京都先端科学大学

KYOTO UNIVERSITY OF ADVANCED SCIENCE

京都発世界人財

Information Processing 2　13

standard input/output, formatted input/output
variadic functions, line input/output
file access, error handling
useful library functions

Ian Piumarta

Faculty of Engineering, KUAS

last week: recursive structures, bit fields, unions

Chapter 6. Structures
6.5 Self-referential Structures
6.6 Table Lookup
6.7 Typedef
6.8 Unions
6.9 Bit-fields

`sizeof()` and counting number of array elements
using `typedef` for structures
`union` and bit fields
data structure: linked list, binary tree

this week: input and output

`getchar()`, input redirection, pipes
`putchar()`, output redirection, pipes
`lower()`
`printf()`
variadic functions
home-made `minprintf()`
`scanf()`
`FILE` access, `stdin`, `stdout`
writing the `cat` program
`stderr` and `exit()`
line input and output
string, char, memory functions

Chapter 7. Input and Output
7.1 Standard input and output
7.2 Formatted output – `printf`
7.3 Variable-length argument lists
7.4 Formatted input – `scanf`
7.5 File access
7.6 Error handling – `stderr` and `exit()`
7.7 Line input and output
7.8 Miscellaneous functions

standard input and output

simplest input mechanism: `int getchar(void)`

- reads one character from the *standard input* (normally the keyboard)
- returns `EOF` when the end of file is reached

simplest output mechanism: `int putchar(int c)`

- writes one character to the *standard output* (normally a ‘terminal’ window)
- returns `c` or `EOF` if an error occurs

the source of input and destination of output can often be changed using *redirection*

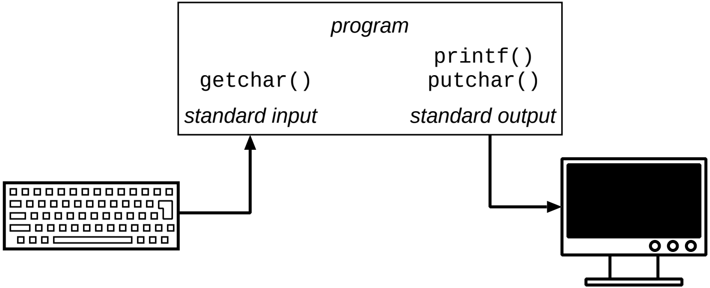
- on Linux (including WSL) and Mac this works as described below
- on Windows it depends on which shell or console you are using

input from keyboard, output to screen

standard input is usually the keyboard

standard output is usually the screen (terminal/console window)

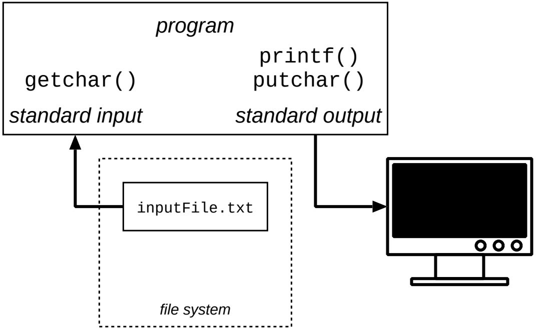
```
$ ./program
```



input from a file, output to screen

standard input can be *redirected* to come from a file instead of from the keyboard

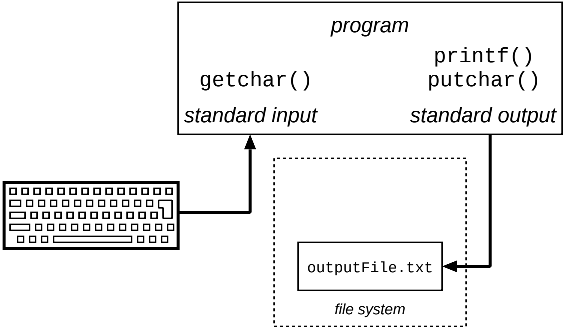
```
$ ./program < inputFile.txt
```



input from keyboard, output to a file

standard output can be *redirected* to go to a file instead of to the screen

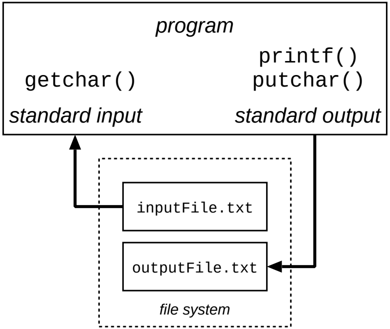
```
$ ./program > outputFile.txt
```



input from a file, output to a file

standard input and output can be *redirected* at the same time

```
$ ./program < inputFile.txt > outputFile.txt
```



