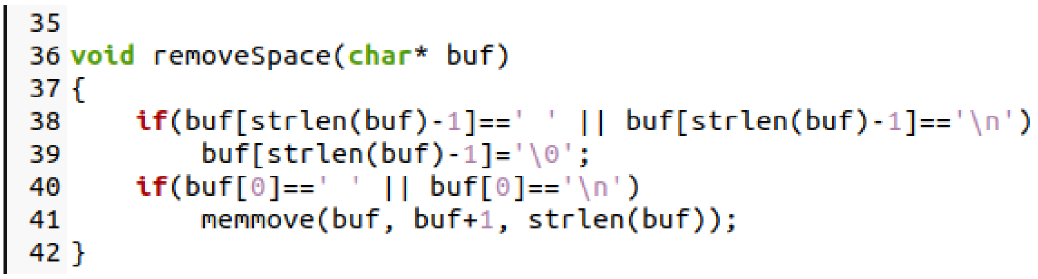
- **<u>removeSpace</u>**



```c

void removeSpace(char* buf)
{
    if(buf[strlen(buf)-1]==' ' || buf[strlen(buf)-1]=='\n')
        buf[strlen(buf)-1]='\0';
    if(buf[0]==' ' || buf[0]=='\n')
        memmove(buf, buf+1, strlen(buf));
}
```

The function "remove space" takes a char pointer as input, which is assumed to point to a null-terminated string.

The function aims to remove leading and trailing whitespaces and newlines from the string

It does this by checking if the last character and the first character of the string are whitespace or newline, and if so, replaces them with null character and moves the entire string forward by one position respectively.

However, it may not handle all cases properly, like when there are multiple spaces or newlines in a row.

It also does not handle the case where the input string is empty or check if the input string is null or if the memory it points to is valid.

The function modifies the original string in-place, it does not create a copy of the original string with the leading/trailing whitespaces removed.

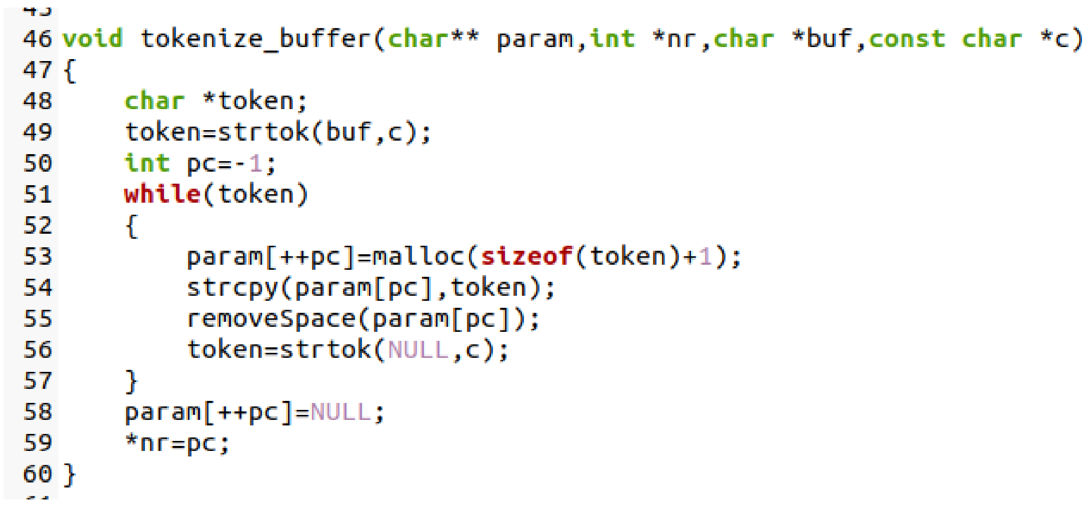
- ⇨ The main data structure used in this function is a null-terminated string, represented by a char pointer "buf."

The function checks the first and last characters of the string to see if they are whitespaces or newlines, using the "strlen" function to get the length of the string and the array indexing operator "[]" to access the individual characters.

It uses the "memmove" function to move the entire string forward by one position to remove the leading whitespace or newline character.

