Data

In this Chapter:

- ❑ Constants
- ❑ Variables
- ❑ Naming Variables
- ❑ Assigning Values to Variables
- ❑ Arithmetic Operators
- ❑ Operator Precedence
- ❑ Random Numbers
- ❑ Determining the Elapsed Time

Program Data

Introduction

Every computer game has to store and manipulate facts and figures (more commonly known as **data**). For example, a program may store the name of a player, the number of lives remaining or the time left in which to complete a task.

We've already seen that all basic data can be grouped into three basic types:

Real values are also known as **floating-point** or simply **float** values.

integer	-	any whole number, positive, negative or zero
real	-	any number containing a decimal point
string	-	any collection of characters (may include numeric characters)

For example, if player *Ian Knot* had *3* lives and *10.6* minutes to complete a game, then:

3	is an example of an integer value
10.6	is a real value
Ian Knot	is an example of a string

Activity 3.1

Identify the type of value for each of the following :

a) -9	b) abc	c) 18	d) 12.8
e) ?	f) 0	g) -3.0	h) Mary had
i) 4 minutes	j) 0.023		

Constants

When a specific value appears in a computer program's code it is usually referred to as a **constant**. Hence, in the statement

```
Print(7)
```

the value 7 is a constant. When identifying a value as a constant, the constant's type is often included in the description, so, for example, 7 is an **integer constant**.

Activity 3.2

What type of constants are the following:

a) -12	b) Elizabeth	c) 3.14	d) 27.0

Variables

Most programs not only need to display information, but also need to store data and calculate results. To store data in AGK BASIC we need to use a **variable**. A variable is, in effect, reserved space within the computer's memory where a single value can be stored. Every variable in a program is assigned a unique name and can store only a single value. When a variable is first created, the type of value it can store (integer, real or string) is specified. No other type of value can be stored in that variable. For

example, a variable designed to store an integer value cannot store a string.

Integer Variables

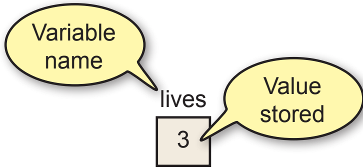
In AGK BASIC variables are created automatically as soon as we mention them in our code. For example, let's assume we want to store the number of lives allocated to a game player in a variable called *lives*. To do this in AGK BASIC we simply write the line:

```
lives = 3
```

This sets up a variable called *lives* in the computer's memory and stores the value 3 in that variable (see FIG-3.1)

FIG-3.1

Storing Data in a Variable



This instruction is known as an **assignment statement** since we are assigning a value (3) to a variable (*lives*).

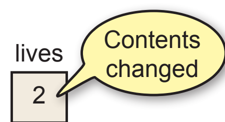
You are free to change the contents of a variable at any time by assigning it a different value. For example, we can change the contents of *lives* with a line such as:

```
lives = 2
```

When we do this, any previous value will be removed and the new value stored in its place (see FIG-3.2).

FIG-3.2

Changing the Value in a Variable



The variable *lives* is designed to store an integer value. In the lines below, *a, b, c, d*, and *e* are also integer variables. So the following assignments are correct

```
a = 200
b = 0
c = -8
```

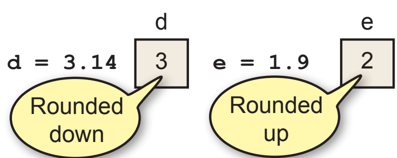
but the lines below will cause problems

```
d = 3.14
e = 1.9
```

since they attempt to store real constants in variables designed to hold integers. AGK BASIC won't actually report an error if you try out these last two examples, it simply rounds the fractional part of the numbers and ends up storing 3 in *d* and 2 in *e* (see FIG-3.3). Fractions of 0.5 and above are rounded up, other values are rounded down.

FIG-3.3

Integer Variables Round Real Values



Real Variables

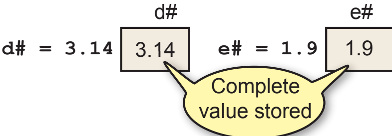
If you want to create a variable capable of storing a real number, then you must end the variable name with the hash (#) symbol. For example, if we write

```
d# = 3.14
e# = -1.9
```

we have created variables named *d#* and *e#*, both capable of storing real values (see FIG-3.4).

FIG-3.4

Real Variables



Any number (real or integer) can be assigned to a real variable, so we could write a statement such as:

```
d# = 12
```

Although we may assign an integer to a real variable, the value will be stored as a real. Therefore, when the statement above has been executed, *d#* will contain 12.0.

If any numeric value can be stored in a real variable, why bother with integer variables? Actually, you should always use integer values wherever possible because some hardware can be much faster at handling integer values than real ones. Also, real numbers can be slightly inaccurate because of rounding errors within the machine. For example, the value 2.3 might be stored as 2.2999987. Another consideration is that a real variable requires more space in the computer's memory than an integer one.

String Variables

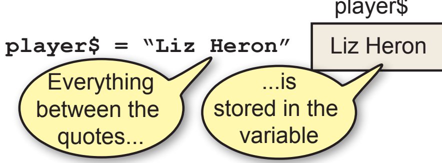
Finally, if you want to store a string value, you need to use a string variable. String variable names must end with a dollar ($) sign. The value to be stored must be enclosed in double quotes. We could create a string variable named *player$* and store the name *Liz Heron* in it using the statement:

```
player$ = "Liz Heron"
```

The double quotes are not stored in the variable (see FIG-3.5).

FIG-3.5

String Variables



Absolutely any value can be stored in a string variable as long as that value is enclosed in double quotes. Below are a few examples:

```
a$ = "?>%"
b$ = "Your spaceship has been destroyed"
c$ = "That costs $12.50"
d$ = ""     rem *** A string containing no characters ***
```

Activity 3.3

Which of the following are valid AGK BASIC statements that will store the specified value in the named variable?

a) `a = 6` b) `b = 12.89` c) `c = "Hello"`
d) `d$ = 5` e) `e$ = 'Goodbye'` f) `f# = -12.5`

Using Meaningful Names

It is important that you use meaningful names for your variables when you write a program. This helps you remember what a variable is being used for when you go back and look at your program a month or two after you wrote it. So, rather than write statements such as

```
a = 3
b = 120
c = 2000
```

a better set of statements would be

```
lives = 3
points = 120
timeremaining = 2000
```

which give a much clearer indication of the purpose of the variables.

Naming Rules

AGK BASIC, like all other programming languages, demands that you follow a few rules when you make up a variable name. These rules are:

- The name should start with a letter.
- Subsequent characters in the name can be a letter, number, or underscore.
- The final character can be a # (needed when creating real variables) or $ (needed when creating string variables).
- Upper or lower case letters can be used, but such differences are ignored. Hence, the terms *total* and *TOTAL* refer to the same variable.
- The name cannot be an AGK BASIC keyword.

➡ A keyword is any term that is used as part of the language. For example, if, then, for, repeat, etc.

This means that variable names such as

```
a, bc, de_2, fgh$, iJKlmnp#
```

are valid, while names such as

```
2a, time-remaining
```

are invalid.

2a - cannot start with a numeric digit.

time-remaining - hyphen not allowed.

The most common mistake people make is to have a space in their variable names (e.g. *fuel level*). This is not allowed. As a valid alternative, you can replace the space with an underscore (*fuel_level*) or join the words together (*fuellevel*). Using capital

letters for the joined words is also popular (*FuelLevel*).

Note that the names *no, no#* and *no$* represent three different variables; one designed to hold an integer value (*no*), one a real value (*no#*) and the last a string (*no$*).

> **Activity 3.4**
>
> Which of the following are invalid variable names:
>
> | | | |
> |---|---|---|
> | a) x | b) 5 | c) “total” |
> | d) al2$ | e) total score | f) ts#o |
> | g) then | h) G2_F3 | |

Named Constants

We have already seen that assigning meaningful names to the variables used in a program aids readability. When a program uses a fixed value which has an important role within the program (for example, perhaps the value 1000 is the score a player must achieve to win a game), then we have the option of assigning a name to that value using the `#constant` statement. The format of the `#constant` statement is shown in FIG-3.6.

FIG-3.6

#constant

value

where:

name is the name to be assigned to the constant value. A common convention is to assign an uppercase name making it easy to distinguish between variable names and constant names.

value is the constant value being named.

For example, we can name the value 1000 WINNINGSCORE using the line:

```
#constant WINNINGSCORE = 1000
```

Since the equal sign (=) is optional, it is also valid to write:

```
#constant WINNINGSCORE 1000
```

Real and string constants can also be named, but the names assigned must NOT end with # or $ symbols. Therefore the following lines are valid:

```
#constant PASSWORD = "neno"
#constant PI 3.14159
```

The value assigned to a name cannot be changed, so having written

```
#constant WINNINGSCORE = 1000
```

it is not valid to try to assign a new value later in the program with a line such as:

```
WINNINGSCORE = 1900
```

The two main reasons for using named constants in a program are:

1) Aiding the readability of the program. For example, it is easier to understand the meaning of the line

```
if playerscore >= WINNINGSCORE
```

than

```
if playerscore >= 1000
```

2) If the same constant value is used in several places throughout a program, it is easier to change its value if it is defined as a named constant. For example, if, when writing a second version of a game we decide that the winning score has to be changed from 1000 to 2000, then we need only change the line

```
#constant WINNINGSCORE = 1000
```

to

```
#constant WINNINGSCORE = 2000
```

On the other hand, if we've used lines such as

```
if playerscore >= 1000
```

throughout our program, every one of those lines will have to be changed so that the value within them is changed from 1000 to 2000.

Summary

- Fixed values are known as constants.
- There are three types of constants: integer, real and string.
- String constants are always enclosed in double quotes.
- The double quotes are not part of the string constant.
- A variable is a space within the computer's memory where a value can be stored.
- Every variable must have a name.
- A variable's name determines which type of value it may hold.
- Variables that end with the # symbol can hold real values.
- Variables that end with the $ symbol can hold string values.
- Other variables hold integer values.
- The name given to a variable should reflect the value held in that variable.
- When naming a variable the following rules apply:

  The name must start with a letter.
  Subsequent characters in the name can be numeric, alphabetic or the underscore character.
  The name may end with a # or $ symbol.
  The name must not be an AGK BASIC keyword.

- Constants can also be assigned a name.

Allocating Values to Variables

Introduction

There are several ways to place a value in a variable. Some of the AGK BASIC statements available to achieve this are described below.

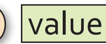
The Assignment Statement

In the last few pages we've used AGK BASIC's assignment statement to store a value in a variable. This statement allows the programmer to place a specific value in a variable, or to store the result of some calculation.

The assignment statement has the form shown in FIG-3.7.

FIG-3.7

The Assignment Statement



The value copied into the variable may be one of the following:

- a constant
- the contents of another variable
- the result of an arithmetic expression

Examples of each are shown below.

Assigning a Constant

This is the type of assignment we've seen earlier, with examples such as

```
name$ = "Liz Heron"
```

where a fixed value (a constant) is copied into the variable. As a general rule, make sure that the value being assigned is of the same data type as the variable. However, an integer value may be copied into a real variable, as in the line:

```
result# = 33
```

The program deals with this by storing the value assigned to *result#* as 33.0.

Activity 3.5

What are the minimum changes required to make the following statements operate correctly?

a) `desc = "tail"` b) `result = 12.34`

If you try copying a real value to an integer variable, the real value will be rounded to the nearest integer and that value stored in the variable. Hence, the line

```
number = 33.5
```

will result in the value 34 being stored in *number* (value rounded up), while the assignment

```
result = 12.2
```

will store 12 in *result* (value rounded down).

Copying Another Variable's Value

Once we've assigned a value to a variable in a statement such as

```
no1 = 12
```

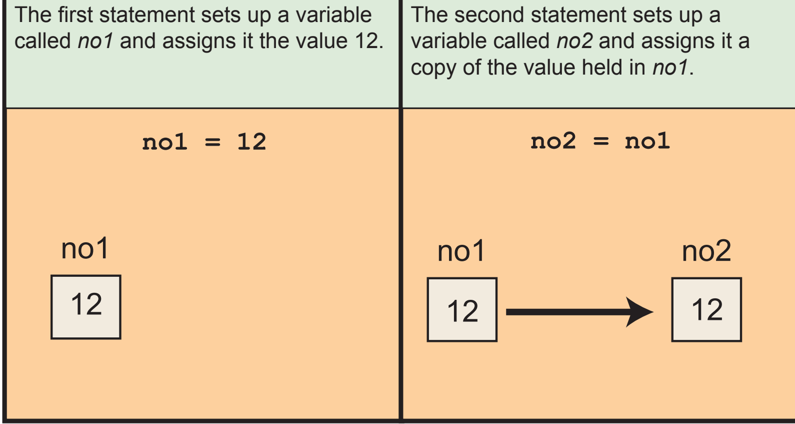
we can copy the contents of that variable into another variable with the command:

```
no2 = no1
```

The effect of these two statements is shown in FIG-3.8.

