A brief history of C/C++

BCPL (Martin Richards), 1967, Cambridge
B (Ken Thompson), 1970, Bell Labs
C (Dennis Richie), 1972, Bell Labs, was used in development of UNIX operating system
K & R C (Brian Kerninghan & Dennis Richie), 1978, the first standard
ANSI (American National Standard Institute) standard, 1989
C99 standard, 1999
C11 standard, 2011
C++ (Bjarne Stroustrup), 1985, Bell Labs
C++ version 2.0, 1989
C++ version 11, 2011
C++ version 14, 2014
C++ version 17, 2017
C++ is hybrid (object oriented and also procedural), C is procedural.
C is considered to be a subset of C++: everything what is written according to C rules is accepted by C++ compiler. However, it is not strictly true, because there are minor incompatibilities.
Java and C# use a lot of ideas and syntax of C/C++.

Why C/C++?

According to so-called TIOBE index (https://www.tiobe.com/tiobe-index/), the 6 most popular languages at autumn 2018 are:

1. Java
2. C
3. C++
4. Python
5. Visual Basic .NET
6. C#

C is

- A general-purpose language (good for any programming task including development of operating systems, embedded software for microcontrollers, etc.).
- Its constructs map very well the machine code instructions. Therefore the executables basing on code in C tend to be very compact and run quickly.
- Portable – C source code written for one platform can be without (or minimal) modifications used on some other platform.
- Oriented to professional programmer.
- For beginners difficult to learn.

C++ adds to C the possibility to write object oriented software.

Program in C

A program in C is a set of functions. Some of them are standard functions from IDE, the others are written (or purchased, downloaded as freeware from Internet, etc.) by programmer(s). Complicated programs consist of hundreds of C functions.

The text of non-standard functions is in source code files (*.c or *.cpp). The programmer may decide freely, how to split a long source code into separate files. Mostly, a single source file should contain a set of preferably closely related functions. For example, functions dealing with data input should be collated into one file; those dealing with data output into another file, etc. Of course, for a very simple program just one source code file is enough. Too long source code files are inconvenient to handle and therefore should be avoided.

An experienced programmer never uses filenames like *File_1.cpp* or *John_2.cpp*. A filename should somehow express the general task of functions located in this file, like *ReadData.cpp* or *Printing.cpp*.

Source code files are compiled separately. It means that everything needed for successful compilation must be present in *.c /*.cpp file itself or in the included *.h files. The compiler will not search missing information from other sources.

Identifiers and keywords (1)

Identifier refers to name given to entities such as variables or functions. Keywords like *int*, *char*, *if*, *else*, etc. are predefined, reserved words that have special meanings to the compiler.

C is case sensitive: all the keywords must be written in lowercase (*int* is a keyword but *Int* or *INT* is not).

Rules for identifiers:

- An identifier in C may include uppercase letters A, B, …Z; lowercase letters a, b, …z; digits 0, 1, …9 and underscore _. Special characters like commas or question marks as well as white characters like spaces or jumps to new line are not allowed.
- The number of characters in an identifier is not restricted but the compiler ignores characters on positions 32 and higher.
- The first character may be an uppercase letter, a lowercase letter or underscore but not a digit.
- Identifiers must be different from keywords.

Good programming practice: you can choose any name for an identifier (excluding keywords). However, if you give meaningful name to an identifier, it will be easy to understand and work on for you and your fellow programmers.

Identifiers and keywords (2)

Examples:

```
int i = 0; // i is the identifier, int is keyword
if (money_on_account != 0)
{……} // money_on_account is the identifier, if is keyword
double moneyOnAccount; // moneyOnAccount is the identifier, double is keyword
MoneyOnAccount = 100.5;
// error: as C is case sensitive, moneyOnAccount and MoneyOnAccount are different
// identifiers and here MoneyOnAccount is not defined
unsigned short int 3s; // error, an identifier cannot start with digit
long long int counter of calls; // error, the identifier cannot contain spaces
```

Good programming practice: examples how to format long identifiers:

```
long long int counter_of_calls;
long long int counterOfCalls;
long long int CounterOfCalls;
int COUNTER; // formally correct but identifiers not containing lowercase letters are
             // not recommended
char _c = 'A'; // formally correct but identifiers starting with _ are not reccomended
```

Statements (1)

Statement in C is a command that instructs the computer to take an action.

Single statement ends with semicolon.

Compound statement consists of several single statements within braces { and }. Also, a compound statement may include other compound statements. Semicolon after closing } is not needed and ignored by the compiler.

There are no specific rules for formatting the C source code: you may put any number of spaces between the entities of code, split the code into the rows as you like, etc.

Good programming practice:

- Do not put several single statements on the same line.
- After braces go to new line like

```
if (i > 10)
{
    printf("greater");
    …………
}
else
{
```

or

Statements (2)

```
if (i > 10) {
      printf("greater");
      …………
 }
else {
```

- Indent code located in braces typing a tab or some spaces before statements like

```
if (i > 10)
{
      printf("greater");
      ………..
      if (j < 20)
      {
            printf("lesser");
            ……………….
       }
 }
```

Declaration of variables

```
<type_keyword> <variable_name_1> = <initial_value_1>,
<variable_name_2> = <initial_value_2>,
………………………………………………………….
<variable_name_n> = <initial_value_n>;
```

The initial value may be a constant, identifier of another variable or an expression.
Initialization is optional.

