COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Miscellaneous functions

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Miscellaneous functions

- Line Input and Output
- Error Handling - Stderr and Exit
- Variable-length Argument Lists

File open

The C library function *FILE *fopen(const char *filename, const char *mode)* opens the filename pointed by filename using the given mode.

Declaration

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

Parameters

filename − This is the C string containing the name of the file to be opened.

mode − This is the C string containing a file access mode.

File open

Mode & Description

1. "**r**“ Opens a file for reading. The file must exist.
2. "**w**“ Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.
3. "**a**“ Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.
4. "**r+**“ Opens a file to update both reading and writing. The file must exist.
5. "**w+**“ Creates an empty file for both reading and writing.
6. "**a+**“ Opens a file for reading and appending.

This function returns a FILE pointer. Otherwise, NULL is returned.

Reading from / Writing to files

- *int getc(FILE *fp*); // returns the next character from the file stream fp
- *int putc(int c, FILE *fp); //* writes the character c to the file stream fp

- *int fscanf(FILE *stream, const char *format, ...); //* reads formatted data
- *int fprintf(FILE *stream, const char *format, ...); //* writes formatted data

- *size_t fread(void *ptr, size_t size, size_t count, FILE *stream); //* reads data from a file into a specified buffer
- *size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream); //* writes data from a buffer to a file

- *int fgetc(FILE *fp); //* returns the next character from the file stream fp

Reading from / Writing to files

The getc and fgetc functions are both used to read a single character from a file, and they are very similar in terms of functionality. Both are part of the <stdio.h> library. However, there are some differences between them.

Macro vs. Function:

- getc is often implemented as a macro, meaning it may directly expand to inline code. This can make it slightly faster in some implementations, as there's no function call overhead.
- fgetc is a true function, ensuring it behaves consistently across different C implementations.

Thread-Safety:

- Since getc can be implemented as a macro, it may not be thread-safe if used in multithreaded applications where multiple threads read from the same file.
- fgetc, being a function, is generally safer in multithreaded contexts.

Reading from / Writing to files

Portability and Consistency:

- fgetc is often preferred when portability and consistent behavior are crucial, as it is guaranteed to be a function in all C standard libraries.
- getc might behave differently on certain systems or compilers due to its implementation as a macro.

Speed: getc may be faster due to its implementation as a macro in some libraries.

Thread-Safety: fgetc is generally safer in multithreaded programs.

Portability: fgetc is more consistent across platforms and is often the preferred choice for more portable code.

File handling: fseek / ftell

- *int fseek(FILE *fp, long int offset, int origin);* // sets file position indicator in the file stream fp, returns zero if successful, otherwise a non-zero value.
  - *fp* is the pointer to a FILE object that identifies the stream.
  - *offset* is the number of bytes to offset from origin.
  - *origin* is the position from where offset is added, it is specified by one of the following constants
    - **SEEK_SET** Beginning of file
    - **SEEK_CUR** Current position of the file pointer
    - **SEEK_END** End of file

- *long int ftell(FILE *fp);* // returns the current file position of the given file stream fp if successful, otherwise -1

File handling: fseek / ftell


```
#include <stdio.h>
int main () {
   FILE *fp;
   int len;
   fp = fopen("file.txt", "r");
   if ( fp == NULL )  {
      printf ("Error opening file");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);
   len = ftell(fp);
   fclose(fp);
   printf("Total size of file.txt = %d bytes\n", len);
   return(0);
}
```

File buffering

The setvbuf function is used to control the buffering behavior of a file stream. This function allows you to specify the type and size of buffering to use with a specific file.

- *int setvbuf(FILE *fp, char *buffer, int mode, size_t size);*
  - *fp* is the pointer to a FILE object that identifies an open stream.
  - *buffer* is the user allocated buffer. If set to NULL, the function automatically allocates a buffer of the specified size.
  - *mode* is the buffering mode to use. It can be one of the following:
    - *_IOFBF* (Full buffering). On output, data is written once the buffer is full. On Input the buffer is filled when an input operation is requested (suitable for large files where chunks of data are read at a time).

File buffering

- *_IOLBF* (Line buffering). On output, data is written when a newline character is inserted into the stream or when the buffer is full, whatever happens first. On Input, the buffer is filled till the next newline character when an input operation is requested (useful for text files where data is processed line-by-line).
- *_IONBF* (No buffering). No buffer is used. Each I/O operation is written/read as soon as possible. The buffer and size parameters are ignored.

