CS Basics
8) I/O and Control

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

Contents

Input and Output

Functions for Input and Output (I/O)

Basic Functions

- `getchar()` and `putchar()`: read or write single characters
- `puts()`: write entire strings

Formatted Input and Output

- `scanf()` and `printf()`: read and write integers, floating point numbers, characters and strings in a formatted way

`getchar()`: Single Character Input

Function `int getchar(void)`

- Part of standard I/O library (`stdio.h`)
- Reads a single character from standard input (`stdin`)

Syntax

```
int i = getchar();
```

Example

```
char c = 0;

int i = getchar();
if (EOF != i)
  c = (char)i;
```

End of file

- Character `EOF` indicates an error *or* the end of input
- `EOF` is a constant, also defined in `stdio.h`

`putchar()`: Single Character Output

Function `int putchar(int c)`

- Writes a single character to standard output (`stdout`)
- Returns `EOF` in case of error, or `c` otherwise

Example

```
char c = 'a';
putchar(c);

if (EOF == putchar('\n'))
  // panic...
```

`puts()` and ~~`gets()`~~

`puts()` : Write a string to `stdout`

- Appends a newline character (`'\n'`) to the output
- Returns `EOF` in case of error

~~`gets()` : Read a string from `stdin`~~

- ⚠ Warning: Using `gets()` is dangerous and only shown for educational purposes here!
- When used incorrectly, other functions (including `scanf()`) may also cause buffer overflows!
- In `C11`, the function has been retired and `gcc` issues a warning when it is used

Example: `gets()` Buffer Overflow

```c
#include <stdio.h>

// Warning: Using gets() is dangerous and shown only
// for educational purposes here!

int main(void) {
  char line[10]; // not initialized
  puts("Hello, what is your name?");
  gets(line); // DANGEROUS!!!
  printf("Hello ");
  puts(line);
}
```

`scanf()`: Read Formatted Input

Function `int scanf(format, ...)`

- Reads (multiple) items from `stdin`
- Parses and stores them in variables

Syntax

```
int scanf(format, arg1, arg2, ... , argN);
```

- Arguments are *pointers* (i.e. addresses) where the values will be stored
- Note: Arrays are implicit pointers to their first element
- Returns the *number of items* read *or* `EOF` in case of error

Example

```
char str[20] = {0};
int i = 0;
float f = 0;
if (3 != scanf("%19s %d %f", str, &i, &f))
  // panic...
```

`scanf()`: Conversion Specifiers

Numbers

- `%d`: signed int (decimal)
- `%i`: signed int (decimal, octal, hexadecimal)
- `%u`: unsigned int (decimal)
- `%o`: unsigned int (octal)
- `%x`: unsigned int (hexadecimal)
- `%f`: float
- `%lf`: double

Characters

- `%c`: character
- `%s`: consecutive non-white-space characters
- `[set]`: consecutive characters from the set

Size Specifiers

- `%hh`: (un)signed char
- `%h`: (un)signed short
- `%l`: (un)signed long
- `%ll`: (un)signed long long
- `%lf`: double

Note: This is not the full list of specifiers, consult library documentation for more.

scanf(): Example

Read some numbers

```
int i1 = 0;
double d1 = 0;
char c1 = 0;

puts("Enter an integer, a double and a char:");
if (3 == scanf("%d %lf %c", &i1, &d1, &c1))
  printf("i1=%d, d1=%lf, c1=%c\n", i1, d1, c1);

/*
  E.g. input is: "1234 10.5 T"
  Output is "i1=1234, d1=10.500000, c1=T"
*/
```

scanf(): Specify Length of Items

For each item, the number of *characters* to read can be specified. Example:

```
int i2 = 0;
double d2 = 0;

puts("Enter an integer (3 digits) and a double (5 chars):");
if (2 == scanf("%3d %5lf", &i2, &d2))
  printf("i2=%d, d2=%lf\n", i2, d2);

/*
  E.g. input is: "1234 10.5"
  Output is: "i2=123, d2=4.000000" (!!!)
*/
```

`scanf()`: Related Functions (I)

The standard library contains other functions similar to `scanf()`

- E.g. `fscanf()`, `sscanf()`, `wscanf()`, ...

