Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 2)

❖ Handling Standard Input/Output

When we say input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement. On the other hand, when we say output, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

1) **stdin** – The standard input for reading (keyboard)
2) **stout** – The standard output for writing (screen)
3) **stderr** – The standard output for writing error messages (screen)

For example, printf normally uses standard output stream.

- Getting Input from the User:

In order to read a character from a file stream the **getc** function can be used. If we would like to read from *stdin*, then it is also possible to use **getchar** function. These functions read the next available character from the screen and return it as an integer. Examine the following example:

```
#include <stdio.h>

int main()
{
   int ch;
   printf("Type in a character: ");
   ch = getc(stdin); // Or similarly ch = getchar();
   printf("You entered %c\n", ch);
   return 0;
}
```

Note that these functions read only single character at a time.

- Printing Output on the Screen:

Similar to getc and getchar functions, there exist **putc** and **putchar** functions to print characters in a file stream. These function also put only single character at a time. Consider the following example in which different usages of the above functions are demonstrated:

```
#include <stdio.h>

int main()
{
   int ch = 65;
   putc(ch, stdout);
   putchar(66);
   putchar('g');
   return 0;
}
```

- The printf Function:

The **printf** function writes the output to the standard output stream and produces the output according to the format provided. So, it is used to print formatted output on the screen.

The formatted string means a character string that may contain format specifiers. For instance,

```
printf("%d + %d is %d\n", 5, 2, 5+2);
```

Here, the output will be "5 + 2 is 7". List of the format specifiers are given in the below table:

%c	Character
%d - %i	Integer
%f	Floating-point
%e - %E	Scientific notation
%g	%f or %e (whichever is shorter)
%G	%f or %E (whichever is shorter)
%s	Null terminated character string
%x	Hexadecimal format
%o	Octal format
%%	Outputs a percent sign (%)

It is also possible to determine the minimum field width in a format specifier by using an integer just after the percent sign. Examine the following examples:

Code	Output
`printf("%5d %-5d",1,1);`	`    1 1`
`printf("%5d %-5d",12,12);`	`   12 12`
`printf("%5d %-5d",123,123);`	`  123 123`
`printf("%5d %-5d",1234,1234);`	` 1234 1234`

As it is seen, by using negative integers, left aligning the output is possible. Moreover, by using "0" right after the percent sign, we can also fill the spaces with zeros as seen in the below code examples:

Code	Output
`printf("%04d", 12);`	0012
`printf("%05d", 9055);`	09055

Furthermore, it is possible to determine the precision of the floating-point numbers to be shown on the screen. This is done by using **%n1.n2f** format specifier. Here **n1** is an integer that determines the total length of the output and **n2** is the number of decimal digits to be used after the dot(.). Examine the following code examples:

Code	Output
`printf("%6.3f", 2.6);`	2.600
`printf("%6.3f", 26.34);`	26.340
`printf("%6.3f", 26.3488);`	26.349
`printf("%6.3f", 260.344);`	260.344

Please note that when the minimum length (provided by n2) is exceeded, the output is rounded.

- The scanf Function:

The **scanf** function reads the input from the standard input stream and scans that input according to the format provided. So, it is used to read formatted inputs. The usage of the scanf function can be seen in the following C code.

```c
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);
   printf("Age: %d\n", age);
   return 0;
}
```

The scanf function can also be used to read floating-point variables, strings, etc. Furthermore, it is possible to read more than one input at a time, and read any desired format as performed in the below example.

```c
#include <stdio.h>

int main()
{
   int numA, numB;
   printf("Enter 2 numbers in a+b form to compute their sum: ");
   scanf("%d+%d", &numA, &numB);
   printf("Sum: %d\n", numA+numB);
   return 0;
}
```

❖ Decision Making

The programs we have been writing so far were somewhat dull as they were always following the same sequence of instructions. One of the main reasons why computers are such powerful tools is the fact that they can decide which instructions to follow depending on simple comparisons.

In C language, **if** and **switch** statements help us to write segments of code that will be executed only under certain conditions. The syntax for the simplest **"if"** statement can be given as follows.

```
if (condition)
{
   expressions...
}
```

Expression is executed only when the condition is true (nonzero). Here condition can be anything that results in an integer value. In execution time, if condition is true (nonzero) then the block of C statements in the **”if”** are executed, otherwise (condition evaluates to zero) they are skipped. Examine the following code:

```c
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);

   if (age < 40)
      printf("You are young.");

   return 0;
}
```

- Relational Expressions:

Relational expressions have the following syntax;

expression1 **relation_operator** *expression2*

Here, expression1 and expressions 2 are both arithmetic expressions and the relation operator is one of the followings:

Relation Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Note that relational operators have lower priorities than arithmetic operator. It means that the expression,

```
3 < 2 * 5 - 4
```

means that `3 < 6`, so the result will be 1 (logic **true**).

It is dangerous to use more than one relational operator in a single relational expression. For instance, the expression (0 < x < 5) is valid in C language. However, this may introduce confusion and should be avoided when possible. Let us examine the following example:

```
3 < 2 < 1  ⟶  (3 < 2) < 1  ⟶  0 < 1  ⟶  1 (true)
```

- Logical Operations:

It is possible to connect logical expressions using logical operations, which usually take the following syntax:

LE1 **logic_operator** LE2

Here, LE1 and LE2 are the logic expressions and the logic operator can be,

&& (and)
|| (or)

The result of **AND** operator is 1 (true) if and only if both logical expressions (LE1 and LE2) are nonzero (true). On the other hand, the result of **OR** operation is 1 (true) if one of the logical expressions (LE1 or LE2) is nonzero (true). The truth tables for the corresponding logic operators can be seen below.

