Text Output and Input; Redirection

see K&R, chapter 7

Simple Output
– puts(), putc(), printf()

Simple Output

- The simplest form of output is ASCII text.
- ASCII occurs as single characters, and as null-terminated text strings – arrays of characters with an ASCII 0 at the end.
- Functions write to *stdout*, or to a file specified by a file handle (FILE * type).

String output Functions – Prototypes

- int puts(char *s* []) ;
  - *s* is the null-terminated text string to be printed
  - *A newline is printed after the string.*
- int fputs(char *s* [], FILE *fileptr) ;
  - *s* is the null-terminated text string to be printed
  - *No* newline at end – string must include it if desired.
  - "fileptr" is the destination
    - e.g. stdout
- These functions return a non-negative number, or EOF to indicate an error.

puts() and fputs() – Simple Examples

```
1  /* minimal hello example */
2  #include <stdio.h>
3
4  int main(int argc, char**argv)
5  {
6      puts("hello, world");
7      puts("goodbye");
8
9      fputs("hello, world", stdout);
10     fputs("goodbye", stdout);
11 }
12
```

(The String Argument to an Output Function)

- The prototype for "fputs()" can be written:
  - int fputs(char *s* [], FILE *fileptr) ;
- Alternatively:
  - int fputs(char **s*, FILE *fileptr) ;
- The two forms of "s" are equivalent:
  - char *s* []
  - char **s*
- …both refer to a char array of unknown length
- But they are *not* the same when creating a variable!

Character Output – Function Prototypes

- int fputc(int c, FILE *fileptr);
  - write a single character to a file handle
- int putc(int c, FILE *fileptr);
  - equivalent to fputc()
- int putchar(int c);
  - equivalent to fputc(c, stdout)
- These functions return the character that was written, or EOF to indicate failure.

Another Alternative – printf()

- `printf()` is another output function
  - Much more versatile
  - More complex
- It can be used like `puts()`, but it won't add a newline.
- int printf(char *s* []) ;
  - Returns the number of characters printed, or a negative number on error.
- There is much more to `printf()` than this, but it can be used this way.

What if you want numeric output?

- The numbers to be displayed must be converted to text strings.
  - `printf()` does this
- More on this later.
  - but here's a taste:

```
printf( "%d", integer);

printf( "%f", float_or_double);
```

Simple Input – fgetc(), fgets()

Simple Input

- A single ASCII character is the simplest input.
- **fgetc()**, **getc()**, **getchar()** functions read and return a character from stdin or an opened file.
  - **ungetc()** function allows inspecting a character "before" getting it
- Text strings can be built from individual characters, or read in all at once.
- **fgets()** function reads input text string from stdin or an opened file.
  - A destination character array *must* first be created to hold the string.

Character Input Prototypes

- int fgetc(FILE *fileptr);
  - read a single character from a file handle
- int getc(FILE *fileptr);
  - equivalent to fgetc()
- int getchar(void);
  - equivalent to fgetc(stdin)
- int ungetc(int c, FILE *fileptr);
  - pushes character c back into the input stream's buffer (but not back into stdin or the file)
- These functions return the character read, or EOF to indicate failure.

fgetc() example - A

- **fgetc()** takes a single character

but

No input is seen until <Enter> is pressed
  - operating system limitation

```c
/* fgetc() example 1
* 2018-06-24
*/
#include <stdio.h>

int main(int argc, char **argv)
{
   char c;

   fputs("Enter one or more characters: ", stdout);
   c = fgetc(stdin);
   printf("first char-->%c<--\n", c);

   return 0;
}
```

- What happens to excess input characters ???

fgetc() example - B

- **fgetc()** takes only the *first* character
- Read additional characters (and discard) until the "newline" is found, or End-Of-Input is reached
  - <Enter> key generates a newline char

```c
/* fgetc() example 2
* 2018-06-24
*/
#include <stdio.h>

int main(int argc, char **argv)
{
   char c;

   fputs("Enter one or more characters: ", stdout);
   c = fgetc(stdin);
   printf("first char-->%c<--\n", c);

   // Discard excess input characters
   do {
      c = fgetc(stdin);
      printf("    %02x -->%c<--\n", c, c);
   } while (c != '\n' && c != EOF);

   return 0;
}
```

Text-string input – first, a word on text strings

- There is no "text string" type!
  - Newer, bigger languages add the type
- C "text string" is an *array* of characters
  - declared as e.g.
    char mystring [20] ; // 20-character array
- C terminates text strings with a NULL character (ASCII 0) that occupies one space
  - Terminating NULL is stored in the array along with the other characters
  - So mystring can only hold 19 "usable" characters
- *Once created, an array cannot be resized*
  - Instead, copy into a new array of different size

String Input Function – Prototype

- int fgets(char *s, int size, FILE *fileptr) ;
  - s points to the destination text string
  - size is the maximum #of characters to accept
    - accepts one less, to leave space for terminating null
  - fileptr is the source
    - e.g. stdin
  - returned text *includes* any newline at end

- Remark: Don't use this deprecated function:
  int gets(char *s) ;
  - Final newline is *removed*
  - NO checking for too many characters!
    - Don't use gets()! Use fgets() instead

Text strings as parameters in fgets( ), fputs( )

- These are both parameters that are strings
  - char the_string[ ]
  - char *the_string
  - these are completely equivalent
  - the length of the argument is unknown, and may change from one function-call to the next
- Both forms match a supplied character array of any length
  - fputs() expects a NULL at the end
- (this is something *different*: char **argv)

What if you need numeric input?

