The C Programming Language

Roman Podraza
Lecture Notes

Bibliography

- B.W. Kernighan, D.M. Ritchie, “The C Programming Language”, Second Edition, Prentice Hall Inc., Englewood Cliffs, NJ, 1988.
- B.S. Gottfried, “Schaum’s Outline of Theory and Problems of Programming with C”, Schaum’s Outline Series in Computers, McGraw-Hill Publishing Company, 1990.
- H. Schildt, „C++: The Complete Reference, Fourth Edition”, McGraw-Hill/Osborne, 2003.

Introduction

- The C Language Elements
  - Comments
    ```
    /* This is a comment. A set of any characters */
    ```
  - Identifiers
    ```
    Variable_Name_2
    ```
  - Keywords are reserved
    ```
    if, for, while
    ```
- Basic Data Types

```
char  (0:255, -128:127)
int
float
double
short int      short
long int       long
long double
unsigned int
signed char
unsigned long int
```


- Literals
  - Integer constants
    ```
    1234, 234l, 534L, 67u, 1092UL
    ```
  - Floating point number constants
    ```
    154.3, 23e-43, 231.0L
    154.3f, 23e-4F
    ```
  - Octal constants
    ```
    077
    ```
  - Hexadecimal constants
    ```
    0xAE77, 0X12
    ```
  - Strings
    ```
    "any text in quotas"
    ```
  - Character constants
    ```
    'A', '\x41', '\101', 65,
    '\0'
    ```
    special characters: `\a, \b, \f, \n, \r, \t, \v, \\, \?, \', \"`
- Defining a constant with a preprocessor directive

```
#define DIMENSION  10
```

- Enumerations

```
enum boolean {NO, YES};
enum days {MONDAY=1, TUESDAY, WEDNESDAY};
```

- Variable declarations

```
int a, b, c;
unsigned int = 0xA2;
signed int d;
char e, f[10];
short sh;
long int lint;


float ff;
double db = 2.0;
long double ldb;

const float pi = 3.14f;
volatile long vollon = 1L;
volatile const int icv = 0;
```


Operators

- Arithmetic:
  `+, *, /, -, %` (modulo only for positive integers)
- Relational:
  `>, >=, <, <=, ==, !=`
  give results equal to 1(TRUE) or 0 (FALSE)
- Logical:
  `&&` (and), `||` (or) - Evaluated always from left hand side until the result is determined
  `!` (not).
- Casting:

```
int  i = (int) (2.5 * 3.61);
```

- Incrementation and decrementation:

```
++, --
int i, j;
i = 7; j = i++; j = ++i;
```

- Bit operators:
  `&` (and), `|` (or), `^` (exor), `<<` (shift left), `>>` (shift right), `~` (one's complement).

- Assignment: =, *op*= (*op*: +, −, *, /, %, >>, <<, &, |, ^)

```
i = 5
i = j = 6      /* i = (j = 6)  */
i −=2          /* i = i − 2    */
i += j = 4     /* i += (j = 4) */
```

- Conditional operator:

```
expression1? expression2 : expression3
c = (a > b) ? a : b
```

- Operator , (comma)

```
i = (j=5, ++j)
```

- Priorities and evaluation order
  - Table of priorities and associativness of operators
  - Priorities of operators in expressions can be defined (changed) by parenthesis

```
i = (2 + 3) * 16
```

- Only for the following operators:
  &&, **||**, , (comma) and conditional operator
  the evaluation order is defined from the left hand side to the right hand side. For all other operators the evaluation order is not defined (and is implementation dependent).

- **Example OP.1**

```
int i = 8, j = 2;

i = ++j * j++; /* i = 8  ?
                  i = 9  ? */
```

- **Example OP.2**

```
int i = 0, j = 0, k = 0;

i++ && j++ || k++;
```

Instructions

- Any expression followed by the semicolon is an instruction (a sentence)

```
expression ;
```

```
i++;
a = 5;
```

- Compound instruction (a block)

```
{
declarations
instructions
}
```

  - a hint: **;** is not used after **}**

- Instruction `if else`
  -
```
if (expression)
  instruction_1
else                // optional part
  instruction _2
```
  -
```
if (expression_1)
  instruction_1
else if (expression_2)
  instruction_2
...
else                // optional part
  instruction_n+1
```


- **Example INSTR.1**

```
if ( a > b )
   x  = a;
 else
   x = b;
if ( x < 0 )
   x = -x;
```

- Loops
  - Instruction `while`

```
while (expression)
  instruction
```

  - Instruction `do - while`

```
do
  instruction
while (expression);
```

  - Instruction `for`

```
for (expression_1; expression_2; expression_3)
  instruction
```

```
expression_1;
while (expression_2) {
  instruction
  expression_3;
  }
```

- **Example INSTR.2**

```
while (1);     /* infinite loop */
for (;;);      /* infinite loop */
```

- **Example INSTR.3**

```
for (i = 0; i < n; i++) /* execute n times*/
 ....
```

- **Example INSTR.4**

```
int tab[10][10];
int i;
for (i = 0, j = 0; i < 10 && j < 10; i++, j++)
 tab[i][j]++;
```

- **Example INSTR.5**

```
int binsearch (int x, int v[], int n)
{
int low = 0, high = n-1, mid;
while (low <= high) {
 mid = (low + high) / 2;
 if (x < v[mid])
    high = mid - 1;
    else if (x > v[mid])
    low = mid + 1;
    else
    return mid;
 }
return (-1);
}
```


- Instruction **`break`**
  - leave the loop (the given level)
  - terminate execution of instruction switch
- Instruction **`continue`**
  - skip to (or after) the end of the loop body
- Instruction **`goto`**
  - skip to the label

```
goto label;
.....
label: i = 5; /* label - identifier followed by ':' */
....          /* skip only within a function */
```

- Instruction **`switch`**

```
switch (expression) {
  case const_expression_1: instructions_1
  case const_expression_2: instructions_2
  ...
  default: instructions
}
```

  - Values of constant expressions have to be unique

- **Example INSTR.6**

```
#include <stdio.h>

int main(void)
{
int c, i, nwhite, nother, ndigit[10];
nwhite = nother = 0;

for (i = 0; i < 10; ++i)
  ndigit[i] = 0;

while ( (c = getchar() ) != EOF )
  {
  switch ( c ) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0']++;
        break;
    case ' ': case '\t': case '\n':
        nwhite++;
        break;
    default:
        nother++;
        break;
    }
  }
```



```
printf ("digits = ");

for (i = 0; i < 10; i++)
  printf ("  %d", ndigit[i]);

printf (", white spaces = %d, other = %d\n",
        nwhite, nother);

return 0;
}
```

- Instruction **return**
  - return a control from a function

    return ***expression***;

