BASIC COMMANDS

- Repeats the given command N time – repeat N <command>
- Listing directory content – ls
  - –l – long format.
  - –a – all, shows also hidden files.
  - –F – include special char to indicate file types.
- The *alias* command: *alias ll ls –laF*
- Copy – cp
  - –i – interactive, prompts the user before overwriting an existing file.
  - –r – recursive, can be used only on directories.
- Removing a file – rm
  - –i – interactive, same as *cp*.
  - –r – recursive, same as *cp*.
  - –f – forced, never prompts the user.
- Other useful commands:
  - cd – Changes the current directory.
  - pwd – Shows the absolute path of the current directory.
  - man – Opens the manual of the given command.
  - mkdir – Creates a new directory.
  - mv – Moves (renames) the name of a file.
  - sleep N – Waits (sleeps) for N seconds.
- Changing the permission of a file/directory – chmod
  - Format: chmod [ugoa][+-][rwx] <file>
    ugoa stands for user/group/others/all
  - Another format: chmod <octal mode> <file>
    The octal mode stands for 3 octal digits: *u[rwx]g[rwx]o[rwx]*
- Printing a message – echo
  - –n – Don't print a newline after the message.
- Printing the content of a whole file
  - cat – prints the content to the screen.
  - more – display the content one page at a time.
  - less – navigate through the file's content.
- History
  - history – shows the latest N commands.
  - !N – executes the N command. If N is negative, N commands back.
  - !! – executes the last command.
  - !<str> - executes the last command that started with the <str> prefix.
  - !$ - the last argument of the last command.
  - !* - all the arguments of the last command.
- Filenames matching
  - - - matches the standard input (instead of "regular" file).
  - * - matches everything.
  - ? – matches one single character.
  - [abc..] – like ?, but matches only one of the characters inside the brackets.
  - [a-z] – like ?, but matches only one of the characters in the given range.
  - [^abc..] and [^a-z] – like the above, but negatively (will not match…).
  - {str1,str2} – Matches either str1 or str2.
  - \ and "" – When we don't want the shell to interpret special characters:

- example: *echo overwrite\?* Or *echo "overwrite?"*

TEXT PROCESSING

- Show only the first lines of a file (or several files) – head
  - –N – Shows only the first N lines of the file.
  - –cN – Shows only the first N bytes of the file.
- Show only the last lines of the file (or several files) – tail
  - –N – Shows only the last N lines.
  - +N – Shows all the lines from the Nth line.
  - –f – Waits for more data added to the file and keeps showing new data.
- Word count – wc
  - –c – Prints the number of bytes (characters) in the file.
  - –w – Prints the number of words in the file.
  - –l – Prints the number of lines in the file.
  - When no flags are given, prints all three values.
- Output management
  - > - Writes the data into a file (overwrites it if existed before).
  - >> - Same as >, but appends the data to an existing file (no overwrites).
  - >& - Also redirects the standard error output to the output file.
  - < - Reads the content of the file as the standard input for a command.
  - >! And >>! – Forces the output redirection to succeed (if the file doesn't exist).
  - | - Pipes. Redirects the output of a command to another command:
    - Example: *ls –l | wc –c*
- Text editing – vi
  - Cursor travel: h (left), j (down), k (up), l (right)
  - Insert mode – Inserting new text.
    - i/I – Starts insert mode in place / from the beginning of the line.
    - a/A – Appends new text after the cursor / at the end of the line.
    - o/O – Opens a line below/above the current one.
    - R – starts overwriting until 'escape' is typed.
  - Command mode – Everything else. Type 'escape' to enter this mode.
    - x/X – Deletes a character under/before the cursor.
    - dd/dw/D – Delete entire line / word / until end of line.
    - r – Replaces current character.
    - yy/yw – Copies current line/word.
    - P/p – Pastes before/after cursor.
    - w/b – Goes to the next/previous word.
    - Ctrl-F/B – Forward / Backward one screen.
    - Ctrl-U/D – Forward / Backward half of a screen.
    - Ctrl-E/Y – Shifts text one line down/up. Cursor stays.
    - % - Moves to the start/end of brackets/parenthesis.
    - $/0/^ – Moves to the end/start/first non blank char of current line.
    - :N – Goes to the Nth line.
    - g/gg – Goes to the end/start of the file.
    - u – Undoes last change.
    - ~ - Switches case and advance the cursor.
    - >> / << - Indents/Unindents the current line in one tab.

