# C Programming in Linux

David Haskins

# C Programming in Linux

C Programming in Linux

ISBN 978-87-7681-472-4

# Contents

# About the author, David Haskins

I was born in 1950 in Chelsea, London, but grew up in New Zealand returning to England in 1966. I have worked in the computer industry since 1975 after a couple of years as a professional drummer.

My first experience was five years as a mainframe hardware engineer for Sperry Univac (now Unisys) followed by 14 years as an analyst programmer with British Telecom in London.

While engaged in a complex task of converting large quantities of geographical data (map coordinate references) I discovered the joys of C – its speed and efficiency. That was in 1985 and I have been a fan of C ever since.

Since 1994 I have been a senior lecturer at the Faculty of Computing, Information Systems and Mathematics at Kingston University, London. This is a mostly technical university that evolved from a former polytechnic college with a long tradition of aeronautical engineering.

I am engaged mainly in teaching many computer languages and internet systems design to a large and multicultural student body.

Most of my academic research and commercial consultancy has been involved with spatial systems design and the large data volumes and necessary processing efficiency concerns has led me to concentrate on C and C++. My teaching web site is at **www.ubiubi.org** which shows some of this material.

A keen Open Systems enthusiast, I have exclusively centred all my teaching on the Linux platform since 2002 and Kingston University is well advanced in delivering dual boot facilities for all its student labs.

I am a keen swimmer and in 2009 completed the annual Lorne *Pier-to-Pub* race in Victoria, Australia which is the largest open-sea swimming race in the world where 4,500 people of all ages swim each January as the shark-spotting planes fly overhead.

When not teaching I am a keen vegetable gardener and amateur musician, playing in jazz groups and in Scottish bagpipe bands. I play the drums, the great highland bagpipe, the clarinet, the guitar and the piano.

# Introduction

## Why learn the C language?

Because the C language is like Latin - it is finite and has not changed for years. C is tight and spare, and in the current economic climate we will need a host of young people who know C to keep existing critical systems running.

C is built right into the core of Linux and Unix. The design idea behind Unix was to write an operating system in C so all you needed to port it to a new architecture was a C compiler. Linux is essentially the success story of a series of earlier attempts to make a PC version of Unix.

A knowledge of C is now and has been for years a pre-requisite for serious software professionals and with the recent popularity and maturity of Open Systems this is even more true. The terseness and perceived difficulty of C saw it being ousted from university teaching during the late 1990s in favour of Java but there is a growing feeling amongst some teaching communities that Java really is not such a good place to start beginners.

Students paradoxically arrive at colleges knowing less about computing than they did ten years ago as programming is seen as too difficult for schools to teach. Meanwhile the body of knowledge expected of a competent IT professional inexorably doubles every few years.

Java is commonly taught as a first language but can cause student confusion as it is in constant flux, is very abstract and powerful, and has become too big with too many different ways to do the same thing. It also is a bit “safe” and insulates students from scary experiences, like driving with air-bags and listening to headphones so you take less care. The core activity of writing procedural code within methods seems impenetrable to those who start from classes and objects.

So where do we start? A sensible place is “at the beginning” and C is as close as most of us will ever need to go unless we are becoming hardware designers. Even for these students to start at C and go further down into the machine is a good idea.

C is like having a very sharp knife which can be dangerous, but if you were learning to be a chef you would need one and probably cut yourself discovering what it can do. Similarly C expects you to know what you are doing, and if you don't it will not warn before it crashes.

A knowledge of C will give you deep knowledge of what is going on beneath the surface of higher-level languages like Java. The syntax of C pretty-well guarantees you will easily understand other languages that came afterwards like C++, Java, Javascript, and C#.

C gives you access to the heart of the machine and all its resources at a fine-grained bit-level.

C has been described as like “driving a Porsche with no brakes” - and because it is fast as well this can be exhilarating. C is is often the only option when speed and efficiency is crucial.

C has been called “dangerous” in that it allows low-level access to the machine but this scariness is exactly what you need to understand as it gives you respect for the higher-level languages you will use.

Many embedded miniaturised systems are all still written in C and the machine-to-machine world of the invisible internet for monitoring and process control runs often uses C.

Hopefully this list of reasons will start you thinking that it might be a good reason to have a go at this course.

## References

The C Programming Language – Second Edition - Kernighan and Richie
ISBN 0-13-11-362-8

The GNU C Library Free Software Foundation C Manual
http://www.gnu.org/software/libc/manual/

MySQL C library
http://dev.mysql.com/doc/refman/5.1/en/index.html

The GD C library for graphics
http://www.libgd.org/Documentation

APXS - the APache eXtenSion tool
http://httpd.apache.org/docs/2.0/programs/apxs.html

Apache
http://httpd.apache.org/docs/2.2/developer/

“The Apache Modules Book” Nick Kew, Prentice Hall
ISBN 0-13-240967-4

A **Source Code Zip File Bundle** is supplied with this course which contains all the material described and a Makefile.

## The teaching approach

I began university teaching later in life after a career programming in the telecommunications industry.

My concern has been to convey the sheer fun and creativity involved in getting computers to do what you want them to do and always try to give useful, practical, working examples of the kinds of things students commonly tell me they want to do.

Learning a language can be a dry, boring affair unless results are immediate and visible so I tend to use the internet as the input-output channel right from the start.

I prefer teaching an approach to programming which is deliberately “simple” using old-fashioned command-line tools and editors and stable, relatively unchanging components that are already built-in to Unix and Linux distributions such as Suse, Ubuntu and Red Hat.

This is in response to the growing complexity of modern Integrated Development Environments (IDEs) such as Developer Studio, Netbeans and Eclipse which give students an illusion that they know what they are doing but generate obfuscation.

My aim is to get students confident and up to speed quickly without all the nightmare associated with configuring complex tool chains.  It is also essentially a license-free approach and runs on anything.

With this fundamental understanding about what is really going on you can progress on to use and actually understand whatever tools you need in your career.

In order to give a sense of doing something real and useful and up to date, the focus is on developing visible and effectively professional-quality web-server and client projects to put on-line, using:

Apache Web server and development libraries.
C language CGI programs (C programming using the “make” utility).
C language Apache modules.
MySQL server with C client library interfaces.
GD graphics library with C interfaces.
Incidental use of CSS, (X)HTML, XML, JavaScript, Ajax.

This course has been designed for and lab-tested by first and second year Computer Science Students at Kingston University, London UK.

# Setting up your System

This book presumes you are using the Linux operating system with either the KDE3.5, KDE4, or Gnome desktop. Specific instructions are included for Ubuntu (and Kubuntu) and OpenSuse 11.

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 libmysqlclient15-dev | libmysqlclient-devel |
| GD and development libraries | libgd2-xpm | gd<br>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/default-server.conf**.

# 1. Chapter One: 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.
l**ist, 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[].

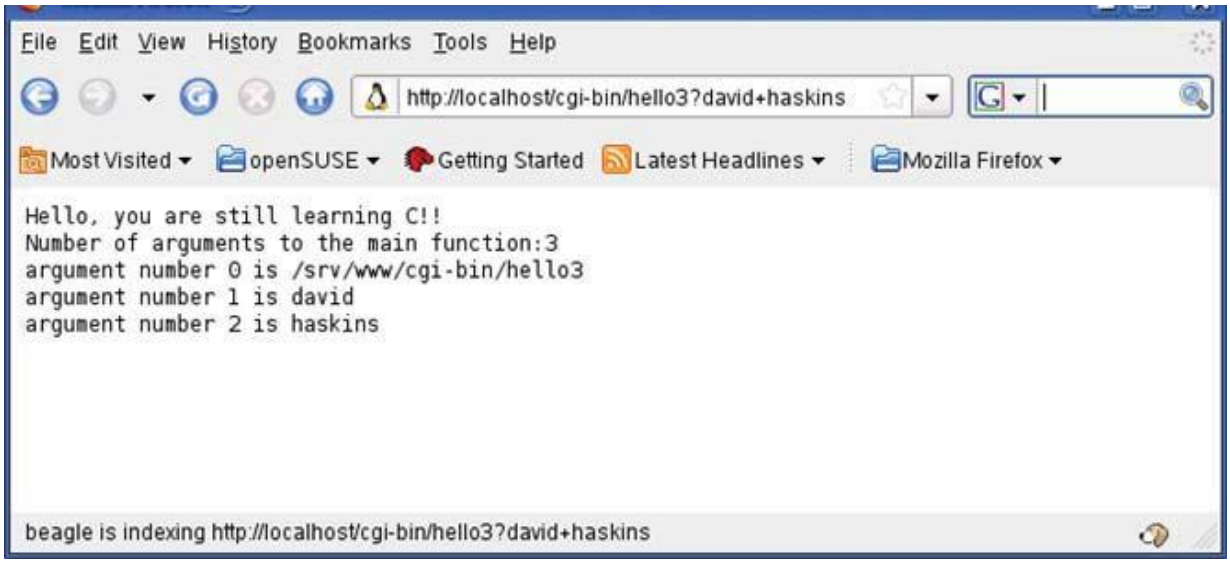
## 1.3 Hello Program 3

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 generated by a C program.