Functions and Program Structure

- Typical structure of source program (code)

```
#include
#define

function prototypes
external declarations

functions
```

- Syntax of functions

```
return_type function_name (parameters)
{
declarations
instructions
}
```

- **Example FUN.1**

```
void dummy (void)
{
}
```


- **Example FUN.2**

```
 /* Calculator of floating point numbers working according to
  Reverse Polish Notation */
#include <stdio.h>
#include <math.h>

#define MAXOP   100
#define NUMBER  '0'

int getop (char[]);
void push (double);
double pop(void);

int main (void)
{
int type;
double op2;
char s [MAXOP];
while ( (type = getop(s)) != EOF )
  {
  switch (type) {
    case NUMBER:  push (atof(s));
                  break;
    case '+':     push (pop() + pop());
                  break;
```

```
    case '*':     push (pop() * pop());
                  break;
     case '-':    op2 = pop();
                  push (pop() - op2);
                  break;
     case '/':    op2 = pop();
                  if (op2 != 0.0)
                    push (pop() / op2);
                   else
                    printf ("error: /0.0\n");
                  break;
     case '\n':   printf ("\t%.8g\n", pop());
                  break;
     default:     printf ("error: unknown" command: %s\n", s);
                  break;
     } /* switch */
   }   /* while */
return 0;
}
/* ------------------------------------ */

#define MAXVAL   100
int sp = 0;
double val[MAXVAL];
```



```
void push (double f)
{
if (sp < MAXVAL)
  val[sp++] = f;
   else
  printf ("error: stack full\n");
}
double pop (void)
{
if (sp > 0)
  return val[--sp];
  else
  printf ("error: stack empty\n");
}
/* ------------------------------- */
#include <ctype.h>
int getch (void);
void ungetch (int);
int getop (char s[])
{
int i, c;
while ( (s[0] = c = getch()) == ' ' || c == '\t')  ;
s[1] = '\0';
if ( ! isdigit (c) && c != '.')
  return c;
```

```
i = 0;
if ( isdigit (c) )
  while ( isdigit (s[++i] = c = getch()) ) ;
if ( c == '.' )
  while ( isdigit (s[++i] = c = getch()) ) ;
s[i] = '\0';
if ( c != EOF )
  ungetch(c);
return NUMBER;
}
/* ------------------------------ */
#define BUFSIZE  100
char buf[BUFSIZE];
char bufp = 0;
int getch(void)
{
return (bufp>0) ? buf[--bufp] : getchar();
}
void ungetch (int c)
{
if (bufp >= BUFSIZE)
  printf ("ungetch: too many characters \n");
 else
  buf[bufp++] = c;
}
```


- The scope of external declarations of variables and functions
  - from the point of the first declaration to the end of source file
- Definitions and declarations of identifiers
  - the definition can appear exactly once (the memory for the object is allocated and the identifier can have its value set)
  - declarations can appear many times provided all declarations are the same
- **Example FUN.3**

```
int sp;
double val[MAXVAL];
extern int first;
int second;
int third = 8;
void dummy (void)
{}
extern int first;
int sum (int x, int y);
int sum (int, int);
void call (void)
{
int z, sum (int, int);
z = sum(3,4);
}
```

```
extern int sp;
extern double val[MAXVAL];
int first = 0;
int second;
int third = 7; /* error */
void dummy (void);
extern int second;
extern int first; /* ? */
int sum (int x, int y)
{
return x+y;
}
```

- Header files
  - *#include*
  - *#define*
  - *function prototypes*
  - *declarations of external variables*
- Static and automatic variables
- **Example FUN.4**

```
int fun (int x)
{
int i;
static int k;
....
}
int fun1 (int y)
{
int i = 1;
static int k = 0;
.....
}
int glob_i = 0;
void fun_v (void)
{
auto int c;
static char ch = 'A';
...
}
```

```
static int glob = 0;

static int fun_v (void);

static int fun_v (void)
{
....
return 1;
}

static int fun_i (void)
{
register int x;
register auto int y;
....
}
```


- Initialization of variables
  - Initialization of static variables - constant expression
  - Initialization of automatic variables - any valid expression
  - **Example FUN.5**

```
int binsearch (int x, int v[], int n)
{
int low = 0;
int high = n - 1;
int mid = (low + high)/2;
....
}
```

  - Initialization of arrays
  - **Example FUN.6**

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
              30 ,31};
char array[] = "array";
char table[] = {'t', 'a', 'b' 'l', 'e', '\0'};
int Year [12] = {31, 28, 31};
int month [2] = {31, 28, 31}; /* error */
```

- Recursion
  - direct
  - indirect

Example FUN.7

```
#include <stdio.h>

void printd (int n)
{
if (n < 0) {
  putchar ('-');
  n = -n;
  }

if (n / 10)
  printd (n / 10);
putchar (n % 10 + '0');
}
```

Example FUN.8

```
void qsort (int v[], int left, int right)
{
int i, last;
void swap (int v[], int i, int j);

if (left >=right)
  return;

swap (v, left, (left+right) / 2);
```



```
last = left;

for (i = left+1; i <= right; i++)
  if (v[i] < v[left])
    swap (v, ++last, i);

swap (v, left, last);

qsort (v, left, last - 1);
qsort (v, last + 1, right);
}

void swap (int v[], int i, int j)
{
int temp;

temp = v[i];
v[i] = v[j];
v[j] = temp;
}
```

Pointers and Addresses

- Pointers always define addresses and types of data being indicated by them (an address without a data type stored has a very limited scope of applications).
- **Example PTR.1**

```
void f (void)
{
int x = 1, y = 2, z[10];
int *ip; /* declaration of a pointer to int */
ip = &x;
y = *ip; /* y = x */
*ip = 0;
*ip = *ip + 10;
*ip -=2;
++*ip;
(*ip)--;
ip = &z[1];
....
}
```

- **Example PTR.2**

```
void g (void)
{
int x = 5, y = 2, *ip = &x;
int *iq;
```



```
double *db_ptr, atof(char *);
double db;
char *fun_ch (char *);

iq = ip;
db_ptr = &db;
*db_ptr = atof ("123.54");
....
}
```

- Pointers and function parameters
- **Example PTR.3**

```
void swap (int x, int y)
{
int temp;
temp = x;
x = y;
y = temp;
}

void test (void)
{
int a = 1, b = 2;
swap (a, b);
printf ("a=%i b=%i\n", a, b);
}
```

```
void swap (int *x, int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}

void test (void)
{
int a = 1, b = 2;
swap (&a, &b);
printf ("a=%i b=%i\n", a, b);
}
```

- **Example PTR.4**

