Chapter 6 Recap
===============

Please note that the following are mainly points with some brief explanations to add to the notes you are compiling. The recap is an opportunity to revisit the work we have spoken about during class.

==

Overview

==

Chapter highlights:
- Arrays
-- One-Dimensional Arrays
- Using a for loop to process arrays
- How an array can replace nested decisions
- Using constants with arrays
- Searching an array for an exact match
- Using parallel arrays
- Searching an array for a range match

* Extra
-- Two-Dimensional Arrays

==

===

Arrays

===

-- So far the only data structure we have learnt to deal with is a single variable. Data is stored in a single memory location. The value can be changed when programmed to and we would declare as many variables as needed for a single programming solution.

-- Up to this point in our journey of learning programming, it has been, the manipulation of data to produce knowledge using variables and the 3 logical structures. We have built our knowledgebase on these topics. Continuously building on previous work, mixing different logical structures where needed, all the while processing the values inside variables.

-- An **Array** is another, new data structure we are being introduced to.

-- With the array we can store several values in the same variable, providing us with an opportunity to decrease processing time of a program, now when dealing with a list of values.

-- When more than one memory location is designated for a single variable, it is an array.

- Example
- When a programmer wishes to store many values of data of the same kind.
    - a list of ages,
    - a list of temperatures recorded for the month in a city,
    - a list of students in a class group,
    - a marksheet containing the marks for a test, assignment, exam, etc.

-- We can use and consider multiple arrays in a single program. One of the thoughts around this might be that in the story, they are related and by accessing one array, it can point to data in another array. (*This concept will be discussed later in the section on parallel arrays*)

-- We may also use an array to count the number of times a value appears, accessing such information with single variables is possible, but might end up being a long process, whereas arrays would be more efficient.

If we are tasked with the result of programming/producing a single value, like calculating the student's grade, or an average, then using an array doesn't really make sense here. BUT if we were required to store the marks for a class group and then process the grades, determine the highest and lowest marks, then using an array is an opportunity we would consider.

-- When a programmer is required to declare an array variable, they are required to dimension the array (also "dimension the variable"). At this point in time, we are creating and processing one-dimensional arrays (1D arrays). This means that there is only a single reference (index) value that is needed to refer to the element/cell in the array.

We are also learning that arrays are static. This means that once the computer is told how many locations to reserve for the array, then that number cannot be changed unless the declaration is changed before executing the program again (not during the processing of a program). This number of elements is usually an unnamed constant value (for example, number of cells are stated in square brackets - [10]) and there are times where we use a named constant (for example, a constant variable, [intSIZE]. This variable would need a value before being used in the declaration of the array).

In the past, programmers were allowed to change the size of an array during processing. But technically this is not allowed for quite some time now in the various programming languages we use. When we processed an array, we had to use a little extra programming to figure out the size so that processing of the data flowed smoothly. Since we could not change the size of a static array, then we had to ask what could be done?

Many of the programming languages now give us the opportunity to access an interesting concept known as "collections" You are going to learn about this in future classes. Given the opportunity to declare a variable that is a collection, we can load as many values (into a set) as we would like during the processing of the program, including adding or subtracting elements during further processing as well. These collections can be accessed in a variety of ways including, in certain cases, using indexes. So, the concept of a dynamic array exists, just that it is a different data structure, like an array, but not an array.

-- **Element**

- Each memory location is called an element (sometimes also referred to as the "cell")

- Each element is referenced by a position number. This reference number is relative to the location of elements in the array. The reference number is referred to by many names; index, element number, or subscript and is always contained within parentheses [] (square brackets).

```
**code  - intNumber[0]
```

- These subscripts may be a constant, a variable or a mathematical expression (*** and is always of an integer data type)

```
**code  - strStudents[5]
        - strStudents[intIndex]
        - strStudents[intNumberOfStudents - 1]
```

***highlighted again in the next section*

-- **Base-Zero system**

- Computers are mostly zero-based, for counting purposes, and many of the programming languages you will use in the future are zero-based as well. This means that when we declare arrays, the first index is always 0. And therefore, a cell with an index of 1, would be the second cell in the array (and so on).

===

One-Dimensional Arrays

===

-- The simplest array structure is the one-dimensional array. One column (or row) of memory locations visualized below:

intNumbers

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	2	3	4	5	6	7	8	9	10

intNum

[0]	1
[1]	2
[2]	3
[3]	4
[4]	5
[5]	6
[6]	7
[7]	8
[8]	9
[9]	10

-- The number in the parentheses is a reference number only (always an integer) and can be a constant, a variable, or an expression.

- intNum[5]
  - the 5 in the brackets represents the 6th element - since we are counting from 0 (0 being the first element in the array) - 5 is the index/subscript

- intNum[intCount]
  - The variable intCount would most likely contain a value based on some programming code being executed and is then used to reference a specific cell in the array

- fltDiscount[intFRIDAY]
  - intFRIDAY is a constant variable
  - If we are counting from Sunday (0) then Friday would contain the value (5).
  - using the constant variable would be better understood as retrieving the discount (from the fltDiscount array) for Friday instead of the programmer reading that the discount is from cell 5 (fltDiscount[5], which is the 6th cell)

- fltSales[intIndex + 1]
  - intIndex + 1 being the expression that will be evaluated first before accessing the cell that you are expecting the access (remember that the mathematical expression must always result in an integer value)

```
-- The declaration in pseudocode
                // one-dimensional array of 10 cells
        **code  num intNumbers[10]

                // one-dimensional array of 5 cells with values loaded at declaration
        **code  num intMarks[5] = 55, 98, 76, 69, 70
```

** *Parallel Arrays - two or more arrays in which values in the same elements relate to each other.*

- **Example - 3 1D-Arrays**
- first one contains the names of cities,
- second one contains the lowest recorded temperature for the month and
- third contains the highest recorded temperature for the month.

Referencing the cells using the same index number in each of the 3 arrays will present information that is related - the name of the city and the 2 temperatures recorded.

	strCities		intLowTemp		intHighTemp
0	Cape Town	0	1	0	19
1	Durban	1	0	1	18
2	Johannesburg	2	1	2	16
3	Pretoria	3	4	3	10
4	East London	4	-3	4	20
5	Kimberley	5	5	5	12
6	Bloemfontein	6	-2	6	12
7	Polokwane	7	2	7	18
8	Upington	8	6	8	15
9	George	9	8	9	12
10	Cradock	10	3	10	17
11	Richards Bay	11	-5	11	18
12	Thohoyandou	12	-2	12	15
13	Mossel Bay	13	6	13	18
14	Gqeberha	14	-4	14	17

Using a for loop to process arrays

-- Using a for..loop to iterate through the cells in arrays is quite an efficient choice for static arrays. Simply because the number of elements is known and the for..loop is known for executing an exact number of times (given the starting and ending number)

- The for..loop is able to deal with the following:
    - -- initialize the loop control variable (likely to be used as the index for the array)
    - -- compare it to the limit (the size of the array)
    - -- alter / increment the variable (iterate through the elements of the array)

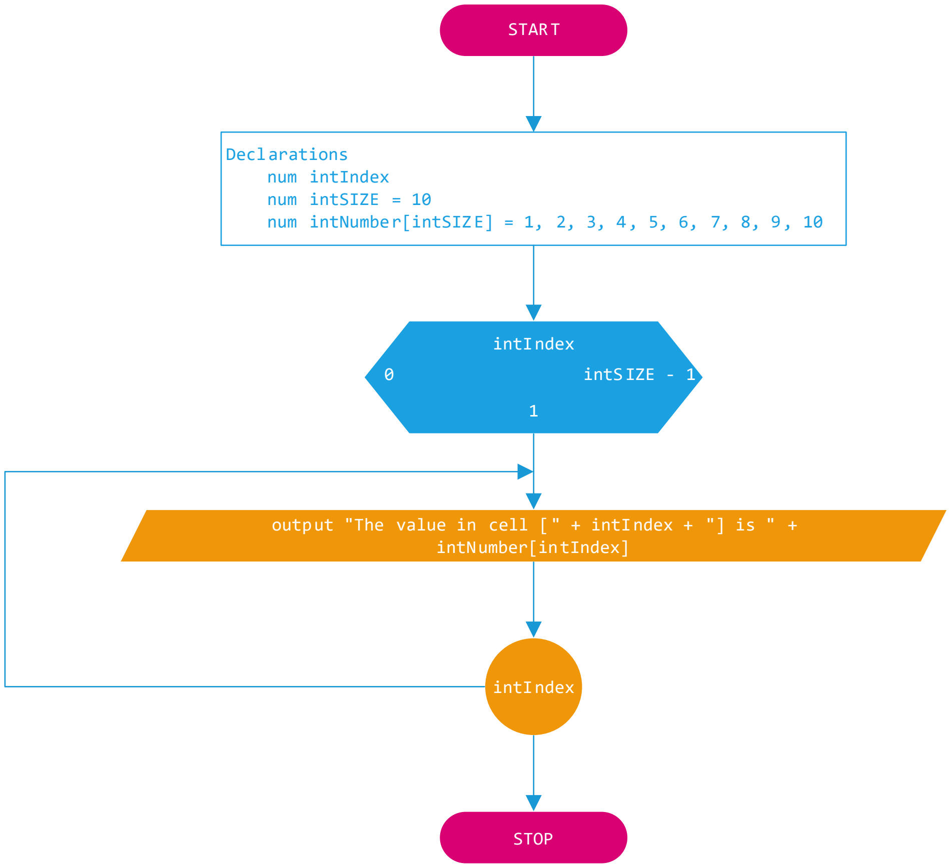
-- The loop stays within the array bounds (we need to understand this for the future, since there is a run-time error that can occur if we attempt to access a cell that does not exist)

-- The loop control variable controls the processing of the elements, and is used as the index (subscript) of the array

- (since the array indexes start from 0 - the highest number the loop can go is one less than the size of the array)

```
0. Start
1. Declarations
        num intIndex
        num intSIZE = 10
        num intNumber[intSIZE] = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

2.      for intIndex = 0 to intSIZE - 1 Step 1
                output "The value in cell [" + intIndex + "] is " + intNumber[intIndex]
        endfor

3. Stop
```

- Take note of the *'ending number'* in the for..loop.
- The variable intIndex is going to count from *0 to 9 (intSIZE - 1)*
- Besides counting, the intIndex variable is also being used to reference each element in the array.



```
Declarations
    num intIndex
    num intSIZE = 10
    num intNumber[intSIZE] = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
output "The value in cell [" + intIndex + "] is " +
intNumber[intIndex]
```

Entering Data in a 1D array

-- Besides loading an array at declaration (the example code in the previous section), we also have the opportunity to enter data into an array, (also known as filling and/or populating an array) with values entered by the user. Since we know the number of elements for the array, then using the for..loop is the best choice.

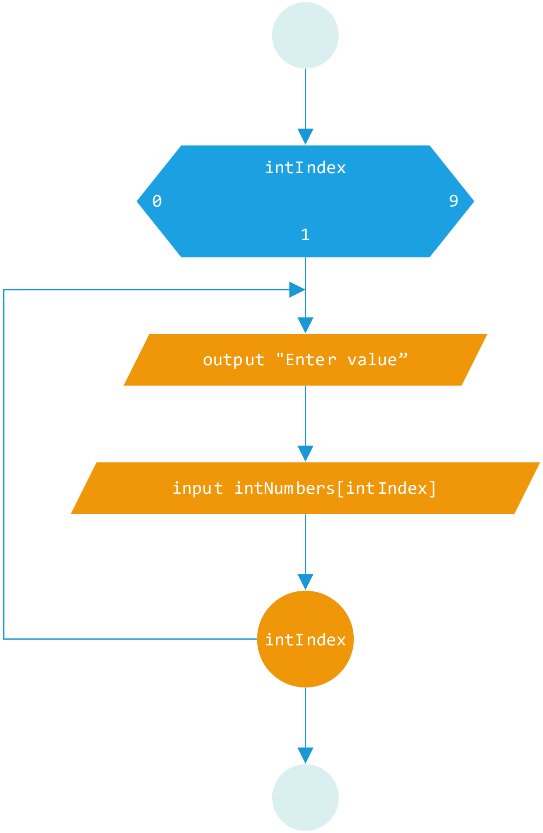
** Remember the brief discussion around dynamic arrays, if we do not know the number of elements, then we would consider using the while..loop or the do..while loop, but this is a story for another time.

** Also remember the notes from the previous chapter - we could use the while..loop or the do..while loop to iterate through the elements of an array, but that would mean that we need to program and take care of the counter/index manually. Not difficult to do, again, the for..loop is just nicer.

