Low-level programming

Lecture 3

Operators

Types conversion

Input / output operations

Institute of Computing Science
Faculty of Computing and Telecommunication

Marcin Radom, Ph.D.

Operators in C language

Arithmetic, logical, bit.

Operators priorities

Arithmetic operators

- **Binary arithmetic operators:** `* / + - %`
- **Integer division** cuts the remainder, only quotient is stored.
- **Modulo division %** (remainder). Statement ***x % y*** provides the remainder of the division of **x** by **y**, 0 if **x** is divisible by **y** without the reminder, e.g., **9 % 4 = 1**

```
int  number,  even;
even = number % 2; // 0 – even (!), 1 – odd
```

- Operator **%** cannot be used for **float** and **double** types.
- Rounding and sign depend on the arguments.
- **+ and – (binary operators)** have lower priority than *** /** and **%.** All five operators have lower priority than unary operators **+** and **–** (these two tells about the sign of a number, usually we use only **–** for negatives).
- Arithmetic operators are computed from left to right side.

Arithmetic operators

- **Example – pattern:**

$$\frac{3x^2+5x-1}{7x[(2x+3)(1-x)+5]+3}$$

```
(3 * x * x + 5 * x - 1) / (7 * x ((2 * x + 3) * (1 - x) + 5)
+ 3)
```

- Action when an overflow (i.e., when the result of an operation is greater than the type range) or an underflow (lower than range) depends on the compiler and computer architecture.

```
int  a = 1700000000,  b = 1900000000, sum;
sum = a + b;  // integer overflow
float  x = 0.5e35,  y = 0.2e5,  z;
z = x * y;    // floating point overflow
```

Arithmetic operators – Example 1

```
int main(){
     // take A value to make// an overflow
     int W, A = 2100000000;
     W = A + A;
     printf("%d\n\n", W); // negative value

     // take X value to make
     // float overflow
     double Z, X = 9E307;
     Z = X + X;
     printf("%32.15le\n\n", Z);

     // division on integers cut the remainder
     int p;
     p = 9/5;
     printf("\n%d\n\n", p); // 1
```

Arithmetic operators – Example 1

```c
// Precision may also suffer when big integer is converted to float!
int k, m = 2111333444;
float f;
//double f;

f = m;
k = (int)f;
printf("\n%d\t%d\n\n", m, k); // m = 2111333444 k = 2111333504

// first add two number (there will be overflow)
// Then convert to long it. Value however would be correct
// if long long variable have been used.
int a = 2000222333, b = 2000222333;
long long n;
n = a + b;
printf("\n%lld\n\n", n); // negative number
// when: n = a + (long long) b; to n = 40004444666
```

Arithmetic operators – Example 1

```
// While adding one small and one big number (in floats) result may be unprecise due to
// lost of precision. The sum should be calculated starting from small values and
// then going to larger ones.
     const long N = 10000000;   // number of sums
     double big = 1.8E18;    // value of large number
     double little =  1.2;   // value of small number

     double S1, S2;
     int i;
     S1 = big;

     // sum of small and large number
     for (i = 0 ; i < N ; ++i)
          S1 = S1 + little;
     printf("S1 = %28.15E\n", S1); // S1 = 1.8E18
```

Arithmetic operators – Example 1

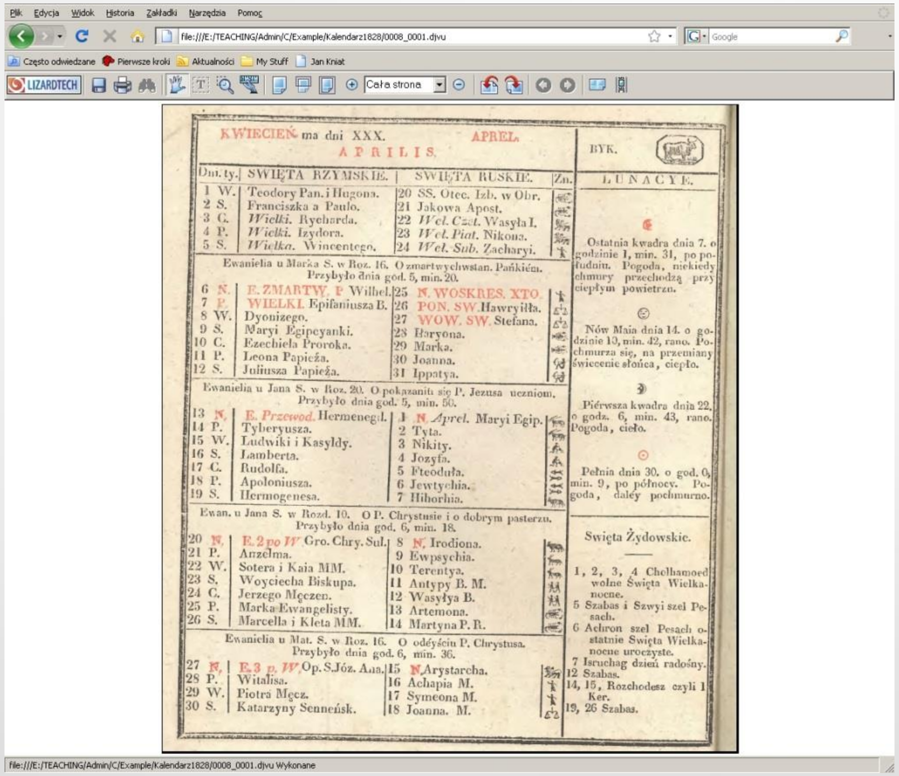
```
S2 = 0;
for (i = 0 ; i < N ; ++i)
   S2 = S2 + little;
S2 = S2 + big;
printf("S2 = %28.15E\n", S2); // small number sum: S2 = 1.8E18
// difference of sums of small and large numbers
printf("S2 - S1 = %23.15E\n\n", S2 - S1); // S2 - S1 = 1.2E7
// in theory – 10 iterations. In practice: infinite loop
// Number 0.1 (1,6*2-4) is represented as 0.100000000149011612 (double precision)
// (http://www.h-schmidt.net/FloatConverter/IEEE754.html)
double X = 0;
int N = 0;
while (X != 1.0){
     X += 0.1;
     ++N;
}
printf("End. N = %d\n", N);
return 0;
}
```