## 1.4 Hello Program 4

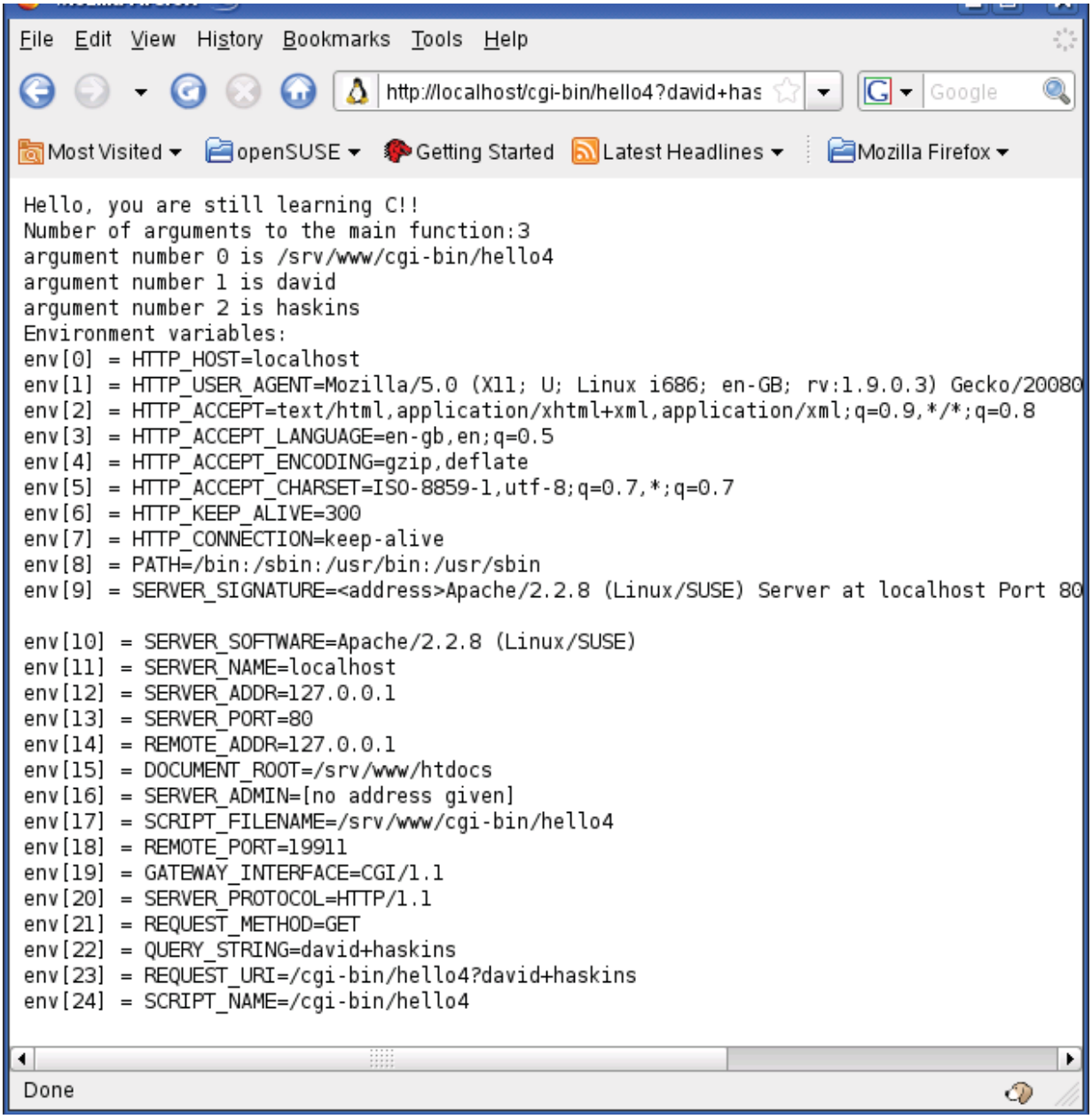
A seldom documented feature of the function signature for “main” is that it can take **three** arguments and the last one we will now look at is char *env[ ] which is also a list of pointers to strings, but in this case these are the **system environment variables** available to the program at the time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




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 the 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 x 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*

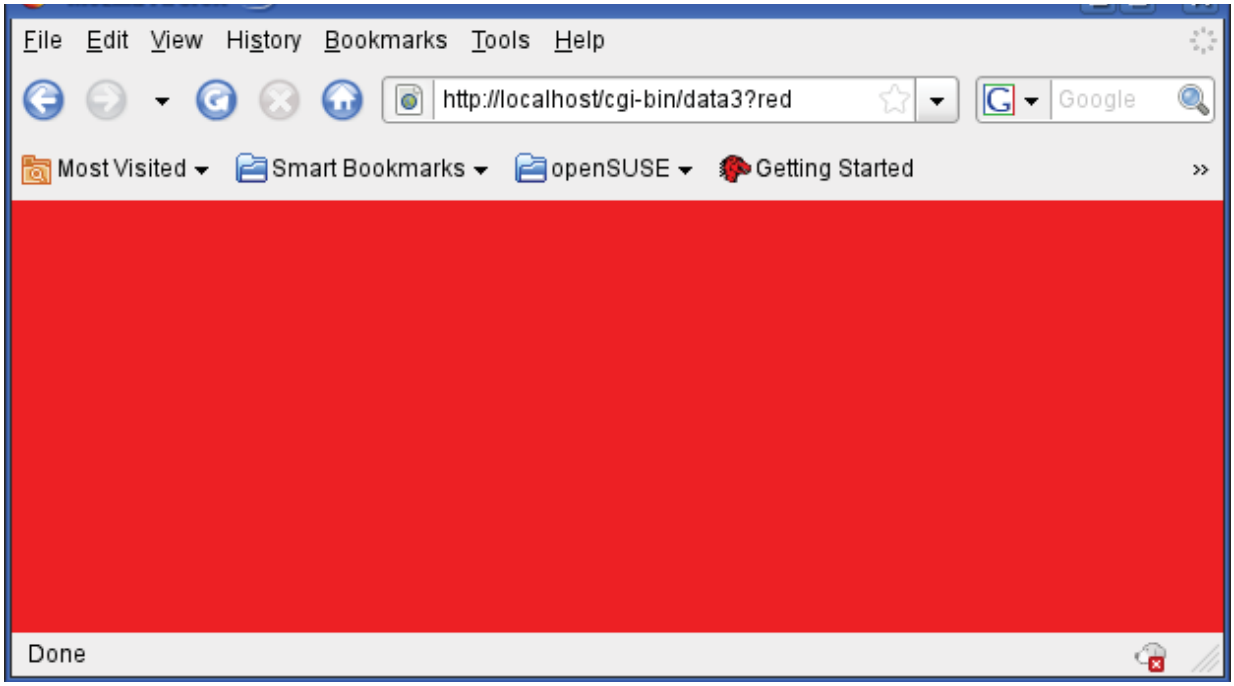
## 2.3 What can a string “mean”