- *size* is the buffer size in bytes.

It returns 0 if successful, a non-zero value if an error occurs.

File buffering


```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w");

    if (file == NULL) {
        perror("Failed to open file");
        return 1;
    }

    // Allocate a custom buffer
    char buffer[1024];

    // Set the file stream to use full buffering with the custom buffer
    if (setvbuf(file, buffer, _IOFBF, sizeof(buffer)) != 0) {
        perror("Failed to set buffer");
    }

    // Write some data
    fputs("This is some buffered data.\n", file);

    // Close the file
    fclose(file);
    return 0;
}
```

File buffering: fflush

The fflush function is used to flush the output buffer of a file stream, meaning it forces any data that is buffered in memory to be written to the file or device.

- *int fflush(FILE *stream);*

FILE *stream is a pointer to the file stream to flush.
Common examples:

- stdout (standard output)
- stderr (standard error)
- file pointer for a specific file opened with fopen

If stream is NULL, fflush flushes all open output streams.
It returns 0 if successful, EOF (usually -1) if an error occurs.

File buffering: fflush

1. Flushing Output Streams:
   - the primary purpose of fflush is to ensure that all buffered output data is written immediately. Without fflush(stdout), the output might not appear immediately on the screen because it may remain in the buffer until a newline character is printed or the program exits.

2. Flushing a File Stream:
   - when writing to files, data is often buffered to improve performance. Using fflush ensures that data is actually written to the file before closing or continuing to the next operation.

File buffering: fflush


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

int main () {
   char buff[1024];
   memset( buff, '\0', sizeof( buff ));
   fprintf(stdout, "Going to set full buffering on\n");
   setvbuf(stdout, buff, _IOFBF, 1024);
   fprintf(stdout, "This is an example of full buffering\n");
   fprintf(stdout, "This output will go into buff\n");
   fflush( stdout );
   fprintf(stdout, "and this will appear after the program sleeps 5 seconds\n");
   sleep(5);
   return(0);
}
```

File handling: feof

The feof function is used to check if the end-of-file (EOF) indicator has been set for a file stream. This is useful when reading data from a file to determine if you have reached the end of the file.

*int feof(FILE *fp);*

It returns a non-zero value (true) if the end-of-file indicator has been set, indicating the end of the file has been reached, 0 (false) if the end of the file has not been reached.

```
while(some_condition) {
    c = fgetc(fp);
    if( feof(fp) ) {
       break ;
    }
    printf("%c", c);
}
```

File handling: feof

The feof function is typically used after an attempt to read from the file has failed. It doesn’t predict the end of the file but rather checks if the last read operation hit the end of the file.

Common Usage Mistake:

a common mistake is to use feof as a loop condition to read from a file. This can lead to errors because feof only returns true after a read operation fails due to reaching the end of the file. Instead, it's better to use the return value of the read operation (e.g., fgetc, fgets, fscanf, etc.) as the condition in a loop, and then check feof afterward to confirm if EOF was reached.

feof checks only after a failed read operation: feof does not predict if you are at the end of the file until a read operation fails due to EOF.

Line Input and Output: fgets

The standard library provides input and output functions for managing reading and writing lines of characters. The fgets function is used to read a line or a specified number of characters from a file into a string (character array). It's part of the <stdio.h> library and is commonly used for reading lines from files or standard input.

*char *fgets(char *str, int n, FILE *fp);*

- char *str is a pointer to an array of characters where the read string will be stored.
- int n is the maximum number of characters to read, including the null terminator ('\0'). fgets reads up to n-1 characters and stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. The resulting line string is terminated with '\0'.

Line Input and Output

- FILE *stream is the file stream to read from (e.g., stdin for standard input or a file opened with fopen).

On success, the function returns the same str parameter passed as argument. If the end-of-file is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned. If an error occurs, a null pointer is returned.

fgets reads characters from the specified file until:

- it reaches the newline character (\n).
- it reaches the end-of-file (EOF).
- it reads n-1 characters.

fgets includes the newline character (\n) in the string if it encounters it before reaching the limit. It always adds a null terminator ('\0') to the end of the string.

Line Input and Output


