String and Math Functions

In this Chapter:

- ❑ Standard String-Handling Functions
- ❑ Adding New String-Handling Functions
- ❑ Updating the StringLibrary File
- ❑ Understanding Cartesian Coordinates
- ❑ Math Functions
- ❑ Using sin() and cos() to Calculate Coordinates

String Functions

Introduction

Unlike numeric variables which hold only a single value, strings can hold a whole collection of characters, perhaps several words or even sentences. For example, it's quite valid to store a line of text in a string variable with a statement such as:

```
poem$ = "Mary had a little lamb"
```

Because a string can contain so many characters, there are several operations that programmers find themselves needing to do with strings. For example, we might want to find out how many characters are in a string, convert a string to uppercase, or extract part of a string.

AGK BASIC contains a number of string-handling functions as part of the core language. These functions are listed and explained in the first part of this chapter.

String-Handling Functions

Len()

The `Len()` function returns the number of characters in a string. The string to be examined is given in parentheses. For example, the expression

```
Len("Hello")
```

would return the value 5 since there are 5 characters in the word *Hello*.

Spaces and any other non-alphabetic symbols within a string also count as characters, so the line

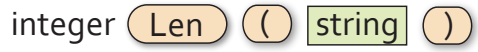
```
Len("Hello, world?")
```

would return the value 13, since it will include the comma, space and question mark in the count.

FIG-9.1

Len()

The `Len()` function has the format shown in FIG-9.1.



where :

string is a string constant, string variable, or string expression.

As with any function that returns a value, this value can be displayed, assigned to a variable, or used in an expression. Hence each of the following lines are valid:

```
Print(Len("Hello"))        'displays 5
result = Len("Hello")      'sets result equal to 5
ans = Len("Hello") *3      'sets ans to 15 (5 x 3)
if Len("Hello") > 3        'condition is true since 5 > 3
```

Of course, it's much more likely that you'll use a string variable as an argument rather than a string constant.

Activity 9.1

In this Activity we are going to make use of our StringLibrary.agc *file which we placed in the* Library *folder in the last chapter.*

Start a new project called *TestLen*. Compile the default code.

Using Windows Explorer, make a copy of the *StringLibrary.agc* file found at *HandsOnAGK/Library* and paste it into the *TestLen* folder.

Modify the contents of *main.agc* to read:

```
rem *** Test Len() Function ***
rem *** Include Library function ***
#include "StringLibrary.agc"

rem *** Generate string ***
text$ = RandomString(-1)
rem *** Print string and its length ***
Print(text$)
Print(Len(text$))
Sync()
do
loop
```

Test and save your program.

Notice that in order to use the *RandomString()* function, it is necessary to add a

```
#include "StringLibrary.agc"
```

command at the start of the program.

Upper()

The `Upper()` function takes a string argument and returns the uppercase version of that string. For example, the line

```
Print(Upper("Hello"))
```

would display the word *HELLO*.

Any characters in the string that are not letters are returned unchanged by this statement. Hence,

```
Print(Upper("Abc123"))
```

would display *ABC123*.

Activity 9.2

What would be the value of *b$* after the following lines are executed?

```
a$ = "1-by-1"
b$ = Upper(a$)
```

FIG-9.2

Upper()

The `Upper()` statement has the format shown in FIG-9.2.

where:

string is any string value.

Lower()

The `Lower()` function takes a string argument and returns the lowercase version of that string. Any non-alphabetic characters in the string are returned unchanged.

```
Print(Lower("Hello"))
```

would display the word *hello*.

FIG-9.3

Lower()

This statement has the format shown in FIG-9.3.

where:

string is any string value.

Left()

It's possible to extract the left-hand section of a string using the `Left()` function. This time you need to include two parameters: the first is the string itself, and the second is the number of characters you want to extract. For example,

```
Print(Left("abcdef",2))
```

would display *ab* on the screen, `Left()` having returned the left two characters from the string *abcdef*.

If the number given is larger than the number of characters in the string as in

```
ans$ = Left("abcdef",10)
```

then the complete string is returned (i.e. *abcdef*)

Should a zero, or negative value be given as in

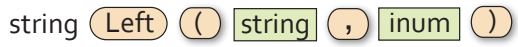
```
result$ = Left("abcdef",0)
```

then the returned string contains no characters. That is, *result$* will hold an empty string.

FIG-9.4

Left()

The `Left()` function has the format shown in FIG-9.4.



where:

string is any string value.

inum is a positive integer value giving the number of characters to be copied. It should be in the range 0 to the number of characters in the string.

Right()

If we want to extract the right-hand part of a string we can use the `Right()` function. For example, the statement

```
Print(Right("abcdef",2))
```

would display *ef* on the screen.

The statement has the format shown in FIG-9.5.

FIG-9.5

Right()

where:

string is any string value.

inum is a positive integer value giving the number of characters to be copied. It should be in the range 0 to the number of characters in the string.

Mid()

This statement extracts a substring from the specified string. The position of the first character and the number of characters to be extracted is given as the second and third arguments to the function. For example, the statement

```
letter$ = Mid("abcdef",4,2)
```

would place the value *de* in *letter$* (extracts 2 characters starting at the 4th character in the string). We can use this statement to access each character in a string. For example, the code snippet

```
text$ = RandomString(-1)
for c = 1 to Len(text$)
   Print(Mid(text$,c,1))
next c
Sync()
```

will display each character of the string stored in `text$` on a separate line.

Activity 9.3

Create a new project, *Letters*, which makes use of the code above to display a generated string and then displays the individual characters of the string. Remember to copy the *StringLibrary.agc* file into the project folder and add a `#include` instruction to your code.

Modify the program so that the characters are displayed in reverse order on a single line.

Change the program so that, rather than display the characters, it counts how many E's are in the string.

FIG-9.6

Mid()

The format for the `Mid()` statement is given in FIG-9.6.

where:

string is any string value.

ipost is a positive integer giving the position of the first character to be extracted. Range 1 to length of string.

inum is a positive integer giving the number of characters to be copied.

Asc()

ASCII character codes are given in Appendix A at the end of the book.

This function returns an integer value representing the ASCII value of the first character in the string supplied. A typical statement such as

```
Print(Asc("ABC"))
```

would display the value 65 since that is the ASCII code for a capital A. Using this function on an empty string as in the line

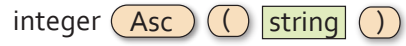
```
result = Asc("")
```

returns the value zero.

FIG-9.7

Asc()

The format for this statement is given in FIG-9.7.



where:

string is any string value, but only the first character is used by the function.

Chr()

The `Chr()` function complements the `Asc()` function by returning the character whose ASCII code matches the specified value. For example, the line

```
Print(Chr(65))
```

would display a capital letter A since the ASCII code for a capital A is 65.

ASCII 32 is the space character. So although it is displayable, there's not much to see!

The value given should lie between 0 and 127. However, only characters with an ASCII code of 32 to 126 are displayable; other values are used for various control purposes and attempting to display such values has no visible effect in AGK BASIC.

We could display all the letters of the alphabet in uppercase using the lines:

```
for c = 1 to 26
   Print(Chr(64+c))
next c
Sync()
```

FIG-9.8

Chr()

The format for this statement is given in FIG-9.8.

where:

ivalue is an integer value. This value must be between 0 and 127, but is more likely to be between 32 and 126, these being the ASCII range of values for all displayable characters.

Activity 9.4

Create a new project called *ASCIITable*.

Code the program so that it displays the numbers 32 to 126 and, beside each number, the corresponding ASCII character.

Get the program to pause after every 25 characters, waiting for 5 seconds before continuing.

Test and save your program.

Str()

The `Str()` function takes a numeric argument and returns a string containing the same digits as the argument. For example, the line

```
result$ = Str(123)
```

will store the string *123* in the variable *result$*

When converting a real value, the number of decimal places required can be specified, as in the line

```
value$ = Str(12.326,2)
```

which will store *12.33* in *value$*. Note that the last digit is rounded.

Perhaps the most useful application of this function is to simplify output involving several values. For example, in past programs we have had to write code such as:

```
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
```

Using the `Str()` function we can now rewrite this as:

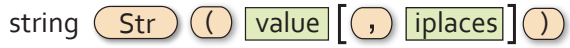
The + operator is used to join two strings.

```
Print("My number was : " + Str(dice))
Print("Your guess was : " + Str(guess))
```

FIG-9.9

Str()

This statement has the format shown in FIG-9.9.



where:

value is any numeric value.

iplaces is an integer value giving the number of decimal places to be stored.

Activity 9.5

Start a new project called *CountZero* and write a program to generate a random number between 1000 and 65000 which displays the number generated and a count of how many zeros are in that number.

(HINT: Convert the number to a string then count the number of zeros in that string.)

Test and save your project.

Val()

This function takes a string argument and returns the integer equivalent. The string should contain only numeric characters. For example, executing the line

```
result = Val("123")
```

will store the value 123 in the variable *result*.

If the string contains a real value, only the integral part will be converted. So the line

```
ans# = Val("123.45")
```

gives *ans#* a value of 123.0 when displayed.

If the string contains a mixture of numeric and non-numeric characters, the value returned is constructed from all numeric characters preceding the first non-numeric character. For example, the call

```
Val("12ABC3")
```

returns the value 12.

If the string starts with a non-numeric character (other than a sign or decimal point) then the function returns zero.

The string may hold a value which represents a number in a different number base. But when the value is not a base 10 number, we need to add a second parameter giving the number's base. So for example, we could convert a string representing a hexadecimal number using the line:

```
num = Val("FE",16)
```

which would store the value 254 (the decimal equivalent of FE_{16}) in *num*.

Although the most obvious number bases for a computer system are 2 (binary), 16 (hexadecimal) and 8 (octal), you can have any integer base you wish. For example, the statement

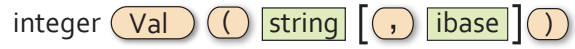
```
v = Val("210",3)
```

states that 210 is a base 3 number and therefore *v* will be set to 21 (2*9+1*3+0*1).

FIG-9.10

Val()

The `Val()` function has the format shown in FIG-9.10.



where:

string is a string containing only numeric characters, a decimal point, or a sign (+ or -).

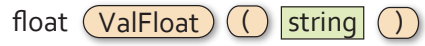
ibase is an positive integer value giving the number base.

ValFloat()

To convert a string to a real number, use `ValFloat()` (see FIG-9.11).

FIG-9.11

ValFloat()



where:

string is a string containing only numeric characters, a decimal point, or a sign (+ or -).

Space()

Although it is easy enough to create a string full of spaces with a line such as

```
text$ = "          "
```

if you want an exact number of spaces in your string, then it's easier to use the `Space()` function which returns a string containing a specified number of spaces.

```
text$ = Space(23)
```

assigns a string containing 23 spaces to the variable *text$*. The format for this statement is shown in FIG-9.12.

FIG-9.12

Space()

where:

ivalue is a positive integer which specifies the number of spaces in the string returned by the function.

Bin()

As you know, the computer uses the binary number system when storing programs and data. If you'd like to see what a specific integer value looks like in binary, this function will do the job for you. It returns a string showing the binary representation of a specified integer value. For example, the instruction

```
binary$ = Bin(65)
```

would assign the string *1000001* (the binary equivalent of 65) to the variable *binary$*.

If a negative value is used, the string returned is in 2's complement form. This means that the instruction

```
Print(Bin(-65))
```

would display the string *11111111111111111111111110111111*. The format for the `Bin()` function is shown in FIG-9.13.

FIG-9.13

Bin()

where:

ivalue is an integer value.

Hex()

Another widely used number system is hexadecimal which uses the letters A to F to represent values 10 to 15. The `Hex()` function returns a string containing the hexadecimal equivalent of a specified integer value. For example,

```
hexadecimal$ = Hex(65)
```

assigns the string *41* to the variable *hexadecimal$*.

For negative values, the hexadecimal string returned is the equivalent of the 2's complement form. Therefore,

```
Print(Hex(-15))
```

displays the string *FFFFFFF1*.

FIG-9.14

Hex()

The format of this function is shown in FIG-9.14.

where:

ivalue is an integer value.

Activity 9.6

The three Buttons files can be found in the *TestButtons* project.

Create a new project, *Conversions*.

Use the Button functions to read in an integer value and then display the equivalent binary and hexadecimal value. Test and save your project.

CountStringTokens()

The string *red,green,blue,yellow* contains the names of four colours, each name being separated by a comma. AGK refers to each of the terms in the string as **tokens** (similar to the idea of tokens within a line of program code) and the character used to separate those tokens (in this case, a comma) as a **delimiter**.

It doesn't matter what characters make up a token, nor which character is used as a delimiter. In fact, you can use several different delimiters in the same string.

FIG-9.15

CountStringTokens()

The function `CountStringTokens()` can be used to find out how many tokens are in a specified string. The statement has the format shown in FIG-9.15.

where

string is a string containing the characters to be processed.

sdelimits is a string giving the delimiters to be assumed when identifying the tokens.

For example, the statement

```
Print(CountStringTokens("red,green,blue,yellow,white",","))
```

will display the value 5.

The line

```
Print(CountStringTokens("1/2:3|4","/:|"))
```

will display the value 4. In this case any of the characters / : or | are taken as delimiters.

GetStringToken()

FIG-9.16

GetStringToken()

To extract the identified tokens from a string we can use the `GetStringToken()` statement (see FIG-9.16).

where

string is a string containing the characters to be processed.

sdelimits is a string giving the delimiters to be assumed when identifying the tokens.

indx is an integer giving the number of the token to be returned (the first token is at position 1).

For example, the line

```
Print(GetStringToken("red/green/blue/yellow","/",3))
```

would display the term *blue* (the third token in the string).

The program in FIG-9.17 displays the number of tokens in a string and then lists them separately.

FIG-9.17

Using the *StringToken* Functions

```
rem *** Using Tokens ***

rem *** Set string and delimiters ***
quote$ = "It is a truth universally acknowledged, that a single
↳man in possession of a good fortune, must be in want of a wife"
delimiters$=" ,"  //Space and comma
```

FIG-9.17
(continued)

Using the *StringToken* Functions

```
rem *** Get and display token count ***
tokens = CountStringTokens(quote$,delimiters$)
Print(tokens)

rem *** Display each token ***
for c = 1 to tokens
    Print(GetStringToken(quote$,delimiters$,c))
next c
Sync()
do
loop
```

Activity 9.7

Start a new project called *Tokens* and implement the code given in FIG-9.17.

Test your code. Test the program again with a quote and delimiters with options of your own. Save your project.

Creating Your Own String Functions

There are several more operations which would be useful to have when manipulating strings, and, although AGK BASIC does not contain commands to perform these operations, we can easily write them ourselves. Some of these are described below.

Pos()

The *Pos()* function returns the position of a specified character in a specified string. For example, the line

```
place = Pos("abcd","c")
```

would assign the value 3 to *place*, since *c* occurs at position 3 in the string *abcd*.

If the character being searched for occurs more than once in the string, then it is the position of the first occurrence that is returned. Hence, the call

```
Pos("abcdc","c")
```

would return the value 3, not 5. If the character being searched for does not occur within the string, then a value of 0 is returned. The mini-spec for this function is:

FUNCTION NAME	:	Pos
PARAMETERS		
In	:	s : string f : character
Out	:	result : integer
PRE-CONDITION	:	None
DESCRIPTION	:	*result* is set to the position at which *f* first occurs in *s*. If *f* does not occur in *s*, then *result* is set to zero.

The code for this function is shown in FIG-9.18.

FIG-9.18

The Pos() Function's Code

```
rem *** Find Position of character in string ***

function Pos(s$, f$)
   rem *** result stays at 0 if no match found ***
   result = 0
   rem *** Make sure we're looking for a single character ***
   first$ = Mid(f$,1,1)
   rem *** FOR each character in s$ DO ***
   for c = 1 to Len(s$)
      rem *** IF that character matches what we're after THEN ***
      if Mid(s$,c,1) = first$
         rem *** Set result to this position and exit loop ***
         result = c
         exit
      endif
   next c
endfunction result
```

Because AGK BASIC allows only string variables and not single character ones (as some other languages offer), we cannot be sure that when the function *Pos()* is called, the second argument, *f$*, contains only a single character. For example, the line

```
result = Pos("abcdef","ei")
```

would be valid, even though there is more than one character in the second parameter. But by including the line

```
first$ = Mid(f$,1,1)
```

in the code for *Pos()*, we extract the first character from *f$*. It is this first character that we then search for in *s$*.

Activity 9.8

Start a new project called *FunctionTester*.

Copy the file *StringLibrary.agc* from the *Library* folder into the new project's folder.

In *main.agc*, add the code for function *Pos()* as given in FIG-9.18.

In the main part of the program, create a random string 30 characters in length and use a call to *Pos()* to display the first occurrence of a capital *D* within the random string. The generated string should also be displayed so you can check that the result from *Pos()* is correct.

Check that *Pos()* also works when the character searched for cannot be found.

Save your project.

Pos() is another function that could prove useful in later projects, so it will be worth adding its code to the *StringLibrary.agc* file in the *Library* folder.

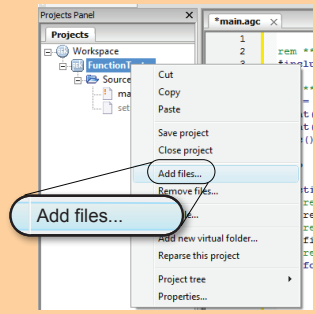
The only thing we have to watch out for here is that we paste the code into the original *StringLibrary.agc* file held in the *Library* folder. FIG-9.19 shows how to

update the original *StringLibrary.agc* file.

