C Lecture -3 Outline

- ➢ Scope
- ➢ Arrays
- ➢ Strings
- ➢ fgets()

Scope

- A variable name is not always defined over the entire length of a program.
- The part of the program in which a variable name is defined is called its *scope*.
- Variable names declared outside of any block have *file scope*.
- Variable names declared inside a block have *block scope*.

File Scope

- The variable a has file scope.
- It is defined outside any blocks, and so, by default the name a will refer to it everywhere in the file.

```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int b = 6;
  }
}
```

Block Scope

- The variable b has block scope.
- It is defined inside a block, and therefore the name b will only refer to it for statements inside that block.

```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int b = 6;
  }
 }
```

Block Scope

* Block scope variables can *shadow* variables of the same name in an wider scope.
* The variable a in the block scope shadows the file scope a inside that block.
* Any calculation using the name a in that block will use the block scope a.

Shadowing is generally regarded as bad practice!

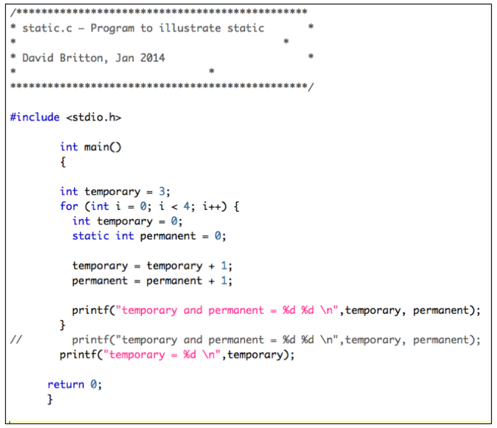
```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int a = 77;
    int b = a;
    // b == 77, not 5!
  }
  //returns the value 5
  //block a out of scope.
  return a;
}
```

Allocation (of memory)

* Just as variable *names* have scope over which they have meaning, the memory used to store their values also has a limited duration, or *lifetime*.
* Block scope variables usually backed by *automatic* allocation. and de-allocation The storage used for the variable is allocated to it when the block it is declared in starts, and then automatically given back when we encounter the end of the block.
* File scope variables, and block scope variables declared with the `static` keyword before their type, have *static* allocation. While the name associated with the storage may only have a limited scope, the storage itself is kept for the duration of the program, along with any values in it.

So that you can re use value
Next time enter block

Example



```
/************************************************
* static.c - Program to illustrate static      *
*                                            *
* David Britton, Jan 2014                      *
*                              *
************************************************/

#include <stdio.h>

        int main()
        {

        int temporary = 3;
        for (int i = 0; i < 4; i++) {
          int temporary = 0;
          static int permanent = 0;

          temporary = temporary + 1;
          permanent = permanent + 1;

          printf("temporary and permanent = %d %d \n",temporary, permanent);
        }
//        printf("temporary and permanent = %d %d \n",temporary, permanent);
        printf("temporary = %d \n",temporary);

      return 0;
      }
```

Arrays

* In the last lecture, we introduced loop constructs, which let us perform the same actions repeatedly until a condition was met.
* It would be useful to be able to represent lists, or *arrays*, of values so that we can use loops to efficiently perform the same actions on each of them.
* C provides a syntax for creating *array variables* which do precisely this (their storage in memory is simply all the values in the array, one immediately after the other).

Declaring an Array

* An array variable is declared much like a scalar variable: the declaration starts with a type specifier, followed by the names we want to use for variables.
* To make a variable an array, we suffix the name with the number of elements we would like the array to hold, surrounded by [ ]
* We can mix array and scalar names in the same declaration, if their base value type is the same.

```
double a[5];
int b[4] = {0,4,2,27};

//q and z are just shorts
//h an array of 56 shorts
short q, h[56], z;
```

Declaring an Array

* We can give all the elements of an array initial values, provided in a comma-separated list surrounded by { }
* If we specify an initialiser list like this, then we can leave out the explicit size of the array between the [ ]s; the compiler will make an array exactly big enough to hold the list.

```
double a[5];
int b[4] = {0,4,2,27};

//q has size 2
double q[] = {0.1, 4.0e1};
```

Using Arrays

* Now we've made an array, how do we access the values in it?
* The values (*elements*) of an array are numbered, *starting at zero*.
* We specify an element using the same [ ] construct we used to declare our array. e.g. x = a[4];
* a[4] specifies the *5th* element of a (counting from 0).
* For maximum speed, C *does not check* that the array is long enough to contain the element you asked for!
* If a was of length 4, then our example would access memory outside the array, with undefined (and potentially bad) effects.