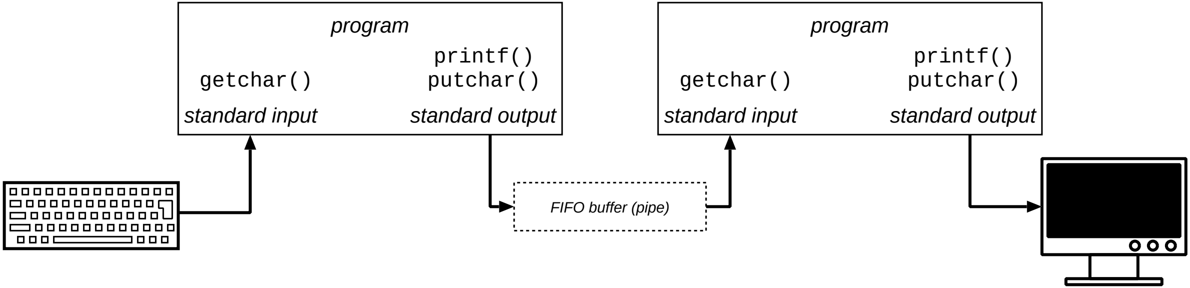
piping output to input of another program

standard output can be *piped* into the standard input of another program

e.g: output generated by `programA` will be read as input by `programB`

a small *buffer* allows `programA` to continue writing even when `programB` is busy

```
$ ./programA | ./programB
```



formatted output

general printing mechanism: `int printf(char *format, ` *arguments...*`)`

- prints `format` on the *standard output*
- embeds zero or more *arguments*, according to '%' conversions in `format`

a conversion specifier contains: % *flags width* . *precision conversion*

flags are zero or more of

−	left-justify the output (default is right-justified)
+	always print a sign character, either '+' or '−'
	(blank) leave a space before a positive number (instead of the '+' sign)
0	pad the output on the left with leading zeros (instead of spaces)

width is the minimum width of the output (padding will be added)

precision applied to	string:	maximum number of characters to print
	floating-point:	number of digits after the decimal point
	integer:	minimum number of digits

formatted output

a conversion specifier contains: % *flags width* . *precision conversion*

conversion	*type*	*representation*
`d i`	`int`	signed decimal integer
`u`	`int`	unsigned decimal number
`o`	`int`	unsigned octal integer (without leading zero)
`x X`	`int`	unsigned hexadecimal number (without leading `0x`)
`c`	`int`	single character
`s`	`char *`	strings (nul-terminated character array)
`f`	`double`	[−]`m.dddddd` where the number of `d`s is given by the precision
`e E`	`double`	[−]`m.dddddde±xx`
`g G`	`double`	`%e` if the exponent is small, otherwise `%f`
`p`	pointer	unsigned hexadecimal (usually)
`%`		print a literal ‘%’ character

formatted output

`int` size can be specified using a character before the conversion

`h`	`short` integer
`l`	`long` integer
`z`	the type appropriate for a `size_t` or `ssize_t` argument

width or *precision* can be specified as '`*`'

- the value is read from the next argument, which must be an `int`

some useful applications of '`*`' using strings:

```
int n = 10;
printf("%*s", n, "");  // print exactly n spaces
printf("%.*s", n, s);  // print at most n characters from string s
```

(the second example can be used to print `char` arrays that are not nul-terminated)

formatted output examples

when printing the string “hello, world” (12 characters)

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

formatted output hints and tips

beware of using `printf` to print a string without conversions

```
char *s = "unknown or unpredictable characters";

printf(s);          // FAILS if s contains a % character
printf("%s", s);    // correct and safe
```

`snprintf` is like `printf` except that it stores its output in a character array

```
int snprintf(char *array, size_t size, char *fmt, args...)
```

