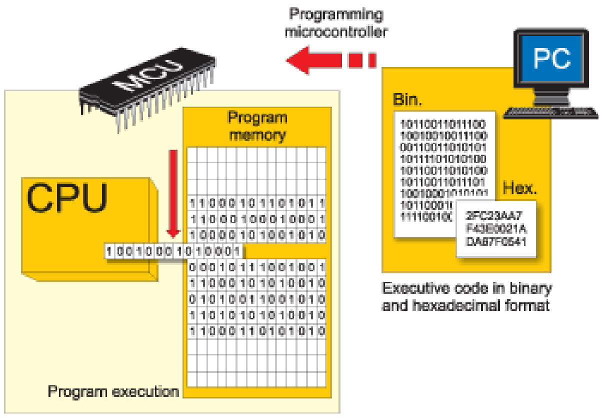
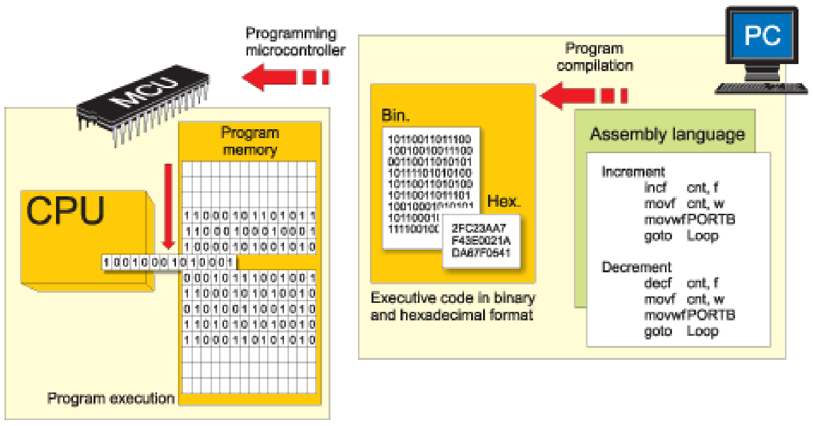
Table of contents

PROGRAMMING LANGUAGES

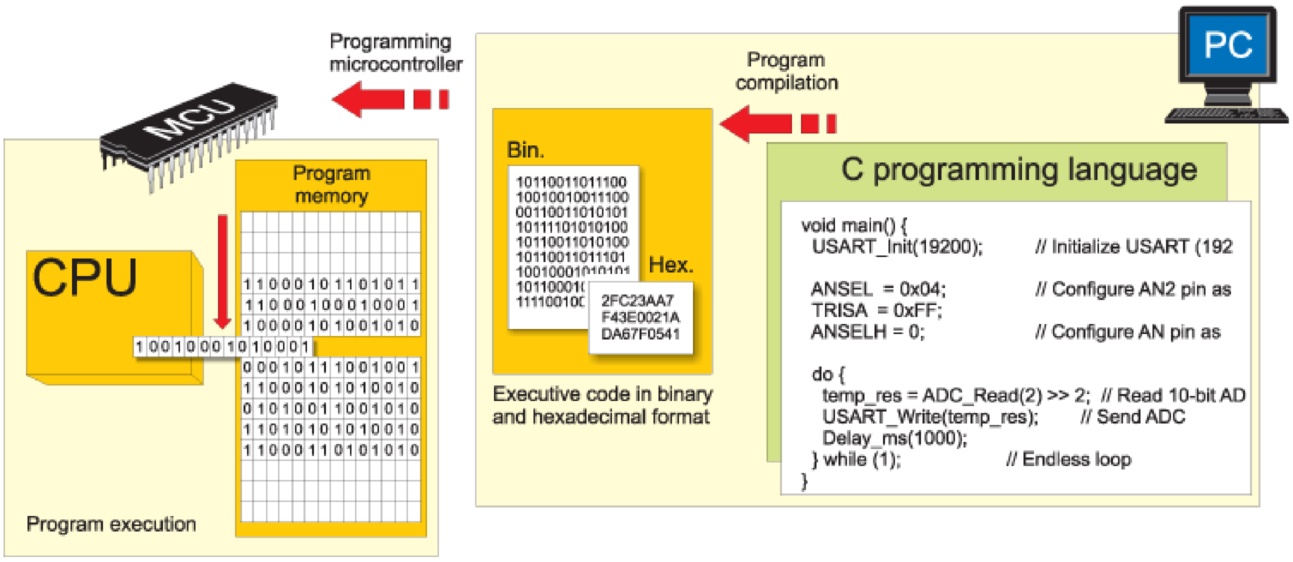
The microcontroller executes the program loaded in its Flash memory. This is the so called executable code comprised of seemingly meaningless sequence of zeros and ones. It is organized in 12-, 14- or 16-bit wide words, depending on the microcontroller's architecture. Every word is considered by the CPU as a command being executed during the operation of the microcontroller. For practical reasons, as it is much easier for us to deal with hexadecimal number system, the executable code is often represented as a sequence of hexadecimal numbers called a Hex code. It used to be written by the programmer. All instructions that the microcontroller can recognize are together called the Instruction set. As for PIC microcontrollers the programming words of which are comprised of 14 bits, the instruction set has 35 different instructions in total.


Programming microcontroller
PC
MCU
Program memory
CPU
Bin.
Hex.
2FC23AA7
F43E0021A
DA87F0541
Executive code in binary and hexadecimal format
Program execution

As the process of writing executable code was endlessly tiring, the first 'higher' programming language called assembly language was created. The truth is that it made the process of programming more complicated, but on the other hand the process of writing program stopped being a nightmare. Instructions in assembly language are represented in the form of meaningful abbreviations, and the process of their compiling into executable code is left over to a special program on a PC called compiler. The main advantage of this programming language is its simplicity, i.e. each program instruction corresponds to one memory location in the microcontroller. It enables a complete control of what is going on within the chip, thus making this language commonly used today.