- Last line mode – when you're about to quit. Type ':' to enter this mode.
  - w – Saves the changes.
  - q – Quits the vi editor.
  - wq – Saves and quit.
  - q! – Forces quit (without saving the changes).
- vi's options are stored in the file *~/.exrc*

CHARACTERS MANIPULATIONS

- Cutting lines – cut
  - –c – Outputs only the following characters.
  - –f – Outputs only the following fields (one based: 1,2,3…).
  - –d – Uses the following characters as fields delimiters.
  - –c or –f must be specified. Values are list (ranges, comma-delimitered).
- Merging files – paste
  - –d – Provides a list of separator characters.
  - –s – Pastes one file at a time. Each file become a single line.
- Translate characters – tr [options] set1 [set2]
  - –d – Deletes characters in set1
  - –s – Replaces every sequence of similar characters with a single character.
  - Special sequences:
    - [:alpha:] - all letters.
    - [:digit:] – all digits.
    - [:alnum:] – all letters and digits.
    - [:space:] – all spaces.
    - [:punct:] – all punctuation marks.
    - [C*N] – N copies of the character C.

LINE MANIPULATIONS

- Sorting lines – sort
  - –f – Ignores case.
  - –r – Reverse order.
  - –n – Numeric sort (as opposed to text based).
- Removing duplicate lines – uniq
  - –i – Ingores case.
  - –c – Prefixes lines with the number of occurrences.
  - –d – Prints only duplicated line.
  - –u – Only prints uniq lines.
  - The uniq commands works only on adjacent lines.
- Comparing test files – diff
  - –i – Ignores case.
  - –b – Ignores changes in number of white (blank) space.
  - –B – Ignores changes in empty lines.

REGULAR EXPRESSION

- The grep command: grep [options] <regex> <files>

  - –v – Prints all lines but those who don't match the pattern.
  - –c – Prints only the number of matched lines.
  - –n – Prints the line and their numbers.
  - –h- Don't print (hide) files' names.
  - –l – Prints only the files' name (list matching files).
- Regular expression rules:
  - . – Matches a single character.
  - [..] – Matches a single character from the list inside the brackets.
  - [^..] – Matches any single character that is not the brackets.
  - [a-z] – Matches a single character from the given range.
  - [[:SYMBOL:]] – same as [:SYMBOL:] from the *tr* command above.
  - * - Matches zero or more occurrences of the preceding character.
  - \C or [C] – Matches exactly the special character C.
  - ^<str> - Matches only if the line starts with the given <str> value.
  - <str>$ - Matches only if the lines ends with the given <str> value.
- Extended regular expression grep: egrep
  - + - Like '*' but with at least one occurrence.
  - ? – Like ',' but with zero or one occurrence.
  - ( ) – Grouping.
  - A|B – Matches regex A or B.
- Fixed strings lookup grep (faster): fgrep
- Stream editor – sed
  - Most commonly used for transformation:
    sed 's/pattern/replacement/[g]' <file>
    - pattern – A basic regular expression.
    - replacement – A string to replace the pattern with.
    - g – Processes several occurrences in the same line.
    - & - Can be used in the replacement to put the matched expression.
    - \(and \) – Denote a group in the pattern.
    - \1, \2, … - Refer to a group from the pattern.

    Example:
    ```
    sed 's/\([[:alpha:]]*\) \([[:digit:]]*\)/\2 \1/' file.txt
    ```

SHELL VARIABLES

- Creates a new variable – set
  - Syntax: *set [variable [= value]] …*
  - When no value is given, works like a flag ("the variable exits").
  - $NAME – Refers to a variable with the name NAME.
- unset NAME – Unsets the variable NAME.
- (A B …) – An array value with the values A, B, …
- $NAME[N] – The Nth value (1 based) of the array variable NAME.
- set NAME=($NAME VAL) – Appends the value VAL to the variable NAME.
- @ NAME = VALUE – Assigns a value to the variable NAME. VALUE can be:
  - Numerical value (only integers!).
  - Arithmetic operation: +, -, *, /, % (modulo).
  - Relational and logical operation: >, <, >=, <=, ==, !=, &&, ||, !
    Expression value would be 0 or 1. Should be surrounded with ( ).
  - The '=' can be replaces with: +=, -=, *=, /=, %=.