- `array` is where the formatted output string will be stored
- `size` is the size of the `array`
- `fmt` and `args`... are format string and arguments, as in `printf`

`snprintf` will not write more than `size` bytes into `array` (including nul terminator)

- return value is the number of bytes needed to store the entire output
- even if `size` was too small (which lets you grow `array` to the correct size)

formatted output hints and tips

dynamically sizing the buffer for `snprintf`:

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

int main(int argc, char *argv[]) {
    char s[] = "my string";
    int  i   = 42;

    char *buffer = 0; // buffer for snprintf output
    size_t  size = 0; // size of buffer

    size = snprintf(buffer, size, "%i %s\n", i, s) + 1; // + 1 for NUL
    printf("allocate buffer of %zu bytes\n", size);
    buffer = malloc(size); // guaranteed to be correct size
    snprintf(buffer, size, "%i %s\n", i, s);
    printf("result: %s", buffer);
    free(buffer);

    return 0;
}
```

variable-length arguments lists

`printf` accepts a variable number of arguments of unknown types

- but it can fetch those arguments correctly, under the control of the format string

functions that take a variable number of arguments are called *variadic*

the correct declaration for `printf` is: `int printf(char *format, ...)`

- the `int` result is the number of characters printed
- the `char *format` controls the number and types of the following arguments
- the `...` means the number and types of remaining arguments may vary
  i.e., the function is variadic

to recover the `...` arguments, `printf` uses several macros defined in <`stdarg.h`>

these macros allow `printf` to read each of the 'anonymous' arguments, one at a time

variable-length arguments lists

```
void f(int a, int b, ...)
  va_list ap;
  va_start(ap, b);
  while (more args) {
    var = va_arg(ap,type);
  ...
  }
  va_end(ap);
}
```

`va_list`

- is a type, similar to a ‘pointer’ to the next argument
- used to declare a variable, conventionally called `ap`

`va_start(ap,` *argName*`)`

- initialises `ap` to point the argument following *argName*

`va_arg(ap,` *type*`)`

- fetches the next argument, which is assumed to be of the given *type*

`va_end(ap)`

- frees any resources being used by `ap`

a minimal `printf` to demonstrate variadic functions

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

int myprintf(char *fmt, ...)
{
  int n = 0;                          // number of characters printed
  va_list ap;                         // argument `pointer'
  va_start(ap, fmt);                  // point ap at the argument following fmt

  for (char *p = fmt;  *p;  ++p) {    // there is still more stuff to print
    if ('%' != *p || !p[1]) {         // not a conversion specification
      putchar(*p);                    // output the character verbatim
      continue;                       // finished with this character
    }
    switch (*++p) {
      case 'd': {
        int i = va_arg(ap, int);      // fetch next argument as an 'int'
        n += printf("%d", i);
        break;
      }
```

a minimal `printf` to demonstrate variadic functions

```
      case 'f': {
        double d = va_arg(ap, double); // fetch next argument as an 'double'
        n += printf("%f", d);
        break;
      }
      case 's': {
        char *s = va_arg(ap, char *);
        while (*s) ++n, putchar(*s++);
        break;
      }
      default:
        ++n, putchar(*p);
        break;
    } // switch
  } // for
  va_end(ap); // release memory used by ap
  return n;
}
```

```
int main()
{
  myprintf("%%<%s> [%d] {%f}\n",
           "hello", 42, 123.456);
  return 0;
}
```

formatted input: `scanf`

the function `scanf` performs `printf`-like conversions in the opposite direction

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

- reads characters from standard input, which must match `format`
- `format` can contain conversion specifications
- results of conversions are stored in variables using remaining *pointer* arguments
- conversion stops when the input does not match `format`
- return value is the number of successful conversions performed

```
#include <stdio.h>

int main()
{
    int i;

    while (1 == scanf("%d", &i)) // %d result needs a POINTER to an int variable
        printf("%d\n", i * i);

    return 0;
}
```

