# Getting started with RAPTOR [Adapted from [**https://raptor.martincarlisle.com/**](https://raptor.martincarlisle.com/) by Dr. Wayne Brown]

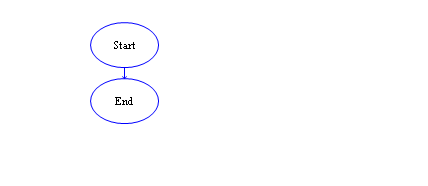
What is RAPTOR?

RAPTOR is a visual programming development environment based on flowcharts. A flowchart is a collection of connected graphic symbols, where each symbol represents a specific type of instruction to be executed. The connections between symbols determine the order in which instructions are executed. These ideas will become clearer as you use RAPTOR to solve problems.

After you open RAPTOR, you will see the following two screens: (1) the main screen below on the left which contains the *Work*, *Watch* and *Symbols* areas and (2) the *MasterConsole* screen on the right which contains the *Output* area.

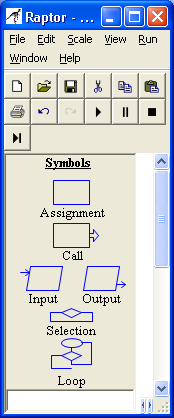
|  |
| --- |
|  |
|  |

When you first open RAPTOR your *MasterConsole* may be shrunk or hidden behind the main screen and you may have to click it to reveal it. You can resize the screens as you wish; it’s most convenient to view them side-by-side



RAPTOR Program Structure

A RAPTOR program is a set of connected symbols that represent actions to be performed. The arrows that connect the symbols determine the order in which the actions are performed. When executing a RAPTOR program, you begin at the Start symbol and follow the arrows to execute the program. A RAPTOR program stops executing when the End symbol is reached. The smallest RAPTOR program (which does nothing) is depicted at the right. By placing additional RAPTOR statements/symbols between the **Start** and **End** symbols you can create meaningful RAPTOR programs.



RAPTOR Statements/Symbols

RAPTOR has six (6) basic symbols, where each symbol represents a unique type of instruction. The basic symbols are shown at the right. The top four statement types, **Assignment**, **Call**, **Input**, and **Output**, are explained in this reading; the bottom two types, **Selection** and **Loops**, will be explained later on.

The typical computer program has three basic components:

* INPUT – get the data values that are needed to accomplish the task.
* PROCESSING – manipulate the data values to accomplish the task.
* OUTPUT – display (or save) the values which provide a solution to the task.

These three components have a direct correlation to RAPTOR instructions as shown in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Symbol** | **Name** | **Description** |
| INPUT |  | *input* statement | Allow the user to enter data. Each data value is stored in a ***variable***. |
| PROCESSING |  | *assignment* statement | Change the value of a ***variable*** using some type of mathematical calculation. |
| PROCESSING |  | *procedure* *call* | Execute a group of instructions defined in the named procedure. In some cases some of the procedure arguments (i.e., ***variables***) will be changed by the procedure's instructions. |
| OUTPUT |  | *output* statement | Display (or save to a file) the value of a ***variable***. |

The common thread among these four instructions is that they all do something to variables! To understand how to develop algorithms into working computer programs, you must understand the concept of a variable. Please study the next section carefully!

RAPTOR Variables

*Variables* are computer memory locations that hold a data value. At any given time, a variable can only hold a single value. However, the value of a variable can vary (change) as a program executes. That's why we call them "variables"! As an example, study the following table that traces the value of a variable called X.

|  |  |  |
| --- | --- | --- |
| **Description** | **Value of X** | **Program** |
| When the program begins, no variables exist. In RAPTOR, variables are automatically created when they are first used in a statement. | Undefined |  |
| The first assignment statement, X←32, assigns the data value 32 to the variable X. | 32 |
| The next assignment statement, X←X+1, retrieves the current value of X, which is 32, adds 1 to it, and puts the result, 33, in variable X. | 33 |
| The next assignment statement, X←X\*2, retrieves the current value of X, 33, multiplies it by 2, and puts the result, 66, in the variable X. | 66 |

During the execution of the previous example program, the variable X stored three distinct values. Please note that the order of statements in a program is very important. If you re-ordered these three assignment statements, the values stored into X would be different.

A variable can have its value set (or changed) in one of three ways:

* By the value entered from an *input* statement.
* By the value calculated from an equation in an *assignment* statement.
* By a return value from a *procedure call* (more on this later).