```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    char buffer[100];
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("%s", buffer);  // Print each line read from the file
    }

    fclose(file);
    return 0;
}
```

Line Input and Output: fputs

The fputs function in C is used to write a string (a null-terminated character array) to a file or to the standard output. It's part of the <stdio.h> library and provides a straightforward way to write text data without formatting.

*int fputs(const char *str, FILE *fp);*

- const char *str is a pointer to the string that will be written to the file. The string should be null-terminated.
- FILE *stream is the file stream where the string will be written. This can be stdout (for standard output) or a file opened with fopen in write or append mode.

It returns a non-negative integer on success and EOF (usually -1) if an error occurs.

Line Input and Output


```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Write a string to the file
    if (fputs("Hello, World!\n", file) == EOF) {
        perror("Error writing to file");
    }

    fclose(file);
    return 0;
}
```

Line Input and Output

No Newline Automatically Added: unlike puts, fputs does not add a newline character at the end of the string. If you want to write a newline, you must include \n at the end of your string.

Null-Terminated Strings: the string must be null-terminated; otherwise, fputs will continue reading memory beyond the intended data until it encounters a null character, which may cause undefined behavior.

File handling: tmpnam

Some useful functions for managing temporary files / filenames.

*char *tmpnam(char *str);*

The tmpnam function in C generates a unique temporary filename that can be used for creating a temporary file. It is defined in the <stdio.h> library.
str is a pointer to a character array where the generated filename will be stored. If str is NULL, tmpnam uses an internal static buffer to store the filename and returns a pointer to that buffer. If the function fails to create a suitable filename, it returns a null pointer.

File handling

Notes:

- The generated filename is unique, but it is not automatically created as a file on the filesystem. You must explicitly open or create the file using this name.
- The returned name is unique for the duration of the program run, but it's possible that other programs could create a file with the same name between calls, leading to potential conflicts.
- tmpnam is generally considered unsafe for creating temporary files, as it can be vulnerable to security issues (e.g., race conditions). The tmpfile() function is usually safer as it directly creates a temporary file.

File handling

Example usage:

```
#include <stdio.h>

int main() {
    char tempName[L_tmpnam];  // Buffer for temporary filename
    char *name;

    // Generate a unique temporary filename
    name = tmpnam(tempName);

    if (name != NULL) {
        printf("Temporary filename: %s\n", name);
    } else {
        perror("Failed to generate a temporary filename");
    }
    return 0;
}
```

File handling: tmpfile

For safer handling of temporary files, consider using *tmpfile()* which creates a unique temporary file that is automatically deleted when closed.

*FILE *tmpfile(void);*
The tmpfile function creates a unique temporary file in binary read/write mode ("wb+"). This file is automatically deleted when it is closed (with fclose) or when the program terminates. tmpfile is part of the <stdio.h> library
If successful, tmpfile returns a pointer to a FILE object that represents the temporary file created. If it fails (e.g., due to a lack of permissions or space), it returns NULL.

File handling

Characteristics:

- Automatic Deletion, The temporary file is deleted automatically when it is closed with fclose() or when the program terminates. This makes tmpfile safer and more convenient than tmpnam since it avoids filename conflicts and is less prone to security vulnerabilities.
- Binary Mode. The file is opened in binary mode ("wb+"), meaning it can read and write binary data. However, text data can also be written and read as long as it is treated appropriately.
- It is useful for storing temporary data that you don’t need to retain after the program finishes.

File handling


```c
#include <stdio.h>

int main() {
    FILE *tempFile = tmpfile();  // Create a temporary file

    if (tempFile == NULL) {
        perror("Failed to create temporary file");
        return 1;
    }

    // Write data to the temporary file
    fputs("This is a temporary file.\n", tempFile);

    // Rewind the file pointer to the beginning of the file
    rewind(tempFile);

    // Read and print the contents
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), tempFile) != NULL) {
        printf("%s", buffer);
    }

    // Close the temporary file, automatically deleting it
    fclose(tempFile);

    return 0;
}
```

File handling: remove

Some useful functions for managing files.

*int remove(const char *filename) ; //* deletes the given filename so that it is no longer accessible, returns 0 in case of success, -1 if the command is unsuccessful.

```
int ret = remove(filename);

If (ret == 0) {
    printf("File deleted successfully");
} else {
    printf("Error: unable to delete the file");
}
```

File handling: rename

*int rename(const char *old_filename, const char *new_filename) ; //* causes the filename referred to by old_filename to be changed to new_filename. On success, zero is returned. On error, -1 is returned.