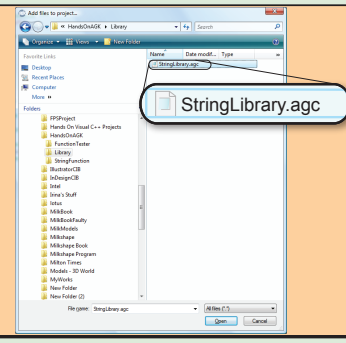
FIG-9.19

Adding a New Function to *StringLibrary.agc*

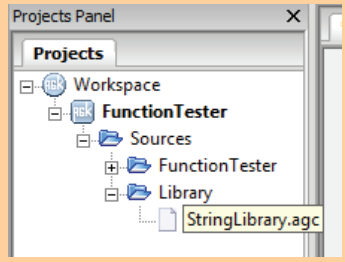
In the Projects Panel, right-click on the project name and select **Add files** from the pop-up menu.



Next, select *StringLibrary.agc* from the *Library* folder.



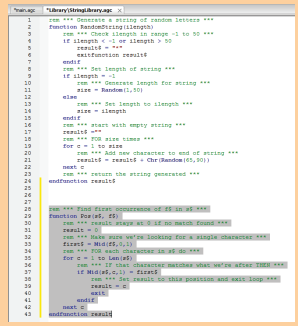
The file is now included in the *Sources* section of the project.



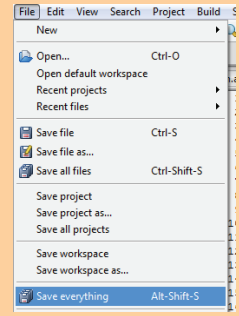
From *main.agc*, we copy the code for *Pos()*.

```
function Pos(s$, f$)
    rem *** result stays at 0 if no match found ***
    result = 0
    rem *** Make sure we're looking for a single character ***
    first$ = Mid(f$,0,1)
    rem *** FOR each character in s$ do ***
    for c = 1 to Len(s$)
        rem *** IF that character matches what we're after THEN ***
        if Mid(s$,c,1) = first$
            rem *** Set result to this position and exit loop ***
            result = c
            exit
        endif
    next c
endfunction result
```

Double clicking on *StringLibrary* in the **Projects Panel** will open the file in the edit area and we can paste the code for *Pos()* to the file.



Finally, selecting **File|Save everything** from the main menu will save the updated *StringLibrary* file.



Activity 9.9

Update the contents of the *StringLibrary.agc* file by adding the *Pos()* function as described in FIG-9.19.

Occurs()

The *Occurs()* function returns how often a specified character appears within a

specified string. Hence, the expression

```
Occurs("abcdc","c")
```

would return 2 since *c* occurs twice within *abcdc*. The mini-spec for the routine is:

FUNCTION NAME	:	Occurs
PARAMETERS		
In	:	s : string f : character
Out	:	result : integer
PRE-CONDITION	:	None
DESCRIPTION	:	*result* is set to the number of times *f* occurs in *s*.

The code for this function is shown in FIG-9.20.

FIG-9.20

The Occurs() Function's Code

```
rem *** Return how often f$ occurs in s$ ***
function Occurs(s$,f$)
   rem *** None found so far ***
   result = 0
   rem *** Make sure only one character ***
   first$ = Mid(f$,0,1)
   rem *** FOR each character in s$ Do ***
   for c = 1 to Len(s$)
      rem *** if it matches req'd character, add 1 to result ***
      if Mid(s$,c,1) = first$
         result = result + 1
      endif
   next c
endfunction result
```

Activity 9.10

Add the code for *Occurs()* to *main.agc* in *FunctionTester*.

In the main part of the program, create a random string 30 characters in length and use a call to *Occurs()* to display how often a capital *S* appears within the random string. The generated string should also be displayed so you can check that the result from *Occurs()* is correct.

Save your project. Add the code for *Occurs()* to *StringLibrary.agc* in the *Library* folder.

Insert()

The `Insert()` function returns a string created by inserting one string into another, starting at a specified position. For example, the line

```
Print(Insert("abcdef ","xy", 4))
```

would display the string *abcxydef* having inserted the string *xy* into string *abcdef* starting at position 4.

If an attempt is made to insert the second string at an invalid position, then the

returned string is an exact match of the first string.

The function's mini-spec is:

```
FUNCTION NAME   :   Insert
PARAMETERS
   In           :   s      : string
                    f      : string
                    p      : integer
   Out          :   result : string

PRE-CONDITION   :   None

DESCRIPTION     :   result is created by inserting f into s at position p.
                    Normally, p should be in the range 1 to Len(s)+1.
                    If p is outside this range result is set equal to s.
```

The code for this routine is given in FIG-9.21.

FIG-9.21

The Insert() Function's Code

```
rem *** Returns string with f$ inserted at position p into s$ ***

function Insert(s$,f$,p)
   rem *** If invalid position, result is original string ***
   if p < 1 or p > Len(s$)+1
      result$ = s$
   else
      rem *** split s$ into two parts & insert f$ in between ***
      result$ = Left(s$,p-1)
      result$ = result$ + f$
      result$ = result$+ Right(s$,Len(s$)-(p-1))
   endif
endfunction result$
```

Notice that the main logic in the function involves splitting the first string into two parts and inserting the second string in between these parts.

Activity 9.11

Add the code for *Insert()* to *main.agc* in *FunctionTester*.

In the main part of the program, call *Insert()* to add *XX* to *ABCDEFGHI* starting at position 2.

Test and save your project.

Also check that the function performs as specified if the insert position given is invalid.

Add the code for *Insert()* to *StringLibrary.agc* in the *Library* folder.

Delete()

The *Delete()* function returns a string created by deleting a specified section of an original string. For example, the line

```
temp$ = Delete("abcdefghi",2,4)
```

would set *temp$* to *afghi* this being created by removing 4 characters, starting at position 2, from the original string *abcdefghi*.

If the start position is invalid, a copy of the original string is returned. If the number of characters to be deleted is too large, then as many characters as possible are removed.

The function's mini-spec is:

FUNCTION NAME	:	Delete	
PARAMETERS			
In	:	s	: string
		st	: integer
		num	: integer
Out	:	sresult	: string
PRE-CONDITION	:	None	
DESCRIPTION	:	*sresult* is equal to *s* with the *num* characters deleted starting from position *st*.	
		If *st* is outside the range 1 to *Len(s)*, *sresult* is equal to *s*.	
		If *num* > *Len(Right(s,Len(s)-st+1))*, *sresult* is set to *Left(s,st-1)*.	

Notice how the mini-spec makes use of other string-handling functions to describe how the value of *result* is determined. Although more difficult to understand than plain English, this approach can often lead to a briefer description and will always be unambiguous.

The code for this routine is given in FIG-9.22.

FIG-9.22

The Delete() Function's Code

```
rem *** Returns string created by deleting num chars ***
rem *** from s$ starting at position st              ***

function Delete(s$, st, num)
   rem *** if invalid position, result is original string ***
   if st < 1 or st > Len(s$)
      result$ = s$
   else
      rem *** Set result to the part of s$ to the left of ***
      rem *** the section to be deleted                   ***
      result$ = Left(s$, st-1)
      rem *** IF not deleting to the end of s$,           ***
      rem *** add right section                           ***
      if st+num-1 <= Len(s$)
          result$ = result$+Right(s$,Len(s$)-(st+num-1))
      endif
   endif
endfunction result$
```

Activity 9.12

Add the code for *Delete()* to *main.agc* in *FunctionTester*.

In the main part of the program, call *Delete()* to delete from position 3 the next 5 characters. Use *ABCDEFGHI* as the string.

Test and save your project.

Also check that the function performs as specified if the start position given is invalid and when more characters than available are to be deleted.

Add the code for *Delete()* to *StringLibrary.agc* in the *Library* folder.

Replace()

The *Replace()* function is designed to return a string constructed by replacing a single character at a specified position in an original string. Therefore the line

```
ans$ = Replace$("abcdef","x",4)
```

sets *ans$* equal to *abcxef* having replaced the fourth character in *abcdef* with an *x*.

If an invalid position is specified, then the original string is returned.

Activity 9.13

Create a mini-spec for the *Replace()* function.

Using the *FunctionTester* project, write code for the *Replace()* function and then test your coding.

Add the code for *Replace()* to *StringLibrary.agc* in the *Library* folder.

Summary

- The `Len()` function returns the number of characters in a specified string.
- The `Upper()` function returns the uppercase equivalent of a specified string.
- The `Lower()` function returns the lowercase equivalent of a specified string.
- The `Left()` function returns a left-hand sub-string from a specified string.
- The `Right()` function returns a right-hand sub-string from a specified string.
- The `Mid()` function returns a specified number of characters from a given position in a specified string.
- The `Asc()` function returns the ASCII code of a specified character.
- The `Chr()` function returns the character whose ASCII code matches a specified value.
- The `Str()` function returns the string equivalent of a specified number.
- The `Val()` function returns the numeric equivalent of a specified string.

- The `Space()` function returns a string containing a specified number of spaces.
- The `Bin()` function returns a string representing the binary equivalent of a specified integer.
- The `Hex()` function returns a string representing the hexadecimal equivalent of a specified integer.
- Use `CountStringTokens()` to count the number of tokens in a string.
- Use `GetStringToken()` to extract a specific token from a string.

Math Functions

Introduction

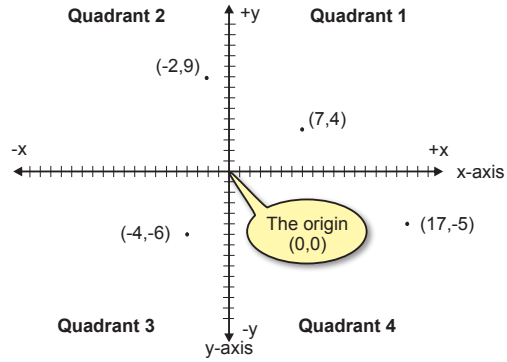
A second important group of standard programming functions is the math functions. All of the math functions not previously covered are given here.

Coordinates

In 2D coordinate geometry objects are positioned by specifying x,y **Cartesian coordinates** (see FIG-9.23).

FIG-9.23

Cartesian Coordinates



From FIG-9.22 we can see that the origin is the position where the two axes cross and that the axes split the area into four quadrants:

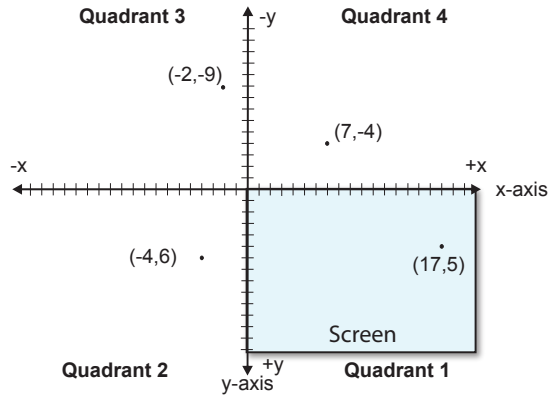
- ➢ quadrant 1: both x and y values are positive
- ➢ quadrant 2: x values are negative and y values positive
- ➢ quadrant 3: both x and y values are negative
- ➢ quadrant 4: x values are positive and y values negative

However, on a computer screen, the y axis has been turned upside down so that positive y values are at the bottom while negative y values are at the top (see FIG-9.24). Also, the top-left point on the screen is taken as the origin so a screen displays only quadrant 1 points.

FIG-9.24

Screen Coordinates

The area shown as the screen is not to scale.



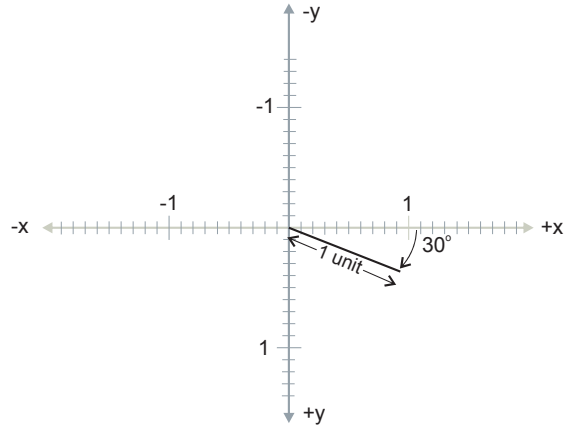
This modification changes the position of the four quadrants. We'll be using this inverted coordinate system, since that's the one we need when creating games.

Cos()

If we draw a line starting at the origin which is exactly one unit in length at an angle of 30° to the x-axis, then we create the setup shown in FIG-9.25.

FIG-9.25

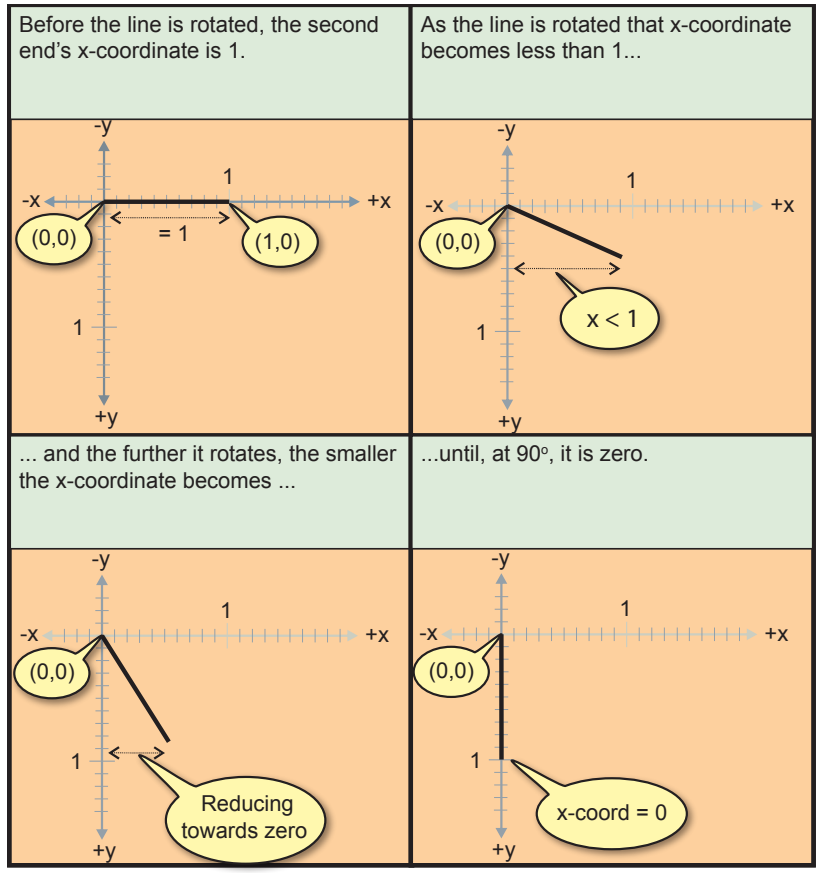
Measuring Angles



We know that one end of the line has the coordinates (0,0), but what are the coordinates of the other end? We'll start by examining the x-coordinate. From FIG-9.26 we can see that the x-coordinate of the end point changes as we rotate the line to 70° from the x-axis and finally to 90°.

FIG-9.26

X-Coordinate Determined by Angle



Activity 9.14

What would be the x-coordinate of the line if it was rotated to

a) 0° b) 90°

Although it is easy enough to work out the x-coordinate when the line lies along one of the axes, things are a bit more difficult when some other angle of rotation is involved. Luckily for us, someone worked all the x-coordinates for every possible angle several hundred years ago and called it the **cosine** of the angle (often shortened to **cos**).

So, if we draw a line starting at the origin which is 1 unit in length and rotate it by an angle of theta (θ), then the x-coordinate for the other end of that line is given by the expression

cosine(θ) or `cos(θ)`

If we rotate our line by more than 90° it moves into quadrant 2 and the x-coordinate will become negative. As we pass 180° and move into quadrant 3, the x-coordinate remains negative, but after 270°, the x-coordinate is once again positive.

Activity 9.15

Load up Microsoft's *Calculator* program.

Choose **View|Scientific.** Make sure it is using decimal and degrees.

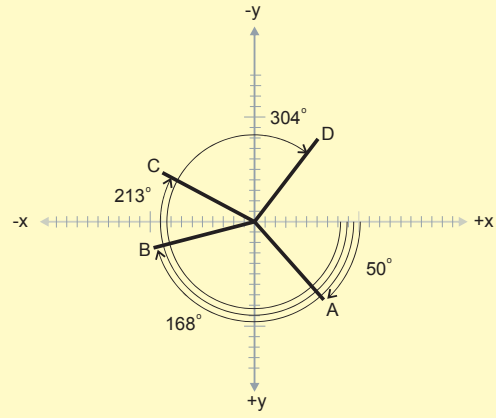
By calculating the cosine of the following angles (to 3 decimal places)

a) 0° b) 90° c) 30° d) 70°

write down the x-coordinate of the lines of 1 unit which have been rotated by the angles given above.

Activity 9.16

By using the cosine function in *Calculator*, determine the x-coordinates of the lines shown below (all lines start at the origin and are 1 unit in length).



AGK BASIC performs this calculation using the `Cos()` function which has the format shown in FIG-9.27.

FIG-9.27

Cos()

where:

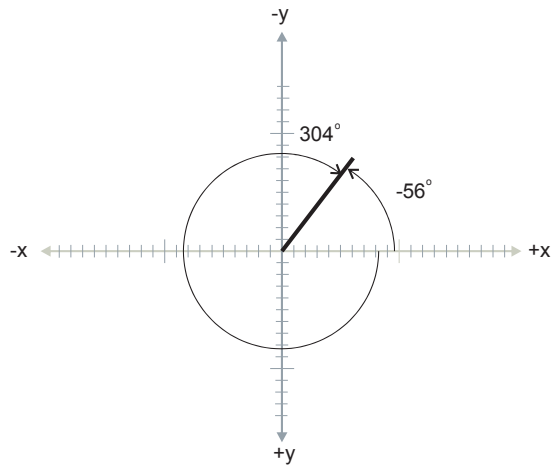
angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

The real value returned by the function gives the x-coordinate of one end of the rotated line (the other end being at the origin).

The angle through which the line has been rotated may also be measured in a counter-clockwise direction, but is then specified as a negative value. This means that the expressions `Cos(304)` and `Cos(-56)` both return the same value (see FIG-9.28).

