Engineering program development 4

Edited by Péter Vass

Lexical elements of C programming language

The lexical elements of C programming language can be divided into the following groups:

- identifiers,
- keywords,
- constants,
- comments,
- operators,
- punctuators.

Lexical elements of C programming language

Identifiers

Identifiers are tokens (also called symbols) which name some entities of the programming languages (e.g. variables, types, labels, functions).
In fact, identifiers provide individual names for the above-mentioned entities by means we can refer for them in the source code.
An identifier may consists of letters, decimal digits and underscore characters.
Identifiers may begin with only letters or an underscore.
Suggestion: try to avoid using identifiers which begin with an underscore character.
The shortest identifier is a single letter.
The lower case and upper case forms of a letter count as two different symbols in an identifier (e.g. *variable_name* and *Variable_name* are two different identifiers).

Lexical elements of C programming language

Keywords

Similarly to other programming languages, there are character sequences with special meaning in C.
These words called keywords cannot use as identifiers because they are reserved for the language.
Keywords are always begin with a lower case and they are defined by the standard of the language.
Among other things, the words represent the types of data (e.g. int, long, float, double, char) and the control structures (e.g. if, else, do, while, for) belong to the group of keywords.

Lexical elements of C programming language

Constants

There are two main groups of constants in C:

- numerical constants (numbers),
- character constants (a single character or a sequence of characters).

Numerical constant

Depending on the properties of numbers and the formats by which they can be represented in computers, different types of numerical constants may be applied in C.

Two main types of them:

- integer constant,
- floating-point constant.

Lexical elements of C programming language

Integer constant

It is a sequence of numerals without a decimal point.

An integer constant may be specified in decimal, octal or hexadecimal number systems.

A decimal integer constant is an integer without leading zero (e.g. 2017)

An octal integer constant is an integer with leading zero (e.g. 03741).

A hexadecimal integer constant is an integer with a leading 0x or 0X (e.g. 0x7e1).

Lexical elements of C programming language

Integer constant

In the hexadecimal number system, the first five letters of the alphabet (a,b,c,d,e or A,B,C,D,E) are used for representing the numbers after 9.

Unsigned integer constants

An integer which is not negative may be stored in the format of unsigned integer constant.

In this case, the information about the sign is not stored in the memory.

An unsigned integer constant is written with a terminal u or U (e.g. 2017u, 03741u, 0X7e1U).

Lexical elements of C programming language

Representation of integer constants with larger range

The storage of a large integer constant in the memory requires more bytes than the default number of bytes used for integer constants.

In order to save the memory, we should store only the really large numbers with larger memory size.
The so-called long integer format may be specified by a terminal l or L (e.g. -20578962L, 0116401222L).

Unsigned long interger constant

It is written with suffix ul or UL (e.g. 56045803496ul).

Lexical elements of C programming language

Floating-point constant

It is a number which contains a decimal point (e.g. 0.589).
Numbers like this are stored in a floating-point format.
The floating-point number format is based on the normal form of numbers (e.g. $-1.58 \cdot 10^{25}$).

Data needed for storing the normal form of a number:

- the integer part of coefficient,
- the fraction of coefficient,
- the power of then (the exponent).

The ways of specifying a floating point constant in a C source code:

- in decimal form with integer and fraction parts (e.g. -25.302),
- in normal form (e.g. -2.5302e2 or -2.5302E2).

Lexical elements of C programming language

Floating-point constant

There are three format for storing floating-point constants in the memory.

These formats differ in the precision and range of number representation:

- single precision floating-point format (float),
- double precision floating-point format (double),
- long double precision floating-point format (long double).

The required memory size (in bytes) of these formats can be different but it depends on the hardware architecture and the C compiler.

The default precision of a floating-point constant is the double.

If we want to specify other precision for a constant, a terminal symbol has to be applied. (f or F means the single precision and l or L indicates the long double precision)

E.g. 5.785f, 5.785e-20l

Lexical elements of C programming language

Character constants

A single character constant

It is a character within single quotes, such as 'z'.
It may be a lower or upper case, a digit and other printable symbols allowed by the standard.
A single character constant has a positive integer value which corresponds to its numeric value in the applied character set (e.g '3' is 51 in ASCII character set)
Certain characters are represented by their escape sequences.
An escape sequence always starts with a slash (\).