**code (*using a for..loop to iterate through 10 cells in a declared array*)

```
for intIndex = 0 to 9 Step 1
        output "Enter value "
        input intNumbers[intIndex]
endfor
```

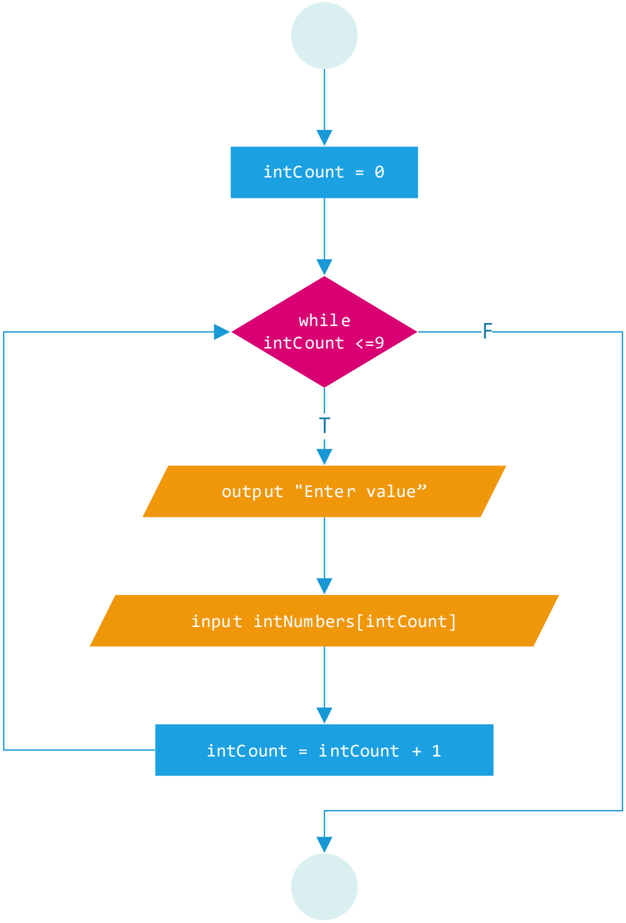
**flowchart



**code (using a while loop - knowing the size of the 1D array (10 cells))

```
        intCount = 0

        while intCount <= 9
                output "Enter value "
                input intNumbers[intCount]
                intCount = intCount + 1
        endwhile
```

**flowchart

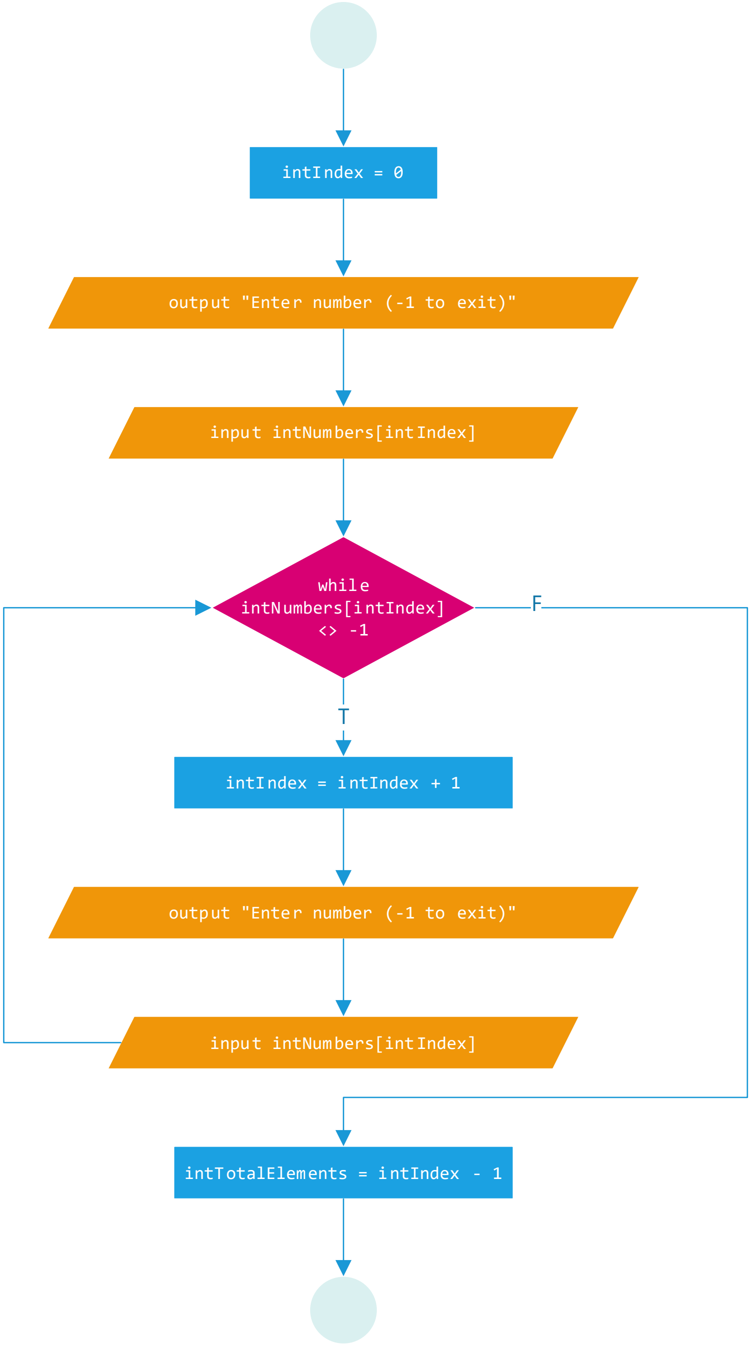


**code

- if it is a dynamic array (we do not know how many cells are in the array)
- *please note that this concept will not be discussed further, since we will be dealing with static arrays in all our examples*

```
        intIndex = 0

        output "Enter number (-1 to exit)"
        input intNumbers[intIndex]

        while intNumbers[intIndex] <> -1
                intIndex = intIndex + 1
                output "Enter number (-1 to exit)"
                input intNumbers[intIndex]
        endwhile

        intTotalElements = intIndex - 1
```

**flowchart


intIndex = 0
output "Enter number (-1 to exit)"
input intNumbers[intIndex]
while intNumbers[intIndex] <> -1
F
T
intIndex = intIndex + 1
output "Enter number (-1 to exit)"
input intNumbers[intIndex]
intTotalElements = intIndex - 1

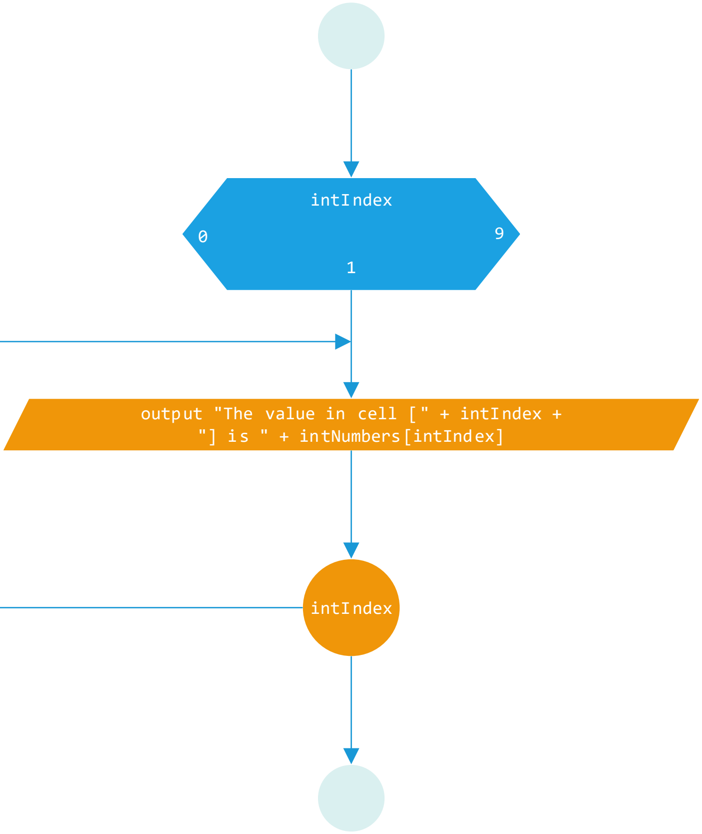
Printing Data from a 1D array

-- After processing the data in the array and producing the required information, you are likely requested to print the values from the array, presenting the data along with any new processed information (presenting a report). The following code uses the for..loop to iterate through the elements of the array.

** code

```
        for intIndex = 0 to 9 Step 1
                output "The value in cell [" + intIndex + "] is " + intNumbers[intIndex]
        endfor
```

**flowchart


intIndex
0
9
1
output "The value in cell [" + intIndex + "] is " + intNumbers[intIndex]
intIndex

Accumulating a set of values in a 1D array (incl. determining highest and lowest values)

-- Accumulating the values in (a numeric) array is a common task that we would execute (along with determining the highest and lowest values). The (accumulating) variable required to store these results is always initialized.

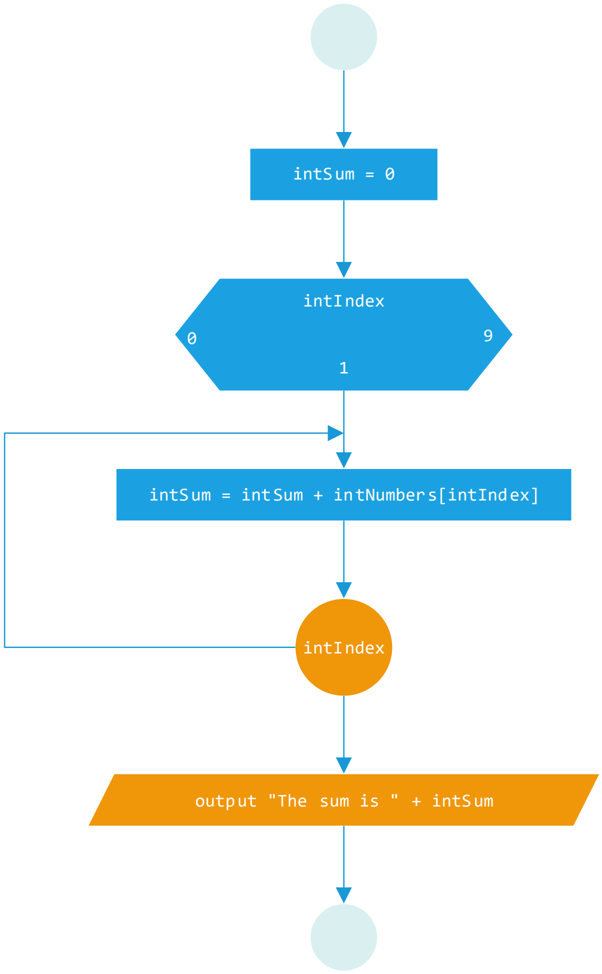
-- The value of the accumulating variable is set to 0.

-- When there is a different task, like determining the highest and lowest values in an array, then the initialized variables will be different depending on the data and the story we are presented with. High is some low value that does not exist within the range of the story and low is some high value that does not exist again, within the range of the story.

**code

```
        intSum = 0
        for intIndex = 0 to 9 Step 1
                intSum = intSum + intNumbers[intIndex]   // accumulation instruction
        endfor
        output "The sum is " + intSum
```

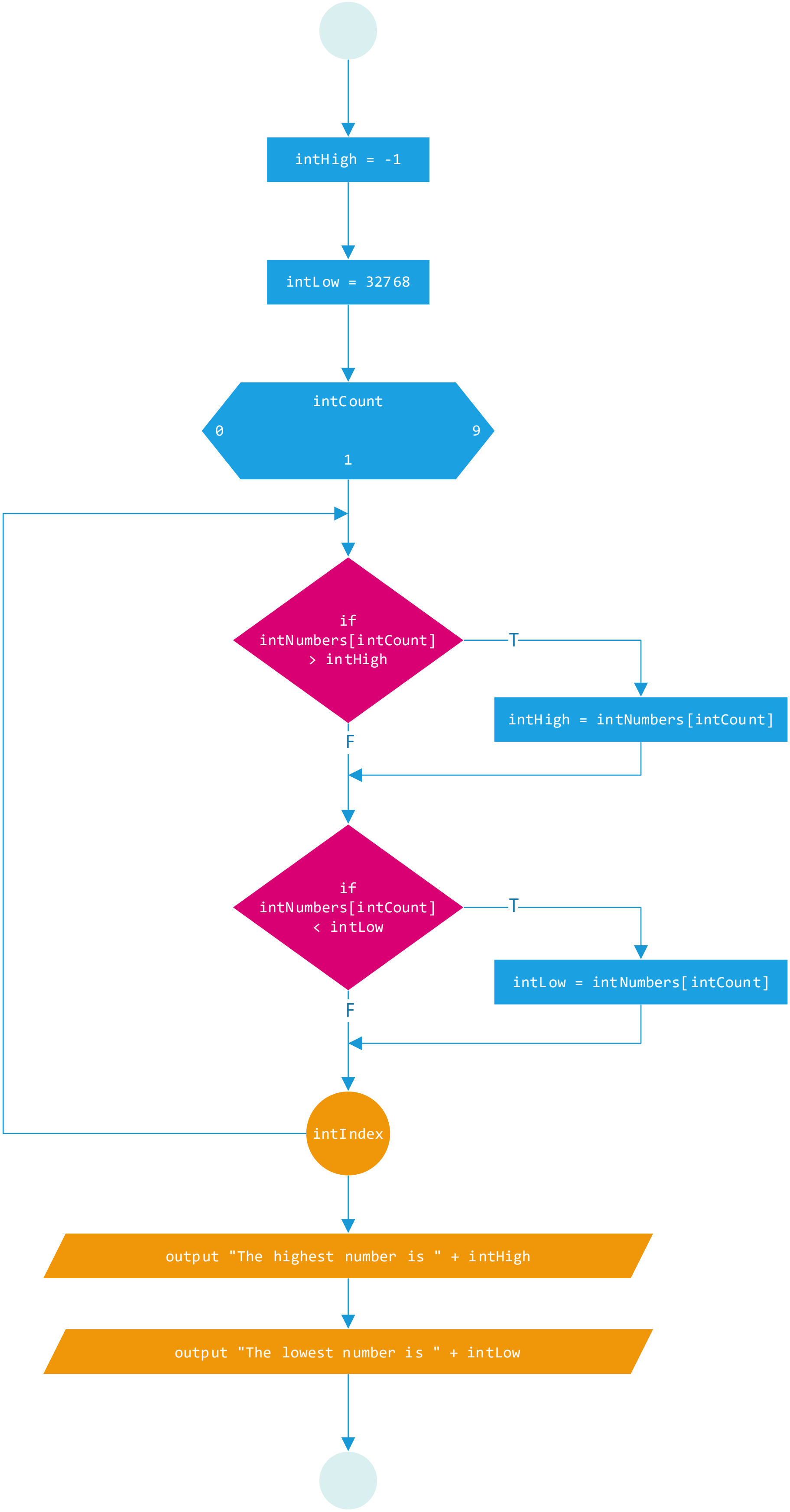
**flowchart


intSum = 0
intIndex
0
9
1
intSum = intSum + intNumbers[intIndex]
intIndex
output "The sum is " + intSum

**code

```
        intHigh = -1
        intLow = 32768

        for intCount = 0 to 9 Step 1
                if intNumbers[intCount] > intHigh then
                        intHigh = intNumbers[intCount]
                endif

                if intNumbers[intCount] < intLow then
                        intLow = intNumbers[intCount]
                endif
        endfor
        output "The highest number is " + intHigh
        output "The lowest number is " + intLow
```

**flowchart

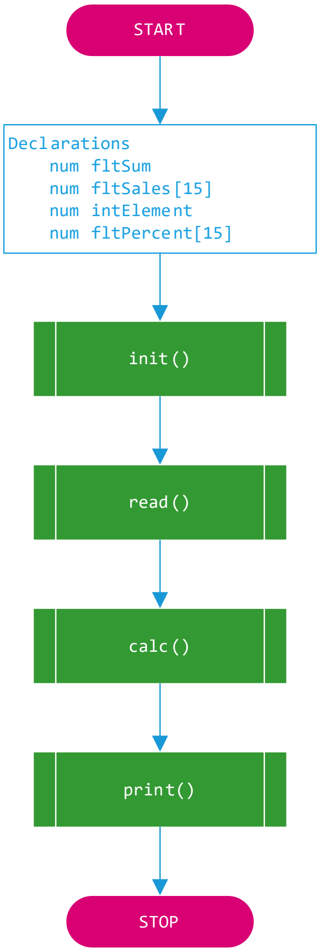

intHigh = -1
intLow = 32768
intCount
0
9
1
if intNumbers[intCount] > intHigh
T
intHigh = intNumbers[intCount]
F
if intNumbers[intCount] < intLow
T
intLow = intNumbers[intCount]
F
intIndex
output "The highest number is " + intHigh
output "The lowest number is " + intLow