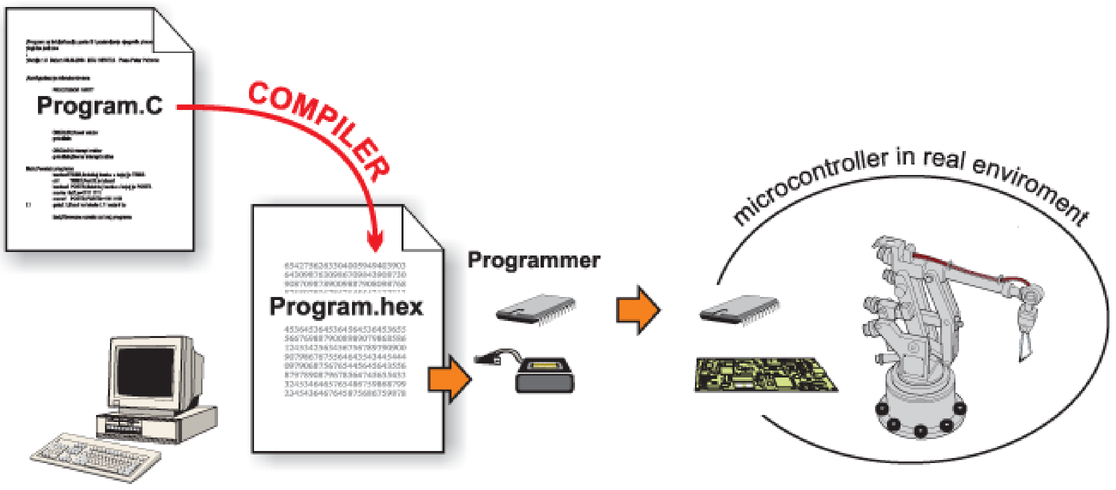


However, programmers have always needed a programming language close to the language being used in everyday life. As a result, the higher programming languages have been created. One of them is C. The main advantage of these languages is simplicity of program writing. It is no longer possible to know exactly how each command executes, but it is no longer of interest anyway. In case it is, a sequence written in assembly language can always be inserted in the program, thus enabling it.



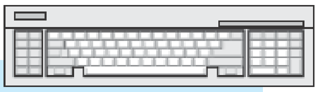
```
void main() {
  USART_Init(19200);          // Initialize USART (192

  ANSEL  = 0x04;              // Configure AN2 pin as
  TRISA  = 0xFF;
  ANSELH = 0;                 // Configure AN pin as

  do {
    temp_res = ADC_Read(2) >> 2;  // Read 10-bit AD
    USART_Write(temp_res);        // Send ADC
    Delay_ms(1000);
  } while (1);                // Endless loop
}
```

Similar to assembly language, a specialized program in a PC called compiler is in charge of compiling program into machine language. Unlike assembly compilers, these create an executable code which is not always the shortest possible.



Figures above give a rough illustration of what is going on during the process of compiling the program from higher to lower programming language.

Here is an example of a simple program written in C language:



```
void main() {

   TRISB = 0;                // All port B pins are configured as
                             // outputs
   PORTB = 0b01010101;       // Logic state on port B pins
}
```

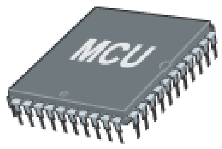
Program written in C

```
; ADDRESS     OPCODE        ASM
; ---------------------------------------------
$0000         $2804         GOTO   _main
$0004         $       _main:
;Test.c,1 :: void main() {
;Test.c,3 :: TRISB = 0;             // All port B pins
$0004         $1303         BCF    STATUS, RP1
$0005         $1683         BSF    STATUS, RP0
$0006         $0186         CLRF   TRISB, 1
;Test.c,4 :: PORTB = 0b01010101; // Logic state
$0007         $3055         MOVLW  85
$0008         $1283         BCF    STATUS, RP0
$0009         $0086         MOVWF  PORTB
;Test.c,5 :: }
$000A         $280A         GOTO   $
```

Compiled Program

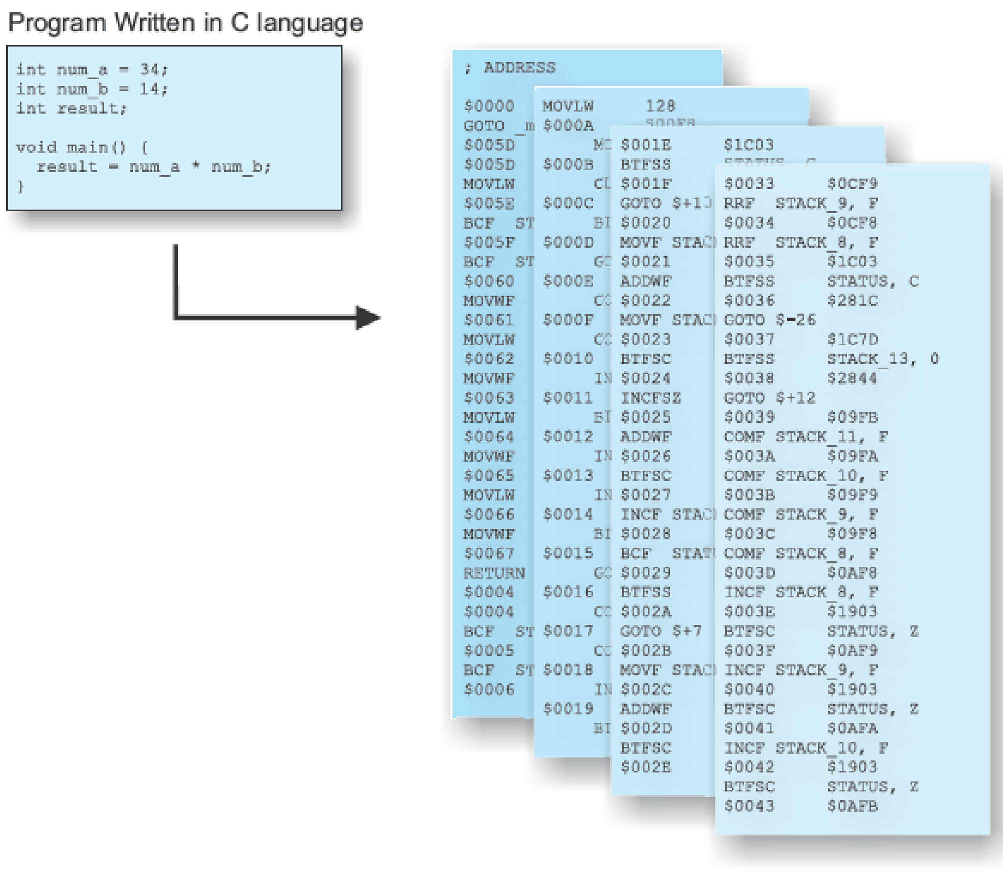
```
:100000000428FF3FFF3FFF3F03138316860155304F
:10001000831286000A28FF3FFF3FFF3FFF3FFF3F5D
:04400E00F22FFFFF8F
:00000001FF
```