Arithmetic operators – Example 2

- **Jaen Meeus algorithm for Easter date** (the result is day and month, no exceptions, only year must be provided)

```
void main(){
     int a, b, c, d, e, f, g, h, i, k, l, m, p, n, y;
     printf("Provide year: ");
     scanf("%d", &y);
     a = y % 19;
     b = y / 100;
     c = y % 100;
     d = b / 4;
     e = b % 4;
     f = ((b + 8) / 25);
      g = (b - f + 1) / 3;
     h = (19 * a + b - d - g + 15) % 30;
     i = c / 4;
     k = c % 4;
     l = (32 + 2 * e + 2 * i - h - k) % 7;
     m = (a + 11 * h + 22 * l ) / 451;
     p = (h + l - 7 * m + 114) % 31 + 1;
     n = (h + l - 7 * m + 114) / 31;
     printf("%\nIn year %d Easter is: %d.%d .\n\n", y, p ,n);
}
```

Arithmetic operators – Example 1, year 1828



file:///E:/TEACHING/Admin/C/Example/Kalendarz1828/0008_0001.djvu

KWIECIEŃ ma dni XXX. APREL.
APRILIS.

BYK.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
1 W.	Teodory Pan. i Hugona.	20 SS. Otec. Izb. w Obr.	
2 S.	Franciszka a Paulo.	21 Jakowa Apost.	
3 C.	*Wielki.* Rycharda.	22 *Wel. Czet.* Wasyła I.	
4 P.	*Wielki.* Izydora.	23 *Wel. Piat.* Nikona.	
5 S.	*Wielka.* Wincentego.	24 *Wel. Sub.* Zacharyi.	

Ewanielia u Marka S. w Roz. 16. O zmartwychwstan. Pańkiém.
Przybyło dnia god. 5, min. 20.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
6 N.	E. ZMARTWY. P Wilhel.	25 N. WOSKRES. XTO	
7 P.	WIELKI. Epifaniusza B.	26 PON. SW. Hawryiłła.	
8 W.	Dyonizego.	27 WOW. SW. Stefana.	
9 S.	Maryi Egipcyanki.	28 Baryona.	
10 C.	Ezechiela Proroka.	29 Marka.	
11 P.	Leona Papieża.	30 Joanna.	
12 S.	Juliusza Papieża.	31 Ippatya.	

Ewanielia u Jana S. w Roz. 20. O pokazaniu się P. Jezusa uczniom.
Przybyło dnia god. 5, min. 50.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
13 N.	E. *Przewod.* Hermenegil.	1 N. *Aprel.* Maryi Egip.	
14 P.	Tyberyusza.	2 Tyta.	
15 W.	Ludwiki i Kasyldy.	3 Nikity.	
16 S.	Lamberta.	4 Jozyfa.	
17 C.	Rudolfa.	5 Fteoduła.	
18 P.	Apoloniusza.	6 Jewtychia.	
19 S.	Hermogenesa.	7 Hihorhia.	

Ewan. u Jana S. w Rozd. 10. O P. Chrystusie i o dobrym pasterzu.
Przybyło dnia god. 6, min. 18.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
20 N.	E. 2 po *W.* Gro. Chry. Sul.	8 N. Irodiona.	
21 P.	Anzelma.	9 Ewpsychia.	
22 W.	Sotera i Kaia MM.	10 Terentya.	
23 S.	Woyciecha Biskupa.	11 Antypy B. M.	
24 C.	Jerzego Męczen.	12 Wasyłya B.	
25 P.	Marka Ewangelisty.	13 Artemona.	
26 S.	Marcella i Kleta MM.	14 Martyna P. R.	

Ewanielia u Mat. S. w Roz. 16. O odéyściu P. Chrystusa.
Przybyło dnia god. 6, min. 36.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
27 N.	E. 3 p. *W.* Op. S. Józ. Ana.	15 N. Arystarcha.	
28 P.	Witalisa.	16 Achapia M.	
29 W.	Piotra Męcz.	17 Symeona M.	
30 S.	Katarzyny Senneńsk.	18 Joanna. M.	

LUNACYE.

Ostatnia kwadra dnia 7. o godzinie 1, min. 31, po południu. Pogoda, niekiedy chmury przechodzą przy ciepłym powietrzu.

Nów Maia dnia 14. o godzinie 10, min. 42, rano. Pochmurza się, na przemiany świecenie słońca, ciepło.

Piérwsza kwadra dnia 22. o godz. 6, min. 43, rano. Pogoda, ciepło.

Pełnia dnia 30. o god. 0, min. 9, po północy. Pogoda, daléy pochmurno.

Swięta Żydowskie.

1, 2, 3, 4 Cholhamoed wolne Święta Wielkanocne.
5 Szabas i Szwyi szel Pesach.
6 Achron szel Pesach ostatnie Swięta Wielkanocne uroczyste.
7 Isruchag dzień radośny.
12 Szabas.
14, 15, Rozchodesz czyli 1 Ker.
19, 26 Szabas.

file:///E:/TEACHING/Admin/C/Example/Kalendarz1828/0008_0001.djvu Wykonane

Relational and logic operators

- Relational operators are: **<, >, >=, <=** . They all have the same priority. After them (from priority point of view) there are: **==** and **!=**
- They have lower priority than arithmetic operators, therefore **i < lim – 1** is same as: **i < (lim – 1)** (so: first subtraction, then the result is compared with the left side)
- **Logical and relational conditions are also integer values!**
  - Condition is **true** if the value is **different than zero**
  - Condition is **false** if the value is **equal to zero**
  - For example: 0 > 1 has logical/arithmetical value 0, condition 0 <= 1 has „logical" value different than zero (because it's true)
- **Example:**

```c
#include <stdio.h>
int main() {
    printf("(0>1) == %d\n", (0>1));
    printf("(0<=1) == %d\n", (0<=1));
    return 0;
}
```

results:

```
(0>1) == 0
(0<=1) == 1
```

Relational and logic operators

- **Example** for relational operators:

```
enum boolean { false, true } ;
enum boolean  lower,  equal,  notequal, greaterorequal;
int  i = 5;
float  x = 12.3;
lower           = i < x;  // true
equal           = i == x; // false
notequal   = i != x; // true
greaterorequal  = i >= x; // false
```

- Logical operators: **||** (or), **&&** (and), **!** (not)
- Statements joined with these operators (**||** and **&&)** are calculated from **left to right**, to the moment when the value has been established.
- If it happens before the end of the statement**, the rest of it will not be computed** (because it is no longer necessary).

