This course has already ended.

« 1 Formatted input and output basics Course materials 3 Loops »
ELEC-A7100 / 2. Input and output / 2 Conditional statements

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.

Conditional statements¶

Statements and blocks¶

A function body consists statements that can contain expressions, and each statement is terminated by semicolon. **Compound statement** is is a group of statements that are separated by opening and closing braces { and }. In terms of program structure, a compound statement itself is a statement. Declarations that are done inside a compound statement block, are only visible inside the block. Similarly, declarations done in the function definition (that can be seen as a top-level compound statement) are visible only inside that function. Usually compound statement blocks are used together with control functions of the program, such as in if-conditions or loops, but it is possible (although not very common) to use them just stand-alone.

Here is an example to illustrate this:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    a = a + 1;
    {
        int b = 6;
        b = b + 1;
    }
    a = a + 5;
    printf("a: %d", a);
    return 0;
}
```

What happens when a program attempts to compile and why?

Task 2.2: Blocks¶

Correct the program so that it compiles and produces the correct output for variable a after the mathematical operations present in the program

Points 1 / 1 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This course has been archived (Saturday, 31 December 2022, 20:00).

Blocks

Select your files for grading

main.c

Choose File No file chosen

Submit

Relational and logical operators¶

Relational and logical operators result in either 1 or 0, depending on whether the condition in the operator is true or false.

The relational operators are:

- `<` - less than
- `<=` - less or equal than
- `>` - greater than
- `>=` - greater or equal than
- `==` - equal **(important: notice the difference to assignment operator with one '=')**
- `!=` - not equal

The following example shows how comparison operators work, and how they relate to integer return values:

```
#include <stdio.h>

int main(void)
{
    int a;
    int ret = scanf("%d", &a);
    if (ret > 0) {
        int a_res = a < 5;
        printf("a less than 5: %d\n", a_res);
        printf("a equal to 5: %d\n", a == 5);
    }

    return 0;
}
```

In above example an integer value is first read to variable 'a' (the implementor has been lazy to not check the return value of the scanf call). Then, variable 'a_res' is set to the result of logical operator a < 5, and becomes either 0 or 1, depending on what user gave as input. Line 7 demonstrates that the relational operators can be used in expression as any other operator, and can therefore be used, for example as function parameters. The printf function will show 1 if user had typed '5', otherwise it will show 0.

In addition, there are logical operators for AND, OR and NOT:

- AND operator is `&&` for example expression `(a < 5 && b > 6)` is true if a is smaller than 5 **AND** b is greater than 6.
- OR operator is `||` for example `(a < 5 || b > 6)` is true if either a is smaller than 5 **OR** b is greater than 6.
- NOT (unary) operator is `!` in front of an expression, and negates the outcome of the expression. For example `!(a < 5)` says 'a is not smaller than 5'. i.e, it is greater or equal to 5.

A common mistake is to confuse **&&** and **||** (the logical operators) with **&** and **|** (bitwise operations, explained in later sections), causing a different outcome. Similarly, it is common to confuse **==** (equality) with **=** (assignment). Because both the assignment and bitwise operators can be used as part of expression, the compiler accepts both forms, but use of wrong operator leads to wrong behavior.

if...else statements¶

The if-else structure can be used to implement decisions in the program, as with most other programming languages. The structure is:

```
if (expression)
{
    statement-1
}
else
{
    statement-2
}
```

If **'expression'** is true, statement-1 is executed. If it is false, **statement-2** is executed. In C language, any non-zero value of expression is interpreted as true, and zero is interpreted as false. **"expression"** can be any C expression, and can contain a function call or technically even just constant value (which would not make any sense in practice). Often relational or logical operators are used here, but they do not have to be used. For example, assuming integer variable a, statement if (a) would test whether variable a contains a non-zero value. **'statement-1'** and **'statement-2'** must either be terminated with semicolon, or they can be compound statements indicated by curly braces.

Here is a simple example:

```
#include <stdio.h>

int main(void) {
    int days, years; // useita muuttujia voidaan esitellä pilkulla erotettuna
    int ret;
    ret = scanf("%d %d", &days, &years);
    if (ret >= 2) {
        if (days > 365) {
            years++;  // tai: years = years + 1;
            days -= 365;  // tai: days = days - 365;
        }
        else
            printf("%d days remaining until the next year\n", 365 - days);

        /* Ei aaltosulkuja else-haarassa -- */
        seuraava rivi suoritetaan kummassakin tapauksessa */
        printf("days: %d  years: %d\n", days, years);
    } else {
        printf("Invalid input!\n");
    }

    return 0;
}
```

Example input:

```
400 2
```

The above example does not use curly braces in the else - branch, which therefore consists of only one statement. Forgetting curly braces unintentionally could also cause buggy behavior. Therefore they are often used for clarity even if they would contain just one statement. The else branch is not mandatory, and can be left out, if there is no viable alternative code to execute.

An if-else construct can have more than two parts: another if statement can follow directly after else, and this can be repeated any number of times, until including the final else in the end, which is not mandatory. For example:

```
#include <stdio.h>

int main(void) {
    int a = 0;
    scanf("%d", &a);  // jos ei numero, a:n arvoksi jää 0

    if (a == 1)
        printf("one\n");
    else if (a == 2)
        printf("two\n");
    else if (a == 3)
        printf("three\n");
    else
        printf("some other number\n");

    return 0;
}
```

Switch¶

The **switch** statement is another way for doing multi-way decisions, when the options are constant integers. The switch statement compares an expression to constant labels listed with word **case**, and in case of matching label executes the following code. Here is an example that reads one character from user, and evaluates whether it is one of a few alternatives:

```
#include <stdio.h>

int main(void)
{
    char a = 0;
    scanf("%c", &a);  // lue yksi merkki käyttäjältä

    switch(a) {
    case '1':  // ASCII '1' on sama kuin kokonaisluku 49
        printf("user typed one\n");
        break;

    case '2':
        printf("user typed two\n");
        break;

    case 'a':
    case 'b':
    case 'c':
        printf("user typed a, b or c\n");
        break;

    default:
    printf("user typed something else\n");
    break;
    }

    return 0;
}
```

Pay attention to the differences between switch case and if-else construct. In the case of switch, multiple statements can follow each case statement without enclosing them inside compound statement (i.e., curly brackets). Usually after the last statement of each branch there is a **break** statement that causes the program to jump out of the switch processing, to the code that follows the ending brace of the whole switch statement (we will see more about **break** shortly). If break is not included, the execution would continue through the next labels: for example, if break was removed from branch '1' above, the program would print two lines of output when user typed character '1'. While this property is a common reason for bugs in C programs, it allows assigning multiple labels for a piece of code, as is done for 'a', 'b' and 'c' above. Finally, a special label "default" is used to match all cases. Usually a good habit is to include break statement also after the final branch, even though it is logically unnecessary. This helps avoiding bugs if program is extended later.

Remember: '1' is also a constant integer, based on the ASCII character encoding table, and is equivalent to 49 in decimal format. Number 1 and character '1' are different values: **printf("%c\n", '1');** outputs 1, but **printf("%d\n", '1');** outputs 49, because the latter prints numeric value corresponding the character constant, but former assumes character encoding.

Note that switch can always be replaced with series of if..else if.. statements that perform the corresponding set of tests.

Task 2.3: Calculator¶

**In this task, you will learn to use conditional statements and formatted input and output using scanf and printf. Two separate functions must be implemented in this task, both of which will be reviewed and scored separately. Both of these functions must be implemented in source.c - file. **

simple_multiply: Implement a function *simple_multiply* which asks user for two integers, multiplies both integers and finally, prints the result in the following format:

```
1 * 2 = 2
```

(`'\n'`) must be present at the end of input values and output value. In the following example, user input (first line) and output of the program with given input values (second line)

```
4 5
4 * 5 = 20
```

simple_math: Write function `void simple_math(void)` that asks three values from the user: number, operator, and another number. Operator should be one of the following characters: '+', '-', '*' or '/'. If some other character is used as operator, the function should print "ERR" (without quotes). The numbers should be **float** type. If user does not type a valid number-operator-number combination, the function should print "ERR". When valid input is given, the function performs the calculation as given, and prints the result on the screen, using the **precision of one decimal**.

The following example shows sample input and output of the program:

```
8 - 2  (user input)
6.0  (answer)

5 * 5  (user input)
25.0  (answer)

8.3 / 5.1  (user input)
1.6  (answer)

-3.456 - 2.31  (user input)
-5.8  (answer)
```

Hint

- Since **scanf** considers space as a delimiter for input, each input value must be given in separate lines or on the same line separated by spaces.
- Check how to scan character constants using **scanf** function and also how to use character constants.

Points 40 / 40 My submissions 2 Deadline Sunday, 13 June 2021, 19:59 To be submitted alone

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

Calculator

Select your files for grading

source.c

Choose File No file chosen

Submit