Languages for Informatics

7 – Input and Output

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Programming
  2. Variables, Data-types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. User defined datatype and data structure
  6. Input and Output
- Basic system programming in Linux (10h)

Overview

1. Strings
2. Standard Input and Output
   - Streams
   - FILE objects
   - Buffering
   - File Access
   - Character I/O
   - Formated I/O
3. Error Handling
4. Arguments
   - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

1. Strings
2. Standard Input and Output
    - Streams
    - FILE objects
    - Buffering
    - File Access
    - Character I/O
    - Formated I/O
3. Error Handling
4. Arguments
    - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

Strings and Character Arrays (1)

- String is a **1-dimensional character array** in C
- terminated by '\0' (ASCII-Code zero) language.
- Typical initialization.
  ```
  char mystring[8] =
  {'P','r','o','g','r','a','m','\0'};
  ```
- Each character is embedded in single quotes '·'.
- The size of a string must also include the space to store the end character.
- Indeed, `printf("%s",mystring)` correctly outputs

```
Program
```

Strings and Character Arrays (2)

- What does this program snippet do?

```
#include <stdio.h>
char str1[] = {'h','e','l','l','o','!'};
char str2[] = {' ','O','U','T','!','\0'};
int main() {
  printf("%s \n",str1);
  return 0;
}
```

- There is no compiling or runtime error....
- ... but printing only ends at the next NULL char.

Result

```
hello!  OUT!
```

String and Character Arrays (3)

- A different way to initialize a character array variable is to

```
char str[4] = "Pine";
char str[] = "Pine"; /*equivalent*/
```

- The double quotes "." append `NUL` to the char array **by default.**
- How to reload character arrays.

Handle with Care

The character array is not a modifiable lvalue. It can only be modified char-by-char, e.g.

```
str[0] = 'W';
```

Use library function, e.g. **`strcpy()`**, to do this job for you!

Manipulating strings

To manipulate a string, you need to access the single chars.
Example:

```
int length(char * string)
{
    int i = 0;
    while (string[i] != '\0')
        i++;
    return i ;
}
```

This function returns the length of the string passed as a parameter.
Note: the pointer is passed, hence the string can be modified and the effect seen outside the function.

String constants

- Yet another way to initialize a string

```
char *strp = "Program";
```

- Here, the string constant is treated as pointer (like the name of arrays):

```
printf("%s %s\n",strp,strp+1);
```

Result

```
Program rogram
```

- as `strp` points to the address where the array of chars ”Program” starts.

String Constants

Example

How about this?

```
char *strp = "Program";
     *strp = "fun";
```

Compiler might only raise

```
warning:  assignment makes integer from
pointer without a cast [-Wint-conversion]
```

but then, we have an undetermined behavior at run-time.

Result

```
Segmentation fault
```

Note

String constants cannot be modified.

String arrays

- A **string array** is an array of strings, each stored as a pointer to an array of chars
- Each string may be of different length.
- **Example**:

```
char str1[] = "hi";   /* length = 3 */
char str2[] = "girls"; /* length = 6 */
char str3[] = "guys";  /* length = 5 */
char *StrArray[] = {str1, str2, str1, str3};
int dim = sizeof(StrArray)/sizeof(*StrArray);
for (int n=0;n<dim;n++)
  printf("%s ", StrArray[n]);
```

String arrays

```
char str1[] = "hi";   /* length = 3 */
char str2[] = "girls";  /* length = 6 */
char str3[] = "guys";   /* length = 5 */
char *StrArray[] = {str1, str2, str1, str3};
int ArrayDim = sizeof(StrArray)/sizeof(*StrArray);
for (int n=0;n<ArrayDim;n++)
  printf("%s ", StrArray[n]);
```

Result

```
hi girls hi guys
```

Note

The array only contains the pointers, not the chars themselves. Hence, the array elements can be modified.

1. Strings
2. Standard Input and Output
   - Streams
   - FILE objects
   - Buffering
   - File Access
   - Character I/O
   - Formated I/O
3. Error Handling
4. Arguments
   - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

Standard Input and Output

Stream (1)

- ANSI C abstracts all I/O as **stream of bytes** moving into and out of a C program.
- UNIX provides a **file descriptor** (FD), an abstract indicator, to access a file or other input/output resources such as *pipe* [module 10] or *network socket* [module 12].
  - Standard input or `stdin` from default input (FD 0).
  - Standard output or `stdout` to default output (FD 1).
  - Standard error or `stderr` for error messages and other diagnostic warnings (FD 2).
- By default, all three are connected to your terminal.