Relational and logic operators

- **Example:**

```
for(i = 0; i < lim-1 && ((c = getchar()) != '\n' && c != EOF; ++i){ //… }
```

  - First condition of the loop is ***i < lim-1*** and it is checked first. Then the other conditions are being checked (the ones joined with **&&**). If the first condition is **false**, the rest **will not be checked / computed**.
  - Numerical value of **false** condition is **0**, **true** – usually **1** (but in fact: *anything other than zero*).
  - If condition ***i < lim-1*** is **false**, then its numerical value will be **0**. It also means than all the statement is false (from **&&**/and logical table).
- Unary **negation operator** ! changes argument different than zero (true) into zero (false), false (0) into true (1), for example:

```
if (!abc)   { /* … */ }
if (abc == 0) { /* … */ } // same as above
```

  - To use any of the above depends only on the programmer preferences

Relational and logic operators

- **Example** for logical operators:

```
int a, b, c;
enum boolean z;
z = a < b && b < c;         // if a < b < c then z = 1
int year = 2000;
enum boolean leap_year;
leap_year = !(year % 4)  && year % 100 || !(year % 400);
int f = 0, k;

// if a < b then the rest of the condition is not checked
// and the f value remain unchanged. If a > b then the second
// statement is checked and the f value is incremented.
( a > b ) && ( k <= f++ );      // optimization
// if k > 5 then the second condition is not checked and the
// b value remains unchanged. If k < 5 then the second condition
// is checked and b value is changed into 7.
( k > 5 ) || (c < ( b = 7 ) );
```

Increment and decrement operators

- In C language the are two operators for incrementing and decrementing values of variables.
- Increment operator **++** adds 1 to its argument, decrement operator **- -** takes 1 from its argument.
- Operators **++** and **- -** can be used as prefixes (e.g. ++n) or as postfix (n++).
- Statement **++**n increases n **before its value will be used for anything else**, when n++ increases it **after n has been (possibly) used in a statement**.
- **Example:**

```
++alfa     --beta       // before
alfa++     beta--  // after

float  x = 2.5,  y;
x ++ ;          // equal to x = x + 1;
++ x ;          // equal to x = x + 1;
x -- ;          // equal to x = x - 1;
-- x ;          // equal to x = x - 1;
// error: (++ and --) can only be used to single variable, NOT IN STATEMENTS as below:
y = ++(2 * x);
```

Increment and decrement operators

- Using **++n** and **n++** can give different effects, e.g.:

```
int n = 5, x;
x = n++; // x = 5
n = 5;
x = ++n; // x = 6

int i = 3,  j = 4,  s;
s = j++ + i;    // s == 7   j == 5
j = 4;
s = ++j + i;    // s == 8   j == 5
```

- Sometimes it does not matter which form is used:

```
if (c == '\n') n++;
```

- In some situations only one of them can be used.

Types conversion (basics)

- **Example:** add text from table ***t*** to the end of table ***s***. Table ***s*** must be big enough (have space for all table ***t*** content):

```
void strcat(char s[], char t[]) {
    int i, j;
    i = j = 0;
    while (s[i] != '\0') i++; // find the end of signs chain s

    while ((s[i++] = t[j++]) != '\0') ; //copy t to the end of s
}
```

- The second loop **while** can be alternatively written as:

```
while (t[j] != '\0') {
    s[i] = t[j];
    i++; j++;
}
```

Bit operators

- C language offers six different bit operators:
  - **&** - bit conjunction (**AND**)
  - **|** - bit alternative (**OR**)
  - **^** - bit symmetric difference **XOR** (eXclusive OR : 'first or the second one, **but not both'** – gives 1 if one and only one argument is equal to 1
  - **<<** - bits moving left
  - **>>** - bits moving right
  - **~** - 1's complement, negation – changes all 1 to 0, all 0 to 1.
- They can be used for integer variables: **char**, **short**, **int**, **long** both with and without sign (negative and positive values)
- Example:

```
int  i = 35,  opposite,  odd_value;
opposite =  ~ i + 1;
odd_value = i & 1; // for odd values the result is 1
          // i:         0000000000100011
          // 1:         0000000000000001
          // odd_value: 0000000000000001
```

Bit operator: &

- Bit conjunction operator **&** is frequently used to **„mask" some set of bits**, e.g.:

```
n = n & 0177;
```

makes 0 for all lower bits of variable n

  - 0177 – octal form of decimal 127
  - 127 is: 01111111 (so 7 lower bits is set to 1
  - & is a bit AND, it means it will gives 0, it at least one of the arguments is 0.

- **E.g.**

```
n:        0 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1    (13755)10
0177:     0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1    (127)10
new n:    0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1    (59)10
```

Bit operator: |

- Bit alternative operator **|** is used to **„set" bits**, e.g.:

```
x = x | SET_ON;
```

where SET_ON is some vector of bits which are either set (=1) or unset (=0).

- Operation above will set all bits in x to 1 if on the same position of SET_ON the value is 1.

- **E.g.**
```
x:          0 1 1 1 0 1 1 0 0 0 1 0 1 1 0 0
SET_ON:     0 1 0 0 1 0 1 0 0 1 1 1 1 1 1 1
result:     0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1
```

Bit operator: ^

- Symmetric difference operator **^**: sets 1 only if in both arguments on the same position are different values (1 and 0 / 0 and 1), 0 if there are the same (both 1's or both 0's), e.g.

  ```
  new_number = number ^ SET_XOR;
  ```

  - **E.g.**
    ```
    number:     0 1 0 1 0 1 0 1
    SET_XOR:    1 1 1 1 0 0 0 0
    new_number: 1 0 1 0 0 1 0 1
    ```

- Other examples:

```
char a = 'r', b = 8;
a = a & 0x5F;       // changes small ASCII letters into large ones
                    // a:         01110010 ('r', 114₁₀)
                    // 0x5F:      01011111 (95₁₀)
                    // result:    01010010 ('R', 82₁₀)
b = b | 0x30;       // changes unary value 0-9 into its sign code in ASCII
                    // b:         00001000  (8₁₀)
                    // 0x30:      00110000 (48₁₀)
                    // result:    00111000 ('8', 56₁₀)
```