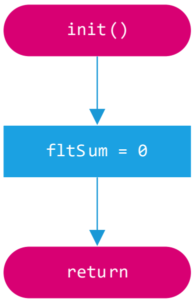
```
*******************************************************************************************************
Take note how all this coding is done separately. This helps you realise something important, unlike a
single variable that would have changed through each iteration of the loop and effectively losing
previously entered values, the array allows for the values to remain "saved" for a slightly longer
period of time, effectively as long as the program is running. Once the program ends, the values in
the array are wiped from memory. Future programming techniques will teach you a more permanent form of
storage, files and then will come the time to learn databases as well
*******************************************************************************************************
```

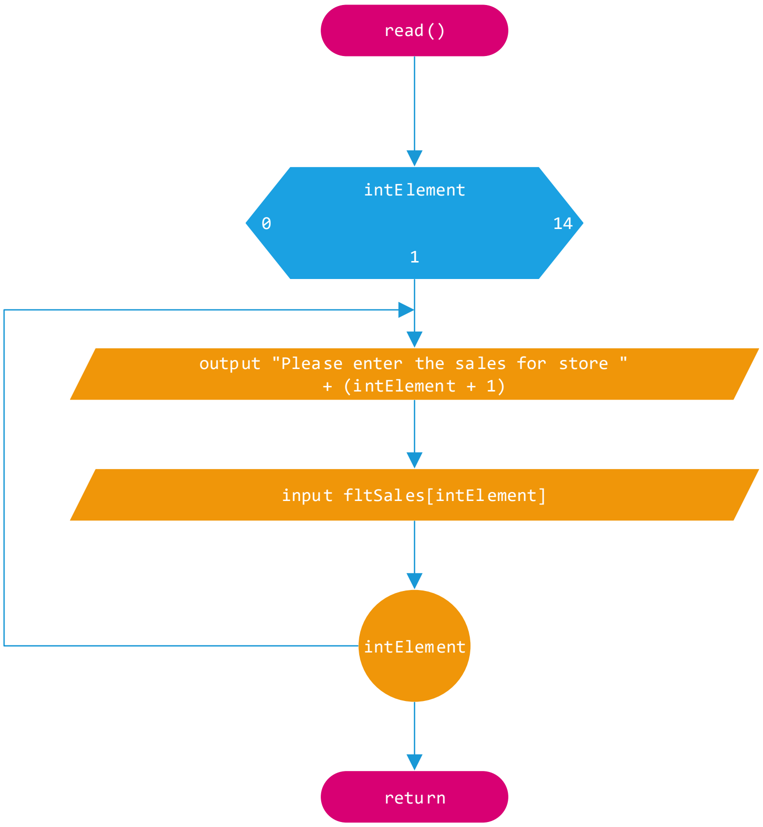
```
-----------------------------------------------------------------------------------------------------
-- Example
        -- OG Flowers Inc. had decided to increase the number of locations for their flower
        arrangement and delivery service a few months back. Since then, the business has increased
        their sales month on month.

        The owner would like to know each store's percentage of total sales of the company. There are
        a total of 15 stores altogether.
-----------------------------------------------------------------------------------------------------
        Given Data
                - 15 Stores
                - Sales per store
        Required Results
                - Percent of total sales per store
        Processing Required
                - Percent (per store) = store sales / total sales (* 100)
-----------------------------------------------------------------------------------------------------
```

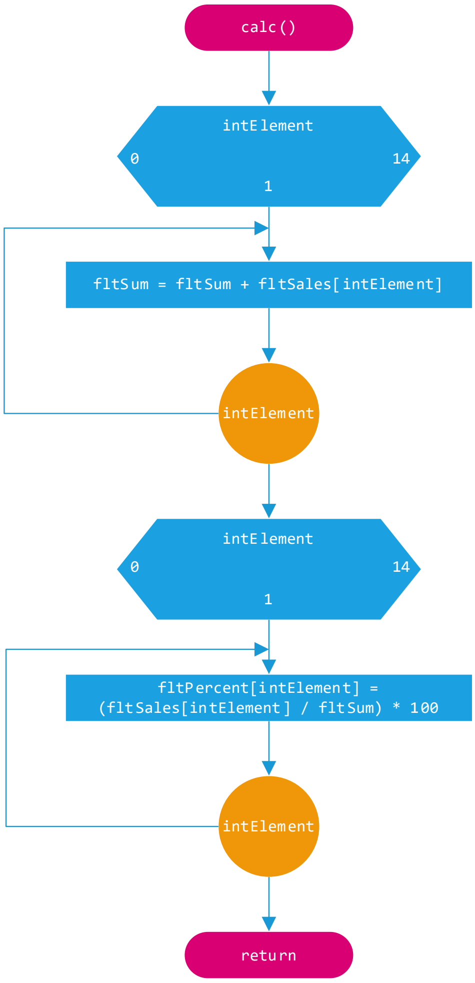
	fltSales		fltPercent
0	10000	0	20
1		1	
2		2	
3		3	
4		4	
5		5	
6		6	
7		7	
8		8	
9		9	
10		10	
11		11	
12		12	
13		13	
14		14	

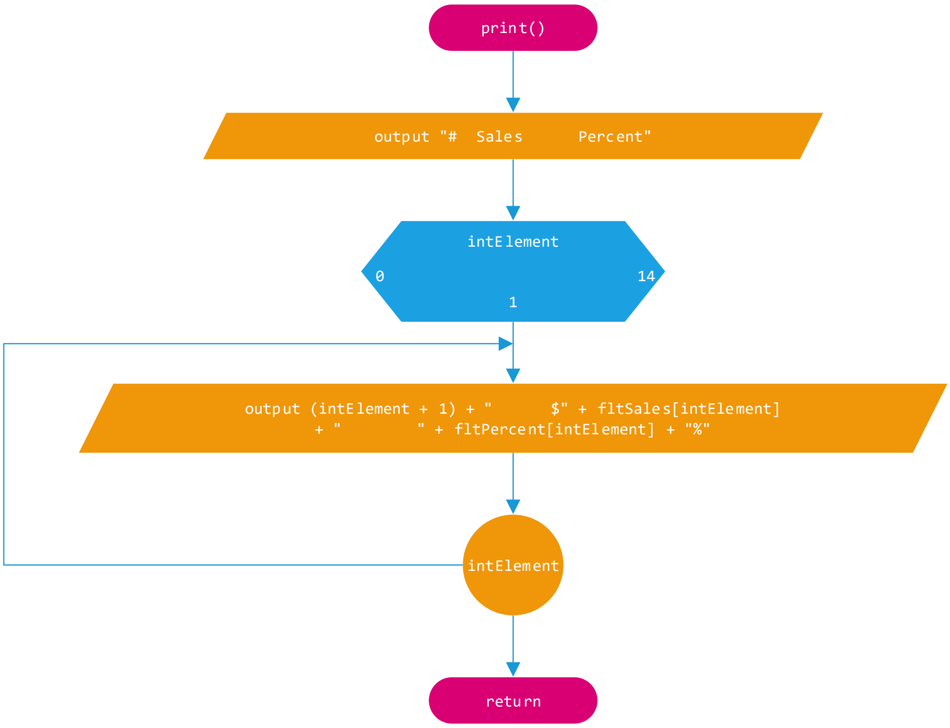
```
----------------------------------------------------------------------------------------------------
        **algorithm/pseudocode

        0. Start
        1. Declarations
                num fltSum
                num fltSales[15]
                num intElement
                num fltPercent[15]

        2.      init()
        3.      read()
        4.      calc()
        5.      print()

        6. Stop
----------------------------------------------------------------------------------------------------
        0. init()
        1.      fltSum = 0
        2. return
----------------------------------------------------------------------------------------------------
        0. read()
        1.      for intElement = 0 to 14 Step 1
                        output "Please enter the sales for store " + (intElement + 1)
                        input fltSales[intElement]
                endfor
        2. return
----------------------------------------------------------------------------------------------------
        0. calc()
        1.      for intElement = 0 to 14 Step 1
                        fltSum = fltSum + fltSales[intElement]
                endfor

        2.      for intElement = 0 to 14 Step 1
                        fltPercent[intElement] = (fltSales[intElement] / fltSum) * 100
                endfor

        3. return
----------------------------------------------------------------------------------------------------
        0. print()
        1.      output "#       Sales    Percent"
        2.      for intElement = 0 to 14 Step 1
                        output (intElement + 1) + "      $" + fltSales[intElement] + "    " +
                                        fltPercent[intElement] + "%"
                endfor
        3. return
----------------------------------------------------------------------------------------------------
```


START
Declarations
num fltSum
num fltSales[15]
num intElement
num fltPercent[15]
init()
read()
calc()
print()
STOP


init()
fltSum = 0
return


read()
intElement
0
14
1
output "Please enter the sales for store " + (intElement + 1)
input fltSales[intElement]
intElement
return


calc()
intElement
0
14
1
fltSum = fltSum + fltSales[intElement]
intElement
intElement
0
14
1
fltPercent[intElement] =
(fltSales[intElement] / fltSum) * 100
intElement
return


print()
output "# Sales   Percent"
intElement
0
14
1
output (intElement + 1) + " $" + fltSales[intElement]
+ "   " + fltPercent[intElement] + "%"
intElement
return

==

How an array can replace nested decisions

==

-- The following is an example and an opportunity to realise how past programming ideas can now change and how they are adapted to use arrays so that the coding & processing of data is more efficient.

-- Consider the following example:

- You are tasked with keeping count on the number of employees who have between 0 and 5 dependents (including 0 and 5)
- Once the information is collected, the application produces (prints) a report on the counts for the dependent categories from 0 to 5.

* *We assume that the application is only looking for the count from 0 to 5 dependents & no more.*

```
        **pseudocode
        0. Start
        1. Declarations
                num intDependents
                num intCount0               // count variable for 0 dependents
                num intCount1               // count variable for 1 dependent
                num intCount2               // count variable for 2 dependents
                num intCount3               // count variable for 3 dependents
                num intCount4               // count variable for 4 dependents
                num intCount5               // count variable for 5 dependents

        2.      intCount0 = 0
        3.      intCount1 = 0
        4.      intCount2 = 0
        5.      intCount3 = 0
        6.      intCount4 = 0
        7.      intCount5 = 0

        8.      output "Enter the number of dependents (-1 to end processing)"
        9.      input intDependents
        10.     while intDependents <> -1
                        if intDependents = 0 then
                                intCount0 = intCount0 + 1
                        else
                                if intDependents = 1 then
                                        intCount1 = intCount1 + 1
                                else
                                        if intDependents = 2 then
                                                intCount2 = intCount2 + 1
                                        else
                                                if intDependents = 3 then
                                                        intCount3 = intCount3 + 1
                                                else
                                                        if intDependents = 4 then
                                                                intCount4 = intCount4 + 1
                                                        else
                                                                intCount5 = intCount5 + 1
                                                        endif
                                                endif
                                        endif
                                endif
                        endif
                        output "Enter the number of dependents (-1 to end processing)"
                        input intDependents
                endwhile

        11.     output "REPORT - EMPLOYEE - DEPENDENT COUNT"

        12.     output "Dependents 0 - count is " + intCount0
        13.     output "Dependents 1 - count is " + intCount1
        14.     output "Dependents 2 - count is " + intCount2
        15.     output "Dependents 3 - count is " + intCount3
        16.     output "Dependents 4 - count is " + intCount4
        17.     output "Dependents 5 - count is " + intCount5

        18. Stop
```

- An interesting result when we look back at the lessons learned from decision logic. There are 6 counting variables created. When the user enters the number of dependents that an employee has, the series of nested if..then statements (positive logic) will determine which counting variable to increment.

- The algorithm is understood and quite simple, but you would notice that there are few instructions that feel like they are duplicated, though they are needed. Given that we are learning about arrays, we could use it to reduce the number instructions being executed.

- Instead of the 6 count variables being declared, an array of 6 elements could be used.

- The 6 elements would have an index from 0 to 5. This index can & will serve a secondary purpose. Not only will it be used to as the index referencing the cell in the array that maintains the count for each of the number of dependents, but the index is now representing the number of dependents as well.

	0	1	2	3	4	5
intCount						

- We will increment each cell value depending on the number of dependents we read in.

- Consider a piece of this edited code from the nested if..then statements

```
if intDependents = 0 then
        intCount[0] = intCount[0] + 1
else
        if intDependents = 1 then
                intCount[1] = intCount[1] + 1
        else
                ...
```

- The condition is testing if the number of dependents inputted is equal to 0 (zero). If true then it will increment the value in cell 0 to keep track of the employee who stated they have zero dependents.

- The above if..then statement is welcomed, but we are still be expected to code the many nested if..then statements similar to the full example above, even though we have introduced the array into the code.

- Another edit - use the variable intDependents as the index for the array since the value of the index is the same as the number of dependents

```
if intDependents = 0 then
        intCount[intDependents] = intCount[intDependents] + 1
        ...
```

- The edit is interesting, using a variable for another purpose instead of just an input, but we still expected to code the many instructions. We haven't really made the application efficient.