Executable Code of the program (HEX code)



ADVANTAGES OF HIGHER PROGRAMMING LANGUAGES

If you have ever written a program for the microcontroller in assembly language, then you probably know that the RISC architecture lacks instructions. For example, there is no appropriate instruction for multiplying two numbers, but there is also no reason to be worried about it. Every problem has a solution and this one makes no exception thanks to mathematics which enable us to perform complex operations by breaking them into a number of simple ones. Concretely, multiplication can be easily substituted by successive addition (a x b = a + a + a + ... + a). And here we are, just at the beginning of a very long story... Don’t worry as far as the higher programming languages, such as C, are concerned because somebody has already solved this and many other similar problems for you. It will do to write a*b.



```c
int num_a = 34;
int num_b = 14;
int result;

void main() {
  result = num_a * num_b;
}
```

```
$0033      $0CF9
RRF  STACK_9, F
$0034      $0CF8
RRF  STACK_8, F
$0035      $1C03
BTFSS      STATUS, C
$0036      $281C
GOTO $-26
$0037      $1C7D
BTFSS      STACK_13, 0
$0038      $2844
GOTO $+12
$0039      $09FB
COMF STACK_11, F
$003A      $09FA
COMF STACK_10, F
$003B      $09F9
COMF STACK_9, F
$003C      $09F8
COMF STACK_8, F
$003D      $0AF8
INCF STACK_8, F
$003E      $1903
BTFSC      STATUS, Z
$003F      $0AF9
INCF STACK_9, F
$0040      $1903
BTFSC      STATUS, Z
$0041      $0AFA
INCF STACK_10, F
$0042      $1903
BTFSC      STATUS, Z
$0043      $0AFB
```

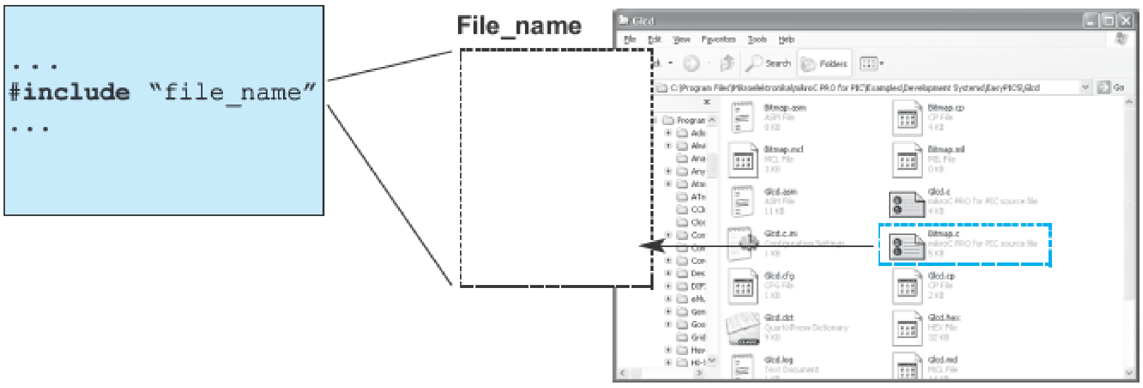
Same program compiled into assembly code

PREPROCESSOR

A preprocessor is an integral part of the C compiler and its function is to recognize and execute preprocessor instructions. These are special instructions which do not belong to C language, but are a part of software package coming with the compiler. Each preprocessor command starts with ‘#’. Prior to program compilation, C compiler activates the preprocessor which goes through the program in search for these signs. If any encountered, the preprocessor will simply replace them by another text which, depending on the type of command, can be a file contents or just a short sequence of characters. Then, the process of compilation may start. The preprocessor instructions can be anywhere in the source program, and refer only to the part of the program following their appearance up to the end of the program.

PREPROCESSOR DIRECTIVE # include

Many programs often repeat the same set of commands for several times. In order to speed up the process of writing a program, these commands and declarations are usually grouped in particular files that can easily be included in the program using this directive. To be more precise, the #include command imports text from another document, no matter what it is (commands, comments etc.), into the program.



PREPROCESSOR DIRECTIVE # define

The *#define* command provides macro expansion by replacing identifiers in the program by their values.

```
#define symbol sequence_of_characters
```

Example:

```
...
#define PI 3.14
...
```

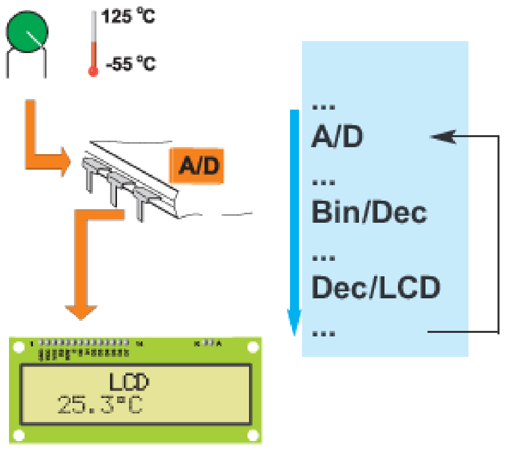
As the use of any language is not limited to books and magazines only, this programming language is not closely related to any special type of computers, processors or operating systems. C language is actually a general-purpose language. However, exactly this fact can cause some problems during operation as C language slightly varies depending on its application (this could be compared to different dialects of one language).

