# Chapter 5

## Variables and Constants

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Storing Information in the Computer</td>
</tr>
<tr>
<td>5.2</td>
<td>Declaring Variables</td>
</tr>
<tr>
<td>5.3</td>
<td>Inputting Character Strings</td>
</tr>
<tr>
<td>5.4</td>
<td>Mistakes in Programs</td>
</tr>
<tr>
<td>5.5</td>
<td>Inputting Numbers</td>
</tr>
<tr>
<td>5.6</td>
<td>Inputting Real Numbers</td>
</tr>
<tr>
<td>5.7</td>
<td>Constants</td>
</tr>
<tr>
<td>5.8</td>
<td>Assignment of Values to Variables</td>
</tr>
<tr>
<td>5.9</td>
<td>Understandable Programs</td>
</tr>
<tr>
<td>5.10</td>
<td>Comments in Programs</td>
</tr>
<tr>
<td>5.11</td>
<td>Exercises</td>
</tr>
<tr>
<td>5.12</td>
<td>Technical Terms</td>
</tr>
</tbody>
</table>
5.1 Storing Information in the Computer

So far the Turing programs you have seen just output messages or the results of a calculation. You can output a series of such items and format them so the output is easily understood. Nothing very exciting happens in programs until you learn how to input information and store it in the computer’s memory.

The kind of information or data we store is mostly either numbers (real or integer) or strings of characters. When you store information in the computer’s memory you must remember where you stored it. To do this you give each location where you will store data a name or identifier. What you store in a named memory location can change; it can vary. You call the locations that hold the data variables. In writing a program that uses variables you must declare the names you intend to give to your variables. You make up the names yourself. After you have established what the names will be, you can input information into the variables from the keyboard, change the values of the variables, and output their values into the window.

5.2 Declaring Variables

In naming variables you must declare what type of information they are to hold, that is, the data type of the variable. A variable to hold numbers can be either a real or int (for integer) data type. Strings of characters can be stored in variables of type string. Variables that you want to use in a program must be declared before you use them.

To declare a variable you write a line in your program which has the form

```
var name-of-variable : type-of-variable
```
A declaration begins with the keyword var (for variable). Here is a declaration which declares a variable called age which is to hold an integer.

```
var age: int
```

The name of the variable age is followed by a colon (:) then its data type which is int. The types are also keywords in the Turing language but the name is something you make up yourself.

### 5.2.1 Names of Variables

Names or identifiers of variables contain letters of the alphabet. After the first letter, you can use digits (0 to 9) and underscores (_), but no special characters such as spaces, hyphens, or periods. For example, page3 is valid name but 3rd is not because it does not have a letter as its first character. If you want to have a name which is really two or more words such as this year, you could write it as this_year but we will write it as one word thisYear using a capital letter to show where the second word begins. Here is a declaration for a variable to store a person’s first name.

```
var firstName: string
```

There is a list of words in Turing that are reserved and may not be used as names of variables. These words are listed in the appendix.

### 5.3 Inputting Character Strings

Here is a program that reads a string and outputs a message with the string in it.

```
put "Enter your first name"
var firstName: string
get firstName
put "Hello ", firstName
```
The first line of the program outputs a message which prompts you to enter your first name. The next line is the declaration of the variable `firstName`. After that is the input instruction. It starts with the keyword `get` and then follows the name of the location where what you input is to be stored. The last statement outputs two items: the string in quotes, then the value of the variable `firstName`. Type the program into your computer and try it. After the prompt

```
Enter your first name
```

appears in the window the cursor will be at the beginning of the line following the prompt. Nothing will happen until you type a name followed by Return. If you do not press Return after typing your name the computer will not know that you have finished. For example, if your name were Diana you might type a nickname such as Di and expect the computer to go on and say `Hello Di`. But it will do nothing until you press Return. Run it again, this time giving your second name instead of your first name. The computer does not tell you that you are mistaken. It cannot tell the difference between a first name and a second name or even a last name. Here is how the Execution window might look after the program has run

```
Enter your first name
Albert
Hello Albert
```

The computer produces the first and third lines with the two `put` instructions; you type in the second. Both what you type and what the computer outputs are displayed in the Execution window in the same way so that when you are finished you cannot tell the `input` from the `output`.

