Introduction to C
Tutorial 2: Variables, Input, Output, and Redirection
Agenda

- Variables
  - In Hardware:
    - Regulating access to computer memory
  - In Human communication:
    - Nouns, properties, measurement, control flags, and more.
- Input/Output
  - Happens outside of code. A block box for the user.
- Redirection
- The Input Buffer
Variables
Computer memory is used to store intermediate results of calculations. Eventually these are shown on the screen or sent to a different output device.

Every memory block has an address.

Address of memory block:
- #1008

Value:
- 91

Here, we refer to memory block 1008 with the name base_grade.

Machine code command:
- #1008 ← 91
- #1008 ← #1008 + 4

Let’s write it a bit more clearly:
- base_grade ← 91
- base_grade ← base_grade + 4

Reading, calculating, and writing to an address.

Writing a set value into a specific address.
A variable is a **memory block that is given a name and has a value**. Before using a variable, we have to **define** it, by giving it a name and type. Definition of a variable in C is executed using a declaration like the following:

\[
\text{int } x;
\]

The word "int" sets the **type** of the variable. The type is a permanent part of the variable. The word "x" is the name of the variable. Ends with a semicolon.

When a program runs, the operating system allocates memory for the variables in the program.
Variable definitions

- You can define multiple variables in one line.

```c
int x, y;
```

- You can set the initial value of a variable when you define it.

```c
int x = 0;
int y = 13;
```

- You can also combine the definition and initial value

```c
int x = 7, y = 12;
```

If you do not set an initial value, the variable contains “garbage” – a value that is not known beforehand and therefore unpredictable.
We call the name of the variable (or function) the **identifier**.

Rules that define what a valid identifier is:
- An identifier can **English** letters, numbers, or an underscore “_”
- Can be of any length
- Cannot start with a number
- There is a list of words that cannot be used – **reserved words/keywords**.

There are 32 reserved words in C. We have encountered a few: **int, return, and others**...
Variable identifier

• Examples of acceptable variable names
  x, factor, total, sort_data, result12, After2OClock, _33

• Examples of **unacceptable** variable names:
  2day, 12, register, you&me, ibm.com

Uppercase and lowercase letters are not the same. Therefore, x and X are two different variable names.

• The computer doesn’t “care” what variable name you chose. Therefore, it’s important for **you** to choose names that have meaning that will help you understand your program.

  ```c
  int final_grade;
  int fg;
  ```

  – Which one is easier to understand?
Assignment to a variable

• You can update the content of a memory block by writing to the variable.
• Writing to a variable is marked by an equals sign “=”

\[ x = 13; \]

You can only write to variables that have been defined!

In this example, we assume that there was a definition for \( x \) earlier in the program “int x;”
Assignment structure

• The assignment is built from two parts:
  
  \[ x = 13 * 5 + 2; \]

  Left side – only the variable name appears.

  Right side – an expression followed by a semicolon.

• Execution of program:
  1. Computes the value of the expression on the right.
  2. Writes the result to the left side variable.
Assignment is not equation!!!!!!!!!

• When performing an assignment, the computer is not “solving the equation.”
• For example, the following line is not a legal assignment.

```
x + 3 = 7;  WRONG!
```

• An assignment as an order of execution.
  – First it executes the right side.
  – Then, it writes to the variable on the left side.
• Therefore, the next assignment is valid, even though it’s not an equation.

```
x = x + 1;
```
What happens in memory?

- We run the following code

```c
int x;
x = 10;
x = 11;
```

- The state of the memory:
Example: Computing an average

• Let’s assume we have two variables, \( a \) and \( b \).
• We give them the values 20 and 12, respectively.
• We save/store the result in the variable \( \text{avg} \).
• We are interested in computing the average of the \( a \) and \( b \) variables.
• The relevant portion of the program:

```c
int a, b, avg;
a = 20;
b = 12;
avg = (a + b) / 2;
```

When the program is done running, the memory that we named \( \text{avg} \) will store the desired value.

In C, forward slash “/” is used for division.
The full program

```c
int main(void) {
    int a, b, avg;
    a = 20;
    b = 12;
    avg = (a + b) / 2;
    return 0;
}
```

This line indicates the start of the `main` function, which is the starting point of the program.

Body of the function (in our case, the body of the program) given inside braces.

We will talk more later about functions in general and `main` in particular.