Truth table for AND		Truth table for OR	
0 && 0	0	0 || 0	0
0 && 1	0	0 || 1	1
1 && 0	0	1 || 0	1
1 && 1	1	1 || 1	1

Also, there exists one more logical operator called as unary **NOT** (**!**) operator which has the following syntax:

! LE1

Here LE1 is a logic expression and the result of **!** (not) operation is the <u>reverse</u> of LE1. That is, if LE1 is nonzero (true) the result is 0 (false) and if otherwise the result is 1 (true).

It is important to know that && and || operators have lower priorities in comparison to arithmetic and relational operators. ! operator has the same priority with other unary operators (e.g. unary sign operators, ++, --). Priorities among logical operators are as follows:

Operator	**Priority**
!	High
&&	
||	Low

Consider the following code example:

```c
#include <stdio.h>

int main()
{
   int number, flag = 0;
   printf("Enter an integer: ");
   scanf("%d", &number);

   if (number >= 5 && !flag)
      printf("OK...");

   return 0;
}
```

In the above code, if the flag variable is logically false (means 0) as in our case, then the expression **!**flag becomes true. If the number is also greater than or equal to 5, then we will see "OK…" in the screen since we will have true inside of the if expression, otherwise, nothing is printed.

- The use of else statement:

Sometimes we want our program to do something under a certain condition and if that condition is NOT met we want our program to carry out some other tasks. The else statement can be used for this purpose. The syntax of the **if-else** statement in this case is as follows:

```c
if (condition)
{
   First block of statements...
}
else
{
   Second block of statements...
}
```

Here, if the expression is nonzero (true) the first block otherwise the second block of C statements will be executed. Let us examine the following C program:

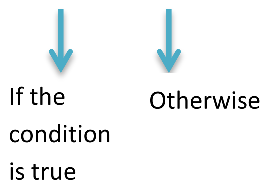
```
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);

   if (age >= 13 && age <= 19)
      printf("You are a teenager...\n");
   else
      printf("You are not a teenager...\n");

   return 0;
}
```

Note that since there is only one statement in statement blocks, we did not have to use the braces { }. However, it is still recommended to use them to improve readability.

- Condition Operator:

It is the only ternary operator in C. It has the following syntax:

```
condition ? exp1 : exp2
```



Here, exp1 is executed if the condition is true (not 0), otherwise exp2 is executed. The result of the operator is returned as the result of the executed expression. For instance;

```
I = R > 0 ? V/R : -1.0;
```

The above code is the same with,

```
if (R > 0)
   I = V / R;
else
   I = -1.0;
```

- Nesting if Statements:

It is important to know that the “if” statements can be used inside of another “if” (or “else”) statements. It is better to explain this case by the help of the following example:

```c
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);

   if (age < 18)
   {
      if (age < 12)
         printf("You are a kid...\n");

      else
         printf("You are a teenager...\n");
   }
   else
      printf("You are an adult...\n");

   return 0;
}
```

Actually, in such cases, we can also use "**else if**" statement which provides a better coding (and readability). Please check the next example:

```c
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);

   if (age < 12)
      printf("You are a kid...\n");
   else if (age < 18)
      printf("You are a teenager...\n");
   else if (age < 40)
      printf("You are still young...\n");
   else
      printf("You are getting old slowly...\n");

   return 0;
}
```

Note that in each run of the program only one block (which satisfies the given condition) is executed.

- switch Statement:

The **switch** statement can be used to make decisions where to jump in the program depending on the value of an expression. Thus, instead of using many nesting if-else statements, in some cases, switch statement makes the life easier. The syntax of the **switch** statement can be given as follows:

```
switch (statement)
{
case constant_expression1:
   First block of statements...
   break;
case constant_expression2:
   Second block of statements...
   break;
default:
   Default block of statements...
}
```

Here, the execution of the program continues from the case expression corresponding to the value expression. Note that case expressions are constant expressions evaluating to often an integer or character value.

Normally after jumping the proper block of C statements, the execution continues for all the remaining program blocks. Using **break** statements at the end of each block, it is possible to prevent this issue.

Finally, if there are no matching case expressions, then the execution jumps to the **default** block and statements written in this block are executed. Please check the following program written in C:

```
#include <stdio.h>

int main()
{
   int categoryId;
   printf("Enter category (1-3): ");
   scanf("%d", &categoryId);

   switch (categoryId)
```

```
    {
     case 1:
        printf("Category 1 is chosen!\n");
        break;
     case 2:
        printf("Category 2 is chosen!\n");
        break;
     case 3:
        printf("Category 3 is chosen!\n");
        break;
     default:
        printf("Error: Wrong category number!\n");
    }
    return 0;
}
```

In the above example, it is expected the user to choose a number between 1-3 and depending on the chosen category, some statements are executed. If the user enters any number other than 1, 2 or 3, then the default block will be executed.

- Using ++ and -- in Expressions:

The last topic will be covered in this lecture notes is the usage of increment (++) and decrement (--) operators in expressions. Although it is not suggested (due to readability considerations) it is possible to use ++ and -- operators in other expressions as introduced in the below examples.

Code	Meaning

`x = j++ + y;`	`x = j + y;`
	`j = j + 1;`

`x = ++j + y;`	`j = j + 1;`
	`x = j + y;`

`x = j + ++y;`	`y = y + 1;`
	`x = j + y;`

Please be careful about the order of increment and decrement operators!