GNG1106
Fundamentals of Engineering Computation

Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

Outline

- Unlike other high-level languages, standard C does not have a data type specific for character strings.
- A string in C is implemented as an array or dynamically allocated array of char-typed values. Example: The string "Pie" is stored as a char array `{'P', 'i', 'e', '\0'}`.
- Every string must end with the Null Character `'\0'`, which serves to indicates the end of the string.
- Null character `'\0'` is not the same as `NULL` (the null address)!
- If a string has N characters, the char array storing it needs to have length at least N+1; the additional array element is for storing the null character.
- In C programming, a string is written as a sequence of characters enclosed in double quotes, E.g., `"Pie"`, not `'Pie'`.
- A string is allowed to include blank spaces and escape characters, for example, `"my apple pie is\ngood"` is a valid string.

String Declaration and Assignment

- `char s1[256];`
  This declares a char array `s1`, which can be used to store a string of up to 255 characters.
- `char s2[10]={'P', 'i', 'e', '\0'};`
  This declares a char array `s2` (which can be used to store a string of up to 9 characters) and fills it with string `"Pie"`.
- `char s3[]="Pie";`
  This declares a char array `s3` and fills it with string `"Pie"`. The length of array `s3` is 4, which can be used to store any string of up to 3 characters.
- One can also declare a pointer to char, `malloc` memory for it, and use it as a char array to store a string.

Printing Strings

- A string can be printed on the screen using `printf` with the "`%s`" conversion specifier.
- It will stop printing out characters when it hits the null character.
- Examples
  - ```
    char s1[256]="hello";
    printf("%s", s1);
    ```
    This prints `hello`.
  - ```
    char s1[256]={'h', 'e', 'l', 'l', 'o', '\0', 'x'};
    printf("%s", s1);
    ```
    This also prints `hello`.

Highlight

All standard C functions for manipulating strings use `'\0'` as the "the end of string" signal. If `'\0'` is missing, they may continue processing the char array beyond its limit and cause strange behaviour.

- C provides various ways to read in strings from the keyboard.
- `char text[256];`
  `scanf("%s", text);`
  - `scanf` will read characters until the next space, carriage return (i.e., the ENTER key) or EOF (a signal invoked by keying in Ctrl-D, Ctrl-Z, or Ctr-X, etc, depending on OS).
- `char text[256];`
  `fgets(text, 100, stdin);`
  - `fgets` reads in a line of characters up to a certain length (in this, 100) from the keyboard and stores them in an array (in this case, `text`).
- But certain issues exist (similar to those with `scanf("%c", &x);`)
- There is also the `gets` function, which is very unsafe and obsolete.

- The `getline` function reads the entire line of input from the keyboard until (and including) the ending ENTER (`'\n'`).

```
char *s;
/* s should be declared as a pointer to char
rather than a char array */
int nBytes=20;
int nBytesRead;
s=(char *)malloc(nBytes);
nBytesRead=getline(&s, (size_t *)&nBytes, stdin);
```

- The `getline` function in the code above tells the program to read in an entire line from the keyboard and store it in the memory pointed by `s` which has been allocated `nBytes` bytes.
- If reading is successful, the return value of `getline` is the number of bytes read, and the read string is stored in the memory block with address `s`. If reading is unsuccessful, `getline` returns -1.
- If the allocated memory for `s` is not enough, `getline` automatically re-allocates memory for `s` to store the read string.
- For more information, see `http://crasseux.com/books/ctutorial/getline.html`

Standard C functions that manipulates strings (declared in `string.h`)

- `strcat` - concatenate two strings
- `strchr` - string scanning operation
- `strcmp` - compare two strings
- `strcpy` - copy a string
- `strlen` - get string length
- `strncat` - concatenate one string with part of another
- `strncmp` - compare parts of two strings
- `strncpy` - copy part of a string
- `strrchr` - string scanning operation

See `https://en.wikibooks.org/wiki/C_Programming/Strings`

Standard C functions that manipulates characters (declared in `ctype.h`)

- `int isalnum(int c);`
  The function returns nonzero if c is alphanumeric
- `int isalpha(int c);`
  The function returns nonzero if c is alphabetic only
- `int iscntrl(int c);`
  The function returns nonzero if c is a control chracter
- `int isdigit(int c);`
  The function returns nonzero if c is a numeric digit
- `int isgraph(int c);`
  The function returns nonzero if c is any character for which either isalnum or ispunct returns nonzero.
- `int islower(int c);`
  The function returns nonzero if c is a lower case character.