  - ++ and -- are also supported.
- Combination of numerical values array and @ can be used.
- $#ARRAY – The numbers of elements in the ARRAY variable.
- $%VAR – The number of character in the regular VAR variable.
- $?VAR – Whether the VAR variable exists or not (returns 0 or 1).
- Modifiers – can be added to the variable name.
  - :r – Variable's root (until the last '.').
  - :e – Variable's extension (after the last '.').
  - :h – Variable's head (file's path).
  - :t – Variable's tail (file's name).
- "..." – Translates variables inside the string (replaces $NAME with its value).
- '…' – Supress variables translations (use %NAME as is).
- `...` - Executes the command and replaces the value with its output.
- Pre-defined shell variables:
  - $noclobber – If set, doesn't allow changing an existing variable.
  - $ignoreeof – If set, forces user to use 'exit' or 'logout' to quit the shell.
  - $user – the name of the user.
  - $home – The current user's home directory.
  - $path – The commands' search path.
  - $shell – The current shell command (/bin/tcsh, /bin/bash, …).

SHELL SCRIPTS

- Runs a shell script in the current shell – *source <file>*
- Runs a shell script in a new shell (different environment) - */bin/tcsh <file>*
- Executable shell scripts:
  - Starts with - *#!/bin/tcsh*
  - Have execution permissions – *chmod +x <file>*
- Special shell scripts, located under the user's home directory:
  - .login – Executed upon login.
  - .logout – Executed just before logout.
  - .tcshrc and .cshrc – Runs whenever a new shell is created.
- Scripts' special variables:
  - $argv – The command line arguments.
    - $argv[0] - The name of the current script.
    - $argv[1], $argv[2], … or $1, $2, …- Other arguments (1 based).
    - $argv[*] or $* – All the command line arguments.
    - $#argv or $# – The number of arguments.
  - $< - Getting an input from the user.
  - $$ - The number of the current process (pid).
- Debugging a shell script – the '-x' option will print every line in the script.
  - Could also be inserted in the first line: *#!/bin/tcsh -x*
- Flow control in shell scripts:
  - `if (expression) command`
  - ```
    if (expression) then
       commands
    else if (expression) then
       commands
    else
    ```
    ```