FIG-9.28

Clockwise and Counter-Clockwise Angles



Sin()

To determine the y-coordinate of our one unit line, we use the `Sin()` function which has the format shown in FIG-9.29.

FIG-9.29

Sin()

where:

angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

The real value returned by the function gives the y-coordinate of one end of the rotated line (the other end being at the origin).

Activity 9.17

Using *Calculator*, write down the y-coordinates of the four lines shown in Activity 9.16.

Dealing with Longer Lines

It's all very well to calculate the end point of a line which is one unit in length, but what about lines that are 2, 4 or 7.5 units long?

Actually, the calculation required is quite simple: if the line is twice as long, the coordinates of its end point are twice the value of those for a one unit line. If the line is four times longer, then the coordinate values are four times as large.

All of this can be simplified to:

```
x-coordinate = length of line * cos(θ)
y-coordinate = length of line * sin(θ)
```

Activity 9.18

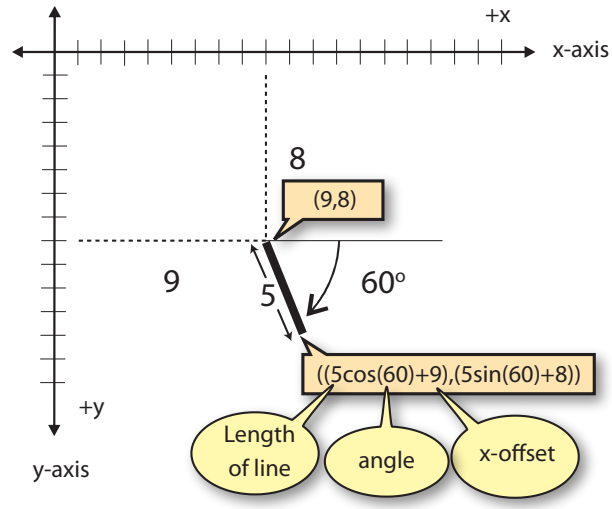
If a line is drawn from the origin and is 3.7 units in length, what are the coordinates of its end point after it has been rotated to an angle of 191.5° ?

Offset Lines

If a line whose fixed end is not positioned at the origin is rotated, calculating the coordinates of the moving end is done by calculating the x and y coordinates as before but then adding the x and y offset values to the results (see FIG-9.30).

FIG-9.30

Offset Lines



Activity 9.19

Calculate the actual coordinates of the rotating end of the line shown in FIG-9.29.

Using Cos() and Sin()

School may teach you the sine and cosine functions for no obvious practical reason, but when it comes to games programming, these are important operations. Using the `Sin()` and `Cos()` functions allows us to perform many operations on the screen graphics. For example, to rotate a sprite about a point on the screen. The program code in FIG-9.31 demonstrates how this is done by rotating a spot-shaped image

about the centre of the screen.

FIG-9.31

Rotating a Sprite

```
rem *** Load image ***
LoadImage(1,"Spot.png")
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Size sprite ***
SetSpriteSize(1,5,-1)
rem *** Position sprite offset from screen centre ***
SetSpritePosition(1,70,50)
angle = 0
do
    angle = (angle+1) mod 360
    SetSpritePosition(1,20*cos(angle)+50,20*sin(angle)+50)
    Sync()
loop
```

Activity 9.20

Start a new project called *Rotation*.

Compile the default code in order to create the *media* subfolder.

From the files you download with this book, copy the file *Spot.png* from the *AGKDownloads/Chapter9* folder into this project's *media* folder.

Change *main.agc* to match the code shown in FIG-9.31.

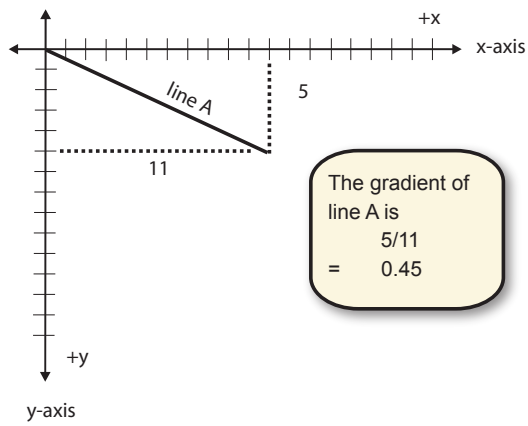
Test and save your project.

Tan()

The last of the traditional trigonometric functions is **tangent** or **tan**. Tangent measures the gradient (or steepness) of a line. Gradient of a line with one end fixed at the origin is just the y-coordinate of the other end of the line divided by the x-coordinate (see FIG-9.32).

FIG-9.32

Gradients



So a line parallel to the x-axis has a gradient of zero, a line at 45° to the x-axis has a gradient of 1 and a line at 90° has an infinite gradient. The `Tan()` function takes the angle of the line and returns its gradient.

FIG-9.33

Tan()

AGK's *Tan()* function has the format shown in FIG-9.33.

where:

angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

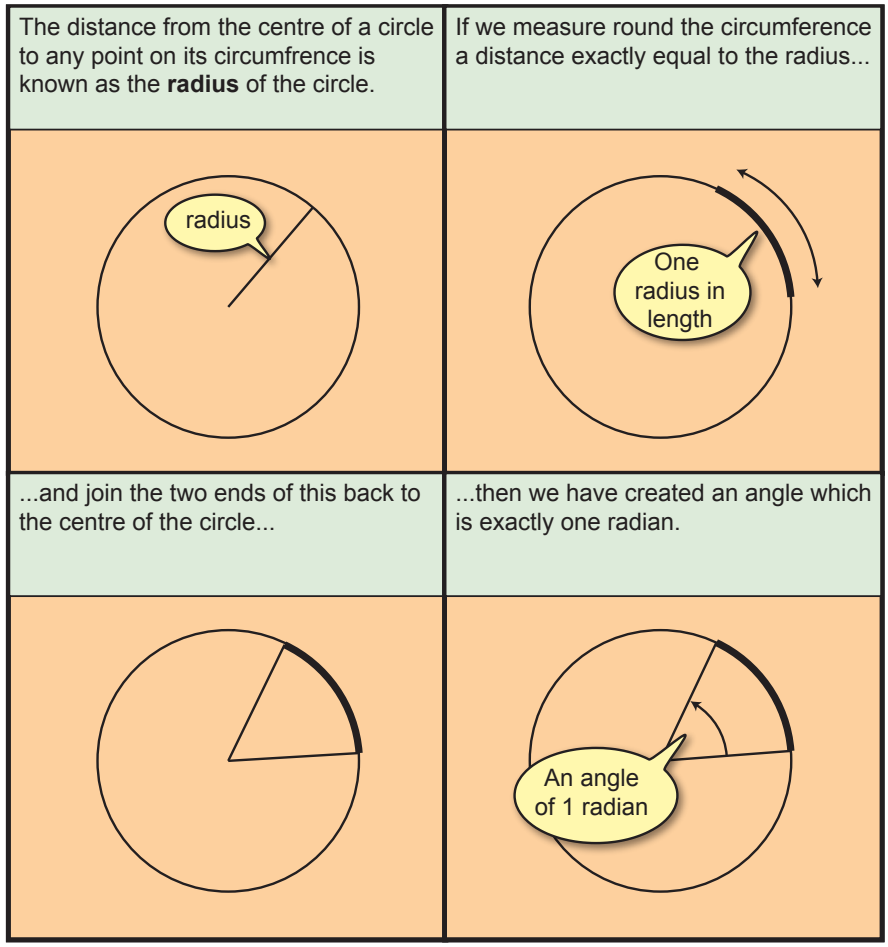
Degrees and Radians

No one knows with certainty why a full circular rotation is divided into 360 units known as **degrees** (or **degrees of arc**, to give them their full title). One theory is that ancient Persian civilizations used a calendar of 360 days (indicating a full rotation of the Earth about the Sun).

An alternative unit of measurement is the **radian**. FIG-9.34 explains how this measurement is derived.

FIG-9.34

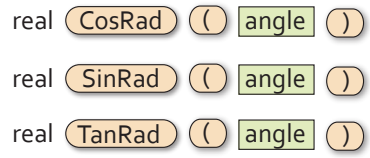
Radians



AGK BASIC has a set of functions equivalent to `Cos()`, `Sin()` and `Tan()` called `CosRad()`, `SinRad()` and `TanRad()` which take an angle given in radians rather than degrees. The syntax for all three of these functions is shown in FIG-9.35.

FIG-9.35

CosRad()
SinRad()
TanRad()



where:

angle is a real number specifying the angle (in radians) through which the line has been rotated.

ACos(), ASin() and ATan()

If you already know the *x* or *y* coordinates of the end point of a line or the gradient of a line, but want to know the angle, then we can make use of the `ACos()`, `ASin()` or `ATan()` functions respectively. For example, looking back at FIG-9.31 we can see the gradient of the line is 5/11 but we don't know the angle the line makes to the x-axis. We can find out using the line

The mathematical names for these functions **arccosine**, **arcsine** and **arctangent**.

```
angle = ATan(5/11)
```

In Activity 9.16 we discovered the coordinates of point A (which was rotated by 50°) to be (0.643,0.766). Using the x-coordinate only, the line

```
angle = ACos(0.643)
```

will give a result of (approximately) 50. And using the y-coordinate only

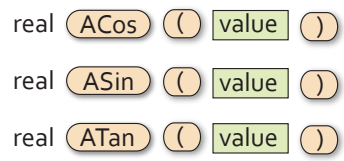
```
angle ASin(0.766)
```

will also give the same result.

The syntax of the three statements are given in FIG-9.36.

FIG-9.36

ACos()
ASin()
ATan()



where:

value is a real number.

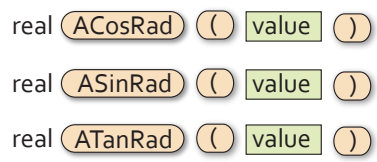
The value returned represents an angle given in degrees.

ACosRad(), ASinRad(), ATanRad()

If you need the angles to be returned in radians rather than degrees, you can make use of the `ACosRad()`, `ASinRad()` and `ATanRad()` functions which are shown in FIG-9.37.

FIG-9.37

ACosRad()
ASinRad()
ATanRad()



where:

value is a real number.

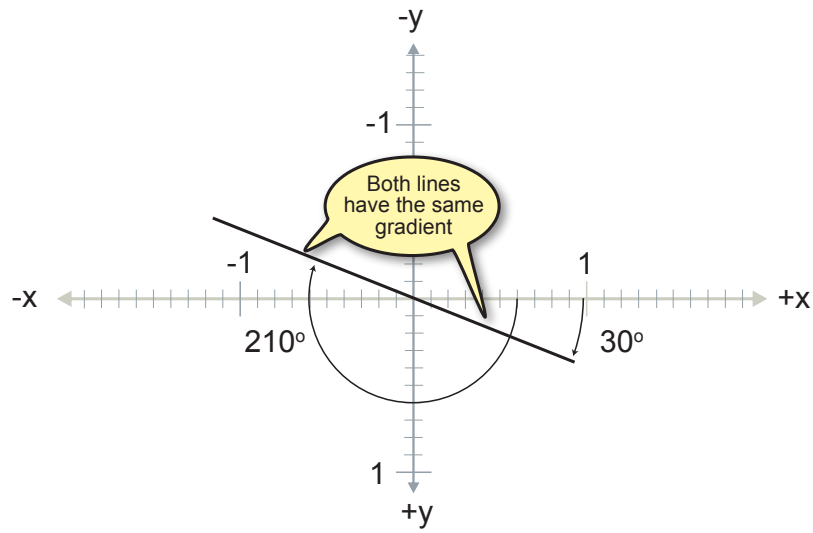
The value returned represents an angle given in radians.

ATanFull() and ATanFullRad()

The main problem with `ATan()` is that it cannot guarantee that the angle returned is the the correct one. Every gradient can be reproduced at exactly two angles. For example, a line at 30° and one at 210° have the same gradient (See FIG-9.38).

FIG-9.38

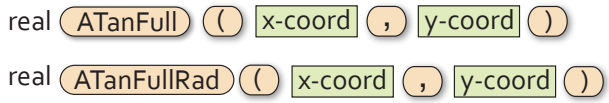
Angles and Gradients



The only way to differentiate between the two lines is to specify the end points of the line rather than the gradient value. This is the option offered by the `ATanFull()` and `ATanFullRad()` functions. The first returns the angle of the line in degrees, the second, in radians. The functions have the format shown in FIG-9.39.

FIG-9.39

AtanFull()
ATanFullRad()



where:

xcoord is a real number giving the x-coordinate of the line whose angle is to be found.

ycoord is a real number giving the y-coordinate of the line whose angle is to be found.

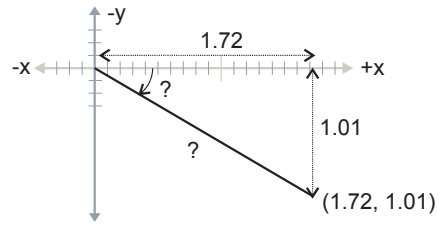
The angle returned is the angle that the specified line (whose second end is assumed to be at the origin), makes with the line from the origin to (0,1) - that is a line along the negative y-axis. Note that this is exactly 90° more than how all other angles are determined.

Sqrt()

If we started by knowing the end points of a line, could we work out the length of that line? Well, if we take a second look at what's going on when we calculate the value of a sine or cosine (see FIG-9.40), we can see how this calculation can be done.

FIG-9.40

End Points and Line Length



From the diagram, we can see that the end point coordinates actually represent the lengths of two sides of a right-angled triangle, so the length of the third side (the line we've drawn and the hypotenuse of the triangle) is given as:

$$\text{length of line} = \sqrt{\text{xcoord}^2 + \text{ycoord}^2}$$

Calculating the square of a value can be done with a line such as

```
xcoord * xcoord
```

or

```
xcoord^2
```

To calculate the square root, we might use

```
length = (xcoord^2 + ycoord^2)^ 0.5
```

However, AGK BASIC provides a `Sqrt()` function which performs the same operation as ^0.5. The statement's format is shown in FIG-9.41.

FIG-9.41

Sqrt()

where:

value is the real value whose square root is to be found. This cannot be a negative value.

So, the length of our line could be calculated as

```
length = Sqrt(xcoord^2 + ycoord^2)
```

Abs()

There are occasions when we want the value of a number without worrying about whether this is a positive or negative number.

In Chapter 4 we displayed the difference between a randomly generated number in the range 1 to 6 and the player's guess at what that number might be. This difference was calculated as:

```
diff = dice - guess
```

However, sometimes that difference would be a negative value when the guess was larger than the dice value. By using the `Abs()` function, which always returns the positive form of the argument, this problem can be eliminated:

```
diff = Abs(dice-guess)
```

FIG-9.42

Abs()

The format for the `Abs()` statement is given in FIG-9.42.

where:

value is a real value.

The absolute value of *value* will be returned by the statement.

Ceil()

Returns the next integer value greater than or equal to the argument. Hence,

```
Ceil(12.1)
```

returns 13 while

```
Ceil(15.0)
```

returns 15.

But remember, when using a negative argument as in

```
Ceil(-14.9)
```

the function returns -14 (which is greater than -14.9) and not -15 (which is less than -14.9).

FIG-9.43

Ceil()

The `Ceil()` function has the format shown in FIG-9.43.

where:

value is the real number whose value is raised to determine the return value.

Floor()

Complementing the `Ceil()` function is the `Floor()` function which returns the largest integer smaller than or equal to the function's parameter. This means that

```
Floor(12.1)
```

returns 12

```
Floor(15.0)
```

returns 15, and

```
Floor(-14.9)
```

returns -15.

FIG-9.44

Floor()

The `Floor()` function has the format shown in FIG-9.44.

where:

value is the real number whose value is lowered to determine the return value.

Trunc()

`Trunc()` is perhaps the simplest of the numeric functions since it returns the integral part of the real number argument, eliminating the fraction. So

```
Trunc(12.9)
```

returns 12 and

```
Trunc(-15.1)
```

returns -15.

FIG-9.45

Trunc()

The format for this statement is given in FIG-9.45.

where:

value is the real number whose value is to be truncated.

Round()

Whereas the `Trunc()` function returns an integer by deleting the fraction part of the parameter, `Round()` returns an integer by rounding the parameter to the nearest integer. Hence,

```
Round(15.1)
```

returns 15 and

```
Round(15.6)
```

returns 16.

Rounding up happens for fractions of over 0.5 for positive value (.0 to 0.5 rounds down).

Where the absolute value of a negative number's fraction is over 0.5, the value is rounded down:

```
Round(-64.6)
```
returns -65

FIG-9.46

Round()

The format for `Round()` is given in FIG-9.46.

where:

value is a real number whose value is to be rounded to the nearest integer.

Fmod()

Whereas the `mod` operator returns the integer remainder when two integer values are divided, the `Fmod()` function returns the complete remainder (integral and fraction) when two real numbers are divided. Hence,

```
Fmod(7.0, 5.0)
```

returns 2.0 since 5.0 divides into 7.0 once with a remainder of 2.

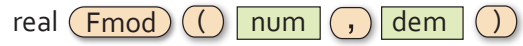
```
Fmod(16.9, 5.1)
```

returns 1.6 since 5.1 divides into 16.9 3 times with a remainder of 1.6.

FIG-9.47

Fmod()

The syntax for the `Fmod()` function is shown in FIG-9.47.



where:

num is a real value giving the numerator of the operation.

dem is a real value giving the denominator.

Activity 9.21

Give the value returned by each of the following function calls:

a) `Sqrt(64)` b) `Abs(-9)` c) `Ceil(-9.1)`
d) `Floor(14.0)` e) `Trunc(12.95)` f) `Round(-16.9)`
g) `Fmod(-12.6,3.2)`

Summary

- Cartesian coordinates use a horizontal x-axis and vertical y-axis to measure positions in a 2D space.
- These axes meet in the middle of that space at a point called the origin.
- All distances in the x and y directions are measured from the origin.
- By convention, points to the right of the origin on the x-axis are assigned positive values; points to the left, negative values.
- Points above the origin on the y-axis are assigned positive values; points below the origin, negative values.
- The axes divide 2D space into four quadrants.