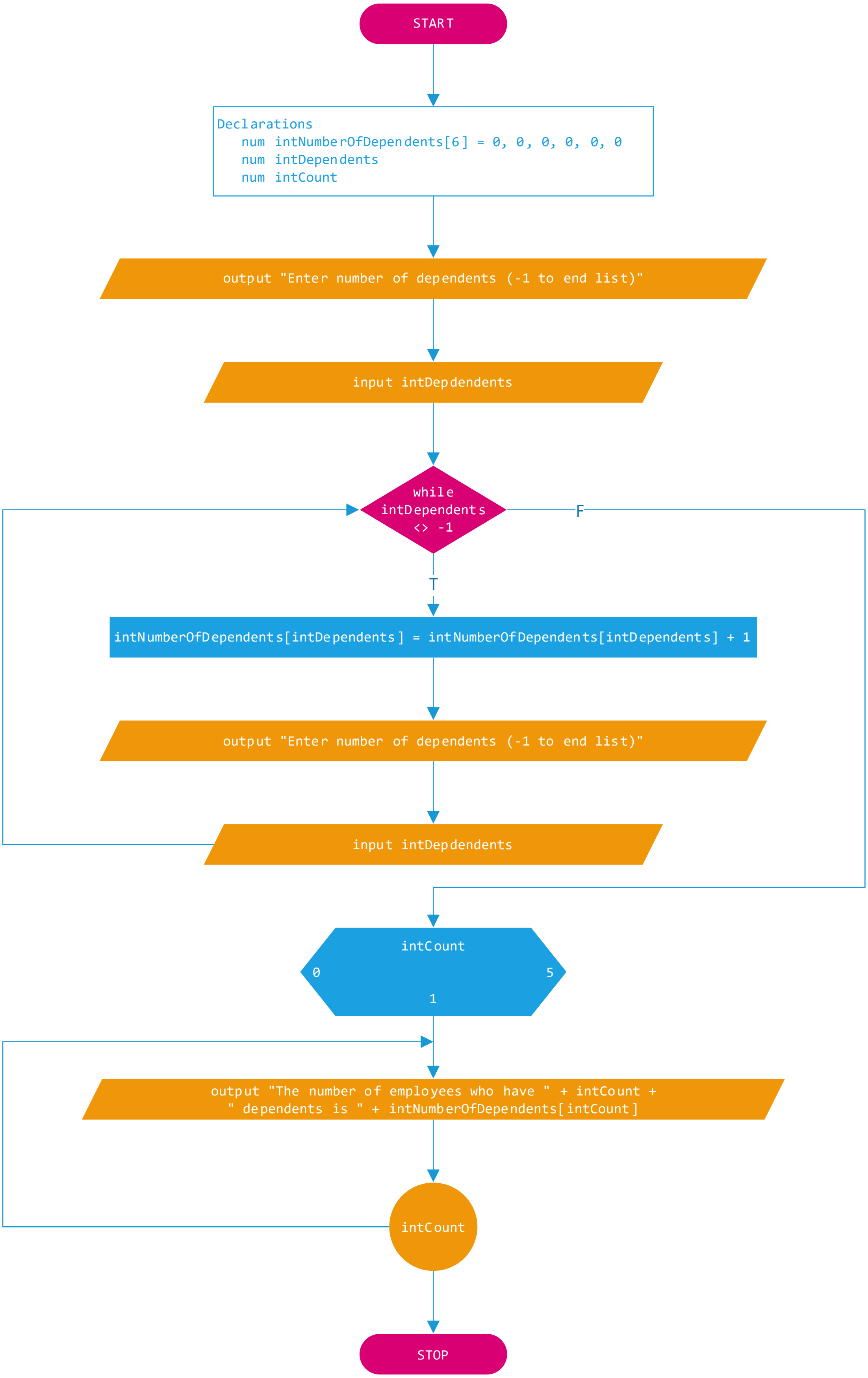
- Now consider the following and remember that the user entering the data, will always enter between 0 and 5 (we are assuming that no errors would occur):

	0	1	2	3	4	5
intDependents	0	0	0	0	0	0

**pseudocode

```
        0. Start
        1. Declarations
                num intNumberOfDependents[6] = 0, 0, 0, 0, 0, 0
                num intDependents
                num intCount

        2.      output "Enter number of dependents (-1 to end list)"
        3.      input intDependents

        4.      while intDependents <> -1
                        intNumberOfDependents[intDependents] = intNumberOfDependents[intDependents] +1
                        output "Enter number of dependents (-1 to end list)"
                        input intDependents
                endwhile

        5.      for intCount = 0 to 5 Step 1
                        output "The number of employees who have " + intCount +
                                " dependents is " + intNumberOfDependents[intCount]
                endfor

        6. Stop
```

**flowchart


START
Declarations
num intNumberOfDependents[6] = 0, 0, 0, 0, 0, 0
num intDependents
num intCount
output "Enter number of dependents (-1 to end list)"
input intDepdendents
while intDependents <> -1
F
T
intNumberOfDependents[intDependents] = intNumberOfDependents[intDependents] + 1
output "Enter number of dependents (-1 to end list)"
input intDepdendents
intCount
0
5
1
output "The number of employees who have " + intCount + " dependents is " + intNumberOfDependents[intCount]
intCount
STOP

** - the code in the textbook is similar but in this solution there are differences

- There was actually no need for the if..then statement, since the intDependents variables stores the number of dependents an employee has, and the variable is also used to reference a specific cell that will increment its value to keep track of the count

- Side note: The above application used a while..loop. In this example, the purpose of the while..loop was not about iterating through the elements of the array. It was dependent on the "number of dependents" the user entered, which could have been any value from 0 to 5, many number of times.

==

Using constants with arrays

==

-- We have learnt at the beginning of our programming journey, that when using a constant variable is sometimes better when dealing with values that need to remain constant throughout the processing of source code.

-- The following are examples of where constants could be used when dealing with arrays

- The size of an array

**code

```
1. Declarations
        num intNUMBEROFSTUDENTS = 10
        num intMarks[intNUMBEROFSTUDENTS] // declare array has 10 cells
```

- The values loaded at declaration into the cells of the array

**code

```
1. Declarations
        string strDAYS[7] = "Sunday", "Monday", "Tuesday", "Wednesday",
                                    "Thursday", "Friday", "Saturday"

2.      output strDAYS[0] // would print ~ Sunday
```

	0	1	2	3	4	5	6
strDays	**Sunday**	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**

- The indexes used (as an array subscript)
    - Imagine that you are expected to save the total sales for each of the 9 provinces in an array called fltSales.
    - To reference a specific cell & retrieve the total sales, we would need to state the index

**code

```
5.      output "The sales for ?? " + fltSales[3]
```

- Which province is being referenced to in the above statement?
- *We would have to refer to our documentation to know that cell 3 is the sales for KwaZulu Natal (not very good if we have to always refer to documentation)*

```
5.      output "The sales for KwaZulu Natal " + fltSales[3]
```

- Now consider that we have a list of the South African provinces as constant variables of an integer datatype. The variable name is the name of the province, and the values will range from 0 to 9 for the list of provinces.

- These variables can now be used as the index in the cell reference (in the array)

**code

```
1. Declarations
        num intEASTERNCAPE = 0
        num intFREESTATE = 1
        num intGAUTENG = 2
        num intKWAZULUNATAL = 3
        num intLIMPOPO = 4
        num intMPUMALANGA = 5
        num intNORTHERNCAPE = 6
        num intNORTHWEST = 7
        num intWESTERNCAPE = 8
        num fltSales [9]

// now if we wish to refer to the sales for a province
10.     output "The sales for Gauteng is " + fltSales[intGAUTENG]
```

- The reference to the sales for the province is now a little better understood as you program a solution.

```
====================================================================================================
Table Lookup
====================================================================================================
```

-- One of the common applications that is associated with arrays is to look up a value in a table/array after you have stored (& processed) data.

- Example
- a 1D Array that contains the number of days for each month in the year
- Read in a month number and print out the number of days in the month

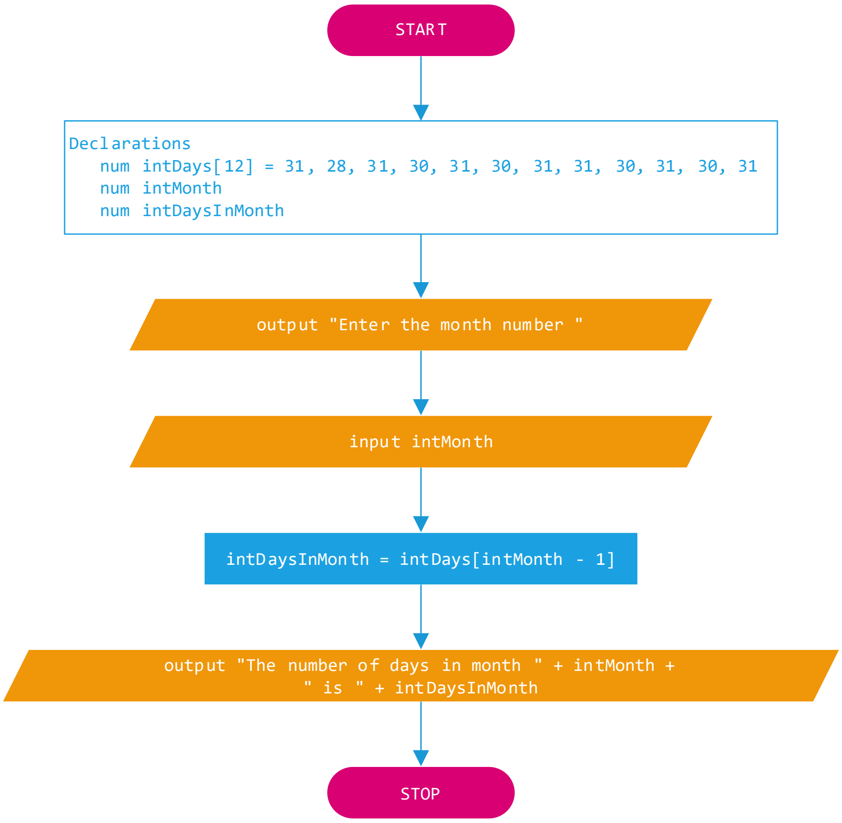
	intDays
0	31
1	28
2	31
3	30
4	31
5	30
6	31
7	31
8	30
9	31
10	30
11	31

**pseudocode

```
0. Start
1. Declarations
        num intDays[12] = 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
        num intMonth
        num intDaysInMonth

2.      output "Enter the month number "
3.      input intMonth

4.      intDaysInMonth = intDays[intMonth - 1]   // why are we subtracting 1?

5.      output "The number of days in month " + intMonth + " is " + intDaysInMonth
6. Stop
```

- The subtracting of the month number is because when the user entered for example 1, for the first month, then we as programmers need to refer to the first cell in the array, which is cell[0]. So, the subtraction here is needed. Again, if the user entered 12, then we need to refer to the 12th month, which is in cell[11].

**flowchart


START
Declarations
num intDays[12] = 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
num intMonth
num intDaysInMonth
output "Enter the month number "
input intMonth
intDaysInMonth = intDays[intMonth - 1]
output "The number of days in month " + intMonth +
" is " + intDaysInMonth
STOP

==

Sequential Search - Searching an array for an exact match

==

-- Searching for a value (requested from the user) in an array is a common algorithm we will come across. Commonly known as a linear search or sometimes a sequential search.

-- There are few interpretations & variations of the sequential search. It important to understand the logic of each of them, as it will become your choice, the programmer, to adapt and use the code as required for the problems you are presented with in the future.

-- Example - Read in 10 numbers into a 1D Array and then prompt the user to search for a specific number. If the number is found or not found, an appropriate message should follow.

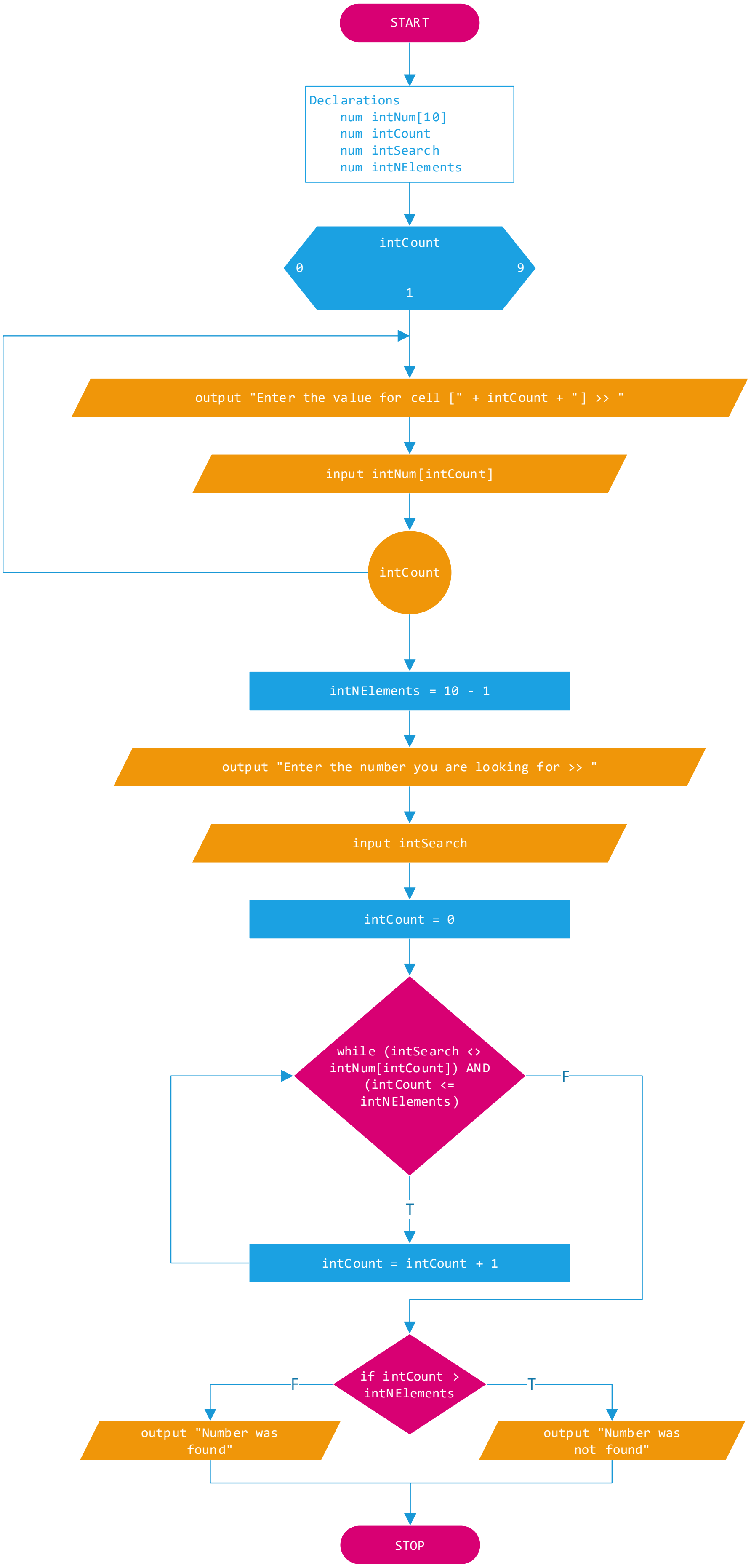
```
----------------------------------------------------------------------------------------------------
        - using a while..loop
----------------------------------------------------------------------------------------------------
```

**pseudocode

```
        0. Start
        1. Declarations
                num intNum[10]
                num intCount
                num intSearch
                num intNElements // will be used to store the last cell index number

                // reading in the values into the array from the user
        2.      for intCount = 0 to 9 Step 1
                        output "Enter the value for cell [" + intCount + "] >> "
                        input intNum[intCount]
                endfor

        3.      intNElements = 10 - 1  // size of the array - 1 | the last index

                // read in the search value
        4.      output "Enter the number you are looking for >> "
        5.      input intSearch

        6.      intCount = 0 // initialize the count / index for searching

        7.      while (intSearch <> intNum[intCount]) AND (intCount <= intNElements)
                        intCount = intCount + 1
                endwhile

        8.      if intCount > intNElements then
                        output "Number was not found"
                else
                        output "Number was found"
                endif

        9. Stop
```

	0	1	2	3	4	5	6	7	8	9
intNum										

**flowchart


START
Declarations
num intNum[10]
num intCount
num intSearch
num intNElements
intCount
0
9
1
output "Enter the value for cell [" + intCount + "] >> "
input intNum[intCount]
intCount
intNElements = 10 - 1
output "Enter the number you are looking for >> "
input intSearch
intCount = 0
while (intSearch <> intNum[intCount]) AND (intCount <= intNElements)
F
T
intCount = intCount + 1
if intCount > intNElements
F
T
output "Number was found"
output "Number was not found"
STOP

- The key to the search is the compound/combined condition for the while..loop.