Lexical elements of C programming language

Some frequently used escape sequences:

\n	new line
\t	horizontal tabulator
\v	vertical tabulator
\a	bell
\\	backslash
\’	single quote
\?	question mark
\”	double quote

Lexical elements of C programming language

String constant or string literal

A string constant is a sequence of characters surrounded by double quotes.
Also escape sequences may be used in a string constant.
Each string constant is automatically terminated with a null character \0 by the compiler
(e.g. ”Speech is the shadow of work.”,
”apple\tpear\tcherry”, ”apple\npear\ncherry\n”).

A very long string in the source code may be split into several lines by means of the slash character:
e.g. ”A peasant, a donkey and a horse \
are going to the market.”

Lexical elements of C programming language

Comments

We may write lines in the source code which contain information about the program or some part of the source code to make the program easier to understand.

The characters typed between /* and */ are ignored by the compiler.

A comment may be split into several lines.

Lexical elements of C programming language

Operators

Operators are symbols by which different types of operations can be specified (e.g. +,-,*,/).
The objects of the operators are the so-called operands (e.g. constants and variables).
Each operator has its own effect on the operands.
The operators and their operands form expressions in the source code.
The result of an expression depends on the values of the operands, the types of operators and the order of the operations.
There some operators which requires only one operands (unary operators), but most of them work with two operands (binary operators).
The operators are usually grouped by their functions.

Lexical elements of C programming language

Most frequently used operators

Arithmetic operators:

+	addition,
-	subtraction,
*	multiplication,
/	division,
%	modulus (it gives the remainder of an integer division)

Assignment operators:

=	single assignment	z = x + y	
+=	add and assignment	y += x	(y = y + x)
-=	subtract and assignment	y -= x	(y = y – x)
*=	multiply and assignment	y *= x	(y = y * x)
/=	divide and assignment	y /= x	(y = y / x)

Lexical elements of C programming language

Most frequently used operators

Relational operators:

<	>	less than	greater than
<=	>=	less than or equal	greater than or equal
==		identical	
!=		not identical	

All of them are binary operators and the result of a relational operation may be 1 (if it is true) or 0 (if it is false).

Example:

```
x = 2.6;
y = 5;
z = (x == y);
printf("The value of z: %d\n", z);
z = (x != y);
printf("The value of z: %d\n", z);
```

Lexical elements of C programming language

Logical operators

&& logical AND (binary),
|| logical OR (binary)
! logical NOT (unary) reverse the logical state

The logical operators requires operands with logical true (1) or false (0) values.
The result of a logical operation may be 1 (if it is true) or 0 (if it is false).

Truth table of logical AND

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

Lexical elements of C programming language

Logical operators

Truth table of logical OR

a	b	a || b
0	0	0
0	1	1
1	0	1
1	1	1

Truth table of logical NOT

a	!a
0	1
1	0

Lexical elements of C programming language

Increment and decrement operators

Both of them are unary operators.

++ increment operator

-- decrement operator

They may be applied before an operand as a prefix or after an operand as a postfix.

The result of these two way of application are different when the increment or decrement operation is included in an expression which contains an other operation using the result of increment or decrement.

In the case of using these operators as a prefix, at first the value of the operand is incremented or decremented then the new value will be used in the other operation.

On the contrary, the operand participates in the other operation (not increment or decrement) with its original value then its value will be incremented or decremented.

Lexical elements of C programming language

Increment and decrement operators

Example for the effect of a post-increment operator:

```
a = 0;
b = 0;
if (a || b++)
    printf("a OR b is true");
else
    printf("a OR b is false");
printf("The value of b: %d", b);
```

Lexical elements of C programming language

Increment and decrement operators

Example for the effect of a pre-increment operator:

```
a = 0;
b = 0;
if (a || ++b)
    printf("a OR b is true");
else
    printf("a OR b is false");
printf("The value of b: %d", b);
```

Lexical elements of C programming language

Precedence of operators

An expression may contain several operators with their operands.

But not all the operators are on the same level.