FIG-3.8

Copying from Another Variable



When the assignment is complete, both variables will contain the value 12. As before, you must make sure the two variables are of the same type, although the contents of an integer variable may be copied to a real variable as in the line:

```
ans# = no1
```

Copying the contents of a real variable to an integer variable will cause rounding to the nearest integer. For example,

```
ans# = -12.94
no1 = ans#
```

will store -13 in *no1*.

Activity 3.6

Assuming a program starts with the lines:

```
no1 = 23
weight# = 125.8
description$ = "sword"
```

which of the following instructions would be invalid?

a) `no2 = no1` b) `no3 = weight#` c) `result = description$`
d) `ans# = no1` e) `abc$ = weight#` f) `m# = description$`

Assigning the Result of an Arithmetic Expression

Another variation for the assignment statement is to have it perform a calculation and then store the result of that calculation in the named variable. Hence, we might write

```
no1 = 7 + 3
```

which would store the value 10 in the variable *no1*.

The example shows the use of the addition operator, but there are 6 possible operators that may be used when performing a calculation. These are shown in FIG-3.9.

FIG-3.9

Arithmetic Operators

Operator	Function	Example
+	addition	no1 = no2 + 5
-	subtraction	no1 = no2 - 9
*	multiplication	ans = no1 * no2
/	division	r1# = n01/ 2.0
mod	remainder	ans = no2 mod 3
^	power	ans = 2^3

The result of most statements should be obvious. For example, if a program begins with the statements

```
no1 = 12
no2 = 3
```

and then contains the line

```
total = no1 - no2
```

then the variable *total* will contain the value 9, while the line

```
product = no1 * no2
```

stores the value 36 in the variable *product*.

The remainder operator (`mod`) is used to find the integer remainder after dividing one integer into another. For example,

```
ans = 9 mod 5
```

assigns the value 4 to the variable *ans* since 5 divides into 9 once with a remainder of 4. Other examples are given below:

`6 mod 3`	gives 0
`7 mod 9`	gives 7
`123 mod 10`	gives 3

If the first value is negative, then any remainder is also negative:

`-11 mod 3`	gives -2

Activity 3.7

What is the result of the following calculations:

a) `12 mod 5`   b) `-7 mod 2`   c) `5 mod 11`   d) `-12 mod -8`

The power operator (^) allows us to perform a calculation of the form x^y. For example, a 24-bit address bus on the microprocessor of your computer allows 2^{24} memory addresses. We could calculate this number with the statement:

```
addresses = 2^24
```

Most of the results produced by these operators are easy to calculate manually as long as you are capable of basic arithmetic. However, the results of some statements are not quite so obvious. For example, you might expect the line

```
ans# = 19/4
```

to store the value 4.75 in *ans#*. In fact, the value stored will be 4.0. This is because the division operator always returns an integer result if the two values involved are both integer. On the other hand, if we write

```
ans# = 19/4.0
```

and thereby use a real value in the calculation, then the result stored in *ans#* will be the expected 4.75.

When using the division operator, a situation that you must guard against is division by zero. In mathematics, dividing any number by zero gives an undefined result, so most programming languages get quite upset if you try to get them to perform such a calculation. AGK BASIC, on the other hand, will, when presented with a line such as

```
ans = 10/0
```

store the value 0 in *ans*.

You might be tempted to think that you would never write such a statement, but a more likely scenario is that your program contains a line such as

```
ans = no1 / no2
```

and if *no2* contains the value zero, attempting to execute the line will still cause a value of zero to be stored in *ans*.

Some statements may not appear to make sense if you are used to traditional algebra. For example, what is the meaning of a line such as

```
no1 = no1 + 3
```

In fact, it means add 3 to *no1*. We can take the literal meaning of the statement to be:

Take the value currently stored in no1*, add 3, and store the result back in* no1.

Another unusual assignment statement is of the form:

```
no1 = -no1
```

The effect of this statement is to change the sign of the value held in *no1*. For example, if *no1* contained the value 12, the above statement would change that value to -12. Alternatively, if *no1* started off containing the value -12, the above statement would change *no1*'s contents to 12.

Activity 3.8

Assuming a program starts with the lines:

```
no1 = 2
v# = 41.09
```

what will be the result of the following instructions?

Treat each statement separately - don't assume the results are cumulative.

a) `no2 = no1^4`
b) `x# = v#*2`
c) `no3 = no1/5`
d) `no4 = no1 + 7`
e) `m# = no1/5`
f) `v2# = v# - 0.1`
g) `no1 = no1 + 1`
h) `no5 = -no1`

The inc and dec Statements

Because adding to or subtract from the existing value in a variable is so common, AGK BASIC has added statements specifically to perform those tasks.

The `inc` statement (short for *increment*) allows you to add 1 or any other value to the current contents of a variable. So rather than write

```
no1 = no1 + 1
```

we can write

```
inc no1
```

and in place of

```
num = num + 7
```

we can write

```
inc num, 7
```

Note that no value needs to be given when 1 is being added but for any other value the amount must be included in the statement

When subtracting, we can use `dec` statement (short for *decrement*) in the same way:

```
dec x       rem *** subtract 1 from x ***
dec y, 3    rem *** subtract 3 from y ***
```

So why offer two ways to achieve the same thing? Using `inc` and `dec` allows the compiler to create more efficient code than is possible when using the started assignment approach.

The format for the `inc` statement is shown in FIG-3.10.

FIG-3.10

The inc Statement

inc variable [, value]

where:

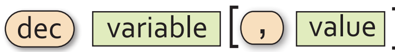
variable is the variable whose value is to be incremented.

value is a numeric value giving the amount to be added to the variable. If *value* is omitted then 1 is added to the variable.

The format for the `dec` statement is given in FIG-3.11.

FIG-3.11

The dec Statement



where:

variable is the variable whose value is to be decremented.

value is a numeric value giving the amount to be subtracted from the variable.

Operator Precedence

Of course, an arithmetic expression may have several parts to it as in the line

```
answer = no1 - 3 / v# * 2
```

and how the final result of such lines is calculated is determined by **operator precedence**.

If we have a complex arithmetic expression such as

```
answer = 12 + 18 / 3^2 - 6
```

then there's a potential problem about what should be done first when calculating the value of the expression. Will we start by adding 12 and 18 or subtracting 6 from 2, raising 3 to the power 2, or even dividing 18 by 3?

In fact, calculations are done in a very specific order according to a fixed set of rules. The rules are that the power operation (`^`) is always done first. After that comes remainder, multiplication and division with addition and subtraction done last. The power operator (`^`) is said to have a **higher priority** than remainder, multiplication and division; they in turn having a higher priority than addition and subtraction. So, to calculate the result of the statement above the computer begins by performing the calculation `3^2` which leaves us with:

```
answer = 12 + 18 / 9 - 6
```

Next the division operation is performed (18/9) giving:

```
answer = 12 + 2 - 6
```

The remaining operators, + and -, because they have the same priority, are performed on a left-to-right basis, meaning that we next calculate 12+2 giving:

```
answer = 14 - 6
```

Finally, the last calculation (14 - 6) is performed leaving

```
answer = 8
```

and the value 8 is stored in the variable *answer*.

Activity 3.9

What is the result of the calculation `12 - 5 * 12 / 10 - 5` ?

Using Parentheses

If we need to change the order in which calculations within an expression are performed, we can use parentheses. Expressions in parentheses are always done first. Therefore, if we write

```
answer = (12 + 18) / 9 - 6
```

then 12+18 will be calculated first, leaving:

```
answer = 30 / 9 - 6
```

➽ Remember we are dividing two integers so we get an integer result: 3.

The next calculation is 30 / 9 :

```
answer = 3 - 6
answer = -3
```

An arithmetic expression can contain many sets of parentheses. Normally, the computer calculates the value in the parentheses by starting with the left-most set.

Activity 3.10

Show the steps involved in calculating the result of the expression

```
8 * (6-2) / (3-1)
```

If sets of parentheses are placed inside one another (this is known as **nested parentheses**), then the contents of the inner-most set is calculated first. Hence, in the expression

```
12 / (3 * (10 - 6) + 4)
```

the calculations are performed as follows:

`(10 - 6)`	giving	`12 / (3*4+4)`
`3 * 4`	giving	`12 / (12 + 4)`
`12 + 4`	giving	`12 / 16`
`12 / 16`	giving	`0`

The order of precedence for all arithmetic operators is shown in FIG-3.12.

FIG-3.12

Operator Priority

Operator	Description	Priority
( )	parentheses	1
^	power	2
*	multiplication	3
/	division	3
mod	remainder	3
+	addition	4
-	subtraction	4

➽ Operators of equal priority are performed on a left-to-right basis.

Activity 3.11

Assuming a program begins with the lines `no1 = 12`, `no2 = 3`, and `no3 = 5` what would be the value stored in *answer* as a result of the line

```
answer = no1/(4 + no2 - 1)*5 - no3^2
```

Variable Range

When first learning to program, a favourite pastime of the beginner is to see how large a number the computer can handle, so people write lines such as:

```
no1 = 123456789000
```

They are often disappointed when the program crashes at this point.

There is a limit to the value that can be stored in a variable. That limit is determined by how much memory is allocated to a variable, and that differs from language to language.

Integer values in AGK BASIC can be in the range -2,147,483,648 to +2,147,483,647 while real values can be stored to about 7 decimal places.

String Operations

The + operator can also be used on string values to join them together. For example, if we write

```
a$ = "to" + "get"
```

then the value *toget* is stored in variable *a$*. If we then continue with the line

```
b$ = a$ + "her"
```

b$ will contain the value *together*, a result obtained by joining the contents of *a$* to the string constant "*her*".

> **Activity 3.12**
>
> What value will be stored as a result of the statement
>
> ```
> term$ = "abc"+"123"+"xyz"
> ```

The Print() Statement Again

We've already seen that the `Print()` command can be used to display values on the screen using lines such as:

```
Print(1)
Print("Hello")
```

We can also get the `Print()` statement to display the answer to a calculation. Hence,

```
Print(7+3)
```

will display the value 10 on the screen, while the statement

```
Print("Hello " + "again") rem ***Note the space after the o***
```

displays

```
Hello again
```

The `Print()` statement can also be used to display the value held within a variable. This means that if we follow the statement

```
number = 23
```

by the lines

```
Print(number)
Sync()
```

our program will display the value 23 on the screen, this being the value held in *number*. Real and string variables can be displayed in the same way. Hence the lines

```
name$ = "Charlotte"
weight# = 95.3
Print(name$)
Print(weight#)
Sync()
do
loop
```

will produce the output

```
Charlotte
95.3
```

Activity 3.13

A program contains the following lines of code:

```
number = 23
Print("number")
Print(number)
Sync()
```

What output will be produced by the two `Print()` statements?

Making Use of PrintC()

Although the `Print()` statement cannot display more than one value at a time, by using `PrintC()`, we can display two or more values on the same line of the screen.

For example, the code

```
capital$ = "Washington"
PrintC("The capital of the USA is ")
Print(capital$)
Sync()
do
loop
```

produces the following output on the screen:

```
The capital of the USA is Washington
```

➽ The second output statement uses `Print()` in order to move the cursor to a new line after all output is complete.

Activity 3.14

Start a new project (called *Name*) that sets the contents of the variable *name$* to *Jaqueline McKinnon* and then uses output statements that display the contents of *name$* in such a way that the final message on the screen becomes:

```
Hello, Jaqueline McKinnon, how are you today?
```

Another way to output a sequence of strings, this time using only a single `Print()` statement, is to join those strings together so only one data value is being output:

```
Print("Hello, " + name$ + ", how are you today?")
```

Activity 3.15

Modify *Name* so that it uses a single `Print()` statement to perform all its output. Test and save the modified code.

Acquiring Data

Data input can come in many forms: mouse, joystick, screen press, and keyboard are perhaps the obvious ones. AGK allows all of these and we'll be looking at each of those methods later in the book.