```
    commands
endif
```

- expression can also be - *-n <file>* - where –n flag can be:
  - -e – True if the file exists.
  - -d – True if the file is a directory.
  - -o – True if the user owns the file.
  - -r – True if the user have read permission on the file.
  - -w – True if the user have write permission on the file.
  - -x – True if the user have execution permission on the file.
  - -z – True if the file is empty (have zero length).
  - -AB… - Several of the flags above can be used together.
- <str> ~=/!~ <pattern> - True if <str> matches/not the regex <pattern>. Example - *('First.Last' ~= [A-Z]*.[A-Z]*)* will result with True (1).
-
```
foreach index (list)
    commands
end
```
-
```
while (expression)
    commands
end
```
  -
```
Example reading from input:
set more_input
while ($?more_input)
    set input = $<
    if ($input == "") then
        unset more_input
    else
        …
    endif
end
```
- break – Exists a command block.
- continue – Continues to the next loop iteration.
-
```
switch (str-value)
    case pattern:
        …
        breaksw
    …
    default:
        …
        breaksw
endsw
```
- Comments are marked with #.
- Stopping the shell script – exit N
  - Success is denoted by N=0.
  - Failure is denoted by any other numeric value.
  - When N is not specified, it is equivalent to N=0 (success).
  - $status or $? – The return value of scripts (after it stopped).

- WATCH OUTs:
  -
```
set hello = "hello world"
if ($hello != goodbye) then echo no
```
    Will fail ("if: Expression Syntax") because $hello is expanded to "hello

world". Use quotes to fix:
```
if ("$hello" != goodbye) then echo no
```
- set prefix = tmp
  echo $prefix_src
  Will fail ("prefix_src: Undefined variable"). Use ${..} to fix:
  ```
  echo ${prefix}_tmp
  ```

- Other useful shell scripting commands:
  - who – Shows who else is logged on. Output:
    ```
    dor      console  Jan 25 22:43
    ```
  - w – Shows who else is logged on and what they are doing. Output:
    ```
    23:49  up 21 days,  1:07, 2 users, load averages:
    1.05 0.93 0.67
    USER     TTY      FROM        LOGIN@  IDLE  WHAT
    dor      console  -          25Jan12 21days -
    ```
  - finger – Provides information about all the logged on users. Output:
    ```
    Login Name       TTY Idle Login  Time  Office  Phone
    dor   Dor Gross *con  21d Jan 25 22:43
    ```

THE C LANGUAGE

- C source filenames should have the extension .c, e.g: *my_source_file.c*
- C header filenames should have the extension .h, e.g: *my_header_file.h*
- Compiling a single file using the gcc command.
  - The command's format : *gcc [options] -o <target> <source-file1> …*
  - Executable filename is denoted by the <target> above.
  - If *–o <target>* isn't given, the executable name would be *a.out*.
  - The *–Wall* option tell the compiler to print all warnings.
  - Only one of the source files given should have a *main* function.
- Variable:
  - Variables are declared at the beginning of every function/block.
  - Primitive types: int, float, double and char (NO built-in boolean types).
- Common header files:
  - stdio.h – Standard input and output.
  - math.h – Math functions.
  - string.h – String manipulation functions.
  - stdlib.h – Miscellaneous functions.
- Input and output (include the stdio.h header):
  - Printing something to the standard output:
    printf("..<format>..", values);
  - Conversion specifiers:
    - %c – Character.
    - %d – Decimal integer.
    - %e – Floating point – scientific notation.
    - %f – Floating point.
    - %s – String.
  - Printing in a fixed size: %Nx
  - Left align printing: %-x or %-Nx

  - Zeroes padding: %0Nx
  - Reading a value from standard input:
    scanf("%d", &my_int);
  - Scanf skips on whitespace when reading numbers, but not when reading characters.
- Flow of control:
  - False expression is a '0' value. Other values are treated as True.
  - ```
    if (expression) {
      …
    } else {
      …
    }
    ```
  - ```
    (expression) ? IF-TRUE : OTHERWISE;
    ```
  - ```
    switch (expression) {
      case value1:
        …
        break;
      case value2:
        …
        break;
      …
      default:
        …
        break;
    }
    ```
  - ```
    while (condition) {
      …
    }
    ```
  - ```
    do {
      …
    while (expression);
    ```
  - ```
    for (initialization; condition; increment) {
      …
    }
    ```
- Functions' headers (prototypes) declare their name, arguments and return value.
  void my_func (int value1, float value2);
- A function doesn't need a prototype (declaration) if it used after it was defined.
- Arrays:
  - Declared: *TYPE NAME[SIZE];*
  - Addressed: *NAME[INDEX]*
- Pointers:
  - Represent the address of a variable.
  - Define with * as follows: *TYPE *NAME;*
  - The actual value (dereference) is notated by: **NAME*
  - Arrays are actually a pointer to a list of value from the same type.
  - Strings are actually a pointer to a list (array) of characters.
- Files (include stdio.h):
  - The FILE pointer represents an instance of a file.
  - To open a file use: *fopen(NAME, MODE)*
    Mode can get one of: "r" for reading mode, and "w" for writing mode.

  - If a file couldn't be opened, a NULL value is returned by fopen.
  - Writing to a file: *fprintf(FILE*, FORMAT, ARGS)*
  - Single character functions (return EOF on failure):
    ```
    int getc(FILE*);
    int putc(FILE*, c);
    ```
- Program's argument:
  - To get program's argument, declare the main function as follows: *main(int argc, char *argv[])*
  - *argc* stands for the number of arguments.
  - *argv* stands for the arguments themselves.
- Converting a string (char*) to an integer value: *atoi(STRING)*
- Variable qualifiers:
  - signed / unsigned.
  - short / long.
  - long – Represent "long int" when no type follows.
  - short – Represent "short int" when no type follows.
  - unsigned – Represents "unsigned int" when no type follows.
  - char is a single byte type ([0,255] or [-128,127]).
  - Getting the size of a data type: *sizeof(TYPE)* or *sizeof(VAR)*
  - Types size (in bytes):
    - char: 1
    - short: 2
    - int: 4 (machine dependent)
    - long: 4
    - unsigned: 4 (machine dependent)
    - float: 4
    - double: 8
    - long double: 12
  - Literal values:
    - Integer: Decimal (1), Hexadecimal, (0x1) Octal (01).
    - Long integers: same as integers but with the suffix L (1L).
    - Floating point: Float (1.0F), Double (1.0 or 1.0D).
    - Character ('c').
    - String ("str").
  - Strings:
    - An array of characters with an additional character '\0'.
    - Therefore the size of a string is one byte larger than the string it's actually representing.
  - Constant value:
    - Declared with the *const* keyword: *const int MAX_SCORE = 100;*
    - *#define* is similar to a const value, saves memory but with no type checking.
- Defining a new type:
  - Associate a type with a new name. *typedef TYPE NAME* Examples:
    ```
    typedef int Inches;
    typedef char String[];
    ```
  - Enumeration are a list of possible values.
    - First value is 0, and every value onwards is increased by 1.
    - Explicit values can be defined.

        - *enum NAME { VALUE0, VALUE1, VALUE2 = N, VALUE3 }*
- Reading and writing from/to standard input/output (include stdio.h):
    - getchar() – Reads a single character from the standard input.
    - When no more input is available, the EOF value will be returned.
