Reading from the Standard Input

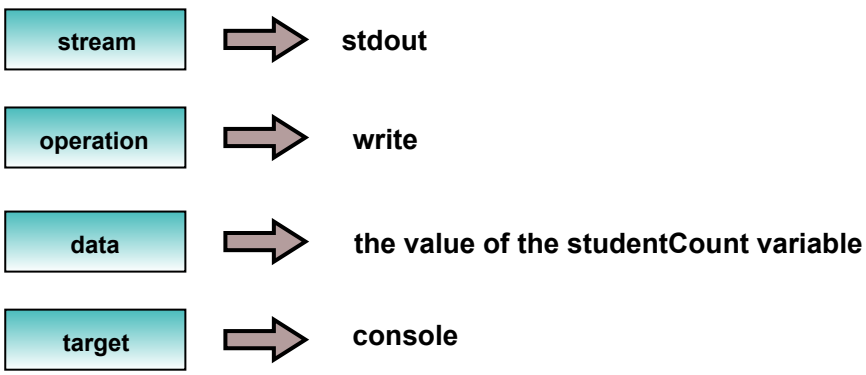
This lesson explains the working of stdout and stdin in more detail and provides a common spacing problem regarding readf.

WE'LL COVER THE FOLLOWING

stdout

As we've seen in the previous chapter, we don't need to type `stdout` when displaying the output because it is implied. What needs to be displayed is specified as an argument. So, the statement `write(studentCount)` is sufficient to print the value of `studentCount`.

To summarize:



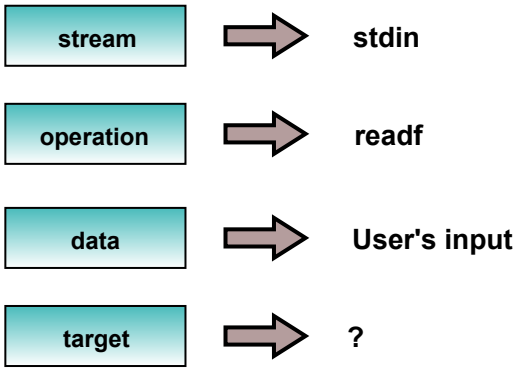
Stdout summary

Any data that is read by the program must first be stored in a *variable*. For example, a program that reads the number of students from the input must store this information in a variable. The type of this specific variable can be `int`.

readf

The reverse of `write` is `readf`; it reads from the standard input. The 'f' in its name comes from "formatted" because what it reads must always be presented in a certain format. In the previous lesson, we've also seen that the standard input stream is `stdin`.
In the case of reading, one piece of the puzzle is still missing: where to store the data that is read.

To summarize:

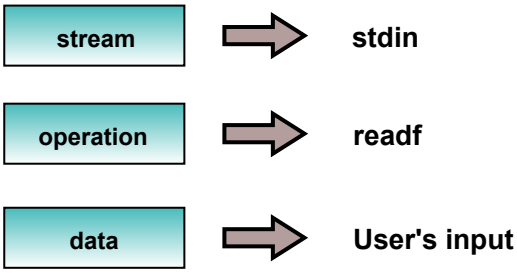


stdin summary

Location of data

The address of a variable specifies where data can be stored. The address of a variable is the exact location in the computer's memory, where its value is stored.

In D, the `&` character is typed before a variable name to specify the address of this variable in memory. `&studentCount` can be read as "the address of `studentCount`." So, the missing piece in the above figure is:



stdin summary

Typing a `&` in front of a variable name means *pointing* at what it represents. This concept is the foundation of references, reference function parameters and pointers which we will see in later chapters.
The first argument to `readf` must be `%s`:

```
readf("%s", &studentCount);
```

Actually, `readf` can work without the `&` character as well:

```
readf("%s", studentCount); // same as above
```

`%s` indicates that the data should automatically be converted to a form that is suited to the type of the variable. For example, when characters '4' and '2' are read in a variable of type `int`, they are converted to the integer value 42.
The program below asks the user to enter the number of students:

> **Note:** To enter the number of students, click on the >_**STDIN** button, write your input there, and then click the **RUN** button.

```
import std.stdio;

void main() {
    write("How many students are there? ");

    /* The definition of the variable that will be used to
    * store the information that is read from the input. */

    int studentCount;

    // Storing the input in the variable

    readf("%s", &studentCount);
    writeln("Got it: There are ", studentCount, " students.");
}
```

Getting input program

Skipping the whitespace characters

Even the *Enter key*, which we press after typing the data, is stored as a special code and is placed into the `stdin` stream. This is useful for the program to detect whether the input takes a single line or spans multiple lines.

Although sometimes useful, such special codes are mostly unimportant for the program and must be filtered out from the input. Otherwise, they block the input and prevent reading other data.