Example:

```
int a = 10, b = a *10, c; // a is now 10, b is 100 but c may have any value
```

Variables may be declared at any place in your function. But before their first usage they must be declared:

```
double d1 = 0.1, d2;
double d3 = sqrt(d2); // error, value of d2 was not specified
```

Statement

```
const <type_keyword> <variable_name_1> = <initial_value_1>,
<variable_name_2> = <initial_value_2>,
………………………………………………………….
<variable_name_n> = <initial_value_n>;
```

declares constant variables. Their initialization is compulsory. Changing its value later is not possible.

```
const pi = 3.14159;
pi = 3.14159265; // error
```

Standard data types

Type	Bytes	Range
unsigned char	1	0….+255
signed char	1	-128….+127
unsigned short int	2	0….+65,535
signed short int	2	-32,768….+32,767
unsigned int	4	0….+4,294,967,295
signed int	4	–2,147,483,648…+2,147,483,647
unsigned long int	4	0….+4,294,967,295
signed long int	4	–2,147,483,648….+2,147,483,647
unsigned long long int	8	0….+18,446,744,073,709,551,615
signed long long int	8	–9,223,372,036,854,775,808…. +9,223,372,036,854,775,807
float	4	$-3.4*10^{38}$….$+3.4*10^{38}$
double	8	$-1.7*10^{308}$….$+1.7*10^{308}$
long double	8	$-1.7*10^{308}$….$+1.7*10^{308}$

Constants (1)

Integer constants may be in decimal, hexadecimal or octal numeral systems:

```
int i = 254;
int j = 0xFE; // 254 in hexadecimal, you may use upper- or lowercase characters
int k = 0376; // 254 in octal, allowed digits are 0,1,….7, 8₁₀ = 10₈
int m = 078; // error, 0 at the beginning means that the value is in octal, but digit 8 in
             // octal numeral system is not recognized
```

Floating point constants may be in decimal or exponential form:

```
double d1 = 31.4159;
double d2 = 0.314159e2; // 31.4159 * 10², E is also allowed
double d3 = 5.2e-4; // 0.00052
double d4 = 1e6; // 1,000,000; mantissa may be written without decimal point
```

Character constants are enclosed in apostrophes, but there are exceptions:

```
char c1 = 'x'; // for compiler it means that value of c1 is 120 or 0x78
char c2 = '+'; // for compiler it means that value of c2 is 43 or 0x2B
char c3 = '5'; // for compiler it means that value of c3 is 53 or 0x35
char cc1 = '5' + 1; // cc1 is now 54 or '6', but not 6
char cc2 = '5' + '1'; // cc2 is now 53 + 51 = 104 or 'h', but neither '6' nor 6
char cc3 = 5 + 1; // cc3 is now 6
```

Constants (2)

However

```
char c4 = '\''; // two apostrophes at the end, value of c4 is 39 or apostrophe by ASCII table
char c5 = '\"' // quotation mark and apostrophe at the end, value of c5 is 34 or quotation
               // mark by ASCII table
char c6 = '\\'; // value of c6 is 92 or backslash by ASCII table
char c7 = '\'   // error
char c8 = ' '; // space or 32 by ASCII table
```

ASCII table contains also some special characters needed to format or correct the text, for example:

```
char c9 = '\n'; // 10 or 0x0A – line feed (or LF) by ASCII table, instructs to continue
                // output on the next line
char c10 = '\t'; // 9 or 0x09 – horizontal tab (or HT)
printf("x = %d\ny=%d\n", x, y); // prints two lines – one for x and the next for y
printf("x = %d\ty=%d\n", x, y); // prints table– one column for x and the second for y
char c11 = '\0'; // all the 8 bits of c11 are zero
```

Arrays (1)

Array is a fixed-size sequental collection of elements of the same type.

One-dimensional array (vector) is declared as

<array_name>[<number_of_elements>]

Examples:

char c = 'A', mc[10]; // array identified with mc occupies 10 bytes

int i1, mi[100], i2; // mi occupies 4*100 bytes

double md[100]; // md occupies 8*100 bytes

Two-dimensional array (matrix) is declared as

<array_name>[<number_of_rows>] [<number_of_columns>]

Example:

int mmi[100][100]; // occupies 4*100*100 bytes

Generally, the number of dimensions is not limited, but three and more dimensional arrays are very seldom used.

To declare an array, one has to specify the type of members, the name of array and the dimension(s). The dimension(s) (enclosed in brackets) must be constant(s) (not variable).