- Any point in 2D space can be uniquely defined by specifying its position perpendicular to the x and y axes.
- A point's position in 2D space is known as the coordinates of the point and are given in the form

  (distance along the x-axis, distance along the y-axis)

  normally this description is shortened to

  (x,y)
- On a computer, the positive section of the y-axis points down.
- A computer screen represents only part of quadrant 1 in 2D space.
- The `Cos()` function returns the cosine of a specific angle given in degrees.
- The `Sin()` function returns the sine of a specific angle given in degrees.
- The `Cos()` and `Sin()` values for the same angle give the end coordinates of a line one unit in length whose other end is at the origin.
- For a line *a* units in length coming from the origin and at an angle of θ^o to the x-axis, the end coordinates are (a*cos(θ),a*sin(θ)).
- For a line of length a whose start point is at position (*m,n*) and lies at θ^o to the x-axis, the other end's coordinates are given as (a*cos(θ)+m, a*sin(θ)+n).
- The `Tan()` function returns the tangent of a specified angle given in degrees.
- Radians are an alternative way of measuring angles.
- One radian is the angle created when two radii of a circle are drawn in such a way that the distance along the arc of the circle's circumference from one radius to the other is exactly equal to the radius.
- The `CosRad()` function returns the cosine of a specific angle given in radians.
- The `SinRad()` function returns the sine of a specific angle given in radians.
- The `TanRad()` function returns the tangent of a specified angle given in radians.
- The `Acos()` function returns the angle of a line drawn from the origin with a specified end x-coordinate. The angle is given in degrees in the range 0^o to 180^o.
- The `Asin()` function returns the angle of a line drawn from the origin with a specified end y-coordinate. The angle is given in degrees in the range -90^o to $+90^o$.
- The `Atan()` function returns the angle of a line with a specified gradient. The angle will lie in the range -90^o to $+90^o$.
- The `AcosRad()` function returns the angle of a line drawn from the origin with a specified end x-coordinate. The angle is given in radians (0 to 2π)
- The `AsinRad()` function returns the angle of a line drawn from the origin with a specified end y-coordinate. The angle is given in radians (-π to π).
- The `AtanRad()` function returns the angle of a line with a specified gradient. The angle is given in radians (-π to π).

- The `ATanFull()` function returns the angle of a line (with specified end points) to the negative part of the y-axis. The result is in degrees (0^o to 360^o).
- The `ATanFullRad()` function returns the angle of a line (with specified end points) to the negative part of the y-axis. The result is in radians (0 to 2π).
- The `Sqrt()` function returns the square root of the function argument.
- The `Abs()` function returns the absolute value of the function argument.
- The `Trunc()` function returns an integer value calculated as the truncated value of the parameter.
- The `Round()` function returns an integer value calculated as the rounded value of the parameter.
- The `Fmod()` function returns the remainder produced by the division of two real values.

Solutions

Activity 9.1

The program should display the random string that was generated and its length.

Activity 9.2

b$ would be set to 1-BY-1.

Activity 9.3

The original version of *Letters*:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
Print("Original string is: "+text$)
for c = 1 to Len(text$)
  Print(Mid(text$,c,1))
next c
Sync()
do
loop
```

This will begin by displaying the original string, then each letter of that string on separate lines.

To display the letters in reverse order, the `for` loop needs to decrement from `Len(text$)` down to 1. So the new code is:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
for c = Len(text$) to 1 step -1
  Print(Mid(text$,c,1))
next c
Sync()
do
loop
```

The final version counts the number of E's in the string:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
count = 0
for c = Len(text$) to 1 step -1
    if Mid(text$,c,1) = "E"
        inc count
    endif
next c
Print("Original string is: "+text$)
PrintC("It contains ")
PrintC(count)
Print(" E's")
Sync()
do
loop
```

Activity 9.4

The code for *ASCIITable*:

```
for c = 32 to 126
    rem *** Display number ***
    PrintC(c)
    PrintC(" is the ASCII code for ")
    rem *** Display character ***
    Print(Chr(c))
    rem *** If 25th update screen and wait 5 secs
    ↳***
    if (c-31) mod 25 = 0
        Sync()
        Sleep(5000)
    endif
next c
Sync()
do
loop
```

Activity 9.5

Code for *CountZero*:

```
rem *** Generate random number ***
num = Random(1000,65000)
rem *** Convert to a string ***
num$= str(num)
rem *** Start count at zero ***
count = 0
rem *** Check each character ***
for c = 1 to Len(num$)
    rem *** If it's a 0, increment count ***
    if Mid(num$,c,1) = "0"
        inc count
    endif
next c
rem *** Display details ***
Print("Original number "+num$)
Print("Contains "+str(count)+" zeros")
Sync()
do
loop
```

Activity 9.6

Create a new project called *Conversions*.
Compile the default code.
Copy the file *Buttons.png* and *Buttons subimages.txt* to the *media* folder (you'll find a copy in *TestButtons*).
Copy *Buttons.agc* into the the *Conversions* folder.

In the *setup.agc* file, change height to 1024 and width to 768.

Code *main.agc* as:

```
#include "Buttons.agc"

SetUpButtons()
rem *** Get value from buttons ***
num = GetButtonEntry()
rem *** Display number in dec, binary and hex ***
Print("Number (base 10  : "+Str(num))
Print("Number (base 2)  : "+Bin(num))
Print("Number (base 16) : "+Hex(num))
Sync()
do
loop
```

Activity 9.7

No solution required.

Activity 9.8

Create a new project called *FunctionTester*.
Copy *StringLibrary.agc* into the *FunctionTester* folder.

The code for *FunctionTester* is:

```
rem *** Test Pos Function ***
#include "StringLibrary.agc"
text$ = RandomString(30)
post = Pos(text$,"D")
Print("String is "+text$)
Print("D at position "+Str(post))
Sync()
do
loop


rem *** Find Position of character in string ***
function Pos(s$, f$)
  rem *** result stays at 0 if no match found ***
  result = 0
  rem *** Make sure we're looking for a single
  ↳character ***
  first$ = Mid(f$,0,1)
  rem *** FOR each character in s$ DO ***
  for c = 1 to Len(s$)
    rem *** IF that character matches what we're
    ↳after THEN ***
    if Mid(s$,c,1) = first$
      rem *** Set result to this position and exit
      ↳loop ***
```

```
        result = c
        exit
      endif
    next c
endfunction result
```

Changing the character searched for to a lowercase letter will guarantee that the letter is not found. The program will display zero for the position found. A better option would be to check for zero being returned with code such as:

```
#include "StringLibrary.agc"
text$ = RandomString(30)
post = Pos(text$,"D")
if post <> 0
    Print("String is "+text$)
    Print("D at position "+Str(post))
else
    Print("D not found in text")
endif
Sync()
do
loop
```

Activity 9.9

In the Projects Panel, right-click the project name and select **Add files** from the pop-up menu.
Select *StringLibray.agc* from the listed files. That file will then be listed in the *Sources* part of the project.
Double-click on the *StringLibrary.agc* file in the Projects Panel. This will open a tab for the file's source code in the edit area.
Copy the code for function *Pos()* from *main.agc* and paste it into *StringLibrary.agc* after the existing function.

Select **Files|Save everything**

Activity 9.10

Updated code for *FunctionTester*:

```
rem *** Test Pos Function ***
#include "StringLibrary.agc"

text$ = RandomString(30)
count = Occurs(text$,"S")
Print("String is "+text$)
Print("S occurs "+Str(count)+" times")
Sync()
do
loop



rem *** Return how often f$ occurs in s$ ***
function Occurs(s$,f$)
  rem *** None found so far ***
  result = 0
  rem *** Make sure only one character ***
  first$ = Mid(f$,0,1)
  rem *** FOR each character in s$ Do ***
  for c = 1 to Len(s$)
    rem *** if it matches req'd character, add 1 to
    ↳result ***
    if Mid(s$,c,1) = first$
      result = result + 1
    endif
   next c
endfunction result



rem *** Find Position of character in string ***
function Pos(s$, f$)
  rem *** result stays at 0 if no match found ***
  result = 0
  rem *** Make sure we're looking for a single
character ***
  first$ = Mid(f$,1,1)
  rem *** FOR each character in s$ DO ***
  for c = 1 to Len(s$)
    rem *** IF that character matches what we're
after THEN ***
    if Mid(s$,c,1) = first$
      rem *** Set result to this position and exit
loop ***
      result = c
      exit
    endif
  next c
endfunction result

rem *** Return how often f$ occurs in s$ ***
function Occurs(s$,f$)
  rem *** None found so far ***
  result = 0
  rem *** Make sure only one character ***
  first$ = Mid(f$,0,1)
  rem *** FOR each character in s$ Do ***
  for c = 1 to Len(s$)
    rem *** if it matches req'd character, add 1 to
    ↳result ***
    if Mid(s$,c,1) = first$
      result = result + 1
    endif
   next c
endfunction result
```

The code for *Occurs()* should be copied from *main.agc* and pasted into *StringLibrary.agc*.

Activity 9.11

Code for *FunctionTester* (previous functions are not shown):

```
text$ = "ABCDEFGHI"
text$ = Insert(text$,"XX",2)
Print("String is "+text$)
Sync()
do
loop

rem *** Inserts f$ at position p in s$ ***
function Insert(s$,f$,p)
  rem *** If invalid position, result is original
  ↳string ***
  if p < 1 or p > Len(s$)+1
    result$ = s$
  else
    rem *** split s$ into two parts & insert f$
    ↳in between ***
    result$ = Left(s$,p-1)
    result$ = result$ + f$
    result$ = result$+ Right(s$,Len(s$)-(p-1))
  endif
endfunction result$
```

Changing the line

```
text$ = Insert(text$,"XX",2)
```

to

```
text$ = Insert(text$,"XX",12)
```

will return the original text since the insert position given is invalid.

Copy and paste the code for the routine into *StringLibrary.agc*.

Activity 9.12

Code for *FunctionTester* (previous functions are not shown):

```
text$ = "ABCDEFGHI"
text$ = Delete(text$,3,5)
Print("String is "+text$)
Sync()
do
loop


rem *** Delete num characters from s$ starting at
position st ***
function Delete(s$, st, num)
  rem *** if invalid position, result is original
  ↳string ***
  if st < 1 or st > Len(s$)
    result$ = s$
  else
```

```
        rem *** Set result to the part of s$ to the
        ↳left of ***
        rem *** the section to be deleted ***
        result$ = Left(s$, st-1)
        rem *** IF not deleting to the end of s$, ***
        rem *** add right section ***
            if st+num-1 <= Len(s$)
                result$ = result$+Right(s$,Len(s$)-
                ↳(st+num-1))
            endif
    endif
endfunction result$
```

Changing

```
text$ = Delete(text$,3,5)
```

to

```
text$ = Delete(text$,13,5)
```

will return the original string since the delete position is invalid.

When the number of characters to be deleted is greater than the number available, all characters after the start position are deleted. Hence,

```
text$ = Delete(text$,3,15)
```

returns

```
AB
```

Copy the functions code and paste it into *StringLibrary.agc*.

Activity 9.13

FUNCTION NAME	:	Replace
PARAMETERS		
In	:	s : string sr : character p : integer
Out	:	result : string
PRE-CONDITION	:	1 <= p <= Len(s)
DESCRIPTION	:	*result* is equal to *s* except that the pth character of *result* is *sr*.

Code for *FunctionTester* (previous functions are not shown):

```
text$ = "ABCDEFGHI"
text$ = Replace(text$,"X",3)
Print("String is "+text$)
Sync()
do
loop


rem *** Replace the pth character in s$ with rs$ ***
function Replace(s$, rs$, p)
    rem *** If invalid position ***
    rem *** return original string ***
    if p < 1 or p > Len(s$)
        exitfunction s$
    endif
    rem *** If rs$ more than one character use left-
    ↳most character ***
    rs$ = Left(rs$,1)
    rem *** Calculate result ***
    result$ = Left(s$,p-1)+rs$+Right(s$,Len(s$)-p)
endfunction result$
```

A line such as

```
text$ = Replace(text$,"X",13)
```

will return the original string since the position specified is invalid.

Copy and paste the code into *StringLibrary.agc*.

Activity 9.14

At 0^o the x-coord is 1
At 90^o the x-coord is 0

Activity 9.15

cos(0) = x-coord = 1
cos(90) = x-coord = 0
cos(30) = x-coord = 0.866
cos(70) = x-coord = 0.342

Activity 9.16

cos(50) = x-coord = 0.643
cos(168) = x-coord = -0.978
cos(213) = x-coord = -0.839
cos(304) = x-coord = 0.559

Activity 9.17

sin(50) = y-coord = 0.766
sin(168) = y-coord = 0.208
sin(213) = y-coord = -0.545
sin(304) = y-coord = -0.829

Activity 9.18

x coord = 3.7cos(191.5) = -3.626
y coord = 3.7sin(191.5) = -0.738

Activity 9.19

x coord = 5cos(60)+9 = 11.5
y coord = 5sin(60)+8= 12.330

Activity 9.20

No solution required.

Activity 9.21

a) 8 b) 9 c) -9
d) 14 e) 12 f) -17
g) -3.0

10

Arrays

In this Chapter:

- ❑ The Limitations of Standard Variables
- ❑ The Concept of Arrays
- ❑ Declaring Arrays
- ❑ Initialising Arrays
- ❑ Accessing Array Elements
- ❑ Array Subscripting
- ❑ Arrays and Counting
- ❑ Arrays and Non-Repeating Values
- ❑ Arrays and Shuffling
- ❑ Arrays and Sorting
- ❑ Arrays and Searching
- ❑ Multi-dimensional Arrays
- ❑ Arrays as Function Parameters

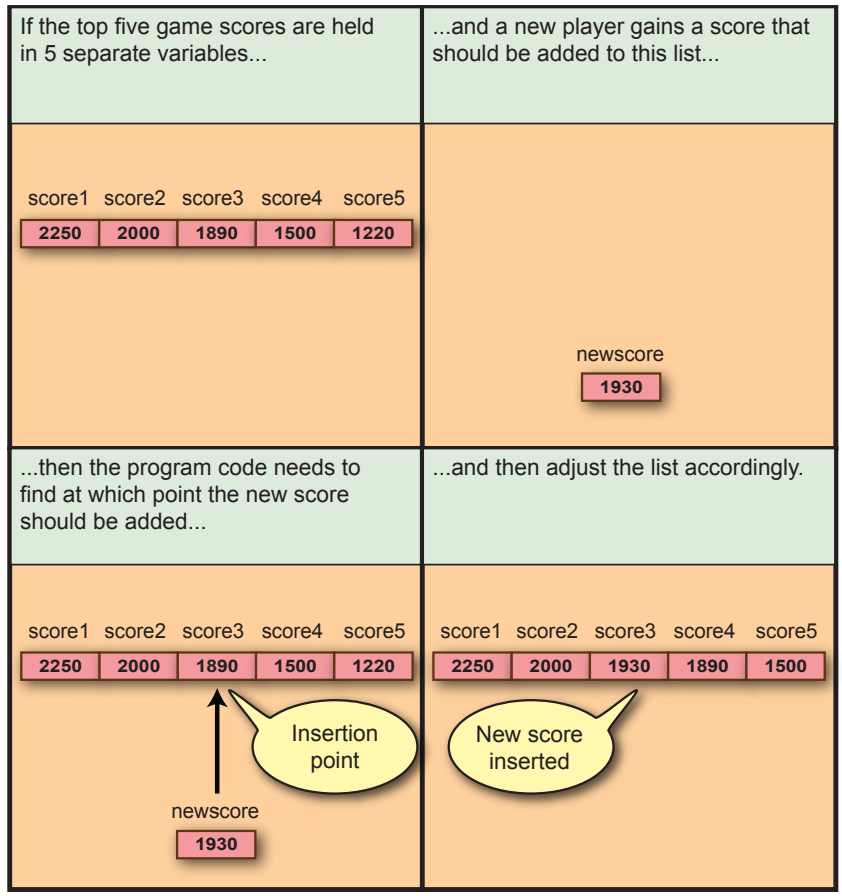
Problems with Simple Variables

There are certain tasks which are very difficult or long-winded when we try to do them using the normal variables we've been dealing with up to now. For example, it's common for a video game to retain the top five scores but, from what we know at the moment, we'd have to set up one variable for each score to be saved.

When a player finishes a game, the program then has to decide if the player's score should be recorded in the top five and, if so, at what position. If the new score is good enough to be recorded as a highest score, then the list must be updated. The whole process is shown in FIG-10.1.

FIG-10.1

Using Regular Variables



Notice that what had been the third and fourth highest scores, have now moved down one position and that the score of 1220 has been lost from the top five.

We need to develop an algorithm which can perform the above task for all possible values which might be placed within the top five scores. One possible structured English solution could use the lines:

```
IF
   newscore > score1:
      score5 = score4
      score4 = score3
```

```
            score3 = score2
            score2 = score1
            score1 = newscore
         newscore > score2:
            score5 = score4
            score4 = score3
            score3 = score2
            score2 = newscore
         newscore > score3:
            score5 = score4
            score4 = score3
            score3 = newscore
         newscore > score4:
            score5 = score4
            score4 = newscore
         newscore > score5:
            score5 = newscore
   ENDIF
```

Activity 10.1

Assuming the following values

