Input/Out in C

Streams

• Communication channels between files (devices) and programs
• No distinction between files and i/o devices
• Used to perform input and output
• The name stream comes from a “stream of bytes” – the way C looks at files and devices

Working with one character at a time

• **getchar** – Get a character from the user
  ```c
  int getchar();
  ```
  • Reads a single character at a time from stdin
  • Returns an unsigned char typecast into an int or EOF in the event of an error
  • To read a character in a variable, use the following code:
    ```c
    char c;
    c = getchar();
    ```
  • Use the function inside a loop to read multiple characters

• **putchar** – Put a character on the screen
  ```c
  int putchar ( int ch );
  ```
  • Display a single character on stdout
  • Returns the output character as an unsigned char typecast into an int or EOF in the event of an error
  • A character can be displayed on the screen by using the following code
    ```c
    char c;
    putchar ( c );
    ```
  • Use the function inside a loop to display a sequence of characters

Working with a line of input at a time

• **gets** – Get a line of characters from the user
  ```c
  char * gets ( char * str );
  ```
  • Reads a line from stdin into the buffer pointed to by str, until a terminating newline or EOF, which is replaced by a null byte
  • To read a line, use the following code:
    ```c
    char str[80];
    gets ( str );
    ```
  • Returns str on success, and NULL on error or when end of file occurs while no characters have been read
— **Caution:** `gets` has been deprecated; don’t use it
   * You cannot tell how many characters `gets` will read
   * It will continue to store characters past the end of buffer, making the code vulnerable to buffer overflow
   * It has been used to break computer security
   * It is preferable to use `fgets` instead

• **fgets** – Get a line of characters from a file stream
  
  — `char * fgets ( char * str, int size, FILE * stream );`
  — Reads in at most `size - 1` characters from `stream` and stores them into the buffer pointed to by `str`, until a terminating newline or EOF
  * If a newline is read, it is stored into the buffer
  * A terminating null byte is stored after the last character
  — To read a line from `stdin`, use the following code:
    ```c
    char str[80];
    int sz = 80;
    fgets ( str, sz, stdin );
    ```
  — Returns `str` on success, and NULL on error or when end of file occurs while no characters have been read

• **puts** – Display a line of characters from a file stream
  
  — `int puts ( const char * str );`
  — Display the string `str` followed by a trailing newline
  — To display a line, use the following code:
    ```c
    char str[80];
    puts ( str );
    ```
  — Returns a non-negative number on success, or EOF on error

**Formatted Input/Output**

**Formatting input with scanf**

• `int scanf ( const char * fmt, ... );`

• `scanf` reads the input from `stdin` and scans it as per the format provided
  — The format string provides directives to process the sequence of input characters
  — If processing of a directive fails, no further input is read and `scanf` returns
  — A sequence of white space characters (space, tab, newline) match any amount of white space in input
  — Any ordinary character (except whitespace or `%`) must match itself in the input

• The format is followed by a number of pointer arguments, appropriate for the value returned by the corresponding conversion specification

• It scans the input using a set of conversion specifiers (all conversion specifiers are preceded by a `%` character)
  — If next item of input does not match the conversion specification, the conversion fails (*matching failure*)
The `%` character may be followed by `*` in which case the input is scanned as directed by conversion specifier but then, discarded

* No corresponding pointer argument is required
* The specification is not included in the count of successful arguments returned by `scanf`

An optional decimal integer specifies the maximum field width

- `%d` Signed or unsigned decimal integer
- `%i` Signed or unsigned decimal, octal, or hexadecimal integer
- `%o` Unsigned octal integer
- `%u` Unsigned decimal integer
- `x` or `X` Unsigned hexadecimal integer
- `%` or `%l` Used with any of the above integer conversion specifiers to indicate a `long` or `short` integer
- `e`, `E`, `f`, `g`, or `G` Floating point number
- `%c` Single character
- `%s` A string, terminated by `NULL`
- `scan set` Scan the string of characters stored in the format
- `%` Skip a percent character in the input

- If the number of conversion specifiers in format exceeds the number of pointer arguments, the results are undefined

**Formatting output with printf**

- Every `printf` statement contains a format string to describe the output format
- A call to the function `printf` has the form

  ```c
  printf ( format_string, arg1, arg2, ... )
  ```

  where zero or more arguments may follow the first argument `format_string`
- The format string must be enclosed in double quotes

**Printing integers**

- Printed by using the conversion specifiers as per the following table

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%d</code></td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td><code>%i</code></td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td><code>%o</code></td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td><code>%x</code> or <code>%X</code></td>
<td>Unsigned hexadecimal integer</td>
</tr>
<tr>
<td><code>%hd</code></td>
<td>Print <code>short</code> integer</td>
</tr>
<tr>
<td><code>%ld</code></td>
<td>Print <code>long</code> integer</td>
</tr>
</tbody>
</table>

- Do not print a negative value with a conversion specifier that expects an unsigned value

**Printing floating-point numbers**
• Can be printed in decimal point notation or exponent notation

\%e or \%E  . Print floating point number in exponent notation
\%f  . Print floating point number in fixed point notation
\%g or \%G  . Print floating point number in fixed point notation (\%f)
if the value does not fit, print it in exponent notation (\%e)
\%Ld  . Print a long double floating point number

Printing strings and characters

• Use the following conversion specifiers

\%c  . Print a character
\%s  . Print a string (array of characters terminated by NULL)

• A string must be terminated by a NULL

• You must enclose a character constant in single quotes and a string constant in double quotes

Printing with field widths and precisions

• An integer inserted between the percent character and the conversion specifier indicates the width of the field used to print the variable value

• All variables printed with a specified field width are right justified

• Precision values can also be specified in the argument list by replacing the field width specifiers with asterisks as follows

  \texttt{printf ( "%*.*f", 7, 2, 98.736 );}

  The result is 98.74 right justified (with two leading spaces)

Using flags in the printf format control string

• Used to supplement the formatting control abilities of printf

• The flags are defined as follows

  - Print the output left-justified
  + Display a plus or minus sign as appropriate
  space  . Print a space before a positive value not printed with the + flag
  #  . Prefix 0 to an octal number
  Prefix 0x or 0X to a hexadecimal number
  Force a decimal point for a floating point whole number
  0  . Pad a field with leading zeros

• You may combine several flags in one conversion specification

Printing escape sequences
• Escape sequences are used to print special characters like newline

\' Single quote character
" Double quote character
? Question mark character
\\ Backslash
\a Audible bell (or visual alert)
\b Backspace
\f Form feed
\n Newline
\r Return to beginning of line
\t Horizontal tab
\v Vertical tab

More input/output functions

• Problems with scanf and solutions

  – scanf creates lots of trouble when reading numbers and characters together, especially in the handling of end of line character
  – From this point onwards, we will not even use it for that reason
  – We’ll use a combination of other routines to get around the problems

    char line[100]; /* Line of keyboard input */

    fgets ( line, sizeof ( line ), stdin );
    sscanf ( line, format, &var1, &var2, ... );