Engineering program development 5

Edited by Péter Vass

Control structures

Control structures are very important parts of C programming language.
By means of them, we can determine the order of execution of the instructions (control flow) in a program.
The following elements are used for building up control structures:

- keywords (e.g. if, else, while etc.),
- round brackets for delimiting the expressions connected to the control statement of a control structure ( ),
- braces for delimiting a block of instructions { },
- statement terminator;

The two main objectives of their applications are the implementation of:

- conditional branches and
- loops in a program.

Control structures

Control structures used for implementing conditional branches

1. The conditional execution of an instruction or a block of instructions

```
        if (expression) instruction
```

or

```
        if (expression){
                instructions
        }
```

The condition is formulated in the form of an expression placed between round brackets. The instruction or block of instructions will be executed only if the logical value of the expression is true. Otherwise, the control flow of the program skips the instruction or instructions and continues with the next section of the code.

A condition is fulfilled when the value of its expression is not zero. On the contrary, the expression is false that is the condition is not fulfilled.

Control structures

Examples:

```
if (a > b) x++;

if (a > b){
  x++;
  y = pow(2, x);
}
```

<u>2. Two-way branching</u>

```
if (expression)
      instruction_1
else
      instruction_2
```

When the logical value of the expression is true (not zero) the instruction(s) belonging to the branch of "if" will be executed. Otherwise, the instruction(s) belonging to the branch of "else" will be performed.

Control structures

Examples:

```
if (a > b)
    x++;
else
  x--;

if (a > b){
    x++;
    y = pow(2, x);
}
else{
    x--;
    y = pow(2, x);
}
```

Control structures

3. Multiway branching

```
if (expression_1)
        instruction_1
else if (expression_2)
        instraction_2
…..
else
        instruction_n
```

When the expression belonging to the branch of "if" is true, the instruction(s) of that branch will be executed and the expressions of other branches will not be evaluated.
When the first expression is false, the expression belonging to the branch of next "else if" will be evaluated.
Several conditional branches can be inserted by the applications of "else if" statements. Each of them has its own condition which differs from the others.
If a condition is fulfilled, the instruction(s) of its branch will be performed and the other branches will not be evaluated.
If neither of the conditions is fulfilled, the instruction(s) belonging to the branch of "else" will be executed.

Control structures

Example:

```
if (x < 0)
    x++;
else if (x > 0)
    x--;
else
    ;
```

A statement terminator (;) without any instruction is called empty statement.

An empty statement does not do anything. We use it in such cases when at least one statement is formally required at a given point of a program but we do not want to execute anything at that point of the program.

When more than one instruction belongs to a branch, they have to be delimited by braces { }.

Control structures

4. Multiway branching with a "switch-case" structure

```
switch (expression){
        case constant_expression_1:
                instruction(s)_1
        case constant_expression_2:
                instruction(s)_2
        …..
        default:
                instruction(s)_n
}
```

A "switch – case" structure is used when the instruction(s) to be executed depends on the value of the expression belonging to the switch keyword. In that case, the expression has not a logical value (unlike the case of "if" an "else if" branches) but a constant.
After the expression has been evaluated, its value is compared to the values of constant expressions following the "case" statements.

Control structures

4. Multiway branching with a "switch-case" structure

```
switch (expression){
        case constant_expression_1:
                instruction(s)_1
        case constant_expression_2:
                instruction(s)_2
        …..
        default:
                instruction(s)_n
}
```

If the value of expression is equal to the value of some constant expression, the instruction(s) belonging to the "case" branch of that constant expression will be executed.
If neither of the constant expressions fits to the value of expression, the instruction(s) assigned to the "default" branch will be performed.

Control structures

```
#include <stdio.h>
main(){
    int n;
    printf("Type a positive integer less than 10: ");
    scanf("%d", &n);
    if (n>10 || n<1){
        printf("Wrong number!\n");
        return -1;
    }
    switch (n){
        case 1:
            printf("The given number is one.\n");
            break;
        case 2:
            printf("The given number is two.\n");
            break;
        case 3:
            printf("The given number is three.\n");
            break;
        default:
            printf("The given number is greater than three.\n");
            break;
    }
    printf("The program has finished.\n");
}
```

Control structures

4. Multiway branching with a "switch-case" structure

The statement "break" at the end of each "case" branch is used for skipping the evaluation of other "case" branches if the instruction(s) before the "break" statement has already been executed.

In such a way, we can ensure that the evaluation of further "case" branches will not continue after a constant expression with the right value has been found.

Without using break statements, all the constant expressions would be compared to the expression belonging to the switch statement.

Of course, only the instruction(s) belonging to the constant expression with the right value will be executed.

The instructions assigned to a case branch must not delimited by means of braces { }, but the colon is compulsory after each constant expression of a case branch.

Control structures

5. Pre-test loop

```
while (expression)
        instruction
```

or

```
while (expression){
        instructions
}
```

We have known its work previously.

