Standard "C" Input/Output

Table of Contents

Output: printf() - syntax & semantics

Syntax

- printf (*control string* , *argument list*)
- Function Prototype: int printf(const char, *format, ...) ;

Semantics

- returns the number of characters that are output or a negative value if an output error occurs.
- Argument list expressions are converted according to the corresponding formatter in the control string.
- Control string formatters are indicated by preceding them with a **%** symbol.
- Argument list expressions and control string formatters are matched on a 1-1 basis.
- Text in the control string that is NOT part of a formatter is output verbatim.

Control String Formatters

char	conversion
%d %i	decimal integer
%u	unsigned decimal integer
%o	unsigned octal integer
%x %X	unsigned hexadecimal integer
%c	character
%s	string or character array
%f	float
%e %E	double (scientific notation)
%g %G	float or double format
%%	outputs a '%' character

Output: printf() - examples

Some Examples

```
printf("The sum of %d, %d, and %d is %d\n", 65, 87,
33, 65+87+33);
      The sum of 65, 87, and 33 is 185

printf("Character code %c has ASCII code %d.\n",
'A', 'A');
      Character code A has ASCII code 65.

printf("Error %s occurred at line %d \n", emsg,
lno);
      Error invalid variable occurred at line 27

printf("Octal form of %d is %o \n", 59, 59);
      Octal form of 59 is 73

printf("Hexadecimal form of %d is %x \n", 59, 59);
      Hexadecimal form of 59 is 3B

printf("Square root of 2 is %f \n", sqrt(2.0));
      Square root of 2 is 1.414214

printf("Square root of 157 is %e \n", sqrt(157.0));
      Square root of 157 is 1.252996e+01

printf("You scored %d out of %d for %d%%.\n",17,
25, 68);
       You scored 17 out of 25 for  68%.
```

Output: printf() - format control

Flag Characters

- Optionally specified between the % and the formatter character
- field width
  - † positive integer giving minimum number of output columns; effect depends upon output type
- precision
  - † period followed by a nonnegative integer giving the minimum number of digits output for (d, i, o, u, x) or the digits to right of decimal point for (e, f) or maximum number of characters for (s)
- alignment
  - † optionally precedes field width; minus sign specifies left-justification, no minus sign specifies right justification
- miscellaneous flags
  - † plus sign optionally preceding field width specifies that a '+' character is prefixed to nonnegative numbers
  - † pound sign optionally preceding field width specifies that a zero be prefixed to octals and a 0X be prefixed to hexadecimals
  - † a zero optionally preceding field width specifies that zeroes are to be used for padding instead of spaces
- longs & shorts
  - † h character optionally preceding field width specifies that a conversion to short int occurs if required prior to output
  - † l or L character optionally preceding field width specifies that a conversion to long int or long double respectively occurs if required prior to output

Screen / Printer Control

Escape Character \

- Non-printable (non-displayable) (control) characters may require use of the escape character (backslash) to specify.

code	control character
\n	new line
\t	tab
\r	carriage return
\f	form feed
\v	vertical tab
\"	double quote
\’	single quote
\\	backslash
\a	alert (bell)
\b	backspace

whitespace characters
blanks, tabs & newlines

- Note: some control character functions may not be supported by particular device drivers (video cards) monitors/printers.

The printf() & scanf() functions are located in the standard I/O library:

#include <stdio.h>

Input: scanf() - syntax & semantics

Syntax

- scanf (*control string* , *argument list*)
- Function Prototype: int scanf(const char, *format, ...) ;

Semantics

- Returns number variables assigned values or **EOF** if an error.
- Argument list expressions are pointer expressions converted according to the corresponding formatter in the control string.
- Standard input file **stdin** (keyboard) supplys the input stream.

Primary differences: printf() / scanf() formatter chars

- %c specifier can match white space characters in the input.
- white space characters in the control string (\t \n) which can match optional white space in the input
- nonwhite space characters (not %), which must match the next input character in the stream
- %n causes no input to occur, scanf stores the number of characters read so far by the current scanf() in the corresponding integer pointer argument
- %% instructs scanf to skip a % sign in the input
- The asterisk character (suppression character *****) preceding a formatter character instructs scanf() to discard the corresponding input data without storing it in a variable
  - e.g., **%*c** would cause scanf() to discard the next input character
- Field width specifiers: scanf() inputs multiple characters
  - e.g., **%s** would cause scanf() to skip white space, read in

Input: scanf() - examples

Some Examples

```
scanf("%d%i%i%i%o%x", &a1, &a2, &a3, &a4, &a5,
&a6);
/* Input echoed as decimal ints */
printf("%d %d %d %d %d", a1, a2, a3, a4, a5, a6);

scanf("%e%f%g", &a1, &a2, &a3);
/* Input echoed as standard reals */
printf("%f %f %f", a1, a2, a3);

scanf("%c%c%s", &c1, &c2, s);
/* Input echo */
printf("\"%c%c  %s\"",  c1, c1, s);
```