formatted input: `sscanf`

conversion	*argument*	*effect*
`d`	`int *`	read a decimal integer
`i`	`int *`	read an integer in decimal, octal (leading `0`), or hexadecimal (leading `0x`)
`o`	`int *`	octal (without leading `0`)
`u`	`unsigned int *`	unsigned decimal integer
`x`	`int *`	hexadecimal integer (without leading `0x`)
`c`	`char *`	characters (without skipping whitespace)
`s`	`char *`	characters (skipping whitespace)
`e f g`	`float *`	floating-point number, option sign and exponent
`%`		matches a literal % character

integer conversions can be modified by `h` (`short`) or `l` (`long`)

floating-point conversions can be modified by `l` (`double`)

assignment can be suppressed with `*` (pointer argument is omitted)

a field width for `c` and `s` limits the number of characters transferred

scanning from a string

blanks in the format string skip any amount of input white space

- space, tab, newline, carriage return

the opposite of `snprintf`, the function `sscanf` reads input from a string

```
int sscanf(char *string, char *format, ...)
```

```
while (fgets(line, sizeof(line), stdin)) {
  int i[3];
  int n = sscanf(line, "%d %d %d", i, i+1, i+2);
  if (n < 1) break;
  switch (n) {
    case 1: printf("scalar   %d\n",         i[0]);             break;
    case 2: printf("2d point %d, %d\n",     i[0], i[1]);       break;
    case 3: printf("3d point %d, %d, %d\n", i[0], i[1], i[2]);  break;
  }
}
```

file access

files other than standard input and output can be accessed and created

for example: `cat` *file1* *file2* ...

- open the next file named on the command line
- copy its contents to the standard output
- close the file

before accessing any file it must be *opened*

`fopen()` opens a file and returns a *file pointer* for subsequent file access

```
#include <stdio.h>
FILE *fp;  // file pointer for access to the file
fp = fopen("test.txt", "r");  // can access test.txt contents using fp
```

note: `FILE` is a type name, not a structure tag

`fopen`

`fopen` is declared as

```
FILE *fopen(char *path, char *mode)
```

- `path` is a filename (absolute, or relative to the working directory)
- `mode` is a string that specifies what kind of access is requested

`"r"`	open the file for reading only
`"r+"`	open the file for reading and writing
`"w"`	open or create the file for writing and truncate it
`"w+"`	open or create the file for reading and writing
`"a"`	open the file for writing and append to it

- the return value is either a pointer to a `FILE` (success) or 0 (error)

an open file is often called a *stream* (think: “flow of characters” in or out)

basic input and output to files

`int getc(FILE *fp)` reads the next character from the stream `fp` and returns it, or `EOF` at end of file

`int putc(int c, FILE *fp)` writes the character `c` to the stream `fp` returning `c`, or `EOF` if an error occurs

when a C program is started, three files are automatically opened

`FILE *stdin`	is connected to the standard input
`FILE *stdout`	is connected to the standard output
`FILE *stderr`	is connected to the standard error stream

(these are all declared in `<stdio.h>`)

`getchar` and `putchar` can be macros, defined using these functions and streams:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc(c, stdout)
```

formatted I/O using file streams

`printf` and `scanf` use `stdout` and `stdin` by default

several variants of them use an explicit file stream instead:

`int fscanf(FILE *fp, char *fmt, ...)` reads from `fp` instead of `stdin`

`int fprintf(FILE *fp, char *fmt, ...)` writes to `fp` instead of `stdout`

the function `int fclose(FILE *fp)` is the opposite of `fopen`

- it closes the file `fp`, making sure all data has been written to the file

we now have everything needed to write the `cat` program:

- for each command-line argument `a`
  - open `a` for reading as a file pointer `fp`
  - while the next character `c` read from `fp` is not `EOF`
    - write `c` to `stdout`
  - close the file pointer `fp`

copying files using file streams

```
#include <stdio.h>

void filecopy(FILE *input, FILE *output)
{
  int c;
  while (EOF != (c = getc(input)))
    putc(c, output);
}

int main(int argc, char *argv[])
{
  for (int i = 1;  i < argc;  ++i) {
    char *path = argv[i];                // the file to copy to the output
    FILE *fp = fopen(path, "r");         // open for reading only
    if (!fp) {                           // open failed
      printf("%s: cannot open: %s\n", argv[0], path);
      return 1; // failure
    }
    filecopy(fp, stdout);                // copy the file to standard output
    fclose(fp);
  }

  return 0; // success
}
```

error handling

the stream `stderr` is reserved for error messages

- `stderr` is an output stream, similar to `stdout`
- by default it is connected to the screen
- it remains connected to the screen even if `stdout` is redirected

the function `void exit(int status)` can be used to terminate the program

- it behaves like '`return status;`' does inside the `main` function

using these two facilities we can improve the error handling of `cat`