```
score1 = 2250 score2 = 2000 score3 = 1890 score4 = 1500
score5 = 1220

newscore = 1900
```

work your way through the algorithm given above to check that the expected result is obtained.

The algorithm is a bit long-winded, but just about acceptable. Now imagine that we had the top ten scores to retain. What would the algorithm look like then? It's going to be long - very long. Luckily, there is a better way to achieve what we're after - **arrays**.

One Dimensional Arrays

Array Concepts

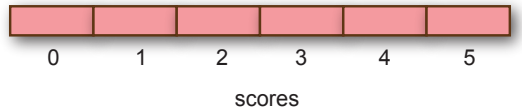
An **array** is a named data variable capable of storing several values at the same time. It is a collection of **elements** or **cells**. Each of these elements holds a single value - just like the regular variables we have used in our previous programs.

Each cell within an array is numbered. The first cell is numbered cell zero, the next cell 1, etc. Exactly how many cells an array contains is determined when the array is first set up.

FIG-10.2 shows how we might visualise a 6 element array called *scores*.

FIG-10.2

Visualising Arrays



The individual cells within the array are identified by a combination of the array name and the cell's number which is known as the **subscript**. The subscript is enclosed within square brackets. For example, the second cell within the array shown

above is identified using the term

```
scores[1]
```

Declaring an Array

Whereas we are free to introduce a standard variable at any point in a program, we need to tell the compiler in advance that we intend to use an array. This is known as an **array declaration**. An array declaration begins with the keyword `dim` followed by the name we wish to assign to the array. The only additional piece of information required is the subscript for the final element of the array - this determines how many elements the array is to contain. So, to set up the *scores* array as shown in FIG-10.2, we would use the declaration

```
dim scores[5]
```

This creates a 6 cell array with the cells numbered 0 to 5.

Arrays can be declared to hold values of any of the types we can use for regular variables: integer, real or string. For example,

```
dim averages#[10] rem *** 11 element real array ***
dim names$[19]    rem *** 20 element string array ***
```

Every cell within the array can then hold a single value of the specified type. It is not possible to create an array with cells of differing types.

Initialising Arrays

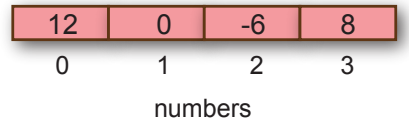
When an array is first set up, every cell in the array contains the value zero (or an empty string when using string arrays). But it is possible to specify a different starting value for each cell when declaring the array by extending the array declaration. For example, the line

```
dim numbers[3] = [12, 0, -6, 8]
```

will create the array setup shown in FIG-10.3.

FIG-10.3

Array Initialisation



If there are too many values specified within the braces, the surplus values are ignored; if there are too few values supplied, then the cells which have not specifically been assigned a value are set to zero.

Accessing Array Elements

We cannot perform operations on an array as if it were a single entity. For example, it would be invalid to try to display all the values held in an array with a statement such as

```
Print(numbers)
```

Instead, we must deal with the individual elements within the array. So, to display the value in the first element in the array *numbers*, we would write

```
Print(numbers[0])
```

To assign a value to the next element we could use a statement such as

```
numbers[1] = 4
```

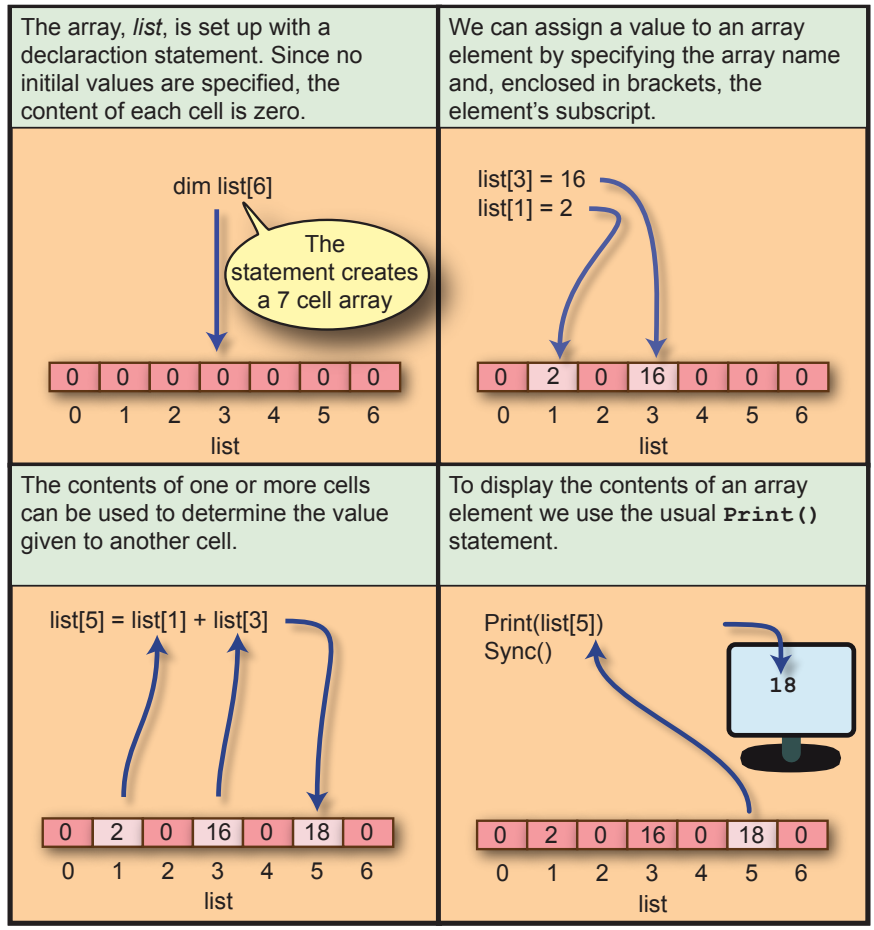
and we could check if the last element contained a value of less than zero with the line

```
if (numbers[3] < 0)
```

In fact, we can use an array element in any statement where we might use a simple variable of the same type. Some more examples are shown in FIG-10.4.

FIG-10.4

Accessing Array Elements



You must ensure that the subscript you supply is a valid one; the compiler will not check that the subscript value is within a range compatible with the size of the array created. Hence, code such as

```
dim list[6]
list[7]= 21;   rem *** Subscript too high ***
```

will be compiled but when the program is running it will halt when the assignment statement is reached and display a message of the form

```
Subscript out of bounds at line 2
```

What Makes Arrays Powerful

If what we've seen up to now was all that could be achieved by arrays, they would be of little more use than simple variables. For example, if we were to simulate the throwing of four dice using four integer variables, we could use the lines:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
dice3 = Random(1,6)
dice4 = Random(1,6)
```

Using an array would require the lines:

```
dim dice[4]
dice[1] = Random(1,6)
dice[2] = Random(1,6)
dice[3] = Random(1,6)
dice[4] = Random(1,6)
```

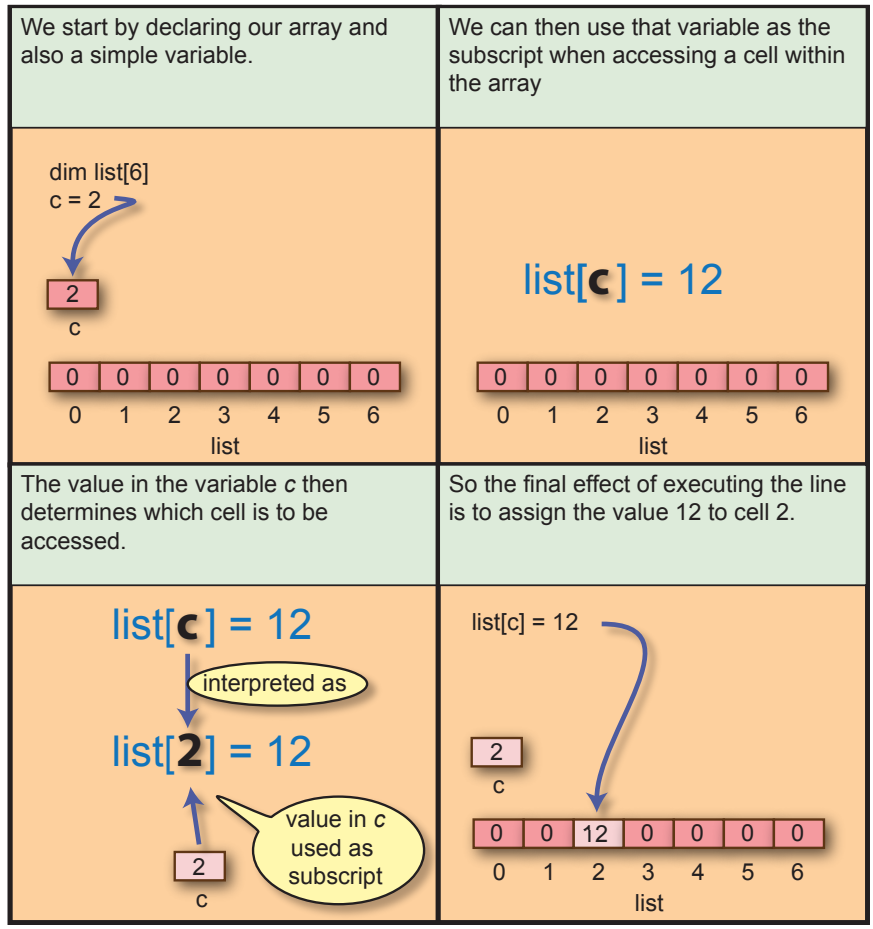
➽ `dice[0]` is unused.

Both segments are equally long winded.

What adds power to the array is the fact that the subscript need not be given as a fixed value. Instead, we are free to use a variable. The value of that variable then determines the value of the subscript and hence which element within the array is to be accessed (see FIG-10.5).

FIG-10.5

Variable Subscripts



If the contents of the variable *c* are changed, then it follows that the array element being accessed will also change.

We need to take only one further step to realise how we can use arrays to create shorter code for our earlier problem of reading in four values.

With the code

```
for c = 0 to 6
```

the variable *c* will, as the loop repeats itself, take on first the value 0, then 1, 2, etc. and finally 6.

So, returning to our dice code, if we use a variable it conjunction with array access, we can assign our four values using the code

```
dim dice[4]
for c = 1 to 4
   dice[c] = Random(1,6)
next c
```

As *c* changes value each time the loop iterates, so the term `dice[c]` in the assignment statement will reference a different element of the array *dice*.

Activity 10.2

Start a new project called *Arrays01*, and, using the code given above, create a complete program which stores the values obtained by four dice throws in the array *dice* (ignore element zero).

By adding a second `for` loop to your code, get the program to display the contents of the array. Test and save your project.

Array Element Zero

Every array always has an element zero - as we have already seen. But there are times when the clarity of an algorithm is better served by ignoring this element. For example, we used elements *dice[1]* to *dice[4]* to store our dice throws, ignoring *dice[0]*. If we want to store information based on the months of the year, we would probably set up the appropriate array

```
dim months[12]
```

and use *months[1]* to *months[12]* since this corresponds to the months of the year.

Of course, there is no reason why we could not use elements *dice[0]* to *dice[3]* and *months[0]* to *months[11]* for our data, but doing that detracts slightly from how we might normally think. And that means we are more likely to make mistakes in our program logic and hence, in these cases, using element zero is probably best avoided.

The only downside of ignoring element zero is that we end up making our arrays one element larger than they need to be. This seems like a small price to pay given the memory available on modern devices.

Array Subscript Options

We've already seen that an array subscript can be given in the form of a constant as

in `dice[1]` or as a variable (`dice[c]`), but it can also be given in the form of an arithmetic expression. For example, in the code

```
dim values[20]
p = 3
values[p*2] = 42
```

will store the value 42 in *values[6]* - which is the seventh cell within the array.

We can even use the contents of one cell as the subscript. So, in the code

```
dim values[20]
values[0] = 9
values[values[0]] = 4
```

will result in the value 4 being stored in *values[9]*.

Activity 10.3

State the contents of each cell in the array *numbers* after the following code has been executed.

```
dim numbers[8]
for p = 0 to 8
      numbers[p] = p*2
next p
numbers[numbers[2]-1] = 23
```

Using Arrays

We've already seen a simple example of how we might make use of an array, but arrays can be used in many more ways. Some examples of how arrays can be used to help create an efficient solution to a problem are shown in this section.

Problem: Multiple Counts

One of the tests used to make sure that a dice is not bias is to check that, for a large number of throws, each number should appear approximately the same number of times.

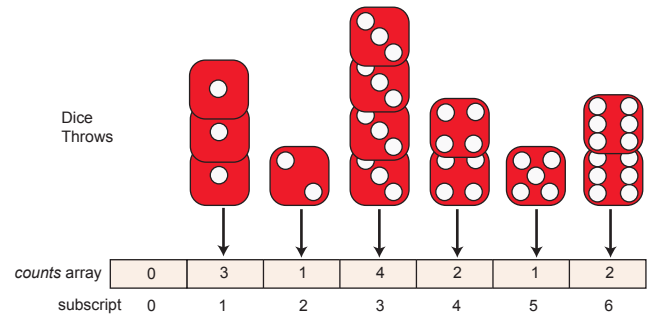
A dice is said to be bias if each number does not have the same likelihood of being thrown.

Solution:

We can use an array to keep count of how often each number occurs. Cell 1 will hold a count of how often the number 1 is thrown, cell 2 the number of times 2 is thrown, etc (see FIG-10.6).

FIG-10.6

Counts Concept



Since the diagram shows only a small number of throws, the distribution of each number can vary more widely.

The structured English for our solution could be written as

```
FOR 1000 times DO
    Throw dice
    Add 1 to appropriate count
ENDFOR
Display all 6 counts
```

The code for the program is given in FIG-10.7.

FIG-10.7

Keeping Multiple Counts

```
dim counts[6]
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
   rem *** Throw dice ***
   dicethrow = Random(1,6)
   rem *** Add to appropriate count ***
   inc counts[dicethrow]
next c

rem *** Display each count ***
for c = 1 to 6
   Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

Activity 10.4

Start a new project *DiceCount*.

Modify *main.agc* to match the code given in FIG-10.7. Test and save your project.

Some games make use of a 10 sided dice. Modify your program so that it will generate numbers in the range 1 to 10 and count how often each value occurs.

As you see, modifying the code for a 10-sided dice requires changes in several lines. To avoid this we could set the number of sides as a named constant.

Modify your code to use a named constant called SIDES for the number of sides on the dice.

Now change the code to deal with a 20 sided dice. How many lines of code need to be modified to handle this?

Problem: Generating Random Non-Repeating Values

Many countries run lottery systems. The simplest of these require you to choose 6 unique numbers in the range 1 to 49. Of course, we can easily get the computer to generate and display six numbers in this range, but we also need to make sure that none of the six numbers are the same.

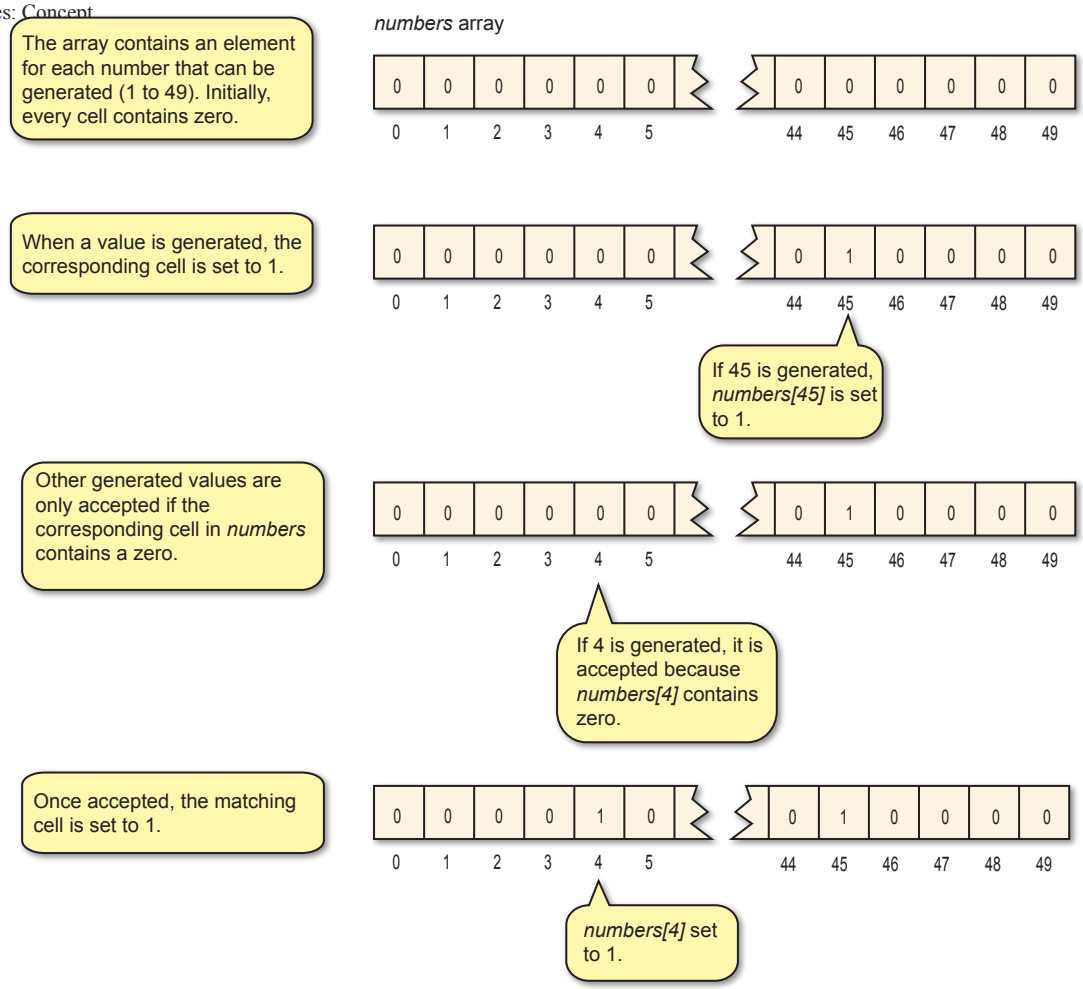
Solution:

To ensure that there are no duplicate values from the second number onwards, we

need to check that the generated number has not already been selected. One way to do this is to set up an array containing a cell for each number that might be generated. Initially all the cells contain zero, but when a number is selected the corresponding cell's value is set to 1. When a number is generated, it can only be added to the list of selected values if its corresponding cell contains a zero at that point (see FIG-10.8).

FIG-10.8

Unrepeated Random Values: Concept



The structured English for our solution would be:

```
Set all cells to 0
Generate a random number in the range 1 to 49
Set the corresponding cell to 1
Display the value
FOR 5 times DO
    REPEAT
        Generate a random number
    UNTIL the corresponding cell is zero
    Set the corresponding cell to 1
    Display the number
ENDFOR
```

The program for this is shown in FIG-10.9.

FIG-10.9

Unrepeated Random Values: Code

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem ***Generate number ***
number = Random(1,HIGHEST)

rem *** Set corresponding cell ***
lottery[number] = 1

rem *** Display value ***
Print(number)
rem *** FOR 5 times DO ***
for c = 1 to 5
   rem *** Generate an unselected number ***
   repeat
       rem ***Generate number ***
      number = Random(1,HIGHEST)
   until lottery[number] = 0
   rem *** Set corresponding cell ***
   lottery[number] = 1
   rem *** Display value ***
   Print(number)
next c
Sync()
do
loop
```

Activity 10.5

Start a new project, *Lottery*. Modify *main.agc* to match the code in FIG-10.9 and test the program.

