COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2023-January 2024

UNIVERSITÀ DEGLI STUDI DI PADOVA

Input and output

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Input and Output

For input and output we shall use the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs.

A text stream consists of a sequence of lines; each line ends with a newline character.

The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with **getchar**:

int getchar(void)

The function

int putchar(int)

is used for output: **putchar**(c) puts the character c on the standard output.

Input and Output

Input and/or output can usually be directed from / to a file with *<filename* or *>filename*:

program <infile causes program to read characters from infile instead of standard input

program >outfile will write the standard output to outfile instead of standard output.

If pipes are supported,

program | anotherprogram

puts the standard output of program into the standard input of anotherprogram.

Each source file that refers to an input/output library function must contain the line *#include <stdio.h>* before the first reference.

Input and Output

- Formatted Output – printf
- Formatted Input – Scanf
- File access
- Line Input and Output
- Error Handling - Stderr and Exit
- Variable-length Argument Lists

Input and Output: formatted output

The output function **printf** (stands for "print formatted“) translates internal values to characters.

The declaration is

*int printf(char *format, arg1, arg2, ...);*

printf allows you to print data of various types, such as integers, floating-point numbers, characters, and strings, with precise formatting, it converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf.

Input and Output

Each conversion specification begins with a **%** and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An *h* if the integer is to be printed as a short, or *l* (letter ell) if as a long.

If the character after the % is not a conversion specification, the behavior is undefined.

Input and Output – basic printf conversion

Character	Argument type; Printed As
d, i	***int***; decimal number
o	***int***; unsigned octal number (without a leading zero)
x, X	***int***; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	***int***; unsigned decimal number
c	***int***; single character
s	***char ****; print characters from the string until a '\0' or the number of characters given by the precision.
f	***double***; *[-]m.dddddd*, where the number of d's is given by the precision (default 6)
e, E	***double***; *[-]m.dddddde+/-xx* or *[-]m.ddddddE+/-xx*, where the number of d's is given by the precision (default 6)
g, G	***double***; use *%e* or *%E* if the exponent is less than -4 or greater than or equal to the precision; otherwise use *%f*. Trailing zeros and a trailing decimal point are not printed
p	***void ****; pointer (implementation-dependent representation)
%	no argument is converted; print a %

Input and Output – basic printf conversion

There is the possibility to specify a width or precision by using an argument by means of the character *, in which case the value is computed by converting the next argument (which must be an int).

For example

- to print at most max characters from a string s,
  *printf("%.*s", max, s);*
- to print characters from a string s in a minimum field width of max length,
  *printf("%*s", max, s);*

Input and Output – basic printf conversion

Let’s consider the precision that relates to string arguments. The following rows shows the effect of a variety of specifications in printing ``hello, world'' (12 characters). We have put colons around each field.

:%s: :hello, world:
:%10s: :hello, world:
:%.10s: :hello, wor:
:%-10s: :hello, world:
:%.15s: :hello, world:
:%-15s: :hello, world :
:%15.10s: :   hello, wor:
:%-15.10s: :hello, wor   :

Input and Output

Common Escape Sequences:

Escape sequences are special characters used within format specifiers to control the formatting of the output. Common escape sequences include:

\n: Newline (line feed)

\t: Tab

\": Double quotation mark

\': Single quotation mark

****: Backslash

Error Handling:

It's essential to ensure that the number and types of arguments provided match the format specifiers in the printf statement. Mismatches can lead to undefined behavior or errors in your program.

Input and Output

If s is an array of char the invocation

printf("%s", s);

is safe, while the invocation

printf(s);

fails if in the string s there are characters %

The function **sprintf** performs the same conversions as printf does, but stores the output in a string:

*int sprintf(char *buffer, char *format, arg1, arg2, ...);*

sprintf formats the arguments in arg1, arg2, etc., according to format as before, but places the result in the array of char buffer instead of the standard output; buffer must be big enough to receive the result.

Input and Output: formatted input

The I/O so far has been character oriented, reading one character at a time.

It is hard to interpret two values on the same line

printf("enter your age and weight ");

A name and a number

printf("enter your name and age ");

Other combinations

These problems could be solved by asking the user for one piece of input per line.

- This might be unnatural.
- It is also awkward when there are many fields.