Array Example

```
#include <stdio.h>

int main() {
    int dataArray[] = {5,4,3,2,1};
    int sum = 0 ;

    for (int i = 0; i < 5; i++)
        sum += dataArray[i];

    printf("Sum of the data is %d \n",sum);

    return 0;
    }
```

12

Array Example Continued

```
#include <stdio.h>

int main() {
    int dataArray[] = {5,4,3,2,1};
    int sum = 0 ;

/*
    for (int i = 0; i < 5; i++)
      sum += dataArray[i];

Now lets replace for-loop with a do...while loop.
We will need to define a loop counter to end the looping */

//  int i = 5;
    int i = sizeof(dataArray) / sizeof(dataArray[0]);

    do {
     sum += dataArray[--i];
    } while (i > 0);

/*  while (i > 0)
        sum += dataArray[--i];
*/
    printf("Sum of the data is %d \n",sum);

    return 0;
    }
```

Comparing and Copying Arrays

- To compare two arrays - compare each element in turn using a loop. Note: `if(array1 == array2)` will always return 'false' because it is comparing the memory addresses of the two arrays which <u>are</u> different).
- Similarly, in general, use a loop to copy the contents of one array to another.

```
#include <stdio.h>

int main() {
    int a[] = {5,4,3,2,1};
    int b[] = {0,0,0,0,0};
    int n = sizeof(a) / sizeof(a[0]);

    for (int i = 0; i < n; i++) {
     if(a[i] != b[i]) { // compare the two arrays
        puts("The arrays are different");
        break;
     }
    }

    for (int i = 0; i < n; i++)
      b[i] = a[i];  // copy one array to another

    for (int i = 0; i < n; i++)
     printf("a[%d]=%d b[%d]=%d\n",i,a[i],i,b[i]);

    return 0;
    }
```

14

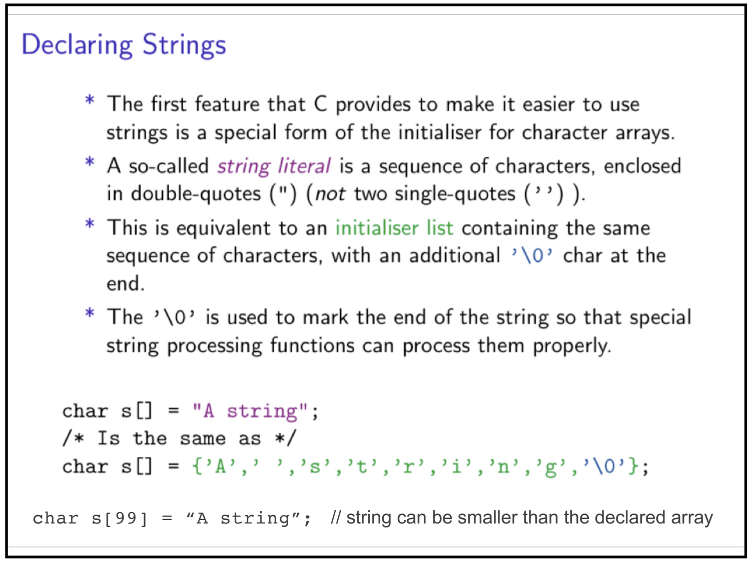
Multi-dimensional Arrays

- Arrays can have more than one dimension. The declaration for a two-dimensional array is
  `type variable[size1][size2]; /* comment */`
- Example:
  `int matrix[2][4]; /* a typical matrix */`
- C does not follow the notation used in other languages, e.g. `matrix[10,12]`.
- to access an element of the two-dimensional array `matrix` we use the notation
  `matrix[1][2] = 10;`
- C allows to use as many dimensions as needed, limited only by the amount of memory available. A four-dimensional array `four_dimensions[10][12][9][5]` is no problem.

