Computer Programming

Input/output functions

Marius Minea

marius@cs.upt.ro

9 November 2015

All inputs must be checked!

A program will not always receive the data it asks for

User may make *mistakes*, or may be **evil**
$\Rightarrow$ program *must check* that data was read correctly

MUST check *return code* of input function (NOT just value read)

Avoid **overflow** when reading *strings* and arrays
  stop reading when array limit is reached

Buffer overflows *corrupt memory* (program data)
$\Rightarrow$ system is *vulnerable* to **intruder attacks**
Unvalidated input may cause *code injection* (attacker runs code)
  $\Rightarrow$ some of the most **dangerous and costly** errors

A badly written program
An ignorant programmer are *worse* than no program(mer) at all!

Always checking for successful (correct) input!

Reading the desired data might not succeed for two reasons:
  system: no more data (end-of-file), read error, etc.
  user: data not in needed format (illegal char, not number, etc.)

A function can report both a *result* and an *error code* as follows:

1. *expand result datatype* to include error code
   `getchar()` : unsigned char converted to int,
      or `EOF` (-1) which is different from any `unsigned char`

2. return type may have a special *invalid/error value*
   `fgets` returns address where the line was read (first argument)
      or `NULL` (invalid pointer value) when nothing read

3. return *error code* and *store result* at given pointer
   `scanf` *returns no. of items read* (can be 0, or `EOF` at end-of-input)
      takes as arguments *addresses* where it should place read data

Process everything until no more (end of input/error)

Checking for end-of-input explicitly is rarely needed.

The point of processing is to *read data*

  ⇒ thus we must check that data was read successfully:

> while (*read successful*) *use data*

On exit from loop, if `feof(stdin)`, input is finished

else input does not match format ⇒ read next char(s) and report

DO NOT write code of the form

```
        while (!feof(stdin))
            scanf("%d", &n);
```

After last good read (number), end-of-input is not yet reached

unless no more separators (whitespace, incl. newline) after it

  ⇒ next read will not succeed, but is not checked

If read is checked (as it *MUST* be), testing `EOF` is not needed:

```
while (scanf("%d", &n) == 1)
  // process n
```

Check bounds when filling an array

Often, we have to fill an array up to some stopping condition:
  read from input upto a given character (period, \n, etc)
  copy from another string or array

Arrays must not be written beyond their length!

*Loop should **first** test array is not full!*

```
for (int i = 0; i < len; ++i) { // limit to array size
  tab[i] = ...;        // assign with value read
  if (normal stopping condition) break/return;
}
// here we can test if maximal length reached
// and report if needed
```

Text lines can be long. Check if truncated!

```c
#include <stdio.h>
int rdline(char line[], size_t size) {
  if (size-- == 0) return 0;        // keep space for '\0'
  for (size_t i = 0; i < size; ++i) { // only up to size
    int c;
    if ((c = getchar()) == EOF) { line[i] = '\0'; return i; }
    if ((line[i] = c) == '\n') { line[++i] = '\0'; return i; }
  }
  line[size] = '\0'; return -1;      // truncated line
} // why is reading directly into line[i] not good?
#define LEN    82
int main(void) {
  char s[LEN]; int cnt;
  if (cnt = rdline(s, LEN)) { // nonzero, something read
    printf("%s", s);              // print read line
    if (cnt == -1) fputs("\nline truncated\n", stderr);
    else if (s[cnt-1] != '\n') fputs("\nEOF, no \\n", stderr);
  } else fputs("\nnothing read\n", stderr);
  return 0;
}
```

Read text line easily: `fgets`

```c
char tab[80];
if (fgets(tab, 80, stdin)) { /* line has been read */ }
else { /* EOF, nothing read */ }
```

Declaration: `char *fgets(char *s, int size, FILE *stream);`
   (all I/O functions *declared* in `stdio.h`)

Reads up to and including newline \n, max. `size-1` characters, stores line in array s, adds '\0' at the end.

Third parameter to `fgets` indicates the *file* from which to read: `stdin` (`stdio.h`) is *standard input* (keyboard unless redirected)

WARNING!   NO reading without checking!
Check successful return code, anything else is too late!
   `fgets` returns `NULL` if nothing read (end-of-file).
   if successful returns address passed as argument (thus non-null)
⇒ Test *non-null* result to find out if read successful

Read line by line until end of input

```
char s[81];
while (fgets(s, 81, stdin)) printf("%s", s);
```

A line with > 80 chars will be read and printed piecewise (OK!)

More complex: can test if read line was truncated:
```
int c; char s[81];
if (fgets(s, 81, stdin))    // line was read
  if (strlen(s) == 80 && s[79] != '\n' // unfinished
    && ((c = getchar()) != EOF) // EOF not reached
      printf("incomplete line: %s\n", s);
      ungetc(c, stdin);      // put char c back
  } else printf("complete line: %s\n", s);
```

C11 standard removed function ~~gets~~: did not limit size read
⇒ it is impossible to use ~~gets~~ safely
⇒ buffer overflow, memory corruption, security vulnerabilities

Printing a string

```
puts("text; newline will be added");
```
Declaration: `int puts(const char *s);`
prints string s followed by newline \n

```
fputs("text with no newline added", stdout);
```
`fputs(s, stdout);` is like `printf("%s", s);`
  prints string s as is, without additional newline
  stdout is *standard output* (screen unless redirected)
Declaration: `int fputs(const char *s, FILE *stream);`

`puts` and `fputs` return `EOF` on error, nonnegative on success

Formatted output: `printf`

```
int printf(const char* format, ...);
```
functions with variable number of parameters: discussed later

First parameter: the *format string*; may contain:
usual characters (are printed)
format specifiers: `%` and a letter:
%c char, %d, %i decimal, %e, %f, %g real, %o octal, %p pointer,
%s string, %u unsigned, %x heXadecimal

Remaining parameters: *expressions*, their *values* are printed
their number and type must correspond to format specifiers

Result: number of characters printed (usually not used/ignored)

Example:
```
printf("square root of %d is %f\n", 3, sqrt(3));
```

Formatted input: `scanf`

```
int scanf(const char* format, ...);
```

First arg: *string*, with format specifiers (some differences to `printf`!)

> Remaining parameters: *addresses* where to store read values

Need *addresses*, NOT necessarily `&` (one way to get addresses)
DON'T use & for strings: array name IS already its address

Returns number of objects read (assigned) (NOT their value!)
or `EOF` when error/end-of-file *before* anything read

WARNING!   MUST CHECK `scanf` return value!
```
double x; float y;    // CAUTION : %f float %lf double
if (scanf("%lf%f", &x, &y) != 2) { /* handle error */ }
else { /* can use x, y */ }
```

WARNING!   MUST give max. string length in format!
```
char str[30];
if (scanf("%29s", str) != 1) { /* handle error */ }
else { /* word (up to first whitespace) has been read in s */ }
```

NEVER use `%s`: ~~`scanf("%s",...)`~~. Leads to buffer overflow.

Process repeatedly: while read successful

Simplest: *exit program*
  primitive, but *incomparably better than continuing with errors*

`void exit(int status)` from `stdlib.h`   ends program

Can write an error function that prints a message and calls `exit()`

```c
#include <stdio.h>
#include <stdlib.h>
void fatal(char *msg)
{
  fputs(msg, stderr); // to screen unless redirected
  exit(EXIT_FAILURE); // or exit(1)
}
```

We can then use this function for *every* read:

```c
if (scanf("%d", &n) != 1) fatal("error reading n\n");
// got here, use n
```

Good practice: | Always print error messages to `stderr` |
  can separate errors from output (using redirection)

Handling input errors

Often, want to repeatedly read and process. A useful pattern:

   while (*read successful*) *process data*

```
while (fgets(...)) { /*process line */}
while ((c = getchar()) != EOF) { /*process c */}
while (scanf(...) == how-many-to-read) { /* use them*/}
```

On loop exit, may test for EOF (normal read), or (format) error.

CAUTION! scanf *does not consume* non-matching input
$\Rightarrow$ *must consume bad input* before trying again

```
int m, n;
printf("Input two numbers: ");
while (scanf("%d%d", &m, &n) != 2) { // while not OK
  for (int c; (c = getchar()) != '\n';) // skip to end of line
    if (c == EOF) exit(1);             // nothing more, done
  printf("try again: ");
}
// can use m and n now
```

Reading a word (string)

Format letter `s`: for reading a *word* (string WITHOUT whitespace)
  CANNOT read a sentence `"This is a test."`

Arrays are ALWAYS limited!
⇒ **MUST** give max. length (a constant) *between* % and s
one less than array length, `scanf` will add `\0`

```
char word[33];
if (scanf("%32s", word) == 1)
  printf("Word read: %s\n", word);
```

scanf with s format *consumes and ignores* initial *whitespace*:
`\t \n \v \f \r` and space, as checked by `isspace()`

CAUTION! Array names *are addresses*, DON'T use &

CAUTION! Format s reads a *word* (up to whitespace), *not a line!*

scanf: matching the format

Besides format specifiers (%), format string may have *ordinary chars*
- `printf`: printed as such
- `scanf`: *must appear in input*

Example: reading calendar date in `dd.mm.yyyy` format

```
unsigned d, m, y;
if (scanf("%u.%u.%u", &d, &m, &y) == 3)
  printf("read 3 values: d=%u, m=%u, y=%u\n", d, m, y);
else printf("error reading date\n");
```

input `5.11.2013` (with periods!) ⇒ d=5, m=11, y=2013
    see later how to enforce *exactly* 2 or 4 digits

scanf reads until input *does not match* format
Non-matching chars are not read; those variables are not assigned

```
scanf("%d%d", &x, &y);
```
    input: `123A`   returns 1;   x = 123, y: unchanged;   input rest: `A`

```
scanf("%d%x", &x, &y);
```
    input: `123A`   returns 2;   x = 123, y = 0xA (10)

Reading strings with certain characters

allowed characters: between [ ] (ranges: with -)
Reading stops at first disallowed character

```
char a[33]; if (scanf("%32[A-Za-z_]", a) == 1) ...
```
  max. 32 letters and _
```
char num[81]; if (scanf("%80[0-9]", num) == 1) ...
```
  string of digits

WARNING! MUST give max. length between % and [ ]

Reading a string *except for disallowed (stopping) chars*:
like above, but use ^ after [to specify *disallowed* chars

```
char t[81]; if (scanf("%80[^\n.]", t) == 1) ...
```
  reads up to period or newline

WARNING! Format is [ ], NOT with s: ~~%20[A-Z]s~~

Reading a fixed number of chars

One character:
```c
int c = getchar(); if (c != EOF) { /*read OK */}
int c; if ((c = getchar()) != EOF) { /*read OK */}
```

With scanf (use char, not int; useful for arrays)
```c
char c; if (scanf("%c", &c) == 1) { /* read OK */}
```

Reading a *fixed number of chars*:
```c
char tab[80]; scanf("%80c", tab);
```
reads EXACTLY 80 chars, *anything* (including whitespace)
DOES NOT add '\0' at end $\Rightarrow$ can't know if all read

Check how many read by initializing with zeroes and testing length:
(or with %n format, see later)

```c
char tab[81] = "";
scanf("%80c", tab);
int len = strlen(tab); // will be between 0 and 80
```

Whitespace handling in `scanf`

Numeric formats and `s` consume and ignore initial whitespace
  `"%d%d"`   two ints separated and possibly preceded by whitespace

In formats c [ ] [ˆ] whitespace are *normal chars* (not ignored)

A *white space* in the format consumes *any* ≥ 0 whitespace in input
`scanf(" ");` consumes whitespace until first non-space char

`"%c %c"` reads char, consumes ≥ 0 whitespace, reads other char
`"%d %f"` is like `"%d%f"` (whitespace allowed anyway)

CAUTION! `"%d "` : space after number consumes ALL whitespace
  (*including* newlines!)

Consume whitespace, but not newline \n:
`scanf("%*[\t\v\f\r ]");`
  blue* modifier means consume and ignore (no address is given)

Consume and ignore with `scanf`

To consume and ignore (skip) data with a given format:
Use * after %, without specifying address where to read
⇒ `scanf` reads according to pattern, but does not store data
and does not count in result (number of read objects)

Example: text with three grades and average, need just average:

```c
int avg;
if (scanf("%*d%*d%*d%d", &avg) == 1) { /* use */ }
else { /* wrong format, handle error */ }
```

Example: consume rest of line

```c
scanf("%*[^\n]");      // consume up to \n, without \n
if (getchar() == EOF) { /* end of input */ }
// otherwise, getchar() has consumed \n, continue
```

Specifying limits in `scanf`

Number between % and format character limits count of chars read
%4d   int, at most 4 chars (initial spaces don't count, sign does!)

```
scanf("%d%d", &m, &n);       12 34    m=12 n=34
scanf("%2d%2d", &m, &n);     12345    m=12 n=34 rest: 5
scanf("%d.%d", &m, &n);      12.34    m=12 n=34
scanf("%f", &x);             12.34    x=12.34
scanf("%d%x", &m, &n);       123a     m=123 n=0xA
```

Format specifiers in `scanf`

`%d`: signed decimal int

`%i`: signed decimal, octal (0) or hexadecimal (0x, 0X) int

`%o`: octal (base 8) int, optionally preceded by 0

`%u`: unsigned decimal int (*warning:* accepts negative and converts)

`%x`, `%X`: hexadecimal int, optionally with 0x, 0X

`%c`: *any* char, including whitespace

`%MAXs`: string of chars, until first whitespace. '\0' is added
MUST have a *constant* `MAX` unless assignment suppressed with ∗

`%a`, `%A`, `%e`, `%E`, `%f`, `%F`, `%g`, `%G`: real (possibly with exponent)

`%p`: pointer, as printed by `printf`

`%n`: writes into argument (`int *`) count of chars read so far
does not read; does not add to count of read objects (return value)

`%[···]`: string of indicated allowed characters

`%[^···]`: string except indicated disallowed chars

`%%`: percent character

Format specifiers in `printf`

`%d`, `%i`: signed decimal int
`%o`: signed octal int, without initial 0
`%u`: unsigned decimal int
`%x`, `%X`: hexazecimal int, without 0x/0X; lower/upper case
`%c`: character
`%s`: string of characters, up to '\0' or indicated precision
`%f`, `%F`: real w/o exponent; 6 decimal digits; no dot if 0 precision
`%e`, `%E`: real with exponent; 6 decimal digits; no dot if 0 precision
`%g`, `%G`: real, like `%e`, `%E` if exp. $< $ -4 or $\geq$ precision; else like `%f`.
Does not print zeroes or decimal point if useless
`%a`, `%A`: hexadecimal real with decimal 2's exponent 0xh.$hhhh$p$\pm d$
`%p`: pointer, usually în hexadecimal
`%n`: writes into argument (`int *`) count of chars written so far
`%%`: percent character

Formatting: modifiers

Format specifiers may have other *optional* elements:

$$\% \; \textit{flag size . precision modifier type}$$

Flags: ∗: field is read but not assigned (is ignored)   (`scanf`)
−: aligns value left for given size   (`printf`)
+: + before positive number of signed type   (`printf`)
space: space before positive number of signed type   (`printf`)
0: left-filled with 0 up to indicated size   (`printf`)

Modifiers:
hh: argument is `char` (for d i o u x X n format) (1 byte)
`char` c; scanf(`"%hhd"`, &c);   in: 123 → c = 123 (1 byte)
h: argument is `short` (for d i o u x X n format), e.g. %hd
l: arg. `long` (format d i o u x X n) or `double` (fmt. a A e E f F g G)
`long` n; scanf(`"%ld"`, &n); `double` x; scanf(`"%lf"`, &x);
ll: argument is `long long` (for d i o u x X n format)
L: argument is `long double` (for a A e E f F g G format)

Formatting: size and precision

Size: an integer
`scanf`: *maximal* character count read for this argument
`printf`: *minimal* character count for printing this argument
right aligned and filled with spaces, or according to modifiers

Precision: only in `printf`; dot . optionally followed by an integer
(if only dot, precision is zero)
minimal number of digits for `diouxX` (filled with 0)
number of decimal digits (for `Eef`) / significant digits (for `Gg`)
`printf("|%7.2f|", 15.234);` | 15.23| 2 decimals, 7 total
maximal number of chars to print from a string (for `s`)
`char m[3]="Jan"; printf("%.3s", m);` (for string w/o '\0')

In `printf`, can have `*` instead of size and/or precision
Then, size/precision is given by next argument:
`printf("%.*s", max, s);`   prints at most `max` chars

Sample formatted output

Floating point numbers in various formats:

```c
printf("%f\n", 1.0/1100);   // 0.000909 : 6 decimal digits
printf("%g\n", 1.0/1100);   // 0.000909091 : 6 significant dig.
printf("%g\n", 1.0/11000);  // 9.09091e-05 : 6 significant dig.
printf("%e\n", 1.0);        // 1.000000e+00 : 6 decimal digits
printf("%f\n", 1.0);        // 1.000000 : 6 decimal digits
printf("%g\n", 1.0);        // 1 : no period and useless zeroes
printf("%.2f\n", 1.009);    // 1.01: 2 decimal digits
printf("%.2g\n", 1.009);    // 1: 2 significant digits
```

Writing integers in table form:

```c
printf("|%6d|", -12); | -12|  printf("|% d|", 12); | 12|
printf("|%-6d|", -12); |-12 | printf("|%06d|", -12); |-00012|
printf("|%+6d|", 12); | +12|
```

Write 20 characters (`printf` returns count of written chars)

```c
int m, n, len = printf("%d", m); printf("%*d", 20-len, n);
```

Examples of formatted input

Two characters separated by a single space (consumed by %*1[ ])

```c
char c1, c2; if (scanf("%c%*1[ ]%c", &c1, &c2) == 2) ...
```

Read an int with exactly 4 digits:

```c
unsigned n1, n2, x;
if (scanf("%n%4u%n", &n1, &x, &n2)==1 && n2 - n1 == 4)...
```

"%n" counts read chars; store counters in n1, n2, then subtract

Reads/checks for a word that must appear:

```c
int nr=0;
scanf("http://%n", &nr); if (nr == 7) { /*appears */}
else { /*  does not reach %n, nr stays 0 */}
```

Ignores up to (and excluding) a given char (\n):

```c
scanf("%*[^\n]");
```

Test for the right number of read objects, not just nonzero!

```c
if (scanf("%d", &n) == 1), not just if (scanf("%d", &n))
```

scanf may also return EOF, which is nonzero!

For integers, test overflow using `extern int errno`;

```c
#include <errno.h> // declares errno and error codes
if (scanf("%d", &x) == 1)) // test reset errno on overflow
  if (errno == ERANGE) { printf("number too big"); errno = 0; }
```

ERRORS with reading from input

NO! `while` (`scanf("%...", ...)`) DON'T test for nonzero result.
It could be > 0 (items read), or -1 (EOF), nothing read!
YES: `while` (scanf("%...", ...) == *how-many-items-wanted*)

NO! `scanf("%20[a-z]s", buf)`. The format is [], not []s
YES: `if` (scanf("%20[a-z]", buf) == 1) ...

NO! `scanf("%20s,%d", name, &grade)`. The s format reads
everything non-whitespace, so it won't stop at comma
YES: `if` (scanf("%20[^,],%d", name, &grade) == 2)
to read a string with no comma (all else allowed, including
whitespace), the comma, and a number