Standard Input and Output

Stream (2)

It is possible to

- redirect the output streams into files,
- make **`stdin`** read from a file,
- chain one **`stdout`** in one process to **`stdin`** in another.

Standard Input and Output

`FILE` objects

- When a stream is associated with a file, the data structure `FILE` within the standard C library maintains the state.
- The `FILE` object keeps information on
  - the **file descriptor**
  - a pointer to a **buffer** for the stream
  - the buffer size
  - a counter for the actual number of chars in the buffer
  - an error **flag**
  - an End-Of-File (EOF) flag

Note

The `FILE` object is transparent to the app.

Standard Input and Output

Buffering (1)

- **Buffering** is used to minimize the number of read/write calls.
- The standard C library `stdio` supports 3 types of streaming: block buffered, line buffered and unbuffered.
  - **Block 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.

Standard Input and Output

Buffering (2)

- **Line buffering** – On output, data is written when a newline character is inserted into the stream or when the buffer is full, what so ever happens first. On Input, the buffer is filled till the next newline character when an input operation is requested and buffer is empty.
  - Unix convention is that `stdin` and `stdout` are line-buffered when associated with a terminal, and fully buffered otherwise.
- **No buffering** – No buffer is used. Each I/O operation is written as soon as possible. The buffer and size parameters are ignored.
  - Unix convention is that `stderr` is always unbuffered.

Standard Input and Output

Buffering (3)

Buffering can be modified by the following C library functions:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buffer, int mode, size_t
    size)
void setbuf(FILE *stream, char *buffer)
```

The routines can be used **after** a stream has been opened, but **before** it is read or written.

Standard Input and Output

Buffering (4)

For the **setvbuf** subroutine, the mode parameter determines how the Stream parameter is buffered:

- _IOFBF Causes input/output to be fully buffered.
- _IOLBF Causes output to be line-buffered. The buffer is flushed when a new line is written, the buffer is full, or input is requested.
- _IONBF Causes input/output to be completely unbuffered.

Note

When an unbuffered stream is specified, the buffer and size arguments are ignored.

Standard Input and Output

Buffering (5)

- the array, containing the data, points to `buffer` having size `size` is used for buffering.
- The optimum size is given by the constant `BUFSIZ` in the **`stdio.h`** library.

The **`setbuf`** subroutine is equivalent to

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

Example

```
#include <stdio.h>
#include <unistd.h>  //sleep

void printstr(char *string) {
  while (*string) {
    putchar(*string++);
    sleep(1);
  }
}
int main(void) {
  char *string="Pisa\n";
  printstr(string);      /* stdout line buffered by default */

  setbuf(stdout, NULL);
  printstr(string);      /* stdout unbuffered */
  return 0;
}
```

Flushing buffers

- At any time, it is possible to force an associated buffer to be written even if it is only partially filled.
- Syntax

```
int fflush(FILE *fp);
```

- This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set.
- Without arguments, `fflush()` flushed all output streams.

File Access

Open a stream (1)

Two functions can be used to open a standard I/O stream:

1. To open a file,

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

   Example:

```
FILE *fp = fopen("myfile.txt","r");
```

2. To associate a new filename with the given open stream and at the same time to close the old file in the stream,

```
FILE *fp = freopen(const char *filename, const char
    *mode, FILE *stream);
```

   Example:

```
FILE *fp=freopen("myfile.txt","r",stdout);
```

File Access

Open a stream (2)

- The argument type `const char *mode` specifies the file access mode.
- There are **r** for read, **w** for write, **a** for append at the end of the file, **r+** and **w+** for read and and, **a+** for read and/or write at the end of the file.

Summary

Restrictions	r	w	a	r+	w+	a+
File must exist already		*		*		
Previous contents of file lost		*			*	
Stream can be read	*			*	*	*
Stream can be written		*	*	*	*	*
Stream can be written only at the end			*			*

File Access

Close a stream

Any file associated with the stream can be disassociated by

```
int fclose(FILE *fp);
```

returning `0` on success.

- Any buffered output data is flushed
- any buffered input data is discarded,
- any allocated buffer is released.
- if it attempts to close a file pointer that isn't currently assigned to a file, the program likely crashes.

Single-Character
Input (1)

The following functions read the next character from the specified stream (e.g. `stdin` or a file) and advance the position indicator for the stream:

```
#include <stdio.h>

int getc (FILE *fp);
int fgetc (FILE *fp);
int getchar (void);
```

This function returns

ASCII the character read as an **unsigned char** cast to an `int`.

-1 in case of error / EOF

Single-Character
Input (2)

- **`fgetc()`** is **a function** that reads the next character from stream

Note

- The C unsigned char type is only 8 bits, ranging from 0 to 255 representing all ASCII chars.
- converted to `int` of size larger than 16 bits,
- to manage errors and `EOF` as *negative* numbers, on top of it.

- **`getc()`** is equivalent to `fgetc()` except that it may be implemented as a **macro** which evaluates stream more than once.
- **`getchar()`** is equivalent to **`getc(stdin)`**.

Single-Character

Output (1)

The following functions appends the character `c` to the specified output stream:

```
#include <stdio.h>

int putc (int c, FILE *fp);
int fputc (int c, FILE *fp);
int putchar (int c);
```

This function returns

ASCII the character read as an **unsigned char** cast to an `int`.

-1 in case of error / EOF

Formated I/O

fscanf()

So far we have seen how to read `char`.
What to do for other types like `int`, `double`, etc.?

- `int fscanf(FILE *file, const char *format, ...);`
- It return number of read items, e.g., for the statement
  `int r = fscanf(f, "%s %d %lf\n",str,&i,&d);`
- the input

  Student 27 92.3

  sets the variable $r \triangleq 3$.
- To read strings, it is necessary to respect the size of the allocated memory,

```
char str[10];
int r = fscanf(f,"%9s %d %lf\n",str,&i,&d);
```

Formated I/O

fprintf()

Analogous can be said for formatted write.

```
int fprintf(FILE *file, const *format, ...);
```

Example. Dump the numbers 1 to 20 to file.

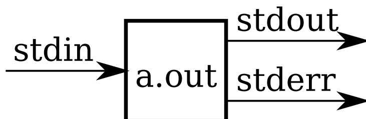
```
#include <stdio.h>
int main() {

  FILE *fp;
  int N = 20;

  fp = fopen("nums.txt", "w");
  for (int i = 1; i<=N; i++)
    fprintf(fp, "%d ", i);

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

1. Strings
2. Standard Input and Output
   - Streams
   - FILE objects
   - Buffering
   - File Access
   - Character I/O
   - Formated I/O
3. Error Handling
4. Arguments
   - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

Error Handling

`Stderr`

- When **`stdout`** is redirected to a file, and one of the files by the program cannot be accessed, all messages go to **`stdout`** (instead of the screen).
- To handle this situation better, the **output stream `stderr`** is assigned to any function, handling arguments of type `FILE*`.



- The error stream **`stderr`** is **unbuffered** i.e., it is immediately written **always to the screen** so that the user can be warned at once.

Error Handling

Application

Write formatted data to stream with **`fprintf`**

`fprintf(stderr,"<Custom message>")`

Example:

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

int main () {
   FILE *f = fopen("nonexistent.txt", "r");
   if (f == NULL) {
    fprintf(stderr,"Cannot open file \n");
    return(EXIT_FAILURE);
   }
   fclose(f);
   return(EXIT_SUCCESS);
}
```

Error Handling

Exit

- So far, we used the **return statement** to return the flow of the execution to the function from where it is called.
  - Predefined keyword.
  - is a jumping statement.
  - For a non-void function, a **return value must be returned**.
- In contrast, **`exit()`** is a **pre-defined library function** of `stdlib.h`.
  - terminates the program's execution and returns the program's **control to the OS or thread** which is calling the program.
  - Its **argument is available to whatever process** called this one, to handle success/failure.
  - At exit, the function **`exit`** may call the register program termination function **`atexit()`** to do customized clean-up. The function is thread-safe i.e., does not induce a data race.

Example

Difference between **return** and **exit**

```
#include <stdio.h>
#include <stdlib.h>
static char *message;
void cleanup(void) {
    printf("message = \"%s\"\n", message);
}
int main(void) {
    char local_message[] = "hello, world";
    message = local_message;
    atexit(cleanup);
#ifdef USE_EXIT
    puts("exit(0);");
    exit(0);
#else
    puts("return 0;");
    return 0;
#endif
}
```

Example

Difference between `return 0` and `exit(0)`

Shell

```
$ gcc -DUSE_EXIT exit.c -o exit && ./exit
exit(0);
message = "hello, world"
$ gcc exit.c -o exit && ./exit
return 0;
message = ""
```

EOF

- For each stream, two flags are maintained in the `FILE` object:
  - Flag for any error
  - Flag for end-of-file
- To test these flags, there exists the following two functions:

Advanced File Error Handling

`ferror` and `clearerr`

- The function **`ferror()`** tests for read/write errors within the stream **only after** the file has been opened.
  - Function prototype