If an operator has a higher precedence than another, its operation has to be evaluated sooner (independently of its position in the expression).

For example, the multiplication always precede the addition.

x = y + 5 * x;

The rules of precedence determine which level of the hierarchy an operator belongs to.

Lexical elements of C programming language

Associativity of operators

If more than one operators of same precedence are found in an expression, their positions in the expression will determine the evaluation order of the operations.

The direction of the evaluation depends on the precedence level.

On some levels the direction is defined from left to right while on the other levels the direction is reversed.

The associativity of operators indicates the right direction of the evaluation order for each precedence level.

Complete tables of precedence and associativity can be found in books and on the Web.

e.g. Brian W. Kernigham, Dennis M Ritchie: C programming languages

Lexical elements of C programming language

The precedence and associativity of operators listed before

The level of precedence decreases downwards.

!	++	--	- (sign op.)		right to left
*	/	%			left to right
+	- (subtraction op.)				left to right
<	>	<=	>=		left to right
==	!=				left to right
&&					left to right
||					left to right
=	+=	-=	*=	/=	right to left

By means of applying round brackets the order of operations may be modified because the evaluation of an expression between brackets always precedes the external operations.

If you are not sure of the precedence and associativity, apply round brackets in your expression to provide the required order of evaluation.

Lexical elements of C programming language

Punctuators

The most frequent symbols used to satisfy the rules of syntax:

( ) parentheses operators (used for changing the precedence of operations, defining conditional expressions, delimiting the list of parameters of a function),

[ ] array subscript operator used for indicating the size of an array or the index of an element of an array (array: one, two or more dimensional dataset of sequential elements with identical data type) ,

{ } braces indicate the start and end of blocks of instructions (in the definitions of functions and control structures),

, comma separates the elements in a list of parameters of a function (function argument list),

; semicolon is a statement terminator,

the initial symbol of a preprocessing directive (e.g. include, define etc.)

Structure of C programs

C programming language supports the so-called function-oriented software design.

In this design, the solution of a complex problem is constructed from the solutions of relatively simple problems. The solutions of these partial problems are implemented in the form of independent, computational units called functions (or subroutines).

Each function is capable of solving some part of the overall problem. But the computational service of a function may be used in the solution of another complex problem. So, there is a possibility of reemploying any function in other programs.

The implementation of a function in a C source code is called definition.

The computational service of a function can be utilized by calling the function in the main program (or main function).

Structure of C programs

The source code of a relatively simple C program is stored in a single file with extension .c (source file).
A program like that is called single-module program.
In the case of a multi-module program, several source files contain the source code of the program.
Independently of the number of modules, a C program may contain the definitions of several functions but a single main function may be defined in it.

Structure of C programs

The general structure of a single-module C program (the source code is stored in a single file of extension .c):

1. preprocessing directives (e.g. #include, #define etc.)
2. declarations of global variables (they can be used within the whole module)
3. declarations of user-defined functions
4. definition of the main function
   declarations of local variables (their use is limited to the main function)
   instructions, control structures
5. definitions of user-defined functions (non-library functions) called inside the main function.

Structure of C programs

Example for the general structure of a single-module C program :

```
#include <stdio.h>                  /*a directive for the preprocessing phase*/

float sum(float, float);            /*declaration of a user-defined function*/
float avg();                        /*declaration of a user-defined function*/
float z;                            /*declaration of a global variable*/

main(){
        float x, y;

        printf("Type two numbers separated by a space character: \n");
        scanf("%f%f", &x, &y);
        z = sum(x, y);
        printf(„The average of the numbers: %f", avg());
}

float sum(float a, float b){        /*definition of a user-defined function*/
        float r;

        r = a + b;
        return r;                   /*return statement assigns an output value to the left side of
                                    the instruction calling the function*/
}
float avg(){                        /*definition of a user-defined function*/

        return z/2;
}
```

Data types in C programming language

C programming language, similarly to other programming languages, defines several basic data types. Each data type has its own properties (e.g. precision and interval of number representation, required space in the memory, the group of operations which may be applied to the given type of data etc.)

In order to give the required information about a variables or function to the compiler, we have to declare it.