Variable definitions must happen in the beginning of the program, before they are used.

This instruction marks the end of the function (and in this case, the program).

It is expected and acceptable to return 0 if the program is successful, and return 1 or other value in case of a problem.
How to write clear code?

- One of the goals of a high level language is working in a manner that is understandable to other programmers.
- However, a high level language does not inherently provide for collaboration between different programmers.
  - How does one go about reading code that is “messy” or organized accordingly to someone else’s style?
  - And what if we want to leave a “note” for clarification?
- We’ll use indentation and comments in our code to address these problems.
Clearer Code 1 - Indentation

• An indentation is equivalent to many spaces together.
• Pressing the Tab key causes an indentation.
• It is customary to add an indentation every time braces are used, or in other specific cases:

  ```
  { int a = 2, b; 
    b = a*4;
  }
  ```

  This line is indented twice because of the additional brackets.

  Code without good indentation is like a book without paragraphs. It’s hard to read!

• It’s important to note that from the compiler’s perspective:
  – Line breaks and extra spaces do not matter.
  – Indents have no practical significance.
• Indents make the code more understandable.
To annotate your code, you can add comments. For example:

```c
int main(void) {
    int a, b, avg;
    a = 20;
    b = 12;
    /*Now, we compute the average*/
    avg = (a + b) / 2;
    return 0;
}
```

Comments are required when code is not clear. Write clear comments that can be read as a story. Comments should be written according to subtasks with meaningful names and in proper English.
Input and Output
What should it express?!

• The program we presented now works correctly
• It calculates the average of two numbers, and saves the result in memory
• But...
  – It always calculates the average of the same exact numbers!
  – It does not print the result, or report on it any other way!
What’s missing?

- In order for the program to calculate the average of any numbers we want, it should ask for values for a and b.
- Each time the program will run, the user should be able to provide other values.
- These numbers will be the **input** of the program.

- We also want the program to show the result of the calculation.
- For example, using the screen.
- The presentation of the results is called **output**.
- The program is like a **black box** for the user - the user enters the input and observes the output (indifferent to the details of its implementation – aka the inside of the box).
Before we can add input and output commands to the program, add the line below at the beginning (the very beginning, even before the main) line:

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

This command instructs the compiler to use the collection of functions defined in a special file called a stdio.h.

• stdio = Standard I/O = Standard Input/Output
• It contains functions with which the program can read (input) and write (output).
Output in C

• Output is performed using the printf function.
• The first program that we saw included the following line:

```c
printf("Hello World!");
```

• The result was the following string of printed letters on the screen: **Hello World!**
What can we print with printf?

- **Fixed strings** – text that does not change between runs of the program.

- **Special characters** – characters that have special behavior: 
  - \n
- For example, \n is called the “newline character.” Printing it results in a new line.

- What will the following line print?

```c
printf("Hello\nWorld!");
```
You can also print variable values with printf.

Include within the string the pair of characters \texttt{%d} (\textit{percent} followed by \textit{d}) where you would like the variable value to appear.

Outside of the string, using a comma separated list, include the variables to be printed in order.

- Include one variable in the list for every variable in the string.

```c
printf("a is %d, and b is %d", a, b);
printf("The average is %d.", avg);
```
#include <stdio.h>

int main(void)
{
    int a, b, avg;
    a = 20;
    b = 12;
    avg = (a + b) / 2;
    printf("The average is %d.\n", avg);
    return 0;
}
• In order to read in a value for a variable, we will use the \texttt{scanf} function.

• A simple read from \texttt{scanf} is executed as follows:

\begin{verbatim}
scanf("\%d", &a);
\end{verbatim}

• The above line reads in a value to the memory at \texttt{a}.

• Similar to \texttt{printf}, we can read in more than one variable at once.

\begin{verbatim}
scanf("\%d\%d", &a, &b);
\end{verbatim}

\textbf{Note the \& sign before the variable.}
Adding input to the program

```c
#include <stdio.h>

int main(void)
{
    int a, b, avg;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    avg = (a + b) / 2;
    printf("The average is %d.\n", avg);
    return 0;
}
```
Success of the input

• Does input always succeed? No.
  – For example, if a user enters letters instead of a number, it fails.

• Therefore, it’s important to always check if the input was as expected.
  – When scanf is done executing, we get a value.
  – We can store this value in a variable.

```c
int a, b, result;
result = scanf("%d%d", &a, &b);
```
Checking the success of the input

- The value returned by the scanf is the **number of values that it was able to accept**.