- The first condition compares the search value to the value in the array (using the index). Notice the index was initialized to 0 before the loop so the first time, the search variable is being compared to the first cell in the array and if not found (** besides the other condition), the intCount variable will increment, giving us access to the next element in the array to continue the search process.

- The second condition is comparing the intCount to the number of elements in the array. More importantly it is making sure that we stay within the bounds of the array and if the intCount exceeds the bounds, then the loop will exit, since there are no more elements in the array to look at.

- The loop will exit if either the search value was found or that we have looked through all the elements did not find the value.

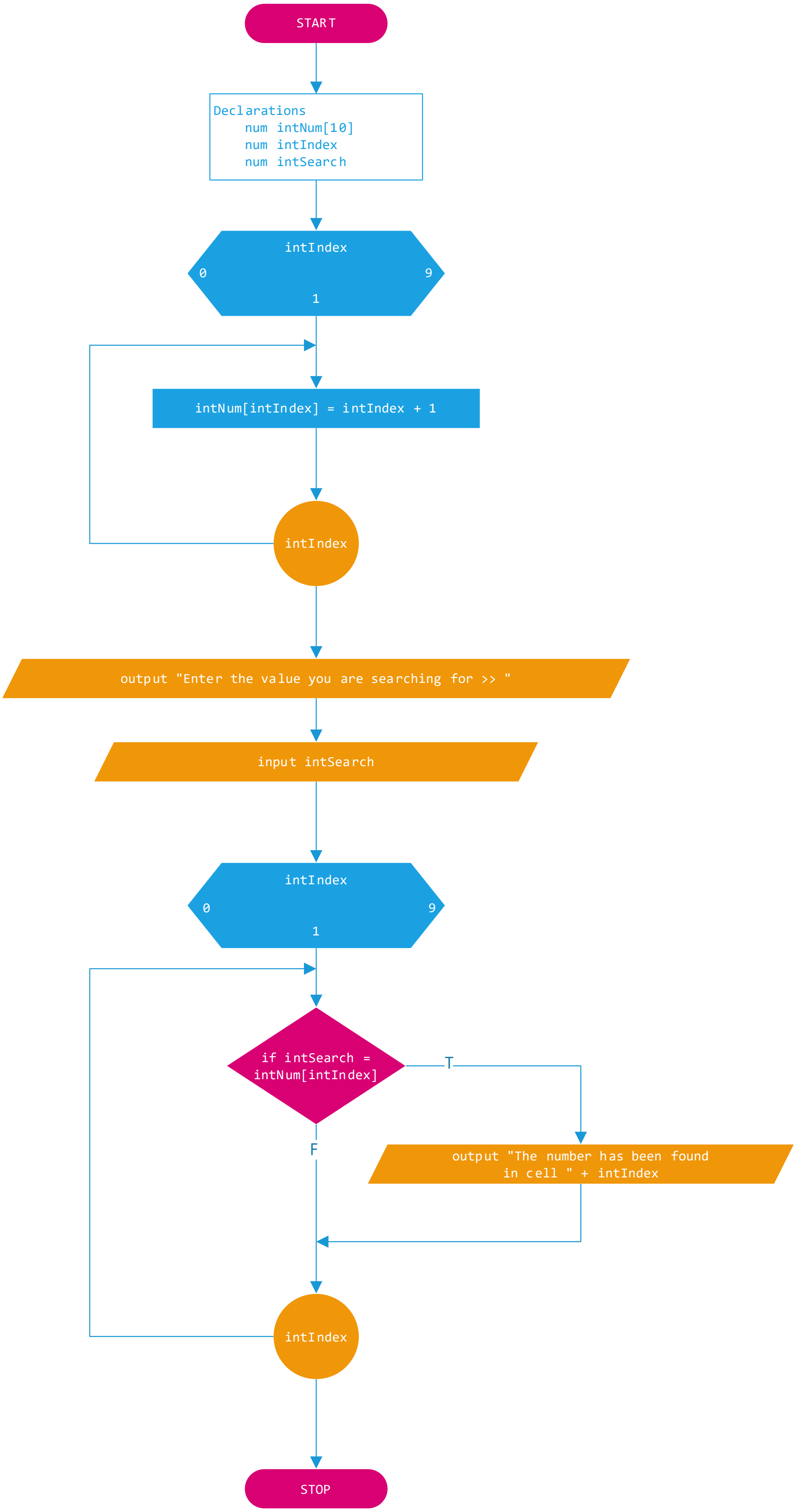
- using a for..loop

** For this example, instead of asking the user to load the values into the array, the array is loaded with a value using the index variable in a separate loop.

```
        **pseudocode
        0. Start
        1. Declarations
                num intNum[10]
                num intIndex
                num intSearch

                // loading the array with the values 1 through to 10
        2.      for intIndex = 0 to 9 Step 1
                        intNum[intIndex] = intIndex + 1
                endfor

        3.      output "Enter the value you are searching for >> "
        4.      input intSearch

        5.      for intIndex = 0 to 9 Step 1
                        if intSearch = intNum[intIndex] then
                                output "The number has been found in cell " + intIndex
                        endif
                endfor

        6. Stop
```

	0	1	2	3	4	5	6	7	8	9
intNum	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

**flowchart


START
Declarations
num intNum[10]
num intIndex
num intSearch
intIndex
0
9
1
intNum[intIndex] = intIndex + 1
intIndex
output "Enter the value you are searching for >> "
input intSearch
intIndex
0
9
1
if intSearch = intNum[intIndex]
T
output "The number has been found in cell " + intIndex
F
intIndex
STOP

- The for..loop is used simply because we know the dimension of the 1D array (10 cells)
- The loop will iterate through each of the elements and the condition in the if..then statement will compare the search value with the value stored in the array cell. If the condition is true, a value has been found and an output statement is executed.

- *But there is no output if the value is not found.*

*** **_What could we do to deal with this?_**

- The following edit would be interesting but is actually incorrect to be inside the if..then..else statement

```
...
5.      for intIndex = 0 to 9 Step 1
                if intSearch = intNum[intIndex] then
                        output "The number has been found in cell " + intIndex
                else
                        output "The number was not found"
                endif
        endfor
...
```

- With the introduction of the else section, this would mean that each time the search does not equal the value in the cell of the array, it would print out "The number was not found". It may seem to be correct, but the many "not found" outputs would not be welcomed by the user.

- We also cannot copy the logic of the previous example where they used the number of elements (after the loop had ended) to check that if the user goes above the size of the array, then the value was not found

```
...
5.      for intIndex = 0 to 9 Step 1
                if intSearch = intNum[intIndex] then
                        output "The number has been found in cell " + intIndex
                endif
        endfor

6.      if intIndex > 9 then
                output "Number was not found"
        endif
7. Stop
```

- The reason why this would not work is because after the loop has completed its cycle, the value of intIndex is greater than the ending number (from the loop code). This would then mean that the result for the condition of the (2nd)if..then statement would always be true and the "...not found" statement would be printed. This might be correct if the number being searched for is never found, but if the number is found, we would print that the number is found and also print the output for not found. This would not look good (& it’s confusing) to the user

- Another possibility is to "kick you out" of the loop once the value has been found

```
...
5.      for intIndex = 0 to 9 Step 1
                if intSearch = intNum[intIndex] then
                        output "The number has been found in cell " + intIndex
                        intIndex = 20 // out of the bounds of the loop
                endif
        endfor

6.      if intIndex <> 20 then
                output "Number was not found"
        endif
7. Stop
```

- So, in the above code - an interesting thought has been executed. If the search value is found, then there is no need to look through the rest of the array. The only way to leave the iteration through the elements of the array is for the loop counter to be greater than the ending number. By assigning the intIndex variable to 20, I am forcing the internal condition that decides to repeat, the loop, to realise that the ending number has passed, and the loop should end.

- This way, if the value is never found, then the intIndex would be 10 after the loop has completed its cycle and it would never be 20. My attempt is to use intIndex as some sort of flag to determine the right course of action.

- There is a better option available to us instead of the added confusion of the using intIndex in the way it was used above.
- We could introduce an actual flag variable to determine if the value is found or not found.

**pseudocode

```
0. Start
1. Declarations
        num intNum[10]
        num intIndex
        num intSearch
        num blnFlag

        // loading the array with the values 1 through to 10
2.      for intIndex = 0 to 9 Step 1
                intNum[intIndex] = intIndex + 1
        endfor

3.      output "Enter the value you are searching for >> "
4.      input intSearch

5.      blnFlag = false // initialized - false = not found

6.      for intIndex = 0 to 9 Step 1
                if intSearch = intNum[intIndex] then
                        output "The number has been found in cell " + intIndex
                        blnFlag = true // change the value, if found
                endif
        endfor

7.      if blnFlag = false then
                output "The number was not found"
        endif

8. Stop
```

- By using the flag variable, the code is a much simpler and easier to understand.

- An interesting result in the previous program, the code searches for the value and if it is found, display a message and if not found, display an appropriate message.

-- There is one interesting difference between the while..loop code and the for..loop code. When the value is found in program that uses the while..loop, the loop will be exited. And in the for..loop version, even after maybe finding the value early in the array, the loop will continuously iterate through the remaining cells still looking for the search value.

-- It may look like it is not an efficient solution, but the code could be utilized if there are duplicate values that were meant to be found in the array. The need might arise if the array contains duplicate values, and you are required to display a found message each time the search value was found.

-- What if we were searching for a unique value in the array, but still wanted to use a for..loop
- Please consider the following edited version of the code, once again

**pseudocode

```
0. Start
1. Declarations
        num intNum[10]
        num intIndex
        num intSearch
        num blnFlag
        num intFoundIndex

        // loading the array with the values 1 through to 10
2.      for intIndex = 0 to 9 Step 1
                intNum[intIndex] = intIndex + 1
        endfor

3.      output "Enter the value you are searching for >> "
4.      input intSearch

5.      blnFlag = false // initialized - false = not found

6.      for intIndex = 0 to 9 Step 1
                if intSearch = intNum[intIndex] then
                        intFoundIndex = intIndex
                        blnFlag = true // change the value, if found
                        intIndex = 10
                        // forcing the loop variable to change - to exit the loop
                endif
        endfor

7.      if blnFlag = false then
                output "The number was not found"
        else
                output "The number has been found in cell " + intFoundIndex
        endif

8. Stop
```

- The loop would still iterate through all the cells in the array, but instead of displaying the message if the value is found inside the if..then statement in the loop, I chose to only display the found message after the loop has ended.

- And I forced the loop to end by changing the value of the intIndex variable. Once the next iteration happens, intIndex would increment to 11 and because it is past the ending number of the loop, it will be exited and the condition of the if..then statement after the loop is about the flag and not the index.

- **Example** - Item Orders
    - Orders require a list product numbers (that have been ordered)
    - Customer presents the list of product numbers into the program

*** Create an array to store all the valid product numbers (verified list) that the company offers from their inventory*

	intValidItems
0	106
1	108
2	307
3	405
4	457
5	688

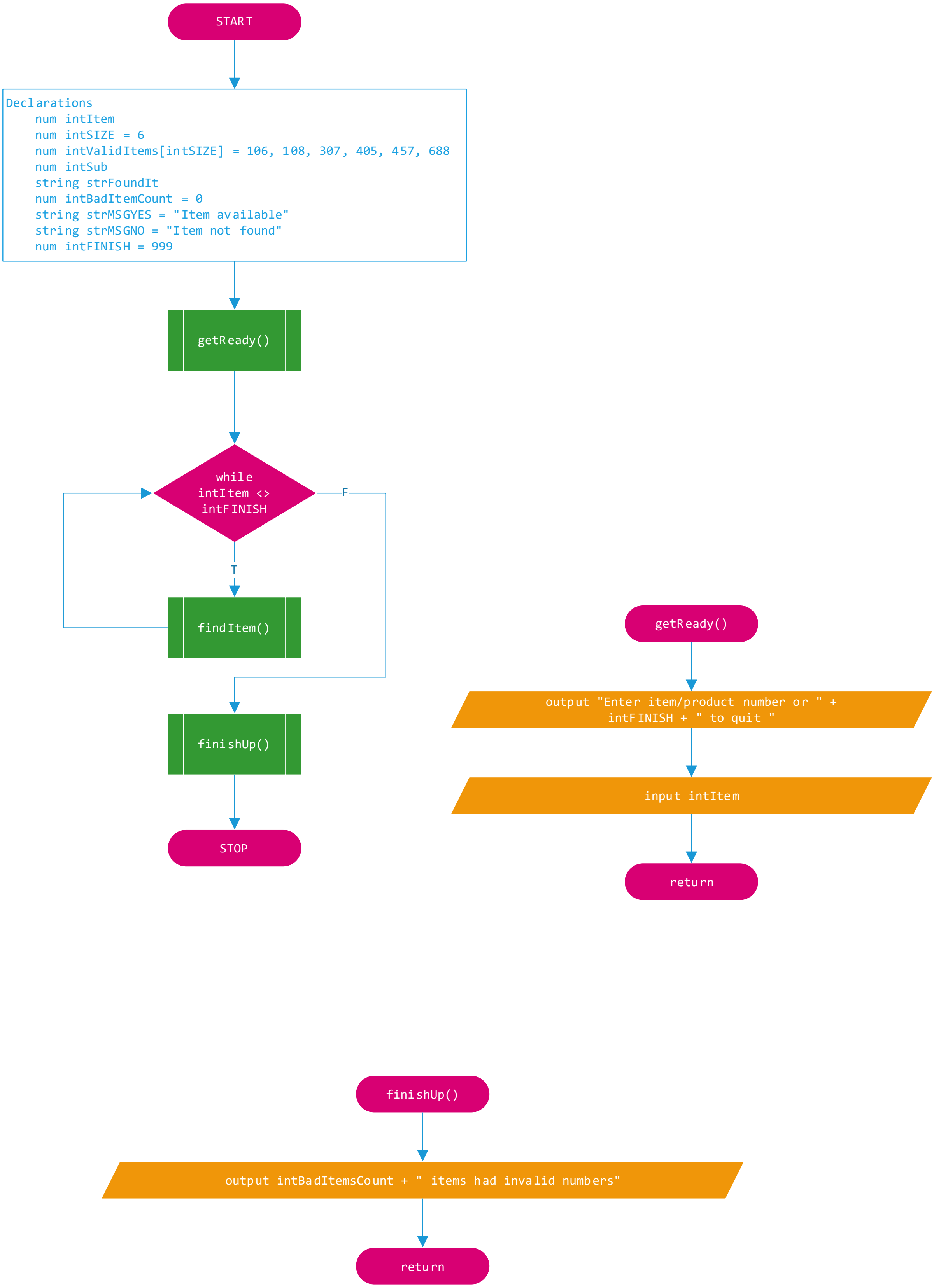
- Search the array for an exact match - comparing the list of items in the order to the stored valid product numbers