Arrays (2)

Initializtion of array members is also possible:

<array_name>[<optional_number_of_elements>] = {
<sequence_of_initial_values_separated_by_commas> }

Example:

```
int mi[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

or

```
int mi[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // dimension is specified by the number of initial
                                            // values
```

To access a member of array, write the array name and the ordering number (index) of the member. The index is also closed in brackets, for example:

```
mc[0] = 10;
```

The index of the first member is always 0. The index of the last member is always dimension -1. Members of *mi* are *mi[0]*, *mi[1]*, …., *mi[9]* but not *mi[10]*.

The index may be an integer expresson:

```
int n, m, mi[400];
n = 5;
m = mi[n * 20] + mi[n * 21] / 2; // mi[100] + mi[105] / 2
```

But

```
mi[n * 100] = 0; // serious error, m[500] does not exist, memory is now corrupted
```

Assignment

Assignment operator in C is =. For example

x = y + z; // sum of *y* and *z* is stored on the memory field identified by name *x*

An expression may contain more than one assignment, for example

a = b = c = x + y; // sum of *x* and *y* is first stored on memory field identified by name *c*.
// After that the value of *c* is copied into *b* and at last the value of *b*
// is copied into *a*.

However,

a = b = c +1 = x + y; // error

On the left from assignment operator must be so called lvalue (here l means left): an identifier or expression that specifies a certain memory field. *c+1* is not such an expression.

On the same time expression:

d = e + (c = a + b);

is correct, because due to parentheses *c* first gets its new value *a* + *b* and then the new *c* is added to *e*.

Good programming practice: do not try to be too clever and write expressions that are difficult to understand.

Arithmetical operators (1)

Let

int a = 15, b = 2, c;

Then

c = a + b; // addition, c is now 17

c = a – b; // subtraction, c is now 13

c = a * b; // multiplication, c is now 30

c = a / b; // division, c is now 7 and not 7.5 or 8

c = -a; // sign operator, c is now -15

c = a % b; // modulus operator, gives the reminder, c is now 1

The sign operator is unary (just one operand). The others are binary operators (two operands).

Fraction resulting from integer division is simply discarded: 5 / 2 is 2, 5 / 3 and 5 /4 are 1, 1 / 5 and 4 / 5 are 0. But 5.0 / 2.0 is 2.5 and 1.0 / 5.0 is 0.2.

Modulus operator is defined only for integer operands: 5 % 3 is 2, 3 % 5 is 3, 6 % 3 is 0, 0 % 3 is 0.

Arithmetical operators (2)

Exercise:

Write a program that reads from keyboard an integer and determines is it even or odd. The result should be printed as "This number is even" or "This number is odd".

Tips:

- For input use standard functions as they were used in the second program: *gets_s* and *atoi*.
- Print the result using standard function *printf*.

Relational operators

Let

```
int a = 15, b = 2, c;
```

Then

```
a == 15 // true, a is equal to 15
a != 15 // false because we claim that a is not 15
a != 10 // true because we claim that a is not 10
a > b // true, a is really greater than b
a >= 15 // true, a is really equal or greater than 15
a < b // false, a is not less than b
a <= 15 // true, a is really equal or less than 15
```

Generally, any value of any type that is not zero is handled as "true" and any value of any type that is zero is handled as "false". But if the result of relational operator is stored or used as an temporary value in other expressions, the "true" is integer 1 and the false is integer 0.

```
c = a == 15; // c is now 1
c = a != 15; // c is now 0
c = (a == 15) + b; // c is now 3
```

Logical operators (1)

Boolean algebra (by George Bool 1815 – 1864) is a branch of algebra in which the variables may have only 2 values: TRUE and FALSE. In traditional algebra, the variables are numbers and the operations between them are addition, subtraction,etc. In Boolean algebra there are only 3 operations:

Logical AND or conjuction or logical multiplication, in C the && binary operator:
TRUE && TRUE = TRUE (read "true and true is true")
TRUE && FALSE = FALSE (read "true and false is false")
FALSE && TRUE = FALSE (read "false and true is false")
FALSE && FALSE = FALSE (read " false and false is false")

Logical OR or disjuction or logical addition, in C the || binary operator:
TRUE || TRUE = TRUE (read "true or true is true")
TRUE || FALSE = TRUE (read "true or false is true")
FALSE || TRUE = TRUE (read "false or true is true")
FALSE || FALSE = FALSE (read " false or false is false")

Logical NOT or negation, in C the ! unary oprator:
!TRUE = FALSE (read "not true is false")
!FALSE = TRUE (read "not false is true")

Logical operators (2)

```
if ((x >= 2) && (x <= 4))
    printf("x is in range 2…4\n");
```

For example, if *x* is 3, the *x>= 2* is 1 (TRUE) and *x <= 4* is also 1 (TRUE). So we get TRUE and TRUE or in other words the result of expression after *if* is 1 (TRUE).

```
if ((x < 2) || (x > 4))
    printf("x is not in range 2…4\n");
```

For example, if *x* is 1, the *x < 2* is 1 (TRUE) and *x > 4* is 0 (FALSE). So we get TRUE or FALSE or in other words the result of expression after *if* is 1 (TRUE).
If *x* is 5, *x < 2* is 0 (FALSE) and *x > 4* is 1 (TRUE). So we get FALSE or TRUE or in other words the result of expression after *if* is 1 (TRUE).
If *x* is 3, *x < 2* is 0 (FALSE) and *x > 4* is also 0 (FALSE). So we get FALSE or FALSE or in other words the result of expression after *if* is 0 (FALSE).