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

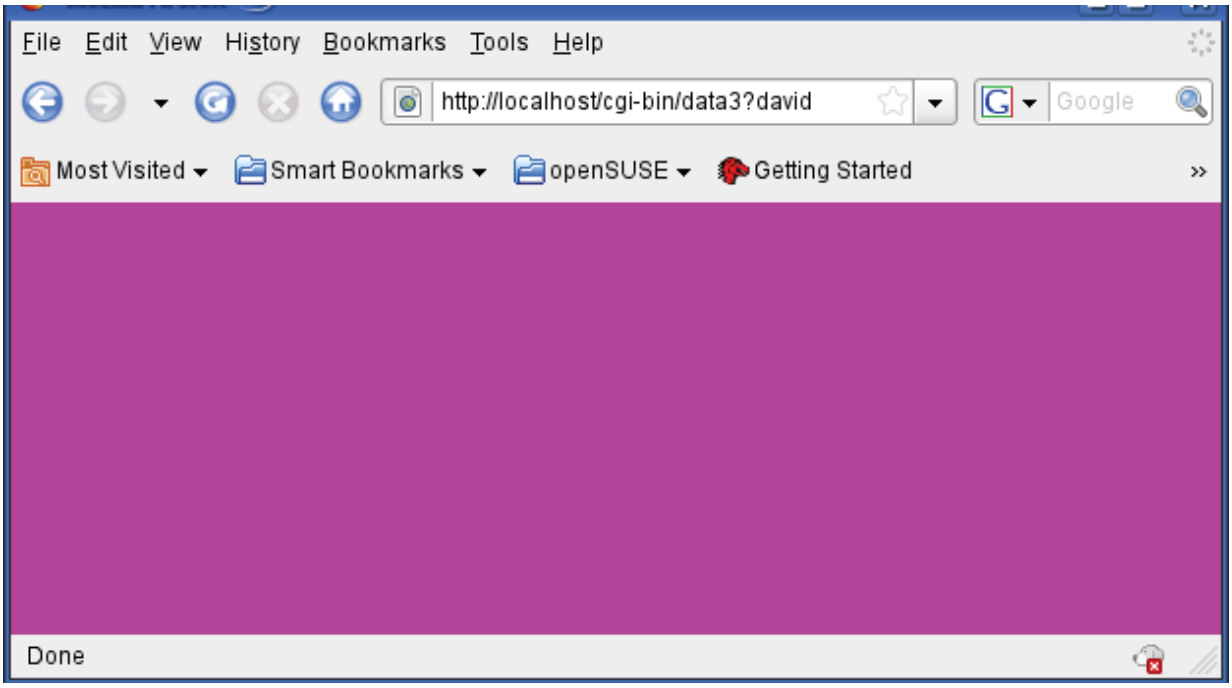
```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_3.c                                   *
*****************************************************************/
#include <stdio.h>
#include <string.h>

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:



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




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.

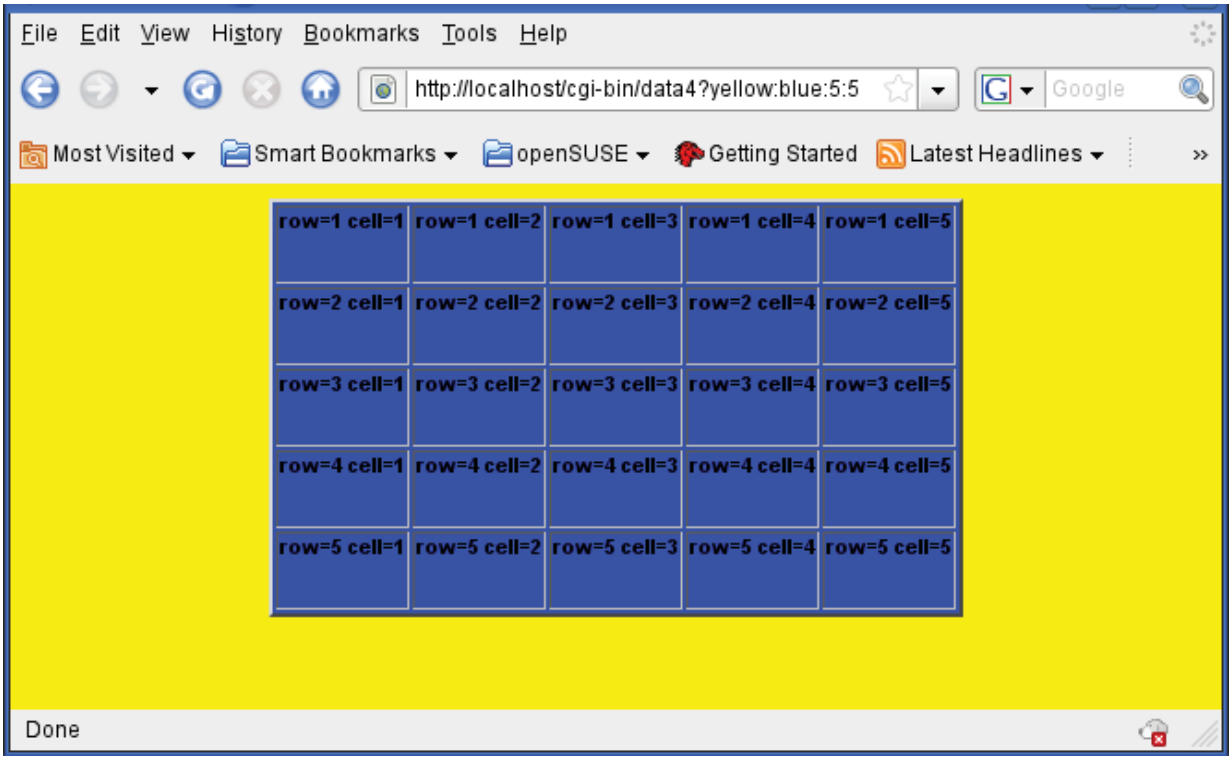
## 2.4 Parsing a string

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:




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:

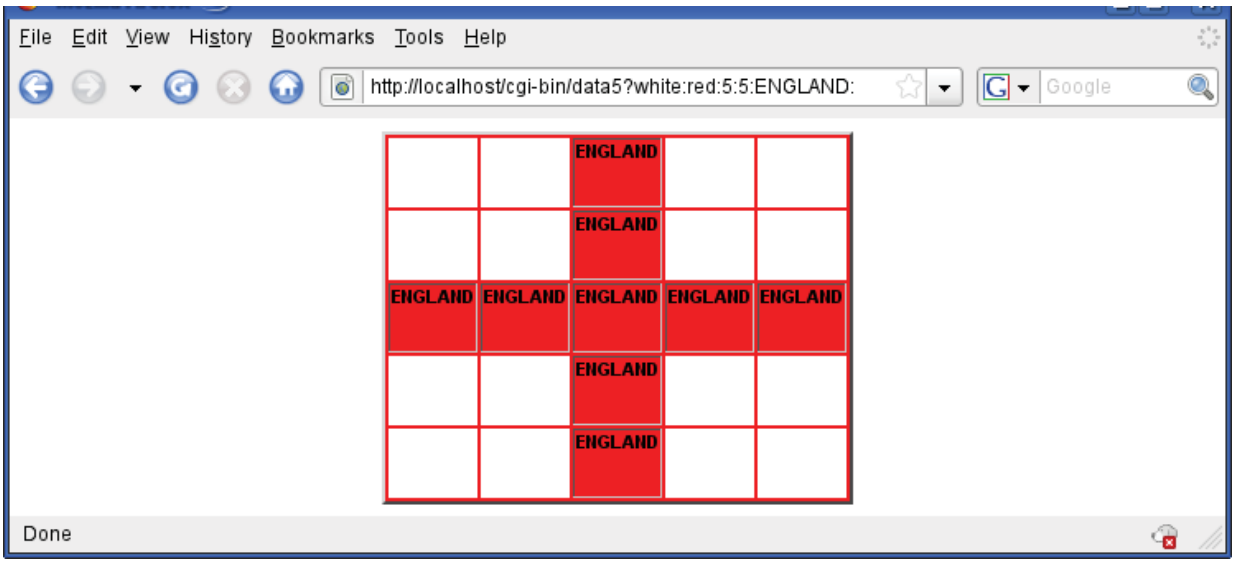
**<html>** **<body>** **<table>** **<tr>** table row **<td>** table data cell

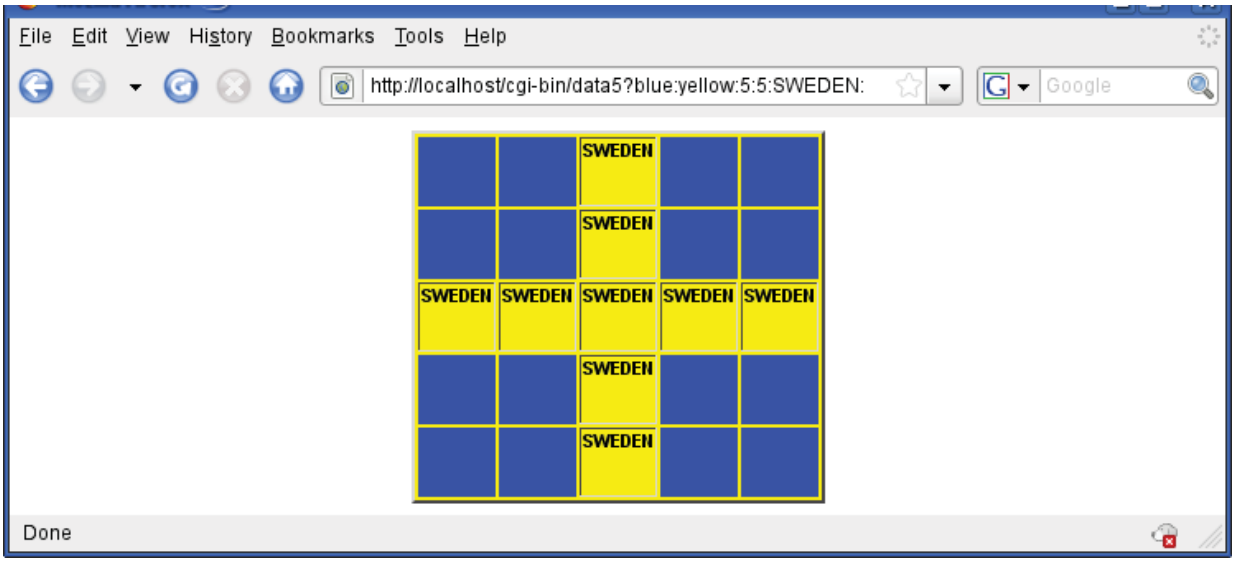
## 2.5 Data and Memory – conclusion

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:



# 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 l**ibrary 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.

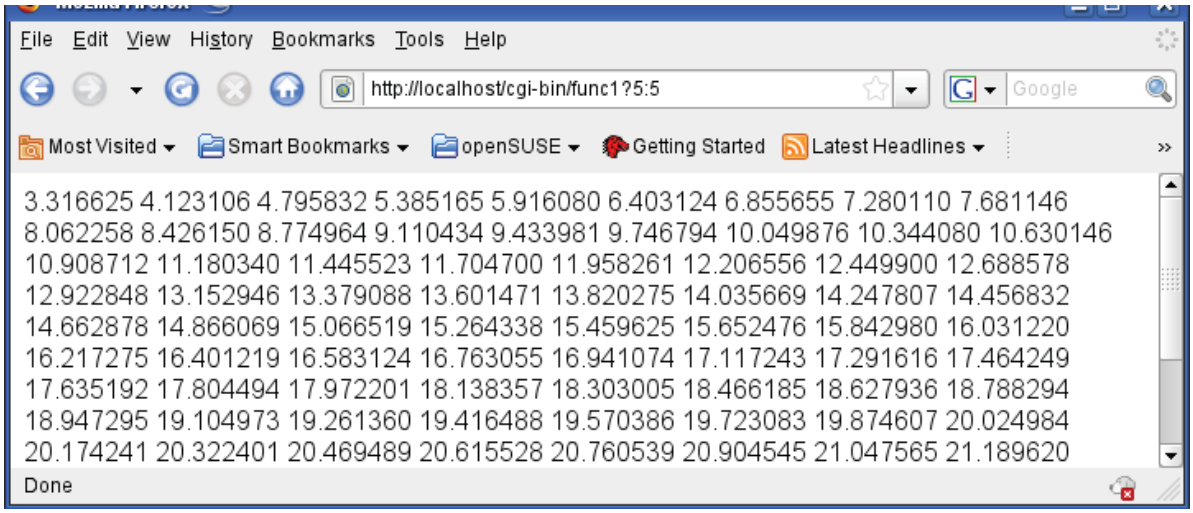
As we learned before, the **scope** of data is restricted to the **function** in which is was declared, so we use **pointers** to data and blocks of data to pass to functions that we wish to do some work on our data. We have seen already that strings are handled as pointers to arrays of single characters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

(by now you should be maintaining a Makefile as you progress, adding targets to compile examples as you go.)




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:


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.

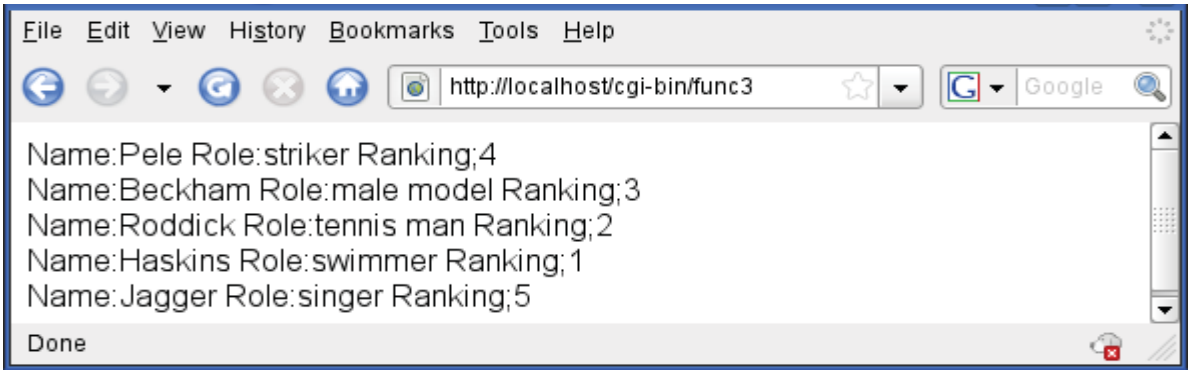
## 3.4 Data Structures

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:




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 )
```

When have a  single  option to test:

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

When 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 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.