We can see this problem in the following program, which reads the number of teachers in addition to the number of students:

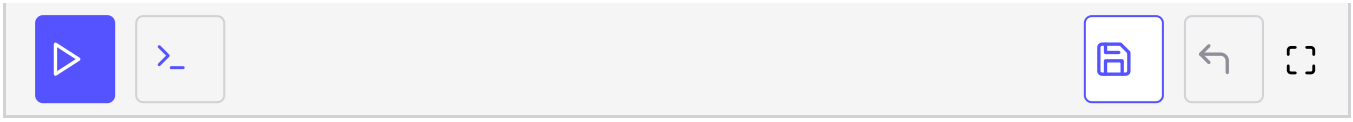
```
/* This program does not execute suucessfully*/

import std.stdio;

void main() {
    write("How many students are there? ");
    int studentCount;
    readf("%s", &studentCount);

    write("How many teachers are there? ");
    int teacherCount;
    readf("%s", &teacherCount);

    writeln("Got it: There are ", studentCount, " students",
            " and ", teacherCount, " teachers.");
}
```



Using enter key in stdin

Unfortunately, the program cannot use that special code when expecting an `int` value:

```
How many students are there? 100
How many teachers are there? 20
  ← An exception is thrown here
```

Although the user enters the number of teachers as **20**, the Enter key that was pressed after **100** is still in the *input stream* and is blocking it. The characters in the input stream appear as follows:

```
100[EnterCode]20[EnterCode]
```

The solution is to use a space character before `%s` to skip the Enter key code that appears before the number of teachers: “ `%s`”. As a general rule, you can use `%s` for all data that is read from the input.

Spaces in format strings are used to read and ignore zero or more invisible characters that would otherwise appear in the input. Such characters are called the **whitespace characters**, and they include actual space character, the code(s) that represent the *Enter* key, the *Tab* key, etc.

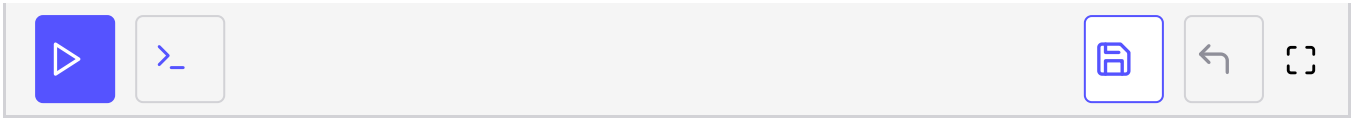
The program above works as expected with the following changes:

```
import std.stdio;

void main() {
    write("How many students are there? ");
    int studentCount;
    readf(" %s", &studentCount);

    write("How many teachers are there? ");
    int teacherCount;
    readf(" %s", &teacherCount);

    writeln("Got it: There are ", studentCount, " students",
            " and ", teacherCount, " teachers.");
}
```



The output:

```
How many students are there? 100
How many teachers are there? 20
Got it: There are 100 students and 20 teachers.
```

Additional information

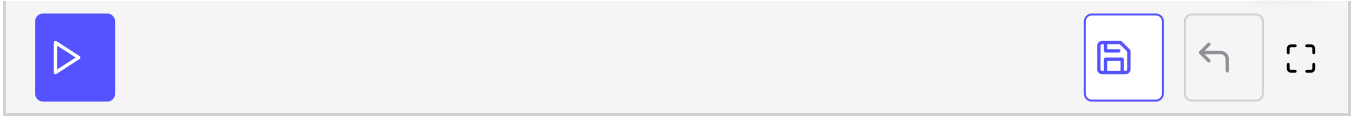
- Lines that start with `//` are single-line comments. To write multiple lines as a single comment, enclose the lines within `/*` and `*/` markers. To comment even more comments, use `/+` and `+/`:

```
// A single line of comment
 /*
   A comment that spans
```

```
   multiple lines
*/
```

- Most of the whitespace in the source code is insignificant. It is good practice to write longer expressions as multiple lines or add extra whitespace to make the code more readable. Still, as long as the syntax rules of the language are observed, the programs can be written without any extra whitespace:

```
import std.stdio;void main(){writeln("Hard to read!");}
```



Hard to read code

It can be hard to read the source code with a small number of whitespaces.

Practice question

For a better understanding, try this problem on your own.

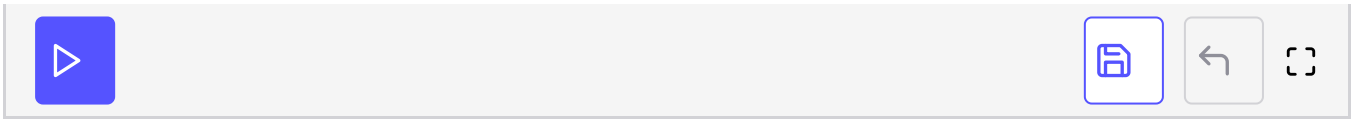
Problem statement

Enter non-numerical characters when the program is expecting integer values and observe how it behaves.

```
import core.stdc.stdio;

immutable char[] nullTerminatedStr = "Hello, World!\0";

int main()
{
    puts(nullTerminatedStr.ptr);
    return 0;
}
```



In the next lesson, you will find a quiz to test the concepts you have learned in this chapter.