```
/* Reading integer numbers */
#include <ctype.h>
#include <stdio.h>
int getch (void);
void ungetch (int);

int getint (int *pn)
{
int c, sign;
while ( isspace(c = getch()) ) /* nothing */ ;
if ( !isdigit(c) && c != EOF && c != '+' && c != '-' ) {
  ungetch (c); /* not a number */
  return (0);
  }
sign = (c == '-') ? -1 : 1;
if (c == '+' || c == '-')
  c = getch ();
for ( *pn = 0; isdigit (c); c = getch() )
  *pn = 10 * *pn + (c - '0');
*pn *= sign;
if (c != EOF)
  ungetch (c);
return c;
}
```



```
#define SIZE 1000

void fun_x (void)
{
int n, array[SIZE], getint (int *);

for (n = 0; n < SIZE && getint (&array[n]) != EOF; n++)
  ; /* read numbers to array */
....
}
```

- Pointers and Arrays
  - Arrays are stored in a continuos memory area starting from the element with index 0
  - Array name is the address (a constant) of the first element (with index 0)
- **Example PTR.5**

```
void ff (void)
{
int x, a[10] = {1, 2, 3};
int *pa;
pa = &a[0];                pa = a;
x = *pa;                   x = *a;
pa = &a[1];                pa = a+1;
x = *pa;                   x = *(a+1);
```

```
x = a[0];                  x = *(a+0);

pa = a;                    x = 8;
a = pa; /* error !!! */    8 = x;  /* error !!! */

*(a+4) = 1;
pa = a+1;
*pa = 31;
*(pa+2) = 44;
pa = a + 5;
x = *(pa – 2);
x = *(a – 5); /* error !!! */
```

- A curiosity

```
a[1] º *(a+1) º *(1+a) º 1[a]
1[a] = 5;
```

- If a pointer `pa` indicates an array element (the array elements have the data type, which is designated to the pointer) then
  - `pa+1` points the next array element
  - `pa–1` points the previous array element
  - `pa+n` points the n-th array element after the current one
  - `pa–n` points the n-th array element before the current one

  provided that arithmetic operations on addresses do not result in an address beyond the array.

- **Example PTR.6**

```
/* Implementation of function calculating length of string */
int strlen (char *s)
{
int n;
for (n = 0; *s != '\0'; s++)
  n++;
return n;
}
/* Calling of function strlen*/
void tt (void)
{
char array[] = "text", *ptr = array;
int i;
i = strlen ("hello, world");
i = strlen (array);
i = strlen (ptr);
....
}
/* Other implementations */
 int strlen (char *s)
 {
 int n;
 for (n = 0; *s != '\0'; s++)
    n++;
 return n;
 }
```

```
int strlen (char s[])
{
int n;
for (n = 0; s[n] != '\0'; n++)
   ;
return n;
}
```

```
int strlen (char *s)
{
int n;
for (n = 0; s[n] != '\0'; n++)
   ;
return n;
}
```

```
int strlen (char s[])
{
int n;
for (n = 0; *s != '\0'; s++)
   n++;
return n;
}
```

- **Example PTR.7**

```
/* Allocation and deallocation of dynamic memory */
#define ALLOCSIZE    10000
static char allocbuf [ALLOCSIZE];
static char *allocp = allocbuf;
char *alloc (int n)
{
if (allocbuf + ALLOCSIZE - allocp >= n) {
  allocp += n;
  return allocp - n;
  }
 else
  return 0; /* return NULL */
}
void afree (char *p)
{
if ( p >= allocbuf && p < allocbuf + ALLOCSIZE )
  allocp = p;
}
```


- Operation on addresses
  - adding and subtracting an integer number
  - comparing addresses (of the same data types)
  - subtracting values of two addresses
- In assignment operation pointers have to be of the same type (except universal pointer type `void *`)
- Pointers to characters and arrays of characters
- **Example PTR.8**

```
/* declaration and assignment to variable of type char*  */
{
char * pmessage;
pmessage = "text";
...
}
```

- **Example PTR.9**

```
/* Comparing declarations and initializations of variables of
  type char * and char [] */
{
char message [] = "text";
char *pmessage = "text";
....
}
```

- In **stdlib** library there are functions of dynamic allocation and deallocation of memory.

```
void *malloc (size_t  size);  /* void *malloc (int size) */
void *calloc (size_t  count, size_t  size);
void free (void *ptr);
```

- **Example PTR.10**

```
#include <stdlib.h>
#include <string.h>
{
char *ctr;
ctr = (char *) malloc (100 * sizeof (char));
...
strcpy (ctr, "100 chars");
....
free (ctr);
}
```

- **Example PTR.11**

```
/*
   Implementations of function copying strings
*/
/*
   In library <string.h> there is a function
   char *strcpy (char *s, const char *t)
*/
```



```
/*
   In the example implementations of function
      void strcpy (char *s, char *t);
   are shown
*/
void strcpy (char *s, char *t)
{
s = t; /* ERROR */
/* no copying */
/* address t is copied into parameter s */
}
void strcpy (char *s, char *t)
{
int i;
i = 0;
while ( (s[i] = t[i]) != '/0')
  i++;
}
void strcpy (char *s, char *t)
{
while (( *s = *t ) != '/0'){
  s++;
  t++;
  }
}
```

```
void strcpy (char *s, char *t)
{
while (( *s++ = *t++ ) != '/0') ;
}
void strcpy (char *s, char *t)
{
while ( *s++ = *t++ ) ;
}
```

- **Example PTR.12**

```
/*   Implementations of function comparing strings */
int strcmp (char *s, char *t)
{
int i;
for (i = 0; s[i] == t[i]; i++)
  if (s[i] == '\0')
    return 0;
return s[i] - t[i];
}
int strcmp (char *s, char *t)
{
for (; *s == *t; s++, t++)
  if (*s == '\0')
    return 0;
return *s - *t;
}
```


- Arrays of pointers and multidimensional arrays
  - Arrays of pointers
  - **Example PTR.13**

```
#include <stdlib.h>
void error (int);
void readline (char *);
void fun_1 (void)
{
char *lineptr [20];
char line1[100];
char *lptr;
lptr = (char *) malloc (100 * sizeof (char));
if ( ! lptr ){
  error (3); return;
  }
lineptr[0] = line1;
lineptr[1] = lptr;
readline (line1);
readline (lptr);
if (strcmp (lineptr[0], *(lineptr+1))
  printf ("Lines are the same\n");
   else
  printf ("Lines are different\n");
return;
}
```

- Multidimensional arrays
- **Example PTR.14**

```
void fg (void)
{
int a[6] = {1, 2, 3};
int b[2][3] =    {   {1, 2, 3},
                     {4, 5}
                 };
}
```

  - Order of array elements:

```
b[0][0]
b[0][1]     b[0][x]
b[0][2]
b[1][0]
b[1][1]     b[1][x]
b[1][2]
```

  - If a multidimensional array is a function parameter then it is possible to omit the first dimension of the parameter.
  - **Example PTR.15**