    - putchar(c) – Writes a single character to the standard output.
    - For files, see getc and putc above.
    - *char *fgets(char *line, int n, FILE*fp);*
        - Reads at most n-1 character from the file represented by fp.
        - Writes the line into the pre-allocated line buffer.
        - Returns NULL when no more data is available.
        - NOTE: Also reads the '\n' if exists.
    - We can also use the predefined FILE* values: *stdout*, *stderr*, *stdin*.
    - To write/read from/to another string, we can use (include string.h):
        - *int sprintf(char *s, const char *format, …);*
        - *int sscanf(const char *s, const char *format, …);*
- Mathematical operations (include math.h):
    - Sometimes require adding the –lm flag to the compiler.
    - Accept *double* values and return *double* values.
    - Examples: sqrt, pow, exp, log, sin, cos, tan.
- The comma ',' operator has the lowers precedence of all other operators.
    - i.e: *expr1, expr2* – expr1 will be evaluated before expr2 will.
- Preprocessing directives:
    - The *#include* directive is used to insert a file into another file.
        - *#include <FILE>* looks for the FILE in the system's include files only (usually under /usr/include).
        - *#include "FILE"* looks for the FILE in the current directory, and if it's not found, looks in the system's include file.
    - The *#define* directive creates a new preprocessed value.
        - Constant are usually declared using the *#define* directive:
          #define MY_CONSTANT_VALUE 8
        - Preprocessing flags are usually defined without any values:
          #define MY_FLAG
    - The *#undex* directive tells the preprocessor to "forget" a definition.
      #undef MY_FLAG
    - Macros are "*#define*" values with parameters.
        - *#define macro(A, B) ((A) > (B))? (A) : (B)*
        - It's a good practice to surround each value with ( ).
        - Macros longer then one line can be cut using '\'.
    - Conditional compilation (preprocessor's conditions):
      #if, #ifdef, #ifndef, #else, #elif, #endif
    - Adding a value to the preprocessor on compilation time:
      gcc –D NAME[=VALUE] …
    - Generating compilation error deliberately: *#error [message]*
    - Complex if statements: *#if defined(VAL1) || defined(VAL2)*
    - Common headers directives:
      #ifndef MY_HEADER_NAME
      #define MY_HEADER_NAME
      …
      #endif

- Functions
  - When missing a return type, *int* is assumed (and issue a complilation warning).
  - *void* can be used for both return type and for arguments (to declare no argument should be given, or no return value).
  - Missing *void* arguments tells the compiler that the argument list is unknown and therefore no type check should be preformed.
  - Missing return value will issue a compilation warning and will result with an unknown return value (garbage).
  - An undefined function is assume to be of the type that receives any number of arguments (i.e, a () argument list), and return int.
- Compiling several files
  - Compile files into object files: *gcc –c source1.c source2.c …*
    The result: *source1.o source2.o …*
  - Creating an executable from the object files: *gcc –o exec source1.o …*
- Storage classes:
  - **auto** – variables declared in a function/block etc.
    The memory is allocated upon entering the function/block.
    The memory is freed when the leaving the function/block.
  - **extern** – global variable with memory allocated when encountered and can be used from declaration onwards.
    A method of sharing variable or values across files/functions/blocks.
    - The definition of an external variable allocates its memory.
    - The declaration of it only uses it for reference.
    - All functions have an extern storage class (can't be nested).
  - **register** – Tells the compiler to store the value of the variable on a register for quicker access.
    This is only a recommendation. The compiler may decline when not enough physical space is available.
  - **static** – Two use cases:
    - Maintain a value of variable the next time a block is reentered.
    - Provides a private mechanism that limits the usage of the variable to the specific file it was declared in. Works for regular variable as well as for functions (static functions can be accessed from the same file).
- More about arrays and pointers:
  - It's a good practice to store the size of an array as a symbolic const:
    #define ARRAY_LEN 10
    int array[ARRAY_LEN];
  - Arrays can be initialized with default values:
    int array[] = {0, 1, 2, 4, 7, 9};
  - When list of initializers is shorter than expected, leftovers are set to 0:
    int array[3] = {3, 4}; / array[2] = 0 */*
  - static and external arrays will be initialized with zero values.
    automatic array will not, and will hold "garbage" as default value.
  - Characters arrays (A.K.A strings) can be initialized is various ways:
    char str1[4] = { 's', 't', 'r', '\0' };
    char str2[] = { 's', 't', 'r', '\0' };
    char str3[] = "str";

  - NOTE: It is illegal to set a pointer to an address of a different type value:
    *int *pi;*
    double d;
    pi = &d; / results with an error */*
  - Using pointers, we can modify the values of given functions' arguments.
  - We can also "return" more values by receiving them as pointer arguments.
  - Unlike pointers, which can be assigned to different addresses, arrays have a fixed address and cannot be modified (cannot point to a different array).
  - In both we can perform arithmetic operations:
    int arr[3];
    *int *p;*
    p = arr;
    *p[2] = *(arr + 2); /* == *(p + 2) == arr[2] */*
  - Dynamic allocation (include stdlib.h):
    - malloc(size_t): Allocate the requested amount of bytes, and returns NULL when the allocation failed (not enough memory), or returns a (void*) pointing to the allocation memory.
    - free(void*): Frees an allocated memory which starts at the address pointed by the given pointer.
  - The sizeof function only works for static arrays.
    For pointers, it will return the size of an address pointer (4/8 bytes).
    - This is because sizeof is a compile-time operator.
- Some characters functions (include ctype.h):
  isspace, isalpha, isdigit, islower, isupper, isalnum, isxdigit, ispunct, isprint, isgraph (non-space printable), *iscntrl, isascii, and also: toupper, tolower.*
- Some strings functions (include string.h):
  - *char *strcat(char *s1, const char *s2);*
  - *int strcmp(const char *s1, const char *s2);*
  - *char *strcpy(char *s1, const char *s2);*
  - *size_t strlen(const char *s);*
  - *char *strchr(const char *s, char c);*
  - *char *strrchr(const char *s, char c);*
  - *char *strstr(const char *s1, const char *substr);*
- Multidimensional arrays:
  - Can be allocated staticly: *int mat[3][5]* or *int mat[][5]* or *int (*a)[5]*
  - Or dynamically: *int *a[3]; a[0] = malloc(4 * sizeof(int));*
- Function pointers:
  - Work like any other pointer.
  - *int (*f_ptr)();* - Is a pointer to function that receives no arguments and returns an int value.
  - *double (*funcs[])(double);* - Is an array of functions that receive a double value and return a double value.
  - We can ease the definition using typedef:
    *typedef double (*Func_ptr)(double);*
    Func_ptr funcs[];
- Structs:
  - Provide a mechanism for defining type with various types.
  - Example of a struct:
    `strcut complex {`