Bit operators: << and >>

- Operators **<<** and **>>** move bits of left argument by the number of positions given by the right argument: in the left direction (**<<**) or in right direction (**>>**). Bits moved outside of a range are given **0** value (also taken from outside range)
- E.g. ***x* << 2** moves bits in *x* by 2 positions left, **it is equivalent to the multiplication x by 4**.
- Examples for *x*:

```
x:       0 0 0 0 0 1 1 1    (7 = 4 + 2 + 1)

new_x:   0 0 0 1 1 1 0 0    (28= 4 + 8 + 16)
```

- Analogously **>>** moves bit in right direction and is equivalent to the division without remainder.
- **When moving in right direction** value without sign (**unsigned**) left over' bits are filled with 0's
- When moving bits of signed value, on some machines such bits are filled with 0's („logical" movement), on the other ones with the sign bit (‚arithmetic' movement).

```
int i = 35, r, s;
r = i << 2;             // equivalent r = i * 4;
s = i >> 3;             // equivalent s = i / 8;
```

Compound assignment operators

- Statements like *a = a + 20* (where the same variable is on the left and on the right side of the assignment operator =) can be also written as *a += 20*. Operator += is called assignment operator.
- For most binary operators there exist assignment operator ***op*=**, where *op* is:

```
*=    /=    %=    +=    -=    <<=    >>=    &=    ^=    |=
```

- **Important!!!** The whole statement on the right side is considered as single argument for the operation:

  stat1 op= stat 2 //stat1 and stat2

  is equivalent to

  stat1 = (stat1) op (stat2)

  where statement stat1 is calculated only once:

Example: `a *= b - c + 1;`

is in fact: `a = a * (b - c + 1);`

and not: `a = a * b - c + 1; (!!!)`

Compound assignment operators

- **Assignment has value and can be placed in statements.** Most often:

```
int c;
// below we read from keyboard a sequence of signs.
// If the sequence ends, getchar returns EOF (end of file
// code). c must be int type (for EOF value for example)
while ((c = getchar()) != EOF)
```

- **Example:**

```
double  price,  increment;
price += increment;   // price = price + increment;

int x = 10;
x -= 5 – 2 ;  //  x = x – (5 - 2)

int i, j, k;
i = (j = 5) + 1; // is equivalent to j = 5; i = j + 1;
i = j = k = 0;   // is equivalent to i = 0; j = 0; k = 0;
```

Conditional operator

- Lets assume we want write the bigger value (out of two) into some **c** variable. In can be done in this way:

```
if (a > b)
        c = a;
else
        c = b;
```

- It can also be done using conditional operator (three-argument): **= ? :**

```
c = ( a > b ) ? a : b;
```

- Conditional statement has general form:

{variable =} ***statement1 ? statement2 : statement3***

How it works: first statement1 is calculated. If it is **true** (different than 0) then statement2 is calculated and its value will be assigned to the opening variable on the left side. If statement1 is **false** (or has value **0**), then statement 3 is calculated and **its** value will be assigned to the opening variable (in the example above it is variable **c**).

Conditional operator

- Parenthesis are not necessary in a conditional statement – because **?** operator priority is very low.
- Conditional statements can narrow down the size of the code:
- **Example:**

```
float x, y, max;

max = x > y ? x : y; //  ()  are not required
```

- **Order/operator *sizeof***
  - Returns the number of bytes of its argument (variable, table, etc.)
  - **Example:**

```
long number_1;
size_1= sizeof number_1;  //  == 4
size_1 = sizeof (long);   //  == 4
```

Comma , as operator

- Statements separated by comma are calculated from left to right, type and value of the results is taken from the right-most argument.
- Using comma operator is limited. It is generally better to separate instruction using semicolons.
- Can be used in loop ***for***.
- **Example:**

```
long a1, a2, a3, s;
// a3 == 84000  s == 84000
s = ( a1 = 52700, a2 = 31300, a3 = a1 + a2 );
s = a1 = 52700, a2 = 31300, a3 = a1 + a2; // s == 52700

float x, y ,z;
z = ( x = 5.3, y = 2.5, y++ );  // y == 3.5  z == 2.5
```

- **Assignment operator**
  - Calculations / assignment are performed from right to left, e.g.:

```
a1 = a2 = a3 = 123;
```

Priorities and joining the operators

- In C every operator has a **priority** (it decides the sequence of calculations, e.g., **a+b*c** – first multiplication, then addition). **Joining** decides from which side the calculations start when same priority operators are being used, e.g., subtraction has left-side joining, so 3-3-3 gives -3.
- **Example:**

```
char a, b, c, d, e;
// left-side joining, so:   (((a + b) - c) - d) + e;
a + b - c - d + e;
// right-side joining, so:   a ? b : (c ? d : e);
a ? b : c ? d : e;
```

Priorities and joining the operators Example 1

```
int main ( ) {
     int nFirst, nSecond= 5, nThird;
     nFirst = 25;
     nThird = nSecond + nFirst;
          printf("\nWThe result of \n"
          „Third = Second + First\n"
          „for Second = 5 and First = 25 \n"
          "-------------------------------\n"
          " Third = %d\n\n", nThird );
      return 0;
}
```

Priorities and joining the operators Example 2

```
int main ( ) {
     double dbA, dbB;
     printf("\nCalculating statement a*a+b+1\n"
     "--------------------------------------\n"
     "Give value of a : ");
     scanf("%lf", &dbA);
     printf("\nEnter value b : ");
     scanf("%lf", &dbB);
     printf("\nResult : a*a+b+1  = %.2lf\n\n", dbA * dbA + dbB + 1);
     return 0;
}
```

Priorities and joining the operators Example 3

```
/*
  k =
     1 + x    for x > 0
     37       for x == 0
     -x - 1   for x < 0
*/
int main ( ) {
     double dbK, dbX;
     char* Text = "\nEnter value x : ";
     printf("%s", Text );

     scanf("%lf", &dbX);
     dbK = dbX > 0 ? 1 + dbX : dbX == 0 ? 37 : -dbX - 1;
     printf("\n Result : k = %5.2lf\n", dbK);
     return 0;
}
```