The declaration of a variable or function has to be done before we should use it in the source code of a program.

Although a declaration provides all the necessary information for the compiler it does not necessarily result in the memory reservation for the given variable or function.

The memory reservation occurs by the effect of the definition.

Each variable or function is needed to define once only in the source code.

A definition is also a declaration if the variable or function has not been declared in the source code.

Data types in C programming language

The most frequently used basic types in C programming language.

char	a single byte used for holding one character in the local character set,
int	the basic type of an integer,
float	the basic type of a number represented by a single precision floating point format,
double	the basic type of a number represented by a double precision floating point format.

There are three additional, basic types in C: **enum** (enumerated type), **struct** (structure type) and **union** (union type).

Void is a special type which indicates the lack of any type and it is used in special cases (e.g. when we want to define a function without an output parameter).

Data types in C programming language

The most frequently used basic types in C programming language.

In addition, different type modifiers may be applied to the basic type specifiers. Thus, different variants of the basic types may be created.

signed char	signed character type
unsigned char	unsigned character type
long int	integer type for large numbers
short int	integer type for smaller numbers
unsigned int	unsigned integer type
unsigned long int	unsigned integer type for large numbers
unsigned short int	unsigned integer type for smaller numbers
long double	high precision double type

The memory space reserved for a variable of a given type depends on the hardware architecture of the computer and the C compiler.

Data types in C programming language

The declaration of variables with basic types

A variable is actually associated with a space in the memory and is identified by a name (identifier) in the source code. The actual value of the variable is stored in that memory space. The value of the variable may change during the execution of a program.

We have to declare each variable in the source code before its first use in any assignment or expression.

In a single-module C program, the declaration of a variable is simultaneously the definition of the variable.

It means that the declaration entails the reservation of required space in the memory.

Data types in C programming language

The declaration of variables with basic types

A declaration statement for a variable always starts with the specification of the type. Then the identifier of the variable follows. The statement always terminates in a semicolon. Example:

```
int var1;
```

We may declare more than one variable with identical type in a single statement. The identifiers of the variables must be separated by comas. Example:

```
double var1, var2, var3;
```

We may also assign initial values (constants) to some of the variables. Examples:

```
float szigma=2.895;
double tau=-1.98, rho;
```

Of course, the initialization of a variable may be performed later in the source code.

Data types in C programming language

Example for displaying the memory reservation of variables with differenet data types

```
#include <stdio.h>
/*sizeof is a unary operator which gives the memory space
reserved for its operand */

main( ){
      char a;
      int b;
      float c;
      double d;

      printf("The memory reservation of a char variable in bytes: %d\n", sizeof(a));
      printf("The memory reservation of an int variable in bytes: %d\n", sizeof(b));
      printf(" The memory reservation of a float variable in bytes: %d\n", sizeof(c));
      printf(" The memory reservation of a double variable in bytes: %d\n", sizeof(d));
}
```

Using of constants

The two, most frequently applied ways of using constants in C programs:

- declaring a variable with a const keyword,
- defining a symbolic constant in a preprocessing directive.

The keyword **const** is a type qualifier which precedes the type specifier and modifier in a declaration statement.

If we apply the const qualifier, we are obligated to assign a value to the variable in the declaration statement. Example.

```
const int x = 10;
```

The value of a variable qualified by const cannot be modified directly in the source code. It keeps its initial value in the program.

We may create a so-called symbolic constant by means of a #define directive. Example

```
#define PI 3.14159
```

In this case, the name of the symbolic constant is PI and its value is 3.14159.

Using of constants

After we have defined a symbolic constant, we may use its name in the expressions of our source code.
The name of a symbolic constant represents its value in any expression.
In the preprocessing phase of the compilation, the compiler substitutes the value of the symbolic constant for each occurrence of the name in the source code.

Using of constants

Example for using constants in the source code

```
#include <stdio.h>
#define PI 3.14159265

main( ){
        float x;
        const int c = 180;
        printf("Type an angle between 0 and 360 degree: ");
        scanf("%f", &x);
        if (x<0 || x>360)
                printf("Wrong data.");
        else{
                x = x * PI / c;
                printf("The value of the angle in radian: %f", x);
        }
}
```