2 Data and Memory

2.1 Simple data types?

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

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

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

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

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

Here we are doing the following:

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

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

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

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

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

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

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

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

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

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

A note on *make* a helpful utility

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

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

to compile everything type **make all**

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

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

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

2.2 What is a string?

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

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

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

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

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

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

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


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

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

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

david

2.3 What can a string "mean"

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

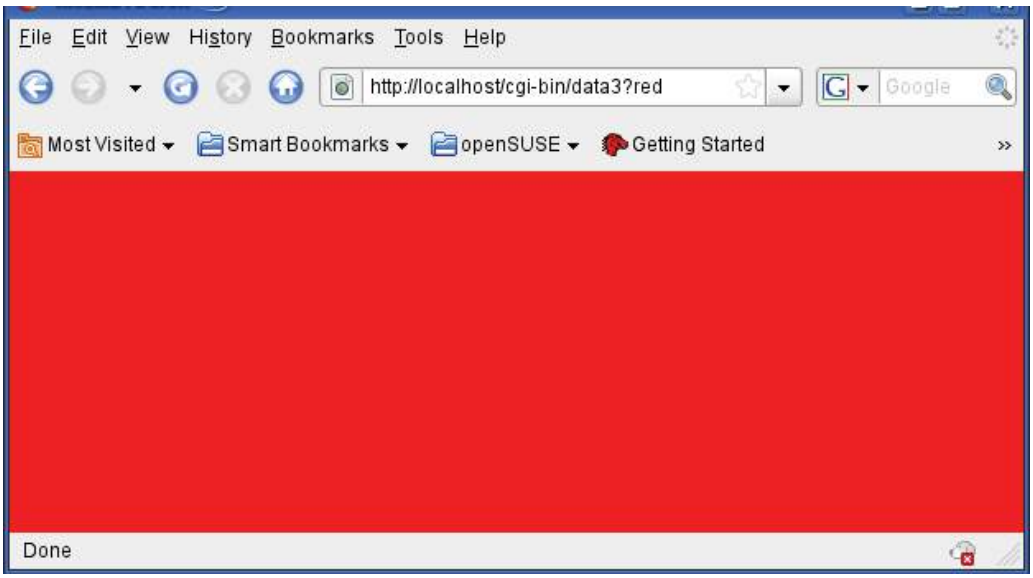
```
/*****************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_3.c                                   *
*****************************************************************/
#include <stdio.h>
#include <string.h>

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

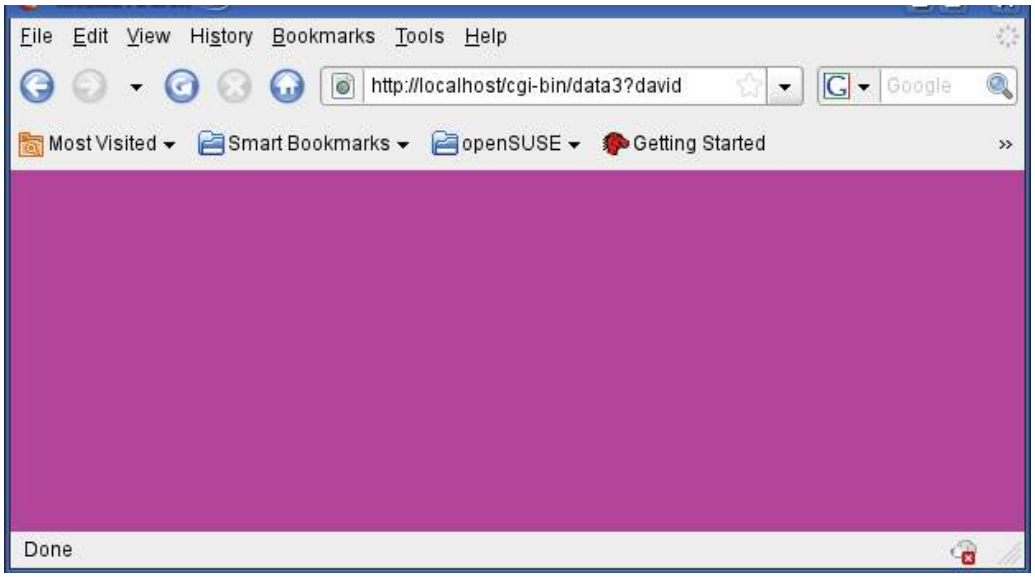
Compile with: **gcc -o data3 chapter2_3.c -lc**

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

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



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.

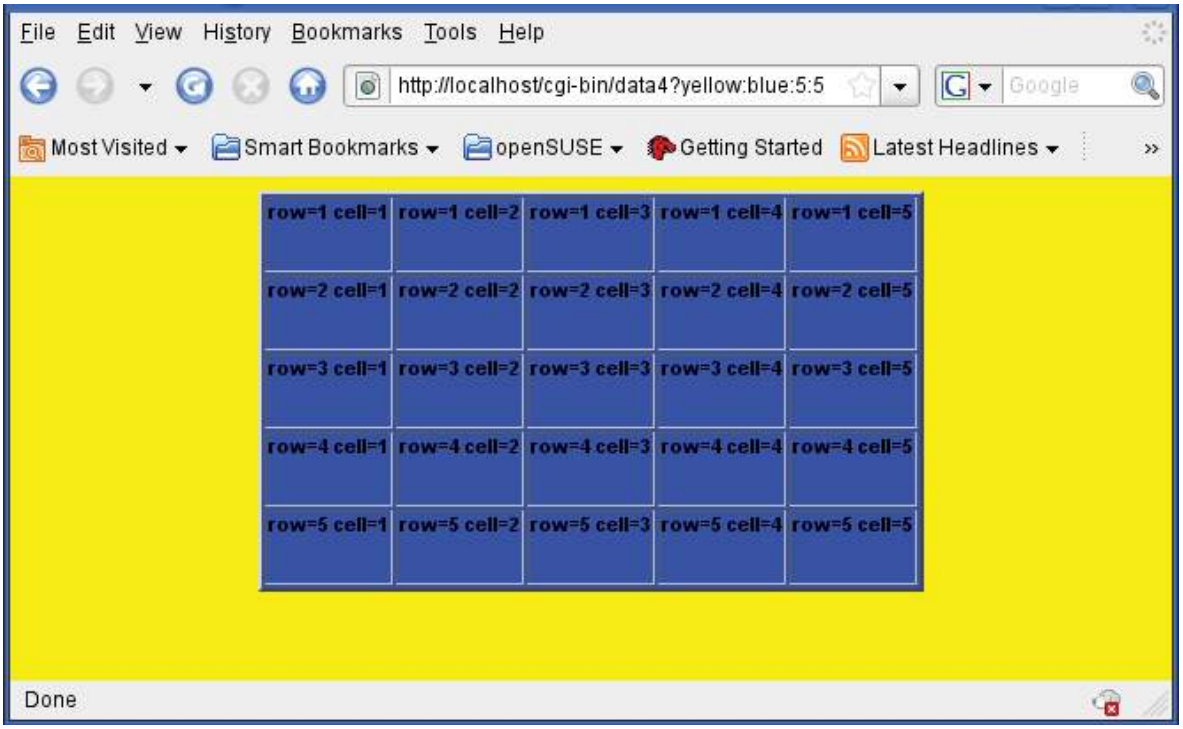
```
/*********************************************************************
* C Programming in Linux (c) David Haskins 2008
* chapter2_4.c                                   *
*********************************************************************/
#include <stdio.h>
#include <string.h>

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: