Week 5: Strings and Files

CS211: Programming and Operating Systems

12 and 13 Feb, 2020

This week, in CS211:

Strings in C

At the end of the last lecture, we studied ***char***acters in C.
Now we will look at **strings**. Usually, these are thought of a collection of letters/characters that make up a word or a line of text.

The C language **does not actually have a `string` data type**. Instead, it uses arrays of type `char`.

If you make a declaration like:

```
char greeting[20]="Hello.  How are you?";
```

the system stores each character as an element of the array `greeting[]`.

Strings in C

Some Important Points:

1. In the above example we declared the array to be of length 20. Even though the string contains 19 characters, an extra ***string termination character*** `\0` (backslash zero) is added to show where the end of the string is.
2. Spaces or even new-line characters do not terminate a string. They are treated just like other characters.
3. Declarations are the only time we can use an “equals” to assign a value to a string. At all other times, we can modify the string one character at a time:
   `greeting[0]='H'; greeting[1]='e';` ...
4. Better still use `strcpy()` – the “string copy” function:
   `strcpy(greeting, "Not too bad");`

Strings in C

The `strcpy()` is one of a collection of functions for dealing with strings. Its definition is to be found in the `string.h` header file. More of this later...

Example: *Write a function that determines the length of a string.*

Strings in C

00StrLength.c

```c
#include <string.h>
int length(char *);

int main(void )
{
  char greeting[20];

  strcpy(greeting, "Hello. How are you?");

  printf("%s\n", greeting);

  printf("That message was %d chars long.\n",
              length(greeting) );

  return(0);
}
```

Strings in C

00StrLength.c

```
int length(char *str)
{
  int i, length=0;
  for (i=0; str[i] != '\0'; i++)
    length++;
  return(length);
}
```

`string.h`

Useful functions defined in `string.h` include:

strncpy

```
char *strncpy(char *dest, const char *source, int n);
```

Copies at most `n` character from the string in `source` to `dest`. The advantage is that we won't copy more characters to `dest` than is allowed

Example

```
char Code[6], Name[20]="Operating Systems";
strcpy(Code, Name); // Bad!  Unexpected Results
strncpy(Code, Name, 6); // OK.
```

string.h

strcat

strcat(): Con**cat**enate two strings, i.e., append one string onto the end of another. E.g,

```
char message1[30]="Hello.";
char message2[30]=" How are you?";
strcat(message1, message2);
```

Now message1 contains ”Hello. How are you?”;

strcmp

`strcmp(char *s1, char *s2)`: **Comp**ares two stings. It returns an integer:

- 0 if they are the same,
- negative if s_1 is the first alphabetically
- positive if s_2 comes first

Example

```
char Name0[20], Name1[20], First[20];
strncpy(Name0, "Richie", 20);\\
strncpy(Name1, "Dennis", 20);\\

if ( strcmp(Name0, Name1) > 0)
    strncpy(First, Name1, 20);
```

string.h

strlen

strlen Takes a single (pointer to) a string as its argument and returns an integer equal to its **len**gth minus 1. (**Why -1?**).

string.h

strstr

```
char *strstr( char *haystack, char *needle);
```

Searches for the first occurrence of the string `needle` in `haystack`.
It returns a pointer to the start of the matching substring.
Moreover, if `needle` is ***not*** found in `haystack` it returns NULL.

Example:

```
if (strstr(Code, "CS") != NULL)
   printf("%s is a CS course\n", Code);
```

String Output

You all know how to use `printf()` with strings:

```
printf("%s%s\n", "Good morning ", name);
```

or

```
printf("%s%8s\n", "Good morning ", name);
```

In the second example the *field width* specifier is given. This causes the second string to be "padded" so that it takes up a total of 8 spaces. This is useful for tabulated output.

One could also use `puts()`: this prints the contents of a string followed by a new-line character.

String Input

Input is a more complicated issue, but there are three basic methods:

- `scanf("%s", name);` reads a the next “word” from the input buffer (usually the key board) and stores it in the array `name[]`. A word is a sequence of characters that does not include a space, tab or newline character.
- to get more control of the input, you could use `getchar()` within a loop:

```
printf("What is your name? ");
for (i=0;
    (myname[i] = getchar()) != '\n';
    i++);
myname[i]='\0';
```

String Input

- `gets(string)`: this reads a line a input and stores it all (except the `'\n'`) in the array pointed to by `string`. This would be very useful, except that `gets()` is known to be buggy and is best avoided.