THE BASICS OF C PROGRAMMING LANGUAGE

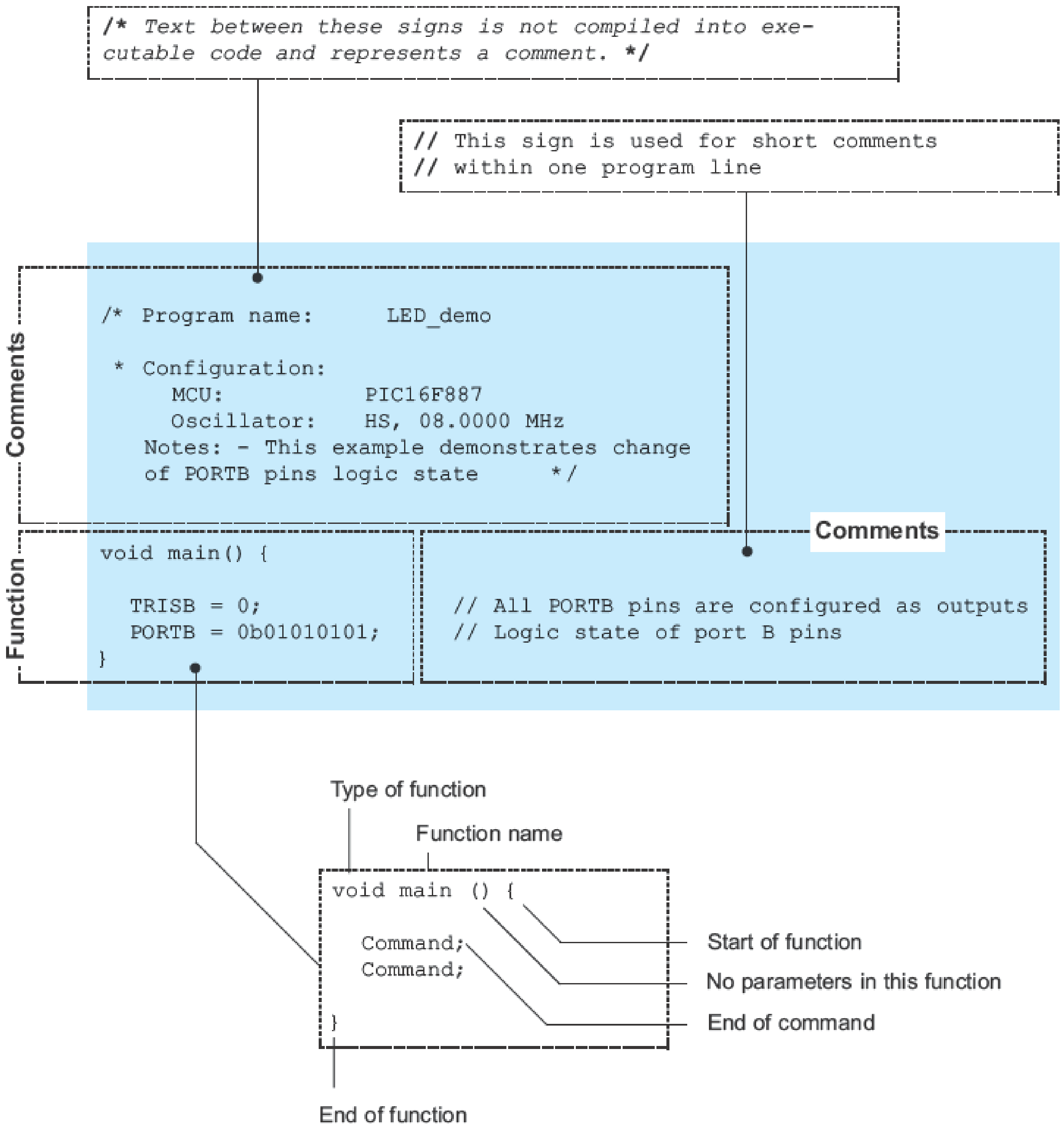
The main idea of writing program in C language is to break a bigger problem down into several smaller pieces. Suppose it is necessary to write a program for the microcontroller that is going to measure temperature and show results on an LCD display. The process of measuring is performed by a sensor that converts temperature into voltage. The microcontroller uses its A/D converter to convert this voltage (analogue value) to a number (digital value) which is then sent to the LCD display via several conductors. Accordingly, the program is divided in four parts that you have to go through as per the following order:

1. Activate and set built-in A/D converter;
2. Measure analogue value;
3. Calculate temperature;
4. Send data in the proper form to LCD display.

As seen, the higher programming languages such as C enable you to solve this problem easily by writing four functions to be executed cyclically and over and over again.



The figure below illustrates the structure of a simple program, pointing out the parts it consists of.



```
/* Text between these signs is not compiled into exe-
cutable code and represents a comment. */
```

```
// This sign is used for short comments
// within one program line
```

```
/* Program name:      LED_demo

 * Configuration:
     MCU:            PIC16F887
     Oscillator:     HS, 08.0000 MHz
   Notes: - This example demonstrates change
   of PORTB pins logic state      */

void main() {

  TRISB = 0;              // All PORTB pins are configured as outputs
  PORTB = 0b01010101;     // Logic state of port B pins
}
```

```
void main () {

   Command;
   Command;

}
```

COMMENTS

Comments are part of the program used to clarify the operation of the program or provide more information about it. Comments are ignored and not compiled into executable code by the compiler. Simply put, the compiler can recognize special characters used to designate where comments start and terminate and completely ignores the text inbetween during compilation. There are two types of such characters. One designates long comments extending several program lines (/* .... */), while the other designates short comments taking up a single line (//). Even though comments cannot affect the program execution, they are as important as any other part of the program, and here is why... A written program can always be improved, modified, upgraded, simplified...It is almost always done. Without comments, trying to understand even the simplest programs is waste of time.

DATA TYPES IN C LANGUAGE

There are several types of data that can be used in C programming language. A table below shows the range of values which these data can have when used in their basic form.

VARIABLES

Any number changing its value during program operation is called a variable. Simply put, if the program adds two numbers (number1 and number2), it is necessary to have a value to represent what we in everyday life call the sum. In this case number1, number2 and sum are variables.