- C has functions that convert a string into a number.
- Apply as needed to the input from fgets().
- No error-checking – these expect the string to "look like" a proper number.
- Two families:
  - "ASCII-to-number" functions are simple; do no error-checking
  - "STR-to-number" functions are more versatile; more effort to use

ASCII-to-number Functions

- int atoi(char str[]);
  - converts ASCII to a signed integer
- long atol(char str[]);
  - converts ASCII to a signed long integer
- long long atoll(char str[]);
  - converts ASCII to signed long long integer
  - non-standard
- double atof(char str[]);
  - converts ASCII to a double-precision floating-point number
  - used for both floats and doubles

STR-to-number Functions

- long strtol(char str[], char **endptr, int base);
  - converts string to signed long integer, assuming the given base (0 means "try to guess")
  - endptr is set to point to where the conversion ended in the supplied string
- unsigned long strtoul(char str[], char **endptr, int base);
  - like strtol(), but converts to unsigned long
- float strtof(char str[], char **endptr);
  double strtod(char str[], char **endptr);
  - converts string to float or double
  - endptr as in strtol()

Direct Numeric Input

- The "scanf()" function provides formatted input
  - Compare to "printf()"
- Examples:

```
int my_int;
float my_float;

scanf( "%d",  & my_int );
scanf( "%f",  & my_float );
```

- The ampersand "&" is *required* on the numeric variables that receive the inputs

Command-Line Redirection

- Standard I/O functions read from/write to the *console*
  - i.e. the screen and keyboard
- These can be redirected from/to text files
  - Use "**<**" and "**>**" for input/output redirection
- Input:
  - `./myprog < datafile.txt`
  - All standard inputs come from "datafile.txt"
  - End of file causes EOF
- Output:

  `./myprog > results.txt`
  - All standard outputs get saved in "results.txt"

Finally

- What if you *DON'T* use input redirection, but your program is looking for the EOF character?
  - Using up the file generates EOF, but there is no file – keyboard input can continue indefinitely
- Generate an EOF on the keyboard by entering "^D" (ctrl-D) on a line by itself
  - This is *different* from ^D stored in a file, where it is a carriage-return character
- Windows/DOS: use ^Z instead of ^D

fgets() versus gets()

- First block: **fgets()** accepts limited-length text string
- do-while() loop consumes, displays excess characters
- Second block: **gets()** accepts as many characters as offered
  - buffer overflow possible
  - Modern gcc compiler includes error-checks

```
/* fgets() versus gets()
* 2018-06-24
*/
#include <stdio.h>

#define MAXBUFFER 6
int main(int argc, char **argv)
{
   char junk;
   char string[MAXBUFFER];    // Holds MAXBUFFER-1
                              // characters, plus NULL
   printf("Begin: input to fgets(): ");
   fgets(string, MAXBUFFER, stdin);
   printf("dest ->%s<-\n", string);
   do {
      junk = getchar();   printf("  %c\n", junk);
   } while (junk != '\n' && junk != EOF);

   printf("Begin: input to gets(): ");
   gets(string);
   printf("dest ->%s<-\n", string);
   do {
      junk = getchar();   printf("  %c\n", junk);
   } while (junk != '\n' && junk != EOF);

   return 0;
}
```

fgets() versus gets() - analyzed

fgets() limits string size

do-while() absorbs excess

gets() overfills string

```c
/* fgets() versus gets()
* 2018-06-24
*/
#include <stdio.h>

#define MAXBUFFER 6
int main(int argc, char **argv)
{
   char junk;
   char string[MAXBUFFER];    // Holds MAXBUFFER-1
                              // characters, plus NULL
   printf("Begin: input to fgets(): ");
   fgets(string, MAXBUFFER, stdin);
   printf("dest ->%s<-\n", string);
   do {
      junk = getchar();   printf("  %c\n", junk);
   } while (junk != '\n');

   printf("Begin: input to gets(): ");
   gets(string);
   printf("dest ->%s<-\n", string);
   do {
      junk = getchar();   printf("  %c\n", junk);
   } while (junk != '\n');

   return 0;
}
```

fgets-gets and Redirection

examine input

redirected input

fgets( ) limits string size

gets( ) tries to accept everything

```
$
$ cat twolines.txt
abcdefghijklm
ABCDEFGHIJKLM
$
$ ./fgets-gets  <  twolines.txt
Begin: input to fgets(): dest ->abcde<-
  f
  g
  h
  i
  j
  k
  l
  m

Begin: input to gets(): dest ->ABCDEFGHIJKLM<-
  
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
$
```