There is really no need to treat the first number any differently from the remaining five. Modify the code so that all 6 numbers are generated within the `for` loop.

The code displays the numbers as they are generated rather than in ascending order. Modify the code so that the six numbers are displayed in ascending order.

(HINT: You will need to remove the existing `Print()` statements from the code.)

Problem: Shuffling

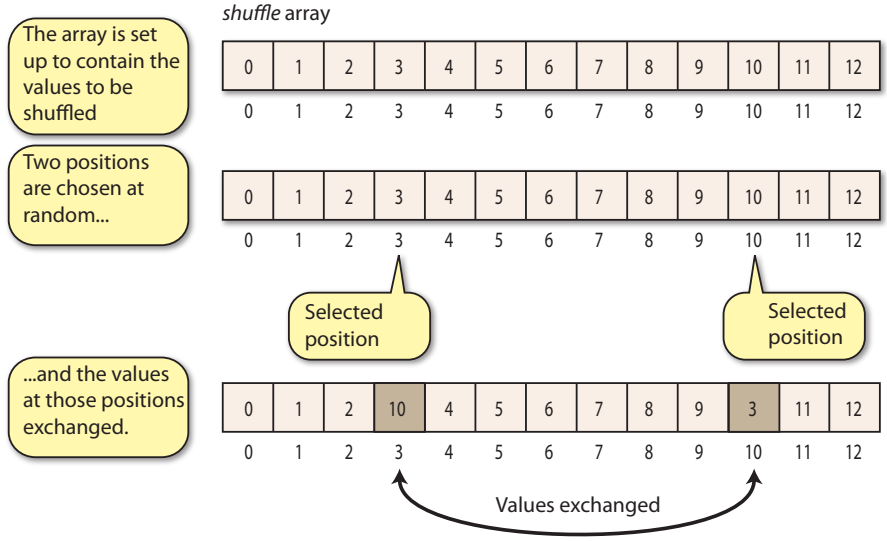
Many applications require items to be re-arranged in a random order. For example, your MP3 player probably offers a shuffle option which will play music tracks in a random order and shuffling is mandatory for almost every card game.

Solution:

If we start by storing a set of values in an array (these may represent music track numbers or playing card values), then we can create a shuffle effect by taking the values at two randomly selected positions within the array and swapping them over (see FIG-10.10).

FIG-10.10

Shuffling: Concept



If we continue to do this many more times, the items will have been effectively shuffled.

The structured English is:

```
Set up all values within an array
FOR 200 times DO
    Generate a first random subscript
    Generate a second random subscript
    Swap the values held at the subscript positions
ENDFOR
```

The code for shuffling an array of 20 values is given in FIG-10.11.

FIG-10.11

Shuffling: Code

```
dim list[20]

rem *** Set up values in array ***
for c = 1 to 20
   list[c] = c
next c
rem *** Shuffle ***
for c = 1 to 200
   rem *** Generate two subscript values ***
   sub1 = Random(1,20)
   sub2 = Random(1,20)
   rem *** Swap values at these positions ***
   temp = list[sub1]
   list[sub1] = list[sub2]
   list[sub2] = temp
next c
rem *** Display shuffled items ***
for c = 1 to 20
   PrintC(Str(list[c])+"  ")
next c
Sync()
do
loop
```

Activity 10.6

Start a new project, *Shuffle*, and implement the code in FIG-10.11. After testing, modify the program so that the contents of *list* are displayed before and after the shuffle.

To simulate a card pack, we would need a 52 element list. The numbers 0 to 12 could represent ace to king of hearts; 13 to 25 diamonds; 26 to 38 spades; and 39 to 51 clubs. Modify the program to shuffle a deck of cards and display the first six "cards" in the list.

It would be better if we could display the value and suit of a card rather than just a number. For example, displaying *2 of diamonds* rather than *14*. Modify your program to do this. For the moment, Ace, Jack, Queen and King can be displayed as 1, 11, 12, and 13 of the appropriate suit. (HINT: Use the division and modulo operators (`/` and `mod`) to determine the suit and value of a card.

A final improvement would be to display the names Ace, Jack, Queen and King as appropriate. Test and save your project.

Problem: Handling an Array that is not Full

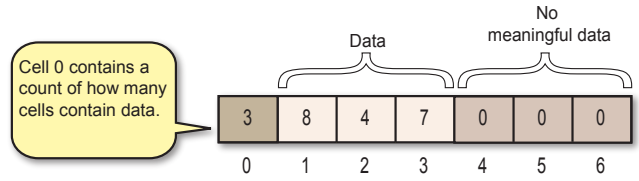
There are times when we will set up an array with space enough to hold a specific number of values, but initially not all the cells will contain meaningful data. For example, a game which remembers the top 5 scores, will contain no top scores when first played. In these situations, we may want to access only the elements of the array in which data has already been placed and so we need to know which cells contain data.

Solution:

One way to handle this problem is to use element zero in the array to keep a count of how many cells in the array contain data (see FIG-10.12).

FIG-10.12

A Data Count



The main steps involved in this setup are shown in FIG-10.13.

FIG-10.13

Using a Data Count

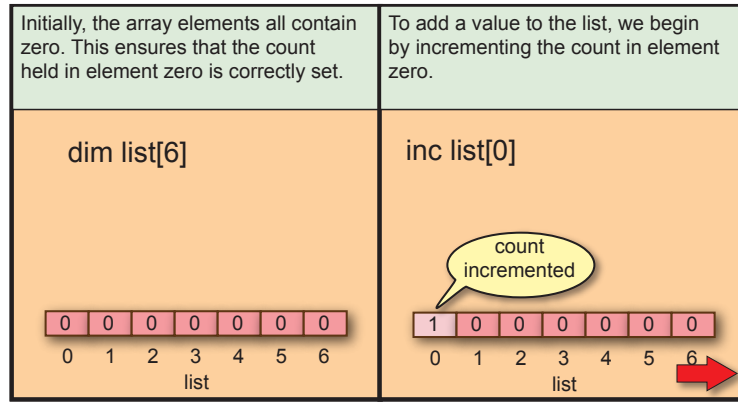
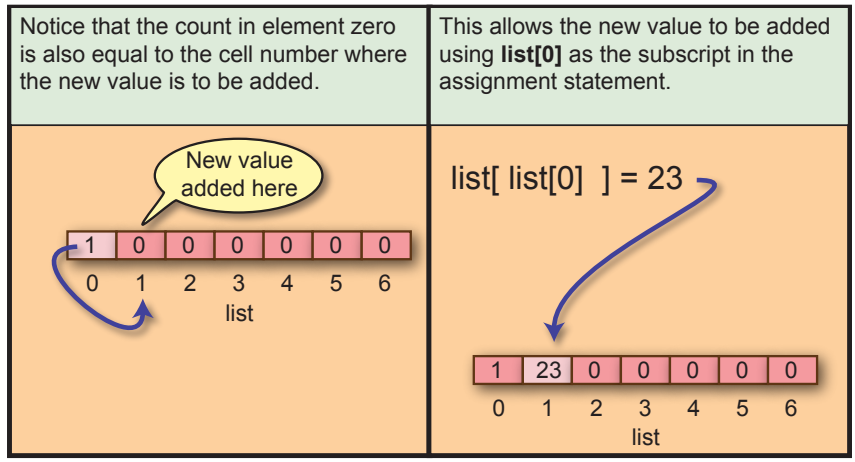


FIG-10.13
(continued)

Using a Data Count



The only check that is required when adding a new value is that the array must not already be full.

The logic required to insert a value into the list can be written in structured English as:

```
Get value to be added
IF the list is not full THEN
    Add 1 to the count in element zero
    Insert the new value at the end of the existing data
ELSE
    Display message "List is full"
ENDIF
```

A menu-driven program demonstrating how values are added to an array using the technique described above is given in FIG-10.14. Note that the program allows four options: add a value, display how many values are held, display the values held, and quit.

FIG-10.14

Implementing a Data Count

```
#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE]

//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-4)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
```

FIG-10.14
(continued)

Implementing a Data Count

```
    //*** Execute option ***
    select option
      case 1:  //*** Add a new value to the list ***
            Print("Enter value to be added : ")
            Sync()
            Sleep(2000)
            value = GetButtonEntry()
         if  list[0] < SIZE
               inc list[0]
            list[list[0]] = value
         else
            Print("List is full")
         endif
         Sync()
         Sleep(2000)
        endcase
        case 2:   //*** Display the number of items in the list ***
            Print("The list contains "+Str(list[0])+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3:   //*** Display the contents of the list ***
            if (list[0] = 0)
                Print("The list is empty")
            else
                Print("Values held are")
                for c = 1 to list[0]
                    PrintC(Str(list[c])+" ")
                next c
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 4
Sleep(2000)
end
```

Activity 10.7

Start a new project called *DataCount* and implement the code given in FIG-10.14.

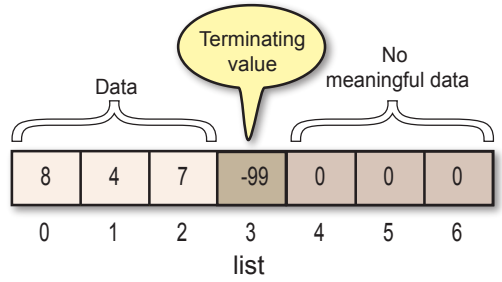
Remember to copy the three files needed to use the *Buttons* functions.

Test and save your project.

Keeping a count of the number of entries in an array is only one way of handling the problem of keeping tabs on just how many elements within an array contain meaningful data. A second approach is to use a "marker" value in the cell following the last value held in the array. For example, we might follow the actual data by a value of, say, -99 (see FIG-10.15).

FIG-10.15

A Sentinel Value

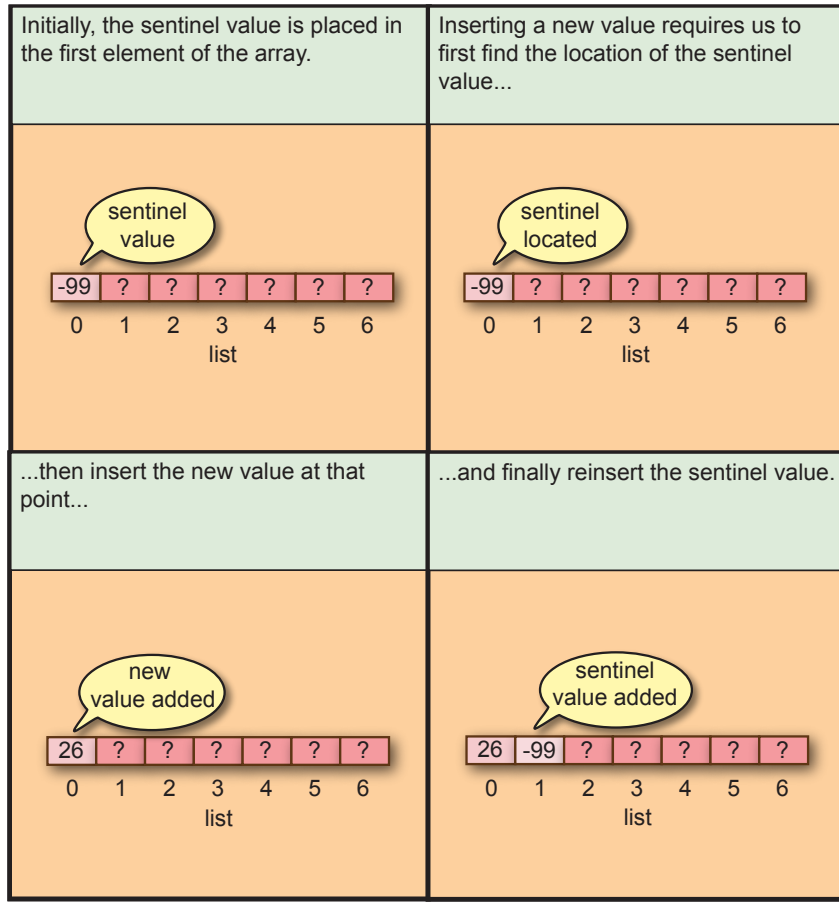


The marker value, often known as the **sentinel**, must be chosen with care. Obviously, we cannot use a value which can occur within the actual data, since such an occurrence would be assumed to be the terminating value. For example, we could safely choose the value -1 as a terminating value if we were sure that all the actual data values were positive.

The main characteristics of this approach are shown in FIG-10.16.

FIG-10.16

Using a Sentinel Value



The complex part of this operation is locating the sentinel value within the array. After several values have been added, there is no easy way of knowing the sentinel's position. To find its location, we must search through the contents of the array.

Searching a list of values for some specific entry is one of the commonest requirements in a software application. There are many ways of searching a list. For the moment,

we will content ourselves by examining only one of these.

If we are looking for the value -99 in a list of values, we can compare -99 with each value in the list and stop when we find a match. This can be achieved by the code:

```
post = 0
while list[post] <> -99
   inc post
endwhile
```

Once the insert position has been found, we need to insert the new value and place -99 to its new position. Assuming we are using the version of the `while` loop given above, this would be achieved using the lines

```
list[post] = value
list[post+1] = -99;
```

We can determine if the list is empty using the expression

```
if list[0] = -99
```

and if it is full using

```
list[SIZE] = -99
```

To count the number of entries in the list, we are forced to search for the sentinel. Its position in the list will be equal to the number of entries. For example, initially -99 is in cell 0 and there are zero entries in the list; when -99 is stored in cell 3 there will be three entries in the list (occupying cells 0, 1 and 2).

Activity 10.8

Using the program you created in Activity 10.7 as a guide, create a new project called *SentinelData* which makes use of a sentinel-based list.

The program should retain the same four options: allowing a value to be added to the data, displaying the number of values already stored, displaying the actual contents of the array, and a quit option.

Test and save your project.

Problem: Inserting a Value into an Array

In the previous problem, new values were inserted at the end of the existing data. However, there are many circumstances when the new value will be required to be positioned elsewhere within that data. As we will see, inserting a new value within existing data causes new problems.

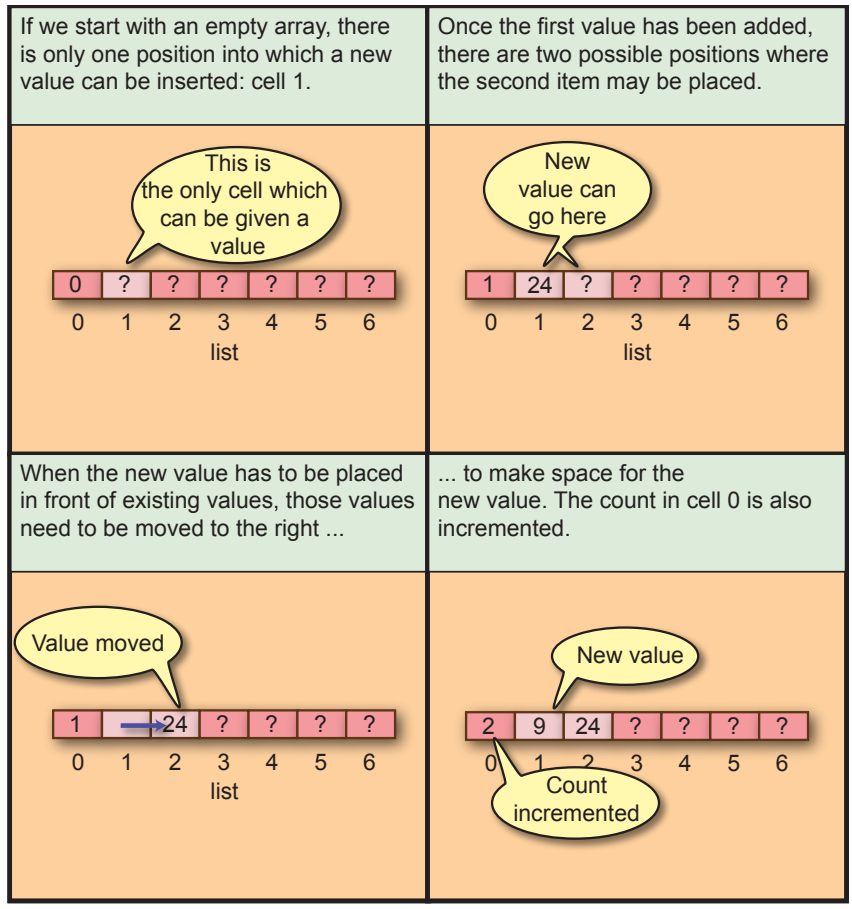
Solution:

When we want to add a value between existing values (just as we might add a character into a misspelled word when using a word processor) then we need to create space at the insertion point by moving other values out of the way.

FIG-10.17 shows the steps involved. for a count-based array.