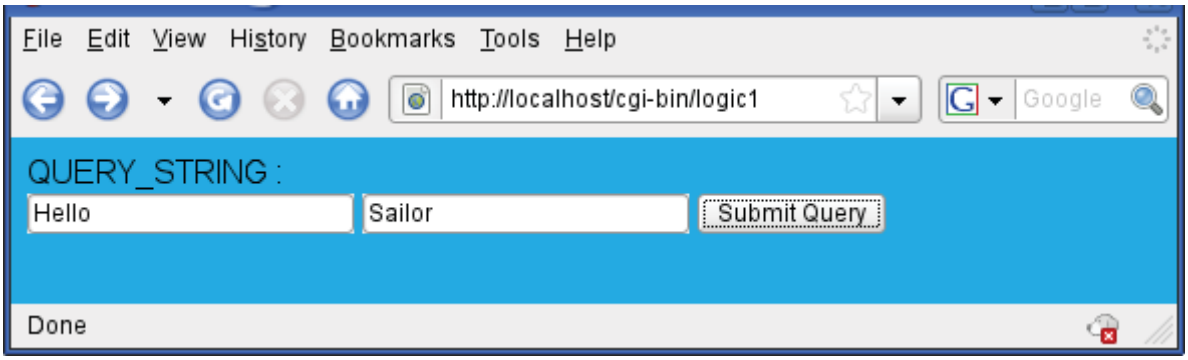
```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter4_1.c                                  *
*****************************************************************/
#include <stdio.h>
#include <string.h>

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.

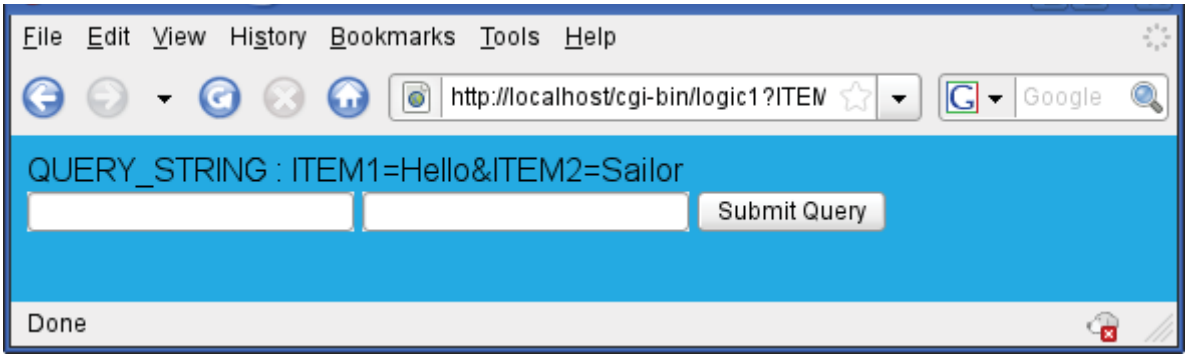
An HTML form by default uses the **GET** method which transmits all form data back to the program or page that contains the form unless otherwise specified in an action attribute. This data is contained in the QUERY_STRING as a series of variable = value pairs separated by the & character.

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:

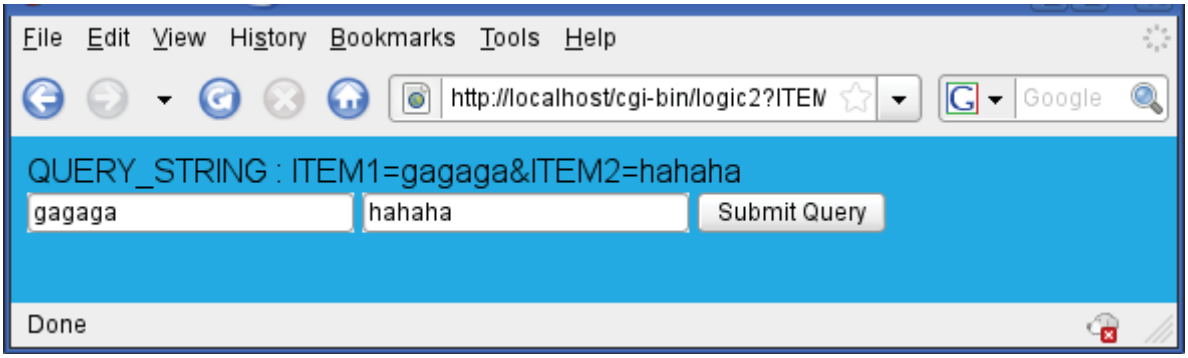


And after submitting the form we see:



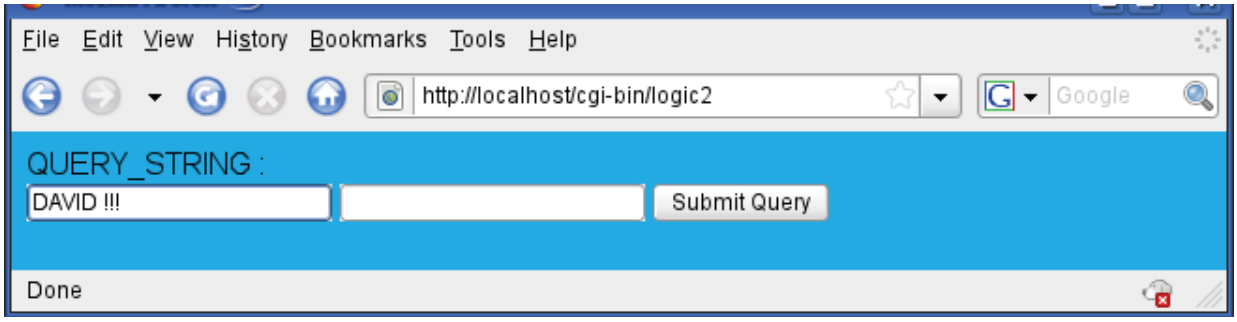

To make much sense of the QUERY_STRING and find a particular value in it, we are going to have to **parse** it, to chop it up into its constituent pieces and for this we will need some **conditional logic** (if, else etc) and some **loop** to count through the characters in the variable. A basic function 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?

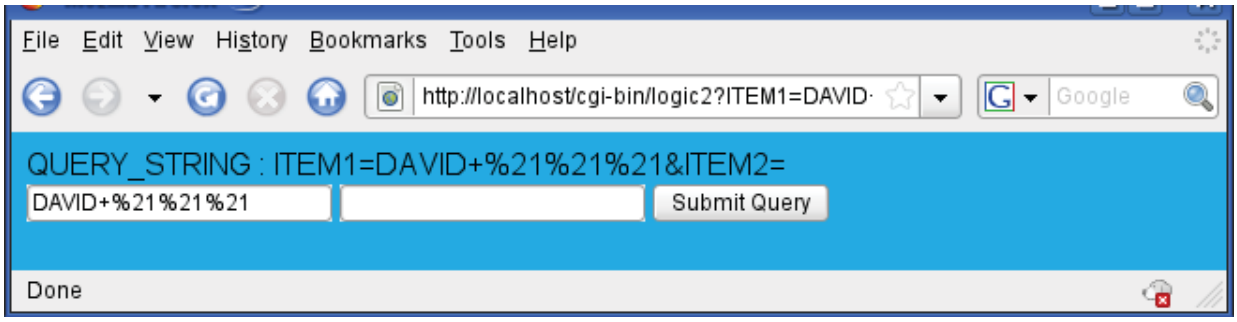



Have a good long look at chapter4_2.c and try it out with characters other than A-Z a-z or numerals and you will see something is not quite right. There is some kind of encoding going on here!

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




I get this result:




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.

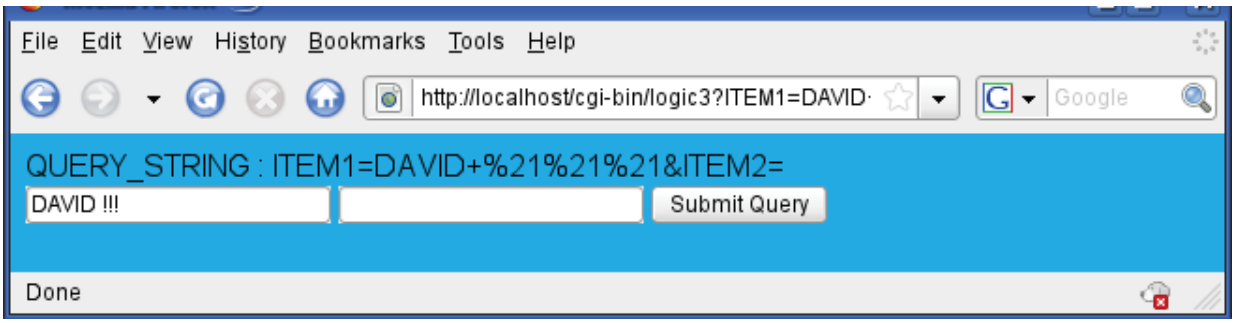
Try to describe the process in **pseudocode** of decoding this QUERY STRING:

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;



```
/*****************************************************************
* 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.

# 5. Database handling with MySQL

## 5.1 On not reinventing the wheel

It is pretty sensible to not start from scratch at every project so we build on work done by others who came this way. All of the C libraries we are using were written by someone from even lower level bits and pieces and we can access and modify this code should we want to. This is what Open Systems is all about.
In practice we simply want to use reliable services, and the first two we are going to use are ubiquitous – databases access and graphical image generation.

## 5.2 MySQL C API

To access the set of MySQL functions we make sure our compiler can access the header and libraries of MySQL and simply call some mysql functions we have not had to write ourselves.
In this next example we join the database code onto the last work we did and insert the decoded data into a database, and read out what we have stored there so far.
In a new terminal window we type:

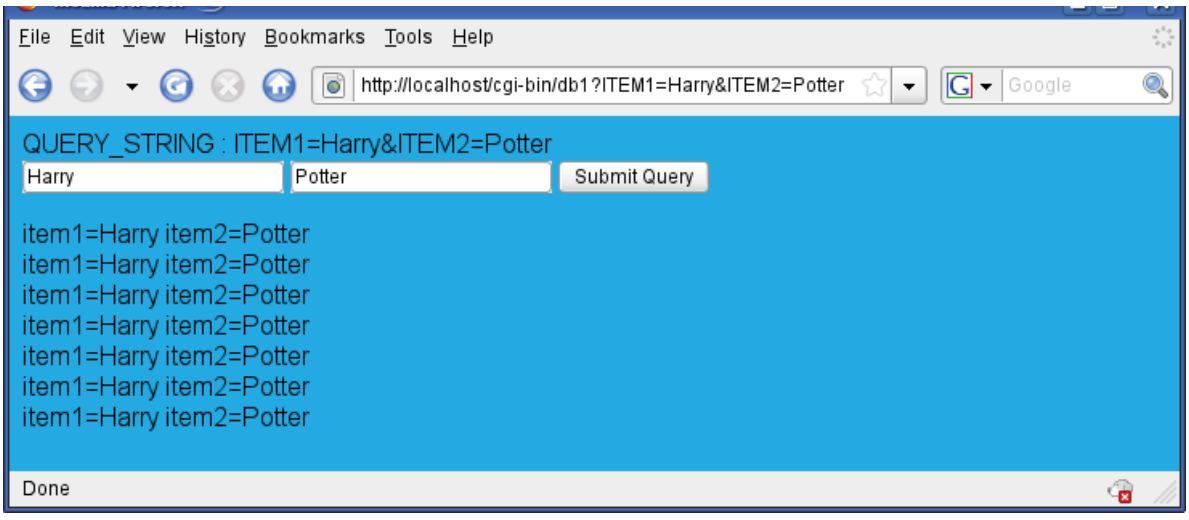
**mysql test**

then:

**create table CIL (ITEM1 varchar(255),ITEM2 varchar(255));**

This creates a table for the program to write to in the test database that should already be created in your MySQL setup.