15

Strings

* Now we know about general arrays, we address how C deals with strings of characters (i.e. "Text Strings").
* Text is "just" a sequence of characters, so we could just deal with strings as arrays of type `char`.
* As we deal with text a lot, C provides some additional features to make dealing with strings a little more convenient than most arrays.
* Behind the scenes, however, strings are essentially just character arrays.

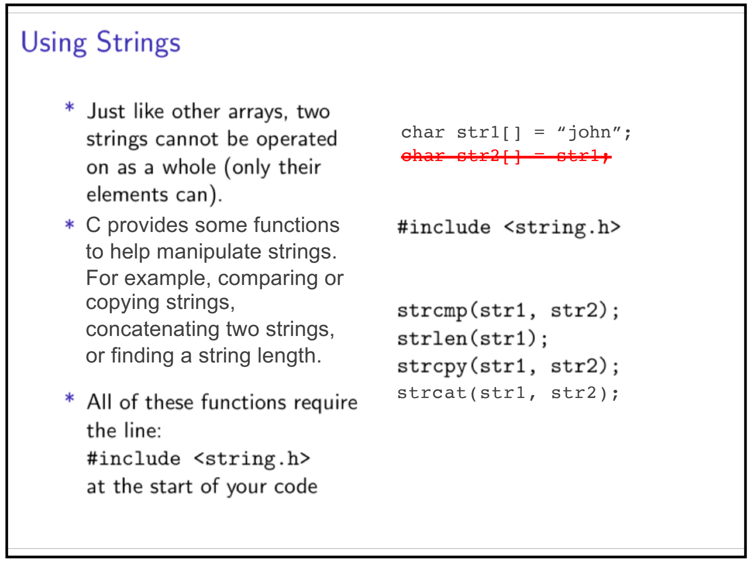


Declaring Strings

* The first feature that C provides to make it easier to use strings is a special form of the initialiser for character arrays.
* A so-called *string literal* is a sequence of characters, enclosed in double-quotes (") (*not* two single-quotes ('')).
* This is equivalent to an initialiser list containing the same sequence of characters, with an additional '\0' char at the end.
* The '\0' is used to mark the end of the string so that special string processing functions can process them properly.

```
char s[] = "A string";
/* Is the same as */
char s[] = {'A',' ','s','t','r','i','n','g','\0'};
```

`char s[99] = "A string";` // string can be smaller than the declared array

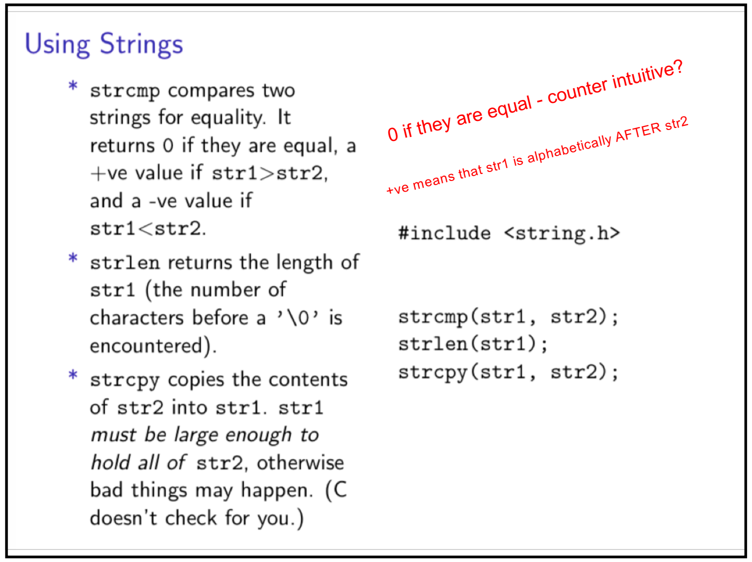


Using Strings

* Just like other arrays, two strings cannot be operated on as a whole (only their elements can).
* C provides some functions to help manipulate strings. For example, comparing or copying strings, concatenating two strings, or finding a string length.
* All of these functions require the line: `#include <string.h>` at the start of your code

```
char str1[] = "john";
```
~~`char str2[] = str1;`~~

```
#include <string.h>

strcmp(str1, str2);
strlen(str1);
strcpy(str1, str2);
strcat(str1, str2);
```