FIG-10.17

Adding a Value to an Array



The value 24 is actually copied into *list[2]*, so *list[1]* is not empty (as suggested by the diagram), but still contains the value 24. This will be overwritten when the new value is inserted.

Activity 10.9

Assuming an array is in the state shown in the diagram below

in which cells may a new value be positioned?

Assuming the value 77 is to be placed in cell 3, show, with the aid of diagrams, the state of the array after:

a) Existing values have been moved to make space for the new value.

b) The new value has been inserted.

When a new item is being added we need to acquire not only its value, but also the cell number into which it is to be placed. This can be done using the code:

```
Print("Enter new value")
Sync()
Sleep(1000)
value = GetButtonEntry()
Print("Enter its position")
Sync()
Sleep(1000)
post = GetButtonEntry()
while (post < 1 or post > list[0]+1)
   Print("Invalid position. Must be in the range 1 to "
   ↳+Str(list[post]+1))
   Print("Re-enter position")
   Sync()
   Sleep(1500)
   post = GetButtonEntry()
endwhile
```

Notice that the code includes a check to insure that the insert position is valid.

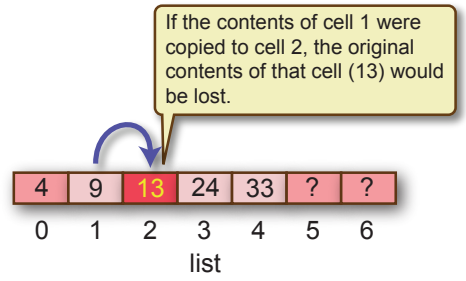
To free space for the new value, we need to move all those values between *post* and *list[0]* up one position within the array. This is done using the following code:

```
for current = list[0] to post step -1
   new = current + 1
   list[new] = list[current]
next current
```

As you can see, it is necessary to move the value at the end of the list first, otherwise you would overwrite the next value (see FIG-10.18).

FIG-10.18

Moving Data



All that remains now is to add the new value and increment the count held in *list[0]*:

```
list[post] = value
inc list[0]
```

Activity 10.10

Modify your *DataCount* project so that, when new data is entered, the program requests an insert position for the data and places the new data at the specified position.

Test and save your project.

Problem: Recording the Top Scores

We started this chapter by looking at what was involved in maintaining a record of the top five scores in a video game. We have now learnt enough about arrays and the techniques employed when using them to tackle that earlier problem. But there are a

couple of new problems to handle:

- When a top score is achieved, the point at which it is inserted in a list is determined by the existing values in that list.
- Once five scores have been recorded within the list, any new score that is added will mean that the lowest score will be eliminated from the list.

Solution:

To insert a new top score, we need to search down the list to find the first entry with a value which is less than the one we wish to add.

```
post = 1
while list[post] >= newscore
   inc post
endwhile
```

This will give us the insert position as long as the *newscore* is greater than at least one of the existing top scores. However, there would be a problem if this was not the case : the **while** loop would fail to terminate! We could try adding a second condition to the loop so that it terminates if we arrive at the end of the array:

```
post = 1
while list[post] >= newscore and post <= SIZE
   inc post
```

Unfortunately, this leaves us with another problem: when *post* is incremented for the last time, it will be set to 6, then, as we loop back and test the first condition

```
list[post] >= newscore
```

we will be trying to access *scores[6]* - a cell which does not exist.

One way to solve this problem is to make the array one cell larger than we need:

```
dim list[SIZE++1]
```

which would mean that the array does contain a cell identified as *scores[6]* although we will never make use of that cell (we only need the top 5 scores).

A second option is to re-organise the conditions within the **while** statement:

```
while post <= SIZE and list[post] >= newscore
```

AGK BASIC implements **short-circuit evaluation**. When two conditions are ANDed together and the first is evaluated to false, the second condition is not tested. This means we won't attempt to access a non-existent element of *list*.

Now we have a situation exactly as before with a value and a position at which it is to be inserted. The only other change we need to make is to allow the lowest value in *scores[5]* to be eliminated when the lower values are shifted to make space for the new value. This can be done by making the first shift from cell 4 to cell 5 (the last cell) thereby overwriting the value previously held in cell 5.

The code for this is:

```
for current = 4 to post step -1
   new = current + 1
```

```
      list[new] = list[current]
   next current
```

A more flexible solution would be to initialise *current* to *SIZE* rather than 4. This would allow the number of high scores to be changed without having to alter any code other than the definition of *SIZE*. This gives us the new line:

```
for current = SIZE to post step -1
```

When we first begin to store the highest scores, *scores* will not be full and so we must increment the count held in *list[0]*, but once we have 5 high scores, the count should remain fixed. This requirement can be handled by the lines

```
if list[0] < SIZE
   inc list[0]
endif
```

The complete code for inserting a new high score is:

```
rem *** Get new score ***
Print("Enter new score")
Sync()
Sleep(1000)
newscore = GetButtonEntry()
rem *** Find insertion point ***
post = 1
while(post <= SIZE and list[post] >= newscore)
   inc post
endwhile
rem *** Create space for new score ***
for current = SIZE to post step -1
   new = current + 1
   list[new] = list[current];
next current
rem *** Add new new score ***
list[post] = newscore;
rem *** Increment count ***
if list[0] < SIZE
   inc list[0]
endif
```

With a few modifications we can make use of the program FIG-10.14 to test our code.

Activity 10.11

Start a new project called, *TopScores*. Compile the default code and copy the files required by the *Button* functions into the appropriate folders in the new project. Copy all the code from the latest version of *DataCount* to *TopScore's main.agc*.

In *TopScores*, modify case 1 in the `select` statement to match the code given above. (There is no requirement to check if the array is full.) Add an extra element to array *list*.

Test your program using the following data for the high scores:
23000, 11000, 17000, 46000, 9000
Display the list to make sure it is in descending order.

Add a new score 31000, and check that the score of 9000 is removed from the list.

Problem: Deleting a Value

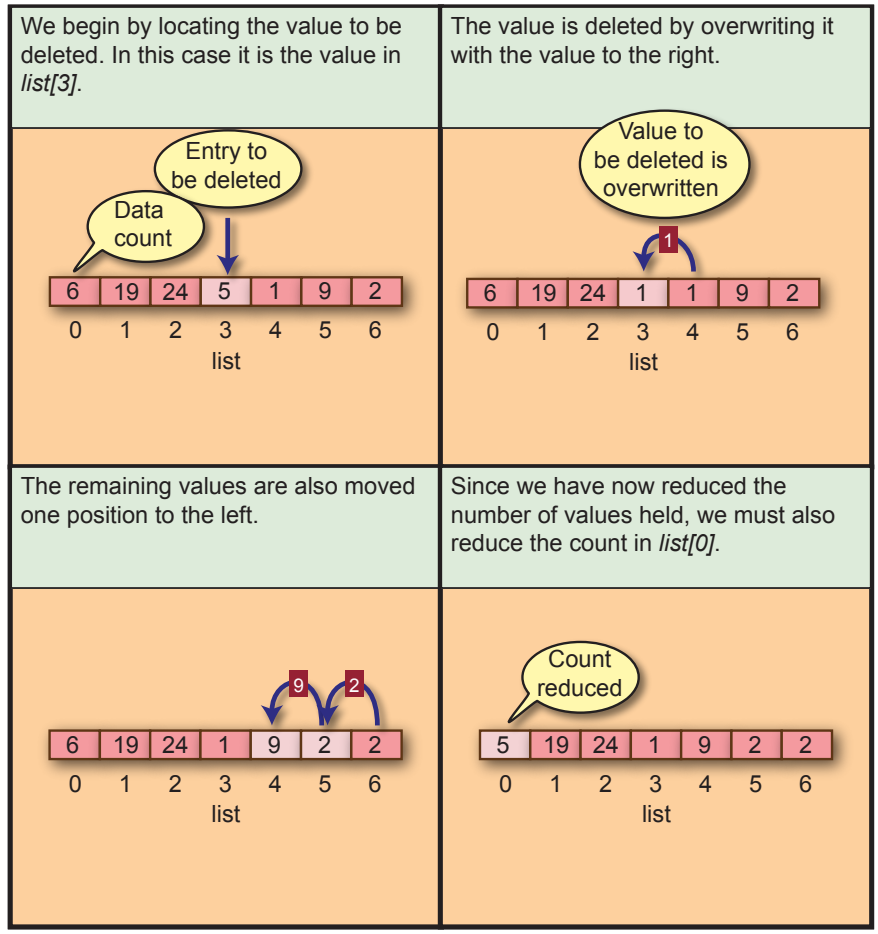
Some situations require an item of data to be deleted from an existing list. Sometimes we need to find and delete a specific value; other times way may want to delete the entry at a specific position in the list irrespective of its value.

Solution:

To delete a value from a list, we must first locate that value and then eliminate it from the list. FIG-10.19 shows the stages involved.

FIG-10.19

Deleting Data



Notice that *list[6]* retains a copy of the final value in the list, but this will have no effect on our code since, by reducing the count, the content of *list[6]* is no longer regarded as part of the valid data.

If we want to delete the value held at position *post*, then the logic required to move the other data items is:

```
rem *** Delete entry by moving subsequent entries to left ***
for current = post+1 to list[0]
   list[current-1] = list[current]
next current
rem *** Reduce count ***
dec list[0]
```

When we accept a value for *post*, we must ensure that we are attempting to delete from a position that contains data, so the following code is required:

```
rem *** Get position ***
Print("Enter position of item to be deleted")
Sync()
Sleep(1000)
post = GetButtonEntry()
while post < 1 or post > list[0]
   Print("The position is invalid. Re-enter.")
   Sync()
   Sleep(1000)
   post = GetButtonEntry()
endwhile
```

Activity 10.12

In *DataCount*, change the displayed menu so that the last two options are

4 - Delete from position
5 - QUIT

Make use of the code given above to add a new `case 4:` option in the `select` statement which deletes the data from a specified position in the list.

Change `case 5` : to be the quit option.

Change the condition in the `until` line to be `option = 5`.

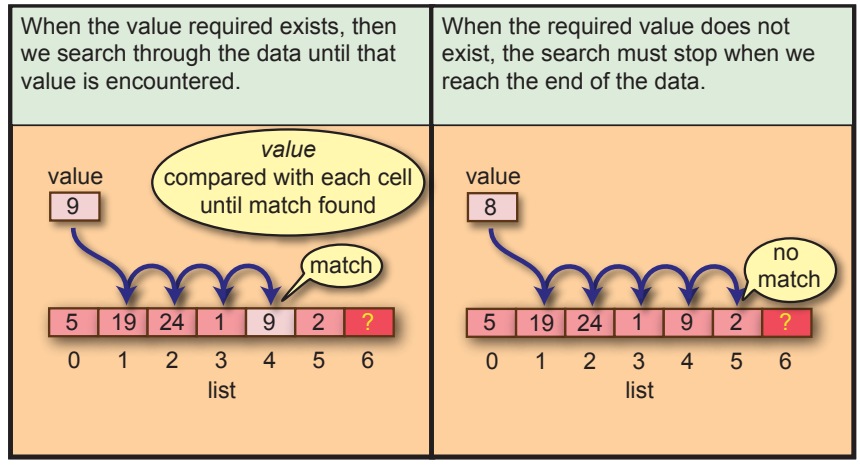
Test your new code by first setting up 5 values in the list and then deleting the last item of data, the third item of data and the first item of data. Display the contents of the list after each delete.

Save this updated version of *DataCount*.

When we want to delete a specific value from the list, we must first locate that value. This requires logic very like that required for locating the sentinel in a list. However, whereas we knew the sentinel would always appear in a sentinel-terminated list, we cannot make the same guarantees for user-selected values (see FIG-10.20).

FIG-10.20

Searching



Our new logic must allow for both possibilities.

What we require can be described in structured English as:

```
Get value to be deleted
Start at beginning of list
WHILE not arrived at end of data AND value to be deleted not found DO
    Move to next entry in the list
ENDWHILE
```

This translates into AGK BASIC as:

```
//*** Enter value to be deleted ***
Print("Enter value to be deleted")
Sync()
Sleep(1000)
value = GetButtonEntry()

//*** Search for value in list ***
post = 1
while (post <= list[0] and list[post] <> value)
   inc post
endwhile
```

Once the `while` loop has been completed, we need to check if the match was found. If it was, the cell we stopped at will contain a value matching the required value and this can be checked with the code:

```
if list[post] = value
```

Having found a match, the contents of the array can then be re-arranged to delete the specified entry and the count decremented.

Activity 10.13

Modify *Datacount* so that the value to be deleted - not its position - is entered. If the value to be deleted cannot be found, an appropriate message should be displayed.

Test Data: list 3, 6, 9, 12
Values to delete : 6, 12, 2

Display the list content after each deletion.

Save your project.

Problem: Converting Numbers to Text

A common requirement in a program handling dates is to display a day or a month in text rather than as a number. For example, sometimes we want to display the word *September* rather than number 9 when showing a date.

Solution:

To perform this task we can set up an array containing the months of the year in text form with the text for each month in the appropriate cell; so, cell 1 would contain the word *January*, cell 2 *February*, etc.

In the code given below, we have a string array, local to a function, which contains

the names of the months of the year, When supplied with the month of the year as a parameter, the function returns the corresponding string.

```
Print(MonthOfYear(8))
Sync()
do
loop

function MonthOfYear(v)
    if v < 1 or v > 12
        exitfunction ""
    endif
    dim month$[12] =["","January","February","March","April",
    ↳"May","June","July","August","September","October",
    ↳"November","December"]
    result$ = month$[v]
endfunction result$
```

Activity 10.14

Start a new project called *UsingStringArrays,* and implement the code given above.

Test and save the project.

Activity 10.15

In project *Shuffle* we made use of two `select` statements to display the card suit and the card values.

Modify *Shuffle* so that it makes use of string arrays to perform these tasks.

Test and save the project.

Dynamic Arrays

Sometimes it is not possible to know how large an array should be at the time we are writing a program. For example, let's say we need an array to hold the score achieved by each player in a multi-player game.

The number of elements needed in the array depends on the number of people who are actually playing on any specific occasion. To handle this situation, AGK BASIC allows the size of an array to be set using a variable. A snippet of the code required is shown below:

```
rem *** Find out how many people are playing ***
Print("Enter the number of players")
Sync()
Sleep(1000)
noofplayers = GetButtonEntry()
rem *** Set up an array of that size ***
dim scores[noofplayers]
```

Activity 10.16

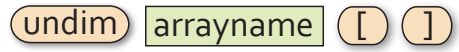
Start a new project called *DynamicArray*. The program should create an array of between 5 and 12 cells (this number to be chosen at random). Place a random value (between 1 and 20) in each cell and finally, display the contents of the array.

The undim Statement

If a program creates a particularly large array with thousands of elements, or has very many arrays, then it will occupy significant amounts of memory. This in turn may slow down the speed at which your program runs. To avoid this, it is possible to delete arrays which are no longer required using the `undim` statement which has the format shown in FIG-10.21.

FIG-10.21

The **undim** Statement



where:

arrayname is the name of the array to be deleted. The array must have been created earlier using a `dim` statement.

For example, if, at the start of a program we had created an array with the line

```
dim list[20]
```

then we could destroy that array later in our code using the line

```
undim list[ ]
```

Multi-dimensional Arrays

Could we represent the game of chess using an array? The problem here is that the chess board has rows and columns, while the arrays we have encountered up to now are just one long list of values. Luckily AGK BASIC allows us to create arrays which have both rows and columns. These are called **two-dimensional arrays**.

To do this we need to start by declaring our array using an extended form of the `dim` statement in which the number of rows and columns are specified. For example, if we wanted to keep the 6 best scores for 5 different players, we could set up a 5 row by 6 columns array called *scores* using the line

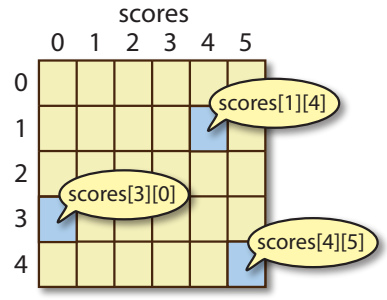
```
dim scores[4,5]
```

This would create the structure shown in FIG-10.22.

FIG-10.22

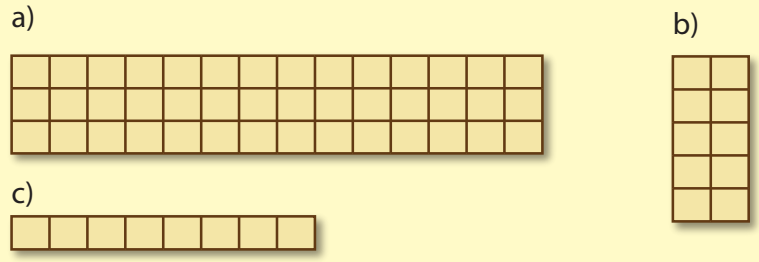
The *scores* 2D Array

This time we will make use of the row zero and column zero in the array.



Activity 10.17

Write the declarations necessary for the array structures pictured below (assume all hold `integer` values; use any name you wish).



To access an individual element within a two-dimensional array, we must specify the array name and the row and column numbers. The row and column values are separated by a comma. For example, we could store the value 23 in the top-left cell of the array *marks*, using the code:

```
marks[0,0] = 23
```

Unfortunately, there is no option to initialise multi-dimensional arrays.

We saw earlier how we could use a `for` loop to access each element of a one-dimensional array in turn. That same technique can be used to access a two dimensional array. The only difference this time is that we need to employ two `for` loops.

Returning to our *scores* array, we could store a random value in each cell using the following code:

```
for row = 0 to 4
   for col = 0 to 5
      scores[row,col] = Random(1,20)
   next col
next row
```

Activity 10.18

Start a new project called *Using2DArrays*. In *main.agc*, create the array *scores* with 5 rows and 6 columns.

Make use of the code given above to store a random value in the range 1 to 20 in each cell of the array.

Add more code to display the contents of each cell in the array. Display values from the same row on one line.

Test and save your program.