Declaring Variables

- Variable name can include any of the alphabetical characters A-Z (a-z), the digits 0-9 and the underscore character '_'. The compiler is case sensitive and differentiates between capital and small letters. Function and variable names usually contain lower case characters, while constant names contain uppercase characters.
- Variable names must not start with a digit.
- Some of the names cannot be used as variable names as already being used by the compiler itself. Such names are called the *key words.*

Type	Size (bits)	Arithmetic Type
bit	1	unsigned integer
char	8	signed or unsigned integer
unsigned char	8	unsigned integer
short	16	signed integer
unsigned short	16	unsigned integer
int	16	signed integer
unsigned int	16	unsigned integer
short long	24	signed integer
unsigned short long	24	unsigned integer
long	32	signed integer
unsigned long	32	unsigned integer
float	24	real
double	24 or 32	real

Pointers

A pointer is a special type of variable holding the address of character variables. In other words, the pointer ‘points to’ another variable. It is declared as follows:

```
type_of_variable *pointer_name;
```

In order to assign the address of a variable to a pointer, it is necessary to use the '=' character and write variable name preceded by the '&' character. In the following example, the pointer ‘multiplex’ is declared and assigned the address of the first out of eight LED displays:

```
unsigned int *multiplex; // Declare name and type of pointer multiplex
multiplex = &display1;   // Pointer multiplex is assigned the address of
                         // variable display1
```

To change the value of the pointed variable, it is sufficient to write the '*' character in front of its pointer and assign it a new value.

```
*multiplex = 6; // Variable display1 is assigned the number 6
```

Similarly, in order to read the value of the pointed variable, it is sufficient to write:

```
temp = *multiplex; // The value of variable display1 is copied to temp
```

Changing individual bits

There are a few ways to change only one bit of a variable. The simplest one is to specify the register name, bit's position or a name and desired state:

```
#define RELAY RA0

RB3 = 0;      // Clear the bit 3 of PORTB
...
RELAY = 1;    // Set the bit named RELAY
```

Declarations

Every variable must be declared prior to being used for the first time in the program. Since variables are stored in RAM memory, it is necessary to reserve space for them (one, two or more bytes). You know what type of data you write or expect as a result of an operation, while the compiler does not know that. Don’t forget, the program deals with variables to which you assigned the names *gate, sum, minimum* etc. The compiler recognizes them as registers of RAM memory. Variable types are usually assigned at the beginning of the program.

```
unsigned int gate1; // Declare name and type of variable gate1
```

Apart from the name and type, variables are usually assigned initial values at the beginning of the program as well. It is not a ‘must-do’ step, but a matter of good habits. In this case, it looks as follows:

```
unsigned int gate1;     // Declare type and name of the variable
signed int start, sum; // Declare type and name of other two variables
gate1 = 20;             // Assign variable gate1 an initial value
```

The process of assigning initial value and declaring type can be performed in one step:

```
unsigned int gate1=20; // Declare type, name and value of variable
```

If there are several variables being assigned the same initial value, the process can be even simplified:

```
unsigned int gate1=gate2=gate3=20;
signed int start=sm=0;
```

- Type of variable is not accompanied by the ‘+’ or ‘-’ sign by default. For example, *char* can be written instead of *signed char* (variable is a signed byte). In this case the compiler considers variable positive values.
- If you, by any chance, forget to declare variable type, the compiler will automatically consider it a signed integer. It means that such a variable will occupy two memory bytes and have values in the range of -32768 to +32767.

CONSTANTS

A constant is a number or a character having fixed value that cannot be changed during program execution. Unlike variables, constants are stored in the flash program memory of the microcontroller for the purpose of saving valuable space of RAM. The compiler recognizes them by their name and prefix *const*.

INTEGER CONSTANTS

Integer constants can be decimal, hexadecimal, octal or binary. In the following example, the constant MINIMUM will be considered a signed integer and stored within two bytes of Flash memory (16 bits):

Radix	Format	Example
binary	0bnumber or 0Bnumber	0b10011010
octal	0number	0763
decimal	number	129
hexadecimal	0xnumber or 0Xnumber	0x2F

```
const int MINIMUM = -100; // Declare constant MINIMUM
```

FLOATING POINT CONSTANTS

Floating point constants consist of an integer part, a dot, a fractional part and an optional e or E followed by a signed integer exponent.

```
const float T_MAX = 32.60;     // Declare temperature T_MAX
const double T_MAX = 3.260E1; // Declare the same constant T_MAX
```

In both examples, a constant named *T_MAX* is declared to have the fractional value 32.60. It enables the program to compare the measured temperature to the meaningful constant instead of numbers representing it.

CHARACTER CONSTANTS (ASCII CHARACTERS)

A character constant is a character enclosed within single quotation marks. In the following example, a constant named *I_CLASS* is declared as **A** character, while a constant named *II_CLASS* is declared as **B** character.

```
const char I_CLASS = 'A';  // Declare constant I_CLASS
const char II_CLASS = 'B'; // Declare constant II_CLASS
```

When defined this way, the execution of the commands sending the *I_CLASS* and *II_CLASS* constants to an LCD display, will cause the characters **A** and **B** to be displayed, respectively.

STRING CONSTANTS

A constant consisting of a sequence of characters is called a string. String constants are enclosed within double quotation marks.

```
const char Message_1[] = "Press the START button"; // Message 1 for LCD
const char Message_2[] = "Press the RIGHT button"; // Message 2 for LCD
const char Message_3[] = "Press the LEFT button";  // Message 3 for LCD
```

In this example, sending the *Message_1* constant to an LCD display will cause the message *'press the START button'* to be displayed.

ENUMERATED CONSTANTS

Enumerated constants are a special type of integer constants which make a program more comprehensive and easier to follow by assigning elements the ordinal numbers. In the following example, the first element in curly brackets is automatically assigned the value 0, the second one is assigned the value 1, the third one the value 2 etc.

```
enum MOTORS {UP, DOWN, LEFT, RIGHT}; // Declare constant MOTORS
```

On every occurrence of the words *'LEFT'*, *'RIGHT'*, *'UP'* and *'DOWN'* in the program, the compiler will replace them by the appropriate numbers (0-3).

OPERATORS, OPERATIONS AND EXPRESSIONS

An operator is a symbol denoting particular arithmetic, logic or some other operation. There are more than 40 operations available in C language, but at most 10-15 of them are used in practice. Every operation is performed upon one or more operands which can be variables or constants. Besides, every operation features priority execution and associativity as well.

ARITHMETIC OPERATORS

Arithmetic operators are used in arithmetic operations and always return positive results. Unlike unary operations being performed upon one operand, binary operations are performed upon two operands. In other words, two numbers are required to execute a binary operation. For example: a+b or a/b.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Reminder

ASSIGNMENT OPERATORS

There are two types of assignments in C language:

- Simple operators assign values to variables using the common '=' character. For example: a = 8
- Compound assignments are specific to C language and consist of two characters as shown in the table. An expression can be written in a different way as well, but this one provides more efficient machine code.