Priorities and joining the operators

Operators	Joining
Function execution: **()** *one-argument postfix :* **[] ->**	left-side
(typ) sizeof *one-argument prefix :* **! ~ ++ -- + - * &**	right-side
*** / %**	left-side
+ -	left-side
<< >>	left-side
< <= > >=	left-side
== !=	left-side
&	left-side
^	left-side
|	left-side
&&	left-side
||	left-side
?:	right-side
= += -= *= /= %= ^= |= <<= >>=	right-side
,	left-side

Operators summary

- Using joining rules and priorities without parenthesis makes program smaller, but increases the chances for errors and bugs.
- A good advice for a start is to use parenthesis more often, even when theoretically they are not required.
- In C **the sequence of calculations for operator arguments is not pre-defined** (except for && || ?: ,), e.g.:

```
X = fun1() + fun2();
```

  Function *fun2()* may be executed before *fun1()* or the opposite…

- Similarly, sequence of calculations for function arguments is not specified, for example:

```
my_function( ++n, reverse( n ) );
```

  - It is not known (i.e., it depends on the compiler) whether **n** will be incremented first, and then send to reverse, or the opposite.

!

Data types conversion

Theory

Types conversion

- If the operator uses arguments with different types they will be converted to a common type using rules:
  - Automatically converted are only such statements where 'lower' type is converted to the 'bigger' type without loosing information (i.e., **widening conversion** – as a result the number of bytes required for the final value is incremented)

```
int li32 = 21212345;
long long li64 = li32;     // extending conversion
```

  - Statements which does not have sense, e.g. table index as a **float** are forbidden: *table[3.14]*
  - Statements where loss of information can occur (i.e., **narrowing conversion** – number of bytes for the result is reduced) are not forbidden, but may trigger warnings from the compiler, e.g.:

```
int li32 = 21212345;
short li16 = li32;  // narrowing conversion
                    // loss of data li16 == -21319
```

Types conversion

- Variables type **char** are usually small (signed) values from 0 to 127 (0 – 7F as hexadecimal), so they can be used in arithmetical statements.
- They can provide good flexibility for conversion issues.
- Example of a program changing **a sequence of digits** into an int:

```
 int number(char s[]) {
      int i, n;
      n = 0;
      for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)   {
           // of s[i] is a digit, then subtracting '0' gives
           // us the numerical value of such a digit
           n = 10 * n + (s[i] - '0’);
      }
      return n;
}
// e.g. for "123„
// i = 0, n = 10 * 0 + ('1' - '0') = 1
// i = 1, n = 10 * 1 + ('2' - '0') = 12
// i = 2, n = 12 * 1 + ('3' - '0') = 123
```

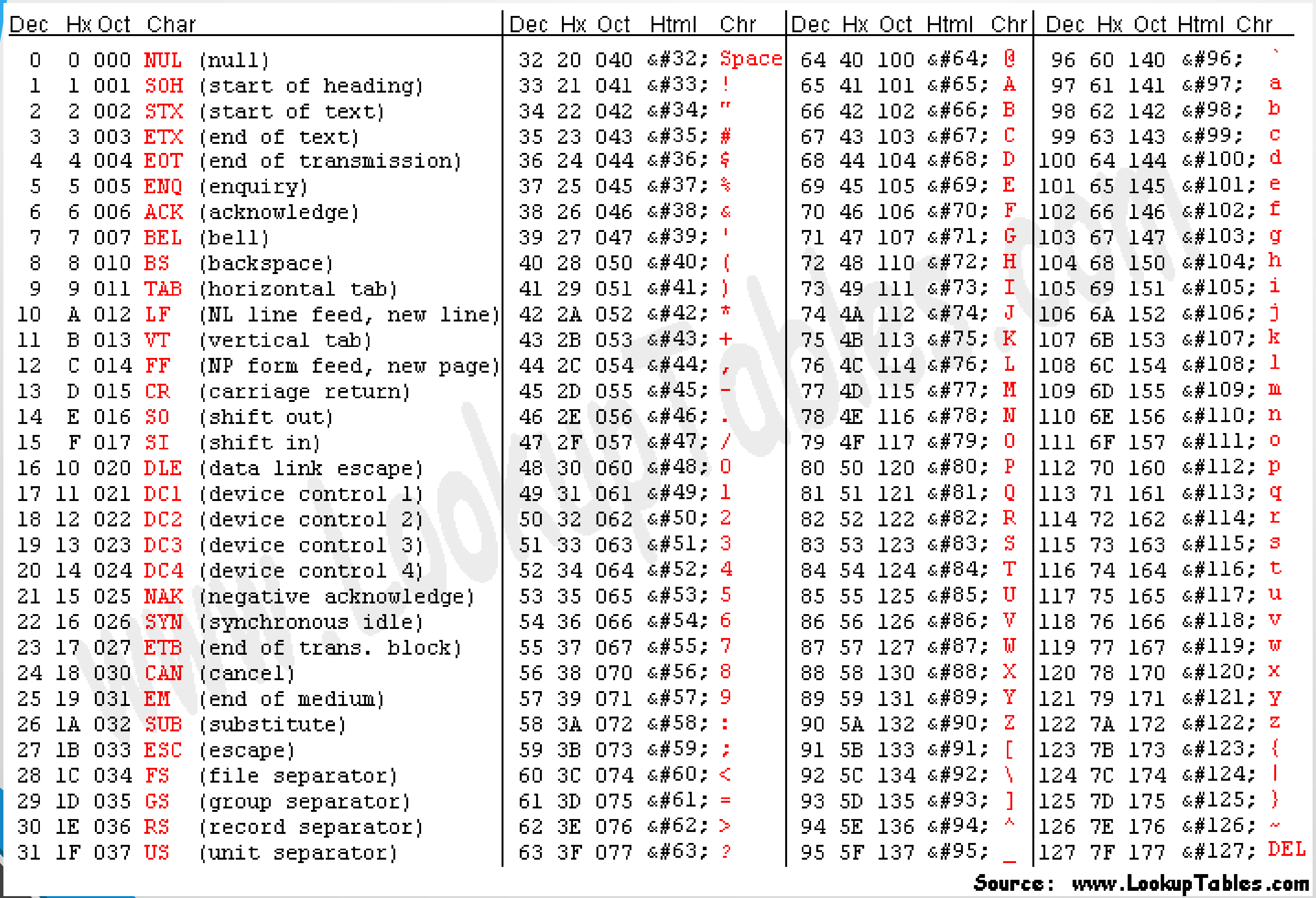
Types conversion

- Another example – a program changing large letters (and only them) into small ones:

```
 int lower(int c){
      if (c >= 'A' && c <= 'Z’)
         return c + 'a' – 'A’;
      else
         return c;
}
// e.g. for 'D’
// ASCII code 'D' == 68, 'a' – 'A' is equal 32,
// so 68+32=100, what gives ASCII code of 'd'
```

- The program works because in ASCII table numerical value distance between letters from a-z to A-Z is the same.