```
void fun_tab (int [][13]);
void kl (void)
{ int tab_2 [5][13];
fun_tab (tab_2);
}
```


  - Multidimensional arrays are often replaced by arrays of pointers
  - **Example PTR.16**

```
void xz (void)
{
char *pname[] = {  "Illegal month",
                   "January",
                   "February",
                   "March"
                 };
char *aname[][15] = {  "Illegal month",
                       "January",
                       "February",
                       "March"
                     };
```

- Command-line parameters
  - Program **echo** prints out command-line parameters

```
echo hello, world
```

  - Structure of command-line parameters

```
int main (int argc, char *argv[]);
argc -> 3
argv  @----->[ @--]----> echo\0
             [ @--]----> hello,\0
             [ @--]----> world\0
             [ 0  ]
```

➢ **Example PTR.17**

```
/* the first version */
int main (int argc, char *argv[])
{
int i;

for (i = 1; i <= argc; i++)
  printf ("%s%s", argv[i], (i<argc-1)?"  ":"");

printf ("\n");
return 0;
}

/* the second version */
int main (int argc, char **argv)
{
while (--argc > 0)
  printf ("%s%s", *++argv, (argc>1)?"  ":"");
printf ("\n");
return 0;
}
```


Structures

- Structure syntax

```
struct tag {
        element_1;
        element_2;
        ....
        element_n;
        } name_1, ...., name_n;
```

- Structure declaration
- **Example STRUCT.1**

```
struct point {
  int x;
  int y;
  } pt_1, pt_2;
struct point pt3;
struct point pt4 = {220, 100};

struct my {
  char c;
  int code;
  };
struct my a_my, b_my = {'A', 23};
```

```
struct {
  double db;
  int it;
  char *t_char[13];
  struct my s_my;
  struct my *s_ptr;
  } str_d;
struct list{
  int value;
  struct list *next;
  }
```

- Dereferencing of structure element

structure_name.element
address_of_structure->element

- **Example STRUCT.2**

```
void fstr (void)
{
struct point pt_1, *point_ptr;
int i, j;
i = pt_1.x;
j = pt_1.y
point_ptr = &pt_1;
i = (*point_ptr).x;
j = point_ptr->y;
....
}
```


- Operations on structures
  - copying / assignment
  - addressing
  - dereferencing of element
  - passing by value for function parameter
  - returning as a function result value

- **Example STRUCT.3**

```
struct point makepoint (int x, int y)
{
struct point temp;
temp.x = x;
temp.y = y;
return temp;
}

struct point addpoint (struct point p1, struct point p2)
{
p1.x += p2.x;
p1.y += p2.y;
return p1;
}
```

Example STRUCT.4

```
/* Binary tree with strings stored in a lexicographic order */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
struct tnode {
  char *word;
  int count;
  struct tnode *left;
  struct tnode *right;
  };
struct tnode *addtree (struct tnode *, char *);
void treeprint (struct tnode *);
int getword (char *, int);
int main (void)
{
struct tnode *root;
char word[MAXWORD];
root = NULL;
while ( getword (word, MAXWORD) != EOF)
  if (isalpha (word[0]) )
    root = addtree (root, word);
treeprint (root);
return 0;
}
```



```
struct tnode *talloc (void);
char *strdup (char *);

struct tnode *addtree (struct tnode *p, char *w)
{
int cond;

if (p == NULL) {
  p = talloc();
  p->word = strdup (w);
  p->count = 1;
  p->left = p->right = NULL;
  }

 else if ( (cond = strcmp (w, p->word)) == 0 )
    p->count++;

 else if (cond < 0)
    p->left = addtree (p->left, w);

 else
    p->right = addtree (p->right, w);

return p;
}
```

```
void treeprint (struct tnode *p)
{
if (p != NULL) {
  treeprint (p->left);
  printf ("%4d %s\n", p->count, p->word);
  treeprint (p->right);
  }
}
int getword (char *word, int lim)
{
int c, getch (void);
void ungetch (int);
char *w = word;
while ( isspace (c = getch()) )
  ;
if (c != EOF)
  *w++ = c;
if ( ! isalpha (c) ) {
  *w = '\0';
  return c;
  }
for ( ; --lim > 0; w++)
  if ( ! isalnum (*w = getch()) ) {
    ungetch (*w);
    break;
    }
```



```
*w = '\0';
return word[0];
}
#include <stdlib.h>
struct tnode *talloc (void)
{
return (struct tnode *) malloc ( sizeof(struct tnode) );
}
char *strdup (char *s)
{
char *p;
p = (char *) malloc ( strlen(s) + 1 );
if (p != NULL)
  strcpy (p, s);
return p;
}
```

- Bit fields
- **Example STRUCT.5**

```
struct bit_fields  {
    unsigned int value:  4;
    unsigned int valid:  1;
    unsigned int new_w:  0;
    unsigned int :       4;
    } bit_struct;
```

```
bit_struct.value = 15;
bit_struct.valid = 0;
```

- Sequencing of bits and sequencing of fields are implementation-dependent.
- It is impossible to get address of bit fields (by using operator &)

```
& bit_struct.value /* error */
```

- Union syntax

```
union tag{
        element_1;
        ....
        element_n;
        } name_1, ...., name_n;
```

- Each element of an union is stored at the same address of memory
- Union declaration
- **Example STRUCT.6**

```
union u_tag {
  int ival;
  float fval;
  char *sval;
  } un;
```

- Dereferencing of union element

```
union_name.element
address_of_union->element
```


- **Example STRUCT.7**

```
un.ival = 5;
un.fval = 2.3f;
fl_1 = un.fval;
```

- Type definition (declaration **typedef**)
  - Declaration **typedef** introduces a new name of type according to the specification. There is no new type. The new name can be used in a scope of the declaration.
- **Example STRUCT.8**

```
typedef int Length;
typedef char *String;
typedef Length *Len_ptr;
typedef String *String_ptr;

void fun_t (void)
{
Length a, b = 1;
int x;
char ch = 'A';
char *str_1 = "text";
String str_2 = "text_2";
char tab[12] = "nothing";
String_ptr sptr;
```

```
x = b;
a = 12+x-b;
str_1 = str_2;
str_2 = &ch;
sptr = &str_2;
**sptr = 'B';
}
```


Preprocessor

- Program in C consists of lexical atoms
- **Example PRE.1**

