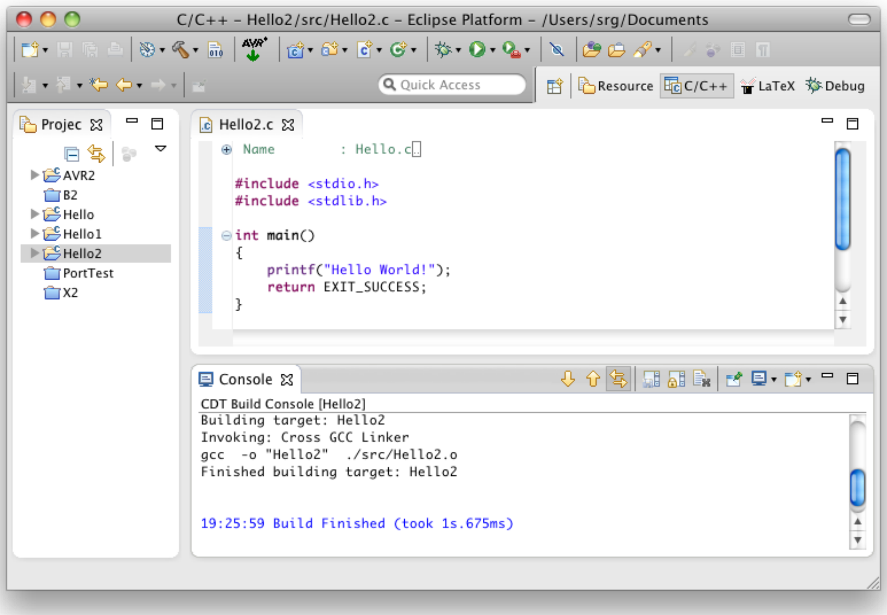
C4

Strings & Structures

Earlier in this module you have used variables to store data, in particular integer and decimal numbers, as well as individual characters. This exercise focusses on two more advanced features of C for data storage: strings and structures. Strings allow you to write programs to manipulate text; the standard C library include a number of utility functions for them. Structures combine multiple variables together, and can be used to represent anything from archive records to multimedia objects, such as images and sounds. In this exercise you will not only write your own functions to compare and manipulate strings and structures, but you will also take advantage of functions available in the C standard library to solve more advanced tasks.



```
 Name        : Hello.c

#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello World!");
    return EXIT_SUCCESS;
}
```

```
CDT Build Console [Hello2]
Building target: Hello2
Invoking: Cross GCC Linker
gcc  -o "Hello2"  ./src/Hello2.o
Finished building target: Hello2

19:25:59 Build Finished (took 1s.675ms)
```

Schedule

Preparation Time	:	3 hours
Lab Time	:	3 hours

Items provided

Tools	:	
Components	:	
Equipment	:	
Software	:	`gcc`, `Eclipse`

Items to bring

Course Textbook - *C Programming in Easy Steps*

Identity card

Laboratory logbook

Version: September 29, 2013

Enrico Costanza, Steve R. Gunn
Electronics and Computer Science
University of Southampton

Before entering the laboratory you should read through this document and complete the preparatory tasks detailed in section 2.

> **Academic Integrity** – *If you wish you may undertake the preparation jointly with other students. If you do so it is important that you acknowledge this fact in your logbook. Similarly, you will probably want to use sources from the internet to help answer some of the questions. Again, record any sources in your logbook.*

You will undertake the exercise working with your laboratory partner. During the exercise you should use your logbook to record your observations, which you can refer to in the future – perhaps to write a formal report on the exercise, or to remind you about the procedures. As such it should be legible, and observations should be clearly referenced to the appropriate part of the exercise. As a guide the ✎ symbol has been used to indicate a mandatory entry in your logbook. However, you should always record additional observations whenever something unexpected occurs, or when you discover something of interest.

For each Task you should create a new directory so that you have a working version of every program at the end of the lab. Remember to place comments in your code.

You will be marked using the standard laboratory marking scheme; at the beginning of the exercise one of the laboratory demonstrators will mark your preparatory work and at the end of the exercise you will be marked on your progress, understanding and logbook.

Notation

This document uses the following conventions:

✎	An entry should be made in your logbook
☆	A hint to application areas—can be ignored

1 Outcomes

Having successfully completed this lab session, you will be able to:

- design and implement simple C programs to manipulate strings;
- design and implement simple C programs that use structures;
- design and implement simple C programs that take advantage of functions defined in existing libraries.