```
ret = rename(oldname, newname);

if(ret == 0) {
    printf("File renamed successfully");
} else {
    printf("Error: unable to rename the file");
  }
```

File close: fclose

*int fclose(FILE *fp); //* closes the file stream fp, all buffers are flushed, returns zero if the stream is successfully closed, otherwise EOF.

```
#include <stdio.h>
int main () {
   FILE *fp;
   fp = fopen("file.txt", "w");

   fprintf(fp, "%s", "This is an example");
   fclose(fp);

   return(0);
}
```

C library function - system()

The C library function ***int system(const char *command)*** passes the command name or program name specified by string command to the host environment to be executed by the command processor and returns after the command has been completed. It allows you to run shell commands, scripts, or other executables as if they were run directly from the command prompt (Windows) or terminal (Linux/Unix). It returns -1 on error, and the return status of the command otherwise.

The exact meaning of the return value is platform-dependent and may need to be interpreted with platform-specific macros like WEXITSTATUS on Unix.

C library function - system()

The following example uses system to list files in a directory (Unix) or display directory contents (Windows):

```
#include <stdlib.h>

int main() {
    // On Unix-based systems (Linux, macOS):
    system("ls -l");  // Lists files in long format

    // On Windows:
    // system("dir");

    return 0;
}
```

C library function - system()

How system works:

1. system opens a command shell or terminal.
2. The command specified in the command argument is executed as if it were typed directly into the shell.
3. The system function waits for the command to finish and then returns the command's exit status.

```
#include <stdlib.h>
int main() {
    // Open a URL in the default web browser
    system("start https://www.example.com");  // Windows
    // On Unix-based systems, you could use:
    // system("xdg-open https://www.example.com");

    return 0;
}
```

C library function - system()

Security Concerns:

- Using system with user-generated or dynamic input is dangerous as it opens the door to command injection attacks. For example, if you use system with user input in the command string, a malicious user might be able to run unintended commands.
- Avoid using system with untrusted input, or sanitize inputs rigorously before use.

Portability:

- Commands are OS-specific, so a command that works on Windows (like dir) may not work on Unix systems (where you'd use ls).
- For cross-platform programs, limit the use of system or add conditional compilation for OS-specific code.

C library function - system()

Blocking Behavior:

- system is synchronous and will block the calling program until the command finishes executing. This may be undesirable in applications requiring high performance or responsiveness.

Error Handling:

- Check the return value of system to determine if the command executed successfully. This can help you handle errors in executing commands.

C library function - system()


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

int main() {
    int status = system("mkdir test_directory");

    if (status == -1) {
        perror("Error executing system command");
    } else {
        printf("Command executed with exit status: %d\n", WEXITSTATUS(status));
    }

    return 0;
}
```

Error handling

C programming language does not provide direct support for error handling, but it provides access at lower level in the form of return values.
Most of the C function calls return -1 or NULL in case of any error and set an error code ***errno***. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So, a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

Error handling – program exit status

So far, we have seen that each program is assigned an input (stdin) and an output stream (stdout) and that both can be redirected or piped.
We have to know that a second output stream, called stderr, is assigned to a program in the same way that stdin and stdout are. Output written on stderr normally appears on the screen even if the standard output is redirected.
You should use stderr file stream to output all the errors.

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. EXIT_SUCCESS is a macro and it is defined as 0.
If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1.

Error handling – program exit status


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

main() {
   int dividend = 20;
   int divisor = 5;
   int quotient;
   if( divisor == 0) {
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(EXIT_FAILURE);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );
   exit(EXIT_SUCCESS);
}
```

Error handling – program exit status

A program can signal errors in two ways.

- First, the diagnostic output that goes to stderr, so it finds its way to the screen instead of disappearing down a pipeline or into an output file.

- Second, the program can use the standard library function exit, which terminates program execution when it is called. The argument of exit is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations. exit calls fclose for each open output file, to flush out any buffered output.

errno, perror() and strerror()

C programming language provides ***perror()*** and ***strerror()*** functions which can be used to display the text message associated with ***errno***.

The C library function *void perror(const char *str)* prints a descriptive error message to stderr. First the string str is printed, followed by a colon then a space and then the textual representation of the current errno value.

The C library function *char *strerror(int errnum)* searches an internal array for the error number errnum and returns a pointer to an error message string. The error strings produced by strerror depend on the developing platform and compiler. This function returns a pointer to the error string describing error errnum.

errno, perror() and strerror()

Let's simulate an error condition and try to open a file which does not exist.