Another way to retrieve information is to access the hardware's timer. AGK offers only two timer options. One gives you access to the time your program has been running to the nearest second, the other gives the same information but this time to the nearest one thousandth of a second.

Timer()

Many of the statements we have looked at so far require you to supply them with information. For example, you have to supply `Print()` with the information you want displayed, while `SetClearColor()` requires the strength of the red, green and blue components that make up the background colour you want to use. Values supplied to commands of this type are known as **in parameters**.

The `Timer()` statement, on the other hand, supplies you with information - the time your program has been running. When a command supplies you with a value, that value is known as a **return value**.

Syntax diagrams for commands that return a value have the format shown in FIG-3.13.

FIG-3.13

Statements that Return a Value

return type (Command Name) (( ) [in parameters] ( ))

Notice that *return type* is not enclosed. That is because the *return type* is information about the type of value returned by the command, but not part of how the command is written.

The syntax diagram for the `Timer()` statement is shown in FIG-3.14.

FIG-3.14

Timer()

The diagram tells use that the `Timer()` statement returns a real value (also known as a **float** value) and that no *in parameters* are required by the statement. Notice that the parentheses must be included in the statement even though no information is placed within them. The actual value returned by `Timer()` is the time your program has been running to the nearest millisecond.

When a statement returns a value (as is the case with `Timer()`), generally we will want to do something with that returned value. Perhaps the most obvious thing to do is to store the result in a variable. Hence, we could add the line:

```
time_elapsed# = Timer()
```

We could then use that value in a calculation, for example

```
minutes = time_elapsed#/60
```

or simply display the value on the screen:

```
Print(time_elapsed#)
```

Activity 3.16

Start a new project called *Time*. Change the code in *main.agc* to read:

```
rem *** Get time passed ***
time_elapsed# = Timer()
do
   rem *** Display time ***
   PrintC("Time elapsed : ")
   Print(time_elapsed#)
   Sync()
loop
```

Compile and run the program.

You should see the time taken since the program started until the `Timer()` *command was executed. This should be much less than 1 second.*

Modify your program by moving the first two lines between the `do` and `loop` statements. Remember to change the indentation of the moved lines.

Compile and run the program. How does the output differ from the first version of the program?

The value returned by a statement doesn't have to be assigned to a variable. In the last exercise we assigned the value returned by `Timer()` to a variable then displayed the contents of that variable on the screen, but we can bypass the need for the variable by just printing the returned value directly with the line

```
Print(Timer())
```

which executes the `Timer()` statement then displays the value returned.

Activity 3.17

Modify *Time* so that the variable *time_elapsed#* is not required.

Test your modified program.

About Sync()

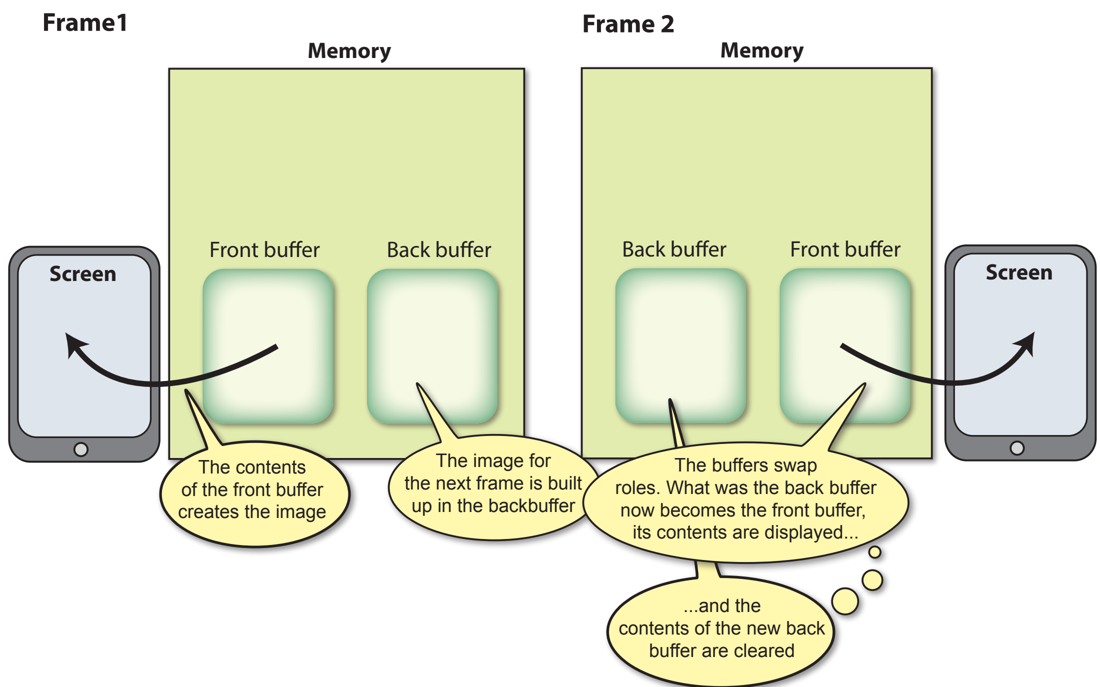
Let's take a moment out to get a deeper understanding of how `Sync()` works.

The contents of your screen are updated several times a second. Each update redraws the entire contents of the screen. Each redrawing is known as a **frame**.

To create a screen display, AGK reserves two areas of memory within your device. These areas of memory are known as **screen buffers**. The contents of one buffer are used to create the frame currently being displayed on the device's screen. This is known as the **screen buffer** or **front buffer**. At the same time, the contents of the second buffer (known as the **back buffer**) are being updated to contain the layout of the next frame.

FIG-3.15 shows how these buffers are used in the construction of a frame.

FIG-3.15 How the Screen Display is Produced



When a `Print()` or `PrintC()` statement is executed, the text to be displayed is copied into the current back buffer.

When a `Sync()` statement is executed, the two areas of memory swap function: what was the back buffer, becomes the front buffer and its contents appears on the screen; and what was the front buffer becomes the back buffer and its contents are cleared. It should be noted that handling the video buffers is not the `Sync()` statements only purpose, it also updates various other aspects of an application. We will examine these other aspects of `Sync()` in later chapters.

Understanding this will give you some insight as to where `Print()` and `PrintC()` statements need to be positioned within your program. Let's see how moving one of those statements affects the display of the *Time* project.

Activity 3.18

Since the message *Time elapsed :* never changes, try moving it before the `do` statement, then re-run your program.

What difference does this make to what is displayed?

After performing this, test, return the `PrintC()` statement to its original position after the `do` statement.

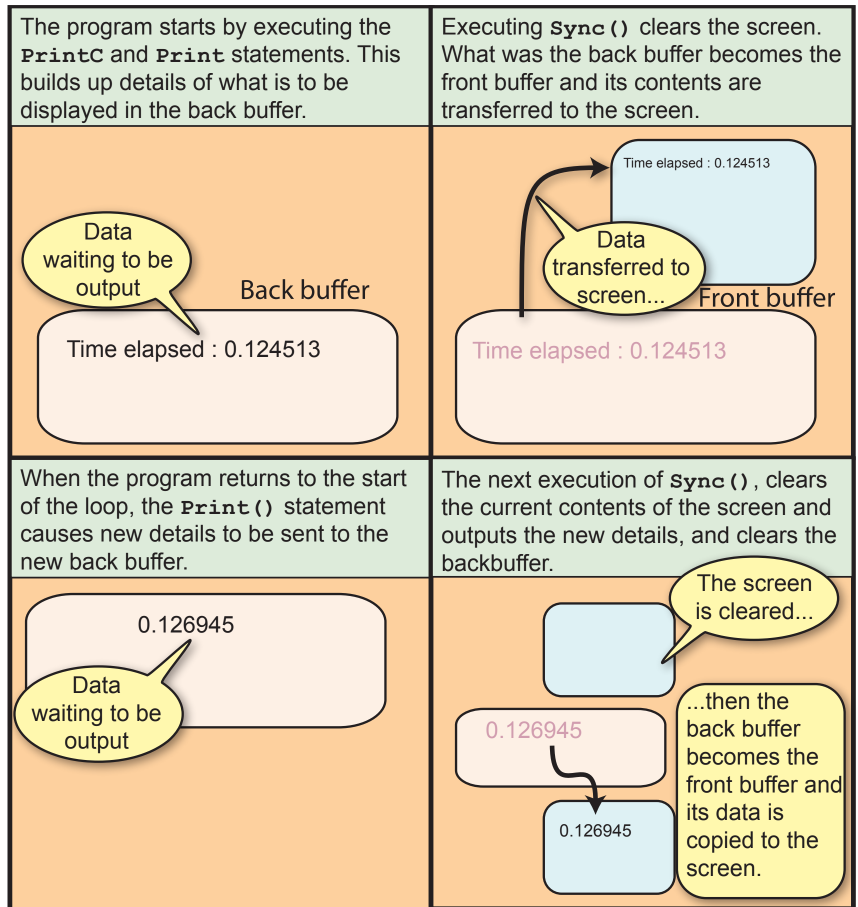
There is no need to resave your program.

So, why does the message no longer appear when we move it before the `do` statement? In fact, the message does appear, but it is gone so quickly that you won't have time to see it. After that, only the time appears.

FIG-3.15 explains the process involved when the first `PrintC()` statement appears before the `do`.

FIG-3.15

How Sync() Operates



The overall effect is that only values printed between one execution of `Sync()` and the next execution of `Sync()` will appear on the screen. If you want text to stay on the screen you need to reprint it between each execution of `Sync()`.

Timing Again

Most people are happier seeing a short period of time displayed in minutes and seconds rather than just seconds. To achieve this we can start by rounding the time elapsed to the nearest second using the line

```
total_seconds = Timer()
```

➽ Remember, moving a real value to an integer variable causes that value to be rounded to the nearest integer.

The number of minutes elapsed can now be calculated as *total_seconds* divided by 60:

```
minutes = total_seconds / 60
```

The remaining seconds (those not converted to minutes) give us the seconds part of our time. This is calculated as

➽ Remember, `mod` gives you the integer remainder after division has taken place.

```
seconds = total_seconds mod 60
```

The final version of our program is shown in FIG-3.16.

FIG-3.16

Displaying Time Elapsed in Minutes and Seconds

```
rem *** Display time elapsed in minutes and seconds ***

do
    rem *** Get time elapsed to nearest second ***
    total_seconds = Timer()
    rem *** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = total_seconds mod 60
    rem *** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

Activity 3.19

Modify your *Time* program to match the code given in FIG-3.16.

Compile and test your code.

ResetTimer()

Although the timer automatically starts tracking time from the moment your program begins executing, you can reset that timer to zero using the `ResetTimer()` statement (see FIG-3.17).

FIG-3.17

GetSeconds()

Notice that this statement has neither in parameters nor a return value, instead it modifies the contents of a variable maintained by AGK itself.

GetMilliSeconds()

While `Timer()` returns the time elapsed since the start of the program (or since the last execution of `ResetTimer()`) in seconds, you can have that same value in

milliseconds by using the `GetMilliSeconds()` statement (see FIG-3.18).

FIG-3.18

GetSeconds()

integer GetMilliSeconds ( )

GetSeconds()

If you are only interested in the time elapsed to the nearest second, you can use the `GetSeconds()` statement rather than `Timer()`. `GetSeconds()` has the format shown in FIG-3.19.

FIG-3.19

GetSeconds()

> **Activity 3.20**
>
> Modify *Time* to use `GetSeconds()` instead of `Timer()`. Test your new code.

Sleep()

It is possible to get a program to do nothing for a set period of time. As a general rule this is undesirable in a highly animated, interactive game, but for simple games such as those we will create in the early chapters of this book, getting a program to stop or slow down can be of use to us. For example, it may be used to give us the time to read a message on the screen.

Halting a program for a specific time is achieved using the `Sleep()` statement (see FIG-3.20).

FIG-3.20

Sleep()

where:

imillisecs is an integer value giving the time in milliseconds for which the program execution is to halt.

> **Activity 3.21**
>
> Modify your *Time* program adding the line
>
> ```
> Sleep(2000)        rem *** halt for 2 seconds ***
> ```
>
> immediately after the line containing `do`.
>
> Run the program. How has the new line affected the program?

Generating Random Numbers

Often in a game we need to throw a dice, choose a card or think of a number. All of these are random events. That is to say, we cannot predict what value will be thrown on the dice, what card will be chosen, or what number some other person will think of.

To help emulate these type of situations AGK BASIC offers several statements for the generation and manipulation of random values.

Random()

The `Random()` statement is used to generate a random number between lower and upper limits (see FIG-3.21).

FIG-3.21

Random()

where

ifrom is an integer giving the lowest value allowed.

ito is an integer giving the highest value allowed.

Activity 3.22

Start a new project (*Dice*) and create code to perform the following logic:

Throw a six-sided dice
Display the value thrown

Test the program by running it several times.

Save and close the project. We will return to this project frequently through the next few chapters.

The statement returns a random integer value in the range *ifrom* to *ito*. For example, if we wanted to simulate the throw of a dice, we could write