6. Post-test loop

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

or

```
do {
        instructions
} while (expression);
```

We have also known its work previously. Do not forget to put a semicolon after the expression of while keyword.

Control structures

<u>7. "for" loop</u>

```
for (init_exp; condition; update_exp)
        instruction
```

or

```
for (init_exp; condition; update_exp){
        instructions
}
```

init_exp is an initialization statement which assigns an initial value to the loop control variable (e.g. i = 0)

The second statement is a *condition* for the loop control variable (e.g. i <= 8). By means of that, we can specify a lower or upper limit for the value of loop control variable. The instructions inside the loop body will be executed whenever the result of its test is true.

The third statement is an update statement which determines how the value of loop control variable changes after each execution of the loop body (e.g. i = i+2).

Each "for" loop can be replaced by a "while" loop but not all the while loops can be replaced by "for" loops.

Control structures

break statement

Break statements can be applied not only in switch-case structures but also in loops.

By means of a break statement the control flow can be jumped out of the loop body. So the repetition of loop body stops at once.

The execution of break is always tied to a conditional expression. If the condition is fulfilled, the execution of the loop body will finish and the program will continue with the execution of next statement after the loop.

continue statement

We can also use it in loop bodies in order to skip the actual iteration of a loop body.

Similarly to break statement, it is always tied to a conditional expression. If the condition is fulfilled, the actual iteration of the loop body will stop and the next iteration will start. So, not the execution of the loop stops but only the actual iteration of the loop body

Control structures

return statement

Return statement is used in the definitions of user-defined functions. It is placed inside the body of a function. By means of a return statement, we specify the exit point of the function.

When the control flow arrives a return statement the execution of the code of a function stops and the control flow returns to the *calling function* (the function in which the call of another function is implemented).

If we apply a return statement in the body of a main function, the execution of the program will stop at that point.

If an expression follows a return statement, the value of the expression will be the value of output parameter of the function (return value).

The type of output parameter of a function is defined in the first row of the function definition.

If we do not want a function to have a return value, we must define the type of output parameter as void.

Control structures

```
#include <stdio.h>

main(){
/*The program reads a line from the standard input and
computes the number of characters in the line*/
      int nc=0;
      char c;

      printf("Type a line then press enter.\n");
      do{
            c = getchar();
            nc++;
            putchar(c);
      }while ( c != '\n');
      printf("\nNumber of characters: %d\n", --nc);
}
```

Control structures

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

main(){
/*The program reads a line from the standard input and
computes the number of characters before the first
whitespace character in the line*/
       int nc=0;
       char c;

       printf("Type a line then press enter.\n");
       do{
              c = getchar();
              if (isspace(c))
                     break;
              nc++;
              putchar(c);
       }while ( c != '\n');
       printf("\nNumber of characters before the first
whitespace character: %d\n", nc);
}
```

Control structures

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

main(){
/*The program reads a line from the standard input and
computes the number of non-whitespace characters in the
line*/
       int nc=0;
       char c;

       printf(" Type a line then press enter.\n");
       do{
              c = getchar();
              if (isspace(c))
                     continue;
              nc++;
              putchar(c);
       }while ( c != '\n');
       printf("\nNumber of non-whitespace characters:
%d\n", nc);
}
```

Control structures

Explanation

The functions getchar( ) and putchar( ) included by the standard input/output function library.

```
c = getchar();
```

The function reads a character from the standard input (keyboard) and returns the character code of the character (a positive integer stored in a single byte).

Here, the character code is given to the variable c.

```
putchar(c);
```

The function has a single input parameter for accepting a character code. The function displays the character in the standard output (on the monitor).

Control structures

Explanation

`isspace(c)`

It is also a standard library function whose declaration is included by the header file ctype.h .

That header file contains the declarations of library functions connecting to the examination and manipulation of characters.

These functions have a single input parameters for accepting a character code.

They perform some kind of comparison regarding the given character.

If the result of the comparison is false, the return value of the functions is zero. On the contrary, the return value is not zero.

Control structures

Explanation

`isspace(c)`

The function isspace examines whether the character whose code is stored in the variable c is a whitespace character (space, tabulators, new line etc.) or not.

If the character is a whitespace character, it returns a non-zero integer value. On the contrary, its return value is zero.

In the example codes, the call of isspace appears in the conditional expression of an if statement.

The value of the expression will be true if the input character of isspace is a whitespace character.

Control structures

<u>Some library functions served for the examination of characters</u>

islower(c)	true if the character is a lowercase
isupper(c)	true if the character is an uppercase
isalpha(c)	true if the character is a letter
isdigit(c)	true if the character is a decimal digit
isprint(c)	true if the character is a printable one (letters digits, space tabulators, new line etc.)
iscntrl(c)	true if the character is a control character (non-printable)

<u>Library functions for converting characters</u>

int tolower(c)	convert an uppercase character into a lowercase one
int toupper(c)	convert a lowercase character into an uppercase one