The function does not use any additional data structures, it only modifies the original string in place

• tokenize_buffer



```c
void tokenize_buffer(char** param,int *nr,char *buf,const char *c)
{
    char *token;
    token=strtok(buf,c);
    int pc=-1;
    while(token)
    {
        param[++pc]=malloc(sizeof(token)+1);
        strcpy(param[pc],token);
        removeSpace(param[pc]);
        token=strtok(NULL,c);
    }
    param[++pc]=NULL;
    *nr=pc;
}
```

The function "tokenize_buffer" takes four parameters: a char double pointer "param," an int pointer "nr," a char pointer "buf," and a char pointer "c."

The function uses the "strtok" function to tokenize the input string "buf" based on the delimiter "c."

It uses a while loop to iterate over the tokens and assigns them to the "param" array using the "malloc" function to allocate memory for each token.

It then uses the "trim" function to remove leading and trailing whitespaces and newlines from each token.

The last element of the "param" array is set to NULL, and the total number of tokens is assigned to "nr."

The function modifies the "param" and "nr" variables in place, and it also dynamically allocates memory to store the tokens.

- ⇨ The main data structures used in this function are:

A char double pointer "param" which is used to store the tokens after tokenizing the input string.

An int pointer "nr" which is used to store the number of tokens generated by tokenizing the input string

A char pointer "buf" which is the input string that needs to be tokenized.

A char pointer "c" which is the delimiter used to tokenize the input string

It also uses a while loop to iterate over the tokens and assigns them to the "param" array.

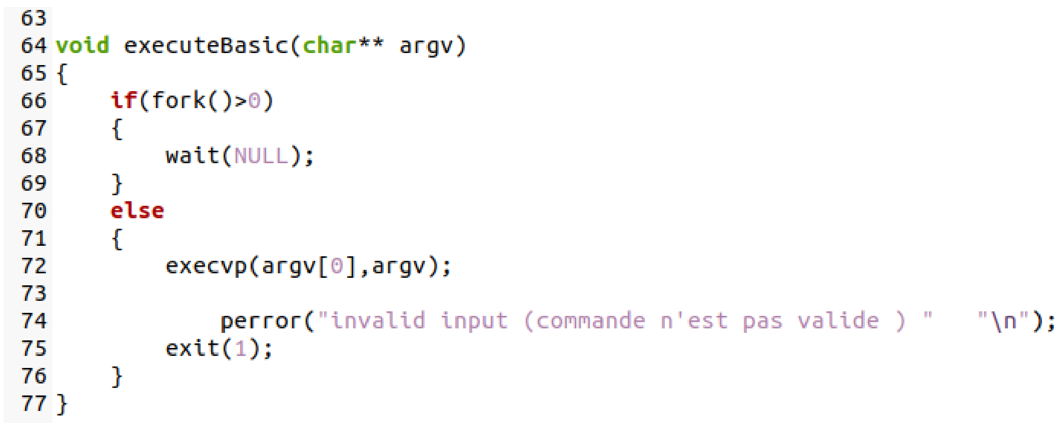
It uses the "malloc" function to dynamically allocate memory for each token.

It then uses the "trim" function to remove leading and trailing whitespaces and newlines from each token.

The last element of the "param" array is set to NULL, and the total number of tokens is assigned to "nr."

The function modifies the "param" and "nr" variables in place, and it also dynamically allocates memory to store the tokens.

• executeBasic



```

void executeBasic(char** argv)
{
    if(fork()>0)
    {
        wait(NULL);
    }
    else
    {
        execvp(argv[0],argv);

            perror("invalid input (commande n'est pas valide ) "   "\n");
        exit(1);
    }
}
```

This function creates a child process and runs a command specified in argv. The parent process waits for the child to finish and if the command is not valid, an error message is displayed and the program exits with a status code of 1

