Using the Editor

CBasic's built in editor has many of the features of a standard text editor with the addition of ***syntax colouring***.

With syntax colouring, every statement or function that is a built-in component of CBasic is highlighted in a user selectable colour. This makes your code more readable, and makes it easier for you to find errors.

Open a new program by selecting **File > New** from the main menu bar and type in the following program statements:

```
openconsole
print "Hello World"
do
until inkey$ <> ""
closeconsole
end
```

Notice the blue colouring of the CBasic program statements. If you make a mistake and misspell a program statement, you will immediately know due to its lack of colour.

Now that we have some text in the editor we can explain some of the key commands the editor responds to.

Key	Action
<Page Up>	Moves one page up
<Page Down>	Moves one page down
Up Arrow	Moves one line up
Down Arrow	Moves one line down
<CTRL> + A	Selects all text
<CTRL> + C	Copies selected text to clipboard
<CTRL> + F	Opens the 'Find' dialog
<CTRL> + H	Opens the 'Find / Replace' dialog
<CTRL> + X	Cuts selected text and copies to clipboard
<CTRL> + V	Pastes text from clipboard at caret position
<HOME>	Moves the caret to the beginning of the current line
<END>	Moves the caret to the end of the current line
<CTRL><HOME>	Moves to the beginning of the document
<CTRL><END>	Moves to the end of the document
<Backspace>	Deletes the previous character before the caret
<Delete>	Deletes the next character after the caret
F4	Runs the current program
F5	Single steps through the current program after a STOP statement
F7	Continues execution of the current program after a STOP statement
<TAB>	Tabs selected text to the right by one tab position
<SHIFT><TAB>	Removes tabs from the beginning of each selected line (de-tab)

Selecting Text

To select text for copying and pasting, place the mouse cursor (I-Beam) over, or just to the left of the first character of the text to be selected.

Press the left mouse button down and **‘drag’** the cursor over the text to select it.

‘Dragging’ means to hold the left mouse button down while moving the mouse.

If any character keys are pressed when text is selected, the selected portion will be replaced with the new text. The selected portion can be deleted by pressing the **<Delete>** key.

Text may also be selected using the keyboard by holding the **SHIFT** key down, and pressing one of **<HOME>**, **<END>**, **<PAGEUP>**, **<PAGEDOWN>**, or the **Arrow** keys.

You can select text from the current caret position, to any position clicked with the left mouse button by holding down the **SHIFT** key.

Try out the editing methods on the little test progam you typed into the editor.

Running the Program

Assuming you still have the program as you typed in above, you can execute it by selecting **Build > Run** from the main menu bar.

You can also run the program by pressing the green **GO** button on the tool bar, or by pressing **F4**.

You should see a **Console Window** open with the message ‘Hello World’ in it.

To close this window just press any key.

Saving the Program

To save the current program to a **source code file** select **File > Save As** from the main menu bar.

This will allow you to give the program a meaningful name, and to select any existing folder in which you wish to save it.

The default file extension for a CBasic source code file is **.cba** and this should not be changed to anything else.

In the next section, we will begin exploring the Creative Basic language and learning the syntax of commands, statements and functions.

Introduction to Programming

Programming

What do we mean when we talk about writing a program for a computer?

A computer is just a box full of electronics with lights on the front. It will do nothing but sit there humming to itself. It needs a **'program'** (otherwise known as software) in order to do something useful.

So how do we write a program to make things happen?

A program consists of a number of **instructions**. Each instruction is usually written on a single line and tells the computer to perform a particular operation.

The **'instruction set'** of a programming language can be thought of as a toolkit. You will need to pick the right tool for each operation you require.

We could sit at the computer, open a word processor, and type some instructions. Ignore for a moment what we type, and assume we just save the file. This would create a text or document file, and the computer would be quite unable to run it as a program. What we actually needed, is an **'executable'** program, or **.exe** file.

For this, we need software that can process our written instructions into an executable program. This is what Creative Basic provides.

Creative Basic provides a nice editor in which to write the instructions, and a help facility to remind us how each instruction is to be used. This is known as the **programming environment or I**ntegrated **D**evelopment **E**nvironment (**IDE**).

Some Essentials

There are a number of important elements in every program. Whether you are writing a program yourself, or trying to understand one that someone else has written, you will need to understand these elements.

We will talk about just four initially. These are:

- **Variables**
- **Functions**
- **Statements**
- **Comments**

Variables

Variables are the quantities used in your program to represent values such as speed, price, diameter, etc.

A program will use the computer's working memory to store changing information during processing. This information is referred to as **DATA**.

An item of information stored in this way is known as a **Variable**.

It is a variable because the computer can alter the stored value whenever instructed by a program statement.

Variable Names

Each variable is given a name of your choice, and this can be any combination of letters and numbers (but no special characters other than Underscore are allowed).

Variable names must begin with a letter (or Underscore _).

They are not case sensitive, and names can be up to **30 characters long**.

You can write for example: **costprice**, **COSTPRICE**, or **CostPrice**.

All of these are acceptable names, and CBasic regards all of them as the same quantity.

CostPrice is probably the preferred form for readability.

Variables are an important part of every program. They represent the quantities you wish to process to obtain a desired result. You can have as many variables in a program as you wish.