It looks like this if it all works out OK.



The data types declared **MYSQL**, **MYSQL_RES**, and **MYSQL_ROW** are **preprocessor definitions** that stand for more complex C declarations in mysql.h and all we need to know is how to call them.

The documentation at MySQL provides all the information you will need to do more complex operations.

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter5_1.c                                                   *
*****************************************************************/
#include <stdio.h>
#include <string.h>
#include <mysql/mysql.h>
#include "c_in_linux.h"

int main(int argc, char *argv[], char *env[])
{
        char value1[255] = "",value2[255] = "",SQL[1024]="";
        int rc = 0;

        MYSQL *conn = NULL;
        MYSQL_RES *result = NULL;
        MYSQL_ROW row;

        printf("Content-type:text/html\n\n<html><body bgcolor=#23abe2>\n");
        strncpy(value1,(char *) getenv("QUERY_STRING"),255);
        printf("QUERY_STRING : %s<BR>\n", value1 );
        printf("<form>\n");
        //call the decode_value function to get value of "ITEM1"
        decode_value( "ITEM1=", (char *) &value1, 255);
        if(strlen(value1) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM1\" value=\"%s\">\n",value1);
        else
                printf("<input type=\"TEXT\" name=\"ITEM1\">\n");
        //call the decode_value function to get value of "ITEM2"
        decode_value( "ITEM2=", (char *) &value2, 255);
        if(strlen(value2) > 0 )
                printf("<input type=\"TEXT\" name=\"ITEM2\"    value=\"%s\">\n",value2);
        else
                printf("<input type=\"TEXT\" name=\"ITEM2\">\n");
        printf("<input type=\"SUBMIT\">");
        printf("</form></body></html>\n");
        //OPEN DATABASE
        conn = mysql_init((MYSQL *) 0);
        mysql_options(conn,MYSQL_READ_DEFAULT_GROUP,"mysqlcapi");
        mysql_real_connect(conn, "localhost","","","test",0, NULL, 0);
        //INSERT IF THERE IS ANY DATA
        if(strlen(value1) > 0 || strlen(value2) > 0)
        {
        sprintf(SQL,"insert into CIL values ('%s','%s')",value1,value2);
                rc = mysql_query(conn,SQL);
        }
        //READ
        rc = mysql_query(conn,"select * from CIL");
        result = mysql_use_result(conn);
        while( (row = mysql_fetch_row(result)) != NULL)
        {
                        printf("item1=%s   item2=%s<br>",row[0],row[1]);
        }
        mysql_free_result(result);
        mysql_close(conn);
 return 0;  }
```

Understood all that?

In barely 10 lines (highlighted) of C we have inserted our data into a database table and can read it out again, which is pretty painless. With **mysql_init** we obtain a pointer to a data structure of type MYSQL, we can then se this to connect to the test database with **mysql_options** and **mysql_real_connect,** then we execute SQL statements just as we would in a terminal session. The results of a query can be retrieved as a sequence of MYSQL_ROW arrays of strings with **mysql_use_results**. We free up the memory used with **mysql_free_result** and close the database with **mysql_close**.

As is usual with C libraries, you need to be able understand the usually sparse documentation to understand the function calls, and for MySQL 5.1 we can find all this information at: http://dev.mysql.com/doc/refman/5.1/en/index.html

**The MySQL 5.1 Reference Manual / Connectors and APIs / MySQL C API includes:**

| | |
|---|---|
| mysql_affected_rows() | mysql_insert_id() |
| mysql_autocommit() | mysql_kill() |
| mysql_change_user() | mysql_library_end() |
| mysql_character_set_name() | mysql_library_init()mysql_list_dbs() |
| mysql_close() | mysql_list_fields( |
| mysql_commit() | mysql_list_processes() |
| mysql_connect() | mysql_list_tables() |
| mysql_create_db() | mysql_more_results() |
| mysql_data_seek() | mysql_next_result() |
| mysql_debug() | mysql_num_fields() |
| mysql_drop_db() | mysql_num_rows() |
| mysql_dump_debug_info() | mysql_options() |
| mysql_eof() | mysql_ping() |
| mysql_errno() | mysql_query() |
| mysql_error() | mysql_real_connect() |
| mysql_escape_string() | mysql_real_escape_string() |
| mysql_fetch_field() | mysql_real_query() |
| mysql_fetch_field_direct() | mysql_refresh( |
| mysql_fetch_fields() | mysql_reload() |
| mysql_fetch_lengths() | mysql_rollback() |
| mysql_fetch_row() | mysql_row_seek() |
| mysql_field_count() | mysql_row_tell() |
| mysql_field_seek() | mysql_select_db() |
| mysql_field_tell() | mysql_set_character_set() |
| mysql_free_result() | mysql_set_local_infile_default() |
| mysql_get_character_set_info() | mysql_set_local_infile_handler() |
| mysql_get_client_info() | mysql_set_server_option() |
| mysql_get_client_version() | mysql_shutdown() |
| mysql_get_host_info() | mysql_sqlstate() |
| mysql_get_proto_info() | mysql_ssl_set() |
| mysql_get_server_info() | mysql_stat() |
| mysql_get_server_version() | mysql_store_result() |
| mysql_get_ssl_cipher( | mysql_thread_id() |
| mysql_hex_string() | mysql_use_result() |
| mysql_info() | mysql_warning_count() |
| mysql_init() | |

# 6. Graphics with GD library

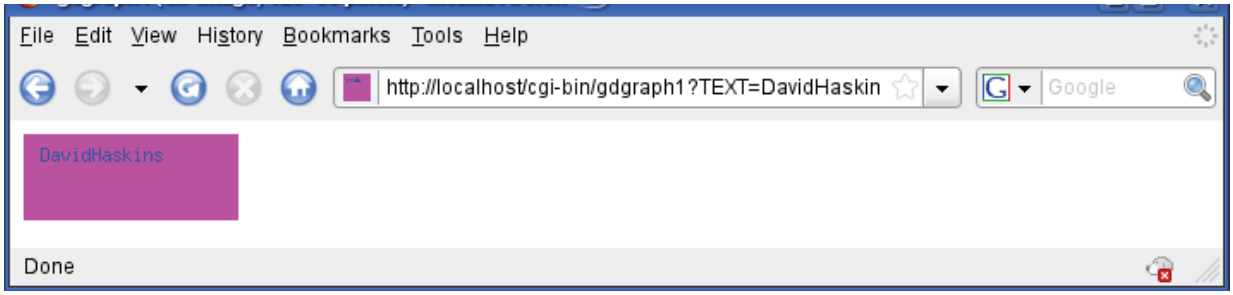
The ability to generate interesting images dynamically to suit the situation is an enjoyable, challenging, and rewarding type of programming. We will be using Thomas Boutell's GD C library which has been around for many years as an open systems project.

## 6.1 Generating binary content

Here we create an image and print into it the value of the TEXT= part of the QUERY STRING and set our content type to image/gif. The **gdImageGif** function writes the binary image out to **stdout** which is the output stream instead of to a file, so binary image data is sent back to the browser.

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter6_1.c                                   *
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gd.h>
#include <gdfonts.h>
#include "c_in_linux.h"

int main(int argc, char *argv[], char *env[])
{
        int text=0,background=0,height=50,width=0;
        char value[255] = "";
        gdImagePtr im_out = NULL;

        decode_value("TEXT=", (char *) &value, 255);
        width = ( strlen(value) * 10 ) + 5;
        im_out = gdImageCreate(width,height);
        background = gdImageColorAllocate(im_out, 255,0,255);
        text  = gdImageColorAllocate(im_out,0,0,255);
        gdImageFilledRectangle(im_out, 0,0, width-1, height-1,  background);
        gdImageString(im_out,gdFontGetSmall(),10,5,(unsigned char *)value,        text);
        printf("Content-type: image/gif\n\n");
        gdImageGif(im_out,stdout);
        gdImageDestroy(im_out);
        return 0;
}
```

The program produces this output when called with ***cgi-bin/gdgraph1?TEXT=DavidHaskins***




While this is not the most fancy or attractive exercise but does at least demonstrate the key principles involved in generating graphical output. With the GD library you can load existing images and generate them in many formats such as **GIF**, **JPEG**, **PNG**, **WMBP** and even create **animated GIFS**.

If you have used PHP development tools you will recognise the GD library functions as they are pretty well identical showing that PHP is a wrapper in this case around the same C library.

The text displayed here is using one of the internal fonts, Tiny, Small, Medium Bold, Large and Giant which are adequate for simple labelling but you can also use TrueType fonts for more attractive output.

Geometric drawing with lines or certain styles, filled and open polygons, and circles, arcs can be created. The main thing to remember is that the origin of an image is the **TOP left hand corner** which might seem unintuitive to anyone who has studies mathematics – quite why this is I have never discovered but can only imagine that the first programmer to do anything in this area happened to not know about graphs in which we think of the origin x=0, y=0 as being at the bottom left hand.

Colours are specified as RGB values in the range 0 to 255 so that white is 255,255,255 and red is 255,0,0. To work out fine-grained hues use a graphics tool like GNU GIMP which has colour pickers so you can find out subtle RGB values. Palettes of colours from one image can be used in another and the closest colour in a palette requested or created if there is space for it to be allocated.