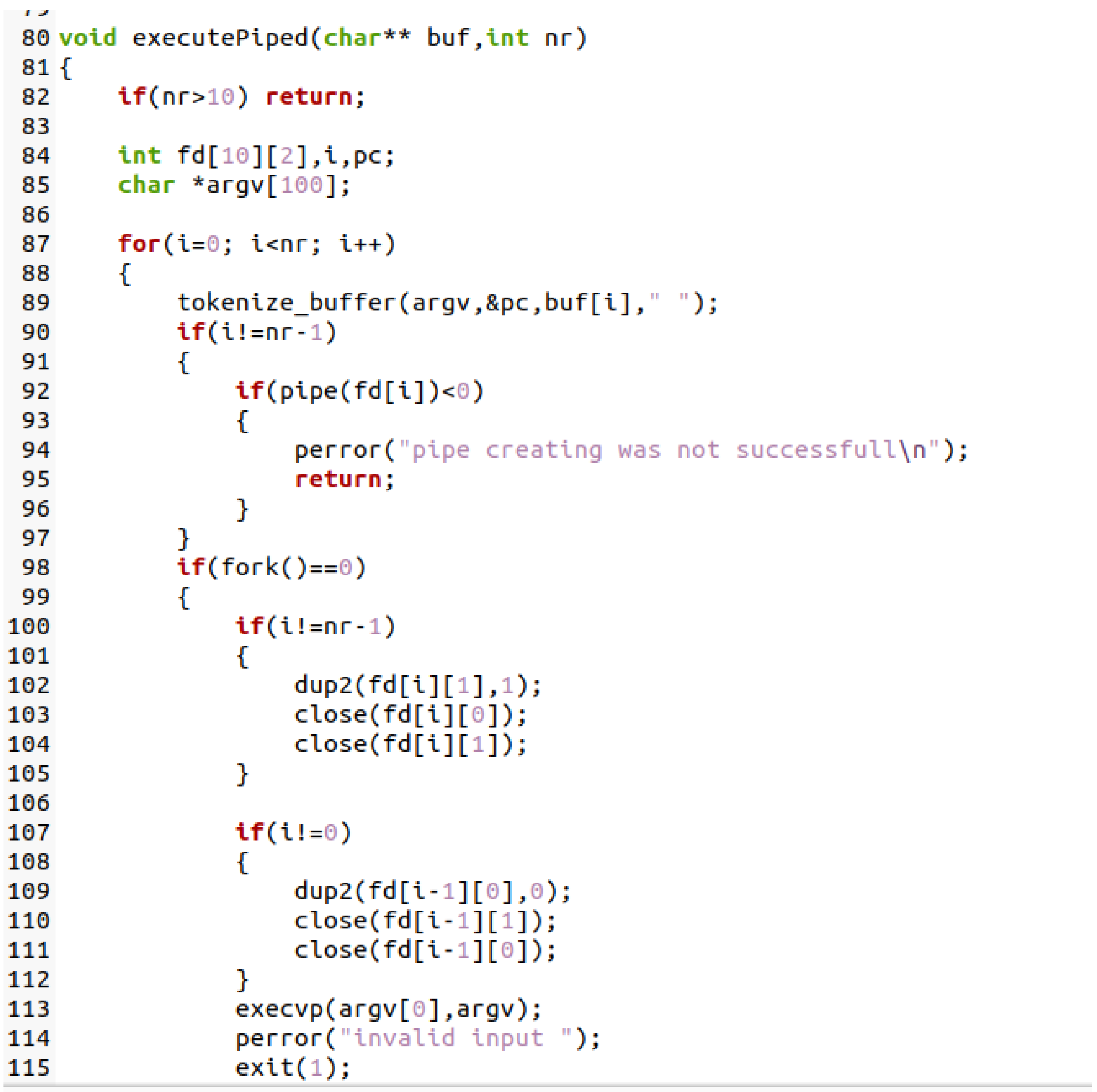
- ⇨ In this case, it is likely that argv is being used to pass the command and its arguments to the execvp function.

The function also uses two other variables, "fork()" and "wait(NULL)". fork() is a system call that creates a new process by duplicating the calling process. The wait(NULL) function suspends the execution of the calling process until one of its child processes terminates.

It also calls the execvp() is a library function that replaces the current process image with a new process image. The argv[0] is the file name or command name to execute and argv is an array of arguments passed to the command.