```
int main (void)
{
int x;
x = 1;
printf ("hello, world");
return x;
}

int   main   (   void   )   {   int   x   ;   x   =   1   ;

printf   (   "hello, world"   )   ;   return   x   ;   }

int  main
 (   void
     )
{int x         ;x=1
   ;printf(
"hello, world"       ) ;return
x
     ;}
```

- Comments in C programming language are omitted by compiler in analysis of a source code.
- The preprocessor is a program that processes a text according to some rules described by preprocessing directives. The result of preprocessing is a text.
- Directives of the preprocessor begin with character **#**, which has to be the first not blank character in a line of source code; a preprocessing directive can appear after some blanks (spaces or tabulations)
- Directive **#include** is used for inclusion a content of a source file into the place of the directive. The included source file is depicted by the parameter of the directive.
  - a parameter in the angle parentheses <> denote a file from a system directory
    ```
    #include <string.h>
    ```
  - a parameter in quotes denote a file from any directory
    ```
    #include "my_hdr.h"
    ```
  - a file name can be given with a directory path
    ```
    #include "/mainp/source/headers/my_hdr.h"
    ```
  - Files included into a source file can have directives **#include**. The directives **#include** cannot have direct or indirect referencing to the file in which they are included.

- Directive **#define** is used for defining a symbol, which is replaced by a sequence of characters (the definition) in the scope of the directive.
  - **#define *name* *replacement_text***
  - The scope of the directive: from the directive to the end of file.
  - Every appearance of lexical atom ***name*** is substituted by the ***replacement_text***. There must not be currently defined ***name*** in the ***replacement_text***. The previously defined symbols can be present in the ***replacement_text*** and they are substituted by their definitions.
  - **Example PRE.2**
    ```
    #define ATOM   1
    #define BTOM   ATOM+2
    #define LONG_TOM 1 + 2 *ATOM – BTOM + (int)('A') –12 \
                     * BTOM
    /* invalid */
    #define CTOM CTOM + 1
    ```
  - The substitution is applied only to lexical atoms, so substitution does not appear in the following line
    ```
    ATOM_var = "ATOM + BTOM";
    ```
    provided the line is in the scope of the definitions from **Example PRE.2**
- It is possible to define a macrodefinition with text parameters with the help of directive **#define**.

```
Example PRE.3
#define MAX(x, y)  x>y ? x : y
       /* no spaces */

x = MAX (5*a, 6-b);
x = 5*a>6-b ? 5*a : 6-b;

i = 1;
j = 2;
x = MAX (i++, j++);
x = i++>j++ ? i++ : j++; /* i+=1; j+=2 */

#define square(x)  x*x
y = square (z+1);
y = z+1*z+1;   /* ?? */

#define sqr(x) (x)*(x)
y = sqr (z+1);
```

y = (z+1)*(z+1); /* OK */

- A hint: Parameters of a macrodefinition should be usually enclosed in parentheses in the definition.

- Operator `#` in directive `#define` is used for a such processing of the parameter that it can be properly presented as a part of a string.

```
define dprint(expr)  printf (#expr "=%g\n", expr)
dprint (x/y);
printf ("x/y" "=%g\n", x/y);
printf ("x/y=%g\n", x/y); /* after concatenation of strings*/
```

  - If in argument of macrocall there is one of the special characters (represented in strings with prefix \) like \, ?, ", ' then in the string resulting from operator # the character will be represented with the expected prefix \ to output the desired character.
- Operator `#` in directive `#define` is used for concatenation of parameters

```
#define paste(front, back) front##back
paste (name, 1) = x;
name1 = x;
```

- Directive `#undef` is used for canceling the current definition

```
#undef getchar
```

- Conditional compilation
  - Directive `#ifdef`

```
#ifdef identifier
/* lines included to compilation if the identifier is
   defined*/
....
#endif
```

- **Example PRE.4**

```
#define TEST
...
#ifdef TEST
printf ("[Debugging]: x = %i\n", x);
#endif
```

- Directive **#ifndef**

```
#ifndef identifier
....
/* lines included to compilation if the identifier is
   not defined*/
....
#endif
```

- Directive **#ifdef #else**

```
#ifdef identifier
....
/* lines included to compilation if the identifier is
   defined*/
....
#else
....
/* lines included to compilation if the identifier is
   not defined*/
....
#endif
```


- Directive **#if**

```
#if const_expression
....
/* lines included to compilation if the const_expression
   is defined*/
....
#endif
```

- **Example PRE.5**

```
#if 'z' - 'a' == 25
#if ('z' - 'a' == 25)
#if defined (SYMBOL)
#if defined SYMBOL
#ifdef SYMBOL
#if defined SYMBOL && defined TEST && SYMBOL > 20
```

- Directive **#if #elif**

```
#if const_expression_1
....
#elif const_expression_2
....
.....
#elif const_expression_n
....
#else
....
#endif
```

- Other directives
  - `#error message`
  - `#line number`
  - `#line number filename`
  - `#pragma preprocessing_tokens`
- Predefined macrodefinitions
  - `__DATE__  /* date of translation */`
  - `__FILE__  /* source-file name */`
  - `__LINE__  /* current line within source file */`
  - `__STDC__  /* conforming translator and level */`
  - `__TIME__  /* time of translation */`

Advanced Topics

- Pointers to functions
  - It is possible to define pointers to functions that can be assigned to, kept in arrays and structures, passed as parameters etc.
  - A pointer to function stores address of a function code and the function characteristics (return type, number and types of parameters.

```
int (*pfi) (char *, char *); /* pointer to function */
int *fi (char *, char *); /* function returning a pointer
*/
```

- **Example ADV.1**

```
/* Sorting of read lines (each lines contains a number)
  according to lexicographic or numeric (option -n) order.
#include<stdio.h>
#include<string.h>
#define MAXLINES 5000 /* max. number of lines to be sorted */
char *lineptr[MAXLINES];

int readlines   (char *lineptr[], int nlines);
void writelines (char *lineptr[], int nlines);
void qsort (void *lineptr[], int left, int right,
            int (*comp)(void *, void*))
int numcmp (char *, char *);
```

```
int main (int argc, char *argv[] )
{
int nlines, numeric = 0;
if ( argc > 1  &&  strcmp (argv[1], "-n") == 0 )
  numeric = 1;
if ( (nlines = readlines (lineptr, MAXLINES)) > 0 ) {
   qsort ( (void **)lineptr, 0, nlines-1,
   (int (*) (void *, void *)) (numeric ? numcmp : strcmp) );
   writelines (lineptr, nlines);
   return 0;
   }
  else {
    printf ("Input too big to sort\n");
    return 1;
    }
}

void qsort (void *v[], int left, int right,
            int(*comp)(void *, void *))
{
int i, last;
void swap (void **, int, int);

if (left >= right )
  return;
```



```
swap (v, left, (left+right)/2);
last = left;
for (i = left+1; i <= right; i++)
  if ( (*comp)(v[i], v[left]) < 0 )
    swap (v, ++last, i);
swap (v, left, last);
qsort (v, left, last-1, comp);
qsort (v, last+1, right, comp);
}