It is variables, and their changing data values, that enable a program to act differently every time it is executed.

All variables should be given meaningful and descriptive names by the programmer. Variable names should relate to the purpose the variable serves in your program. A variable name must start with a letter and can contain only letters, numerical digits, and underscores (but no spaces or other special characters). If a variable name contains multiple "words," the name is more "readable" if each word is separated by an underscore character or starts with a capital letter. The table below shows some examples of good, poor, and illegal variable names.

|  |  |  |
| --- | --- | --- |
| **Good variable names** | **Poor variable names** | **Illegal variable names** |
| taxRate  tax\_rate  distance\_in\_miles  distanceInMiles  mpg | a (not descriptive)  milesperhour (hard to read)  my4to (not descriptive) | 4sale (does not start with a letter) sales tax (includes a space) sales$ (includes invalid character) |

IMPORTANT: If you give each value in a program a meaningful, descriptive variable name, it will help you think more clearly about the problem you are solving, and it will help you find errors in your program.

One way of understanding the purpose of variables is to think of them as a means to communicate information between one part of a program and another. By using the same variable name in different parts of your program you are using the value that is stored at that location in different parts of your program. Think of the variable as a place holder or storage area for values between each use in your program computations.

When a RAPTOR program begins execution, no variables exist. The first time RAPTOR encounters a new variable name, it automatically creates a new memory location and associates this variable name with the new memory. The variable will exist from that point in the program execution until the program terminates. When a new variable is created, its initial value determines whether the variable will store numerical data or string (i.e. textual) data. This is called the variable's data type. A variable's data type cannot change during the execution of a program. In summary, variables are automatically created by RAPTOR and can hold either:

* Numbers e.g., 12, 567, -4, 3.1415, 0.000371, or
* Strings e.g., “Hello, how are you?”, “James Bond”, “The value of x is ”

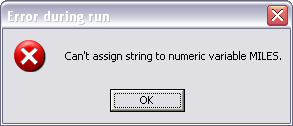
Below you’ll see some common errors when using variables:

Error 1: "*Variable ??? does not have a value*". There are two common reasons for this error:

1) The variable has not been given a value. 2) The variable name was misspelled.

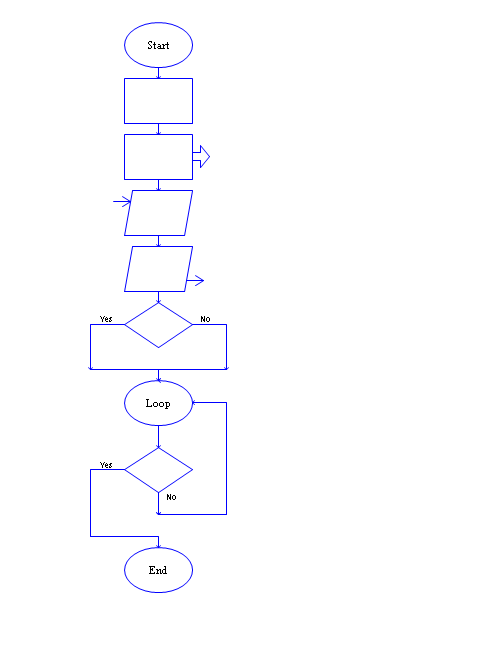


Error 2: "*Can't assign string to numeric variable ???*". This error will occur if your statements attempt to change the data type of a variable.



RAPTOR Statements/Symbols

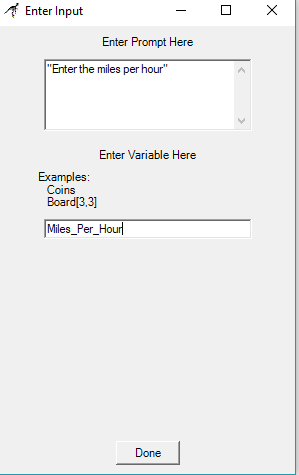
The following sections provide details about each of the four basic statements: Input, Assignment, Call, and Output.



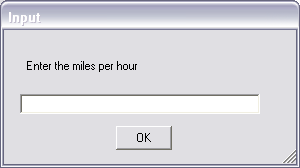
Input Statement/Symbol

An input statement/symbol allows the user of a program to enter a data value into a program variable during program execution. It is important that a user knows exactly what type of value is expected for input. Therefore, when you define an input statement you specify a string of text that will be the *prompt* that describes the required input. The prompt should be as explicit as possible. If the expected value needs to be in particular units (e.g., feet, meters, or miles) you should mention the units in the prompt.