Finally, if the execvp function returns an error, perror() prints a message to stderr and exit(1) terminates the program with exit status 1.

• executePiped



```
void executePiped(char** buf,int nr)
{
    if(nr>10) return;

    int fd[10][2],i,pc;
    char *argv[100];

    for(i=0; i<nr; i++)
    {
        tokenize_buffer(argv,&pc,buf[i]," ");
        if(i!=nr-1)
        {
            if(pipe(fd[i])<0)
            {
                perror("pipe creating was not successfull\n");
                return;
            }
        }
        if(fork()==0)
        {
            if(i!=nr-1)
            {
                dup2(fd[i][1],1);
                close(fd[i][0]);
                close(fd[i][1]);
            }

            if(i!=0)
            {
                dup2(fd[i-1][0],0);
                close(fd[i-1][1]);
                close(fd[i-1][0]);
            }
            execvp(argv[0],argv);
            perror("invalid input ");
            exit(1);
```

This function takes two arguments, "buf" and "nr", where "buf" is a double pointer to a char and "nr" is an integer. It is used to execute multiple commands connected by pipes.

It creates a number of child processes, one for each command, and connects them using pipes. Each child process replaces its own image with the program specified in the corresponding element of "buf" by calling execvp(argv[0],argv). The function calls tokenize_buffer() to tokenize the buffer.

It uses two nested for loops, in the first for loop, it creates a pipe for each command except the last one, and in the second for loop, it creates a new child process for each command. Within this loop, it uses dup2() to redirect standard input and output to the appropriate pipe, and finally, it calls execvp() to execute the command.

If execvp() returns an error, the function calls perror() to print an error message and exit() to terminate the program with a status code of 1.

It also uses wait(NULL) to wait for the child process to complete.

It limits the number of commands to be executed to 10, if it exceeds this number the function returns without doing anything.

- ⇨ The main structure of this function is as follows:

Check if the number of commands to be executed is greater than 10. If so, return without doing anything.

Initialize variables including an array of two integers "fd" to create pipes between child processes, an iterator variable "i", a variable "pc" to keep track of the number of arguments of the command, and an array of character pointers "argv" to hold the arguments of the command.

Use a nested for loop to create pipes for each command except the last one, and to create a new child process for each command.

Within the inner for loop, use dup2() to redirect standard input and output to the appropriate pipe and execvp() to execute the command.

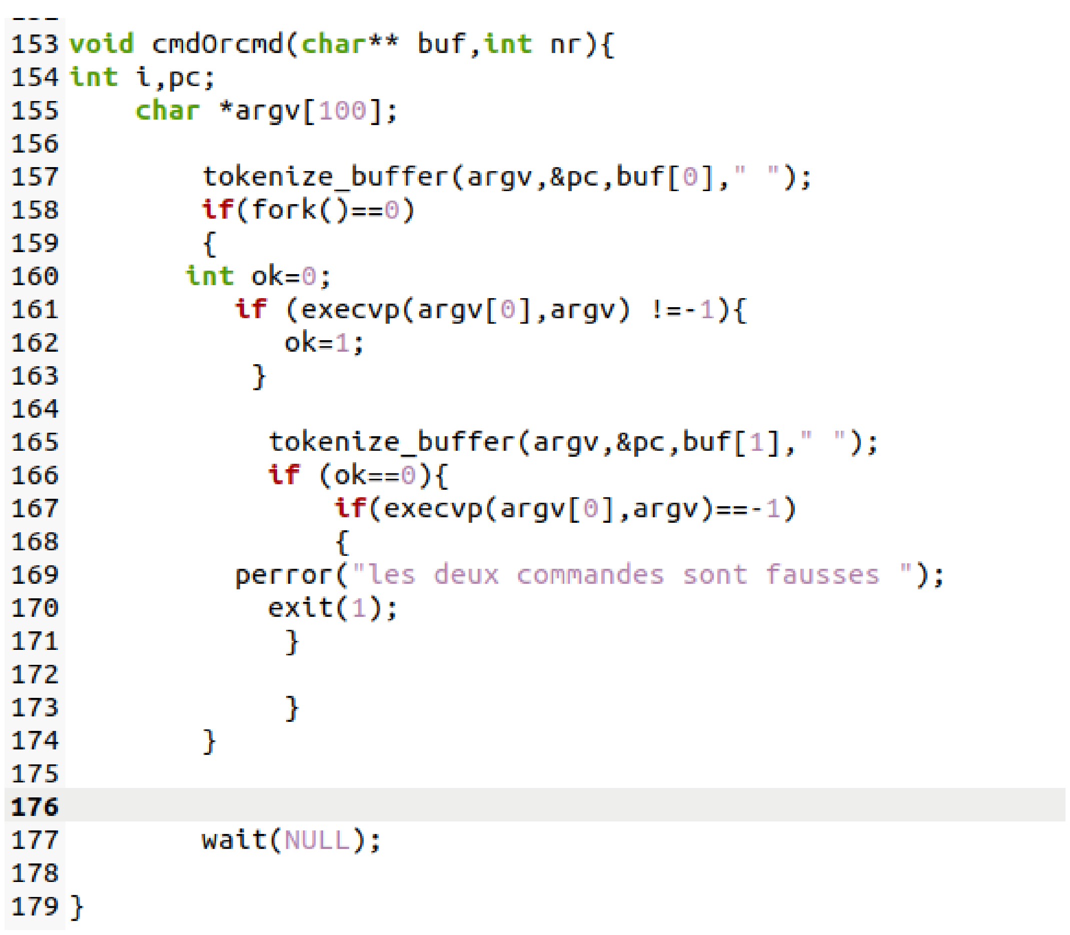
If execvp() returns an error, call perror() to print an error message and exit() to terminate the program with a status code of 1.

