank ( &myteam[i] );
        for(i=0;i<5;i++) show  ( myteam[i] );
        return 0;
}
```

The results are shown here, as usual in a browser:


File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/func3
Google
Done

```
Name:Pele Role:striker Ranking;4
Name:Beckham Role:male model Ranking;3
Name:Roddick Role:tennis man Ranking;2
Name:Haskins Role:swimmer Ranking;1
Name:Jagger Role:singer Ranking;5
```

This is a very powerful technique that is quite advanced but you will need to be aware of it. The idea of **structure**s leads directly to the idea of **classes** and **objects**.

We can see that using a struct greatly simplifies the business task of passing the data elements around the program to have different work done. If we make a change to the definition of the struct it will still work and we simply have to add code to handle new properties rather than having to change the argument lists or signatures of the functions doing the work.

The definition of the structure does not actually create any data, but just sets out the formal shape of what we can instantiate. In the main function we can express this instantiation in the form shown creating a list of sequences of data elements that conform to the definition we have made.

You can probably see that a struct with additional functions or methods is essentially what a class is in Java, and this is also the case in C++. **Object Oriented languages** start here and in fact many early systems described as "object oriented" were in fact just built using C language structs.

If you take a look for example, at the Apache server development header files you will see a lot of structs for example in this fragment of httpd.h:

```
struct server_addr_rec {
   /** The next server in the list */
   server_addr_rec *next;
   /** The bound address, for this server */
   apr_sockaddr_t *host_addr;
   /** The bound port, for this server */
   apr_port_t host_port;
   /** The name given in "<VirtualHost>" */
   char *virthost;
};
```

Dont worry about what this all means – just notice that this is a very common and very powerful technique, and the design of data structures, just like the design of database tables to which it is closely related are the core, key, vital task for you to understand as a programmer.

You make the philosophical decisions that the **world is like this** and can be modelled in this way. A heavy responsibility – in philosophy this work is called **ontology** (what exists?) and **epistemology** (how we can know about it?). I bet you never thought that this was what you were doing!

3.5 Functions, pointers and structures – conclusion

We have used some simple data types to represent some information and transmit input to a program and to organise and display some visual output.

We have used HTML embedded in output strings to make output visible in a web browser.

We have learned about creating libraries of functions for reuse.

We have learning about data structures and the use of pointers to pass them around a program.

Exercise:

Using C library functions create a program that:

- opens a file in write mode,
- writes a command line argument to the file
- closes the file
- opens the file in read mode
- reads the contents
- closes the file
- prints this to the screen

This will give you experience with finding things out, looking for suitable library functions, and finding examples on-line or from a book.

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:

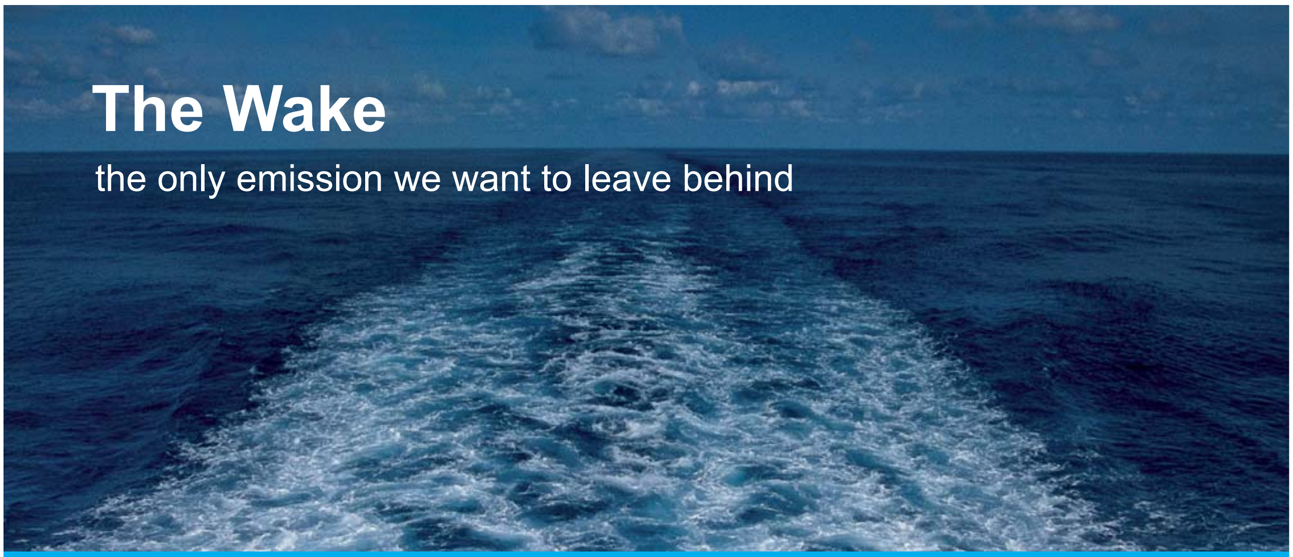
```
for( initial value; keep on until ; incremental change )
        { do this; and this; and this; }
```

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 )
```