```
int ferror(FILE *Stream);
```

  - returns non-zero value upon error.
- `int feof(FILE *Stream)` tests the end-of-file flag.
- The function **`clearerr()`** resets the error indicators for the given stream
  - Function prototype

```
int clearerr(FILE *Stream);
```

Advanced File Error Handling

Example (1)

Code snippet for testing of the two error flags:

```
if (!fgets(str, LINE_LEN, stdin)) {
    /* fgets returns NULL on EOF and error; let's see
    what happened */
    if (ferror(stdin)) {
        /* handle error */
    } else {
        /* handle EOF */
    }
}
```

Advanced File Error Handling

Example (2)

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

int main(void)
{
   FILE *stream;
   char *string = "Programming in C makes fun";
   stream = fopen("ferror.txt","r");
   if (stream == NULL) {                  // if file does not exist
      fprintf(stderr, "File not found");
      exit(1);
   }
   fputs(string, stream);
   if (ferror(stream))  {                 // if file is not writeable
      fprintf(stderr,"write error\n");
      exit(1);
   }
   if (fclose(stream))                    // if diskspace<ok, media removed etc.
      fprintf(stderr, "Writing interrupted");
   return 0;
}
```

Advanced File Error Handling

Example (2)

If the file is missing, **stderr()** writes

Shell

```
File not found
```

else if the file exists, **ferror()** writes

Shell

```
write error
```

as file is open for read-only!

```
...
stream = fopen("ferror.txt","r");
if (stream == NULL) {
    fprintf(stderr, "File not
    found");
    exit(1);
}
fputs(string, stream);
if (ferror(stream)) {
    fprintf(stderr,"write error\n"
    );
    clearerr(stream);
}
...
```

1. Strings
2. Standard Input and Output
   - Streams
   - FILE objects
   - Buffering
   - File Access
   - Character I/O
   - Formated I/O
3. Error Handling
4. Arguments
   - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

Command-line Arguments (1)

- When executed in a shell (sh, bash, zsh), the **return value** of the program is **stored in the variable `$?`**
- Example of the program execution with different number of arguments

```c
int main () {
    FILE *f = fopen("nonexistent.txt", "r");
    if (f == NULL) {
        fprintf(stderr,"Cannot open file \n");
        return(EXIT_FAILURE);
    }
    fclose(f);
    return(EXIT_SUCCESS);
}
```

Shell

```
$ ./stderr
Cannot open file
$ ./stderr; echo $?
Cannot open file
1
```

Command-line Arguments (2)

- There is a way to pass command-line arguments to a program when it begins executing.
- When **`main`** is called with **two arguments**,
  - The number of arguments is stored in `int argc` (number plus one).
  - and `char *argv` is **a pointer to an array of char. strings** that contain the arguments, one per string.

Example (1)

Concatenate strings

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  if (argc == 1) {
     fprintf(stderr, "Usage: %s <string 1> ... <string N>\n", argv[0]);
     exit(1);
    }
  for (i = 1; i < argc; i++) {
    printf("Arg.No %d%s",i,(i < argc-1) ? "**" : "++");//if-else
    printf("%s\n", argv[i]);
   }
  return 0;
}
```

Shell

```
$ ./concat
Usage:  ./concat <string 1> ...  <string N>
$ ./concat first second third
Arg.No 1**first
Arg.No 2**second
Arg.No 3++third
```

Example (2)

Write to and read chars from file

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

int main(int argc, char *argv[]) {

  FILE *fp;
  char buf[] = "Programming in C makes fun
      !";
  int len = strlen(buf);

  if (argc != 2) {
     fprintf(stderr, "Usage: %s <filename
     >\n", argv[0]);
     return 1;
     }
  else { fp = fopen(argv[1], "w");

     for (int i = 0; i<len; i++)
       fputc(buf[i],fp);
    fclose(fp);
    }

return 0;
}
```

Shell

```
$ ./WriteCharToFile chars.txt
```

Example (2)
Write to and read chars from file

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  int c;

  if (argc != 2) {
     fprintf(stderr, "Usage: %s <filename
     >\n", argv[0]);
     return 1;
     }
  else { fp = fopen(argv[1], "r");

    while ((c = fgetc(fp)) !=  EOF)
       fputc(c, stdout);

    fclose (fp);
    }

return 0;
}
```

Shell

```
$ ./ReadCharFromFile chars.txt
Programming in C makes fun!
```

1 Strings
2 Standard Input and Output
- Streams
- FILE objects
- Buffering
- File Access
- Character I/O
- Formated I/O
3 Error Handling
4 Arguments
- Command-line Arguments
5 Line Input and Output
6 Miscellaneous

Line Input and Output

Input

The standard C library function