### 5.3.1 Strings Containing Blanks

When you input a string in the normal way it is assumed that there are no blanks in it. For example, if you enter your first and last names in the previous program your Execution window will look like this
Enter your first name
Albert Einstein
Hello Albert

The last name is ignored. This is because the \texttt{get} operator for a string variable reads characters until it hits "white space" such as a blank or a Return. Whatever comes after that is not read. You could get it to read the whole name if you put it in quotation marks as here

Enter your first name
"Albert Einstein"
Hello Albert Einstein

We call a quoted string or a string surrounded by white space a \texttt{token} and say we are using token-oriented input. Turing includes a version of the \texttt{get} command that allows an entire line including spaces to be input. (See Chapter 10 for information on "line-oriented input").

\section*{5.4 Mistakes in Programs}

All variables used in programs must be declared before you use them. If, by mistake, you omit a declaration the computer will give you an error message indicating that in a certain statement you are attempting to use an undeclared variable. This kind of error is called a \texttt{syntax error}. Turing, like the English language, has rules of grammar or syntax that must be obeyed. One of these syntax rules is that variables used in programs must be declared before use.

If, by mistake, you spell a variable's name differently in the declaration and the statement where it is used you get the same syntax error. You are using a variable that has not been declared. For example, if you forgot to capitalize the \texttt{N} in \texttt{firstName} in the \texttt{get} statement you would get a syntax error message. When you get such a message, change the program to correct the error and run it again.
5.5 Inputting Numbers

When you are inputting numbers you must remember that there are two kinds of numbers: real numbers and integers. A variable declared as int can hold an integer such as 23 but not a real number such as 8.47. A variable declared as real can hold a real number such as 8.47. It can also hold 23.0, which is a real number that we can consider to be the same as the integer 23. A string variable can hold a string of characters, such as Albert, but not a number as such.

Here is a program that reads an integer and outputs its square.

```turing
var number: int
put "Enter an integer"
get number
put "The square of ", number, " is ", number * number
```

Here the declaration of the variable number precedes the prompt. It does not matter in what order these are given since the prompt does not use the variable. Here is a sample execution

Enter an integer
12
The square of 12 is 144

Try running the program yourself.

5.5.1 Mistakes in Data

If in the previous program you enter a real number the computer will report an error. This is called an execution or runtime error since the error is not in the program itself but occurs during the running of the program. Here is an example where a data error occurs.

Enter an integer
8.47
Line 3: Illegal integer input
You should not enter a real number or a character string when the computer expects an integer.

### 5.6 Inputting Real Numbers

Here is an example using real numbers.

```
put "Enter the radius of a circle"
var radius: real
get radius
put "Area is ", 3.14159 * radius ** 2
```

Here is a sample Execution window.

```
Enter the radius of a circle
35
Area is 3848.44775
```

Notice that the input of an integer is legal for a real variable. At most 6 digits appear to the right of the decimal point in the area. If fewer appear it is because they are zero. Since exponentiation (**) has higher precedence than multiplication, the radius is squared before multiplying by 3.14159.

If the radius is large the area is output in the exponent form. Here is such an example Execution window.

```
Enter the radius of a circle
6754
Area is 1.433084e8
```

The digit 8 following the e is the power of 10 that is to multiply the significant digits part. For small values the exponent form is also used. For example,

```
Enter the radius of a circle
.0000897
Area is 2.527752e-8
```

The exponent form is used automatically by the computer when the number it has to output is too large or too small for its
normal form of real numbers. You can use real numbers in the exponent form in your program or input if you want to.