Types conversion



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Types conversion

- In header file **<ctype.h>** there are some interesting conversion functions.
- One of them is function **tolower(c)** which return value of a lower sign letter if c is big letter.
- For example *if(c > '0' && c < '9')* checks if **c** is a digit – it can be replaced by **isdigit(c)** from **<ctype.h>**.
- C language does not precise whether **char** values are with sign or not, so after conversion to **int** there can be some issues, e.g.:

```
char c = 255;
int a = c;                        // a == ?
```

Result depends on the machine / system / compiler.

Types conversion

- **Logical and relational statements** joined with **||** and **&&** will have value of 1 if they are true, and 0 if they are false.
- For example:

```
int d = c >= '0' && c <= '9'; // d = 1 if c is a digit
```

  Assign to the variable ***d*** value ***1*** if and only if **c** is a digit (ASCII code between 48 (zero digit) and 57 (nine digit)).

- For instructions like **if, for** or **while** 'true' in their conditional part means any value other than ZERO („not zero”).
- In other words: **while**(0) will never start, and **while**(1) is infinite loop.

Types conversion

- In general if arguments of some binary operator (e.g. ***** or **-**) have different types, the „smaller” type is promoted to a „larger” one before computing the statement.
- In other words in implicit conversions (done by the compiler automatically) smaller type is temporarily extended to larger type and the result (theoretically) is given as a larger type.
- **Rules for arithmetic conversions:**
  - If one of two arguments is **long double**, then the second one will be extended to **long double**
  - In other case, if any argument type is **double**, the second one will be implicitly converted to **double**
  - In other case, if any argument type is **float**, the second one will be implicitly converted **float**
  - Alternatively, all other objects like **char** and **short** are converted to **int**
  - Then, if any argument has qualifier **long**, then the second one will be converted to **long**.

Types conversion

- Rules become more complicated for arguments with **unsigned** qualifier.
- The result of comparison of two objects, of which one is a number with sign (**signed**), and the second one is without sign (**unsigned**) depends on the machine – **the results may vary!**
- Variables with **unsigned** can have greater values than the ones with **signed**, because in opposition to **signed**, no bit has to „remember" the sign, variables with **unsigned** are only positive.
- **Rules for arithmetic conversion when at least one argument is unsigned:**
  - If one of two arguments is **unsigned long int**, the second one will be converted (extended to) **unsigned long int**
  - In other case, if any argument is **long int**, and the second one is **unsigned int**, results depends on the fact whether **long int** can store **any value** from **unsigned int**. If yes, the argument **unsigned int** is converted to **long int**. If not, both will be converted to **unsigned long int**.
  - In other case, if any argument is **unsigned int**, then the second one is also converted to **unsigned int**.

Types conversion

- Floating point arithmetic can be calculated using single-precision – change from C99 version.
- Shorter integer types in combination with longer one with a sign does not carry no-sign property on the resulting type – change from C99 version.
- **Example:**

```
int i; char c;
i = c;  // lower value written to larger ones – OK
        // no loss due to conversion
c = i;  // Attention – this can cause a loss of data:
        // e.g. i == 258, c == ?
```

- *c = i;* - if ***i*** is larger than 255, loss of data will occur.
- If ***x*** is type **float** while variable ***y*** is type **int** then specific conversions occur. However, **y=x** (so **float** into **int**) will cause the loss of fraction part.
- There is a problem when there is no prototype of a function (then all **char** and **short** become **int**, while **float** becomes **double**).

Types conversion - casting

- In any statement we can explicitly force the conversion of types using unary casting operation.
- **Construction:**

  (**type**) statement

  Forces conversion of statement in specific type.
- Casting works as if the value of a statement should (must) be assigned to variable having *type.*
- E.g. function **sqrt()** (from: ***math.h***) calculates square root from a given number and demands the value to be **double**. If n is e.g. type **int,** then casting is requited:

```
sqrt( (double)n );
```

Types conversion - casting

- **ATTENTION**: casting creates **temporary value** of a given type. The original variable (n from the previous example) remains unchanged (its type obviously – also).
- **Example:**

```
int how_many = 27;
float that_much = 1.4;

how_many = how_many + that_much ;
/* conversion: 27 to 27.0, floating point addition, conversion 28.4 to
28, assignment */

how_many = how_many + (int) that_much ;
/* conversion 1.4 to 1, floating point addition, assignment */
```

- Casting operator has the same priority as any other unary operator.
- In general it is a suggestion to the compiler that the programmer knows what he's doing. If he really is... well, depends on skill.

Types conversion - casting

- If arguments types are declared in **function prototype** (it informs the compiler on the types of in/out parameters of a function), e.g.:

  ```
  double sq_root(double);
  ```

  then its calling:

  ```
  double result = sq_root(2);
  ```

  will force automatic conversion of integer value 2 into floating-point value 2.0 (double precision) and such a value will be send to function sq_root, <u>without additional casting in the code.</u>
- The above will works if a function has no prototype on a condition, that **sq_root** will be in the same files as function from which **sq_root** will be called and its (**sq_root**) definition will be declared before its calling.
- If such a function will be below its calling (in a code), compiler will give warning. Errors can also occur.
- **If there is no prototype, function is assumed to have all argument of type int.**

Naming new types

- In a C language there is order **typedef** which can be used to create new names for types. <u>It does not create a new type, it only provide additional name for the existing type.</u>

- **General formula:**

```
typedef    type    new_identifier
```

- **Examples:**

```
typedef   char*  string;
string  S1,  S2,  S3 = "text";

typedef   int   num;
num  k;
int  l = 5;
k = l; // type num is „equal” to type int

typedef   long    BIG;
unsigned   BIG  ww;    // error: long in type is really signed long
```

Input / output data operations

Functions **putchar**, **getchar**, **printf**, **scanf**

Output function: *putchar*

- Function from **`stdio.h`**, declared as:

```
int  putchar ( int c ) ;
```

- Sends sign *c* on standard output (***stdout***), in default state: PC monitor.
- Returns the value of the sign if everything went well, returns EOF code if there was any error / problem.
- **Example:**

```
char cc = 'R’;
putchar ( cc );  // 'R' on screen
```

- Results can be redirected to the file using sign **>**. If program uses function *putchar*, then:

```
prog > outfile
```

forces program *prog* to write its results to *outfile*, and not on the standard output.