```
FILE *pf;
pf = fopen ("unexist.txt", "rb");

if (pf == NULL) {
    fprintf(stderr, "Value of errno: %d\n", errno);
     perror("Error printed by perror");
    fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
}
```

Value of errno: 2

Error printed by perror: No such file or directory

Error opening file: No such file or directory

C - Variable Arguments

Sometimes, you may need a function, which can take a variable number of parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation, and you are allowed to define a function which can accept variable number of parameters based on your requirement. In C, you can create functions that accept a variable number of parameters using variadic functions. The C Standard Library provides macros in <stdarg.h> to handle variable arguments. The most common example is the printf function, which can accept a varying number of arguments.

```
return_type funct(int, ... ) {
   .
   .
   .
}
```

C - Variable Arguments

The function funct() has its last parameter as ellipses, i.e. three dots (...) and the one just before the ellipses is always an int which will represent the total number variable parameters passed. To use such functionality, you need to make use of ***stdarg.h*** header file which provides the functions and macros to implement the functionality of variable arguments.

Let's see the 5 steps to be followed to implement such functionality of variable arguments.

C - Variable Arguments

1. Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of parameters.

2. Create a ***va_list*** type variable in the function definition. This type is defined in stdarg.h header file.

3. Use int parameter and ***va_start*** macro to initialize the va_list variable to a parameter list. The macro va_start is defined in stdarg.h header file.

4. Use ***va_arg*** macro and va_list variable to access each item in parameter list.

5. Use a macro ***va_end*** to clean up the memory assigned to va_list variable.

C - Variable Arguments

Following the 5 steps we can write down a simple function which can take the variable number of parameters and return their average.

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

double average(int num,...) {

   va_list valist;
   double sum = 0.0;
   int i;

   /* initialize valist for num number of arguments */
   va_start(valist, num);
   /* access all the arguments assigned to valist */
   for (i = 0; i < num; i++) {
      sum += va_arg(valist, int);
   }

   /* clean memory reserved for valist */
   va_end(valist);

   return sum/num;
}
```

```
int main() {
   printf("Average of 2, 3, 4, 5, 6 = %f\n", average(5, 2,3,4,5,6));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

C - Variable Arguments

The va_start() macro initializes valist for subsequent use by va_arg() and va_end() and must be called first.
The parameter num is the name of the last parameter before the variable argument list, that is, the last parameter of which the calling function knows the type.
The va_arg() macro expands to an expression that has the type and value of the next argument in the call. The argument ap is the va_list ap initialized by va_start(). Each call to va_arg() modifies valist so that the next call returns the next argument. The argument type is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to type.

C - Variable Arguments

The first use of the va_arg() macro after that of the va_start() macro returns the parameter after num. Successive invocations return the values of the remaining parameters.

If there is no next parameter, or if type is not compatible with the type of the actual next parameter, random errors will occur.

If valist is passed to a function that uses va_arg(valist,type) then the value of valist is undefined after the return of that function.

C - Variable Arguments

va_end()

Each invocation of va_start() must be matched by a corresponding invocation of va_end() in the same function. After the call va_end(valist) the variable valist is undefined. Multiple traversals of the list, each bracketed by va_start() and va_end() are possible. va_end() may be a macro or a function.

C - Variable Arguments

If you want to support different types, you need a mechanism (e.g., format specifiers) to indicate the types of arguments, similar to printf.

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

void print_values(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);

    while (*fmt != '\0') {
        if (*fmt == 'd') {              // Format specifier 'd' for integer
            int i = va_arg(args, int);
            printf("%d ", i);
        } else if (*fmt == 'f') {       // Format specifier 'f' for double
            double d = va_arg(args, double);
            printf("%f ", d);
        }
        fmt++;                          // Move to next character in fmt string
    }

    va_end(args);
    printf("\n");
}

int main() {
    print_values("dfd", 42, 3.14, 7);     // Expected output: "42 3.140000 7"
    print_values("ffd", 2.71, 1.61, 12);  // Expected output: "2.710000 1.610000 12"
    return 0;
}
```

C - Variable Arguments

Important Points

Type Consistency: All arguments in a variadic function are assumed to have known types; otherwise, you risk undefined behavior. In the example, we assume all extra arguments are of type int.

No Type Checking: Unlike functions with a fixed parameter list, variadic functions don’t provide compile-time type checking for additional arguments.

Argument Limitations: You can’t pass structures by value or other complex types as variadic arguments reliably, as this may lead to portability issues.