From the Linux manual page from `gets()`:

```
BUGS
Never use gets().  Because it is impossible to tell
without knowing the data in advance how many chars
gets() will read, and because gets() will continue to
store characters past the end of the buffer, it is
extremely dangerous to use.  It has been used to
break computer security.  Use fgets() instead.
```

String Input

- `fgets(string, n, stdin)`: reads in a line of text from the keyboard (standard input) and stores at most `n` characters in array `sting`. The new line charater is stored.

Which ever you use is a matter of choise. My preference is always to write functions that use `getchar()` and related functions, particularly if reading from a file.

Multidimensional Arrays

If an array (particularly of integers or floats) is like a mathematical vector, then how do we define a matrix?

A matrix is a two-dimensional array. For example, to declare a 3×4 matrix of floats, we would use the syntax:

```
float A[3][4];
```

So

$$A = \begin{pmatrix} A[0][0] & A[0][1] & A[0][2] & A[0][3] \\ A[1][0] & A[1][1] & A[1][2] & A[1][3] \\ A[2][0] & A[2][1] & A[2][2] & A[2][3] \end{pmatrix}$$

In general an $n \times m$ array is declared as

```
float A[n][m];
```

Multidimensional Arrays

If a program has the line:

`int A[3][4];`

What really happens is that the system creates **three** arrays, each of length **four**. More precisely, it

- declares 3 pointers to type `int`: `A[0]`, `A[1]`, and `A[2]`,
- space for storing an integer is allocated to each of the addresses `A[0]`, `A[0]+1`, `A[0]+2`, `A[0]+3`, `A[1]`, `A[1]+1`, . . . , and `A[2]+3`.

. . . .

This means that if `A[][]` is declared as a two-dimensional 3×4 array, then the following are equivalent:

- `A[1][2]`
- `*(A[1] + 2)`
- `*( *(A + 1) + 2)`
- `*( &A[0][0] + 4 + 2)`

Multidimensional Arrays

01Matrix.c

```c
#include <stdio.h>
int main(void )
{

  int A[3][4]={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

  printf("A[1][2] = %d\n", A[1][2]);
  printf("*(a[1]+2) = %d\n", *(A[1] + 2));
  printf("*(*(A+1)+2) = %d\n", *( *(A + 1) + 2));
  printf("*(&A[0][0] + 4 + 2) = %d\n",
    *( &A[0][0] + 4 + 2));

  return(0);
}
```

Multidimensional Arrays

In another example , we'll sum all the entries of a 3×4 array.

02Sum_a_matrix.c

```
#include <stdio.h>

int sum(int A[][4]);

int main(void )
{
  int n;
  int A[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

  n = sum(A);

  printf("Sum of the entries in A is %d \n",n);
  return(0);
}
```

Multidimensional Arrays

02Sum_a_matrix.c

```c
int sum(int A[][4])
{
  int i,j, ans=0;
  for (i=0; i < 3; i++)
    for (j=0; j< 4; j++)
      ans += A[i][j];

  return(ans);
}
```

Important: Notice that this function is defined only for arrays of size 3×4. Even if we passed n and m as arguments to the function, we would still have to declare that a has 4 columns.

Multidimensional arrays often occur when dealing with arrays of strings.
Recall that in C, a string (collection of characters) is stored as a `char` array.

```
char Name[20]="Ada Lovelace";
```

This means that we have declared `Names` to be an array of 15 characters:

- `'A'` is stored in `Name[0]`
- `'d'` is stored in `Name[1]`
- `'a'` is stored in `Name[2]`
- ...
- `'c'` is stored in `Name[10]`
- `'e'` is stored in `Name[11]`
- and `'\0'` is stored in `Name[12]`.

The remaining components, `Name[13]`, ..., `Name[19]` are unused.

If a single string is stored as a character array, then an array of strings is an ***Array of Arrays of chars***, more often called a ***two dimensional array***.

Example

```
char Names[10][20];
strcpy(Names[0],"A. Lovelace");
strcpy(Names[1],"C. Babbage");
...
strcpy(Names[8],"D. Richie");
strcpy(Names[9],"K. McNulty");
```
[a]

[a] For more about Donegal's greatest computer scientist, see https://en.wikipedia.org/wiki/Kathleen_Antonelli

We can think of this as a matrix, and visualise it as

	0	1	2	3	4	5	6	7	8	9	10	11
Name[0]	A	.		L	o	v	e	l	a	c	e	\0
Name[1]	C	.		B	a	b	b	a	g	e	\0	-
	⋮				⋮				⋮			⋮
Name[8]	D	.		R	i	c	h	i	e	\0	-	-
Name[9]	K	.		M	c	N	u	l	t	y	\0	-

. . . .

Clearly there is some waste of memory space. On another day, we might study the use of “ragged arrays” can avoid this.

Files

Most useful programs obtain their input from a **file**, and store their output to a file.