#include<math.h>

int numcmp (char *s1, char *s2)
{
double v1, v2;
v1 = atof (s1);
v2 = atof (s2);
if (v1 < v2)
  return -1;
 else if (v1 > v2)
  return 1;
 else
  return 0;
}
```

```
void swap (void *v[], int i, int j)
{
void *temp;

temp = v[i];
v[i] = v[j];
v[j] = temp;
}
void writelines (char *lineptr[], int nlines)
{
int i;
for (i = 0; i < nlines, i++)
  printf ("%s\n", lineptr[i]);
}
#include <stdlib.h>
#define BUFSIZE 80
int readlines (char *lineptr, int nlines)
{
int c, i, minus, lines = 0;
char buf[BUFSIZE];
while (1)
  {
  while ( c = getchar() == ' ' || c == '\t' )
  ; /* white chars at the beginning */
  i = 0;
```



```
  minus = 0;
  if ( c=='+' || c=='-' ){
    minus = (c == '-');
    c = getchar();
    while ( c == ' ' || c == '\t' )
      c = getchar(); /* white chars after a sign */
    }
  if ( isdigit(c) || c == '.' )
    if (minus)
      buf[i++] = '-';

  while ( isdigit(c) && i < BUFSIZE ) {
    buf[i++] = c;
    c = getchar();
    }
  if ( c == '.' && i < BUFSIZE ) {
    buf[i++] = c;
    c = getchar();
    }

  while ( isdigit(c) && i < BUFSIZE ) {
    buf[i++] = c;
    c = getchar();
    }
```

```
  if ( i > 0 && lines < nlines ) {
    buf[i] = '\0';
    lineptr[lines++] = (char *) malloc (strlen (buf) + 1 );
    strcpy (lineptr [lines-1], buf);
    }
  while ( c != '\n' && c != EOF )
    c = getchar ();
  if ( c == '\n' )
    continue;
  if ( c == EOF )
    break;
  }
return lines;
}
```

- Complex declarations
  - A list of simplified declarations

```
int i;
int *p;
int a[];
int f();
int **pp;
int (*pa)[]; /* !!! */
int (*pf)(); /* !!! */
```

```
int *ap[];
int aa[][];
int af[]();  /* ??! */
int *fp();
int fa()[];  /* ??! */
int ff()():  /* ??! */
int ***ppp;
```



```
int (**ppa)[];  /* !!! */
int (**ppf)();  /* !!! */
int *(*pap)[];  /* !!! */
int (*paa)[][]; /* !!! */
int (*paf)[](); /* ??! */
int *(*pfp)();  /* !!! */
int (*pfa)()[]; /* ??! */
int (*pff)()(); /* ??! */
int **app[];
int (*apa[])[]; /* !!! */
int (*apf[])(); /* !!! */
int *aap[][];
int aaa[][][];
```

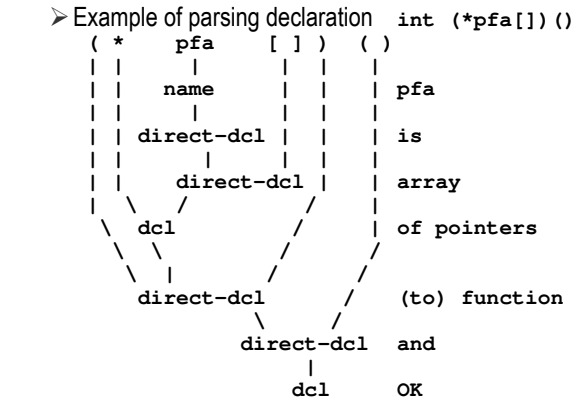
```
int aaf[][]();  /* ??! */
int *afp[]();  /* ??! */
int afa[]()[]; /* ??! */
int aff[]()(): /* ??1 */
int ***fpp();
int (*fpa())[]; /* !!! */
int (*fpf())(); /* !!! */
int *fap()[];   /* ??! */
int faa()[][];  /* ??! */
int faf()[]();  /* ??! */
int *ffp()();   /* ??! */
int *ffa()()[]; /* ??! */
int fff()()();  /* ??! */
```

- Parser of complex declarations
- Simplified grammar of complex declarations (for lexical analysis)

```
dcl:            optional  *s  direct-dcl

direct-dcl:     name
                ( dcl )
                direct-dcl ( )
                direct-dcl [ optional size ]
```

- Example of parsing declaration `int (*pfa[])()`



- **Example ADV.2**

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

#define MAXTOKEN 100
enum {NAME, PARENS, BRACKETS}; /* , (), [] */
```



```
void dcl (void);
void dir_dcl (void);

int gettoken (void);
int tokentype;
char token [MAXTOKEN];
char name [MAXTOKEN];
char datatype [MAXTOKEN];
char out[1000];

int main (void)
{
while (gettoken() != EOF) {
  strcpy (datatype, token);
  out[0] = '\0';

  dcl();

  if (tokentype != '\n')
    printf ("syntax error \n");
   else
    printf ("%s: %s %s\n", name, out, datatype);
  }
return 0;
}
```

```
void dcl (void)
{
int ns;
for (ns = 0; gettoken() == '*';  )
  ns++;

dir_dcl();

while (ns-- > 0)
  strcat (out, " pointer to");
}

void dir_dcl (void)
{
int type;
if (tokentype == '('){
  dcl();
  if (tokentype != ')')
    printf ("error: missing )\n");
  }
else if (tokentype == NAME)
  strcpy (name, token);
else
  printf ("error: expected name or (dcl) \n");
```



```
while ( (type=gettoken()) == PARENS || type == BRACKETS )
  if (type == PARENS)
    strcat (out, " function returning");
   else {
      strcat (out, " array");
      strcat (out, token);
      strcat (out, " of");
      }
}
int gettoken (void)
{
int c, getch (void);
void ungetch (int);
char *p = token;
while ( (c = getch()) == ' ' || c == '\t' ) /* */;
if ( c == '(' ) {
  if ( (c = getch()) == ')' ) {
    strcpy (token, "()");
    return (PARENS);
    }
   else {
    ungetch (c);
    return tokentype = '(';
    }
  }
```

```
else if ( c == '[') {
  for (*p++ = c; (*p++ = getch()) != ']'; )
    ;
  return tokentype = BRACKETS;
  }
else if ( isalpha(c) ) {
  for (*p++ = c; isalnum ( c = getch() ); )
    *p++ = c;
  *p = '\0';
  ungetch (c);
  return tokentype = NAME;
  }
else
  return tokentype = c;
}
```

- Functions with variable number of parameters
  - Syntax of function

    return_type function_name (const_parameter_list, ...)
    body of the function
  - At least one parameter has to appear on the ***const_parameter_list***
- **Example ADV.3**