Low-speed Engines Medium-speed Engines Turbochargers Propellers **Propulsion Packages** PrimeServ

The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo. Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front!
Find out more at www.mandieselturbo.com

Engineering the Future – since 1758.
MAN Diesel & Turbo

Click on the ad to read more

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.

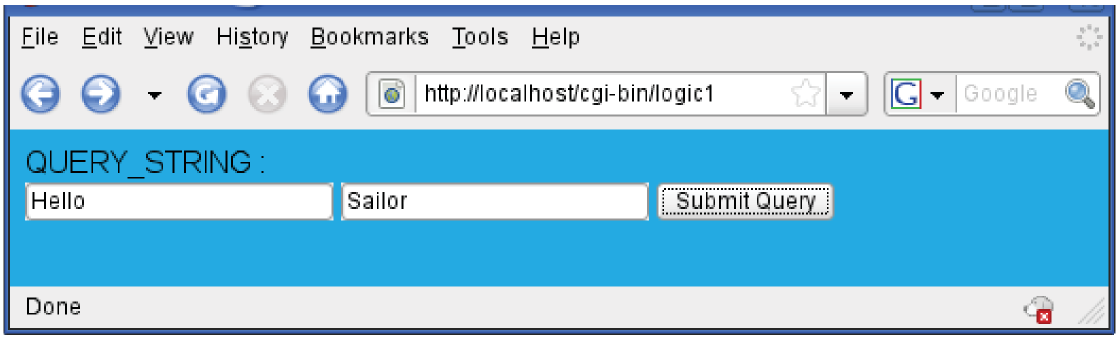
```
/*********************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter4_1.c                              *
*********************************************************************/
#include <stdio.h>
#include <string.h>

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:

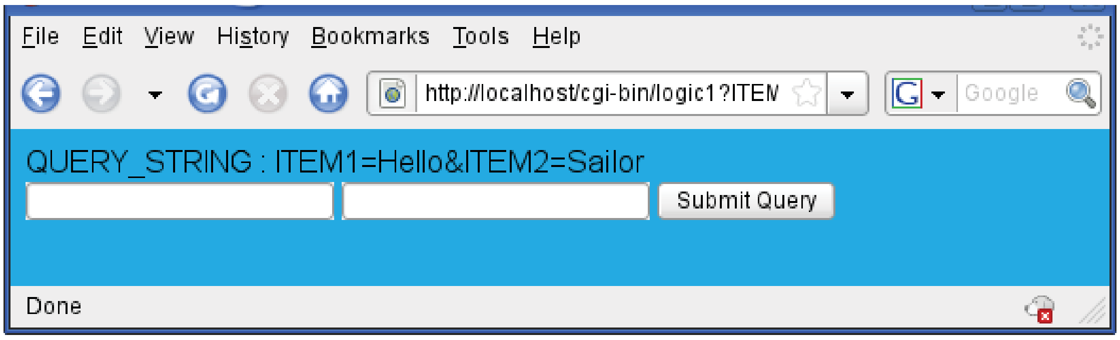

File Edit View History Bookmarks Tools Help
http://localhost/cgi-bin/logic1
Google
Submit Query
Done

QUERY_STRING :

Hello

Sailor

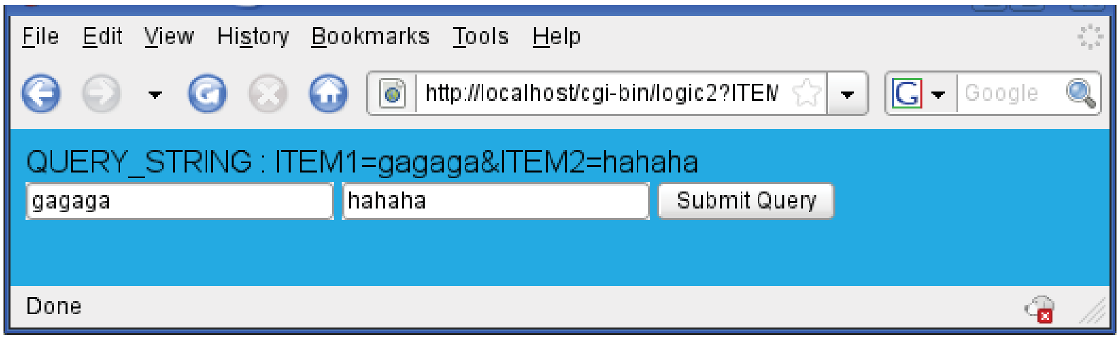
And after submitting the form we see:


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.

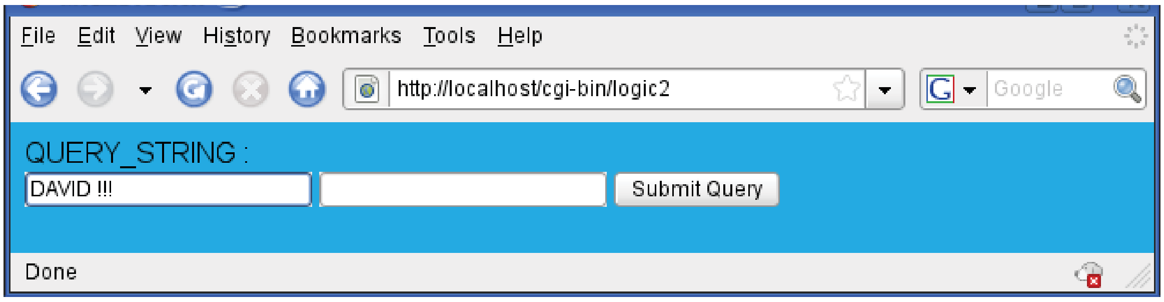
In the next example we add this function and the noticeable difference in the output is that we can insert the extracted values into the HTML boxes after we have parsed them. We seem to have successfully created something like a java getParameter() function – or have we?


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

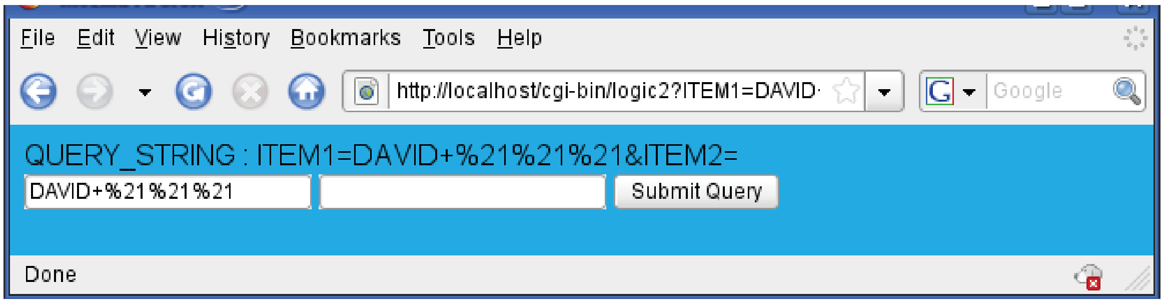
Have a good long look at chapter4_2.c and try it out with characters other than A-Z a-z or numerals and you will see something is not quite right. There is some kind of encoding going on here!

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.

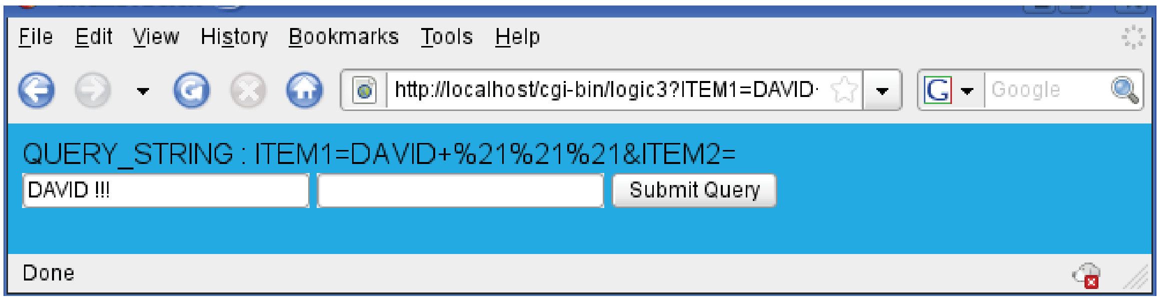
Try to describe the process in **pseudocode** of decoding this QUERY STRING:

```
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;



http://localhost/cgi-bin/logic3?ITEM1=DAVID·

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.