```
dice_throw = Random(1,6)
```

which would store a random value between 1 and 6 in *dice_throw*.

Notice that the syntax diagram tells us the parameters may be omitted allowing us to write a line such as

```
value = Random()
```

When no range of values is supplied, as in this example, the statement creates a random number in the range 0 to 65,535.

The program in FIG-3.22 shows another use of the `Random()` statement to create a random background colour for the app window.

FIG-3.22

Random Background Colour

```
rem *** Cycle through random background colours ***
do
    rem *** Generate a random value for each colour ***
    red = Random(0,255)
    green = Random(0,255)
    blue = Random(0,255)

    rem *** Clear the screen using the new colour ***
    SetClearColor(red,green,blue)
    Sync()
loop
```

Activity 3.23

Start a new project (*Background*) and enter the code given in FIG-3.22.

What happens when you run the program?

Immediately after the `Sync()` statement, add the lines

```
rem *** wait for 0.5 seconds ***
Sleep(500)
```

which will get the program to pause for half a second after each screen update. What difference does this make to the program?

Activity 3.24

Modify your *Background* project eliminating the need for the *red*, *green* and *blue* variables. Test your program to ensure it still works correctly.

We have already seen that the value returned by a statement can be assigned to a variable or displayed using a `Print()` statement, but we can also use the value returned by one statement as the parameter to another directly, without using a variable. Hence, we can replace the lines

```
red = Random(0,255)
green = Random(0,255)
blue = Random(0,255)
SetClearColor(red,green,blue)
```

with the line

```
SetClearColor(Random(0,255),Random(0,255),Random(0,255))
```

SetRandomSeed()

Computers can't really think of a random number all by themselves. Actually, they cheat and use a mathematical algorithm to calculate an apparently random number. As long as you don't know that algorithm, you won't be able to predict what number the computer is going to come up with, but because the numbers generated are not truly random, they are often referred to as **pseudo random numbers**.

The mathematical formula used needs to be supplied with an initial number to get started. This is known as the **seed value**. This seed value determines exactly what set of pseudo random numbers will be generated - use the same seed value on a second occasion and exactly the same set of numbers will be generated. To prevent this happening, the random number generator in AGK defaults to using the time from the system clock as a seed value. This ensures that a different value is used each time a program is run.

If you want to use your own seed value, you can do so using the `SetRandomSeed()` statement. The most likely reason for doing this is to ensure you use the same seed value on each run and hence the same set of random values. Normally, of course, you wouldn't want the same set of values, but it can be extremely useful when trying to find mistakes in a program. The `SetRandomSeed()` has the syntax shown in FIG-3.23.

FIG-3.23

SetRandomSeed()

SetRandomSeed

where:

iseed is an integer value which is used as the start-up for the formula used in the generation of pseudo random values.

> **Activity 3.25**
>
> Modify your *Dice* project so that the program starts by setting the seed value to 12.
>
> Run the program three times and check that the same number is generated each time. Remove the `SetRandomSeed()` line after testing is complete.

RandomSign()

A final statement that makes use of a random value is `RandomSign()` (see FIG-3.24).

FIG-3.24

RandomSign()

integer RandomSign (ivalue)

where:

ivalue is an integer value which will be returned as either its original value or as a negated form of the original. In other words, if *ivalue* was 12 then the returned value will be either 12 or -12. Each return option has a 50% chance of occurring.

One possible use for such a statement is to emulate any situation with two possible outcomes each with an equal possibility of occurring - for example, the flip of a coin.

User Input

For many games, the most important method of obtaining data is from the user. The game player, will be moving a mouse, a joystick, tapping on the screen, or typing at the keyboard. AGK has statements available for handling all of these (and more) but at this stage using these statements are a bit beyond what we have learned. On the other hand, being able to enter simple values is very useful when trying to demonstrate some of the fundamental concepts in programming.

To allow us a simple way to enter integer values, two functions are included in the download for this book. These functions are:

The term **function** may, for the moment, be taken to have the same meaning as **program statement.**

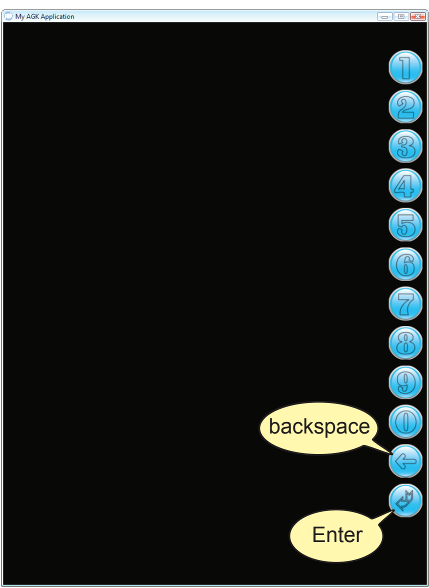
SetUpButtons() This function sets up 12 round buttons on the right of the app window. The buttons are labelled 0 to 9, 🡐(*backspace*) and ⏎(*Enter*).

GetButtonEntry() This function allows you to type in an integer value using the 12 buttons. Pressing the *backspace* button will remove the last character entered. Pressing *Enter* completes the data entry and returns the value entered.

The screen displayed when the buttons are used is shown in FIG-3.25.

FIG-3.25

Buttons Layout



The buttons are placed along the right edge to make them easy to press when the app is being used on a handheld device. If you want to use these new functions in any of your projects, you have to follow a few simple steps. These are shown in FIG-3.26.

FIG-3.26

Using the Buttons

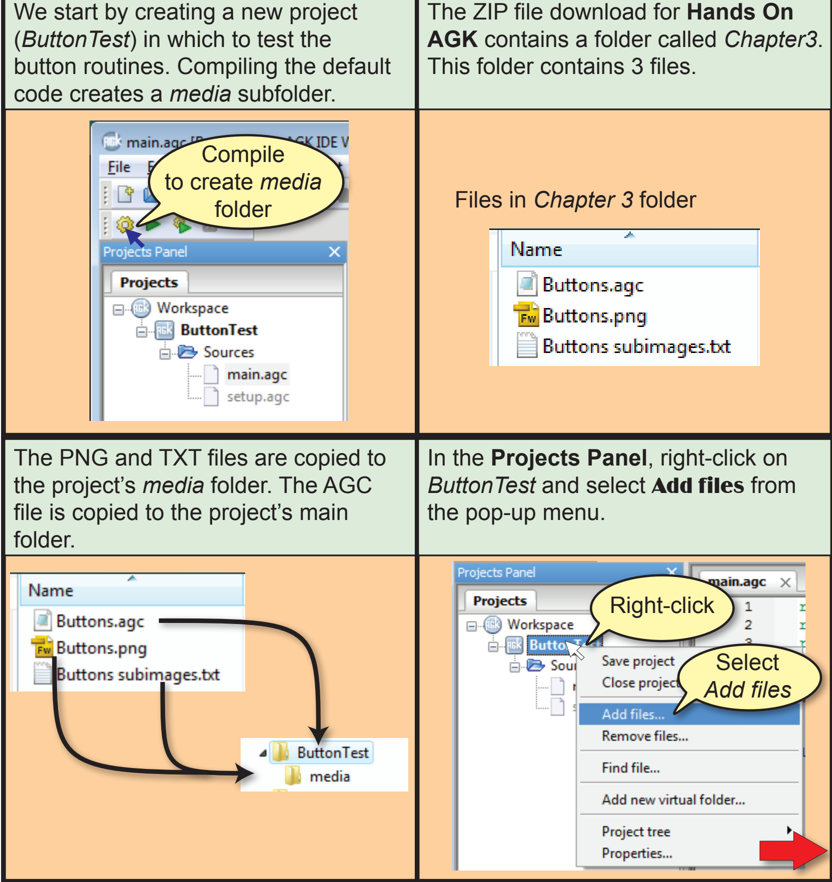
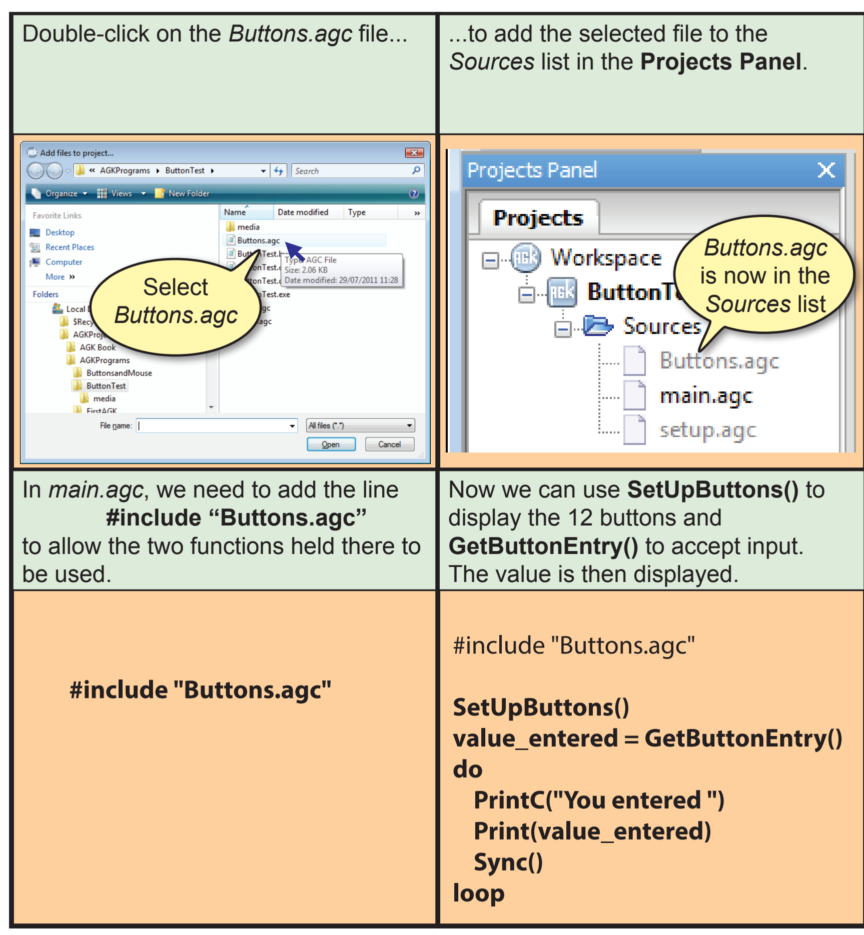


FIG-3.26

Using the Buttons



The complete code (with comments) for *main.agc* is shown in FIG-3.27.

FIG-3.27

Button Input

```
rem *** Command to include other source files used ***

#include "Buttons.agc"

rem *** Display the buttons ***
SetUpButtons()
rem *** Get an integer value from the buttons ***
value_entered = GetButtonEntry()
do
   rem *** Display the value entered ***
   PrintC("You entered ")
   Print(value_entered)
   Sync()
loop
```

The buttons are best suited to an app window optimised for the iPad's resolution of 1024 pixels high by 768 pixels wide, so we need to change the appropriate lines within the project's *setup.agc* to:

```
width=768
height=1024
```

Activity 3.26

Start a new project called *TestButtons*.

Compile the project in order to create the *media* subfolder.

From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the *TestButtons* project's *media* folder.

From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Modify the contents of the project's *main.agc* so that the code matches that given in FIG-3.25.

Modify *setup.agc* so that the width is set to 768 and the height to 1024.

Compile and run the program, checking that you can enter and delete characters using the buttons.

Check that the number displayed when you press the *Enter* key matches the value you typed in.

Save and close your project.

Activity 3.27

Reload your *Dice* program.

Make the necessary adjusts to allow you to use button input in the program.

Modify the logic of *main.agc* to match the following structured English description:

```
Display the set of input buttons
Generate a random number between 0 and 9
Display "Guess what my number is"
Get a value entered on the buttons
Display "My number was " and the game's number
Display "Your guess was " and the value entered
```

The last two displays should appear on screen at the same time.

Compile and check your program by running it three times.

Resave your project.

We will be making use of the button input code in a few programs. The process for using the code is always the same:

```
Copy the three files to the project's folders
Add a #include statement to the start of main.agc
Call the functions as required by the program logic
Modify the dimensions specified in setup.agc
```

Summary

- The assignment statement takes the form

```
variable = value
```

  value can be a constant, other variable, or an expression.

- The value assigned should be of the same type as the receiving variable.
- Arithmetic expressions can use the following operators:

```
^ mod * / + -
```

- Calculations are performed on the basis of highest priority operator first and a left-to-right basis.
- The power operator has the highest priority; multiplication and division and the remainder operator the next highest, followed by addition and subtraction.
- Terms enclosed in parentheses are always performed first.
- The + operator can be used to join strings.
- AGK uses a pseudo random number algorithm to create apparently random numbers within a specified range.
- The values generated are determined by an initial seed value.
- The default seed value for the algorithm is taken from the system's time.
- The seed value can also be set in the program code.
- Random integer values within a specified range can be created.

Testing Sequential Code

The programs in this chapter are very simple ones, with the statements being executed one after the other, starting with the first and ending with the last. In other words, the programs are sequential in structure.

Every program we write needs to be tested. For a simple sequential program which accepts user input, the minimum testing involves thinking of a value to be entered, predicting what result this value should produce, and then running the program to check that we do indeed obtain the expected result from that test data.

The program below (see FIG-3.28) reads in a value from the buttons and displays the square root of that number.

FIG-3.28

Calculating the Square Root

```
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Display prompt ***
Print("Enter a number : ")
Sync()
Sleep(2000)
rem *** Get value ***
no = GetButtonEntry()
rem *** Calculate square root ***
sqroot# = no^0.5
do
   rem *** Display result ***
   PrintC("Square root of ")
   PrintC(no)
   PrintC(" is ")
   Print(sqroot#)
   Sync()
loop
```

Activity 3.28

Start a new project called *SquareRoot*.

Perform the operations necessary so you can make use of button input in the program. Set the app windows dimensions to 1024 x 768.

Recode *main.agc* to match the lines given in FIG-3.28.

Compile the program but do not run it.

To test this program we might decide to enter the value 16 with the expectation of the displayed result being 4.

Activity 3.29

Test *SquareRoot* using the value 16.

Did you achieve the expected result?

Perhaps that one test would seem sufficient to say that the program is functioning correctly. However, a more cautious person might try a few more values just to make sure. But what values should be chosen? Should we try 25 or 9, 3 or 7?

As a general rule it is best to think carefully about what values you choose as test data. A few carefully chosen values may show up problems when many more randomly chosen values show nothing.

When the test data involves numeric values only, perhaps the most obvious categories are positive numbers, negative numbers and zero (which is neither negative or positive).

We have already tried a positive number (16), so perhaps we should try -9, say, and, of course, zero.

But in each case it is important that you work out the expected result before entering your test data into the program - otherwise you have no way of knowing if the results you are seeing on the screen are correct.

Activity 3.30

What results would you expect from *SquareRoot* if your test data was
0 and -9

Run the program with these test values and check that the expected results are produced.

When the value being entered by the user is a string, perhaps the test data could be:

a string with zero characters (just press the *Enter* when asked for data)
a string with only a single character
a string containing multiple characters

Of course, these suggestions for creating test data will almost certainly need to be modified depending on the nature of the program you are testing.

Solutions

Activity 3.1

a) Integer	b) String	c) Integer	d) Real
e) String	f) Integer	g) Real	h) String
i) String	j) Real		

Activity 3.2

a)	-12	integer constant
b)	Elizabeth	string constant
c)	3.14	real constant
d)	27.0	real constant

Activity 3.3

a) Valid
b) Invalid. Stores 13
c) Invalid - not a string variable
d) Invalid - remove $ from variable name or put double quotes round the 5.
e)Invalid. Must be double quotes, not single quotes.
f) Valid.

Activity 3.4

a) Valid
b) Invalid. Must start with a letter
c) Invalid. Names cannot be within quotes.
d) Valid
e) Invalid. Spaces are not allowed in a name
f) Invalid. # must appear at the end of the name
g) Invalid, then is a BASIC keyword
h) Valid

Activity 3.5

a) `desc$="tall"`
b) `result#= 12.34`

Activity 3.6

a) Valid
b) Invalid. Fraction part rounded
c) Invalid. A string cannot be copied to an integer variable
d) Valid
e) Invalid. A real cannot be copied to a string variable
f) Invalid. A string cannot be copied to a real variable

Activity 3.7

a) 2
b) -1
c) 5
d) -4

Activity 3.8

a) *no2* is 16
b) *x#* is 82.18
c) *no3* is zero
d) *no4* is 9
e) *m#* is 0.0
f) *v2#* is 40.99
g) *no1* is 3
h) *no5* is -2

Activity 3.9

The result is 1
The expression is calculated as follows:

12-5* 12/10-5
12-60/10-5
12-6-5
6-5
1

In fact, AGK BASIC doesn't currently abide by the rules of priority completely with it performing the division before the multiplication in this example which results in an answer of 2 rather than 1!

Activity 3.10

Steps:

8*(6-2)/(3-1)
8*4/(3-1)
8*4/2
32/2
16

Activity 3.11

answer =	no1 / (4 + no2 - 1) * 5 - no3 ^ 2	
answer =	12 / (4 + 3 - 1)	* 5 - 5 ^ 2
answer =	12 / (7 - 1)	* 5 - 5 ^ 2
answer =	12 / 6	* 5 - 5 ^ 2
answer =	12 / 6	* 5 - 25
answer =	2	* 5 - 25
answer =	10	- 25
answer =	-15	

Activity 3.12

term$ will hold the string *abcl23xyz*

Activity 3.13

Output:

number
23

Activity 3.14

The program code:

```
name$ = "Jaqueline McKinnon"
do
  PrintC("Hello, ")
  PrintC(name$)
  Print(", how are you today?")
  Sync()
loop
```

Note the spaces inside the quotes to make sure there are gaps either side of the name.

Activity 3.15

The program code:

```
name$ = "Jaqueline McKinnon"
do
  Print("Hello, "+name$+", how are you today?")
  Sync()
loop
```

Activity 3.16

Modified code:

```
do
    rem *** Get time passed ***
```

```
        time_elapsed# = Timer()
        rem *** Display time ***
        PrintC("Time elapsed : ")
        Print(time_elapsed#)
        Sync()
    loop
```

The time displayed on the screen now updates continuously.

Activity 3.17

Modified code:

```
do
    rem *** Display time passed ***
    PrintC("Time elapsed : ")
    Print(Timer())
    Sync()
loop
```

Activity 3.18

Modified code:

```
PrintC("Time elapsed : ")
do
    rem *** Display time passed ***
    Print(Timer())
    Sync()
loop
```

Each time the `Sync()` statement is executed, only the contents of `Print()` or `PrintC()` statements executed since the previous execution of `Sync()` are displayed. Since the `PrintC()` statement above is executed only once, its message disappears the second time the `Sync()` statement is executed.

Activity 3.19

No solution required.

Activity 3.20

Modified code:

```
rem *** Display time elapsed in ***
rem *** minutes and seconds ***
do
    rem *** Get time elapsed to nearest second ***
    total_seconds = GetSeconds()
    rem *** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = total_seconds mod 60
    rem *** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

Activity 3.21

Modified code:

```
rem *** Display time elapsed in ***
rem *** minutes and seconds ***
do
    Sleep(2000) rem *** halt for 2 seconds ***
    rem *** Get time elapsed to nearest second ***
    total_seconds = GetSeconds()
    rem *** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = total_seconds mod 60
    rem *** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

The change means that the screen is only updated every 2 seconds so we see the time pass in 2 second steps.

Activity 3.22

Program code:

```
rem *** Dice program ***
rem *** Simulates the roll of a 6-sided dice ***
rem *** Throw dice ***
dice = Random(1,6)
do
  rem *** Display value thrown ***
  PrintC("Value thrown was : ")
  Print(dice)
  Sync()
loop
```

Activity 3.23

The colours change so quickly that there is not time to update the whole background before the colour changes again so bands of colour appear.

Modified code:

```
rem *** Cycle through random background colours ***
do
    rem *** Generate value for each colour ***
    red = Random(0,255)
    green = Random(0,255)
    blue = Random(0,255)

    rem Clear the screen using the new colour ***
    SetClearColor(red,green,blue)
    Sync()
    rem *** wait for 0.5 seconds ***
    Sleep(500)
loop
```

Now there is enough time to show the selected colour over the whole background before another colour is generated.

Activity 3.24

Modified Code:

```
rem *** Cycle through random background colours ***
do
    rem Clear the screen using random colour ***
    SetClearColor(Random(0,255),Random(0,255),
    ↳Random(0,255))
    Sync()
    rem *** wait for 0.5 seconds ***
    Sleep(500)
loop
```

Note The symbol ↳ is used to indicate the continuation of a single line of code.

Activity 3.25

Modified code:

```
rem *** Dice program ***
rem *** Simulates the roll of a 6-sided dice ***

rem *** Seed random number generator ***
SetRandomSeed(12)
rem *** Throw dice ***
dice = Random(1,6)
do
  rem *** Display value thrown ***
  PrintC("Value thrown was : ")
  Print(dice)
  Sync()
loop
```

The program always generates a 6.

Activity 3.26

No solution required.

Activity 3.27

Reload your *Dice* project.
Modify the *startup.agc* file setting the width to 768 and the height to 1024.
From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* folder.
From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Right click on **Dice** in the Projects Panel.
Select **Add files** from the popup menu.
Select *Buttons.agc* from the files listed.

Program code:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
do
   rem *** Display values ***
   PrintC("My number was : ")
   Print(dice)
   PrintC("Your guess was : ")
   Print(guess)
   Sync()
loop
```

Activity 3.28

Start a new project called *SquareRoot*.
Compile the project to create the media folder.
Modify the *startup.agc* file setting the width to 768 and the height to 1024.
From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* folder.
From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Right click on **SquareRoot** in the Projects Panel.
Select **Add files** from the popup menu.
Select *Buttons.agc* from the files listed.

Change the contents of *main.agc* to match that given in FIG-3.24.
Compile the program.

Activity 3.29

Running the program using the value of 16 gives the result 4.0.

Activity 3.30

The expected result using the value zero would be zero.
Using -9 should result in an error since negative values do not have a square root.

Selection

In this Chapter:

- **if..endif** Statement
- Conditions
- Relational Operators
- Boolean Operators
- **if..then** Statement
- Nested if Statements
- Testing Selection Structures

Binary Selection

Introduction

As we saw in structured English, many algorithms need to perform an action only when a specified condition is met. The general form for this statement was:

```
IF condition THEN
    action
ENDIF
```

Hence, in our guessing game, we described the response to a correct guess as:

```
IF guess = dice THEN
    Say “Correct”
ENDIF
```

As we’ll see, AGK BASIC also makes use of an `if` statement to handle such situations.

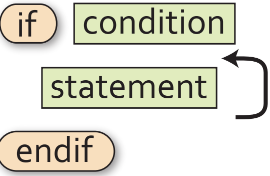
if

In its simplest form, the `if` statement in AGK BASIC takes the format shown in FIG-4.1.

FIG-4.1

if (format 1)

Unlike the IF in structured English, AGK BASIC does not use the word **`then`**.



where:

condition is any term which can be reduced to a true or false value.

statement is any executable AGK BASIC statement.

The diagram also tells us that we can have as many statements between *condition* and `endif` as we require.

If *condition* evaluates to true, then the set of statements between the `if` and `endif` terms are executed; if *condition* evaluates to false, then the set of statements are ignored and execution moves on to any statements following the `endif` term.

Condition

Generally, the condition will be an expression in which the relationship between two quantities is compared. For example, the condition

```
no < 0
```

will be true if the content of the variable *no* is less than zero (i.e. negative).

A condition is sometimes referred to as a **Boolean expression** and has the general format given in FIG-4.2.

FIG-4.2

Boolean Expression

value1 | relational operator | value2

where:

value1 and **value2** — may be constants, variables, or expressions.

relational operator — is one of the symbols given in FIG-4.3.