Example: `sscanf()`

- Scans from a given string instead of `stdin`

```
int i3 = 0;
double d3 = 0;
char c3 = 0;
char str[] = "1234 10.5 T";

if (3 == sscanf(str, "%d %lf %c", &i3, &d3, &c3))
  printf("i3=%d, d3=%lf, c3=%c\n", i3, d3, c3);

/*
  Output is "i3=1234, d3=10.500000, c3=T"
*/
```

`scanf()`: Related Functions (II)

As a replacement for `gets()`, `fgets()` is a recommended alternative

Syntax

```
char *fgets(char *buf, int n, FILE *fp);
```

- Reads up to `n-1` characters (or until a `'\n'`) from input stream `fp` into `buf`
- Returns the pointer to `buf` on success, or `NULL` if an error occurred

Example:

```
char line[256] = {0};
int a = 0, b = 0, c = 0, d = 0;

while (NULL != fgets(line, sizeof line, stdin)) {
  sscanf(line, "%d %d %d %d", &a, &b, &c, &d); // bug!
  printf("%d + %d + %d + %d = %d\n", a, b, c, d, (a + b + c + d));
}
```

`printf()`: Write Formatted Output

Function `int printf(format, ...)`

- Prints texts and possibly other items to `stdout`
- Items and their formatting are specified using placeholders
- Returns the number of characters printed (excluding the terminating null byte)

Syntax

```
int printf(format, arg1, arg2, ... , argN);
```

Example

```
double d1 = 12345678.12345678;
char str1[] = "Some nice floats...";
printf("%s\n%f %E %e %g\n%a\n", str1, d1, d1, d1, d1, d1);

/*
  Output is:
  Some nice floats...
  12345678.123457 1.234568E+07 1.234568e+07 1.23457e+07
  0x1.78c29c3f35ba2p+23
*/
```

`printf()`: Conversion Specifiers

Numbers

- `%d`, `%i`: signed int (decimal)
- `%u`: unsigned int (decimal)
- `%o`: unsigned int (octal)
- `%x`, `%X`: unsigned int (hexadecimal)
- `%f`, `%F`, `%e`, `%E`, `%g`, `%G`, `%a`, `%A`: float or double

Characters

- `%c`: character
- `%s`: consecutive characters (strings)

Size Specifiers

- `%hh`: (un)signed char
- `%h`: (un)signed short
- `%l`: (un)signed long
- `%ll`: (un)signed long long

Note: This is not the full list of specifiers, consult library documentation for more.

printf():Field Width and Precision

Per item, minimum length and precision can be given

- Format: `minlen.precision`
- Minimal length: if not long enough, gap is filled with spaces
- Precision: Max. number of decimal places (or characters for strings)

Example

```
  double d2 = 101.593039;
  printf("%f %.2f %.0f %10.2f\n", d2, d2, d2, d2);

  char str2[] = "ABCDEFGHIJKLMNOP";
  printf("%1s %14.5s %.5s\n", str2, str2, str2);

/*
  Output is:
  101.593039 101.59 102      101.59
  ABCDEFGHIJKLMNOP           ABCDE ABCDE
*/
```

`printf()`: Flags

Each item can be adjusted using additional flags

- `-` : left justified
- `+` : for numbers, the sign will always be printed
- `0` : use zeros instead of spaces
- `' '` (space character) : precede positive numbers with space
- `#` : prefix numbers with `0` or `0x` respectively (for `%o`, `%x` and `%X`), always print the decimal point (for all float and double specifiers) or prevent truncation of zeros (for `%g` and `%G`)

printf(): Example of Flags

```c
int i1 = 1234;
double d3 = 12.0, d4 = -3.3;

printf(":%6d %7.0f %10.1e:\n", i1, d3, d4);
printf(":%-6d %-7.0f %-10.1e:\n", i1, d3, d4);
printf(":%+6d %+7.0f %+10.1e:\n", i1, d3, d4);
printf(":%-+6d %-+7.0f %-+10.1e:\n", i1, d3, d4);
printf(":%7.0f %#7.0f %7g %#7g:\n", d3, d3, d4, d4);
```

```
/*
  Output is:
  :  1234      12   -3.3e+00:
  :1234   12      -3.3e+00  :
  : +1234     +12   -3.3e+00:
  :+1234  +12     -3.3e+00  :
  :     12     12.    -3.3 -3.30000:
*/
```

`printf()`: Related Functions

Like for `scanf()`, the standard library also provides related functions for `printf()`

- E.g. `fprintf()`, `snprintf()`,[1] `wprintf()`, ...