```
if (x) // the same as if (x != 0)
{ // a trick used by experienced programmers
  …………..
}
```

```
if (!x) // the same as if (x == 0) because !0 gives 1
{ // a trick used by experienced programmers
  ……………
}
```

Logical operators (3)

Exercise:

Write a program that reads from keyboard a text (max 80 characters) and determines is the first character a digit or an English alphabet letter (uppercase or lowercase) or some other character. The result should be printed as "The text starts with digit" or " The text starts with uppercase letter" or " The text starts with lowercase letter" or " The text starts with unidentified character" .

Tips:

- For input use standard function *gets_s*.
- Print the result using standard function *printf*.
- To reason out the solution open the ASCII table (http://www.asciitable.com/) and think a bit!
- First draw the flowchart and only after that start to write code!

Looping (1)

The computer executes the program statements sequentially: the first statement is executed first, the second statement after it, etc. The *if* and *if else* branching statements allow to select, which statements to execute and which to skip.

The loop statement allows to execute a single or compound statement multiple times

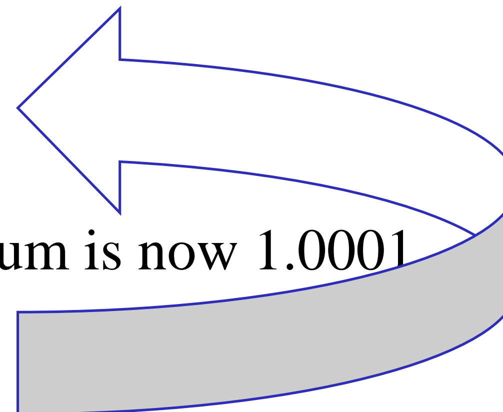
```
while (<condition to repeat>)
    <statement>
```

Example: we need to find the average of elements in an array:

```
double array[7] = { 1.0001, 1.256, 111.4, 44.5, 6.789, 12. 47, 54.98}, average, sum = 0.0;
int i = 0;
while (i < 7) // when becomes FALSE, the repeating execution of loop body ends
{
    sum = sum + array[i];
    i = i +1;
}
average = sum / 7;
```

When the loop runs first:

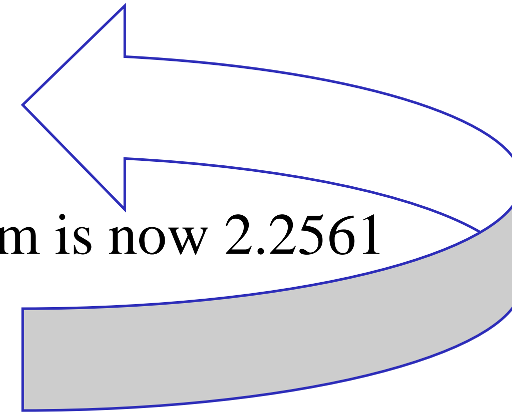
```
while (i < 7) // 0 < 7 is TRUE, execute the loop
{
    sum = sum + array[i]; // sum = 0 + 1.0001, i.e. sum is now 1.0001
    i = i +1; // i is now 1
} // jump back to row containing the while keyword
```



Looping (2)

When the loop runs the second time:

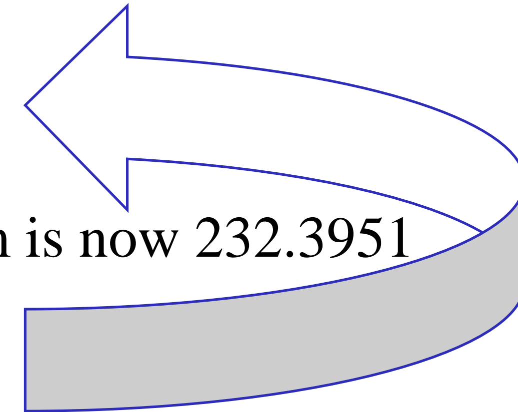
```
while (i < 7) // 1 < 7 is TRUE, execute the loop
{
    sum = sum + array[i]; // sum = 1.0001 + 1.256, i.e. sum is now 2.2561
    i = i +1; // i is now 2
} // jump back to row containing the while keyword
```



……………………………………………………..

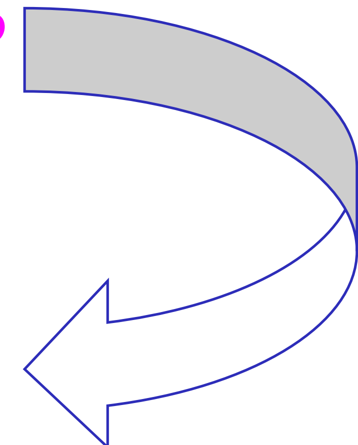
When the loop runs seventh time:

```
while (i < 7) // 6 < 7 is TRUE, execute the loop
{
    sum = sum + array[i]; // sum = 177.4151 + 54.98, i.e. sum is now 232.3951
    i = i +1; // i is now 7
} // jump back to row containing the while keyword
```

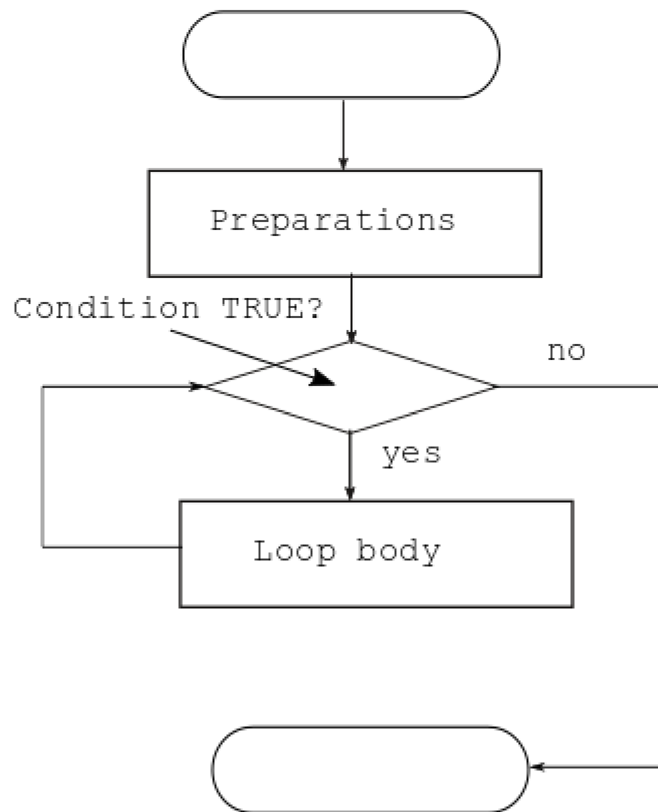


When the loop runs eighth time:

```
while (i < 7) // 7 < 7 is FALSE, jump to the statement following the loop
{
    sum = sum + array[i];
    i = i +1;
}
average = sum / 7;
```



Looping (3)



Generally, the preparations are needed for setting the variables the loop uses into their initial state. The condition is an expression enclosed into parentheses. This expression is recalculated and tested on each iteration. If the result of expression is TRUE (i.e. nonzero), the loop body statement is executed. The loop body should also modify the variables the condition uses. If the result is FALSE (i.e. zero), the body is not executed and the statement executed next is the stament located right after the loop body.

The condition may be FALSE right at the beginning. In that case the loop is not executed.

If the condition never becomes FALSE (for example, because there are errors in test expression or in the loop body), the loop never terminates and the program hangs.

Looping (4)

Exercise:
Write a program that generates 50 pseudo-random integers. If the integer is less than 500, print it (each on its own command prompt window row), otherwise ignore.

Tips:

- To generate psedo-random integers use standard function *rand*. Example:

```
#include "stdlib.h"
int r = rand(); // returns a pseudo-random integer, has no arguments
```

- Print the results using standard function *printf*.

Exercise:
Write a program that fills an integer array

```
int Fib[20];
```

with members of Fibobacci sequence 1, 1, 2, 3, 5, 8, ….

$f_0 = 1$

$f_1 = 1$

$f_i = f_{i-1} + f_{i-2}$ for each $i > 1$

To check the results print them using standard function *printf*.

Looping (5)

Exercise:
Write a program that fills an integer array
int Rand[50];
with pseudo-random numbers, finds the maximum of them and using the standard function *printf* prints the result.

Exercise:
Write a program that:

1. Reads from keyboard a text consisting of any number but no more than 80 characters.
2. Checks does this text contains characters that are not English alphabet uppercase or lowercase letters. If the answer is yes, the program must print message "Illegal input" and quit.
3. Determines how many letters "a" (uppercase as well as lowercase) it contains.
4. Prints the result.

Tips:

- For input use standard function *gets_s*. Remember that the array filled by *gets_s* is a string, i.e. its last character is automatically set to '\0'.
- Print the result using standard function *printf*.
- First draw the flowchart and only after that start to write code!

Looping (6)

Sometimes it may happen that we need to jump out from the loop earlier, i.e. when the condition is still TRUE. In that case we may jump to the statement following the loop using statement:

```
break; // just one word in statement
```

Example:

```
int array[10], i = 0, x;
char line[81];
printf("Type 10 even integers");
while (i < 10)
{
    gets_s(line);
    x = atoi(line);
    if (x % 2 == 0) // check is the inputted integer even
        array[i] = x; // yes, it is even, insert into array
    else
    {
        printf("Error\n"); // no, it is odd, print error message
        break; // break the loop off, jump out although i < 10
    }
    i = i + 1;
}
```

Looping (7)

Exercise:
Write a program that:

1. Reads from keyboard a text consisting of any number but no more than 80 characters.
2. Determines how many undercase vowels ('a', 'e', 'i', 'o', 'u', 'y') it contains.
3. Prints the result.

Tips:

- For input use standard function *gets_s*. Remember that the array filled by *gets_s* is a string, i.e. its last character is automatically set to '\0'.
- To check is the character an undercase vowel, first declare and initialize an array of vowels. Thus, the program will contain two loops: the outer loop takes the characters from input text and the other (inner or nested loop) checks can it found from the array of vowels. If the result of searching is positive, increment the counter of vowels and jump out from the inner loop.
- Print the result using standard function *printf*.
- First draw the flowchart and only after that start to write code!

Looping (8)

Exercise:

Write a program that reads from keyboard characters using standard function *_getche:*

```
#include "conio.h"
char c = _getche();
```

_getche forces the program to stop and wait for a keypress. It prints the character corresponding to pressed key (additional ENTER is not needed) and returns the ASCII code.

The characters should be stored in array