FIG-4.3

The Relational Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

From our syntax diagram, we can see that each of the following are valid conditions:

```
no1 < 7
answer# <> no1# * 2
gender$ = "female"
```

The values being compared should normally be of the same type, but it is acceptable to mix integer and real numeric values as in the conditions:

```
v > x#
t# < 12
```

However, it is not possible to compare a numeric against a string value. Therefore, conditions such as

```
name$ = 34
no1 <> "16"
```

are invalid.

Activity 4.1

Which of the following are NOT valid Boolean expressions?

a) `no1 < 0`
b) `name$ = "Fred"`
c) `no1 * 3 >= no2 - 6`
d) `v# => 12.0`
e) `total <> "0"`
f) `address$ = 14 High Street`

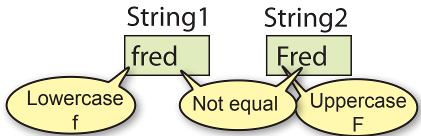
When two strings are checked for equality as in the condition

```
if name$ = "Fred"
```

the condition will only be considered true if the match is an exact one. Even the slightest difference between the two strings will return a *false* result (see FIG-4.4).

FIG-4.4

String Comparison 1



Spaces count as characters too. So if one or more spaces are included in a string, their number and positions within two strings must also match if the strings are to be considered equal. Since spaces are so important, you will occasionally see the space represented within a string as a triangle. So rather than show the contents of a string as

```
Hello world
```

you may see

```
Hello∆world
```

This is only done when clarification of the exact contents of a string is required. For example, the strings *hello* and *hello*∆ are not equal because the second string contains a space character after the letter *o*.

Not only is it valid to test if two string values are equal, or not, as in the conditions

```
if name$ = "Fred"
if village$ <> "Turok"
```

it is also valid to test if one string value is greater or less than another. For example, it is true that

```
"B" > "A"
```

Such a condition is considered true not because B comes after A in the alphabet, but because the coding used within the computer to store a "B" has a greater numeric value than the code used to store "A".

The method of coding characters is known as ASCII (American Standard Code for Information Interchange). This coding system is given in Appendix A at the back of the book.

If you are comparing strings which only contain letters, then one string is less than another if that first string would appear first in an alphabetically ordered list. Hence,

```
"Aardvark"    is less than    "Abolish"
```

But watch out for upper and lower case letters. All upper case letters are less than all lower case letters. Hence, the condition

```
"A" < "a"
```

is true.

If two strings differ in length, with the shorter matching the first part of the longer as

```
"abc" < "abcd"
```

then the shorter string is considered to be less than the longer string. Also, because the computer compares strings using their internal codes, it can make sense of a condition such as

```
"$" < "?"
```

which is also considered true since the $ sign has a smaller value than the ? character

in the ASCII coding system.

Activity 4.2

Determine the result of each of the following conditions (true or false). You may have to examine the ASCII coding at the end of the book for f).

a) `"wxy" = "w xy"` b) `"def" < "defg"` c) `"AB" < "BA"`
d) `"cat" = "cat."` e) `"dog" = "Dog"` f) `"*" > "&"`

Structured English to Code

It is not always obvious how to translate an IF statement written in structured English. In fact, some may take a great deal of coding. For example, the structured English

```
IF the text entered contains any punctuation marks THEN
    Remove the punctuation marks from the text
ENDIF
```

would require several lines of programming code to achieve. On the other hand, some statements that might look difficult to code are very simple:

Structured English:

```
IF number is negative THEN
    Make it positive
ENDIF
```

Code:

➽ Notice the use of indentation in the program listings. BASIC does not demand that this be done, but indentation makes a program easier to read - this is particularly true when more complex programs are written.

```
if number < 0
   number = -number
endif
```

Structured English:

```
IF number is even THEN
    Display "Even number"
ENDIF
```

Code:

```
if number mod 2 = 0
   Print("Even number")
endif
```

If you wanted the display to update immediately, you would also add `Sync()` after the `Print()` statement.

Activity 4.3

Start a new project *EnglishToCode*. The program will accept values from the screen buttons we used previously. The program should implement the following logic:

```
Read in values for no1 and no2
IF no1 is exactly divisible by no2 THEN
        Display "Exactly divisible"
ENDIF
```

Test your program.

Place the lines
```
do
loop
```
at the end of your code.

Using if

Activity 4.4

Load *Dice*, the project you created in Chapter 3.

Modify the program so that, after the player has typed in his guess, the program displays the word *Wrong* if the *guess* and *dice* values are not equal.

Test and save your program.

As we have already said, the syntax diagram for the `if` statement shows us that we can have more than one statement between the condition and the term `endif`. For example, if a game which used two dice required the dice to be re-thrown if they both showed the same value, then we would write:

```
if dice1 = dice2
   dice1 = Random(1,6)
   dice2 = Random(1,6)
endif
```

Activity 4.5

Modify the latest version of *Dice* so that, when the number generated differs from the guess, the program displays the word *Wrong* and also the difference between the two numbers. For example if the computer generates the value 8 and the player guesses 3 then the output would be:

```
Wrong. You were out by 5
My number was 8
Your guess was 3
```

Compound Conditions - the *and* and *or* Operators

Two or more simple conditions (like those given earlier) can be combined using either the term `and` or the term `or` (just as we did in structured English in Chapter 1).

The term `and` should be used when we need two conditions to be true before an action should be carried out. For example, if a game requires you to throw two sixes to win, this could be written as:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
if dice1 = 6 and dice2 = 6
   Print("You win!")
   Sync()
endif
```

The statements `Print("You win!")` and `Sync()` will only be executed if both conditions, `dice1= 6` and `dice2 = 6`, are true.

You may recall from Chapter 1 that there are four possible combinations for an `if` statement containing two simple expressions. Because these two conditions are linked by the `and` operator, the overall result will only be true when both conditions are true. These combinations are shown in FIG-4.5.

FIG-4.5

AND Combinations

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

We link conditions using the `or` operator when we require only one of the conditions given to be true. For example, if a dice game produces a win when the total of two dice is either 7 or 11, we could write the code for this as:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
total = dice1 + dice2
if total = 7 or total = 11
   Print("You win!")
   Sync()
endif
```

The four possible combinations for two conditions linked by an `or` are shown in FIG-4.6.

FIG-4.6

OR Combinations

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

When you use multiple conditions linked with `and` or `or`, each condition must be properly formed; you cannot shorten things the way you might in standard English. Hence, the compiler would not accept

```
if total = 7 or 11
```

Activity 4.6

Start a new project called *TwoDice*. Create a program using the two-dice code given above.

Add statements to display the values thrown on the two dice. This should appear irrespective of the values thrown. You will have to reposition the `Sync()` statement to get the program to operate correctly.

Test and save your program.

There is no limit to the number of conditions that can be linked using `and` and `or`. For example, a statement of the form

IF condition1 AND condition2 AND condition3

means that all three conditions must be true, while the statement

IF condition1 OR condition2 OR condition3

means that at least one of the conditions must be true.

Activity 4.7

Modify your *TwoDice* project so that the *You win!* message also appears if both dice have equal values.

Test and save your program.

Activity 4.8

Start a new project called *ThreeDice*.
In this game three dice are thrown. If at least two dice show the same value, the player has won.

Write a program which implements the following logic:

```
Throw all three dice
IF any two dice match THEN
    Display "You win!"
ENDIF
Display the value of each dice
```

Test and save your program.

A complex condition can also contain a mix of `and` and `or` operators. An obvious example of this is the description of how to save a file in AGK:

```
IF Save button pressed OR Ctrl key down AND S key pressed THEN
    Save current file
ENDIF
```

The trouble with conditions like this is that they are open to more than one interpretation. We could take it to mean:

> that we must press the *S* key while either clicking on the **Save** button or holding down the *Ctrl* key

rather than the intended

> either clicking on the **Save** button or holding down the *Ctrl* key while pressing the *S* key.

Once we start to create conditions containing both `and` and `or` operators, we need to be aware that Boolean operators (AND, OR and NOT) have a priority order just as arithmetic operators do. In a condition that contains both `and` and `or`, the `and` operator takes precedence over the `or` operator. Knowing this eliminates any ambiguity in the conditions for saving a file in the example above.

The normal rule of performing the `and` operation before `or` can be modified by the use of parentheses. Expressions within parentheses are always evaluated first. Hence, if we really did have to click on the press the *S* key while pressing the **Save** button or holding down the *Ctrl* key, we would write the condition as

```
(Save button pressed OR Ctrl key down) AND S key pressed
```

Activity 4.9

Write down formal conditions (including any necessary parentheses) for the following situations:

a) In the game of Monopoly any one of three situations causes your piece to "go to jail". These are: landing on the "Go to Jail" square, picking up a "Go to Jail" card, and, throwing the same value on both dice three times in a row.

b) In a video game, one way to win is to collect 10,000 gold pieces; an alternative is to free the princess from the tower and slay the dragon.

c) In a game of cards, you lose 100 points if you hold either the King or Queen of Spades when the Ace of Diamonds is played.

The not Operator

AGK BASIC's `not` operator works in exactly the same way as that described in Chapter 1. It is used to negate the final result of a Boolean expression.

In the *ThreeDice* project you created in Activity 4.8, the `if` statement used was

```
if dice1 = dice2 or dice1 = dice3 or dice2 = dice3
   Print("You win")
endif
```

Now, if we wanted to change the game to display "You lose" instead of "You win" then we would have to test for the opposite condition.

Activity 4.10

Without using the `not` operator, write down the condition that should be tested when displaying "You lose" in the dice game.

As you can see, working out the opposite condition takes a few moments - you may even have got it wrong on your first attempt. It's much easier, given that you already have the condition required for the "You win" message, just to add a `not` to the condition:

```
if not(dice1 = dice2 or dice1 = dice3 or dice2 = dice3)
   Print("You lose")
endif
```

Note that the original condition is placed in parentheses. This is because the `not` operator has an even higher priority than `and` and `or`. Without the parenthesis, the `not` operation would be applied to the first term only - `dice1 = dice2`.

The Boolean operator priority is shown in FIG-4.7.

FIG-4.7

Boolean Priority

Operator	Priority
( )	1
not	2
and	3
or	4

else - Creating Two Alternative Actions

In its present form the `if` statement allows us to perform an action when a given condition is met. But sometimes we need to perform an action only when the condition is not met. For example, when the user has to guess the number generated by the computer, we use an `if` statement to display the word "Correct" when the user guesses the number correctly:

```
if guess = number
   Print("Correct")
endif
```

However, shouldn't we display an alternative message when the player is wrong? One way to do this is to follow the first `if` statement with another testing the opposite condition:

```
if guess = dice
   Print("Correct")
endif

if not guess = dice
   Print("Wrong")
endif
```

We could also have written

```
if guess <> dice
```

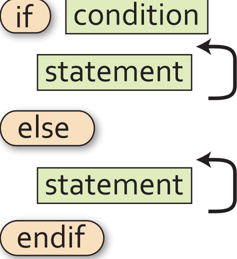
Although this will work, it's not very efficient since we always have to test both conditions - and the second condition can't be true if the first one is! As an alternative, we can add the word `else` to our original `if` statement and follow this by the action we wish to have carried out when the stated condition is false:

```
if guess = dice
   Print("Correct")
else
   Print("Wrong")
endif
```

This gives us the longer version of the `if` statement format as shown in FIG-4.8.

FIG-4.8

if ..else..endif



Note that we can have an unlimited number of statements between `else` and `endif`.

> **Activity 4.11**
>
> In your *Dice* program, modify the existing `if` statement to match the version given above so that either "Correct" or "Wrong" is displayed. Remove the code to calculate the difference between the *dice* and *guess* values.
>
> Test and save your program.

Activity 4.12

Start a new project called *TwoNumbers*.

Make use of the button input files to read in two integer values and then display the smaller of the two numbers. Also display a message indicating whether this smaller value is an odd or even number.

The program should use the following logic:

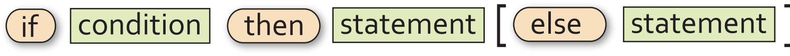
```
Display a prompt message for first number
Read the first number
Display a prompt message for the second number
Read the second number
IF first number is less than the second number THEN
        Set answer to first number
ELSE
        Set answer to second number
ENDIF
Display answer
IF answer is an even number THEN
        Display "Even"
ELSE
        Display "Odd"
ENDIF
```

The Other if Statement

AGK BASIC actually offers a second version of the `if` statement which has the format shown in FIG-4.9.