```
FILE *fp = fopen(path, "r");
if (!fp) {
  fprintf(stderr, "%s: cannot open: %s\n", argv[0], path); // use stderr for errors
  exit(1); // failure
}
```

(this code will continue to work even if moved out of `main` to another function)

error handling

`int ferror(FILE *fp)` returns non-zero if an error has occurred on `fp`

`int feof(FILE *fp)` returns non-zero if end of file has occurred on `fp`

- note that the program must already have tried to read beyond the end of file
- this function does not tell you whether the next `getc(fp)` will return `EOF`

`void clearerr(FILE *fp)` clears any error (or end of file) condition on `fp`

line-oriented input and output

```
char *fgets(char *line, int size, FILE *fp)
char *fputs(char *line, FILE *fp)
ssize_t getline(char **lineptr, size_t *linesize, FILE *fp)
```

```
#include <stdio.h>  // getline()
#include <stdlib.h> // free()

int main() {
  char   *line = 0;     // allocated and set by getline()
  size_t  linesize = 0; // set by getline()

  while (getline(&line, &linesize, stdin) >= 0) // sets line and linesize
    printf("%zd: %s", linesize, line);

  if (line) free(line); // getline() allocates line by calling malloc()

  return 0;
}
```

miscellaneous functions: `ungetc`

`int ungetc(int c, FILE *fp)` pushes the character `c` back onto the stream `fp`

this is similar to the `ungetchar(int c)` function we wrote

at least one character can be pushed back onto a stream

some systems allow more than one character to be pushed back

- but portable code should assume one character is the limit

`ungetc` works with any functions that read from a `FILE *`

- `scanf()`, `getc()`, `getchar()`, etc.

this allows your program to read ‘one character too far’ (e.g., to find the end of a line, a delimiter, etc.) and then ‘undo’ the reading of that one character

miscellaneous functions: string operations

```
#include <string.h>

char *s, *t;
int c, n;
```

`strcat(s, t)`	concatenate `t` to end of `s`
`strncat(s, t, n)`	concatenate at most `n` characters of `t` to end of `s`
`strcmp(s, t)`	return negative, zero, or positive for `s < t`, `s == t`, or `s > t`, respectively
`strncmp(s, t, n)`	same as `strcmp()` but only compares first `n` characters
`strcpy(s, t)`	copy `t` to `s`
`strncpy(s, t, n)`	copy at most `n` characters of `t` to `s`
`strlen(s)`	return length of `s`
`strchr(s, c)`	return pointer to first `c` in `s`, or NULL if not present
`strrchr(s, c)`	return pointer to last `c`x in `s`, or NULL if not present

miscellaneous functions: character tests

```
#include <ctype.h>

int c;
```

`isalpha(c)`	non-zero if `c` is alphabetic, `0` if not
`isupper(c)`	non-zero if `c` is upper case, `0` if not
`islower(c)`	non-zero if `c` is lower case, `0` if not
`isdigit(c)`	non-zero if `c` is digit, `0` if not
`isalnum(c)`	non-zero if `isalpha(c)` or `isdigit (c)`, `0` if not
`isspace(c)`	non-zero if `c` is blank, tab, newline, return, form-feed, vertical tab
`toupper(c)`	return `c` converted to upper case
`tolower(c)`	return `c` converted to lower case

miscellaneous functions: mathematical operations