Use wait(NULL) to wait for the child process to complete.

Close file descriptors when they are no longer needed.

This function is intended to execute a set of commands that are connected with pipes, it creates child processes, one for each command and connect them using pipes. It uses execvp() to execute each command, by tokenizing the buffer and splitting it into arguments using tokenize_buffer() function.

•cmdOrcmd



```
void cmdOrcmd(char** buf,int nr){
int i,pc;
    char *argv[100];

        tokenize_buffer(argv,&pc,buf[0]," ");
        if(fork()==0)
        {
       int ok=0;
          if (execvp(argv[0],argv) !=-1){
             ok=1;
           }

            tokenize_buffer(argv,&pc,buf[1]," ");
            if (ok==0){
                if(execvp(argv[0],argv)==-1)
                {
          perror("les deux commandes sont fausses ");
            exit(1);
             }

             }
        }


        wait(NULL);

}
```

The function creates a child process and attempts to execute two given command buffers, if the first one fails it try the second one. It also print an error message if both fails.

⇨ The main structure of this function is as follows:

The function takes in two buffers of characters (buf[0] and buf[1]) and an integer (nr) as arguments.

It tokenizes each buffer into an array of strings (argv) and stores the number of tokens in pc.

It uses the fork() system call to create a child process.

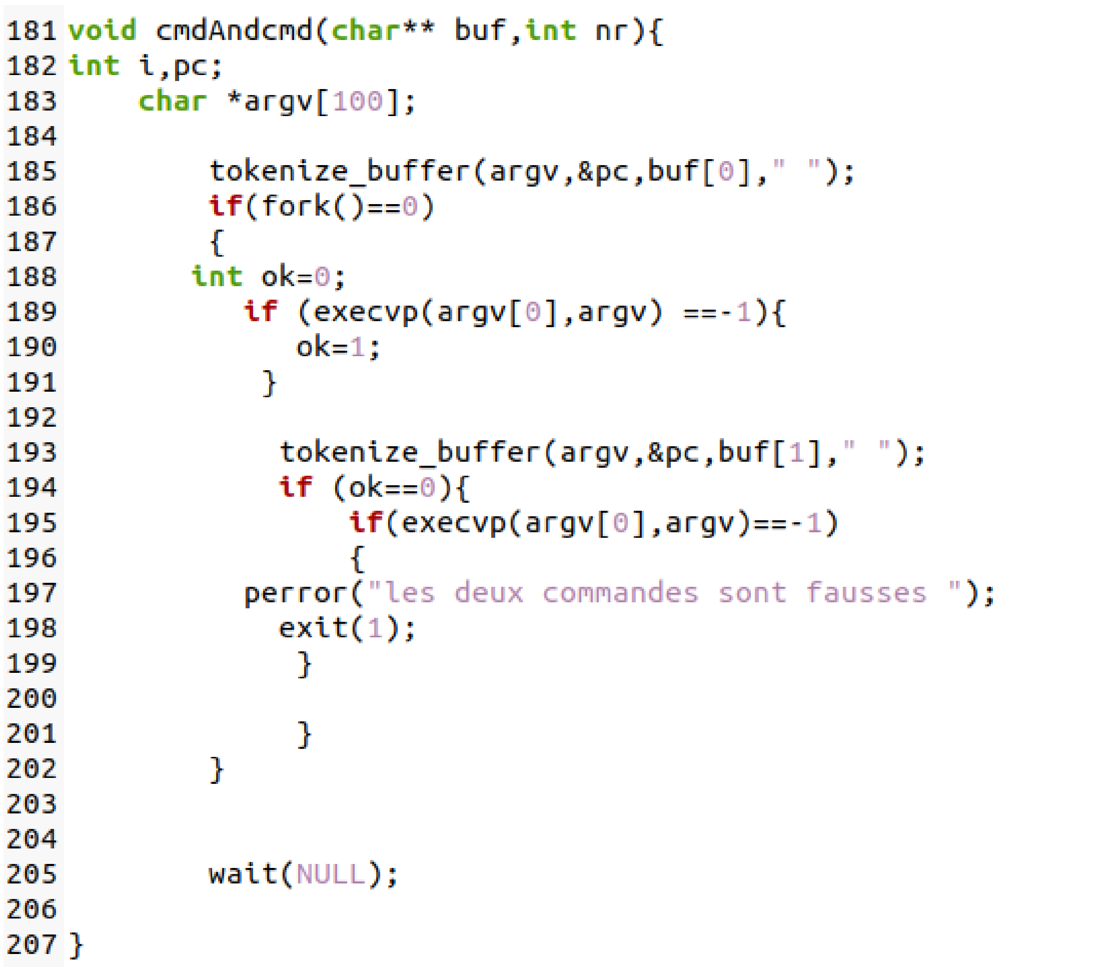
The child process then attempts to execute the first buffer using the execvp() system call.

If the execvp() call returns -1 (indicating an error), the child process then attempts to execute the second buffer using execvp().

If both execvp() calls return -1, the child process prints an error message and exits with status 1.