- Using *putchar(c)* is equivalent to *putc(c, stdout).*

Output function: *puts*

- Function from **`stdio.h`**, declaration:

```
int  puts ( char  *text );
```

- Writes text in table *text* and the new line sign on the standard output: by default: on screen.
- Return positive value or EOF when there was an error.
- **Example:**

```
#include <stdio.h>
int main( )
{
    char *nn = "Some text";
     puts(nn);
    return 0;
}
```

Output function: *printf*

- Function from **`stdio.h`**, declaration:

```
    int  printf ( const char *format,  statement, statement,
... );
```

- Function *printf* transforms its argument according to the rules defined by special statement in format block. Then it sends the formatted text to the standard output. It also returns the number of written signs.
- Inside *format* block there are both signs to send into **stdout** and so called **conversion patterns**.
- Each conversion pattern starts with %, ends with some characteristic sign for a given pattern (number, letter, etc.)

Output function: *printf*

- **Example:**

```
int colors = 256;
printf( "%d", colors );
double size = 15.72;
printf( "%lf", size );
char *Text= "Documentation.";
printf ( "%s", Text);
```

- **Conversion pattern:**

  % [description] [length] [.precision] [prefix] conversion_sign

- <u>Between % and some end sign there may be:</u>
  - <u>Description:</u>
    - **-** (<u>minus</u>) – order to move the argument to the left end of a field (or add spaces from the right side).
    - **+** (<u>plus</u>) – sign of a number e.g. +35
    - <u>space</u> – space sign instead of minus e.g. 35
  - <u>Length:</u>
    - Number describing the <u>minimal</u> size of a field. Converted argument will be written into a field of a size at least *length*. If necessary, field will be extended to the full required size from the left (or from the right if ordered, e.g. by minus sign).

Output function: *printf*

  - Length of a field can be replaced by *, what means that the required value must be calculated using another argument (must be type **int**), e.g.

    ```
    printf ("Width trick: %*d \n", 5, 10);
    // Width trick:     10
    ```

- Precision:
  - dot separates length of a field from precision
  - number defining precision – e.g. number of digits for the decimal fraction, maximal number of signs for a text, or minimal number of digits for an integer (zeroes will be added if necessary).
- Prefix:
  - letter **h** if an integer argument must be written as **short** (*s* is reserved for string – sequence of signs), letter **l** – if variable is type **long**.

- **Example:**

```
printf ("Enter number in different systems: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
// 100 64 144 0x64 0144
printf ("Add spaces to the beginning: %5d \n", 1977);
// _1977
```

Output function: *printf*

- **Examples:**

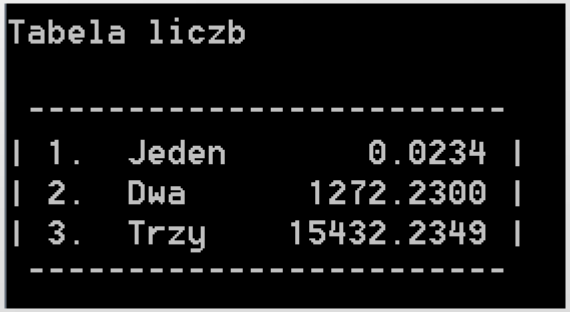
```
int Alfa = 5;
float Beta = 12.45;
printf ( "Result: \n Alfa = %d,\t Beta = %f\n", Alfa, Beta + 500);
// Result: Alfa = 5,    Beta = 512.450000

char option = 'X’;
char *Text = "Program description.";
printf("Selected option: %c : %31s", option, Text);
// Selected option: X :      Program description.

int cats= 2, *wsk_k = & cats;
float test = 23.345678;
double sum = -0.01234567;
printf("Number of cats: %d", *wsk_k );
// Number of cats: 2
printf("\nTest result = %12.3f\n Sum = %.5lf\n", test + 5, sum);
// Test result = 28.345
// Sum = -0.01234
```

Output function: *printf*

```
void main ( ) {
     int alfa = 105000;
     printf("%d", alfa);
     //%hd – short -> overflow, negative
     printf("\n\n");

     float result = 187.457f;
     printf("\nResult: %f\n\n", result);  //  round up

     long long a = 3111222333ll;
     int  i = 3111222333;
     printf("\nLong Long data: a = %lld\nInt data : i = %d\n", a, i);

     // overflow, negative
     double power = 125.4567890123456;
     printf("%lf", power);      // .2 , 12.2 – additional spaces
     printf("\n............\n\n");
```

Output function: *printf*

```c
// conversion pattern in table of signs
 long a = 5, b = 7;
 int  i = 1, j = 3;
 char Pattern[ ] = "\nLong data : a = %ld\tb = %ld"
                   "\nInt data: i = %d\tj = %d\n";
printf(Pattern, a, b, i, j);

// text formatting in printf, example:
printf("\nTable of numbers\n\n"
   " ------------------------\n"
   "| 1.  One   %10.4lf |\n"
   "| 2.  Two   %10.4lf |\n"
    "| 3.  Three %10.4lf |\n"
   " ------------------------\n\n",
        0.0234, 1272.23, 15432.2349321);
}
```



```
Tabela liczb

 -------------------------
| 1.  Jeden        0.0234 |
| 2.  Dwa       1272.2300 |
| 3.  Trzy     15432.2349 |
 -------------------------
```

Output function: *printf*

- Precision can be replaced by *****, what means that the necessary value must be taken from the next argument (must be type **int**), e.g.:

```
printf ("%.*s", max, s);
```

writes down maximally *max* signs from chains 's'