5.7 Constants

Sometimes we use memory locations to hold information that remains constant. Such constants can be declared just as variables are declared but in this case a value must be assigned to the constant at the time of declaration. For example, we could declare a constant \( \pi \) by this declaration

\[
\text{const } \pi : \text{real} := 3.14159
\]

Constants can also be strings. For example we could declare a constant \( \text{prompt} \) using this declaration

\[
\text{const } \text{prompt} : \text{string} := "Enter your name \\
\]

Notice that the name of the constant is followed by the type then := and then the value. As with variable declarations, constant declarations can appear anywhere in the program as long as the declaration precedes the use of the constant. We could use such a constant in our area program in this way

\[
\text{const } \pi : \text{real} := 3.14159 \\
\text{var } \text{radius}: \text{real} \\
\text{put } "\text{Enter radius } \ldots " \\
\text{get } \text{radius} \\
\text{put } "\text{Area is }, \pi \times \text{radius } \ldots 2
\]

Here is a sample Execution window.

Enter radius 5
Area is 78.53975

The result is basically the same as before. We give names to constants, such as 3.14159, to help make our programs more understandable. Notice that the prompt here has two dots after it so that the cursor remains on the same line for you to enter the
radius. The prompt has a blank at the end of it so that there will be a space before the radius (before 5).

Here is example of a program that uses a string constant.

```plaintext
const prompt : string := "Enter your first name"
var firstName : string
put prompt ..
get firstName
put "Hello ", firstName
```

Placing the text of the prompt in a constant declared at the top of the program allows you to change the text without having to search through the program. This becomes more important when you have large programs and want to use the same prompt several times.

## 5.8 Assignment of Values to Variables

So far we have just input values into the location of variables. We can also assign a value to a variable in an assignment statement. Here is an example program using an assignment statement.

```plaintext
const pi : real := 3.14159
var radius: real
var area: real
put "Enter radius ".
get radius
area := pi * radius ** 2
put "Area is ", area
```

Because radius and area are both real variables they can, if you want, be declared in the same declaration. For example

```plaintext
var radius, area: real
```

instead of

```plaintext
var radius : real
var area : real
```
The second last statement assigns the value of the arithmetic expression

\[ pi \times radius \times radius \]

to the variable \( area \). Then, in the following put statement, the value of \( area \) is output. This program does exactly the same thing as before except that now it is clear that you are computing the area of the circle.

Assignment statements are used when an intermediate value in a calculation has to be stored in memory, as well as to make the program easier to understand.

We can think of := as a left pointing arrow that moves the value into the variable. This is the same pair of symbols (:=) used when the value is assigned to a constant in a declaration.

Assignment statements can also be used with strings. Here is a program that reverses the order of two entered items.

```turing
var firstItem, secondItem : string
put "Enter the first item " ..
get firstItem
put "Enter the second item " ..
get secondItem
put "Here are the items reversed: " ..
put secondItem, " ", firstItem
```

## 5.9 Understandable Programs

Several times we have mentioned that we want our programs to be understandable. This is so you yourself can understand what is happening in the program even as you create it. You are less likely to make mistakes this way. If you have a written record of a program and come back to it after a day or so it is much easier to see what it does. Also another person, such as a teacher, can quickly read the program and perhaps see what is wrong with it if it is not doing what you expect it to do. Understandability is so important that computer scientists believe
that a program which is not easily understood is virtually worthless even though it may give a correct answer.

One way we have of trying to make programs easy to read is to put each statement on a separate line. This does not matter to the computer. We could, if we liked, run the program all together like this

```plaintext
const pi := 3.14159 var radius, area: real put "Enter radius" ..
get radius area := pi * radius ** 2 put "Area is ", area
```

But this is much harder to read. Sometimes a program will have a rather long line and we must start a new line. This is fine provided we do not break the line in the middle of a keyword, or variable name, or a number, or a quoted string. If you must break a quoted string in a `put` statement insert a quote and a comma before the break and quotes to begin the continuation. For example

```plaintext
put "Here is a very long line that had to ",
   "be broken"
```

## 5.10 Comments in Programs

There are a number of ways to make a program more easily understood. We try to choose names for variables which tell the reader precisely what the values to be stored in a variable location are to represent. We choose good variable names.

Another way to make a program understandable is to add comments to it to explain what it does or how it does it if it is at all obscure. Most good programs require only a few comments but they can be helpful.