- `int isprint(int c);`
  The function returns nonzero if c is space or a character for which isgraph returns nonzero.
- `int ispunct(int c);`
  The function returns nonzero if c is punctuation
- `int isspace(int c);`
  The function returns nonzero if c is space character
- `int isupper(int c);`
  The function returns nonzero if c is upper case character
- `int isxdigit(int c);`
  The function returns nonzero if c is hexa digit
- `int tolower(int c);`
  The function returns the corresponding lowercase letter if one exists and if isupper(c); otherwise, it returns c.
- `int toupper(int c);`
  The function returns the corresponding uppercase letter if one exists and if islower(c); otherwise, it returns c.

Outline

Files

- Files are used for permanent (or relative longer term) data storage.
- Files are saved on a secondary storage device (e.g. a hard drive/USB stick/CD etc.)
- We have been using files.
  - `myAssignment.pdf`
  - `myProgram.c`
  - `myProgram.exe` (on Windows) or myProgram (on Linux/MacOS) compiled from `myProgram.c`
  - CodeBlocks, etc.
- A file is essentially a sequence of bytes stored on a secondary storage device.
- The OS maintains the file system (as a tree of directories/folders in which files reside) and allows programs to access the files.

ASCII Files

- In an ASCII file, each byte is the binary ASCII code for a character.
- An ASCII file can be viewed directly using a text editor (e.g. notepad on Windows, vi/vim/emacs/pico/... on linux/MacOS) or any program that can display ASCII files (e.g. cat command on linux/MacOS).
- A file containing a C code we write is actually an ASCII file.
- ... play with some text editor and text display program on command line ...

FILE type and Stream

- C provides a data type called FILE, which is in fact a structure type defined in `stdio.h` and "`typedef`-ed" as `FILE`
- An instance (i.e., variable) of FILE type is called a stream.
- A stream is a high-level concept, which is a block of memory serving as a "communication channel" or "conveyor belt" between the program and a file/physical device (e.g., keyboard, screen, hard drive, port, etc)
  - `stdin`: "standard input stream" declared globally in `stdio.h`, which is the pointer to the stream between the program and the keyboard
  - `stdout`: "standard output stream" declared globally in `stdio.h`, which is the pointer to the stream between the program and the screen
- When accessing a file on the disk, a stream must be "opened" (i.e., a block of memory must be allocated to store the stream).

- The `FILE` type has been defined to have a sophisticated set of members, including for example,
  - the memory block needed for transporting data
  - the current access position in the stream
  - ...
- C provides a family of standard functions for manipulating `FILE`-typed variables, i.e., streams
- We will only access a stream through these functions without being concerned with the low-level (i.e., internal) details of the stream.
- These functions allow us to access a file, namely, read or write a file.

Opening and Closing a Stream

- Before accessing a file we must "open" a stream.
- After accessing the file, we must "close" the stream.
  - Otherwise memory leak may occur and file integrity is at risk!

```
FILE *fp; // declare a pointer to a stream
fp=fopen(fileName, mode)
// open the stream and let fp point to it
if (fp!=NULL)
{
     /* access the file via accessing the stream
*/
     ...
    fclose(fp); // close the stream
}
```

fopen

- `fileName` is the string containing the file name, e.g. `"myData.txt"`
- `mode` is a string (not a character) indicating the "mode" (i.e., purpose) of accessing the file
  - `"w"`: write an ASCII file (create the file if it does not exist, or ERASE an existing file with the given name and write to it)
  - `"r"`: read an ASCII file
  - `"a"`: append to an ASCII file
  - a few other modes for accessing an ASCII file:
    - `"w+"` for write and then read
    - `"r+"` for read and then write
    - `"a+"` for read and append
  - `"rb`: read a binary file
  - `"wb"`: write a binary file
- The function `fopen` allocates memory for a stream. If the allocation is successful, it returns the address of the memory, otherwise it returns the `NULL` pointer.

fclose

- The function `fclose` has return type `int`.
- `fclose(fp)` "closes" the stream, i.e., releasing the memory storing the stream (which is pointed to by `fp`).
- `fclose` returns 0 if it has successfully closed the stream, or `EOF` (usually having value -1) if an error has occurred.
- A stream that is not closed by `fclose` will only get closed when the program exits.