Input Data

```
120 -120 0120 0x12 120 12
3.1415 0.31415e+01 31.415e-01
...into that good night
```

Output

```
120 -120 80 18 80 18
3.1415 3.1415 3.1415
".. .into that good night"
```

Input: scanf() - scan sets

Set Input

- Allows for the input of specific sequences of characters

Specification

- Set of characters enclosed in brackets []

Semantics

- Input stream is read (from the current position), for characters in the scan set, every matched character is appended to the input string, readings stops at the first encountered character that is not in the set.

Example

```
scanf("%[.adgihnt  ]", s);
printf("\"%s\"",  s);
```

Input Data

```
...into that good night
```

Output

```
"...int"
```

Inverted Scan Sets

- A circumflex (caret ^) character is placed after the [bracket.
- Input stream is read (from the current position), for characters NOT in the scan set, every non-matched character is appended to the input string, readings stops at the first encountered character that is IN the set.
- e.g., scanf("%[^aeiou]", s);

Considerations: printf() & scanf()

- Complexity
  - printf() & scanf() have options other than those covered herein.
  - knowledge of printf() & scanf() is required to understand legacy **C** code, but should be avoided when possible.

- Pitfalls
  - When reading strings scanf() does not check to determine if the corresponding string variable has enough storage space.
  - It is the programmer's responsibility to ensure that the string variable has enough space allocated.
  - If a string variable does not have the necessary storage, the variable will be filled with input characters, but will not have a null terminating character appended making it an invalid string.

- Common Errors
  - printf() & scanf() do not check the control string until runtime
  - e.g., printf("%d %d \n");
    - † would most likely cause an execution error

  - printf() & scanf() perform no checking of the argument list
  - e.g., scanf("%ld", &c);
    - † attempts to store a long integer into a character
    - † may cause a memory protection fault or crash the system.

Standard “C” Files

- Defined in stdio.h

file	purpose	connection
stdin	**standard input**	**keyboard**
stdout	**standard output**	**monitor**
stderr	**standard error**	**monitor**

- Defines file type: **FILE** (file pointer)

Standard File Functions: fprintf() & fscanf()

- Prototypes:

```
int fprintf( FILE *fp, const char *format, ...);
int fscanf( FILE *fp, const char *format, ...);
```

- Analogous to printf() & scanf()

- Equivalent to standard file I/O:

```
fprintf( stdout , ... );    «=» printf( ... );
fscanf( stdin , ... );      «=» scanf( ... );
```

File Modes

- File access (read, write, append) must be specified for each file:

mode	access
"r"	reading
"w"	writing
"a"	appending
"rb"	binary reading
"wb"	binary writing
"ab"	binary appending
multiple concurrent access	
"*mode* +"	reading & writing

Opening & Closing Files

- Files must be opened before use:

```
FILE *fopen(const char *filename, const char *mode);
```

  - returns a pointer to the opened file if successful or a NULL pointer upon failure.

- Files must be closed after use:

```
int fclose(FILE *stream);
```

  - returns zero if the file is closed successfully or EOF upon failure.

- Files opened for reading & writing should be fixed-length record files for efficiency.
  - Between reads/writes a call to **fflush()** must be made to flush the file buffer or a call to a file position function **fseek()** or **fsetpos()**.

"C" File I/O Calls

Open

```
FILE *ifp, *ofp;

ifp = fopen("input.txt", "r");          // open for reading
ofp = fopen("output.txt", "w");         //open for writing
```

Close

```
int status;
status = fclose(ifp) ;
fclose(ofp);
```

† failure to close a file before program termination may result in loss of data at the end of the file due to the file buffer not being flushed.

Creation & Deletion

- A file that does not exist that is opened for writing or appending will be automatically created.
- A file that does exist that is opened for writing will be automatically overwritten.
- A file that does exist that is opened for appending will be written to after the end of the file.

File Echo Example

Read & echo a file of SSNs and names

```
#include <stdio.h>

main()
{
  int ssn;
  char        name[25];
  FILE        *customers;

  if ((customers = fopen("customers.txt", "r")) ==
NULL)
      printf("* Error: File is inaccessible\n");
  else {
       printf("%-30s%-10s\n", "NAME", "SSN");
      fscanf(customers, "%d%s", &ssn, name);

      while (!feof(customers)) {
              printf("%-30s%-10s\n", name, ssn);
              fscanf(customers, "%d%s", &ssn, name);
      }

      fclose(customers);
  }

  return EXIT_SUCCESS;

}
```

The feof() function is a boolean function that checks end of file status:

int feof(FILE *stream) ;