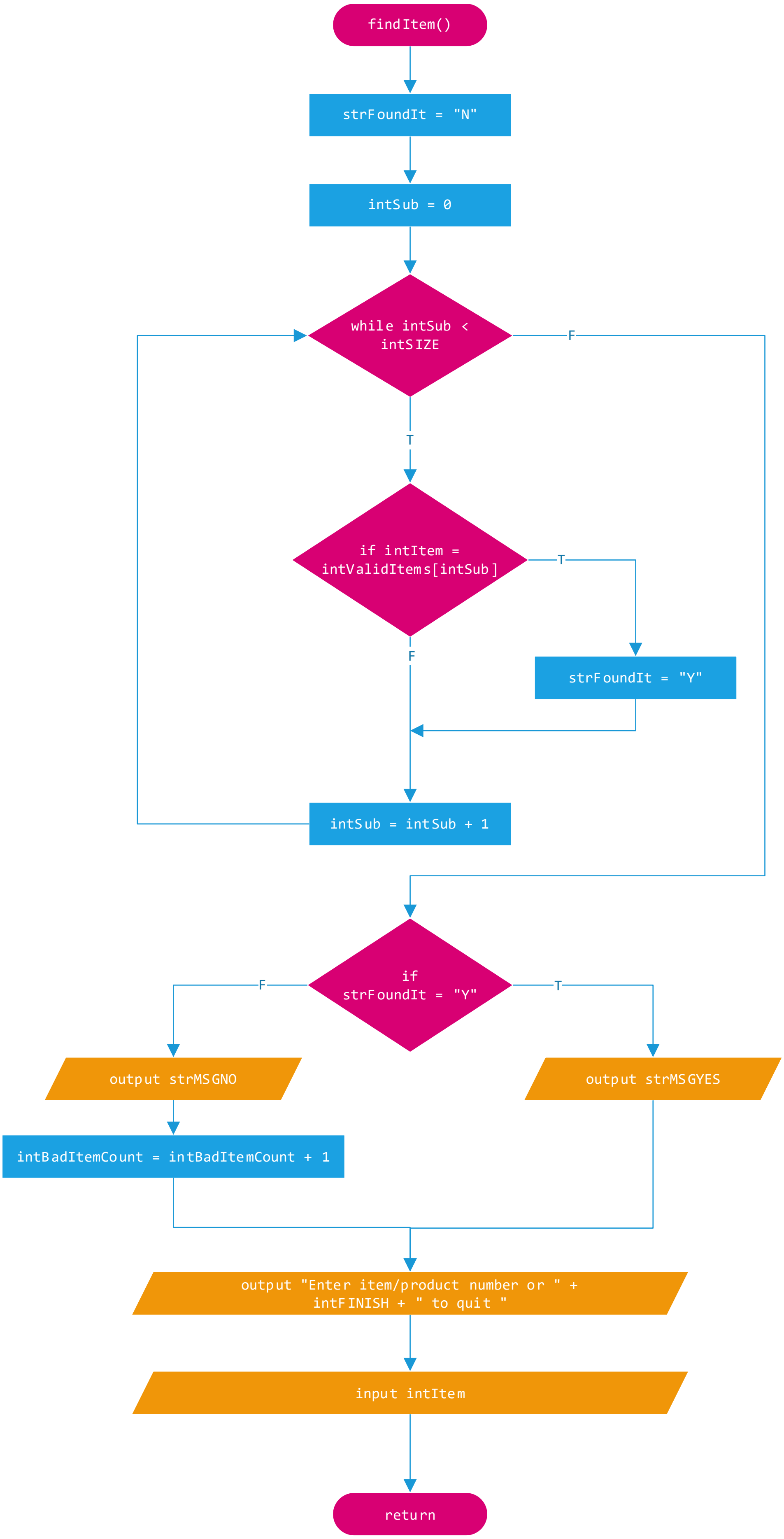
- Also required to keep track of the invalid product numbers entered by the user
- the algorithm is verifying the product/item's availability

```
**pseudocode
0. Start
1. Declarations
        num intItem
        num intSIZE = 6           // constant - size of the array
        num intValidItems[intSIZE] = 106, 108, 307, 405, 457, 688
        num intSub                // subscript - index
        string strFoundIt         // the flag variable
        num intBadItemCount = 0
        string strMSGYES = "Item available"
        string strMSGNO = "Item not found"
        num intFINISH = 999       // constant - sentinel value

2.      getReady()       // primer read

3.      while intItem <> intFINISH
                findItem()
        endwhile

4.      finishUp()

5. Stop
```

```
0. getReady()
1.      output "Enter item/product number or " + intFINISH + " to quit "
2.      input intItem
3. return
```

```
----------------------------------------------------------------------------------------------------

        0. findItem()
        1.      strFoundIt = "N"
        2.      intSub = 0

        3.      while intSub < intSIZE
                        if intItem = intValidItems[intSub] then
                                strFoundIt = "Y"
                        endif
                        intSub = intSub + 1
                endwhile

        4.      if strFoundIt = "Y" then
                        output strMSGYES
                else
                        output strMSGNO
                        intBadItemCount = intBadItemCount + 1
                endif

        5.      output "Enter next item number or " + intFINISH + " to quit"
        6.      input intItem

        7. return

----------------------------------------------------------------------------------------------------

        0. finishUp()
        1.      output intBadItemCount + " items had invalid numbers"
        2. return

----------------------------------------------------------------------------------------------------
        **flowchart
```


START
Declarations
num intItem
num intSIZE = 6
num intValidItems[intSIZE] = 106, 108, 307, 405, 457, 688
num intSub
string strFoundIt
num intBadItemCount = 0
string strMSGYES = "Item available"
string strMSGNO = "Item not found"
num intFINISH = 999
getReady()
while intItem <> intFINISH
F
T
findItem()
finishUp()
STOP
getReady()
output "Enter item/product number or " + intFINISH + " to quit "
input intItem
return
finishUp()
output intBadItemsCount + " items had invalid numbers"
return


findItem()
strFoundIt = "N"
intSub = 0
while intSub < intSIZE
F
T
if intItem = intValidItems[intSub]
T
strFoundIt = "Y"
F
intSub = intSub + 1
if strFoundIt = "Y"
F
T
output strMSGNO
output strMSGYES
intBadItemCount = intBadItemCount + 1
output "Enter item/product number or " + intFINISH + " to quit "
input intItem
return

- Remember the theory of a flag variable from the previous sections/notes
- A flag variable is used to indicate whether an event has occured

- The logic of searching an array (using the program shared above)

-- set a subscript variable to zero (0) to start at the first element

-- Initialize the flag variable to false to indicate the desired value has not been found

  -- Examine each element in the array

  -- If the value matches, set the flag to true

  -- If the value does not match, increment the subscript and examine the next array element

Binary Search

-- A faster method of searching for an element in an array is the binary search.

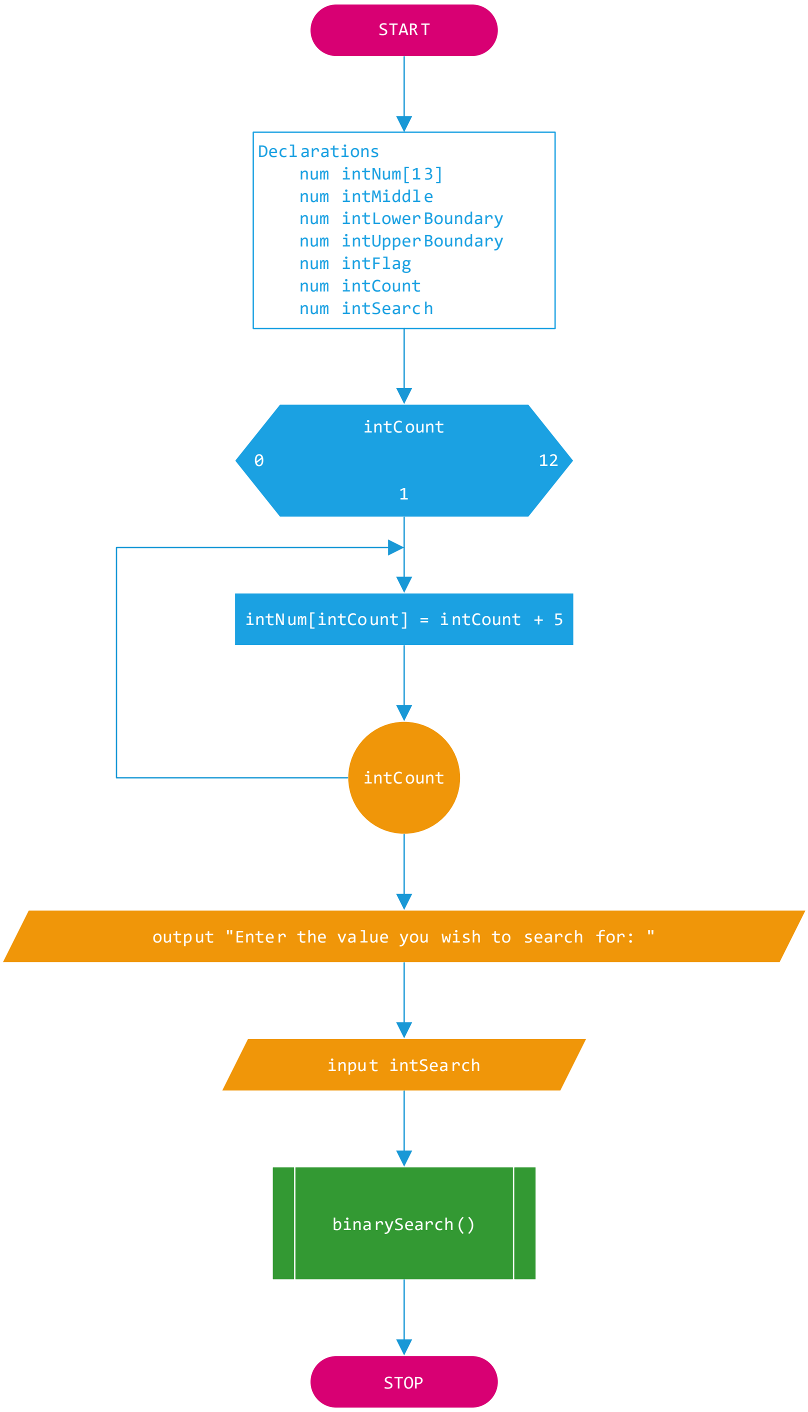
-- The technique requires that we determine the mid-point of the all or part of the array. If we find the value at the mid-point, then flow of the loop will end with the element number. If the value is not found at the mid-point, we will check to see if the value is lower or higher than the middle value. The boundaries are then reset to select a new section of the array in which the search value should be found. The process starts again, determine the mid-point, check if the value is found, if not, determine if lower or higher and reset the boundaries once again. We are constantly dividing the array in half (binary is 2) to locate the value.

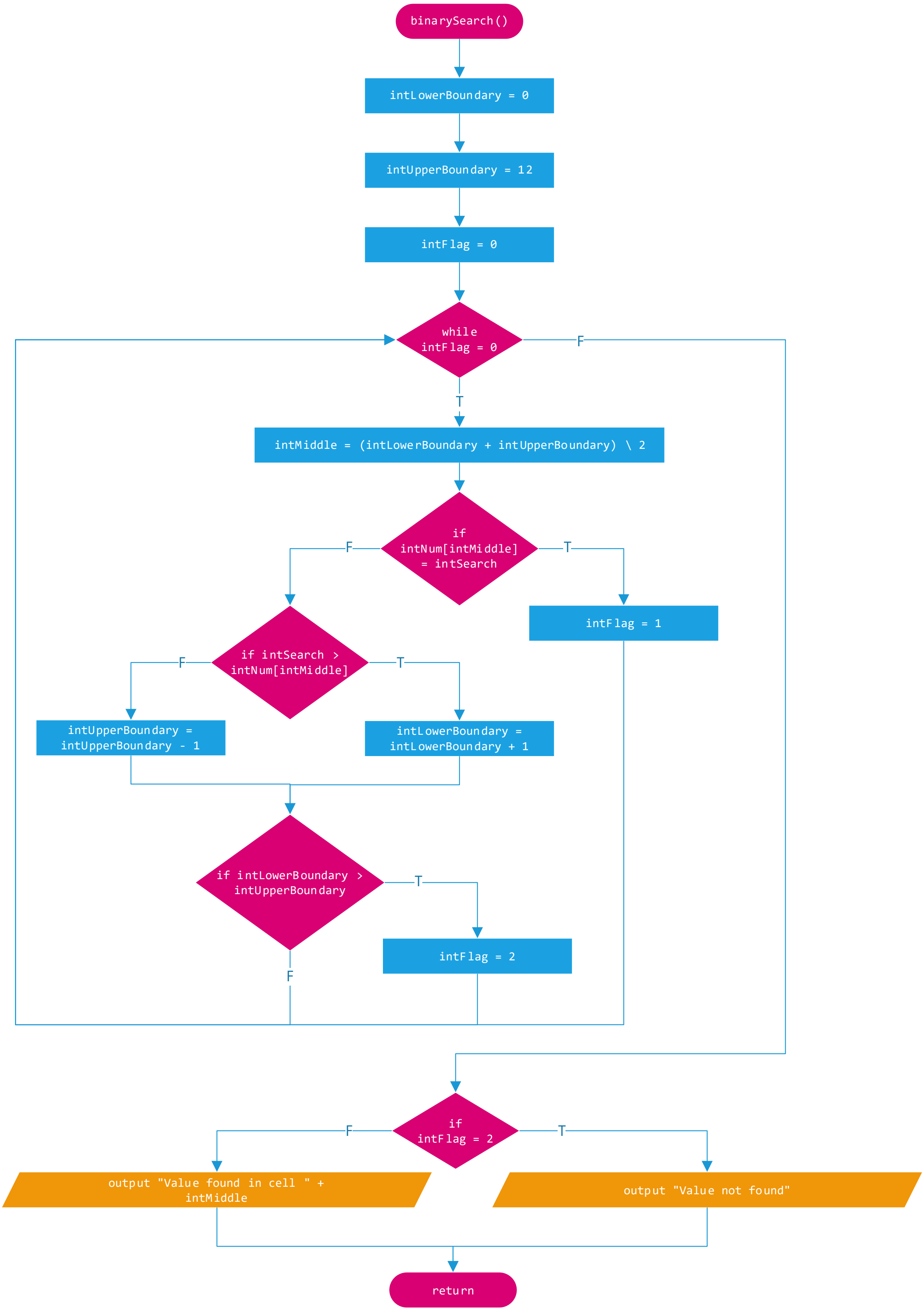
-- This method is much more efficient. If we consider searching an array of 1000 elements, the sequential search requires that we iterate through 1000 cells. But in the binary search, it would take less than 10 comparisons to find the value, if it exists.