## 6.2 Using TrueType Fonts

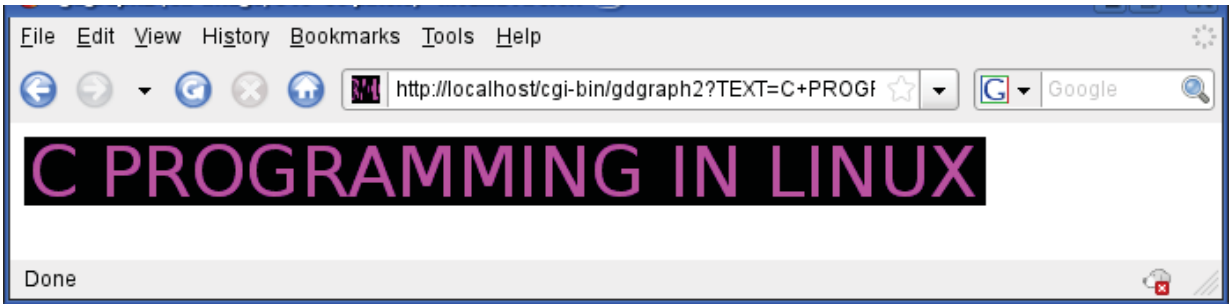
In this example we modify the previous program to generate a label using TrueType font.

```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008 * chapter6_2.c                    *
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gd.h>
#include <gdfonts.h>
#include "c_in_linux.h"

int main(int argc, char *argv[], char *env[]) {
        int text=0,background=0,height=50,width=0,x=0,y=0,size=30,string_rectangle[8];
        double angle=0.0;
        char value[255] = "";
        //OpenSuse TrueType Font
        char font[256] = "/usr/share/fonts/truetype/DejaVuSans.ttf";
        //Ubuntu TrueType Font
        // char font[256] =
        "/usr/share/fonts/truetype/ttf-dejavu/DejaVuSans.ttf";
        char *err = NULL;
        gdImagePtr im_out = NULL;

        decode_value("TEXT=", (char *) &value, 255);
        //call gdImageStringFT with NULL image to obtain size
        err = gdImageStringFT(NULL,&string_rectangle[0],0,
                font,size,angle,0,0,value);
        x = string_rectangle[2]-string_rectangle[6] + 6;
        y = string_rectangle[3]-string_rectangle[7] + 6;
        // create an image big enough for the string plus a little space
        im_out = gdImageCreate(x,y);
        //allocate colours
        background = gdImageColorAllocate(im_out, 0,0,0);
        text  = gdImageColorAllocate(im_out,255,0,255);
        //get starting position
        x = 3 - string_rectangle[6];
        y = 3 - string_rectangle[7];
        //draw the string
        err = gdImageStringFT(im_out,&string_rectangle[0],
                text,font,size,angle,x,y,value);
        //output
        printf("Content-type: image/gif\n\n");
        gdImageGif(im_out,stdout);
        gdImageDestroy(im_out);
        return 0;   }
```

We call this program as before with ***cgi-bin/gdgraph1?TEXT=C+PROGRAMMING+IN+LINUX***

to get this kind of output which you will probably see is more likely to be a useful kind of tool. The location and contents of your systems fonts will vary but the code gives an example:

- OpenSuse /usr/share/fonts/truetype/*.ttf
- Ubuntu. /usr/share/fonts/truetype/ttf-dejavu/*.ttf




To get any good at using a library like GD you have to be prepared to experiment and take a lot of time to understand the function parameters, looking in great detail at the available documentation at: http://www.libgd.org/Documentation

## 6.3 GD function reference

A full detailed set of documentation is maintained at: http://www.libgd.org

GD contains a wealth of functionality for all kinds of drawing and many formats, as well as TrueType fonts and animated Gif images. A categorised list of functions follows:

**Image creation, destruction, loading and saving:**

```
gdImageCreate(int sx,  int  sy)
gdImageCreateFromJpeg(FILE *in)
gdImageCreateFromPng(FILE *in)
gdImageCreateFromGif(FILE *in)
gdImageCreateFromGd(FILE *in)
gdImageCreateFromWBMP(FILE *in)
gdImageDestroy(gdImagePt rim)
void gdImageJpeg(gdImagePt rim, FILE*out, int quality)
void gdImageGif(gdImagePt rim, FILE*out)
void gdImagePng(gdImagePtr im, FILE*out)
void gdImageWBMP(gdImagePtr im, int fg, FILE*out)
void gdImageGd(gdImagePtr im, FILE*out)
```

**Drawing Functions:**

```
void gdImageSetPixel(gdImagePtr im, int x, int y, int color)
void gdImageLine(gdImagePtr im, int x1, int y1, int x2, int y2, int color)
void gdImageDashedLine(gdImagePtr im, int x1, int y1, int x2, int y2, int color)
void gdImagePolygon(gdImagePtr im, gdPoint Ptr point s, int point sTotal, int color)
void gdImageOpenPolygon(gdImagePtr im, gdPoint Ptr point s, int point sTotal, int
        color)
void gdImageRectangle(gdImagePtr im, int x1, int y1, int x2, int y2, int color)
void gdImageFilledPolygon(gdImagePtr im, gdPoint Ptr point s, int point sTotal, int
        color)
void gdImageFilledRectangle(gdImagePtr im, int x1, int y1, int x2, int y2, int color)
void gdImageArc(gdImagePtr im, int cx, int cy, int w, int h, int s, int e, int color)
void gdImageFilledArc(gdImagePtr im, int cx, int cy, int w, int h, int s, int e, int color,
        int style)
void gdImageFilledEllipse(gdImagePtr im, int cx, int cy, int w, int h, int color)
void gdImageFillToBorder(gdImagePtr im, int x, int y, int border, int color)
void gdImageFill(gdImagePtr im, int x, int y, int color)
void gdImageSetAntiAliased(gdImagePtr im, int c)
void gdImageSetAntiAliasedDontBlend(gdImagePtr im, int c)
void gdImageSetBrush(gdImagePtr im, gdImagePtr brush)
void gdImageSetTile(gdImagePtr im, gdImagePtr tile)
```

```
void gdImageSetStyle(gdImagePtr im, int *style, int styleLength)
void gdImageSetThickness(gdImagePtr im, int thickness)
void gdImageAlphaBlending(gdImagePtr im, int blending)
void gdImageSaveAlpha(gdImagePtr im, int saveFlag)
void gdImageSetClip(gdImagePtr im, int x1, int y1, int x2, int y2)
void gdImageGetClip(gdImagePtr im, int *x1P, int *y1P, int *x2P, int *y2P)
```

**Query Functions:**

```
int  gdImageAlpha(gdImagePtr im, int color)(MACRO)
int  gdImageGetPixel(gdImagePtr im, int x, int y)
int  gdImageBoundsSafe(gdImagePtr im, int x, int y)
int  gdImageGreen(gdImagePtr im, int color)(MACRO)
int  gdImageRed(gdImagePtr im, int color)(MACRO)
int  gdImageSX(gdImagePtr im)(MACRO)
int  gdImageSY(gdImagePtr im)(MACRO)
int  gdImageTrueColor(im)(MACRO)
```

**Text-handling functions:**

gdFontPtr gdFontGetSmall(void )
gdFontPtr gdFontGetLarge(void )
gdFontPtr gdFontGetMediumBold(void )
gdFontPtr gdFontGetGiant(void )
gdFontPtr gdFontGetTiny(void )
void gdImageChar(gdImagePtr im, gdFontPtr font, int x, int y, int c, int color)
void gdImageCharUp(gdImagePtr im, gdFontPtr font, int x, int y, int c, int color)
void gdImageString(gdImagePtr im, gdFontPtr font, int x, int y, unsigned char*s, int color)
void gdImageString16(gdImagePtr im, gdFontPtr font, int x, int y, unsigned short *s, int color)
void gdImageStringUp(gdImagePtr im, gdFontPtr font, int x, int y, unsigned char*s, int color)
void gdImageStringUp16(gdImagePtr im, gdFontPtr font, int x, int y, unsigned short*s, int color)

char *gdImageStringFT(gdImagePtr im, int *brect, int fg, char *fontname, double ptsize, double angle, int x, int y, char*string)
char *gdImageStringFTEx(gdImagePtr im, int *brect, int fg, char *fontname, double ptsize, double angle, int x, int y, gdFTString ExtraPtr strex)
char *gdImageStringFTCircle(gdImagePtr im, int cx, int cy, double radius,
double textRadius, double fillPortion, char*font, double point s,
char*top, char*bottom, int fgcolor)
char *gdImageStringTTF(gdImagePtr im, int *brect, int fg, char *fontname,
double ptsize, double angle, int x, int y, char *string)

**Color-handling functions:**

int gdImageColorAllocate(gdImagePtr im, int r, int g, int b)
int gdImageColorAllocateAlpha(gdImagePtr im, int r, int g, int b, int a)
int gdImageColorClosest(gdImagePtr im, int r, int g, int b)
int gdImageColorClosestAlpha(gdImagePtr im, int r, int g, int b, int a)
int gdImageColorClosestHWB(gdImagePtr im, int r, int g, int b)
int gdImageColorExact(gdImagePtr im, int r, int g, int b)
int gdImageColorResolve(gdImagePtr im, int r, int g, int b)
int gdImageColorResolveAlpha(gdImagePtr im, int r, int g, int b, int a)
int gdImageColorsTotal(gdImagePtr im)(MACRO)
int gdImageRed(gdImagePtr im, int c)(MACRO)
int gdImageGreen(gdImagePtr im, int c)(MACRO)
int gdImageBlue(gdImagePtr im, int c)(MACRO)
int gdImageGetInterlaced(gdImagePtr im)(MACRO)
int gdImageGetTransparent(gdImagePtr im)(MACRO)

void gdImageColorDeallocate(gdImagePtr im, int color)
void gdImageColorTransparent(gdImagePtr im, int color)
void gdTrueColor(int red, int green, int blue)(MACRO)
void gdTrueColorAlpha(int red, int green, int blue, int alpha)(MACRO)

**Resizing functions:**

void gdImageCopy(gdImagePtr dst, gdImagePtr src, int dstX, int dstY, int srcX, int srcY, int w, int h)
void gdImageCopyResized(gdImagePtr dst, gdImagePtr src, int dstX, int dstY, int srcX, int srcY, int destW, int destH, int srcW, int srcH)
void gdImageCopyResampled(gdImagePtr dst, gdImagePtr src, int dstX, int dstY, int srcX, Int srcY, int destW, int destH, int srcW, int srcH)
void gdImageCopyRotated(gdImagePtr dst, gdImagePtr src, doubledstX, doubledstY, int srcX, int srcY, int srcW, int srcH, int angle)
void gdImageCopyMerge(gdImagePtr dst, gdImagePtr src, int dstX, int dstY, int srcX, int srcY, int w, int h, int pct)

void gdImageCopyMergeGray(gdImagePtr dst, gdImagePtr src, int dstX, int dstY, int srcX,
int srcY, int w, int h, int pct)
void gdImagePaletteCopy(gdImagePtr dst, gdImagePtr src)
void gdImageSquareToCircle(gdImagePtr im, int radius)
void gdImageSharpen(gdImagePtr im, int pct)