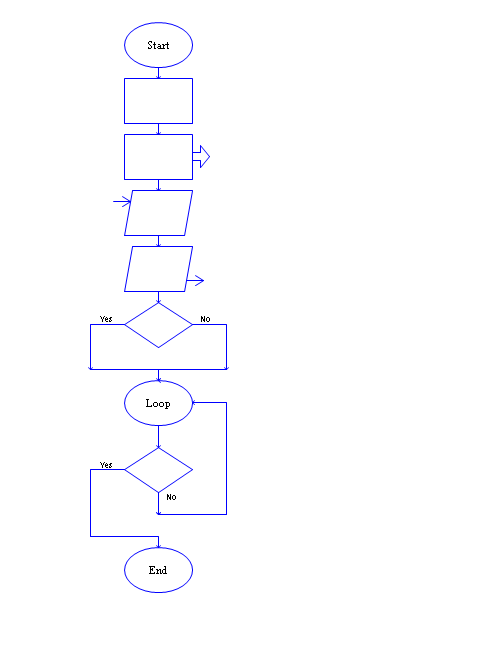
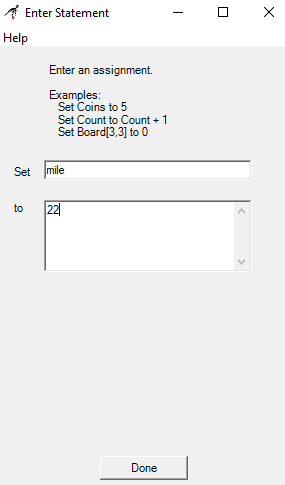
When you define an input statement, you must specify two things: a prompt and the variable that will be assigned the value enter by the user at run-time. This is shown below on the left.



At run-time, an input statement will display an input dialog box, an example of which is shown to the right. After a user enters a value and hits the enter key (or clicks OK), the value entered by the user is assigned to the input statement's variable.



**Assignment Statement/Symbol**



The assignment symbol is used to perform a computation and then store the results in a variable. The definition of an assignment statement is performed using the dialog box shown on the right. The variable to be assigned a value is entering into the "Set" field, and the computation to perform is enter into the "to" field. The example on the right sets the value of the variable mile to 22.

An assignment statement is displayed inside its RAPTOR symbol using the syntax: Variable ← Expression

One assignment statement can only change the value of a single variable, that is, the variable on the left-hand side of the arrow. If this variable did not exist prior to the statement, a new variable is created. If this variable did exist prior to the statement, then its previous value is lost and its new value is based on the computation that is performed. No variables on the right-hand side of the arrow (i.e., the expression) are ever changed by the assignment statement.

**Expressions**

The expression (or computation) of an assignment statement can be any simple or complex equation that computes a single value. An expression is a combination of values (either constants or variables) and operators. Please carefully study the following rules for constructing valid expressions.

A computer can only perform one operation at a time. When an expression is computed, the operations of the equation are not executed from left to right in the order that you typed them in. Rather, the operations are performed based on a predefined "order of precedence." The order that operations are performed can make a radical difference in the value that is computed. For example, consider the following two examples:

x ← (3+9)/3 x ← 3+(9/3)

In the first case, the variable x is assigned a value of 4, whereas in the second case, the variable x is assigned the value of 6. As you can see from these examples, you can always explicitly control the order in which operations are performed by grouping values and operators in parenthesis. The exact "order of precedence" is

1. compute all functions, then
2. compute anything in parentheses, then
3. compute exponentiation (^, \*\*) i.e., raise one number to a power, then
4. compute multiplications and divisions, left to right, and finally
5. compute additions and subtractions, left to right.

An operator or function directs the computer to perform some computation on data. Operators are placed between the data being operated on (e.g. X/3) whereas functions use parentheses to indicate the data they are operating on (e.g. sqrt(4.7) ). When executed, operators and functions perform their computation and return their result.