Highlight

Call `fclose` to close a stream as soon as it is no longer needed!

Write to ASCII FIle: `fprintf`

```
fprintf(fp, "%d%f\n", intVar, floatVar);
```

- Similar input parameters as `printf` except an additional parameter, the FILE pointer, included as the first parameter
- The function `fprintf` is used to write formatted text into a stream (opened with mode `"w"`) pointed to by input ASCII FILE pointer. The text written to the stream is then stored to the file (for which the stream was opened) by the OS,
  - When passing `stdout` as the FILE pointer to `fprintf`, the function behaves the same way as `printf`.
- `fprintf` returns an int-typed value equal to the number of characters successfully written to the file.
- It returns a negative number if a writing error is encountered.
- `fprintf` uses the same conversion specifiers as `printf`.

Read from ASCII FIle: `fscanf`

```
fscanf(fp, "%d%f", &intVar, &floatVar);
```

- Similar input parameters as `scanf` except an additional parameter, the FILE pointer, included as the first parameter
- The function `fscanf` is used to read formatted text from a stream (opened with mode `"r"`) pointed to by input ASCII FILE pointer. The content of the stream is the content of the file (for which the stream is opened).
  - When passing `stdin` as the FILE pointer to `fscanf`, the function behaves the same way as `scanf`.
- `fscanf` returns an int-typed value equal to the number of items successfully read from the file.
- It returns EOF if there is a reading error or the reading reaches the end of the file (i.e., there is nothing to read)
- `fscanf` uses the same conversion specifiers as `scanf`.

Other Standard C Functions for ASCII File I/O

- `fputc`: write a character to a stream
- `fputs`: write a string to a stream
- `fgetc`: read a character from a stream
- `fgets`: read a string.from a stream
- `getline`: read an entire line from a stream

getline

- The `getline` function reads the entire line of input from an opened stream (say, the stream pointed to by `fp`) until (and including) the ending ENTER (`'\n'`).

```
char *s;
/* s should be declared as a pointer to char
rather than a char array */
int nBytes=20;
int nBytesRead;
s=(char *)malloc(nBytes);
nBytesRead=getline(&s, (size_t *)&nBytes, fp);
```

- The `getline` function in the code above tells the program to read in an entire line from the stream and store it in the memory pointed by `s` which has been allocated `nBytes` bytes.
- If reading is successful, the return value of `getline` is the number of bytes read, and the read string is stored in the memory block with address `s`. If reading is unsuccessful, `getline` returns -1.
- If the allocated memory for `s` is not enough, `getline` automatically re-allocates memory for `s` to store the read string.
- For more information, see `http://crasseux.com/books/ctutorial/getline.html`

How do we know if we have read to the end of a file?

1. When creating the file, add a "header" at its beginning to indicate where its end is.
   - e.g. in the first line, indicate the total number of lines
2. When creating the file, add an "ending signal" at the end of the file.
   - e.g. in a file containing positive integers, add a negative integer in the end.
3. Test the return value of the reading functions
   - `fscanf`, `fgetc` and `getline` returning EOF
   - `fgets` returning NULL
4. Use the `feof` function
   - `feof(fp);`
   - It returns TRUE if and only if the file position pointer is at the end of the file

rewind

- To re-read a file from the beginning, we must first ensure that the "file position pointer" is re-positioned at the beginning of the file.
- The file position pointer points to the location or byte the file (in fact, stream) where the next read or write operation will occur.
- The C standard function `rewind` (declared in `stdio.h`) re-positions the file position pointer to the beginning of the file.

```
rewind(fp);
```

Programming Example

1. Write a program that asks the user to enter a list of numbers. The user will first specify the length of the list and enter the numbers one by one. The program then write the numbers into a file, called "data.txt"
2. Write another program that prints the content of "data.txt"

Coding Demonstration

https://github.com/hjleed/GNG1106_Archive/tree/main/week11_codes