**Miscellaneous Functions:**

int gdImageCompare(gdImagePtr im1, gdImagePtr im2)
gdImageInterlace(gdImagePtr im, int int erlace)
gdFree(void *ptr)

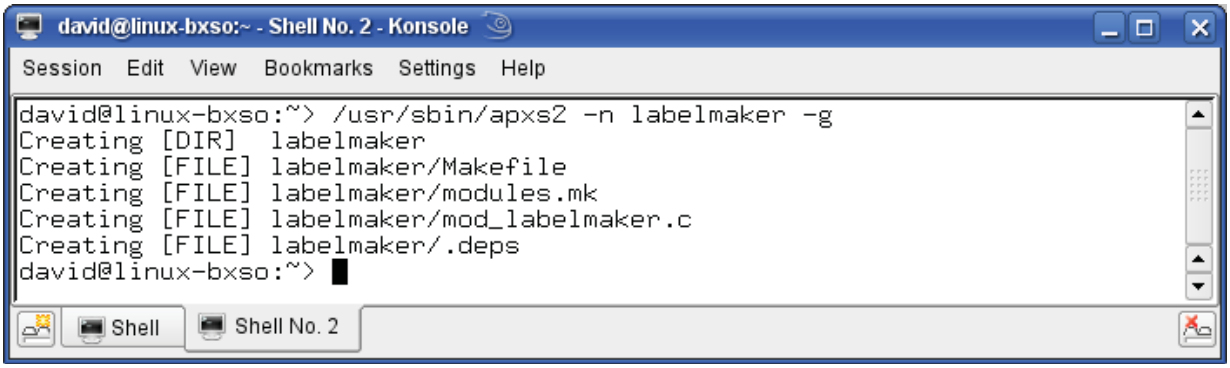
In order to use a library like this you will need familiarity with the arguments which are often data types defined within the library itself such as gdmagePtr which is a pointer to some kind of structure containing all the data for an image to be processed or stored. These may all seem unusual but after a while you will begin to get used to the syntax and on-line documentation and begin to see patterns in the complexity.

# 7. Apache C modules

## 7.1 Safer C web applications

In real life few web administrators would dream of letting anyone run C programs as CGI content generators because of the risk of crashes and core dumps. However the Apache server is itself written in C there are simple utilities that come with its development tools that permit you to create code stubs into which you can place your C programs and run them as Apache modules when they are loaded as part of the server and managed safely in a kind of "sand-box". Here we will take an earlier example and turn it into an Apache module.

A utility called apxs2 is included in the Apache2 development libraries which can be invoked to generate a code stub for a program which can be compiled into a module that is loaded and managed by the Apache web server. These modules can be used to perform a huge variety of tasks but in our case we will do something which is akin the an ISAPI DLL found in the IIS server. The exact location of the apxs2 utility will change according to the Linux distribution you are using but with OpenSuse it runs like this.
In a terminal type: **/usr/sbin/apxs2 -n labelmaker –g**




This creates a folder of the name you give it (labelmaker) and a **Makefile**, a **modules.mk** file which can be used by the Make utility, and a file called **mod_labelmaker.c**.

The C file generated is kind of like a Hello World for Apache. It may look like a complex thing but it does supply a long explanatory comment header which is worth reading. The idea is that when Apache starts any modules in a specified location which are configured as needing to be loaded in the server configuration files, will be loaded. The *_register_hooks function lists the names and signatures of functions that can be called at specific stages in the Apache server process. In this case if the name http://localhost/labelmaker is called this module will be asked to handle whatever happens in the *_handler function.

The configuration of the server can be a bit fiddly but in OpenSuse we have to add this to the file

**/etc/apache2/default-server.conf**

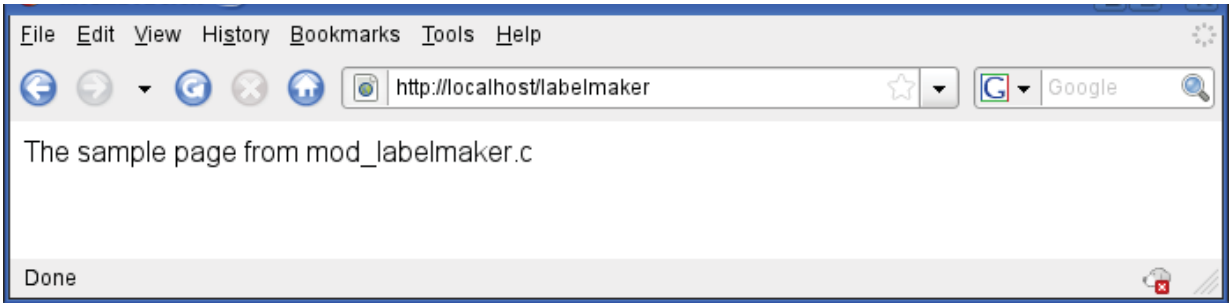
```
<Location /labelmaker>
        SetHandler labelmaker
</Location>
```

and in **/etc/config.sys/apache2** we **add** the name of our module labelmaker to long comma-separated list in the line starting

APACHE_MODULES=".....,labelmaker"
Now go to the folder labelmaker and type:

**sudo /usr/sbin/apxs2 -c -i mod_labelmaker.c**
**sudo /etc/init.d/apache2 restart**

Call this in a browser like this:

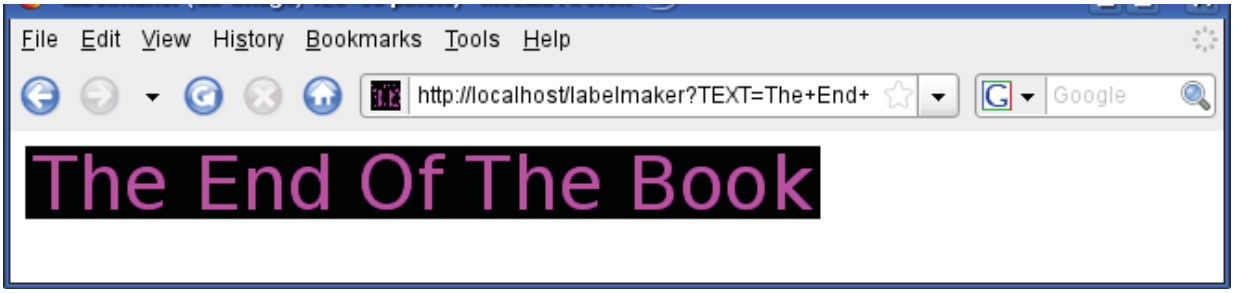



```
#include "httpd.h"
#include "http_config.h"
#include "http_protocol.h"
#include "ap_config.h"

/* The sample content handler */
static int labelmaker_handler(request_rec *r)
{
    if (strcmp(r->handler, "labelmaker")) {
        return DECLINED;
    }
    r->content_type = "text/html";

    if (!r->header_only)
        ap_rputs("The sample page from mod_labelmaker.c\n", r);
    return OK;
}

static void labelmaker_register_hooks(apr_pool_t *p)
{
    ap_hook_handler(labelmaker_handler, NULL, NULL, APR_HOOK_MIDDLE);
}

/* Dispatch list for API hooks */
module AP_MODULE_DECLARE_DATA labelmaker_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                  /* create per-dir    config structures */
    NULL,                  /* merge  per-dir    config structures */
    NULL,                  /* create per-server config structures */
    NULL,                  /* merge  per-server config structures */
    NULL,                  /* table of config file commands       */
    labelmaker_register_hooks  /* register hooks                      */
};
```

## 7.2 Adding some functionality

Now we can plug in the work we did for the graphics library in Chapter 6 as a replacement handler function (in the code Chapter7_1.c there are BOTH handlers, one commented out). Note the (highlighted) call to a modified decode_value function that uses the r->args pointer to get the QUERY_STRING rather than getenv() . Also Apache handles the output a bit differently too – get get a pointer to the array of bytes in the image by calling gdImageGifPtr then the ap_rwrite function outputs the data. We have to free the pointer with gdFree after the output call.

```
static int labelmaker_handler(request_rec *r)
{
        void    *iptr;
        int sz = 0;

   if (strcmp(r->handler, "labelmaker")) {
      return DECLINED;
   }
                r->content_type = "Content-type: image/gif";
   if (!r->header_only){
        int text=0,background=0, x=0,y=0,size=30,string_rectangle[8];
        double angle=0.0;
        char value[255] = "Hello";
        char font[256] = "/usr/share/fonts/truetype/DejaVuSans.ttf";
        char *err = NULL;
        gdImagePtr im_out = NULL;
        decode_value(r,"TEXT=", (char *) &value, 255);
        err=gdImageStringFT(NULL,&string_rectangle[0],0,
                font,size,angle,0,0,value);
        x = string_rectangle[2]-string_rectangle[6] + 6;
        y = string_rectangle[3]-string_rectangle[7] + 6;
        im_out = gdImageCreate(x,y);
        background = gdImageColorAllocate(im_out, 0,0,0);
        text  = gdImageColorAllocate(im_out,255,0,255);
        x = 3 - string_rectangle[6];
        y = 3 - string_rectangle[7];
        err = gdImageStringFT(im_out,&string_rectangle[0],text,
                font,size,angle,x,y,value);
        iptr = gdImageGifPtr(im_out,&sz);
        ap_rwrite(iptr, sz, r);
        gdFree(iptr);
        gdImageDestroy(im_out);
   }
   return OK;
}
```




## 7.3 Apache Modules Conclusion

Whilst tricky to write and debug, this is probably the most rewarding and esoteric area where you can do real, commerically useful and safely deployable web content generation. It is easy to see how this example could be extended with parameters for colours and fonts to make a useful web content tool.

There is very little clear simple material about apache modukles but start with the on-line documentation at http://httpd.apache.org/docs/2.2/developer/

One recent book worth looking at is “The Apache Modules Book” Nick Kew, Prentice Hall.