Operator	Example	
	Expression	Equivalent
+=	a += 8	a = a + 8
-=	a -= 8	a = a - 8
*=	a *= 8	a = a * 8
/=	a /= 8	a = a / 8
%=	a %= 8	a = a % 8

INCREMENT AND DECREMENT OPERATORS

Increment and decrement by 1 operations are denoted by '++' and '--'. These characters can either precede or follow a variable. In the first case (++x), the x variable will be first incremented by 1, then used in expression. Otherwise, the variable will be first used in expression, then incremented by 1. The same applies to the decrement operation.

Operator	Example	Description
++	++a	Variable "a" is incremented by 1
	a++	
--	--b	Variable "b" is decremented by 1
	b--	

RELATIONAL OPERATORS

Relational operators are used in comparisons for the purpose of comparing two variables which can be integers (int) or floating point numbers (float). If an expression evaluates to true, a 1 is returned. Otherwise, a 0 is returned. This is used in expressions such as ‘if the expression is true then...’

Operator	Meaning	Example	Truth condition
>	is greater than	b > a	if **b** is greater than **a**
>=	is greater than or equal to	a >= 5	If **a** is greater than or equal to **5**
<	is less than	a < b	if **a** Is less than **b**
<=	is less than or equal to	a <= b	if **a** Is less than or equal to **b**
==	is equal to	a == 6	if **a** Is equal to **6**
!=	is not equal to	a != b	if **a** Is not equal to **b**

LOGIC OPERATORS

There are three types of logic operations in C language: logic AND, logic OR and negation (NOT). For the sake of clearness, logic states in tables below are represented as logic zero (0=false) and logic one (1=true). Logic operators return true (logic 1) if the expression evaluates to non-zero, and false (logic 0) if the expression evaluates to zero. This is very important because logic operations are commonly used upon expressions, not upon single variables (numbers) in the program. Therefore, logic operations refer to the truth of the whole expression.

For example: `1 && 0` is the same as `(true expression) && (false expression)`

The result is 0, i.e. - *False* in either case.

Operator	Logical AND		
	Operand1	Operand2	Result
&&	0	0	0
	0	1	0
	1	0	0
	1	1	1

Operator	Logical OR		
	Operand1	Operand2	Result
||	0	0	0
	0	1	1
	1	0	1
	1	1	1

Operator	Logical NOT	
	Operand1	Result
!	0	1
	1	0

BITWISE OPERATORS

Unlike logic operations being performed upon variables, the bitwise operations are performed upon single bits within operands. Bitwise operators are used to modify the bits of a variable. They are listed in the table below:

Operand	Meaning	Example	Result	
~	Bitwise complement	a = ~b	b = 5	a = -5
<<	Shift left	a = b << 2	b = 11110011	a = 11001100
>>	Shift right	a = b >> 3	b = 11110011	a = 00011110

&	Bitwise AND	c = a & b	a = 11100011
			b = 11001100
				c = 11000000

|	Bitwise OR	c = a | b	a = 11100011
			b = 11001100
				c = 11101111

^	Bitwise EXOR	c = a ^ b	a = 11100011
			b = 11001100
				c = 00101111

HOW TO USE OPERATORS?

Except for assignment operators, two operators must not be written next to each other.

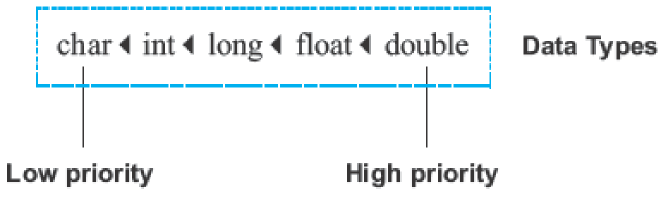
```
x*%12; // such expression will generate an error
```

Operators are grouped together using parentheses similar to arithmetic expressions. The expressions enclosed within parentheses are calculated first. If necessary, multiple (nested) parentheses can be used. Each operator has its priority and associativity as shown in the table.

Priority	Operators	Associativity
High	() [] -> .	from left to right
	! ~ ++ -- +(unary) -(unary) *Pointer &Pointer	from right to left
	* / %	from left to right
	+ -	from left to right
	< >	from left to right
	< <= > >=	from left to right
	== !=	from left to right
	&	from left to right
	^	from left to right
	|	from left to right
	&&	from left to right
	||	from right to left
	?:	from right to left
Low	= += -= *= /= /= &= ^= |= <= >=	from left to right

DATA TYPE CONVERSION

The main data types are put in hierarchical order as follows:



If two operands of different type are used in an arithmetic operation, the lower priority operand type is automatically converted into the higher priority operand type. In expressions free from assignment operation, the result is obtained in the following way:

- If the highest priority operand is of type *double*, then types of all other operands in the expression as well as the result are automatically converted into type *double*.
- If the highest priority operand is of type *long*, then types of all other operands in the expression as well as the result are automatically converted into type *long*.
- If the operands are of *long* or *char* type, then types of all other operands in the expression as well as the result are automatically converted into type *int*.

Auto conversion is also performed in assignment operations. The result of the expression right from the assignment operator is always converted into the type of variable left from the operator. If the result is of higher-ranked type, it is truncated or rounded in order to match the type of variable. When converting real data into integer, numbers following the decimal point are always truncated.

```
int x;      // Variable x is declared as integer int
x = 3;      // Variable x is assigned value 3
x += 3.14; // Number PI (3.14) is added to variable x by performing
            // the assignment operation

/* The result of addition is 6 instead of expected 6.14. To obtain the
expected result without truncating the numbers following the decimal
point, common addition should be performed (x+3.14), . */
```

CONDITIONAL OPERATORS

A condition is a common ingredient of the program. When met, it is necessary to perform one out of several operations. In other words *'If the condition is met (...), do (...). Otherwise, if the condition is not met, do (...)'*. Conditional operands **if-else** and **switch** are used in conditional operations.

CONDITIONAL OPERATOR if-else

The conditional operator can appear in two forms - as **if** and **if-else** operator. Here is an example of the **if** operator:

```
if(expression) operation;
```

If the result of *expression* enclosed within brackets is not 0 (true), the *operation* is performed and the program proceeds with execution. If the result of *expression* is 0 (false), the *operation* is not performed and the program immediately proceeds with execution.