```
char text[21];
```

The reading must be stopped when

- The array text already contains 20 characters (i.e. is full). In that case put *'\0'* at the end of array (i.e. text[20]), print the result and exit.
- The user presses ESC (it means that _getche returns 27). In that case put *'\0'* after the last significant character (i.e. replace 27 with zero), print the result and exit.

Tips:

- Print the result using standard function *printf:*

```
printf("Result is: %s\n", text);
```

- First draw the flowchart and only after that start to write code!

Looping (9)

Sometimes it may happen that we are not able to set the condition. In that case we may write the condition as 1 (i.e. always TRUE) and include the check "continue or break" into the loop body.

Example: we need a random number that exceeds 100:

```
int ran_val;
while (1)
{
    ran_val = rand();
    if (ran_val > 100)
        break;
}
```

This code snippet may be written shorter:

```
while ((ran_val = rand()) < 100);
```

As *(ran_val = rand())* is in parentheses,the first step is to get the pseudo-random value and store it in variable *ran_val*. This is also the value which will be compared with 100. Semicolon at the end of statement means that the loop body is empty.

Good programming practice: do not try to be too clever and write expressions that are difficult to understand.

Looping (10)

Exercise:
Write a program that reads from keyboard characters using standard function *_getche*. If the character is not a lowercase vowel ('a', 'e', 'i', 'o', 'u', 'y'), ignore it. If it is, increase the counter corresponding to it. If the user presses ESC, stop reading, print the resulting tabel and quit.

Tips:

- At first create an array of counters:
  `int counters[6];` // here is the declaration without initializations
  and initialize it with zeroes. Create also an array of vowels and initialize it with 'a', 'e', …
  `char vowels[6];`
  Searching of the character from array vowels you will also get the index for counters.
- Print the rows of resulting table using standard function *printf* according to the following format:
  `printf("%c: %d\n", vowels[i], counters[i]);`
- First draw the flowchart and only after that start to write code!

Functions (1)

A program in C is a set of functions. Each function accomplishes a particular task.

Each C function has its name (must be unique within the current program). The function name is a regular C identifier (like a variable name) and, altough may be chosen freely, should be meaningful, i.e. somehow express the task the function must perform.

Exception: the program must contain one and only one function which has name *main* (this is the C keyword). The operating system starts the execution from the *main* function which calls some other functions, those also may call other functions etc. Programs not containing the *main* function cannot run.

Functions produce actions and may or may not provide values that they return to the calling function. For example, standard function *sqrt* calculates the square root and returns the result to the calling function:

```
double x = 100, y = sqrt(x); // result returned by sqrt are assigned to y
```

Standard function *exit* terminates the program immediately but returns nothing:

```
#include "stdlib.h"
exit(0); // here 0 is the status code, traditionally code 0 means that there were no errors
```

Functions (2)

The general form of a function definition is:

```
<return_value_type> <function_name>(< formal_ parameter_list>) // called also as header
{
 <function_body> // sequence of statements
}
```

The parameter list specifies the names and types of formal parameters:

```
(<type of parameter_1> <name_of_formal_parameter_1>,
 <type of parameter_2> <name_of_ formal_ parameter_2>,
 ………………………………………………………………
<type of parameter_n> <name_of_ formal_ parameter_n>)
```

A function may not have formal parameters. In that case

```
<return_value_type> <function_name>( )
{
 <function_body>
}
```

A function body may not contain definitions of other functions.

Functions (3)

If the function just produce some actions but does not return any values to the calling function, its return value type is *void* (C keyword).

If the function has return value, its body must contain at least one statement

return <expression_to_find_value_to_be_sent_to_calling_function>;

Examples:

```
int main() // int is the return value type, main is the name, parameter list is empty
{ // body starts
    printf("Hello");
    return 0; // returns value 0
 } // body ends
```

```
double Max(double a, double b) // but not double max(double a, b)
{ // returns the biggest of the two double values, a and b are formal parameters
  if (a > b)
    return a;
  else
    return b;
}
```

Functions (4)

```
void PrintMax(double a, double b)
{ // prints the biggest of the two double values but returns nothing
 if (a > b)
     printf("%lg\n", a);
 else
     printf("%lg\n", b);
  return; // here the return without value is allowed but not obligatory
}

double LawOfCosines(double side1, double side2, int angle)
{ // side1 and side 2 are sides of a triangle, angle (in radians) is between them
  // computes the third side of triangle
  // Checking of input:  the arguments must be positive
 if (side1 >= 0 && side2 >= 0 && angle >= 0)
     return sqrt(side1 * side1 + side2 *side2 -2 * side1 * side2 * cos(angle));
 else
     return NAN; // as the return value is not void, the function must return something,
                       // NAN (not a number) is a constant telling that there was no answer
}
```

Functions (5)