```
#include <math.h>
```

all angles are measured in radians (2π radians in a circle)

all arguments and results are `double`

`sin(x)`	sine of x
`cos (x)`	cosine of x
`atan2(y, x)`	arctangent of $\frac{y}{x}$
`exp(x)`	exponential function e^x
`log(x)`	natural (base e) logarithm of x $(x > 0)$
`log10(x)`	common (base 10) logarithm of x $(x > 0)$
`pow(x, y)`	x^y
`sqrt(x)`	square root of x $(x \geq O)$
`fabs(x)`	absolute value of x

miscellaneous functions: storage management

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

the type `void *` is used for 'generic' pointers (a pointer to any or unknown type)

- `void *` pointers behave as if they point to an object 1 byte large
- `void *` pointers can be converted to/from any other pointer type

`void *malloc(size_t n)` returns a pointer to a block of memory `n` bytes large

- the memory is *not* initialised (it contains random junk!)
- if the memory cannot be allocated, `0` (NULL) is returned

`void *calloc(size_t n, size_t size)` returns a pointer to memory for an array of `n` objects of the specified `size`

- the memory is cleared (filled with zeros), useful for allocating a `struct`

```
struct Foo *foop = calloc(1, sizeof(struct Foo));
```

miscellaneous functions: storage management

`void free(void *p)` frees the memory pointed to by `p`

- `p` must have been allocated by `malloc` or `calloc`
- memory can be freed in any order

`void *realloc(void *p, size_t size)` changes the size of the memory at `p`

- the memory at `p` is reallocated to contain `size` bytes of data
- the original contents of the memory are copied into the new memory
- `p` is `0` then `realloc` behaves like `malloc(size)`
- if new memory cannot be allocated then `0` (NULL) is returned
- if `p` is `0` (NULL) then `realloc` behaves like `malloc(size)`
- the memory should eventually be `free()`d, just as above

`realloc()` is very useful for buffers that have to grow larger on demand

miscellaneous functions: storage management

it is an error to access memory after it has been freed

a typical error with freeing memory is:

```
for (p = head;  p != 0;  p = p->next) // WRONG
  free(p); // this invalidates the memory at *p
```

(`p` no longer points to valid memory when `p->next` is evaluated)

the correct way is to access `p->next` *before* freeing `p`:

```
for (p = head;  p != 0;  p = q) {
  q = p->next;
  free(p);
}
```

miscellaneous functions: random number generation

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

`int rand(void)` returns a random integer between 0 and `RAND_MAX`

one way to produce random floating-point numbers in the interval $[0, 1)$ is:

```
#define frand()  ((double)rand() / ((double)RAND_MAX+1))
```

the sequence of random numbers is typically the same

`srand(unsigned)` can be used to ‘seed’ the generator for more random sequence

one way to do this (on Unix-like systems) is using the current time

```
#include <sys/time.h>
```

```
struct timeval tv;
gettimeofday(&tv, 0);
srand(tv.tv_usec); // microseconds part of current time
```

miscellaneous functions: command execution

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

`int system(char *s)` executes the command in the string `s`

- this can be written exactly as you would enter it at a shell prompt
- the result is the exit status of the command `s`

```
system("date");
```

next week we look at more operating system services

next week: operating system interface

Chapter 8. The UNIX System Interface
8.1 File descriptors
8.2 Low-level I/O – `read()` and `write()`
8.3 `open()`, `creat()`, `close()`, `unlink()`
8.4 Random access – `lseek()`
8.5 Example – an implementation of `fopen()` and `getc()`
8.6 Example – listing directories
8.7 Example – a storage allocator

`read()` and `write()`
`copy` program
`getchar()` function
`fcntl`
variadic `error()` function
implementation of `fopen()` and `getc()`
implementation of `fopen()`
`dirent` and directory listing
`fsize` and listing directories
diy storage allocator

assignment

please download assignment (projects) from

MS Team “Information Processing 2”, “General” channel

projects

complete any combination of projects, up to 50 points

[10] prime number generator

[10] Monte Carlo simulation to calculate π experimentally

[5] improve the RPN calculator program with `scanf`

[10] add variables to the RPN calculator program

[5] data type and arithmetic operations for 3×3 matrices

[10] data type and arithmetic operations for 3-element vectors

[10] printing the contents of a file hierarchy

[5] searching for specific names in a file hierarchy

13:00	5	
13:05	5	
13:10	5	
13:15	5	
13:20	5	
13:25	5	
13:30	5	
13:35	5	
13:40	5	
13:45	5	
13:50	5	
13:55	5	
14:00	5	
14:05	5	
14:10	5	
14:15	5	
14:20	5	
14:25	5	
14:30	10	*break*
14:40	90	exercises
16:10	00	end