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



```
void minprintf (char *fmt, ...)
{
va_list ap; /* argument pointer */
char *p, *sval;
int ival;
double dval;

va_start (ap, fmt); /* ap points argument after fmt */
for (p = fmt; *p; p++) {
  if (*p != '%') {
    putchar (*p);
    continue;
    }
  switch (*++p) {
    case 'd':    ival = va_arg (ap, int);
                 printf ("%d", ival);   break;
    case 'f':    dval = va_arg (ap, double);
                 printf ("%f", dval);   break;
    case 's':    for (sval = va_arg (ap, char *); *sval; sval++)
                 putchar (*sval);       break;
    default:     putchar (*p);          break:
    }
  }
va_end (ap); /* clear */
}
```

- **`void va_start (va_list ap, parmN);`**
  Macrodefinition is called before the variable parameter list can be accessed. Argument pointer **`ap`** set to the first element of the variable parameter list (or after the last one from the constant parameter list).
- **`type va_arg (va_list ap, type);`**
  Macrodefinition is called to access successive elements of the variable parameter list. The outcome of the macrodefinition is undefined if it is called and there is no more parameters to be processed. The function should know (check) the number and types of parameters from the variable parameter list.
- **`void va_end (va_list ap);`**
  Macrodefinition should be called if **`va_start`** was executed. The macrodefinition blocks accessing the variable parameter list.

Example ADV.4

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

int min_arg (int, ...);

int main (void)
{
int i;
int j = 3;
```



```
for (i = -10; i <= 10; i++)
  printf ("%d is the least from %d, %d & %d\n",
    min_arg (3, i, j, 0), i, j, 0);

return 0;
}

int min_arg (int arg_no, ...)
{
int i;
int min, next_arg;
va_list ap;

va_start (ap, arg_no);
min = va_arg (ap, int);

for (i = 2; i < arg_no; i++)
  if ( (next_arg = va_arg (ap, int)) < min )
    min = next_arg;
va_end ap;

return (min);
}
```

Input / Output Functions

- In the standard input/output library (`<stdio.h>`) there is a definition of structure **`FILE`**, which contains implementation information (buffer address, a current position in the file, opening mode, error, EOF, etc.)
- In a program a pointer to the structure FILE is declared to depict a file.

```
FILE *fp;
```

- Function

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

is used for opening a file with the given name and in the given mode. The file name can be a full one; it can contain the directory path. The directory path is constructed according to rules defined by the operation system e.g.

```
"../Dir12/my_file.txt.1"
"$HOME/Subdir/ala.c"
"c:\dos\sys\file.prg"
```

  - If the file cannot be opened the function returns **`NULL`**. It is possible to open concurrently up to **`FOPEN_MAX`** files including always opened streams: **`stdin`**, **`stdout`** and **`stderr`**.
  - A file can be opened in one of the following modes:

`"r"`	read text file
`"rb"`	read binary file

`"r+"`	read and write text file
`"rb+", "r+b"`	read and write binary file
`"w"`	write text file
`"wb"`	write binary file
`"w+"`	write, read and create text file
`"wb+", "w+b"`	write, read and create binary file
`"a"`	append to text file
`"ab"`	append to binary file
`"a+"`	append to, write, read and create text file
`"ab+", "a+b"`	append to, write, read and create binary file

- Function

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

closes a file associated with **`fp`**. File buffers are swept and released. The function returns zero if the file was closed properly or **`EOF`** otherwise.

- Function

```
int feof (FILE *fp)
```

checks the condition of end-of-file for a file associated with **`fp`**. The function returns zero if the condition is not satisfied or a number different from zero otherwise.

- Function
  `int fflush (FILE *fp)`
  flushes the buffer associated with (file) `fp`. The function returns zero if there was no error or `EOF` otherwise.
- Function
  `int ferror (FILE *fp)`
  checks whether an error has been encountered for a file associated with `fp`. The function returns zero if the condition is not satisfied or a number different from zero otherwise. The indicator of error is set until one of the following functions is called: `clearerr`, `fclose` or `rewind`.
- Macrodefinition `errno` declared in `<errno.h>` is expanded into a global variable of type `volatile int`. At the beginning of the program the variable is set to zero. Some library functions (e.g. from `stdio`) store an error number in this variable. The meaning of error codes of the variable is implementation dependent.
- Function
  `int perror (const char *string)`
  checks value of variable `errno` and writes the standard error message to stream `stderr`. The error message is preceded by the `string` (the function parameter) followed by character `:`.

- Function
  `char *strerror (int error)`
  (from library `string`) returns address of the standard message associated with value of `error` . Number and types of error codes are implementation dependent.
- Functions
  ```
  int printf (char *format, ...)
  int fprintf (FILE *fp, char *format, ...)
  int sprintf (char *str, char *format, ...)
  ```
  are used for formatted output of variable number of data. The format is defined by parameter `format`.
  - The functions print the formatted data respectively to:
    - stream `stdout`
    - file defined by parameter `fp`
  - starting from address `str`
  - Integer values returned by the functions are numbers of characters printed if function call was successful. Otherwise a negative number is returned. Up to 509 characters is guaranteed by ANSI standard to be outputted by a single function call.
  - Parameter `format` is a string, where conversion specifications are started with character **%**. Data from the variable parameter list are printed successively according to successive conversion specifications. String `format` contains any

characters (including control ones like `\n`, `\t`), which are printed to a proper output.

- Conversion specifications
  - start with character `%`
  - can have the following (optional) fields:
    - character – : left alignment
    - character `+` : number is preceded by its sign
    - character `space` : number is preceded by – or `space`
    - character `#` : applicable to numeric data;
      - decimal point is printed for conversions `e`, `E`, `f`, `g`, `G`;
      - zero after decimal point is printed for conversions `g` and `G`;
      - `0x`(`0X`) is printed for conversions `x` (`X`)
    - character 0
      - zeros are printed instead of leading spaces for conversions `d`, `e`, `E`, `f`, `g`, `G`, `i`, `o`, `u`, `x`, `X`
      - if precision is given or character - appears in the specification then the leading zeros option is ignored for conversions `d`, `i`, `o`, `u`, `x`, `X`
    - first number : minimum width of output field
    - character . : separator of two numbers

    - second number : precision
      - number of characters for a string
      - number of digits after the decimal point
      - minimum number of digits for integer number
    - character `h` : short value
      or `l` (`L`) : long value
  - have one of following letter denoting a conversion type:
    - `c` `int` or `unsigned integer` to single character
    - `d` `int` to signed integer number
    - `e` `double` to floating point number in format [-]d.dddddd e+/-dd; at least one digit appears on the left hand side of decimal digit and on the right hand side of it there is number of digits resulting from the precision (default number is 6); for precision equal to 0 the decimal point is not printed
    - `E` like conversion `e`; `e` in number format is replaced by `E`
    - `f` `double` to floating point number in format [-]d.dddddd; at least one digit appears on the left hand side of decimal and on the right hand side of it there is number of digits resulting from the precision (default number is 6); for precision equal to 0 the decimal point is not printed

- `g` `double` to floating point number in format `e` or `f` number of digits is equal to precision; usually format `f` is chosen; if exponent is smaller than -4 or if it is greater than the precision then format `e` is taken
- `G` like `g` but choice is between formats `E` or `f`
- `i` like `d`
- `n` associated with a pointer to `int`, where the number of characters printed so far is stored; it does not influence the printing
- `o` `int` to octal number
- `p` `void*` to address in format defined by implementation
- `s` `char*` to string of characters; number of characters is limited by the precision
- `u` `int` to unsigned integer
- `x` `int` to hexadecimal number with digits `a–f`
- `X` `int` to hexadecimal number with digits `A–F`

➢ Precision in a conversion specification can be depicted by character `*` and then the precision is defined value of the successive argument from the variable argument list