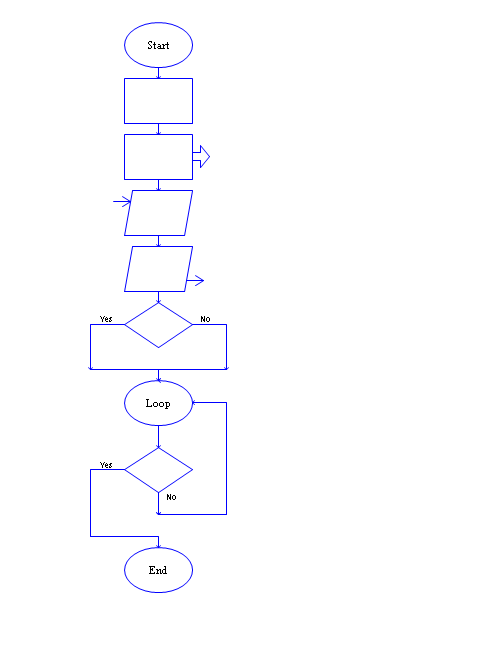
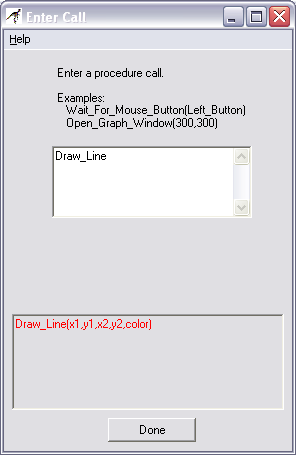
The following table describes some built-in operators and functions. Full details concerning these operators and functions can be found in the RAPTOR help screens.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| + | addition | 3+4 is 7 |
| - | subtraction | 3-4 is -1 |
| - | negation | -3 is a negative 3 |
| \* | multiplication | 3\*4 is 12 |
| / | division | 3/4 is 0.75 |
| ^ or \*\* | exponentiation, raise a number to a power | 3^4 or 3\*\*4 is 3\*3\*3\*3=81 |
| rem or mod | remainder (what is left over) when the right operand divides the left operand | 10 rem 3 is 1 and 10 mod 4 is 2 |
| sqrt | square root | sqrt(4) is 2 |
| abs | absolute value | abs(-9) is 9 |
| ceiling | rounds up to a whole number | ceiling(3.14159) is 4 |
| floor | rounds down to a whole number | floor(9.82) is 9 |
| sin | trig sin(angle\_in\_radians) | sin(pi/6) is 0.5 |
| cos | trig cos(angle\_in\_radians) | cos(pi/3) is 0.5 |
| random | generates a random value in the range [1.0, 0.0) | random \* 100 is some value between 0 and 99.9999 |
| Length\_of | returns the number of characters in a string variable | Example ← "Sell now" Length\_of(Example) is 8 |

The result of evaluating of an expression in an assignment statement must be either a single number or a single string of text. Most of your expressions will compute numbers, but you can also perform simple text manipulation by using a plus sign (+) to join two or more strings of text into a single string. You can also join numerical values with strings to create a single string. The following example assignment statements demonstrate string manipulation.

Full\_name ← "Joe " + "Alexander " + "Smith"

Answer ← "The average is " + (Total / Number)



Procedure Call Statement/Symbol

A procedure is a named collection of programming statements that accomplish a task. Calling a procedure suspends execution of your program, executes the instructions in the called procedure, and then resumes executing your program at the next statement. You need to know two things to correctly use a procedure: 1) the procedure's name and 2) the data values that the procedure needs to do its work, which are called arguments.

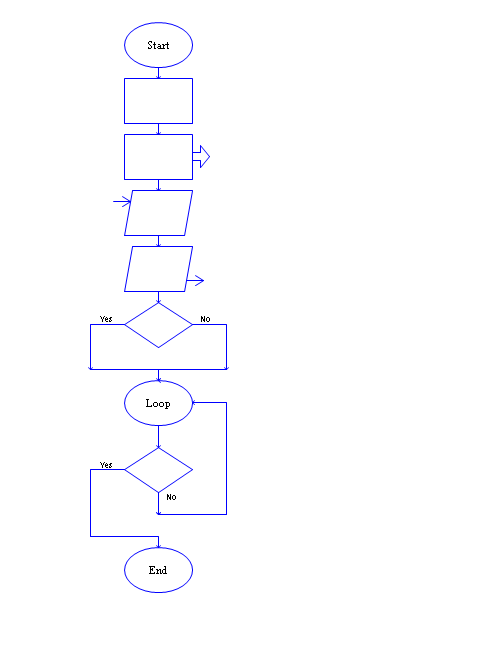
RAPTOR attempts to minimize the number of function names you need to memorize by displaying any procedure name that partially matches what you type into the "Enter Call" window. For example, after entering the single letter "d," the lower portion of the window will list all built-in procedures that start with the letter "d". The list also reminds you of each procedure's required arguments. In the example to the right, the lower box is telling you that the "Draw\_Line" procedure needs 5 data values (or parameters): the x and y coordinates of the starting location of the line, (x1, y1), the x and y coordinates of the ending location of the line, (x2, y2), and the line's color. The order of the argument values must match the arguments defined by the procedure. For example, Draw\_Line(Blue,3,5,100,200) would generate an error because the color of the line must be the last argument value in the argument list.