The parent process then waits for the child process to exit using the wait() system call.

•cmdAndcmd



```
void cmdAndcmd(char** buf,int nr){
int i,pc;
    char *argv[100];

        tokenize_buffer(argv,&pc,buf[0]," ");
        if(fork()==0)
        {
       int ok=0;
          if (execvp(argv[0],argv) ==-1){
            ok=1;
           }

            tokenize_buffer(argv,&pc,buf[1]," ");
            if (ok==0){
                if(execvp(argv[0],argv)==-1)
                {
          perror("les deux commandes sont fausses ");
            exit(1);
             }

             }
        }


        wait(NULL);

}
```

This function takes in two command buffers and an integer, creates two child processes, the first one tries to execute the first command buffer and the second one tries to execute the second command buffer if the first one fails. If both execvp() calls fail, the second child process prints an error message and exits. Both parent process wait for the child process to exit.

- ⇨ The function takes in two buffers of characters (buf[0] and buf[1]) and an integer (nr) as arguments.

It tokenizes each buffer into an array of strings (argv) and stores the number of tokens in pc.

It uses the fork() system call twice to create two child processes.

The first child process attempts to execute the first buffer using the execvp() system call.

If the execvp() call returns -1 (indicating an error), the second child process attempts to execute the second buffer using execvp().

If both execvp() calls return -1, the second child process prints an error message and exits with status 1.

Both parent process then waits for the child process to exit using the wait() system call.

the function returns nothing.