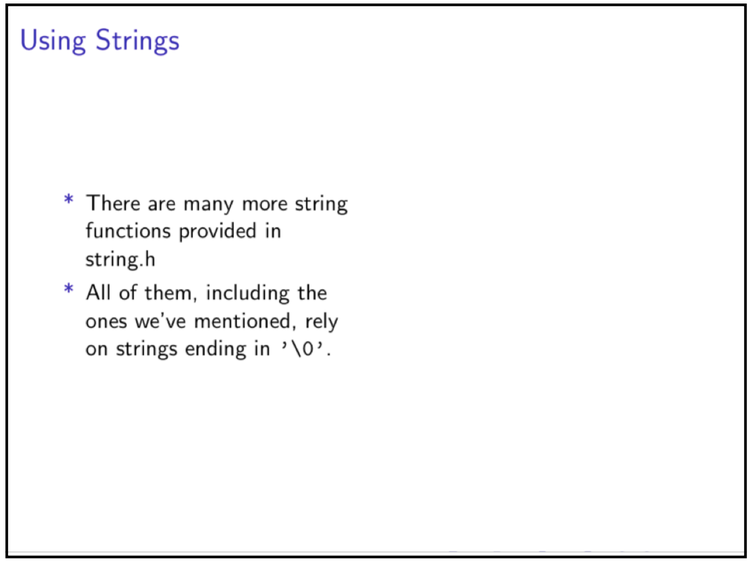
Using Strings

* strcmp compares two strings for equality. It returns 0 if they are equal, a +ve value if str1>str2, and a -ve value if str1<str2.
* strlen returns the length of str1 (the number of characters before a '\0' is encountered).
* strcpy copies the contents of str2 into str1. str1 *must be large enough to hold all of* str2, otherwise bad things may happen. (C doesn't check for you.)

0 if they are equal - counter intuitive?

+ve means that str1 is alphabetically AFTER str2

```
#include <string.h>

strcmp(str1, str2);
strlen(str1);
strcpy(str1, str2);
```



Using Strings

* There are many more string functions provided in string.h
* All of them, including the ones we've mentioned, rely on strings ending in '\0'.

Reading in Strings with `fgets()`

The standard function `fgets` can be used to read a string from the keyboard. The general form of an `fgets` statement is:

`fgets(name, size, stdin);`

- `name`
  is the name of a character array, aka a string variable. The line, including the end-of-line character, is read into this array.
- `size`
  `fgets` reads until it gets a line complete with ending `\n` or it reads `size` - 1 characters. It is convenient to use the`sizeof` function: `fgets(name, sizeof(name), stdin);`
  because it provides a way of limiting the number of characters read to the maximum number that the variable can hold.
- stdin
  is the file to read. In this case the 'file' is the standard input or keyboard. Other files will be discussed later under file input/output.

`fgets()` Example-1

Read in a string and output it's length

```
#include <string.h>
#include <stdio.h>

char line[100]; /* Line we are looking at */

int main()
{
    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    printf("The length of the line is: %d\n", strlen(line));
    return (0);
}
```

Output:

```
Enter a line: test
The length of the line is:   5
```

`test` has only 4 characters - but `fgets` also gets the `\n` character.

`fgets()` Example-2

Read in first and last name, print out full name

```
#include <stdio.h>
#include <string.h>

char first[100];        /* first name of person we are working with */
char last[100];         /* last name                                */
char full[200];         /* full name of the person (computed)       */

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

`fgets()` Example-2 continued

Output of `fgets` - Example 2:

```
Enter first name: John
Enter last name: Lennon
The name is John
 Lennon
```

What happened ? Why is the last name in a new line ?

- The `fgets` command gets the entire line, including the end-of-line. For example, "John" gets stored as {'J','o','h','n','\n','\0'}.
- This can be fixed by using the statement
  `first[strlen(first)-1] = '\0';`
  which replaces the end-of-line with an end-of-string character and so end the string earlier.