FIG-4.9

if..then..else



As with the previous `if` statement, the `else` section is optional but this version uses the word `then` and omits the `endif` term. Also, as the syntax diagram shows, you are restricted to a single statement after the `then` and `else` terms.

A major restriction when using this version of the `if` statement is that the `else` section of the statement must appear on the same line of the screen as the rest of the statement.

This means that the code you added in Activity 4.10 would have to be written as:

```
if dice = guess then Print("Correct") else Print("Wrong")
```

This lack of indented layout is enough to have the hardened programmer throw up her hands in horror!

Even when a single statement within the `if` statement is sufficient for the logic being coded, it is probably best to avoid this version of the `if` statement, since the requirement to place the `if` and `else` terms on the same line does not allow a good layout for the program code.

Activity 4.13

a) What is a Boolean expression?
b) How many relational operators are there?
c) If a condition contains `and`, `or` and `not` operators, which will be performed first?

Summary

➢ Conditional statements are created using the `if` statement.

➢ A Boolean expression is one which gives a result of either true or false.

➢ Conditions linked by the `and` operator must all be true for the overall result to be true.

➢ Only one of the conditions linked by the `or` operator needs to be true for the overall result to be true.

➢ When the `not` operation is applied to a condition, it reverses the overall result.

➢ The statements following a condition are only executed if that condition is true.

➢ Statements following the term `else` are only executed if the condition is false.

➢ A second version of the `if` statement is available in AGK BASIC in which `if` and `else` must appear on the same line.

Multi-Way Selection

Introduction

A single `if` statement is fine if all we want to do is perform one of two alternative actions, but what if we need to perform one action from three or more possible actions? How can we create code to deal with such a situation?

In structured English we use a modified IF statement of the form:

```
IF
    condition 1:
        action1
    condition 2:
        action 2
ELSE
        action 3
ENDIF
```

However, this structure is not available in AGK BASIC and hence we must find some other way to implement multi-way selection.

Nested if Statements

There are two main ways of achieving multi-way selection in AGK BASIC. One is to use nested `if` statements - where one `if` statement is placed within another. For example, let's assume in the *Dice* project that we want to display one of three messages: *Correct*, *Your guess is too high*, or *Your guess is too low*. Our previous solution allowed for two alternative messages, *Correct* or *Wrong,* and was coded as:

```
if guess = dice
   Print("Correct")
else
   Print("Wrong")
endif
```

In this new problem the `Print("Wrong")` statement needs to be replaced by the two alternatives, *Your guess is too high* or *Your guess is too low.* But we already know how to deal with two alternatives - use an `if` statement. The `if` statement for this situation would be:

```
if guess > dice
   Print("Your guess is too high")
else
   Print("Your guess is too low")
endif
```

If we now remove the `Print ("Wrong")` statement from our earlier code and substitute the four lines given above, we get:

```
if guess = dice
   Print("Correct")
else
   if guess > dice
      Print("Your guess is too high")
   else
      Print("Your guess is too low")
   endif
endif
```

We have created a nested `if` situation, where the `if guess > dice` statement is inside the `else` section of the `if guess = dice` statement.

Activity 4.14

Modify your *Dice* project so that the game will respond with one of three messages as shown in the code given above.

Test and save your program.

Activity 4.15

Start a new project called *Number*.

The program should generate a random number in the range -12 to +12.

The program should now display one of the following messages: *Negative* (if the number is less than zero), *Zero* (if the number is zero), or *Positive* (if the number is greater than zero). Finally, the value of the number should also be displayed.

Test and save your program.

There is no limit to the number of `if` statements that can be nested. Hence, if we required four alternative actions, we might use three nested `if` statements, while four nested `if` statements could handle five alternative actions. To demonstrate this we'll take our number guessing game a stage further and have it display one of five possible messages:

Your guess is too high	(if the guess is more than 2 above the dice)
Your guess is slightly too high	(if the guess is no more than 2 above the dice)
Correct	(if the guess equals the dice)
Your guess is slightly too low	(if the guess is no more than 2 below the dice)
Your guess is too low	(if the guess is more than 2 below the dice)

Activity 4.16

Reload *Dice*.

Modify the code so that it displays one of the five messages given above under the appropriate conditions. (HINT: You'll have to calculate the difference between the *guess* and *dice* values again.)

Test and save your program.

Mutually exclusive conditions refers to a set of conditions where no more than one of those conditions can be true at the same time.

When we have a set of mutually exclusive conditions, as in the *Dice* example given above, following the standard layout of indenting within an `if` statement results in the layout shown below:

```
if diff > 2
   Print("Your guess is too low")
else
   if diff > 0
      Print("Your guess is slightly too low")
   else
      if diff = 0
```

```
            Print("Correct")
         else
            if diff >= -2
               Print("Your guess is slightly too high")
            else
               Print("Your guess is too high")
            endif
         endif
      endif
   endif
```

In a situation that included even more options, the indentation can be so extreme that you may reach the right-hand margin! To solve this problem we often re-arrange the layout of nested `if` statements to be

```
if diff > 2
   Print("Your guess is too low")
else if diff > 0
   Print("Your guess is slightly too low")
else if diff = 0
   Print("Correct")
else if diff >= -2
   Print("Your guess is slightly too high")
else
   Print("Your guess is too high")
endif endif endif endif
```

with each option given the same indention as the last, and with the closing set of `endif` keywords placed on a single line. This gives a much neater layout which is still easy to follow.

> **Activity 4.17**
>
> Modify the layout of your *Dice* program to conform to this new layout style for multi-way selection. Resave your project.

elseif

The only problem with the previous solution is the need for so many `endif` terms at the end of the selection process. To avoid this we can replace the separate `else if` terms with the single word `elseif`. When we do this, only a single `endif` term is required at the end of the structure:

```
if diff > 2
   Print("Your guess is too low")
elseif diff > 0
   Print("Your guess is slightly too low")
elseif diff = 0
   Print("Correct")
elseif diff >= -2
   Print("Your guess is slightly too high")
else
   Print("Your guess is too high")
endif
```

> **Activity 4.18**
>
> Modify *Dice* to use the `elseif` term. Resave your project.

The select Statement

An alternative, and often clearer, way to deal with choosing one action from many is to employ the `select` statement. The simplest way to explain the operation of the `select` statement is simply to give you an example.

In the code snippet given below we display the name of the day of week corresponding to the number generated. For example, 1 results in the word *Sunday* being displayed.

```
day = Random(0,8)
select day
   case 1:
      Print("Sunday")
   endcase
   case 2:
      Print("Monday")
   endcase
   case 3:
      Print("Tuesday")
   endcase
   case 4:
      Print("Wednesday")
   endcase
   case 5:
      Print("Thursday")
   endcase
   case 6:
      Print("Friday")
   endcase
   case 7:
      Print("Saturday")
   endcase
endselect
Print(day)
Sync()
```

Once a value for *day* has been generated, the `select` statement chooses the `case` statement that matches that value and executes the code given within that section. All other `case` statements are ignored and any instructions following the `endselect` statement are executed. For example, if *day* = 3, then the statement given beside `case 3` will be executed (i.e. `Print("Tuesday")`) and the remainder of the whole `select.. endselect` structure ignored with the next statement executed being `Print(day)`. If *day* were to be assigned a value not given in any of the `case` statements (e.g. 0 or 8), the whole `select` statement would be ignored and no part of it executed and the next statement to be executed would be `Print(day)`.

Optionally, a special `case` statement can be added just before the `endselect` keyword. This is the `case default` option which is used to catch all other values which have not been mentioned in previous `case` statements. For example, if we modified our `select` statement above to end with the code

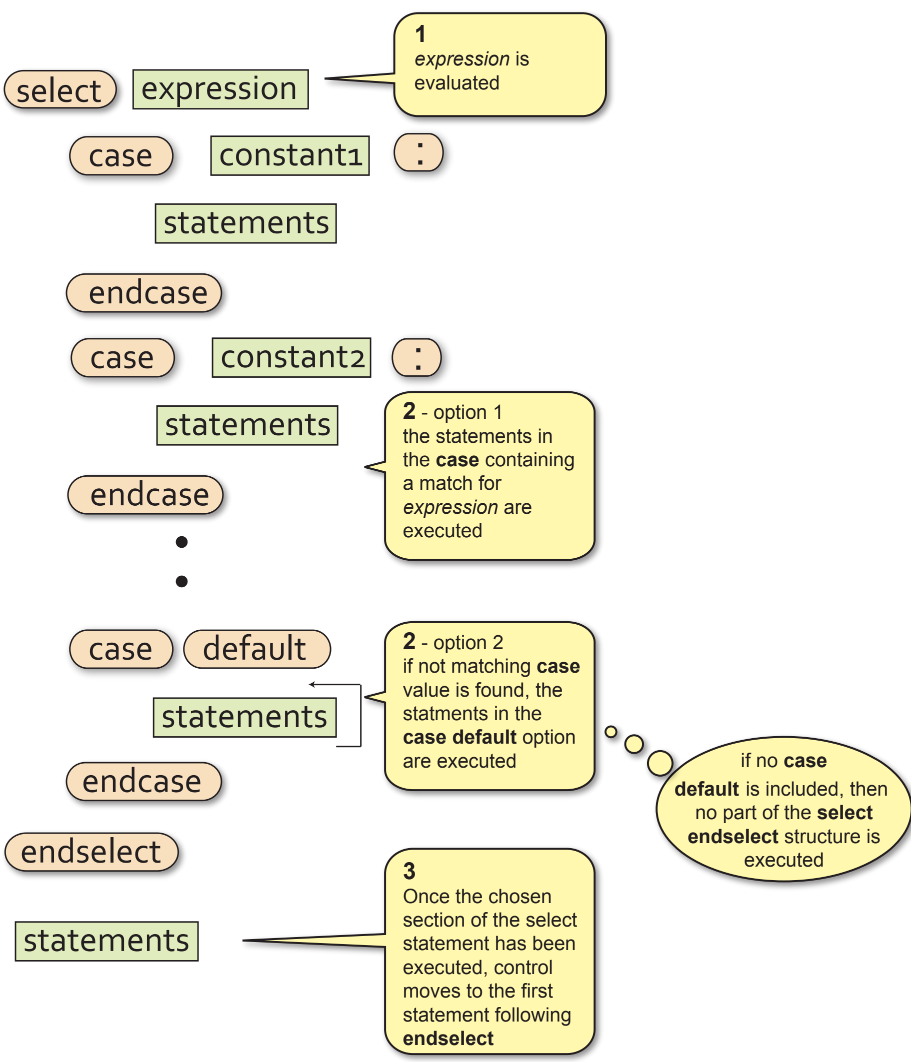
```
   case 7:
      Print("Saturday")
   endcase
   case default
      Print("Invalid day")
   endcase
endselect
```

then, if a value outside the range 1 to 7 is generated, the statement in the `case default` option will be executed.

FIG-4.10 shows how the `select` statement is executed.

FIG-4.10

How select Works



Several values can be specified for each `case` option. If the value of the term given in the `select` statement matches any of the values listed in a `case` statement, then the statement(s) in that `case` option will be executed. For example, using the lines

```
num = Random(1,10)
select num
   case 1,3,5,7,9:
      Print("Odd")
   endcase
   case 2,4,6,8,10:
      Print("Even")
   endcase
endselect
print(num)
Sync()
```

the word *Odd* would be displayed if any odd number between 1 and 9 was entered.

The values given beside the `case` keyword may also be a string as in the example below:

GetName() is assumed to be a user-written function that allows the player to enter their name.

```
name$ = GetName()
select name$
   case "Jack","Jill" :
      Print("Hello friend")
   endcase
   case default
      Print("I do not know your name")
   endcase
endselect
Sync()
```

Although the `case` value may also be a real value as in the line

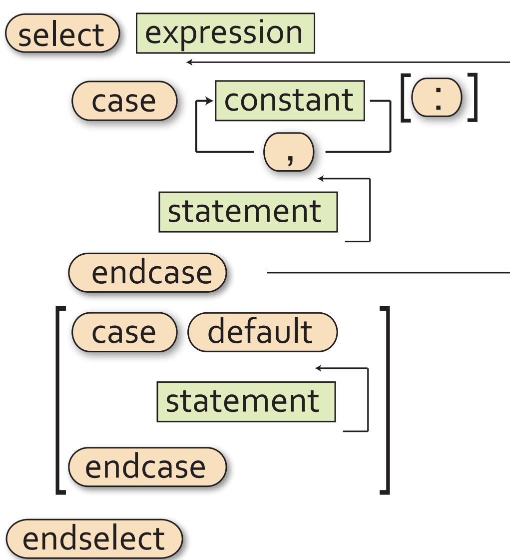
```
CASE 1.52
```

it is a bad idea to use these since the machine cannot store real values accurately. If a real variable contained the value 1.52000001 it would not match with the `case` value given above.