```
void PrintBar(int length)
{ // prints bar like *********
  int i = 0;
  if (length <= 0) // checks is the parameter length (number of * to print) correct
 {
        return; // errors, returns immediately, return statement is obligatory
  }
  while (i < length)
  {
        printf("*");
        i = i + 1;
   }
   printf("\n");  // cursor to the next row
   return; // return statement is optional, but useful for setting breakpoints
 }
```

Functions (6)

Expression calling the function:

<function_name>(< actual_ parameter_list>)

The parameter list specifies the values of actual parameters:

(<expression_presenting_the_value_of_actual_parameter_1>,

<expression_presenting_the_value_of_actual_parameter_2>,

…………………………………………………………………

<expression_presenting_the_value_of_actual_parameter_n>)

If the function has no parameters, the calling expression is

<function_name>( )

When a function is called then

1. The expressions presenting the values are performed and the results copied into formal parameters
2. The executing of calling function is interrupted and the computer starts to execute the called function
3. When the called function ends, its return value (if exists) is thrown back into calling function
4. The calling function resumes its work

Functions (7)

In case of function

```
double Max(double a, double b)
{
 if (a > b)
     return a;
 else
     return b;
}

double m, x = 5.0, y = 6.0;
m = Max(x, y); // value of x is copied into a and value of y is copied into b
               // the function returns 6.0 which is assigned to m
m = Max(log10(x) + log(x), y *y) + 10;
          // 1. Value of x is copied into formal parameter of  log10 (base 10 logarithm).
          // 2. log10 returns the logarithm, this is stored in a temporary (and unvisible
          // for us) variable.
          // 3. Value of x is copied into formal parameter of  log (base e logarithm).
          // 4. log returns the logarithm, this is stored in another temporary variable.
          // 5. The values of temporary variables are added, the sum copied into a.
          // 6. y * y is calculated and the result is copied into b.
          // 7. Max start to run, finds the bigger value and sends back to calling function.
          // 8. The value assigned to m is the result of Max + 10
```

Functions (8)

Function definition may be in one source code file and the call to it in another one. Therefore, to check and compile the call, the compiler needs the declaration (prototype) of function:

<return_value_type> <function_name>(<parameter_list>); // semicolon at the end

Example:

```
double Max(double a, double b);
```

or as the variable names in the declaration do not have any sense, simply

```
double Max (double, double);
```

The declarations are mostly located at the beginning of source code file before the definitions of functions. In large programs the declarations are collated into a separate *.h file which is included together with the standard *.h files.

Calls to not declared functions are errors.

Declarations of standard functions are in standard *.h files.

Functions (9)

Exercise:

Write of function that finds and returns the biggest of the given three numbers. Its declaration must be

```
int MaxOfThree(int, int, int);
```

The parameters are the given values. The function must return the biggest number.

To test the function write function *main* that reads from keyboard the initial data, calls your function and types the result.

Tips:

- For input use standard functions as they were used in the second program: *gets_s* and *atoi*. For output use *printf*.
- As an alternative, instead of reading test data from keyboard you may write them directly into *main*.
- Do not forget to put the declaration (prototype) after *#include* directives.
- Remember that the two or even all the three input parameters may be identical. Try the following tests:

```
int m1 = MaxOfThree(1, 2, 3);
int m2 = MaxOfThree(3, 2, 1);
int m3 = MaxOfThree(1, 3, 2);
int m4 = MaxOfThree(2, 2, 3);
int m5 = MaxOfThree(2, 2, 2);
```

Functions (10)

Exercise:

Write of function that converts a temperature value from Celsius to Farhenheit or vice versa:

$T_F = T_C * 9 / 5 + 32$

$T_C = (T_F - 32) * 5 / 9$

The declaration must be

```
double TempConverter(double, char);
```

The first parameter is the temperature to convert. The second parameter must be 'F' or 'f' if the input data is in Farhenheit or 'C' or 'c' if in Celsius.

To test the function write function *main* that reads from keyboard the initial data, calls your function and types the result.

Tips:

- For input use standard functions *gets_s*, *atof* and *_getche*. For output use *printf*. As an alternative, instead of reading test data from keyboard you may write them directly into *main.*
- If the second parameter is wrong (not from the 4 allowed characters), the output value must be *NAN*. Therefore the *main* must check the result calling standard function *isnan* (its argument should be the output of *TempConverter*). If *isnan* returns TRUE (i.e. 1), the *main* must print "No solution". *NAN* and *isnan* need header file *math.h*.

Preprocessor directives (1)

C/C++ compilers perform some preparatory work on source code before compiling (preprocesses the source code). Preprocessing is controled by preprocessor directives:

- Each preprocessor directive must be on its own source code row.
- The first non-white character on directive row must be #.
- Do not use semicolons. Jump to the new row marks the end of directive. However, backslash \ at the end of directive row means that the directive will continue on the next row.

#include <filename> // for example #include <stdio.h>
#include "filename" // for example #include "stdio.h"
is the same as if you had typed the entire contents of included file into your source code at the point where the directive line appears.

The difference between the form with angle brackets and the form with quotation marks is the order in which the preprocessor searches for the files to be included (turn attention that the path is incomplete). In our examples the both forms lead to the same result. Who is interested in details may read https://msdn.microsoft.com/en-us/library/36k2cdd4.aspx

You cannot use C/C++ standard tools (functions like *printf* or constants like *NAN*) without including standard header files presenting the definitions and declarations the compiler needs. In Visual Studio: to know which *.h header file is required, click on the name of function and press F1 - it opens the proper page of Microsoft documentation.

Preprocessor directives (2)