2 Preparation

Read chapters 8 and 9 of the course textbook (McGrath, M, “C Programming in Easy Steps,” In Easy Steps Ltd., 2012), then answer the following questions and exercises on your log book.

- How are strings similar and different from arrays? Is a string just an array of chars?
- If you see the following statement: `a.value = b->value;` what can you assume about the variables a and b?
- What is the purpose of a structure?
- Write in your logbook the declaration of a structure which contains a string and an integer and code to access the elements of the structure and assign them values.
- What do you need to do in your program and at compile time to use a C function from a library?

Below (Listing 1) is a C implementation of bubble sort. Comment the code to show you understand what is going on.

LISTING 1: a bubble sort implementation

```c
void sort_array(const int size, const float input[], float output[])
{
    int j = 0, swapped = 0;
    float tmp;

    for (j=0; j<size; j++)
    {
        output[j] = input[j];
    }

    while (swapped == 0)
    {
        swapped = 1;
        for (j=0; j<(size-1); j++)
        {
            if (output[j] > output[j+1])
            {
                    swapped = 0;
                    tmp = output[j];
                    output[j] = output[j+1];
                    output[j+1] = tmp;
                }
            }
        }

        return;
}
```

3 Laboratory Work

Open the Eclipse application on the computer you are using and when asked for the workspace you wish to use, make sure the checkbox for *use as default* is not ticked and type `H:\ELEC1201\Labs\C4`.

Remember to create a new C project (File -> New -> New project.. -> C/C++ -> C Project), for each task of the exercise and name it with the appropriate part of the exercise for easy future reference, e.g. `C4_3.1.1`. You can select "Hello World ANSI C Project" as "Project type" and replace the contents with your own code, or create an "Empty Project" and add in a new source file.

Remember, if you are asked if you want to "Switch perspective" answer positively.

3.1 Copying & Comparing

3.1.1 Copying Strings

Write a function to copy the content of a string into another string. Your function should have the following prototype:

```
void copy (char result[], const char input[]);
```

Write a program to demonstrate your function in action.

3.1.2 Copying Structures

Write a function to copy the content of a structure into another structure. Define the structure so that it contains (at least) an integer and a string. Write a program to demonstrate your function in action.

3.1.3 Comparing Strings

Write a function to compare two strings equivalent to strcmp. In other words the function should have the following prototype:

```
int strcmp ( const char * str1, const char * str2 );
```

The function should return *zero* if the strings are identical. If the strings are not identical, the function should return +1 if str1 comes before str2 in alphabetical order and -1 if str2 comes before str1. Write a program to show your function in action.

3.1.4 Comparing Structures

Write a function to compare the content of two structures. The function should return 0 if the structures content is identical, 1 otherwise. Define the structure so that it contains (at least) an integer and a string. Write a program to show your function in action.

Following the definition above, the function you wrote will tell you whether the structures are "the same" or "different" – however, could you say whether one structure is > or < the other? For example, think about a structure that contains a string and an int variable. Hint: think about a spreadsheet application (e.g. OpenOffice Calc, LibreOffice Calc or MS Excel) where you have the following table:

Fruit	Quantity
Apples	10
Pears	8
Bananas	3
Oranges	2

How can you sort this data in the spreadsheet application (you could represent each row of the table in the kind of structure described above)? Take notes in your logbook about this issues. ✐

Please modify your compare function so that it returns zero if the structures are identical, a positive value if first structure is "more than" (however you define this concept of "more than") the second and a negative value otherwise.

3.2 Bubble Sort (again)

Write a function that uses bubble sort to sort an array of strings. Please start from the implementation of bubble sort you developed in the previous lab, or the one you annotated in preparation for this lab, and use a similar function signature.

Hints:

- To declare the array of strings use something like: `char words[N_STR][STR_SIZE];` where `N_STR` and `STR_SIZE` are defined as constants.
- The sort function should have the following signature:

```
void sort_strings(const int size, const char src[size][STR_SIZE],
char dst[size][STR_SIZE]);
```

4 Optional Additional Work

4.1 Extracting Unique Words from Text

Your task for this part is to use the strtok function to parse words out of the following string and add them to an array of strings so that each word appears in the array only once.