# 8. The Ghost project

## 8.1 A PHP web site generator project

The ability to write short programs in C to automate tedious tasks or to do things that would otherwise take hours of fiddling about with cumbersome tools such as doing mail-merge, is one on the things you will be most pleased you have learned how to do. This project is such a time-saver. Ghost is a lightweight PHP generator for you to customise.

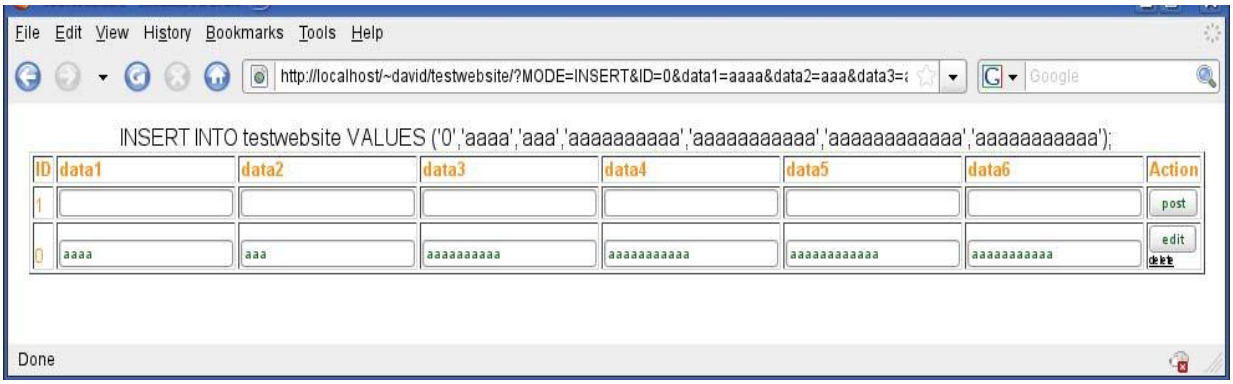
If you find yourself having to build PHP web sites all the time, a quick way to generate all the parameter-passing, decoding, forms building and database management code in one step would be useful. Tools like Ruby on Rails offer such functionality but are infinitely more complex to set up and run and you end up with needing to learn yet another language to go any further.

Probably the best way to start with this tool is to compile and run it. Unzip the ghost.zip source into your public_html folder which creates a folder called ghost. The Makefile contains a target **g1** that compiles and links ghost. So go to **public_html/ghost** and type: **make g1** .

To run the site generator type:

- **./ghost testwebsite data1 data2 data1 data3 data4 data6 data6**
- This will create:
- a folder **public_html/testwebsite**
- a mysql database table called **testwebsite** with text fields data1 data2 data1 data3 data4 data6 data6
- a **testwebsite.css** file
- empty **header.html** and **footer.html** pages
- **index.php** that demonstrates a form handling entry, edit & update, and delete to the database table for the data items specified.

In a browser what you see is this at **http://localhost/~yourname/testwebsite**



The idea behind this is that all the mechanical bits to create and manage the form content are done and can be customised. This screen shot shows the result of submitting one record. Thde top row is for entering new data, the lower row(s) allow editing or deleting of records. It is a framework that allows you to take and use parts in your own website design.

Let us examine this code in sections.

The first section declares the required data and creates the folder and CSS file.

```
int main(int argc, char *argv[])
{
	FILE *out = NULL;
	MYSQL *conn = NULL;
	MYSQL_RES *result = NULL;
	MYSQL_ROW row;
	MYSQL_FIELD *field;
	char SQL[STRINGSIZE]="";
	char BIT[STRINGSIZE]="";
	char SQLINSERT[STRINGSIZE]="";
	char SQLUPDATE[STRINGSIZE]="";
	char SQLDELETE[STRINGSIZE]="";
	int rc=0, i=0, num_fields=0;

	//CREATE DIRECTORY////////////////////////////////
	sprintf(BIT,"mkdir ~/public_html/%s",argv[1]);
	system(BIT);
	//BUILD CSS/////////////////////////////////////////
	sprintf(BIT,"../%s/%s.css",argv[1],argv[1]);
	out = fopen(BIT,"w");
	fprintf(out,"table {width:800;vertical-align:top;}\n");
	fprintf(out,"th {font-size:80%;color:#fd9208;vertical-align:bottom;text-align:left;}\n");
	fprintf(out,"td {font-size:80%;color:#fd9208;vertical-align:bottom;text-align:left;}\n");
	fprintf(out,"input {font-size:80%;color:#196419;}\n");
	fprintf(out,"a  {font-size:65%;color:#000000;}\n");
	fclose(out);
```

Next the header.html and footer.html files are generated. These files is loaded by the PHP file and could be used as a generic common header and footers. The CSS file is referenced from the header.html file.

```
//BUILD HEADER HTML//////////////////////////////////
sprintf(BIT,"../%s/head.html",argv[1]);
out = fopen(BIT,"w");
fprintf(out,"<html>\n");
fprintf(out,"<head>\n");
fprintf(out,"<link rel=\"stylesheet\" type=\"text/css\"href=\"%s.css\" />\n",argv[1]);
fprintf(out,"<title>%s</title>\n",argv[1]);
fprintf(out,"</head>\n");
fprintf(out,"<body>\n");
fprintf(out,"<center>\n");
fprintf(out,"<table><tr><td>\n");
fprintf(out,"</td></tr></table>\n");
fclose(out);
//BUILD FOOT HTML//////////////////////////////////////
sprintf(BIT,"../%s/foot.html",argv[1]);
out = fopen(BIT,"w");
fprintf(out,"</body></html>\n");
fclose(out);
```

Next we create the data base.

```
//OPEN DATABASE//////////////////////////////////////
conn = mysql_init((MYSQL *) 0);
mysql_options(conn,MYSQL_READ_DEFAULT_GROUP,"mysqlcapi");
mysql_real_connect(conn, "localhost","","","test",0, NULL, 0);
//CREATE TABLE//////////////////////////////////////
sprintf(SQL,"drop table if exists %s",argv[1]);
rc = mysql_query(conn,SQL);
sprintf(SQL,"create table %s (ID varchar(255)",argv[1]);
for(i=2; i < argc; i++)
{
        sprintf(BIT,",%s varchar(255)",argv[i]);
        strcat(SQL,BIT);
}
strcat(SQL,");");
rc = mysql_query(conn,SQL);
```

The complicated part starts now, of generating a php script. The best way to understand this is to examine the actual output of the program when we view the source of the page in the browser.

The top row is a form with a text box for each column defined in the table generated by running the ghost program.

```
<tr>
<form><input type='hidden' name='MODE' value='INSERT'>
<td><input type=hidden name='ID' value='1'>1 </td>
<td><input type=text name='data1'></td>
<td><input type=text name='data2'></td>
<td><input type=text name='data3'></td>
<td><input type=text name='data4'></td>
<td><input type=text name='data5'></td>
<td><input type=text name='data6'></td>
<td><input type='submit' value='post'></td>
</form>
</tr>
```

For each row in the table we now generate a form allowing editing of the data and an anchor link to do a delete operation.

```
<tr>
<form><input type='hidden' name='MODE' value='UPDATE'>
<td><input type=hidden name='ID' value='0'>0</td>
<td><input type=text name='data1' value='aaaaaaaaaaa'></td>
<td><input type=text name='data2' value='aaaaaaaaaaaaaaa'></td>
<td><input type=text name='data3' value='aaaaaaaaaaaaaaa'></td>
<td><input type=text name='data4' value='aaaaaaaaaaaaa'></td>
<td><input type=text name='data5' value='aaaaaaaaaaaaaaa'></td>
<td><input type=text name='data6' value='aaaaaaaaaaaa'></td>
<td><input type='submit' value='edit'></form>
<a href='?ID=0&MODE=DELETE'>delete</a></td
</tr>
```

Close examination of the file index.php will allow you to see where all this happens, and to work backward to find where in the ghost.c source code this PHP code is generated. A good idea is to use a highlighter pen on a printout as we are embedding a language (HTML) inside another language (PHP) which is in turn inside another language so very very careful use is made of the escape characters '\ 'to express quotation marks both single and double where necessary to make it all work. This may seem complex – but the speedy prototyping that ghost permits makes it worthwhile to spend time customising the C code so the PHP that you want and the database you want come out the way you want it.

Here is the part of the PHP file index.php which generates the edit or delete rows. The static HTML is highlighted and the other parts are inserted by MySQL PHP function calls.

```
$result = mysql_query("select * from testwebsite");
while($row = mysql_fetch_array($result))
{
        echo "<tr>";
        echo "<form><input type='hidden' name='MODE' value='UPDATE'>";
        for($i=0;$i < mysql_num_fields($result); $i++)
        if($i==0)
                echo "<td><input type=hidden name='" .
                        mysql_field_name($result,$i) . "' value='" .
$row[mysql_field_name($result,$i)] .
                "'>".
              $row[mysql_field_name($result,$i)] .
                "</td>";
        else
                echo "<td><input type=text name='" .
                        mysql_field_name($result,$i) .
                        "' value='" .
                        $row[mysql_field_name($result,$i)] .
                                "'></td>";
        echo "<td><input type='submit' value='edit'></form>";
        echo "<a href='?ID=" .
                $row[mysql_field_name($result,0)] .
                "&MODE=DELETE'>delete</a>";
        echo "</td></tr>";
}
mysql_free_result($result);
mysql_close($con);
```

As you can see a great deal of tedious and repetitive works has been automated. You can move on by modifying the PHP code or go deeper to customise the C program which generates all of it.

I personally use ghost frequently to save time on site-building and this is why I wrote it. I got bored making mistakes writing virtually identical code to decode HTML forms and populate or update databases.

# Conclusion

Now you have worked through these simple examples I hope you can see why a knowledge of the C language is always going to be a useful and continually practical skill.

After 25 years I still regularly write C programs to do everyday tasks quickly and effectively and once written they form part of a set of durable tools that suit me and which are portable.

This short book contains a number of “tricks” that I have learned over the years for which there is little explicit documentation, and it frankly presupposes a familiarity with Linux. After testing the material with students of a wide range of experience I am confident that an attentive careful student will get all this material working and can start from there to discover the joy of C. Many of my students say “This is much more fun that Java, I can see what is really going on!” which is most gratifying and makes me confident you will find the material useul.

Those interested might ask the publisher to commission me to write a companion volume :

*C++ Standard Template Library Programming in Linux*

Good luck and happy programming.

David
March 31st 2009