A comment in the program begins with a `%` sign and ends with a Return. We could add this comment to our `circle` program

```plaintext
% Computes the area of a circle given its radius
```

A comment can be placed anywhere you like in a program. It is ignored by the computer; it is just for the reader’s benefit.
Sometimes we want to give a program a name so you can refer to it by name. We will be storing programs as files on the disk memory and files must have a name. A comment placed as the first line of a program can give its file name. We use a standard form which you may use. For example, as a first line in the CircleArea program we would have a comment

% The "CircleArea" program

Here is the CircleArea program with comments.

```turing
% The "CircleArea" program
% Computes the area of a circle given its radius
const pi : real := 3.14159
var radius: real
var area: real
% Ask the user for the radius
put "Enter radius " ..
get radius
% Calculate the circle's area
area := pi * radius ** 2
% Output the result
put "Area is ", area
```

Turing programs stored on disk should have .t added to their names to identify that they are Turing programs. This program would be stored as CircleArea.t.

### 5.11 Exercises

1. There are 2.54 cm in one inch. Write a program to input the length of a desk in inches and output its length in centimeters. Use a constant for the conversion factor. Be sure to prompt for the input and to label the output.

2. Write a program that asks for a person's year of birth and outputs their age in the current year. Write the program to work for whatever the current year happens to be.
3. Write a program that inputs the starting time and finishing time of an automobile trip as well as the distance in kilometers traveled and outputs the average speed in km/hr. The times are to be given in two parts: the hours, then the minutes.

4. Write a program that reads in four numbers and then outputs the four numbers all on one line with commas between the numbers.

5. Write a program to input a name and output a message that greets a name. For example, if the name is ÔSueÕ, then the message might be ÔHello Sue!Õ. Use constants for the greeting.

6. Write a program to calculate and output the product of three numbers entered via the keyboard. Also output the square of the product.

7. Write a program that inputs five full names and outputs the names in reverse order.

8. Write a program that inputs a first name and a last name then outputs this in the form

   last name, first name

   Try to write a version which inputs the two names from a single input line.

9. Experiment with programs where you purposely make syntax errors to see what the error messages are like.

10. Experiment with programs where you purposely input the wrong type of data to see what happens. For example, enter an integer when a string is expected.

11. See what happens when you run this program

    
    var age: int
    put "Enter age"
    get age
    age := age + 1
    put "age is ", age

    How do you interpret the assignment statement

    age := age + 1
Would another variable called `ageNextYear` make the program more understandable?

12. Experiment with adding comments to a program. See if you can add a comment to the end of a line. Can a comment be in the middle of a line?

13. The Prom Committee at your school will sell tickets at $65 per person. Expenses include the cost of the food, the DJ, the hall, the decorations, and the waiting staff. To these expenses, add $100 for miscellaneous expenditures. Write a program to ask for each of the expense totals incurred and then calculate and output to the committee how many tickets they must sell to break even.

14. A student wrote 5 tests. Ask the student for their name and then what each test is marked out of and what mark they received on each test. At the end of the run calculate and output the percentage for each test as well as the average on all five tests also as a percent. When querying the student, address each request with their name. Make sure that output statements include the name of the student.

15. Ask the user for a real number which expresses the area of a figure. Assume that the figure was a circle. Output the radius and then the circumference of the circle. Now assume that the figure was a square. Output the length and width and then the perimeter of the square. Use complete statements in each case.

### 5.12 Technical Terms

<table>
<thead>
<tr>
<th>Data</th>
<th>Reserved Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory of computer</td>
<td>type of variable</td>
</tr>
<tr>
<td>variable</td>
<td>real</td>
</tr>
<tr>
<td>declaration of variable</td>
<td>int</td>
</tr>
<tr>
<td>data type</td>
<td>string</td>
</tr>
<tr>
<td>name of variable</td>
<td>var</td>
</tr>
</tbody>
</table>
get **statement**

**prompt**

**token**

**token-oriented input**

**quoted string**

**syntax error**

**execution error**

**run time error**

**constant**

**declaration of constant**

**assignment statement**

**comment**

**program name**