-- One important point to note, the values in the array would need to be sorted, numerical or alphabetical in order for this type of search to work.

	intNum
0	5
1	6
2	7
3	8
4	9
5	10
6	11
7	12
8	13
9	14
10	15
11	16
12	17

```
**pseudocode
        0. Start
        1. Declarations
                num intNum[13]
                num intMiddle
                num intLowerBoundary
                num intUpperBoundary
                num intFlag
                num intCount
                num intSearch

                // populating the array with values
        2.      for intCount = 0 to 12 Step 1
                        intNum[intCount] = intCount + 5
                endfor

        3.      output "Enter the value you wish to search for: "
        4.      input intSearch

        5.      binarySearch()

        6. Stop
----------------------------------------------------------------------------------------------------
        0. binarySearch()

        1.      intLowerBoundary = 0    // index of the first element
        2.      intUpperBoundary = 12   // index of the last element

        3.      intFlag = 0             // flag value 0 = not yet found

        4.      while intFlag = 0
                        intMiddle = (intLowerBoundary + intUpperBoundary) \ 2

                        if intNum[intMiddle] = intSearch then
                                intFlag = 1     // value found
                        else
                                if intSearch > intNum[intMiddle] then
                                        intLowerBoundary = intLowerBoundary + 1
                                else
                                        intUpperBoundary = intUpperBoundary - 1
                                endif

                                if intLowerBoundary > intUpperBoundary then
                                        intFlag = 2 // value never going to be found
                                endif
                        endif
                endwhile

        5.      if intFlag = 2 then
                        output "Value not found"
                else
                        output "Value found in cell " + intMiddle
                endif

        6. return
----------------------------------------------------------------------------------------------------
        **flowchart
```


START
Declarations
num intNum[13]
num intMiddle
num intLowerBoundary
num intUpperBoundary
num intFlag
num intCount
num intSearch
intCount
0
12
1
intNum[intCount] = intCount + 5
intCount
output "Enter the value you wish to search for: "
input intSearch
binarySearch()
STOP


binarySearch()
intLowerBoundary = 0
intUpperBoundary = 12
intFlag = 0
while
intFlag = 0
F
T
intMiddle = (intLowerBoundary + intUpperBoundary) \ 2
if
intNum[intMiddle]
= intSearch
F
T
intFlag = 1
if intSearch >
intNum[intMiddle]
F
T
intUpperBoundary =
intUpperBoundary - 1
intLowerBoundary =
intLowerBoundary + 1
if intLowerBoundary >
intUpperBoundary
T
intFlag = 2
F
if
intFlag = 2
F
T
output "Value found in cell " +
intMiddle
output "Value not found"
return

	intNum
0	5
1	7
2	9
3	12
4	15
5	20
6	25
7	28
8	33
9	35
10	40
11	44
12	47

Order of comparisons to find the element number of 44

	Boundary		Index	Element
	Lower	Upper	Middle	
1	0	12	6	25
2	7	12	9	35
3	10	12	11	44

Number 44 is found in three comparisons, compared to 12 with the sequential search

Using parallel arrays

-- More than one 1D Array in a program, once visualized is likely to represent parallel arrays.
-- Each element in one array is associated with an element in the same relative position in the other array

- **Example**
- 2 arrays - each with six elements
  - * valid item numbers (product id numbers)
  - * valid item prices (product prices)
- Each price in the valid item price array is in the same position as the corresponding item in the valid item number array

- We can use this as an opportunity to look through the item array and when found, retrieve the item price from the other array

	intValidItems		fltValidPrices
0	106	0	0,59
1	108	1	0,99
2	307	2	4,50
3	405	3	15,99
4	457	4	17,50
5	688	5	39,00

-- 2 or more arrays contain related data
-- The subscript being used is what relates the arrays (elements at the same position in each array are logically related)
-- Parallel arrays are useful when there is a story that matches the understanding of the data being presented

```
        **pseudocode
        0. Start
        1. Declarations
                num intItem
                num fltPrice
                num intSIZE = 6         // constant - size of the array
                num intValidItems[intSIZE] = 106, 108, 307, 405, 457, 688
                num fltValidPrices[intSIZE] = 0.59, 0.99, 4.50, 15.99, 17.50, 39.00
                num intSub                  // subscript - index
                string strFoundIt
                num intBadItemCount = 0
                string strMSGYES = "Item available"
                string strMSGNO = "Item not found"
                num intFINISH = 999     // constant - sentinel value

        2.      getReady()      // primer read

        3.      while intItem <> intFINISH
                        findItem()
                endwhile



        4.      finishUp()

        5. Stop
```

```
----------------------------------------------------------------------------------------------------
        0. getReady()
        1.      output "Enter item number or " + intFINISH + " to quit "
        2.      input intItem
        3. return

----------------------------------------------------------------------------------------------------
        0. findItem()
        1.      strFoundIt = "N"
        2.      intSub = 0

        3.      while intSub < intSIZE
                        if intItem = intValidItems[intSub] then
                                strFoundIt = "Y"
                                fltPrice = fltValidPrices[intSub]
                        endif
                        intSub = intSub + 1
                endwhile

        4.      if strFoundIt = "Y" then
                        output strMSGYES
                        output "The price of " + intItem + " is " + fltPrice
                else
                        output strMSGNO
                        intBadItemCount = intBadItemCount + 1
                endif

        5.      output "Enter next item number or " + intFINISH + " to quit"
        6.      input intItem

        7. return

----------------------------------------------------------------------------------------------------
        0. finishUp()
        1.      output intBadItemCount + " items had invalid numbers"
        2. return

----------------------------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------------------------
```

-- When we consider the programming around a sequential search, it is important to realise that the code can and should be made more efficient through a few tweaks when the storyline allows us the opportunity.

- When processing the sequential search through the elements of the array, the search should stop when the match is found (unless we are search for duplicate values)
- Set a variable to a specific value instead of letting normal processing set it
- The larger the array, the better the need to improve the program by exiting early

***** to reconsider the above code that was dealing with the search - take note of the edited condition for the while..loop***

```
while (intSub < intSIZE) AND (strFoundIt = "N")
```

- Now the loop will not only exit when we have gone through all the elements in the array but also when the value being searched for has been found

----

- **Example - Recorded Temperatures for a List of Cities**
    - You are required to enter a list of 5 cities and their recorded high temperatures for a specific day in the month

    - Once the data has been loaded into the arrays, you are required to calculate the average temperature of the list of 5 cities

    - You are also required to determine the highest and lowest recorded temperatures and display the temperatures along with the city names.

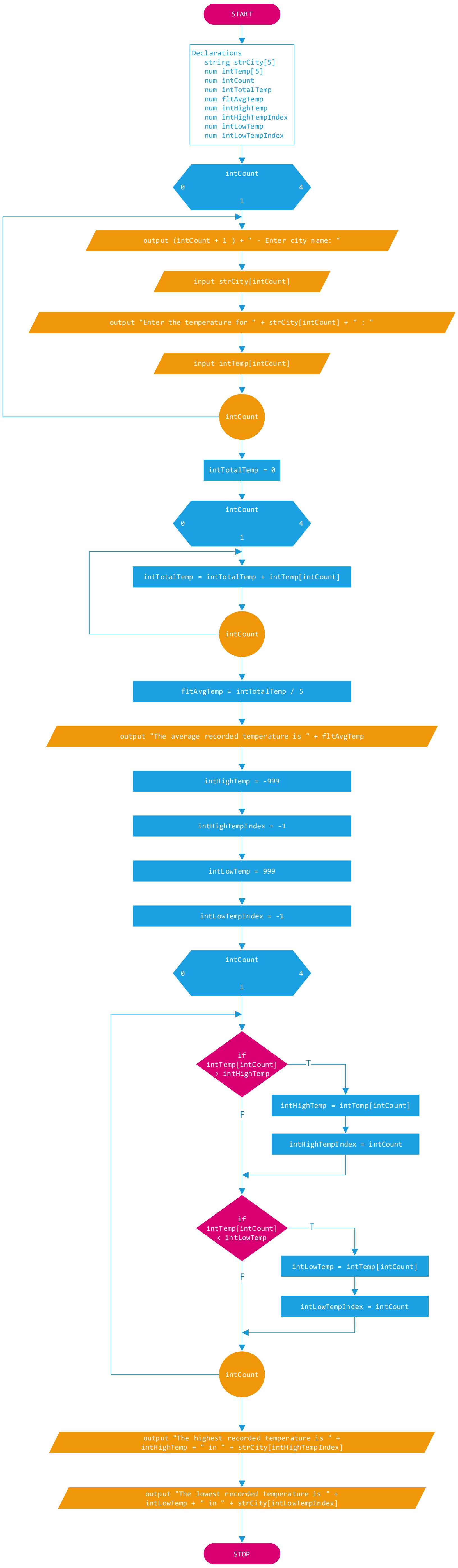
strCity	intTemp
Durban	22
Johannesburg	19
Cape Town	18
Upington	26
Makhado	24
Tshwane	20

```
        0. Start
        1. Declarations
                string strCity[5]
                num intTemp[5]
                num intCount
                num intTotalTemp
                num fltAvgTemp
                num intHighTemp
                num intHighTempIndex
                num intLowTemp
                num intLowTempIndex

        2.      for intCount = 0 to 4 Step 1
                        output (intCount + 1 ) + " - Enter city name: "
                        input strCity[intCount]
                        output "Enter the temperature for " + strCity[intCount] + " : "
                        input intTemp[intCount]
                endfor

        3.      intTotalTemp = 0
        4.      for intCount = 0 to 4 Step 1
                        intTotalTemp = intTotalTemp + intTemp[intCount]
                endfor
        5.      fltAvgTemp = intTotalTemp / 5
        6.      output "The average recorded temperature is " + fltAvgTemp

        7.      intHighTemp = -999
        8.      intHighTempIndex = -1

        9.      intLowTemp = 999
        10.     intLowTempIndex = -1

        11.     for intCount = 0 to 4 Step 1
                        if intTemp[intCount] > intHighTemp then
                                intHighTemp = intTemp[intCount]
                                intHighTempIndex = intCount
                        endif
                        if intTemp[intCount] < intLowTemp then
                                intLowTemp = intTemp[intCount]
                                intLowTempIndex = intCount
                        endif
                endfor

        12.     output "The highest recorded temperature is " + intHighTemp +
                        " in " + strCity[intHighTempIndex]

        13.     output "The lowest recorded temperature is " + intLowTemp +
                        " in " + strCity[intLowTempIndex]

        14. Stop
```

- Note how the variable that represents index is used in multiple lines of code to refer to information from both the arrays

----

** flowchart


START
Declarations
string strCity[5]
num intTemp[5]
num intCount
num intTotalTemp
num fltAvgTemp
num intHighTemp
num intHighTempIndex
num intLowTemp
num intLowTempIndex
intCount
0
4
1
output (intCount + 1) + " - Enter city name: "
input strCity[intCount]
output "Enter the temperature for " + strCity[intCount] + " : "
input intTemp[intCount]
intCount
intTotalTemp = 0
intCount
0
4
1
intTotalTemp = intTotalTemp + intTemp[intCount]
intCount
fltAvgTemp = intTotalTemp / 5
output "The average recorded temperature is " + fltAvgTemp
intHighTemp = -999
intHighTempIndex = -1
intLowTemp = 999
intLowTempIndex = -1
intCount
0
4
1
if intTemp[intCount] > intHighTemp
T
intHighTemp = intTemp[intCount]
intHighTempIndex = intCount
F
if intTemp[intCount] < intLowTemp
T
intLowTemp = intTemp[intCount]
intLowTempIndex = intCount
F
intCount
output "The highest recorded temperature is " + intHighTemp + " in " + strCity[intHighTempIndex]
output "The lowest recorded temperature is " + intLowTemp + " in " + strCity[intLowTempIndex]
STOP

==

Searching an array for a range match

==

-- Remember checking for a value within a range (decision logic notes [positive and negative logic - nested if..then statements])

- Example (in the textbook)
- Read customer order for the item numbers, the item numbers need to match the available item numbers to determine the correct price of an item.
- Now suppose you wish to determine the discount based on the quantity ordered

Quantity	Discount %
0 - 8	0
9 - 12	10
13 - 25	15
26 or more	20

(Now we have seen the above before, and the development of positive or negative logic (nested if..then statement) to process the information)

- Again - we wish to read the customer order data and determine the discount percentage based on the quantity ordered. If the customer has ordered 11 items, then the discount is 10%.

** *The following is the thoughts shared around this topic*

**** 1 **** Setup an array to contain every possible discount for every item quantity you think might be ordered

** The following array contains the discounts (76 times) for 0 items, 1 item, 2 items, and so on...

```
        num fltDISCOUNTS[76] = 0, 0, 0, 0, 0, 0, 0, 0, 0,
                               0.1, 0.1, 0.1, 0.1,
                               0.15 ..., 0.2
```

** The array contains 76 possible variations - the discounts for each quantity imagined.

** There is too much repetition, difficult to use or even immediately understand

** This array has us assuming that a customer would order no more than 76 items, but what if the customer orders more than that, then we would have to recode the array.

** Large arrays require more memory to store this data

** Mentioned above - too much repetition

**** 2 **** Instead of causing a certain level of confusion with the above statement made, let's look at using parallel arrays

** *please be reminded of the following style of code using nested if..then statements*

```
        if intQuantity <= 8 then
                fltDiscount = 0
        else
                if intQuantity <= 12 then
                        fltDiscount = 0.10
                else
                        if intQuantity <= 25 then
                                fltDiscount = 0.15
                        else
                                fltDiscount = 0.20
                        endif
                endif
        endif
```

& there is a reason I wish to show you the following

```
if intQuantity > 8 then                         // or intQuantity >= 9
        if intQuantity > 12 then                // or intQuantity >= 13
                if intQuantity > 25 then        // or intQuantity >= 26
                        fltDiscount = 0.20
                else
                        fltDiscount = 0.15
                endif
        else
                fltDiscount = 0.10
        endif
else
        fltDiscount = 0
endif
```

?? NOTE on the example code presented (Fig 6-15) ??

