Course
ELEC-A7100
Course materials
Your points
Microsoft Teams
Code Vault

This course has already ended.

« 2. Input and output Course materials 2 Conditional statements »
ELEC-A7100 / 2. Input and output / 1 Formatted input and output basics

Second round tasks as a `ZIP file.`

Now that the C basic data types and syntax start to be in order, in this module we get familiar with **input** and **output**, and **control structures** that are essential in programming, such as loops and conditional statements. If you have programmed before with any language, basics of control structures are probably familiar, but the syntax in C differs from many other languages, particularly Python.

After the module you can read user input to different types of variables, and output data using varying formatting rules. The conditional and loop statements will also become familiar.

Formatted input and output basics¶

Formatted output¶

The **printf** function can be used for outputting information from the programs. The printf function takes a string as a parameter, and can optionally have any number of additional parameters for variables that are printed as part of the string. For example, the following code sample prints "The number is 50", followed by newline character (not visible in the output), that causes the following output to appear at the beginning of the next line. The printf function interface is defined in include header 'stdio.h'. Therefore the `#include` directive is needed in the beginning of the program always when printf is used.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int number = 50;
6      printf("The number is %d\n", number);
7
8      return 0;
9  }
```

The *printf* parameters are placed in the output string by using **formatting conversion specifications**. In its basic form, a conversion specification consists of a percent sign (%) and a letter that indicates the type of conversion. The conversion specification is replaced by the parameter given in the printf call. If there are multiple parameters (separated by comma), **multiple conversion specifications need to be used**. The number of conversion specifications must be the same as the number of parameters in the printf call. The type of conversion specification must be compatible with the data type of the corresponding parameter.

Here are some conversion types:

- **%d**: (int) – integer in decimal format
- **%u**: (unsigned int) – unsigned integer in decimal format
- **%o**: (unsigned int) – octal number
- **%x, %X**: (unsigned int) – hexadecimal number, using either lowercase letters (former) or uppercase letters (latter)
- **%c**: (int) – single character based on the used character encoding (e.g., ASCII), as discussed earlier with the character constants.
- **%s**: (char*) – string. We will take a more detailed look into strings in the next section.
- **%f**: (double) – floating point number (format: n.nnnnnn). Default number of decimals included in output is 6.
- **%e, %E**: (double) – floating point number (format: n.nnnnnnE+-xx)
- **%g, %G**: (double) – choose either %f or %e format, depending on the value of exponent.

Task 2.1: Pi¶

Change the implementation of the above implementation in such a way that 3.14159265 value is stored as a floating-point number and the value of the variable (including decimals) must be printed.

Points 5 / 5 My submissions 1 Deadline Sunday, 13 June 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Pi

Select your files for grading

main.c

Choose File No file chosen

Submit

The following adjustments can be made on the formatting specification before one of the above letters, after the percent sign. Different adjustments can be combined, but they need to be in the following order:

- **number** (e.g `%4d`): Denotes field length/width.
- **minus** (-) (e.g. %-4d): align the output left of the available field, when the field length is specified.
- **plus** (+) (%+4d): for numeric conversion types, always include sign (+ or -).
- **0** for numeric conversion types with specified length, pad the field with leading zeros instead of space.
- **period followed by number** (`%4.1f`): for floating point numbers, the precision (number of decimals following the point).
- **h or l** (`%ld`): specifies that the argument is interpreted as short (h) or as long (l) form of the basic data type

More details can be found, for example, in the K&R book.

Below there are some examples about formatted output. The square brackets are not part of the formatting specification, but we use them to illustrate the width of the output field.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int numA = 10;
6      float numB = 2.54;
7      float numC = 0.000001;
8      printf("At least five characters long: [%5d]\n", numA);
9      printf("Length is six, one decimal shown: [%6.1f]\n", numB);
10     printf("Float number, aligned left: [%-10.2e]\n", numC);
11     printf("Number with leading zeros: [%05d]\n", ++numA);
12     return 0;
13 }
```

Line 11 above shows an example of using the unary increment operator as the printf parameter. Because the increment operator is prefixed, the increment is done before the value of expression is determined, and the value of **numA** is 11 when the printf function is called. If the parameter had been **numA++**, the call would have printed 10, because the increment is done after the value of expression is determined.

The printf call does not automatically start a new line. Multiple consecutive printf outputs will be shown on a single line, unless start of the new line is enforced by the `'\n'` special character. `'\n'` is not shown to user, but has its own ASCII encoding (10), that tells the console to change the line of output, and move to the beginning of the next line. `'\n'` does not have to be at the end of the output string, as above, but could be included in any place. Wherever it is, a new line is started at that point, and the following character appears on the beginning of the next line. Here are few of the special characters:

Here are few of the special characters:

- `\t`: tab – creates space horizontally on the screen.
- `\\`: produces a single backslash
- `\"`: produces a quote sign (")
- `\'`: produces a single quote

There are also some others that you can study from the K&R book, or from other material.

Formatted input¶

scanf is another function defined in the standard I/O library (stdio.h). It reads formatted user input, and applies similar conversion specifications as printf (there are some differences between the two in how e.g. long data types are handled, but for now you can assume that they are roughly similar).

Below is an example of two scanf calls.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a;
6      float b, c;
7      int ret_a, ret_b;
8
9      ret_a = scanf("%d", &a);
10     ret_b = scanf("%f %f", &b, &c);
11     printf("ret_a = %d, a = %d\n", ret_a, a);
12     printf("ret_b = %d, b = %f, c = %f\n", ret_b, b, c);
13     return 0;
14 }
```

Example input:

```
3
5.6,9.2
```

On lines 5-7 there are a few variable declarations. This time they are not initialized, so their initial values are unpredictable. The first scanf call (line 9) expects a single integer value from user, and places it in variable 'a'. The function returns when user has pressed 'enter' to start a new line. The second scanf call (line 10) expects from user two float values separated by comma, and places them in variables 'b' and 'c'.

The scanf function has an integer return value, that in the above example is stored in variables 'ret_a' for the first call and 'ret_b' for the second call. The return value tells how many fields were read successfully. If everything went correctly, the ret_a should contain value 1 after the call, and ret_b should contain value 2 after the call. If user gave misformatted input, the return value will be smaller, for example 0. Therefore checking for the return value is recommended after the scanf call (you will see soon, how). If user had entered invalid input, the contents of variables a, b and c could remain uninitialized. As with printf, the number of scanf parameters following the string needs to match the number of conversion specifications.

The scanf function stops reading if the formatting specification does not match the given input, and returns the number or correctly read parameters. For example, if there have been multiple values on a single line of input, and one of them does not match the format specification, **the next call to scanf encounters the same input again**. scanf also **ignores whitespace** characters in the user input. Whitespace characters are, for example space, tab and newline characters. An exception to this rule is reading a character using scanf (the '%c' formatting conversion): it accepts also whitespace characters.

For basic data types the scanf parameters need the & operator as prefix. This relates to addresses and pointers that we will look at more later. For now, just include them in your scanf calls, and assume that in near future you will know why.

« 2. Input and output Course materials 2 Conditional statements »