Example: `snprintf()`

- Prints formatted data to a given string instead of `stdout`

```c
char str3[21] = {0};
int outlen = snprintf(str3, 11, "%d + %d is %d (really!)", 3, 5, 3 + 5);
puts(str3);
printf("%d characters written.\n", outlen);

/*
  Output is:
  3 + 5 is 8
  20 characters written.
*/
```

[1] Do not use `sprintf`, it does not check the length of data written!

Outline of an Interactive C Program

```c
int main(void) {
  // declare some variables
  char c = 0, d = 0;
  float x = 0, y = 0;

  // read some data
  int i = getchar();
  if (EOF != i)
    c = (char)i;
  if (scanf("%f", &x) != 1)
    return EXIT_FAILURE;

  // ... process ...

  // output results
  putchar(c);
  printf("\n%3d %7.4f\n", i, x);
}
```

A Bad Example

```c
int main(void) {
  char name[11];
  int nb_grades;
  float grade, sum, avg;

  puts("Hello, what is your name?");
  scanf("%[^\n]", name); // read as long as not '\n'
  printf("Number of grades?");
  scanf("%d", &nb_grades);

  for (int i = 0; i < nb_grades; i++) {
    printf("Enter grade %d: ", i + 1);
    scanf("%f", &grade);
    sum += grade;
  }
  avg = sum / nb_grades;
  printf("%s, your average grade is: %.1f\n", name, avg);
}
```

Control Statements

Branching: `if`, `else` (I)

Syntax

```
if (expression) statement1 [else statement2]
```

- If `expression` is true, `statement1` is executed (and not `statement2`)
- If `expression` is false, `statement2` is executed (and not `statement1`)
- The `else` clause is optional

Example

```
if (status == 'S')
  tax = 0.20 * pay;
else
  tax = 0.14 * pay;
```

Branching: `if`, `else` (II)

Using block statements, multiple statements can be grouped into one

```
if (circle) {
  scanf("%f", &radius);
  area = PI * radius * radius;
  printf("Area of the circle = %f\n", area);
} else {
  scanf("%f %f", &length, &width);
  area = length * width;
  printf("Area of rectangle = %f\n", area);
}
```

Loops: The `while` Statement (I)

Syntax

```
while (expression) statement
```

- The `statement` will be executed as long as `expression` is true

Example

```
int digit = 0;
while (digit <= 9) {
  printf("%d\n", digit);
  digit++;
}
```

Loops: The `while` Statement (II)

```c
int main(void) {
  char name[11] = {0};
  int nb_grades = 0;
  float grade = 0, sum = 0;

  puts("Hello, what is your name?");
  fgets(name, sizeof name, stdin);

  printf("Hello %sEnter your grades, end with Ctrl-D:\n", name);
  while (1 == scanf("%f", &grade)) {
    sum += grade;
    nb_grades++;
  }

  if (nb_grades > 0)
    printf("Your average grade is: %.1f\n", sum / nb_grades);
}
```

Loops: `do ... while` Statements (I)

Syntax

```
do statement while (expression);
```

- Executes `statement` as long as `expression` is true
- The statement is executed *at least once*, since `expression` is only evaluated afterwards

Example

```
digit = 0;
do
  printf("%d\n", digit++);
while (digit <= 9);
```

Loops: `do ... while` Statements (II)