3-Dimensional Arrays and Higher

There are situations where we may need an array with even more dimensions. For example, if our players played with three levels of difficulty, then we would need an array which had three dimensions (5 players, 6 scores, 3 levels). We would define such an array with the statement:

```
dim scores[4,5,2]
```

AGK BASIC allows for arrays of up to 8 dimensions.

Arrays and Functions

Arrays cannot be used as function parameters nor as return values. If you want to make use of a non-local array within a function, then you must declare the array as a global variable as in the line

```
global dim numbers[20]
```

Summary

- Arrays can be used to hold a collection of values.
- Every value in an array must be of the same type.
- Arrays are created using the `dim` statement.
- The number of elements in an array can be specified as a constant, variable, or expression.
- Using a variable or expression to set the array's size allows that size to be varied each time the program is run.
- Numeric arrays are created with the value zero in every element.
- String arrays are created with empty strings in every element.
- The space allocated to an array can be freed using the `undim` statement.
- An array element is accessed by giving the array named followed by the element's subscript value enclosed in square brackets.
- The first element in an array has a subscript value of zero.
- The subscript can be a constant, variable or expression.
- Arrays can have up to eight dimensions.
- An array cannot be passed as a parameter to a function.
- A function cannot return an array as a result.

Solutions

Activity 10.1

The condition `newcore > score3` is true, so the lines executed will be

```
score5 = score4
score4 = score3
score3 = newscore
```

Activity 10.2

The code for *Array01*:

```
rem *** Using Arrays ***

rem *** Declare array ***
dim dice[4]
rem *** Store values in array ***
for c = 1 to 4
    dice[c] = Random(1,6)
next c
rem *** Display the values held ***
for c = 1 to 4
    Print(dice[c])
next c
Sync()
do
loop
```

Activity 10.3

The `for` loop will result in the following values being stored in *numbers*:

0,2,4,6,8,10,12,14,16

The final statement uses the contents of `numbers[2]` - which is 4 - minus `1` (which gives a result of 3) as the subscript in the expression

```
numbers[numbers[2]-1] = 23
```

so the line can be interpreted as

```
numbers[3] = 23
```

so the final contents of the array are

0,2,4,23,8,10,12,14,16

Activity 10.4

Modified code for *DiceCount*:

```
rem *** Dice throw counter ***
rem ** Declare array ***
dim counts[10]
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
  rem *** Throw dice ***
  dicethrow = Random(1,10)
  rem *** Add to appropriate count ***
  inc counts[dicethrow]
next c
rem *** Display each count ***
for c = 1 to 10
  Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

DiceCount with a constant:

```
rem *** Dice throw counter ***
#constant SIDES 10
rem ** Declare array ***
dim counts[SIDES]
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
  rem *** Throw dice ***
  dicethrow = Random(1,SIDES)
  rem *** Add to appropriate count ***
  inc counts[dicethrow]
next c

rem *** Display each count ***
for c = 1 to SIDES
  Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

The only change required to deal with a 20-sided dice is:

```
#constant SIDES 20
```

Activity 10.5

Modified code for *Lottery*:

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem *** FOR 6 times DO ***
for c = 1 to 6
  rem *** Generate an unselected number ***
  repeat
    rem ***Generate number ***
    number = Random(1,HIGHEST)
  until lottery[number] = 0
  rem *** Set corresponding cell ***
  lottery[number] = 1
  rem *** Display value ***
  Print(number)
next c
Sync()
do
loop
```

Modified code for *Lottery* (numbers in ascending order):

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem *** FOR 6 times DO ***
for c = 1 to 6
  rem *** Generate an unselected number ***
  repeat
    rem ***Generate number ***
    number = Random(1,HIGHEST)
  until lottery[number] = 0
  rem *** Set corresponding cell ***
  lottery[number] = 1
next c
rem *** Display subscript of cells containing 1 ***
for c = 1 to HIGHEST
    if lottery[c] = 1
        Print(c)
    endif
next c
Sync()
do
loop
```

Activity 10.6

Modified code for *Shuffle*:

```
dim list[20]

rem *** Set up values in array ***
for c = 1 to 20
  list[c] = c
next c
Print("Original Order")
rem *** Display contents ***
for c = 1 to 20
    PrintC(Str(list[c])+" ")
next c
Print("")
```

```
rem *** Shuffle ***
for c = 1 to 200
  rem *** Generate two subscript values ***
  sub1 = Random(1,20)
  sub2 = Random(1,20)
  rem *** Swap values at these positions ***
  temp = list[sub1]
  list[sub1] = list[sub2]
  list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("Shuffled order")
for c = 1 to 20
  PrintC(Str(list[c])+" ")
next c
Sync()
do
loop
```

Card version of *Shuffle*:

```
#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
  list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
  rem *** Generate two subscript values ***
  sub1 = Random(1,SIZE)
  sub2 = Random(1,SIZE)
  rem *** Swap values at these positions ***
  temp = list[sub1]
  list[sub1] = list[sub2]
  list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
  PrintC(Str(list[c])+" ")
next c
Sync()
do
loop
```

Notice that the value stored in *list[c]* is *c-1* (so that we are storing 0 to 51 rather than 1 to 52).

Named suits version of *Shuffle*:

```
#constant SIZE 52
dim list[SIZE]
rem *** Set up values in array ***
for c = 1 to SIZE
  list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
  rem *** Generate two subscript values ***
  sub1 = Random(1,SIZE)
  sub2 = Random(1,SIZE)
  rem *** Swap values at these positions ***
  temp = list[sub1]
  list[sub1] = list[sub2]
  list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
  PrintC(Str(list[c] mod 13+1) +" of ")
  select list[c] / 13
      case 0:
          Print("Hearts")
      endcase
      case 1:
          Print("Diamonds")
      endcase
      case 2:
          Print("Spades")
      endcase
      case 3:
          Print("Clubs")
      endcase
  endselect
next c
Sync()
do
loop
```

When the value of the card is displayed in the statement

```
PrintC(Str(list[c] mod 13 + 1 )+ " of  ")
```

the expression `list[c] mod 13` makes sure we have a value in the range 0 to 12. Since this is one less than the actual value of the card, we add 1 to the value (with the term `+1`).

The expression `list[c] /13` in the `select` statement determines the suit. Hearts cards have values between 0 and 12, so any of these values will give an answer of 0 when divided by 12 (remember integer division is performed); 13 to 25 is the diamonds (division by 12 gives a result of 1); etc. So the `select`'s expression will give a result between 1 and 4 giving the suit of the card.

The named cards version of *Shuffle*:

```
#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
  list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
  rem *** Generate two subscript values ***
  sub1 = Random(1,SIZE)
  sub2 = Random(1,SIZE)
  rem *** Swap values at these positions ***
  temp = list[sub1]
  list[sub1] = list[sub2]
  list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
    select list[c] mod 13+1
        case 1:
            PrintC("Ace")
        endcase
        case 11:
            PrintC("Jack")
        endcase
        case 12:
            PrintC("Queen")
        endcase
        case 13:
            PrintC("King")
        endcase
        case default
            PrintC(List[c] mod 13+1)
        endcase
    endselect
    PrintC(" of ")
    select list[c] / 13
        case 0:
            Print("Hearts")
        endcase
        case 1:
            Print("Diamonds")
        endcase
        case 2:
            Print("Spades")
        endcase
        case 3:
            Print("Clubs")
        endcase
   endselect
next c
Sync()
do
loop
```

The new `select` statement displays the appropriate term for cards with values 1, 11, 12, or 13, all other cards have their numeric value displayed.

Activity 10.7

No solution required.

Activity 10.8

Code for *SentinelData*:

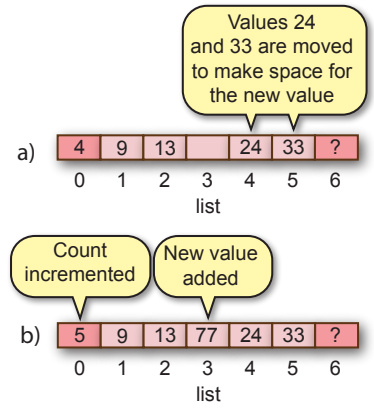
```
#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE]

rem *** Add sentinel value ***
list[0] = -99
//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-4)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    //*** Execute option ***
    select option
     case 1://*** Add a new value to the list ***
            Print("Enter value to be added : ")
            Sync()
            Sleep(2000)
            value = GetButtonEntry()
            rem *** IF list not full ***
            if list[SIZE] <> -99
                rem *** Search for sentinel ***
                post = 0
                while list[post] <> -99
                    inc post
                endwhile
                rem *** Insert new value... ***
                list[post] = value
                rem ***...followed by sentinel ***
                list[post+1] = -99
       else
         Print("List is full")
       endif
       Sync()
       Sleep(2000)
        endcase
        case 2:  //*** Display the number of items in
                 ↳the list ***
            rem *** Search for sentinel ***
            post = 0
            while list[post] <> -99
                inc post
            endwhile
            Print("The list contains "+Str(post)
            ↳+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3:  //*** Display the contents of the
                 ↳list ***
            if (list[0] = -99)
                Print("The list is empty")
            else
                Print("Values held are")
                post = 0
                while list[post] <> -99
                    PrintC(Str(list[post])+" ")
                    inc post
                endwhile
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 4
Sleep(2000)
end
```

Activity 10.9

A new value could be placed in cells 1, 2, 3, 4, or 5.



Activity 10.10

Modified version of *DataCount*:

```
#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE]

//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-4)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    //*** Execute option ***
    select option
     case 1://*** Add a new value to the list ***
            Print("Enter value to be added : ")
            Sync()
            Sleep(2000)
            value = GetButtonEntry()
            if  list[0] < SIZE
                rem *** Get insert position ***
                Print("Enter position")
                Sync()
                Sleep(1000)
                post = GetButtonEntry()
                while post < 1 or post > list[0]+1
                    Print("Position must be between
                     ↳1 and "+Str(list[0]+1))
                    Sync()
                    Sleep(1000)
                    post = GetButtonEntry()
                endwhile
                rem *** Make space for new value ***
                for c = list[0] to post step -1
                    list[c+1] = list[c]
                next c
                rem *** Increment count ***
                inc list[0]
                rem *** Insert new value ***
                list[post] = value
```

```
            else
              Print("List is full")
            endif
            Sync()
            Sleep(2000)
        endcase
        case 2:  //*** Display the number of items in
                 ↳the list ***
            Print("The list contains "+Str(list[0])
            ↳+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3:  //*** Display the contents of the
                 ↳list ***
            if (list[0] = 0)
                Print("The list is empty")
            else
                Print("Values held are")
                for c = 1 to list[0]
                    PrintC(Str(list[c])+" ")
                next c
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 4
Sleep(2000)
end
```

Activity 10.11

Code for *TopScores*:

```
#include "Buttons.agc"

#constant SIZE 5

dim list[SIZE+1]

//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-4)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    //*** Execute option ***
    select option
       case 1://*** Add a new value to the list ***

            rem *** Get new score ***
            Print("Enter new score")
            Sync()
            Sleep(1000)
            newscore = GetButtonEntry()
            rem *** Find insertion point ***
            post = 1
            while post <= SIZE and list[post] >=
            ↳newscore
                inc post
            endwhile
            rem *** Create space for new score ***
            for current = SIZE to post step -1
                new = current + 1
                list[new] = list[current];
            next current
            rem *** Add new new score ***
            list[post] = newscore;
            rem *** Increment count ***
            if list[0] < SIZE
                inc list[0]
            endif
        endcase
        case 2:  //*** Display the number of items in
                 ↳the list ***
            Print("The list contains "+Str(list[0])
            ↳+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3:  //*** Display the contents of the
                 ↳list ***
            if (list[0] = 0)
                Print("The list is empty")
            else
                Print("Values held are")
                for c = 1 to list[0]
                    PrintC(Str(list[c])+" ")
                next c
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 4
Sleep(2000)
end
```

Activity 10.12

Modified code for *DataCount*:

```
#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE+1]

//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - Delete from position")
    Print("5 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-5)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 5)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    //*** Execute option ***
    select option
        case 1://*** Add a new value to the list ***
            Print("Enter value to be added : ")
            Sync()
            Sleep(2000)
            value = GetButtonEntry()
            if  list[0] < SIZE
                rem *** Get insert position ***
                Print("Enter position")
                Sync()
                Sleep(1000)
                post = GetButtonEntry()
                while post < 1 or post > list[0]+1
                    Print("Position must be between
                     ↳and " + Str(list[0]+1))
                    Sync()
                    Sleep(1000)
                    post = GetButtonEntry()
                endwhile
                rem *** Make space for new value ***
                for c = list[0] to post step -1
                    list[c+1] = list[c]
                next c
                rem *** Increment count ***
                inc list[0]
                rem *** Insert new value ***
                list[post] = value
            else
```

```
                Print("List is full")
            endif
            Sync()
            Sleep(2000)
        endcase
        case 2:  //*** Display the number of items in
                 ↳the list ***
            Print("The list contains "+Str(list[0])
            ↳+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3://*** Display contents of list ***
            if (list[0] = 0)
                Print("The list is empty")
            else
                Print("Values held are")
                for c = 1 to list[0]
                    PrintC(Str(list[c])+" ")
                next c
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Delete from a specified
                ↳position ***
            rem *** Get position ***
            Print("Enter position of item to be
            ↳deleted")
            Sync()
            Sleep(1000)
            post = GetButtonEntry()
            while post < 1 or post > list[0]
                Print("The position is invalid.
                ↳Re-enter.")
                Sync()
                Sleep(1000)
                post = GetButtonEntry()
            endwhile
            rem *** Delete entry ***
            for current = post+1 to list[0]
                list[current-1] = list[current]
            next current
            rem *** Reduce count ***
            dec list[0]
        endcase
        case 5: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 5
Sleep(2000)
end
```

Activity 10.13

Modified code for *DataCount*:

```
#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE+1]

//*** Repeat until quit selected ***
SetUpButtons()
repeat
    //*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - Delete value")
    Print("5 - QUIT")
    //*** Get option ***
    Print("Enter option required(1-5)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 5)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    //*** Execute option ***
    select option
        case 1://*** Add a new value to the list ***
            Print("Enter value to be added : ")
            Sync()
            Sleep(2000)
            value = GetButtonEntry()
            if  list[0] < SIZE
                rem *** Get insert position ***
                Print("Enter position")
                Sync()
                Sleep(1000)
                post = GetButtonEntry()
                while post < 1 or post > list[0]+1
                    Print("Position must be between
                    ↳and " + Str(list[0]+1))
                    Sync()
                    Sleep(1000)
                    post = GetButtonEntry()
                endwhile
                rem *** Make space for new value ***
                for c = list[0] to post step -1
                    list[c+1] = list[c]
                next c
                rem *** Increment count ***
                inc list[0]
                rem *** Insert new value ***
                list[post] = value
            else
                Print("List is full")
            endif
            Sync()
            Sleep(2000)
        endcase
        case 2:  //*** Display the number of items in
                 ↳the list ***
            Print("The list contains "+Str(list[0])
            ↳+" entries")
            Sync()
            Sleep(2000)
        endcase
        case 3://*** Display contents of list ***
            if (list[0] = 0)
                Print("The list is empty")
            else
                Print("Values held are")
                for c = 1 to list[0]
                    PrintC(Str(list[c])+" ")
                next c
            endif
            Sync()
            Sleep(2000)
        endcase
        case 4: //*** Delete value ***
            //*** Enter value to be deleted ***
            Print("Enter value to be deleted")
            Sync()
            Sleep(1000)
            value = GetButtonEntry()
            //*** Search for value in list ***
            post = 1
            while (post <= list[0] and list[post] <>
            ↳value)
                inc post
            endwhile
            rem *** IF match found, delete entry ***
            if list[post] = value
                rem *** Delete entry ***
                for current = post+1 to list[0]
                    list[current-1] = list[current]
                next current
                rem *** Reduce count ***
                dec list[0]
            endif
        endcase
        case 5: //*** Quit program ***
            Print("Quitting program in 2 seconds")
            Sync()
        endcase
    endselect
until option = 5
Sleep(2000)
end
```

Activity 10.14

No solution required.

Activity 10.15

Modified code for *Shuffle*:

```
#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
  list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
  rem *** Generate two subscript values ***
  sub1 = Random(1,SIZE)
  sub2 = Random(1,SIZE)
  rem *** Swap values at these positions ***
  temp = list[sub1]
  list[sub1] = list[sub2]
  list[sub2] = temp
next c
rem *** Display shuffled items ***
dim values$[13]=["","Ace","2","3","4","5","6","7",
              ↳"8","9","10","Jack","Queen","King"]
dim suits$[4]=["","Hearts","Diamonds","Spades",
              ↳"Hearts"]
Print("First six cards")
for c = 1 to 6
  PrintC(values$[list[c] mod 13 + 1])
  PrintC(" of ")
  Print(suits$[list[c] / 13 + 1])
next c
Sync()
do
loop
```

The three display statements could even be combined into a single line:

```
Print(Str(values$[list[c] mod 13 + 1]))+" of "+
     ↳Str(suits$[list[c] / 13 + 1]))
```

Activity 10.16

Code for *DynamicArray*:

```
rem *** Decide size of array ***
size = Random(5,12)
rem *** Set up array ***
dim list[size]
rem *** Store a value in each cell ***
for c = 0 to size
    list[c] = Random(1,20)
next c
rem *** Display the contents of the array ***
for c = 0 to size
    Print(list[c])
next c
Sync()
do
loop
```

Activity 10.17

a) `dim matrix[2,13]`

b) `dim matrix [4,1]`

c) `dim list[7]` This is a one-dimensional array

Activity 10.18

Code for *Using2DArrays*:

```
rem *** Set up array ***
dim scores[4,5]
rem *** Store values in arrays ***
for row = 0 to 4
    for col = 0 to 5
        scores[row,col] = Random(1,20)
    next col
next row
for row = 0 to 4
    for col = 0 to 5
        PrintC(Str(scores[row,col])+" ")
    next col
    Print("")
next row
Sync()
do
loop
```