?? Thoughts - Not that there is an error - but logically, we could have found a better approach to the solution. Take note of the code above - using nested if..then statements would definitely be a worthwhile coded solution, even with the introduction of arrays - however I feel that the example below is adding a little more and does present an interesting, albeit a slightly less confusing solution ??

** The solution presented :
Consider using parallel arrays (size 4) to store the discounts and the range limits
****
The above negative logic code - relook at the conditions and the comments made. The range limits match the negative logic > 8 is also >= 9
Unfortunately, in this example - I cannot leave out "the last condition" (like we do when using nested if..then statements) because the logic calls for the testing of each of the 4 limits from the array
****

	0	1	2	3
Discount	**0**	**0,1**	**0,15**	**0,2**
Quantity Limits	**0**	**9**	**13**	**26**

**pseudocode

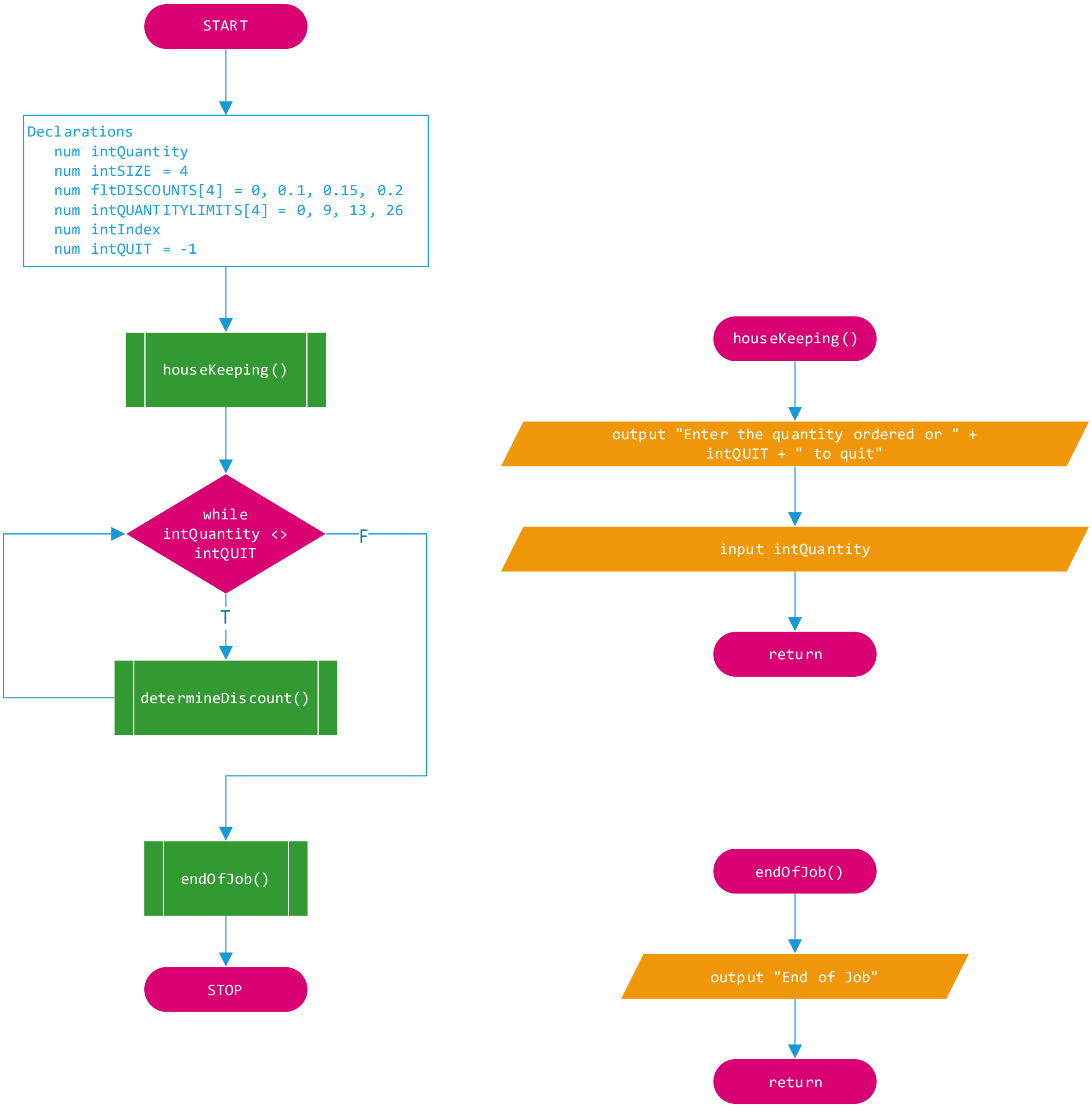
```
0. Start
1. Declarations
        num intQuantity
        num intSIZE = 4
        num fltDISCOUNTS[4] = 0, 0.1, 0.15, 0.2
        num intQUANTITYLIMITS[4] = 0, 9, 13, 26
        num intIndex
        num intQUIT = -1

2.      houseKeeping()

3.      while intQuantity <> intQUIT
                determineDiscount()
        endwhile

4.      endOfJob()

5. Stop
```

```
0. houseKeeping()
1.      output "Enter the quantity ordered or " + intQUIT + " to quit"
2.      input intQuantity
3. return
```

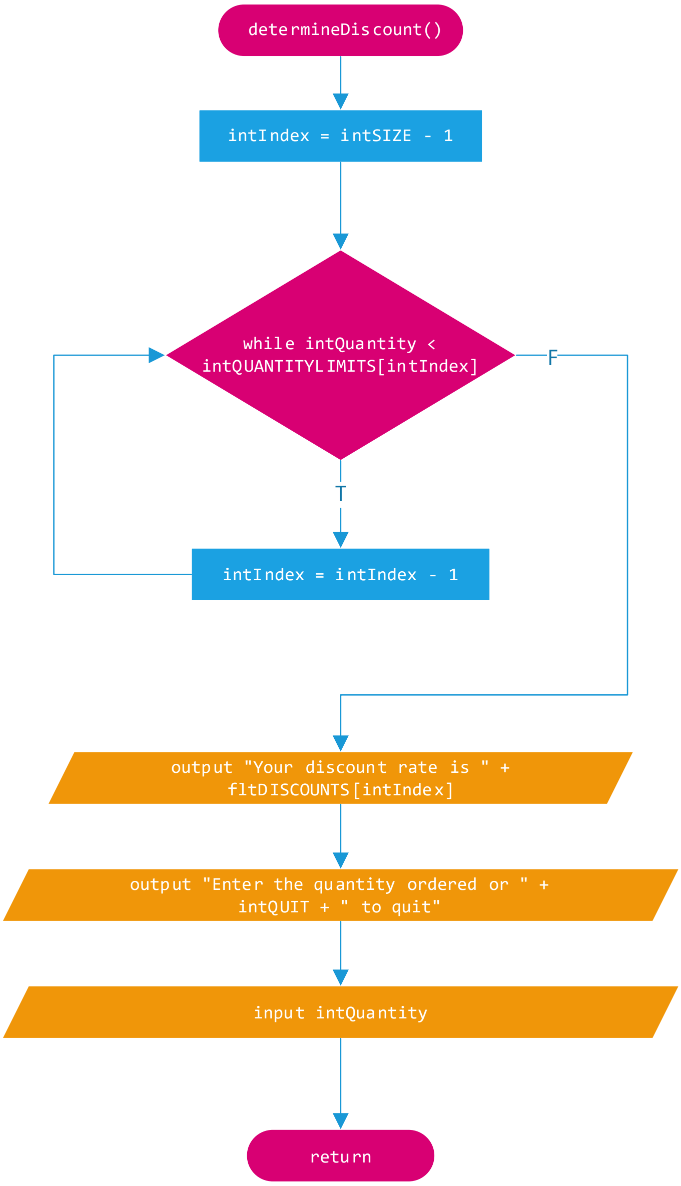
```
0. determineDiscount()
1.        intIndex = intSIZE - 1

2.        while intQuantity < intQUANTITYLIMITS[intIndex]
                  intIndex = intIndex - 1
          endwhile

3.        output "Your discount rate is " + fltDISCOUNTS[intIndex]

4.        output "Enter the quantity ordered or " + intQUIT + " to quit"
5.        input intQuantity
6. return
```

```
0. endOfJob()
1.        output "End of Job"
2. return
```

**flowchart



```
Declarations
   num intQuantity
   num intSIZE = 4
   num fltDISCOUNTS[4] = 0, 0.1, 0.15, 0.2
   num intQUANTITYLIMITS[4] = 0, 9, 13, 26
   num intIndex
   num intQUIT = -1
```


determineDiscount()
intIndex = intSIZE - 1
while intQuantity < intQUANTITYLIMITS[intIndex]
F
T
intIndex = intIndex - 1
output "Your discount rate is " + fltDISCOUNTS[intIndex]
output "Enter the quantity ordered or " + intQUIT + " to quit"
input intQuantity
return

Recap + Extra

-- Every array has a finite size

- the number of elements is static (dynamic arrays is a thought that has been discussed in these notes earlier - collections in programming)

-- Array elements are all the same data type

-- Each element then occupies the same number of bytes in memory

-- To access data, we need to use a subscript value - a value that then references the memory location of the cell in the array

-- When an invalid subscript is entered/used

- Most programming language engines will produce a run-time error
- Some engines might access a memory location outside the array (but there is nothing there to be seen)

-- Presenting invalid subscript values in your code is a logical error

-- Mentioned above - when you use a subscript that is not within the range of indexes in the array - you will be presented with an "Array Out of Bounds" error - a run-time error - the processing is paused/broken

-- Your coding should be able to deal with and prevent out of bound errors before they occur

---

** The following information is not immediately apart of the learning unit - but felt it is important for you to have an understanding of the idea of multi-dimensional arrays. The reason for this is because in the not-so-distant future, you will find a need for such programming and the code and lessons shared here will hopefully assist you in your programming journey

===

Two-Dimensional Arrays

===

-- Two-dimensional arrays is a block of memory locations associated with a variable name and designated by row and column numbers.

-- Each element is referenced by 2 indexes and in the array it is written as

**code - Array[row #, column #]

-- The row number is always first and the column number is always second

-- The indexes are again allowed to be a constant or a variable or an expression, and always of an integer data type.

-- Whatever processing takes place - it is important to remember that the indexes are always in the same place, **row index first, column index second**.

-- We usually use 2D Arrays when we have a table of values. They can effectively replace parallel arrays, if the data is of the same datatype.

intNumber

	0	1	2
0	**1**	**2**	**3**
1	**4**	**5**	**6**
2	**7**	**8**	**9**

**code

Declarations

```
        num intNum[3,3]  // 2D Array 3x3 = 9 elements
```

intNumber[2,1] is equal to 8 – row 2 – column 1

intNumber

	0	1	2
0	**1**	**2**	**3**
1	**4**	**5**	**6**
2	**7**	**8**	**9**

?? Entering data

-- We load a 2D array with nested loops. (most common practice is to use 2 for..loop’s)

-- The data is normally loaded row-by-row. This means that the outer loop will represent access to the row and the inner loop will represent the access to the column. This idea will mean that while the row remains constant (when loading the row), the column index number varies (through the inner loop)

```
        **pseudocode
        2.      for intCount1 = 0 to 3 Step 1
                        for intCount2 = 0 to 3 Step 1
                                output  "Enter value for cell [" + intCount1 + "," + intCount2 + "] - "
                                input intNum[intCount1, intCount2]
                        endfor
                endfor
```

- The output (prompt) statements for entering data:

```
                Enter value for cell[0,0] -
                ...
                Enter value for cell[1,2] -
                ...
                Enter value for cell[2,2] -
```

?? Printing

-- In the same way that the array is loaded with data, we would use the same loops to print the data.

-- Given that we are learning our programming using pseudocode, the printed data will not look like table format that we are able to visualize above. However, we can add a few extra lines of code to allow us to understand what it is that is being printed

```
        **pseudocode
        2.      for intCount1 = 0 to 3 Step 1
                        output "Row " + intCount1
                        for intCount2 = 0 to 3 Step 1
                                output  "Column "  + intCount2 + " = " + intNum[intCount1, intCount2]
                        endfor
                endfor
```

- The output (display) statements:

```
                Row 0
                Column 0 = 1
                Column 1 = 2
                Column 2 = 3
                Row 1
                Column 0 = 4
                Column 1 = 5
                Column 2 = 6
                Row 2
                Column 0 = 7
                Column 1 = 8
                Column 2 = 9
```

?? Accumulating

**pseudocode

```
2.      intGrandTotal = 0
3.      for intCount1 = 0 to 3 Step 1
                intRowTotal = 0
                intColTotal = 0
                for intCount2 = 0 to 3 Step 1
                        intGrandTotal = intGrandTotal + intNum[intCount1, intCount2]
                        intRowTotal = intRowTotal + intNum[intCount1, intCount2]
                        intColTotal = intColTotal + intNum[intCount2, intCount1]
                endfor
                output "The total for row " + intCount1 + " is " + intRowTotal
                output "The total for column " + intCount1 + " is " + intColTotal
        endfor
4.      output "The grand total is " + intGrandTotal
```

- The output statements:

```
The total for row 0 is 6
The total for column 0 is 12
The total for row 1 is 15
The total for column 1 is 15
The total for row 2 is 24
The total for column 2 is 18
The grand total is 45
```

intNumber

	0	1	2	
0	1	2	3	6
1	4	5	6	15
2	7	8	9	24
	12	15	18	45

-- Future visualized stories - a mix of 1D and 2D arrays

strAssessments

0	1	2	3	4	5	6
T1	T2	T3	A1	A2	A3	Exam

	strStuds								intFinalMarks	
0	John	19	57	17	86	32	12	44		0
1	Jack	85	36	11	77	74	58	87		1
2	Jane	78	55	7	60	52	53	23		2
3	Joe	29	1	75	49	9	97	64		3
4	Jace	22	30	19	73	28	30	96		4
5	Jennifer	12	95	78	66	20	16	48		5
6	June	5	52	58	35	54	96	21		6
7	Jackie	10	2	70	67	27	78	50		7
8	Joan	15	28	9	87	73	99	49		8
9	Jonathon	0	90	86	61	92	83	69		9
10	Jessie	24	8	53	56	100	69	100		10
11	Jessica	81	51	35	32	31	83	47		11
12	Johan	28	77	96	15	84	58	3		12
13	Joesph	54	26	56	41	73	82	11		13
14	Jethro	57	97	68	49	16	42	20		14

fltAvgMarks							
intHighMarks							
intLowMarks							
	0	1	2	3	4	5	6