As mentioned, the other form combines both **if** and **else** operators:

```
if(expression) operation1; else operation2;
```

If the result of *expression* is not 0 (true), *operation1* is performed, otherwise *operation2* is performed. After performing either operation, the program proceeds with execution.

The syntax of the **if-else** statement is:

```
if(expression)
    operation1;
else
    operation2;
```

If either *operation1* or *operation2* is compound, a group of operations these consist of must be enclosed within curly brackets. For example:

```
if(expression) {
    operation1;
    operation2;
}
else {
    operation3;
    operation4;
}
```

The **if-else** operator can be written using the conditional operator '?:' as in example below:

```
(expression1)? expression2 : expression3
```

If *expression1* is not 0 (true), the result of the whole expression will be equal to the result obtained from *expression2*. Otherwise, if *expression1* is 0 (false), the result of the whole expression will be equal to the result obtained from *expression3*.

```
maximum = (a > b)? a : b // Variable maximum is assigned the value of
                         // larger variable (a or b)
```

Switch OPERATION

Unlike the **if-else** statement which makes selection between two options in the program, the **switch** operator enables you to choose between several operations. The syntax of the **switch** statement is:

```
switch (selector) {         // Selector is of char or int type
    case constant1:
        operation1;         // Group of operators are executed if
        ...                 // selector and constant1 are equal
        break;
    case constant2:
        operation2;         // Group of operators are executed if
        ...                 // selector and constant2 are equal
        break;
        ...
    default:
        expected_operation;// Group of operators are executed if no
        ...                 // constant is equal to selector
        break;
}
```

The **switch** operation is executed in the following way: *selector* is executed first and compared to *constant1*. If match is found, statements in that case block are executed until the *break* keyword or the end of the **switch** operation is encountered. If no match is found, *selector* is further compared to *constant2* and if match is found, statements in that case block are executed until the *break* keyword is encountered and so on. If the selector doesn’t match any constant, operations following the **default** operator are to be executed.

PROGRAM LOOP

It is often necessary to repeat a certain operation for a couple of times in the program. A set of commands being repeated is called the program loop. How many times it will be executed, i.e. how long the program will stay in the loop, depends on the conditions to leave the loop.

While LOOP

The **while** loop looks as follows:

```
while(expression){
    commands;
    ...
}
```

The *commands* are executed repeatedly (the program remains in the loop) until the *expression* becomes false. If the *expression* is false on entry to the loop, then the loop will not be executed and the program will proceed from the end of the **while** loop.

A special type of program loop is the *endless loop*. It is formed if the condition remains unchanged within the loop. The execution is simple in this case as the result in brackets is always true (1=1), which means that the program remans in the same loop:

```
while(1){
    ... // Expressions enclosed within curly brackets will be
    ... // endlessly executed (endless loop).
}
```

For LOOP

The **for** loop looks as follows:

```
for(initial_expression; condition_expression; change_expression) {
    operation;
    ...
}
```

The execution of such program sequence is similar to the **while** loop, except that in this case the process of setting initial value (initialization) is performed within declaration. The *initial_expression* sets the initial variable of the loop, which is further compared to the *condition_expression* before entering the loop. *Operations* within the loop are executed repeatedly and after each iteration the value of expression is changed. The iteration continues until the *condition_expression* becomes false.

```
for(k=1; k<5; k++){ // Increase variable k 5 times (from 1 to 5) and
    operation;      // repeat expression operation every time
    ...
}
```

Operation is to be performed five times. After that, it will be validated by checking that the expression k<5 is false (after 5 iterations k=5) and the program will exit the **for** loop.

Do-while LOOP

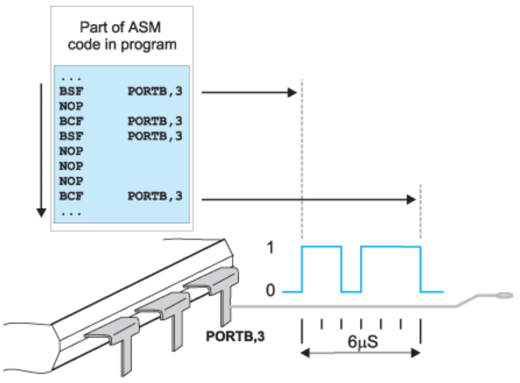
The *do-while* loop looks as follows:

```
do {
    operation;
    ...
} while (check_condition);
```

In this case, the *operation* is executed at least once regardless of whether the condition is true or false as the expression *check_condition* is executed at the end of the loop. If the result is not 0 (true), the procedure is repeated. In the following example, the program remains in **do-while** loop until the variable a reaches 10 (a million iterations).

```
a = 0;                  // Set initial value

do {

    a = a+1;            // Operation in progress

} while (a <= 10);      // Check condition
```

WRITING CODE IN ASSEMBLY LANGUAGE



PIC MCU instructions may also be directly embedded “in-line” into C code using the directives #asm, #endasm or the statement asm();.
The #asm and #endasm directives are used to start and end a block of assembly instructions which are to be embedded into the assembly output of the code generator. The #asm block is not syntactically part of the C program, and thus it does not obey normal C flow-of-control rules. This means that you should not use this form of in-line assembly inside C constructs like if(), while() and for() statements. However this is the easiest means of adding multiple assembly instructions.
The asm() statement is used to, typically, embed a single assembler instruction. This form looks and behaves like a C statement. Only one assembly instruction may be encapsulated within each asm() statement. You can specify more than one assembly instruction inside one asm() statement by separating the instructions with a \n character, (e.g. asm("movlw 55\nmovwf _x");) although code will be more readable if you one place one instruction in each statement and use multiple statements.

You may use the asm(" ") form of in-line assembly at any point in the C source code as it will correctly interact with all C flow-of-control structures.