```
#define <object_macro> <replacement_value>
```

for example

```
#define PI 3.14159 // PI is the object macro, 3.14159 is the replacement value
```

A token is the smallest element of text that is meaningful for the compiler. The tokens are identifiers, keywords, constants, operators (i.e. =, ==, +, -, && etc.).

When the preprocessor finds a token matching the object macro, it replaces it with the replacement value. For example

```
double y = sin(PI * 3 / 2);
```

actually means

```
double y = sin(3.14159 * 3 / 2);
```

But for example in declaration

```
double yPI23;
```

nothing is replaced because here *PI* is not a separate token but a part of token.

Good programming practice: define your macro in uppercase and only in uppercase letters.

There are also function macros, but they are out scope of our course.

Preprocessor directives (3)

Advantage of macros: suppose we have a big C program handling files. Starting our work we assume that the length of a file is 10,000 bytes and we write this integer into code in a lot places. However, later we discover that the correct length is 20,000 bytes. To fix the error we have to find all the occurances of 10,000 and replace them with 20,000.

As a rule, a big project includes at least one header file created by the programmer himself / herself. This file contains the prototypes of all the non-standard functions of that project and is included like the standard header files.

Now if we had put into this *.h file directive:

```
#define FILE_LENGTH 10000
```

and everywhere in our code instead of 10,000 had written FILE_LENGTH then all we have to do to fix the error is to replace in this definition 10,000 with 20,000.

In addition, using of macros instead of constants makes our code more readable.

Input / output standard functions (1)

We already know some mathematical functions, *gets_s*, *atoi, atof, _getche, exit, printf.* Some remarks:

If the input data is not correct (for example, contain spaces or letters) and therefore the converting into *int* or *double* is not possible, *atoi* and *atof* return zero. Consequently, output value zero may mean that the human operator typed zero and also that the human operator made an error. We'll deal with better input functions later.

_getche has a "sister" function *_getch*. The only difference is that when the human operator presses a key, *_getche* echoes it on the screen but *_getch* does not. *_getch* is good to stop the program.

Examples:

```
int main()
{
  ........................ // do something
  _getch();  // program stops here and waits for a keystroke
}
```

or

```
printf("Read the message. When ready, press a key");
_getch(); // any key is OK
```

Input / output standard functions (2)

The last example, however, may not run correctly because the keyboard buffer may already contain some characters (for example, the operator had inadvertently hit a key) that are not poped out yet. If this is the situation, *_getch* reads the first of them and the program rushes on without any waiting.

To clean the keyboard buffer, write:

```
while (kbhit() != 0) // or simply while (kbhit())
    _getch();
printf("Read the message. When ready, press a key");
_getch();
```

kbhit returns TRUE if there is at least one character waiting for reading into the memory and FALSE if the keyboard buffer is empty. But *kbhit* does not pop out the character.

printf has at least one parameter: the text called as format string. This text may contain format specifiers starting with %. The format specifiers are placeholders that are replaced by values specified in the parameter list that follows the format string.

Examples:

```
printf("Hello\n"); // no format specifiers, after printing the cursor is set to the next line
printf("x = %d\ny = %d\n", x, y); // format specifiers %d are replaced by values x and y
                                  // x and y must be of type int
printf("%d%% full!\n", n); // prints for example "100% full", here %% means one %
```

Input / output standard functions (3)

Specifier	Type of the corresponding argument
%d or %i	signed int
%hd or %hi	signed short int
%hhd or %hhi	signed char, printed as number
%ld or %li	signed long int
%lld or %lli	signed long long int
%u or %x or %X	unsigned int, x means hexadecimal in lowercase and X in uppecase
%hu or %hx or %hX	unsigned short int
%hhu or %hhx or %hhX	unsigned char, printed as number
%lu or %lx or %lX	unsigned long int
%llu or %llx or %llX	unsigned long long int
%c	signed char, printed as symbol

```
char c = 'A';
printf("%hhd %c", c, c); // prints "65 A"
unsigned char uc = 'Z';
printf("%hhu %hhx %hhX", uc, uc, uc); //prints "90 5a 5A"
```

Input / output standard functions (4)

Specifier	Type of the corresponding argument
%f or %F	double
%e or %E	double, scientific notation (mantissa & exponent), lowercase or uppercase
%g or %G or %lg or %lG	double, actually %f, %e or %E depending on which is shorter

In *%f, %F, %e* and *%E* specifiers the default value of digits after decimal point is 6.
To change it complete the specifier with precision:

%.<number_of_digits_after_decimal_point><f_or_F_or_e_or_E>

```
double d = 314159.265;
printf("%f    %.3f", d, d); // prints "314159.265000    314159.265"
printf("%e    %.3e", d, d); // prints "3.141593e+05    3.142e+05", rounded
```

But in %g and %G the precision has another meaning:

%.<number_of_digits_to_print><g_or_G>

```
printf("%g    %.3G", d, d); // prints "314159    3.14E+05", rounded
```

Be careful: printing with small number of digits may lead to corrupted results!

It is also possible to justify the fields, left-pad with zeroes, set the width of field, etc.
Read the specification of *printf* on http://www.cplusplus.com/reference/cstdio/printf/