When a procedure call is displayed in your RAPTOR program you can see the procedure's name and the argument values that will be sent to the procedure when it is called. For example, when the first procedure call on the right is executed it will draw a red line from the point (1,1) to the point (100,200). The second procedure call will also draw a line, but since the arguments are variables, the exact location of the line will not be known until the program executes and all the argument variables have a value.

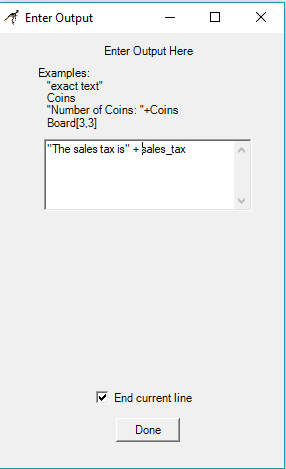


RAPTOR defines too many built-in procedures to describe them all here. You can find documentation on all built-in procedures in RAPTOR's help screens. In addition, your instructor will introduce relevant procedures as we tackle various problem-solving tasks in the coming lessons.

Output Statement/Symbol



In RAPTOR, an output statement displays a value to the *MasterConsole* window when it is executed. When you define an output statement, the "Enter Output" dialog box asks you to specify two things:



* What is the text or expression to display?
* Should the output be terminated by a new line character?

The example output statement on the right will display the text, "*The sales tax is*" followed by the value of variable *sales\_tax* on the output window and terminate the text with a new line. Since the "End current line" is checked, any future output will start on a new line below the displayed text.

|  |  |
| --- | --- |
| **Non-user-friendly output** | **User-friendly output** |
|  |  |
| Example output: 2.5678 | Example output: Area = 2.5678 square inches |

Your instructor (or a homework assignment) will often say “Display the results in a user-friendly manner”. This means you should display some explanatory text explaining any numbers that are output to the MasterConsole window. An example of "non-user-friendly output" and "user-friendly output" is shown below.

What you have hopefully learned…

* The basic structure and types of statements of a RAPTOR program.
* What a variable is and how variables are used.
* How to write computations (i.e., expressions) that calculate desired values.
* How to get input values into a program and how to display output values.

Reading Self-Check

|  |  |
| --- | --- |
| 1. Label the following RAPTOR identifiers as (**G**) good, (**P**) poor, or (**I**) Illegal. If illegal then explain why. | |
| \_\_\_\_ 1) This\_Is\_A\_Test  \_\_\_\_ 2) U\_2  \_\_\_\_ 3) Money$  \_\_\_\_ 4) Thisisanawfullylongidentifiername  \_\_\_\_ 5) Mickey-Mouse | \_\_\_\_ 6) 365\_Days  \_\_\_\_ 7) Variable  \_\_\_\_ 8) Is This Identifier Legal  \_\_\_\_ 9) Why\_Isn’t\_This\_One\_Legal |
| 1. Calculate the result of the following expressions (or indicate if the expression contains errors) | |
| \_\_\_\_\_\_\_\_\_\_ 1) 46 / 2  \_\_\_\_\_\_\_\_\_\_ 2) 4 + 6 \* 2  \_\_\_\_\_\_\_\_\_\_ 3) 12 / 3 / 2  \_\_\_\_\_\_\_\_\_\_ 4) (4 + 2) / (5 – 3) + 2  \_\_\_\_\_\_\_\_\_\_ 5) 46 / 3  \_\_\_\_\_\_\_\_\_\_ 6) 46 **rem** 3 | \_\_\_\_\_\_\_\_\_\_ 7) 3(4 + 3)  \_\_\_\_\_\_\_\_\_\_ 8) 6 \*\* **sqrt**(4)  \_\_\_\_\_\_\_\_\_\_ 9) 77 + -11  \_\_\_\_\_\_\_\_\_\_ 10) **floor**(77.25)  \_\_\_\_\_\_\_\_\_\_ 11) **floor**(-77.25) |