  If the value returned from `scanf` is less than the number of variables that we wanted to enter, then input was not successful.

- Bad input can cause unexpected behavior.

- When input fails, exit the program.

```c
int a, b, vars_n;
vars_n = scanf("%d%d", &a, &b);
if (vars_n < 2)
    return 1;
```

We’ll learn about “if” later.

2 because we want to verify that two variables were entered.

Here “return 1” signifies that we exit the program due to error.
#include <stdio.h>

int main(void)
{
    int a, b, avg, assigned_items_n;
    printf("Enter the first value:");
    assigned_items_n = scanf("%d", &a);
    if (assigned_items_n < 1)
        return 1;
    printf("Enter the second value:");
    assigned_items_n = scanf("%d", &b);
    if (assigned_items_n < 1)
        return 1;
    avg = (a + b) / 2;
    printf("The average is %d.\n", avg);
    return 0;
}
Redirection
Using the program we made

• Assume that we saved our file in `average.c`.
• After we compile, we’ll get a file called `average.exe`.
• The screenshot below shows a sample of the output using command prompt.

```
C:\Sfat-C\average>average.exe
Enter first value: 16
Enter second value: 128
The average is 72.
C:\Sfat-C\average>
```
Getting input from a file

- Currently, the program waits for input from the user (using the keyboard)
- We can make the program read input from a file using redirection.
- For example, we’ll create a file called input.txt that will have the following two lines:
  
  290
  1026

  You can make this file with notepad or other simple programs

- **Note:** the file extension used for an input file does not matter. You can call it “demo.in” or any other file extension.
Getting input from a file

• We’ll use command prompt in the following way:
  • After the name of the program, we add a “<“ symbol.
    – It’s the less than symbol, but here it means “from inside”
  • After the symbol, add the name of the file with input.

The program shows output and does not wait for input, because input came from the file
Sending output to a file

• In a similar manner, we use the greater than symbol to send output to a file ">"
  – The greater than symbol is similar to saying “to”
• **Note**: if the file already exists, the content will get overwritten!

The output is not sent to the screen – so the user is typing in the “dark”

Again, the file extension isn’t important.

"average to output.txt"
Sending output to a file.

• Open the file output.txt (with Notepad, for example), to check the output:

```
Enter first value: Enter second value: The average is 226.
```
What happens if we use both together?
The Input Buffer
The newline character stays in the input buffer, so what happens in the next scanf command?
And what happens if we type two numbers?

```
10 9
```

---

```
scanf("%d", &num1);
... 
scanf("%d", &num2);
```
How does *scanf* read from the buffer?

- When the program starts, the buffer is empty, so the program waits for the user to type.
- Each time the user presses ENTER, the characters go inside the buffer – (including the newline from the ENTER)!
- Each time that the buffer is not empty, *scanf* will try to read from it.
  - Whether or not the read was successful, only the characters that were read are emptied from the buffer.
  - If the buffer is emptied before *scanf* is finished is reading, the user is prompted to enter an additional line.
- *scanf* ignores *whitespace* (spaces, tabs, etc.) except in special cases that will be covered next week.
What is the behavior of the following program?

```c
int main()
{
    int a, b, c, d;
    scanf("%d%d", &a, &b);
    scanf("%d%d", &c, &d);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>20</td>
<td>b</td>
</tr>
<tr>
<td>30</td>
<td>c</td>
</tr>
<tr>
<td>40</td>
<td>d</td>
</tr>
</tbody>
</table>

Input:
10 20 30 40

Example
What is the behavior of the following program?

```c
int main()
{
    int a,b,c,d;
    scanf("%d%d", &a, &b);
    scanf("%d%d", &c, &d);
    return 0;
}
```

**Input:**
5 9 80
140 70

<table>
<thead>
<tr>
<th>Value</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>80</td>
<td>c</td>
</tr>
<tr>
<td>140</td>
<td>d</td>
</tr>
</tbody>
</table>
What is the behavior of the following program?

```c
int main()
{
    int a, b, c, d;
    scanf("%d%d", &a, &b);
    scanf("%d%d", &c, &d);
    return 0;
}
```

**Input:**

```plaintext
4 3 2 a
1
```

**Value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>???</td>
<td>d</td>
</tr>
</tbody>
</table>