```
char *fgets(char *str, int n, FILE *stream);
```

- reads a line from the specified 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.
- It returns the same same string if no error, and the null pointer otherwise.

Line Input and Output

Output

The standard C library function

```
char *fputs(const char *str, FILE *stream);
```

- writes a string to the specified stream up to but not including the null character.
- The array `str` must contain the null-terminated sequence of characters to be written.
- and `stream` is the pointer to a `FILE` object that identifies the stream where the string is to be written.

Example

Earlier, we wrote a sequence of integers to the file `nums.txt`.

Shell

```
$ cat nums.txt
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20
```

Let us read out the numbers in line-buffered and in fully-buffered mode, respectively.

Demonstration

```
#include <stdio.h>
#include <unistd.h> //sleep
#include <stdlib.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  int BUFSIZE = 25;
  char buffer[BUFSIZE];

  if (argc != 3) {
     fprintf(stderr, "Usage: %s <mode> <filename>\n", argv[0]);
     return 1;
     }
  else { fp = fopen(argv[2], "r");

    if (atoi(argv[1]) == 0)
      setvbuf ( fp , buffer, _IOLBF , BUFSIZE );  /* line buffered mode */
    else
      setvbuf ( fp , buffer, _IOFBF , BUFSIZE );  /* fully buffered mode */
    while (fgets(buffer, sizeof(buffer), fp) != 0)
    {
    fputs(buffer, stdout);
    fflush(fp);
    sleep(1);
    }

    fclose (fp);
```

1. Strings
2. Standard Input and Output
   - Streams
   - FILE objects
   - Buffering
   - File Access
   - Character I/O
   - Formated I/O
3. Error Handling
4. Arguments
   - Command-line Arguments
5. Line Input and Output
6. Miscellaneous

Functions in `string.h`

Many functions to manipulate strings are already defined in **`string.h`**. To use them, just include the library.

- `int strlen(const char* str)`: returns the length of the string parameter;
- `int strcmp(const char *s1, const char *s2)`: compares lexicographically the two strings, and returns:
  - 0 if they are equal;
  - a value < 0 if $s1 < s2$;
  - a value > 0 if $s2 > s1$;

Functions in `string.h` (2)

- `char *strcpy(char *s2, const char *s2)`: copies the string *s*2 in *s*1, including the end char. The previous content of *s*1 is lost. *s*1 has to be big enough to contain *s*2. It returns the pointer to *s*1;
- `char *strcat(char *s1, const char *s2)`: concatenates strings *s*1 and *s*2, adding a copy of *s*2 to the end of *s*1. Enough space has to be allocated, and it returns the pointer to *s*1;
- many of these functions have a version that works on up to *n* out of all chars: `strncmp`,`strncpy`, `strncat`....

Functions in `string.h` (3)

- `char *strtok(char *s, const char *delim)`: splits *s* in substrings (*token*) using the chars specified in *delim* as separators. To correctly use the function:
  - at the first call you need to pass the pointer to *s*. It returns the pointer to the first token;
  - in following calls, the function takes in input a pointer to `NULL` and returns at each invocation the pointer to the next token. If no token can be computed, it returns `NULL`.

  Warning: the original string is modified from the function (hence, destroyed).

strtok example

```
#include <string.h>
#include <stdio.h>
int main () {
   char str[80] = "Hi - PDS class - strtok test.";
   const char s[2] = "-"; char *token;

   token = strtok(str, s); /* get the first token */

   /* walk through other tokens */
   while( token != NULL ) {
      printf( " %s\n", token );
      token = strtok(NULL, s);
   }
   return 0;
}
```

You get as tokens: `Hi , PDS class , strtok test.`

strtok example (2)

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "hi everyone. We are having fun!";
    char* p;
    p = strtok(s, " "); // first call
    while (p != NULL)
    {
        printf("%s\n", p);
        p = strtok(NULL, " ");
    }
    return 0;
}
```

You get as tokens: `hi`, `everyone.`, `We`, `are`, `having`, `fun!`

Quiz

```
#include <stdio.h>
int main()
{
    printf("Hello  ");
    printf("everybody");
    getchar();
    return 0;
}
```

What will be the output?

1. Hello everybody
2. Hello
   everybody
3. nothing
4. Depends on terminal configuration