For example, in next week's lab (TBC) you'll develop a crossword helper that uses data stored in a file.

Further details can be found in Chap. 22 of King's "***C Programming***" or Chap 11 of Kelley and Pohl's "***A Book on C***".

Taking input from a file is not much different that taking input from the keyboard. All we do is:

1. Declare an identifier for the file, (`FILE *`)
2. open the file, (`fopen`)
3. read from it,
4. close the file. (`fclose`)

Declaring a ***File Identifier*** is easy:

```
FILE *datafile;
```

So `datafile` is now a pointer that we can associate with a file or, more generally, a `stream`.

```
fileptr = fopen(char *FileName, char *Mode);
```

The `fopen()` function is used for `f`ile `open`ing. It takes two arguments: the `name` of the file to open and the ***mode*** it will operate in. A file pointer is returned.

The most important modes for file operation are `r`eading and `w`riting, but there is also `a`ppending.

```
fileptr = fopen(char *FileName, char *Mode);
```

Read mode: "r"

Use `fopen(FileName, "r")` to open a file that we want to `read` from. It is assumed that the file already exists. If it doesn't, `NULL` is returned.

Example

```
FILE *infile;
infile = fopen("OldFile.txt", "r");
```

Write mode: `"w"`

Use `fopen(FileName, "w")` to open a file we want to write to. If the file does **not** already exist, it is created. If it is already in the file system, the contents are deleted.

Example

```
FILE *outfile;
outfile = fopen("NewFile.txt", "w");
```

There is also ***append*** mode: `"a"`, used to to append data to end of the file. The file is opened in **write** mode, but new data is added to the end, i.e., its existing contents are not overwritten.

In our examples, we assume that

- The we only want to read from the file.
- That we know its name in advance.

So our code includes

```
FILE *fileptr;
fileptr=fopen("list.txt", "r");
```

If the file can't be opened, `NULL` is returned.

When we are done, we should close the file

```
fclose(fileptr);
```

Example

Give a segment of code that prompts the user for name of an input file, and opens it in `r`ead mode. If a file *cannot* be opened, an error should be returned.

```
int main( void)
{
  char infilename[20];
  FILE *infile;

  printf("Enter file to read from: ");
  scanf("%s", infilename);
  infile=fopen(infilename, "r");

  if (infile == NULL)
  {
    printf("Error: couldn't open for reading");
    return (EXIT_FAILURE);
  }
  else
     printf("Opened %s for reading\n", infilename);
```

Apart from `fopen` and `fclose`, the important functions for manipulating files are

- **Reading:** `fgetc` and `fgets` (also: `fscanf`)
- **Writing:** `fputc`, `fputs` and `fprintf`
- **Check and change file counter:** `rewind`, but also `ftell` and `fseek`.

Reading from a file

There are quite a number of functions for reading data from a file. We'll look at two functions: `fgetc()` and `fgets()`

- `fgets(string, n, fileptr)`
  reads in a line of text from the *fileptr* stream.
  and stores at most `n-1` characters in the array *string*.
  Reading stops after an EOF or a newline. If a newline is read, it is stored in the string. A `\0` is stored after the last read character.
- `fgetc(fileptr)`
  reads and returns the next character in the file. If the end of the file is reached, `EOF` is returned.

Also: `fscanf(fileptr, "%s", CharArray);`
works rather like `scanf()` except that the input stream is `fileptr` rather than *stdin*.

Example 1: Using `fgets`

Example 1: Write a function that counts the number of lines in a file using `fgets()`

Example 1: Using fgets

03CountLinesWithfgets.c

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

int file_length(FILE *);

int main( void)
{
  char FileName[30];
  FILE *file;

  strcpy(FileName, "03CountLinesWithfgets.c");
  file=fopen(FileName, "r");

  printf("%s has %d lines\n", FileName,
    file_length(file));
  return(EXIT_SUCCESS);
}
```

Example 1: Using `fgets`

03CountLinesWithfgets.c

```c
int file_length(FILE *file)
{
  int lines;
  char dummy[100];

  rewind(file);

  lines=0;
  while( fgets(dummy, 100, file) != NULL)
    lines++;

  rewind(file);

  return(lines);
}
```

Example 2: Using `fgetc`

We'll redo this example but using `fgetc`. It reads one character at a time so we'll just count the number of times a newline is read. Note that `EOF` — `End of File` — is returned when we try to read beyond the end of the file.

Example 2: Using `fgetc`

04CountLinesWithfgetc.c

```
int file_length(FILE *file)
{
  int lines;
  char c;

  rewind(file);

  lines=0;
  do {
    c = fgetc(file);
    if (c == '\n')
      lines++;
  } while(c != EOF);

  rewind(file);
  return(lines);
}
```