The general format of the `select` statement is given in FIG-4.11.

FIG-4.11

select..endselect



where:

expression is a variable or expression which reduces to a single integer, real or string value.

value is a constant of any type (integer, real or string).

statement is any valid AGK BASIC statement (even another `select` statement!).

Activity 4.19

Start a new project, *Days*.

The program should generate a random number in the range 0 to 8 and display the corresponding day of the week if the number is in the range 1 to 7. For any other value, the message *Invalid day* should be displayed.

Test and save your program.

Activity 4.20

Start a new project, *Cards*.

Generate a random number in the range 1 to 13 (the number represents the value of a playing card - 11, 12 and 13 being the Jack, Queen and King).

The program should display the message *Court card* if 11, 12, or 13 is generated and *Spot card* for all other values.

Test and run your program.

Not all multi-way selection situations can be coded using the select..endselect statement. For example, let’s say a number can be in the range 1 to 1000 and we want to perform specific actions for each of the groupings 1 to 200, 201 to 400, 401 to 600, 601 to 1000 then, since it would be impractical to list all the possible values for each group in a `case` line, we would have to code such a problem using nested `if` statements.

Testing Selective Code

When a program contains one or more `if` structures, our test strategy has to change to cope with this. For every `if` statement within a program we need to create at least two test values: one which results in the condition within the `if` statement being true, the other in the condition being false. Therefore, if a program contained the lines

```
no = GetButtonEntry()
if no mod 2 = 0
   Print("This is an even number")
endif
```

then we need to have a test value for *no* which is even and another which is odd. For example, we could choose the values 10 and 3.

This also applies to *less than or equal to* and *greater than or equal to* operators.

Another important test for conditions involving *less than*, or *greater than* operators is to find out what happens when the variable’s value is exactly equal to the value against which it is being tested. For example, if a program contained the lines

```
if result < 0
   Print("Negative")
else
   Print("Positive")
endif
```

then we would want to include zero as one of our test values, giving us three test

values: one less than zero, zero, and one greater than zero. So we could use, say, -7, 0 and 8.

Some of our projects don't allow for user input - instead they use randomly generated values. So we have no control over what values will be used when the program is run!

For test purposes, in a situation like this, we can modify the program's code temporarily so we can control the value used. Hence, in our *Numbers* project, for example, we could change the line

```
no = Random(-12,12)
```

to

```
no = -7
```

Now we can run the program and see if we get the expected result.

In the next two runs of the program we would change the assignment line to 0 and then 8 to get our other two test values. Once we have satisfied ourselves that the expected results have been obtained then we must restore the original code line to the program allowing the value of *no* to be generated randomly once more.

When an `if` statement contains more than one condition linked with `and` or `or` operators, testing needs to check each possible combination of true and false settings. For example, if a program contained the line

```
if dice1 = 6 and dice2 = 6
```

then our tests should include all possible combinations of true and false for the two conditions. A possible set of values is shown in FIG-4.10.

FIG-4.10

Test Data and Condition Results

dice1	dice2	Result
3	5	false, false
1	6	false, true
6	4	true , false
6	6	true , true

In a complex condition it is sometimes not possible to create every theoretical combination of true and false. For example, if a program contains the line

```
if total = 7 or total = 11 or dice1 = dice2
```

then the combinations of true and false for the three conditions are shown in FIG-4.11.

FIG-4.11

Three Condition Permutations

total=7	total=11	dice1=dice2
false	false	false
false	false	true
false	true	false
false	true	true
true	false	false
true	false	true
true	true	false
true	true	true

But the last two combinations in the table are impossible to achieve since *total* cannot

contain the values 7 and 11 at the same time (the conditions are mutually exclusive). So our test data will have test values which create only the remaining 6 combinations.

Activity 4.21

Suggest a set of test values for the latest version of the *Dice* project (Activity 4.17).

How would we have to modify the program's code in order to use these test values?

Summary

- The term **nested if statements** refers to the construct where one or more `if` statements are placed within the structure of another `if` statement.
- Multi-way selection can be achieved using nested `if` or by using the `select` statement.
- The `select` statement can be based on integer, real or string values.
- The `case` line can have any number of values, each separated by a comma.
- The `case default` option is executed when the value being searched for matches none of those given in the CASE statements.
- Testing a simple `if` statement should ensure that both true and false results are tested.
- Where a specific value is mentioned in a condition (as in no < 0) , that value should be part of the test data.
- When a condition contains `and` or `or` operators, every possible combination of results should be tested.
- Nested `if` statements should be tested by ensuring that every possible path through the structure is executed by the combination of test data.
- `select` structures should be tested by using every value specified in the `case` statements.
- `select` should also be tested using a value that does not appear in any of the `case` statements.

Solutions

Activity 4.1

a) Valid.
b) Valid.
c) Valid.
d) Invalid. => is not a relational operator (should be >=)
e) Invalid. Integer variable compared with string.
f) Invalid. 14 High Street should be in double quotes.

Activity 4.2

a) False. Only the second string contains a space.
b) True. “def”is shorter and matches the first three characters of “defg”.
c) True. “A” comes before “B”.
d) False. Only the second string contains a full stop.
e) False. Only the second string contains a capital D.
f) True. “*” has a greater ASCII coding than “&”

Activity 4.3

Program code:

```
rem *** include Buttons code ***
#include "Buttons.agc"
rem *** Setup the buttons for input ***
SetUpButtons()
rem *** Get the first value ***
Print("Enter first value :")
Sync()
Sleep(2000)
no1 = GetButtonEntry()
rem *** Get the second value ***
Print("Enter second value : ")
Sync()
Sleep(2000)
no2 = GetButtonEntry()
rem *** if no remainder, display message ***
if no1 mod no2 = 0
    Print("Exactly divisible")
    Sync()
endif
do
loop
```

Activity 4.4

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message if guess is wrong ***
if guess <> dice
    Print("Wrong")
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.5

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message and difference ***
rem *** if guess is wrong ***
if guess <> dice
  PrintC("Wrong. You were out by ")
  difference = dice - guess
  Print(difference)
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

You may get a negative value displayed when the guess is greater than the random number generated.

Activity 4.6

Code for *TwoDice*:

```
rem *** Two dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** Check for a win ***
total = dice1 + dice2
if total = 7 or total = 11
    Print("You win!")
endif
rem *** Display dice values ***
PrintC("Value of dice 1 : ")
Print(dice1)
PrintC("Value of dice 2 : ")
Print(dice2)
Sync()
do
loop
```

Activity 4.7

Modified code for *TwoDice*:

```
rem *** Two dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** Check for a win ***
total = dice1 + dice2
if total = 7 or total = 11 or dice1 = dice2
    Print("You win!")
endif
rem *** Display dice values ***
PrintC("Value of dice 1 : ")
Print(dice1)
PrintC("Value of dice 2 : ")
Print(dice2)
Sync()
do
loop
```

Activity 4.8

Code for *ThreeDice*:

```
rem *** Three Dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
dice3 = Random(1,6)
rem *** IF any two dice match THEN ***
if dice1 = dice2 or dice1 = dice3 or dice2 = dice3
    Print("You win!")
endif
rem *** Display values ***
PrintC("dice 1: ")
Print(dice1)
PrintC("dice 2: ")
Print(dice2)
PrintC("dice 2: ")
Print(dice3)
Sync()
do
loop
```

Activity 4.9

a) IF player lands on "Go to Jail" OR player picks up a "Go to Jail" card OR player throws three doubles in a row THEN

b) IF 10,00 gold pieces collected OR princess freed AND dragon slayed THEN

c) IF (holding King of Spades OR holding Queen of Spades) AND Ace of Diamonds played THEN

Activity 4.10

```
dice1 <> dice2 and dice1 <> dice3 and dice2 <> dice3
```

Activity 4.11

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
if guess = dice
  Print("Correct")
else
  Print("Wrong")
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.12

Code for *TwoNumbers*

```
rem *** Smaller odd/even ***

rem *** include Buttons functions ***
#include "Buttons.agc"
rem *** Display buttons ***
SetUpButtons()
rem *** Get numbers ***
Print("Enter first number ")
Sync()
Sleep(2000)
no1 = GetButtonEntry()
Print("Enter second number ")
Sync()
Sleep(2000)
no2 = GetButtonEntry()
rem *** Determine smaller value ***
if no1 < no2
    answer = no1
else
    answer = no2
endif
rem *** Display smaller ***
PrintC("Smaller value is ")
Print(answer)
rem *** Determine if answer is odd or even ***
if answer mod 2 = 0
    Print("This is an even number")
else
    Print("This is an odd number")
endif
Sync()
do
loop
```

Activity 4.13

a) A Boolean expression is an expression whose result is either true or false.
b) Six. <, <=, >, >=, =, <>
c) `not` is performed first, `and` next and `or` last. This order will change if parentheses are used.

Activity 4.14

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
if guess = dice
    Print("Correct")
else
    if guess > dice
        Print("Your guess is too high")
    else
        Print("Your guess is too low")
    endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.15

Code for *Number*:

```
rem *** Random number between -12 and 12 ***

rem *** Generate number ****
no = Random(-12,12)
rem *** Display number's sign ***
```

```
if no < 0
    Print("Negative")
else
    if no = 0
        Print("Zero")
    else
        Print("Positive")
    endif
endif
rem *** Disply number ***
Print(no)
Sync()
do
loop
```

Activity 4.16

Modified code for *Dice*:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
diff = dice - guess
if diff > 2
    Print("Your guess is too low")
else
    if diff > 0
        Print("Your guess is slightly too low ")
    else
        if diff = 0
            Print("Correct")
        else
            if diff >= -2
                Print("Your guess is slightly too
                 high")
            else
                Print("Your guess is too high")
            endif
        endif
    endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.17

The multi-way selection section of *Dice*'s code should now be have the following layout:

```
if diff > 2
    Print("You guess is too low")
else if diff > 0
    Print("Your guess is slightly too low ")
else if diff = 0
    Print("Correct")
else if diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif endif endif endif
```

Activity 4.18

New new multi-way selection coding in *Dice* should now be:

```
if diff > 2
    Print("You guess is too low")
elseif diff > 0
    Print("Your guess is slightly too low ")
elseif diff = 0
    Print("Correct")
elseif diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif
```

Activity 4.19

Code for *Days*:

```
rem *** Display day of the week ***

rem *** Generate value ***
day = Random(0,8)

rem *** Display day of week ***
select day
     case 1:
       Print("Sunday")
     endcase
     case 2:
       Print("Monday")
     endcase
     case 3:
       Print("Tuesday")
     endcase
     case 4:
       Print("Wednesday")
     endcase
     case 5:
       Print("Thursday")
     endcase
     case 6:
       Print("Friday")
     endcase
     case 7:
       Print("Saturday")
     endcase
     case default
       Print("Invalid day")
     endcase
endselect
rem *** Display number generated ***
Print(day)
Sync()
do
loop
```

Activity 4.20

Code for *Cards*:

```
rem *** Cards ***

rem *** Generate card value ***
card = Random(1,13)

rem *** Display card type ***
select card
    case 11,12,13:
        Print("Court card")
    endcase
    case default
        Print("Spot card")
    endcase
endselect
Print(card)
Sync()
do
loop
```

Note that all of the spot cards can be handled in the `case default` option because there is no chance of an invalid value being used.

Activity 4.21

The test data needs to cover all the possible paths through the nested if statements. In doing this we will have tested each condition for both true and false options.

So possible values are

dice	guess	Expected results
8	2	Your guess is too low
5	4	Your guess is slightly too low
7	7	Correct
2	4	Your guess is slightly too high
3	8	Your guess is too high

In addition, we would expect the values of dice and guess to be displayed.

Since the dice values are randomly generated it would be impractical to use our test data. We can overcome this problem by setting the variable dice to a specific value rather than determining its value using `Random()`. Once testing is complete, the random assignment can be restored.