The following example shows both methods used:

```
unsigned int var;

void main(void){

    var = 1;

#asm         // like this...
    BCF 0,3
    BANKSEL(_var)
    RLF (_var)&07fh
    RLF (_var+1)&07fh
#endasm
             // do it again the other way...
    asm("BCF 0,3" );
    asm("BANKSEL _fvar");
    asm("RLF (_var)&07fh" );
    asm("RLF (_var+1)&07fh" );
}
```

ARRAYS

A group of variables of the same type is called an array. Elements of an array are called components, while their type is called the main type. An array is declared by specifying its name, type and the number of elements it will comprise:

```
component_type array_name [number_of_components];
```

Such a complicated definition for something so simple, isn’t it? An array can be thought of as a shorter or longer list of variables of the same type where each of these is assigned an ordinal number (numbering always starts at zero). Such an array is often called a vector. The figure below shows an array named *shelf* which consists of 100 elements.

Array "shelf"	Elements of array	Contents of element
7	shelf[0]	7
23	shelf[1]	23
34	shelf[2]	34
0	shelf[3]	0
0	shelf[4]	0
12	shelf[5]	12
9	shelf[6]	9
...	...	...
...	...	...
23	shelf [99]	23

In this case, the contents of a variable (an element of the array) represents a number of products the shelf contains. Elements are accessed by indexing, i.e. by specifying their ordinal number (index):

```
shelf[4] = 12;     // 12 items is ‘placed’ on shelf [4]
temp = shelf [1]; // Variable shelf[1] is copied to
                   // variable temp
```

Elements can be assigned contents during array declaration. In the following example, the array named calendar is declared and each element is assigned specific number of days:

```
unsigned char calendar [12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

TWO-DIMENSIONAL ARRAY

Apart from one-dimensional arrays which could be thought of as a list, there are also multidimensional arrays in C language. In a few following sentences we are going to describe only two-dimensional arrays called *matrices* which can be thought of as tables. A twodimensional array is declared by specifying data type of the array, the array name and the size of each dimension. Look at the example below:

```
component_type array_name [number_of_rows] [number_of_columns];
```

number_of_rows and *number_of_columns* represent the number of rows and columns of a table, respectively.

```
int Table [3][4]; // Table is defined to have 3 rows and 4 columns
```

This array can be represented in the form of a table.

table[0][0]	table[0][1]	table[0][2]	table[0][3]
table[1][0]	table[1][1]	table[1][2]	table[1][3]
table[2][0]	table[2][1]	table[2][2]	table[2][3]

Similar to vectors, the elements of a matrix can be assigned values during array declaration. In the following example, the elements of the two-dimensional array **Table** are assigned values. As seen, this array has two rows and three columns:

```
int Table[2][3] = {{3,42,1}, {7,7,19}};
```

The matrix above can also be represented in the form of a table the elements of which have the following values:

3	42	1
7	7	19

POINTERS

A pointer is a special type of variable holding the address of character variables. In other words, the pointer ‘points to’ another variable. It is declared as follows:

This is how a function looks like:

```
type_of_variable *pointer_name;
```

In order to assign the address of a variable to a pointer, it is necessary to use the '=' character and write variable name preceded by the '&' character. In the following example, the pointer ‘multiplex’ is declared and assigned the address of the first out of eight LED displays:

```
unsigned int *multiplex; // Declare name and type of pointer multiple
multiplex = &display1;    // Pointer multiplex is assigned the address of
                         // variable display1
```

To change the value of the pointed variable, it is sufficient to write the '*' character in front of its pointer and assign it a new value.

```
*multiplex = 6; // Variable display1 is assigned the number
```

Similarly, in order to read the value of the pointed variable, it is sufficient to write:

```
temp = *multiplex; // The value of variable display1 is copied to temp
```

FUNCTIONS

Every program written in C language consists of larger or smaller number of functions. The main idea is to divide a program into several parts using these functions in order to solve the actual problem easier. Besides, functions enable us to use the skills and knowledge of other programmers. For example, if it is necessary to send a string to an LCD display, it is much easier to use already written part of the program than to start over.

Functions consist of commands specifying what should be done upon variables. They can be compared to subroutines. As a rule, it is much better to have a program consisting of large number of simple functions than of a few large functions. A function body usually consists of several commands being executed by the order they are specified.

Every function must be properly declared so as to be properly interpreted during the process of compilation. Declaration contains the following elements:

- Function name
- Function body
- List of parameters
- Declaration of parameters
- Type of function result

This is how a function looks like:

```
type_of_result function_name (type argument1, type argument2,...) {
    Command;
    Command;
    ...
}
```

Example:

```
/* Function computes the result of division of the numerator number by
   the denominator denom.
   The function returns a real. */

real div(int number, int denom);
```

Note that a function does not need to have parameters, but must have brackets to be used for entering them. Otherwise, the compiler would misinterpret the function.

If the function, after being executed, returns no result to the main program or to the function it is called by, the program proceeds with execution after encountering a closing curly bracket. Such functions are used when it is necessary to change the state of the microcontroller output pins, during data transfer via serial communication, when writing data on an LCD display etc. The compiler recognizes those functions by the type of their result specified to be **void**.

```
void function_name (type argument1, type argument2,...) {
    Command;
}
```

Example:

```
void interrupt() {
    cnt++;             // Interrupt causes cnt to be incremented by 1
    T0IF = 0;          // Reset Timer0 Flag
}
```

The function can be assigned an arbitrary name. The only exception is the name **main** which has a special purpose. Namely, the program always starts execution with this function. It means that every program written in C language must contain one function named 'main' which does not have to be placed at the beginning of the program.

If it is necessary that called function returns results after being executed, the return command, which can be followed by any expression, is used:

```
type_of_result function_name (type argument1, type argument2,...) {
    Command;
    ...
    return expression;
}
```

If the function contains the **return** command without being followed by *expression*, the function stops its execution when encounters this command and the program proceeds with execution from the first command following a closing curly bracket.

DECLARATION OF A NEW FUNCTION

Apart from the functions that C language 'automatically' recognizes, there are also completely new functions being often used in programs. Each 'non-standard' function should be declared at the beginning of the program. The function declaration is called a prototype and looks as follows:

```
type_of_result function_name (type1, type2, ..) ;
```

FUNCTION LIBRARIES

Names of all functions being used in C language are stored in the file called header. These functions are, depending on their purpose, sorted in smaller files called libraries. Prior to using any of them in the program, it is necessary to specify the appropriate header file using the **#include** command at the beginning of the program. If the compiler encounters an unknown function during program execution, it will first look for its declaration in the specified libraries.