```
  float re;
  float im;
};
```

- Creating an instance of the new type: *struct complex comp1, comp2;*
- Can be eased using typedef:
  `typedef struct complex Complex.`
- And even:
```
typedef struct complex {
  …
} Complex;
```
- And even more:
```
typedef struct {
  ...
} Complex;
```
- Structs can be initialized like array: *Complex c = {1.0, 2.3};*
- Accessing a struct member is done using the '.' operator: *c.re = 3;*
- Accessing a struct pointer member is done using the '->' operation: *Complex *pc; pc->re = 3;*

- Unions:
  - Defined using the same syntax as structs.
  - But its members share the same memory storage.
  - A common method is to keep another variable to select the correct field.
  - Can be initialized like structs (and arrays), but only according to the first type:
```
typedef union {
 int i;
 floar f;
} IorF;
IorF v1 = {1};  /* v1.i == 1 */
IorF v2 = {2.0};  /* v2.i == 2, v2.f != 2.0 */
```
- Structs can hold pointer to the same struct, but:
  - Inside the struct definition, the "typedef"-ed name cannot be used.
    Wrong:
```
typedef struct num {
  Number* n;
} Num;
```
  - The struct can only hold pointers and not static members.
    Wrong:
```
typedef struct num {
  struct num n;
} Num;
```
  - Correct:
```
typedef struct num {
  struct num *n;
} Num;
```

C Operators Presedence and Order of Evaluation

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right