Output function: *printf*

- Declaration of *printf* in library **`stdio.h`** have pattern:

```
int printf(char *fmt, …);
```

- **…** (three dots) means unprecise number of arguments (their number and types are unknown) – they can be places only as the last arguments of a function.
- Library **`stdarg.h`** has macros allowing the creation of functions with unspecified number of arguments.
- Macro **va_start** initiates a variable (*ap* in example) for pointing on the first unspecified argument from a list …
- Another variable must be created: **va_list** to call the unspecified arguments of such a function

```
// ap – pointer for arguments
va_list ap;  // points unspecified arguments, one by one
```

- Macro **va_start** as a first argument takes variable *va_list*, and as a second – last specified argument, in our case it is *fmt*.

```
va_start(ap, fmt)  // points 1-st unspecified argument
```

Output function: *printf*

- Each calling of macro **va_arg** gives one argument and moves *ap* on the next *va_arg(ap, <type>)*.
- Name of the type is necessary to identify the searched value and the size of a step (i.e., how much *ap* must be moved). E.g.:

```
va_arg(ap, int);
```

- After all calling, macro **va_end** must be used, it will clear all variables connected with the calls (must be used before ending of a function).
- **Example of simple my_printf:**

```
#include <stdarg.h> /* minimal printf with unspecified arguments */
void minprintf(char *fmt, …){
     va_list ap; /* points each unspecified argument one by one */
     char *p, *sval;
     int ival;
     float dval;
     /* ap point 1, unspecified argument; fmt – last specified argument */
     va_start(ap, fmt);
     for (p = fmt; *p; p++) {
          // search in fmt the begging of conversion pattern
          if (*p != '%') {
               // sign outside of a pattern will be put on screen
               putchar(*p);
               continue;
          }
```

Output function: *printf*

```
switch (*++p) {
     case 'd’:          // unspecified argument is type int
          ival = va_arg(ap, int); // takes the argument
          printf("%d", ival);
          break;
     case 'f’:          // unspecified argument is type float
          dval = va_arg(ap, float); // takes the argument
          printf("%f", dval);
          break;
     case 's':          // writes text sign by sign
          for (sval = va_arg(ap, char *) ; *sval; sval++)
               putchar(*sval);
          break;
     default:
          putchar(*p);
          break;
     }
}
va_end(ap); /* clear all in the end */
}
```

//the C library macro **void va_end(va_list ap)** allows a function with variable arguments which used the //**va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined

Input function - *getchar*

- Function from the library **`stdio.h`**, declaration:

```
int getchar(void);
```

- This function reads sign by sign from the standard input (default: keyboard), returning each time the sign from input or symbolic constant EOF as **int**.
- Reading signs ends when special sign code EOF (End Of File) is reached. Usually it has value -1, but all equations should be done with EOF

```
if (getchar() != EOF)  { /* … */ }
```

better than:

```
if (getchar() != -1) {  /* … */ }
```

- In many environment keyboard can be replaced by file using sign **<**, e.g.:

```
prog < infile
```

means *prog* will read sign from file *infile*, and not from a keyboard.

Input functions – *getch, getche*

- Function from a library **`conio.h`**, declaration:

```
int getch( );
```

- Non-standard function, apart from reading a sign from keyboard it also allows to read a code of a pressed key.
- Reads from buffer if there is something in it, if it is empty calls from *getchar*. Without echo.
- Returns ASCII code or 0.
- **Example:**

```
char new_one;
new_one = getch();
```

- **getche**
  - As **`getch +`** **echo** (return sign (sign code) taken from keyboard buffer)

Input function - *scanf*

- Function from a library **`stdio.h`**, declaration:

```
int scanf(const char *format,  pointer, pointer, ...);
```

- Syntax as in **printf**, but this function *reads* from the standard input (e.g. keyboard). Interprets read signs with the patterns given in *format* and remembers the results in memory areas given by pointers. Therefore every argument must be a pointer.
- Stops when reads all given data in *format* / *pointers* or when a given value does not match conversion patter in *format*.
- Returns the number of successfully read data.
- Further calling of **scanf** starts the reading from the next unread sign by previous **scanf**.
- Reads all signs from ***stdin***.
- **Format defines conversion pattern.**

Input function - *scanf*

Argument type	Conversion sign
char	%c
short	%hd
int	%d, %i
long	%ld
long long	%lld
float	%f, %e
double	%lf, %le
long double	%Lf, %Le
char*	%s

Input function - *scanf*

- **Examples:**

```c
int number_of_pieces;
scanf ( "%d", & number_of_pieces);

double length;
scanf( "%lf", & length);
scanf( "%lf%d", & length, & number_of_pieces);

int lamps, chairs, *wsk = &chairs;
float temp;
double price;
char option;

scanf( "%d%d%f%lf", &lamps, wsk, &temp, &price);
// We write: 1 5 SP 3 4 7 Enter - 2 5 . 4 Enter
// 3 . 9 9 Enter
// Results: lamps == 15  chairs == 347  temp == -25.4   price == 3.99
```

Input function - *scanf*

```
// reading single signs
fflush(stdin);  // cleaning keyboard buffer
scanf( "%c", &option);
// A Enter
// option== 'A'

// reading texts
char Text[16]; // 16-element table
scanf ( "%15s", Text); // reads all sign TO SPACE SIGN or max. 15 signs
// A l f a  Enter
// Text == "Alfa"

scanf ( "%15[ -~]", text ); // reads text separated by spaces
// A l a  SP  m a  SP  k o t a . Enter
// Text == "Ala ma kota."
```

Input function - *scanf*

- **Examples:**

```
 int main ( ) {
    int alfa1 =  10;
    scanf("%f", &alfa1); // error

    char znak1, znak2;
    int alfa;
    scanf("%d", &alfa); // correct

    // fflush(stdin);

    // znak2 = getch(); // reads sign after ENTER
    scanf("%c", &znak1); // reads space / new line sign
    // better: scanf("%1s", &znak1);
    int alfa2 = 3, *wsk_alfa2 = & alfa2;

    scanf("%d", wsk_alfa2);          // ok
    double metric = 5.5;
    scanf("%f", &metric );           //   %lf
    int a = 5;
    char NN[5];
    //   scanf("%s", NN); // does not check memory size
    scanf("%4s", NN);
    return 0;
}
```

Questions?