COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2023-January 2024

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

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

- *int setvbuf(FILE *fp, char *buffer, int mode, size_t size);* // defines how a file stream should be buffered.
  - *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* specifies a mode for file buffering –
    - *_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 and the buffer is empty.

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 and buffer is empty.
- *_IONBF* (No buffering). No buffer is used. Each I/O operation is written as soon as possible. The buffer and size parameters are ignored.

- *size* is the buffer size in bytes.

- *int fflush(FILE *stream);* // flushes the output buffer of a stream, returns a zero value on success otherwise EOF.

File buffering


```
#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

*int feof(FILE *fp);* // tests the end-of-file indicator for the file stream fp, returns a non-zero value when end-of-file indicator associated with the stream is set, otherwise zero.

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

Line Input and Output

The standard library provides input and output functions for managing reading and writing lines of characters.

*char *fgets(char *str, int n, FILE *fp);* // reads a line from the specified file stream and stores it into the string pointed to by str.
It 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'.
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.

Line Input and Output

*int fputs(const char *str, FILE *fp);* // writes a string (which needs not contain a newline) to the specified file stream up to but not including the null character, returns a non-negative value on success, otherwise EOF on error.

```
#include <stdio.h>
int main () {
   FILE *fp;
   fp = fopen("file.txt", "w+");
   fputs("This is c programming. ", fp);
   fputs("This is a system programming language.", fp);
   fclose(fp);
   return(0);
}
```

File handling

Some useful functions for managing temporary files / filenames.

*FILE *tmpfile(void);* // creates a temporary file in binary update mode (wb+), the temporary file created is automatically deleted when the stream is closed (fclose) or when the program terminates, returns a stream pointer to the temporary file created, if the file cannot be created, then NULL is returned.

*char *tmpnam(char *str);* // generates and returns a valid temporary filename which does not exist. If str is NULL then it simply returns the tmp file name. If the function fails to create a suitable filename, it returns a null pointer.

File handling

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

*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

*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 returns -1 on error, and the return status of the command otherwise.

```
int main () {
   char command[50];

   strcpy( command, "ls -l" );
   system(command);
   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.

```
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.