> "C (pronounced like the letter C) is a general-purpose computer programming language developed between 1969 and 1973 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system. Although C was designed for implementing system software, it is also widely used for developing portable application software. C is one of the most widely used programming languages of all time and there are very few computer architectures for which a C compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++, which began as an extension to C."
> (from Wikipedia)

Hints:

1. Copy and paste the text into a constant string in your code. E.g.
   `char test[] = "C (pronounced like..."`
2. Do not copy and paste from the .pdf file, the same text is available among the lab files: c_wiki_short.txt.
3. Declare a fixed size array of strings large enough to contain all the words you need to store.
4. As a separator for strtok you can use the following string: `" ;().,\n"`

4.2 Using an array of structures for counting the frequency of words

Modify the prior program to keep track of the frequency of each word in the text. Use the following structure to store your data:

```
typedef struct {
    char word[32];
    int count;
} Entry;
```

Rather than working with an array of strings, work with an array of Entry structures. Every time a word is added for the first time to the array set count to 1. If the same word appears again, increment the count.

Sort the array of structures by count, to see what are the most frequent words in the text. You could now use this information to create a tag cloud!

Hint: check you get the correct results using the following information.

The words "and" "computer" "system" "languages" "most" "used" "widely" "which" "software" "a" "of" appear 2 times;
the words "is" and "programming" appear 3 times;
the words "for" and "the" appear 4 times;
the word "C" appears 7 times;
all other words appear once.

4.3 Quick Sort: Passing a function (pointer) to a function

Very often you will find that the tasks you need to do in programming are already implemented in functions contained in existing libraries, so it is important to start very early on to get used to take advantage of existing functions, and the potential challenges associated with this.

Sorting is a very common task in programming, so a very efficient sorting function is included in the essential C libraries:

```
void qsort(void *base, size_t nel, size_t width,
int (*compar)(const void *, const void *));
```

You should recognize the basic structure of a C function prototype and pointer arguments. However, what data types are `void` and `size_t`? And what are all those extra parenthesis in the second half of the line?

The qsort function is used to sort arrays, regardless of the type of the array. Having completed Section 3.1.4 and Section 3.2, you should intuitively see that the same sorting mechanism can be applied to different “things,” as long as you have a way to compare a pair of them. In Lab 3 you used the < or > operators to compare 2 integer numbers, while in Section 3.2 you should have used the `strcmp` function to compare 2 strings.

We know that the keyword `void` can be used to say “nothing” e.g. to say that a function does not return any value, or does not take any argument. However, the same keyword can be used with pointers (note: only with pointers not with variables!) to define a pointer of “generic type.” The type `size_t` is equivalent to unsigned int (for more information see this Wikipedia article).

The first argument then is the pointer to the array that we want to sort. The second argument is the size of the array in the sense of the number of elements, the third argument is the size in bytes of each element (as returned by the `sizeof()` operator), while the last argument is a pointer to a function that `qsort` will use to compare pairs of elements in the array we want to sort.

A generic pointer can, and somehow should, look a bit intimidating: without knowing what type of data we have, how can one do anything with it? Very often void pointers need to be casted back to pointers of some other type before being used – it is crucial to note that when doing so one needs to know what type of data was there to start with!

Let's look at a practical example: say that we want to use `qsort` to sort an array of integers. We then need to implement a function to compare two integers. However, the signature of the function you write needs to be compatible with what `qsort` requires. You can use something like:

```
int int_cmp(const void* a, const void* b)
{
    int result = 0;
    /* calculate result here.. */
    return result;
}
```

If you are sure that the generic pointers are actually integer pointers, you can cast them back in the following way:

```
const int * ptr_a = (const int *) a;
const int * ptr_b = (const int *) b;
```

Then you can actually just dereference them and compare them:

To call qsort you also need to cast the array pointer to a void pointer, as follows:

```
void * generic_ptr = (void *) array;
qsort(generic_ptr, array_size, sizeof(int),
int (*compar)(const void *, const void *));
```

The full example is in the file qsort_int.c. Create a project for it, compile it and run it.

Starting from the example qsort_int.c (but create a new project) write a program to sort the data defined in Section 3.1.4 of this lab using the qsort function. Using the structure comparison function you wrote in Section 3.1.4 as a starting point (you will need to modify it to make it work with qsort).