Example: Transform a string to uppercase

```
puts("Enter a string to be transformed to uppercase:");
char line[256] = {0};
if (NULL == fgets(line, sizeof line, stdin))
  return EXIT_FAILURE;

uint32_t size = 0;
do
  size++;
while (line[size] != 0);

uint32_t pos = 0;
do {
  putchar(toupper(line[pos++]));
} while (pos < size);
```

Loops: The `for` Statement (I)

Syntax

```
for (expression1; expression2; expression3)
  statement
```

- `expression1`: can be used to initialize parameter(s)
- `expression2`: is the loop condition (must be true for the loop to continue execution)
- `expression3`: can be used to alter parameter(s)

Examples

```
for (int digit = 0; digit <= 9; digit++) {
  printf("%d\n", digit);
}

for (int i = 0, j = 1; i < 10; i++, j++) {
  printf("%d %d\n", i, j);
}
```

Loops: The `for` Statement (II)

Uppercasing using `for`

```
for (uint32_t pos = 0; pos < size; pos++) {
  putchar(toupper(line[pos]));
}
```

The `switch` Statement (I)

Evaluates a particular statement out of a group of statements

Syntax

```
switch (expression) {
case const1:
  statement1
case const2:
  statement2
  ...
case constN:
  statementN
default:
  statementDef
}
```

The `switch` Statement (II)

Evaluation

- `const1` to `constN` are integer constants and must be unique per switch statement
- If the value of `expression` matches one of the constants, execution jumps to the respective `statement`
- If `statement` does not end with a `break` instruction, execution *falls through (!)* to the next `statement`

Optional `default` case

- The case `default` matches any value of `expression`
- Also ending `statementDef` with `break` is recommended!

The `switch` Statement (III)

Example using fall-through:

```
int choice = 0;
puts("Enter a letter to choose a color (RGB):");
choice = getchar(); // no error check!
switch (choice) {
case 'r':
case 'R':
  printf("Red.\n");
  break;
case 'g':
case 'G':
  printf("Green.\n");
  break;
case 'b':
case 'B':
  printf("Blue.\n");
  break;
default:
  printf("Not a valid color.\n");
  break;
}
```

The `break` Statement (I)

Used to exit from a loop- or switch statement

- Can be used within `while`, `do-while`, `for` and `switch` statements

Syntax

```
break;
```

Example (`switch`)

```
case 'B':
  printf("Blue.\n");
  break;
```

The `break` Statement (II)

Example (`while`)

```
uint64_t val = 0, exponent = 0, res = 1, i = 0;

puts("Enter value and exponent:");
scanf("%lu %lu", &val, &exponent);

while (1) {
  if (i++ == exponent)
    break;
  res *= val;
}

printf("%lu to the power of %lu = %lu\n", val, exponent, res);
```

The `continue` Statement

Used to skip the current pass in a loop

- Does not terminate the loop
- Skips the remaining statements in the loop body and returns to the loop condition

Example

```
int64_t val = 0;
for (int i = 0; i < 3; i++) {
  puts("Enter a number:");
  if (1 != scanf("%ld", &val))
    continue;

  if (val < 0) {
    printf("Negative numbers have no sqrt!\n");
    continue;
  }

  printf("Square root of %ld is %lg\n", val, sqrt(val));
}
```

The `goto` Statement (I)

Equivalent to `JMP` in assembly

- Continues execution at the given label
- Label must reside in the same function and be unique for that function
- *Using `goto` should be generally avoided!*

Syntax

```
goto label;
...
label: statement
```

The goto Statement (II)

```c
int main(void) {
  int64_t val = 0;
  for (int i = 0; i < 3; i++) {
    puts("Enter a number:");
    if (1 != scanf("%ld", &val))
      continue;

    if (val < 0)
      goto error_negative_number;

    printf("Square root of %ld is %lg\n", val, sqrt(val));
  }

  return EXIT_SUCCESS;

error_negative_number:
  printf("Negative numbers have no sqrt!\n");
  return EXIT_FAILURE;
}
```

Conclusion

Conclusion

Input and Output

- Read a string, parse it
- Print out formatted output
- More about files and other functions later

Control statements

- Similar to Java: `for`, `while`, `if`, `do`, `switch`, `continue`, `break`
- `goto` should be avoided