```
printf ("%.*s", field_width, string);
```


▾ **Example IO.1**

```
char *format;
char *string = "hello, world";
printf (format, string);

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

char *s = "%s%n%";
printf (s);
printf ("%s", s);
```

▾ Functions

```
int scanf (char *format, ...)
int fscanf (FILE *fp, char *format, ...)
int sscanf (char *str, char *format, ...)
```

is used to formatted inputting a variable number of values.

- Conversion specifications from format define the way of translating input characters into values. The functions get data respectively from stream **`stdin`**, from file defined by **`fp`** and from memory starting from address **`adr`**. The value returned by the functions is a number of successfully performed conversions. If no conversions were possible then the functions return **`EOF`**. The functions collect characters into lexical atoms separated by white characters. The white characters are omitted for all conversions except **`c`** and **`[`**.
- Arguments on variable parameter list have to be addresses of data of appropriate types according to successive conversion specifications. If on the variable parameter list there is less arguments than number of conversion specifications then behaviour of the functions is undefined.
- Conversion specifications consist of
  - character **`%`**
  - a set of optional fields
    - character **`*`** : omitting a currently read lexical atom
    - number : maximal width of field of read characters
    - letter **`h`** : modifier for **`short int`**
      or **`l`** (**`L`**) : modifier for **`long int`** (**`long double`**)
  - letter - a specification of conversion

- **`c`** converts into characters (**`char`**) a number of characters defined by a field width (default: 1) and stores them in memory from address **`char *`** passed as the argument; **`'\0'`** is not added
- **`d`** converts an atom into integer number stored at address **`int *`**
- **`e`** converts an atom representing number in exponential form into double number stored from address **`double *`**
- **`E`** like conversion **`e`**
- **`f`** converts an atom representing number in decimal form into double number stored from address **`double *`**
- **`g`** converts an atom representing number in exponential or decimal form into double number stored from address **`double *`**
- **`G`** like conversion **`g`**
- **`i`** like conversion **`d`**
- **`n`** does not read a data; a number of read characters up to this point is stored under the given address **`int *`**
- **`o`** converts an atom representing an octal number and stores it at address **`int *`**
- **`p`** converts an atom representing address in an implementation dependent form and stores it at address **`void *`**

- **s** converts an atom (without white characters) into a string written into memory starting from address **char ***; character **'\0'** is appended at the end of string
- **u** converts an atom into unsigned integer number, which is stored at address **unsigned int ***
- **x** converts an atom representing hexadecimal number into unsigned integer number, which is stored at address **unsigned int ***
- **X** like conversion **x**
- **%** the format fits to the single character %; no input is performed (like for other "common" characters)
- **[/]** a set of characters between **[/]** has to fit characters read from input; the string is written at address **char *** and terminated by **'\0'**; a character not belonging to the set terminates the lexical atom; if after **[** there is **^** it means that valid are characters **NOT** belonging to the set; if the first character of the set is **]** or **^]** then **]** belongs to the set; the meaning of – is defined by implementation - often it denotes the range of character set, e.g. **[a-z]**

Example IO.2

```
/* Let us read a string "hello, world" into char array[20] */
  scanf ("%[^abcd]", array);
/* in the array string "hello,worl" can be found */
```

Example IO.3

```
int main (void)
{
int i, c;
char chartab[16];
scanf ("%d, %*s %%%c %5s %s", &i, &c, chartab, &chartab[5]);
printf ("i = %d, c = %c, text = %s\n", i, c, chartab);
return 0;
}
```

```
    $ test
    45, ignore_it %C read_this_text**
    i = 45, c = C, text = read_this_text**
```

Function

char *fgets (char *string, int n, FILE *fp)
reads characters from file **fp** to memory starting from address **string**. The reading is terminated if **n-1** characters has been read or end of line (**\n**) has been encountered or end of file (**EOF**) has appeared. The string is terminated by **'\0'**. If **'\n'** was read then it is written to the memory. Function returns the address of the string if everything was OK or **NULL** if **EOF** or an error was encountered.

- Function

  `int fputs (char *string, FILE *fp)`
  writes characters stored at address **`string`** into file **`fp`**. Character **`'\0'`** is not written. The function returns non-negative number if it has terminated correctly or **`EOF`** otherwise.
- Function

  `char *gets (char *string)`
  reads characters from standard input file to memory starting from address **`string`**. The reading is terminated if end of line (**`\n`**) has been encountered or end of file (**`EOF`**) has appeared. The string is terminated by **`'\0'`**Character**`'\n'`** is not written to the memory. Function returns the address of the string if everything was OK or **`NULL`** if **`EOF`** or an error was encountered.
- Function

  `int puts (char *string)`
  writes characters stored at address **`string`** into standard output file. Character **`'\0'`** is replaced into **`'\n'`**. The function returns non-negative number if it has terminated correctly or **`EOF`** otherwise.
- Function

  `int ungetc (int c, FILE *fp)`
  gets back character **`c`** to file fp opened for reading. The function returns value **`c`** if it has terminated correctly or **`EOF`** otherwise.

- ANSI C Libraries - Selected functions
  - string
    - strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr
  - ctype
    - isalpha, isupper, islower, isdigit, isspace, isalnum, toupper, tolower
  - stdlib
    - calloc, malloc, realloc, free,
    - system,
    - abort, exit, atexit,
    - bsearch, qsort,
    - rand, srand.
  - math
    - exp, log, log10, cosh, sinh, tanh, ceil, fabs, floor, pow, sqrt, acos, asin, atan, atan2, cos, sin, tan.
  - setjmp, stdarg
  - signal
  - assert
  - time
  - limit, float