Input and Output: formatted input

The function **scanf** is the input analog of printf, providing many of the same conversion facilities in the opposite direction. The scanf function is part of the stdio.h library, and its syntax is as follows

*int scanf(char *format, ...)*

scanf reads characters from the standard input, interprets them according to the specification provided in format string, and stores the results through the remaining arguments. These remaining arguments, that ***must be pointers***, indicate where the corresponding converted input shall be stored.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns the number of successfully matched and assigned input items. This can be used to decide how many items were found.

Input and Output: formatted input

On the end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- blanks or tabs, which are not ignored
- ordinary characters (not %), which are expected to match the next non-white space character of the input stream
- conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

Input and Output: formatted input

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made.

An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. White space characters are blank, tab, newline, carriage return, vertical tab, and form feed.

The conversion character indicates the interpretation of the input field.

Input and Output – basic scanf conversion

Università degli Studi di Padova

Character	Input data; Argument type
d	decimal integer; ***int ****
i	integer; ***int ****. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); ***int ****
x	hexadecimal integer (with or without leading 0x or 0X); ***int ****
u	unsigned decimal integer; ***unsigned int ****
c	characters; ***char ****. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use ***%1s***
s	character string (not quoted); ***char ****, pointing to an array of characters long enough for the string and a terminating ***'\0'*** that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; ***float ****
%	literal %; no assignment is made.

Input and Output: formatted input

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list.

```
#include <stdio.h>
main() /* rudimentary calculator */
{
    double sum, v;
    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Input and Output: formatted input

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with the scanf statement:

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

to read input lines that contain dates of the form
25 Dec 1988
The scanf statement is

```
int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);
```

Input and Output: formatted input

scanf ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with string variant input management of scanf.

Warning: the arguments to scanf must be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.

Input and Output: formatted input

It's crucial to validate the input when using scanf to avoid unexpected behavior or errors. For example, you should check the return value to see if the input was successfully read, and you can use conditional statements to prompt the user for input again if needed.

There is a function sscanf that reads from a string instead of the standard input:

int sscanf(char *string, char *format, arg1, arg2, ...)

It scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

File access

In C programming, file access functions are part of the standard C library (<stdio.h>) and are used to interact with files on a computer's file system. These functions allow you to perform various file-related operations, such as opening, reading, writing, closing, and manipulating files.

So far, we have seen how to read from the standard input and to write to the standard output, which are automatically defined for a program by the local operating system.

Now we are going to see how to access a file that is not already connected to the program. How can we arrange for the named files to be read - that is, how to connect the external names to the statements that read the data?

File access

The rules are simple. Before a file can be read or written, it has to be opened by the library function **fopen**. fopen takes an external filename, and after some negotiations with the operating system returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the file pointer, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. The definitions in <stdio.h> include a structure declaration called FILE.

File access: fopen

The only declarations needed for a file pointer are:

*FILE *fp;*

*FILE *fopen(char *name, char *mode);*

These statements mean that *fp* is a pointer to a *FILE*, and *fopen* returns a pointer to a *FILE*. *FILE* is a type name, like *int*, not a structure tag; therefore, it is defined with a typedef.

The call to fopen in a program is

fp = fopen(name, mode);

The first argument of fopen is a character string containing the **name of the file**. The second argument is the mode, also a character string, which indicates **how one intends to use the file**.

File access: fopen

Allowable modes include read ("**r**"), write ("***w***"), and append ("***a***"). Some systems distinguish between text and binary files; for the latter, a "***b***" must be appended to the mode string.

*FILE *file = fopen("example.txt", "r"); // Open for reading*

- If a file that does not exist is opened for writing or appending, it is created if possible.
- Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them.
- Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, *fopen* will return **NULL**.

File access: getc and putc

How to read or write the file once it is open?

getc returns the next character from a file; it needs the file pointer to tell it which file.

*int getc(FILE *fp)*

getc returns the next character from the stream referred to by fp; it returns EOF for end of file or error.

putc is an output function:

*int putc(int c, FILE *fp)*

putc writes the character c to the file fp and returns the character written, or EOF if an error occurs.

Like getchar and putchar, getc and putc may be macros instead of functions.

File access: fprintf

For formatted input or output of files, the functions fscanf and fprintf may be used. These are identical to scanf and printf, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.
The fprintf function allows you to write formatted data to a file, similar to the printf function for standard output. It takes a file stream and a format string, along with additional arguments for variable data.

Syntax:

*int fprintf(FILE *stream, const char *format, ...);*

Example:

fprintf(file, "This is an integer: %d\n", 42);

File access: fscanf

The fscanf function is used to read formatted data from a file, similar to the scanf function for standard input. It takes a file stream and a format string, along with pointers to variables where the data will be stored.

Syntax:

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

Example:

```
int number;
fscanf(file, "This is an integer: %d", &number);
```

File access: fread

The *fread* function is used to read data from a file into a specified buffer. You provide the buffer, the size of each element to be read, the number of elements to be read, and the file stream.

Syntax:

*size_t fread(void *ptr, size_t size, size_t count, FILE *stream);*

Example:

```
char buffer[100];
fread(buffer, sizeof(char), 100, file);
```

File access: fwrite

The *fwrite* function is used to write data from a buffer to a file. You provide the buffer, the size of each element to be written, the number of elements to be written, and the file stream.

Syntax:

*size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);*

Example:

```
char data[] = "This is a sample text.";
fwrite(data, sizeof(char), strlen(data), file);
```

File access: fseek and ftell

fseek allows you to set the file position indicator within a file stream. *ftell* is used to determine the current file position.

Syntax:

*int fseek(FILE *stream, long int offset, int origin); (for fseek)*

Syntax:

*long int ftell(FILE *stream); (for ftell)*

Example:

```
fseek(file, 0, SEEK_SET); // Move to the beginning of the file
long position = ftell(file); // Get the current position
```

File access: rewind and feof

The *rewind* function is used to reset the file position indicator to the beginning of the file.

Syntax:

```
void rewind(FILE *stream);
```

Example:

```
rewind(file);
```

The *feof* function is used to check if the end-of-file has been reached in a file stream. It returns a non-zero value if the end-of-file indicator is set.

Syntax:

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

Example:

```
if (feof(file)) {
    // End of file reached
}
```

File access: fclose

The *fclose* function is used to close an open file, releasing any resources associated with it. It should be called when you're done working with a file to ensure proper file management. fclose flushes the stream pointed to by stream (writing any buffered output data) and closes the underlying file descriptor, freeing the file pointer for another file.

Syntax:

*int fclose(FILE *stream);*

Upon successful completion, 0 is returned. Otherwise, EOF is returned. In either case, any further access (including another call to fclose()) to the stream results in undefined behavior.

Example:

fclose(file);

File access: remove and rename

The *remove* function is used to delete a file, and the *rename* function is used to change the name of a file.

Syntax:

*int remove(const char *filename); (for remove)*

Syntax:

*int rename(const char *old_filename, const char *new_filename); (for rename)*

Example:

```
remove("oldfile.txt");
rename("newfile.txt", "renamedfile.txt");
```

File access: stdout, stdin, stderr

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them.
These files are

- the standard input
- the standard output
- the standard error;

the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>. Normally stdin is connected to the keyboard and stdout and stderr are connected to the screen, but stdin and stdout may be redirected to files or pipes.

File access: example


```
#include<stdio.h>
#include<conio.h>
main()
{
      FILE *fp;
      char ch;
      fp = fopen("hello.txt", "w");
      printf("Inserisci il dato: ");
      while( (ch = getchar()) != EOF) {
        putc(ch,fp);
      }
      fclose(fp);
      fp = fopen("hello.txt", "r");

      while( (ch = getc(fp)! = EOF)
        printf("%c",ch);

      fclose(fp);
}
```

File access: example


```
#include <stdio.h>
#include <stdlib.h>

int main() {
   // create file pointers.
   FILE *names = fopen("names.txt", "r");
   FILE *greet = fopen("greet.txt", "w");

   // check if all is fine
   if (!names || !greet) {
      fprintf(stderr, "Apertura del file fallita!\n");
      return EXIT_FAILURE;
   }

   // Time of greetings
   char nome[20];
   // keep on reading until ....
   while (fscanf(names, "%s\n", nome) > 0) {
      fprintf(greet, "Ciao, %s!\n", nome);
   }

   // When the feof is reached prints a message on stdout
   if (feof(names)) {
      printf("The greetings are over!\n");
   }

   return EXIT_SUCCESS;
}
```