Variable Type Definition

A variable can be specified as one of a number of types.

You tell CBasic what **type** you want your variable to be using the **DEF** statement. (DEF is short for 'Define').

For example ..

DEF Number **AS** FLOAT

This defines the variable 'Number' as a 'float' type, which will store decimal numbers.

CBasic also supports the colon **':'** as an alternative to the AS keyword.

Another example using the colon form is ..

DEF Num1, Num2 **:** INT

This defines both variables 'Num1' and 'Num2' as Integers, which will hold integer (whole) numbers.

Notice that you can define several variables in a single DEF statement.

A final example shows a variable defined to hold String data ..

DEF Name **AS** STRING

The variable 'Name' is declared as a String variable, and can hold information such as "Mr John Smith".

String variables store text, and get their name from the term 'a *string* of characters'.

Data Types

In most applications, you will encounter both numeric and alphabetic (string) data. Numeric Data will be either **Integer** or **Decimal** quantities.

Integer Variables

Here is a small program demonstrating the use of Integer variables. You can make use of your previously saved console skeleton program to test this out. Alternatively, just type the statements into the CBasic editor window.

```
openconsole
def  i, j, total as int
i = 5
j = 3
total = i + j
print "The total is: ", total
print "Press Any Key to Continue"
do:until inkey$ <> ""
closeconsole
end
```

Variable 'i' is set to the integer value '5', and variable 'j' with the value '3'
The variable 'total' is then calculated as the sum of 'i' and 'j'
The 'print' command prints the result stored in variable 'total', which will be '8'

You can use descriptive text to explain the quantities that are shown on the screen.

Notice that the calculation of 'total' must be placed after the statements defining 'i' and 'j'.

All Variables, should be set (initialised) on the left-hand side of a statement before they are used.

Integer addition, subtraction and multiplication are straightforward, but division always needs care. Any fractional part resulting from division will be discarded.

For example ..

Div = i / j (ie. 5 / 3) if used in the above code, would result in Div = 1

(5 / 3 is actually 1.66666666, but the decimal part is lost since an integer variable cannot hold decimal values).

Note: All CBasic numeric variables are converted internally into Double precision decimals while calculations are performed, and the result is then assigned based on the data type of the result variable. This ensures maximum accuracy.

If you require an integer result rounded up to the nearest integer, use the CEIL() function. To round down, use the FLOOR() function as described in the Help - Users Guide.

To round the result to the nearest integer simply add 0.5 to the division. In the above example ..

Div = i / j + 0.5 will result in 2.16666666, which will be truncated to Div = 2

Decimal Variables

Now let us examine another important type of variable - the decimal or **'float'** type.

The strange name goes back in history as the 'floating-point' type - meaning that the decimal point can be moved to represent very large, or very small numbers.

We will try another example program. Either close the existing program, and re-open the skeleton console program - or delete the test statements for the previous example.

Enter these lines following the 'openconsole' statement (or modify the test lines of the previous program) to read as follows:

```
openconsole
def  x,y,product as float
' setprecision 8
X = 5
Y = 3.303
product = x * y
print "The product is:  ", product
do:until inkey$ <> ""
closeconsole
end
```

When you run this program, you get the answer 16.51, not the 16.515 you were expecting.

We have defined the three variables as type **float**, so they are pre-defined to hold decimal numbers. 4 bytes (32 bits) of storage are allocated for variables of this type, and as a result, numbers can only accurately represented to **8 significant figures**.

Decimal numbers are tricky and need to be handled with care. They are not necessarily absolutely accurate because of the limited storage space allocated to each value.

So why do we get the answer 16.51?

Delete the quote at the beginning of the **'set precision'** line (this changes it from a comment line to an active statement), and run the program again.

This time you get the result 16.51499939. Oh dear! This looks even worse.

If the **'setprecision'** statement ' is not specified, **CBasic defaults to a precision of two decimal places for output.** This can be extended using the **'Setprecision'** statement to however many decimal places you require.

Internally of course, CBasic is always working to the highest accuracy the variable can handle. When we remove the comment, and so set the precision to 8 decimal places, we see the longer, but still inaccurate answer. However it is now apparent where the 16.51 answer came from.

The quantity after the second decimal place, is less than half a unit, so two decimal places, will display 16.51, and the remainder is discarded.

Unfortunately, even if you increase the 'setprecision' statement value to 15 decimal places, you will still not get the correct answer 16.515.

You are encountering a disturbing feature of decimal numbers called **'precision'**.

In our example, the float variable can only accurately represent 16.514999. Any remaining digits are not meaningful.

Now set the precision back to 8 places, and edit the variable definition line to read ..

```
def x, y, product  as  double
```

Run the program again - the result is now 16.51500000. At last, we have the correct answer.

Why is this? We have now defined the working variables as type **'double'** precision.

Double precision variables use 8 bytes of storage (64 bits). Consequently, this type is accurate to **16 significant figures**.

Finally, change the 'setprecision' statement value to 15 decimal places again, and run the program. You will see that now there is a digit 1 at the end of the result.

The 16 significant figures limit has again been reached, so the final digit is again not meaningful.

Although this loss of precision might seem alarming, for a **'float'** type, taking the first 8 significant figures means the most you can be in error is 1 in 100 million - not too bad at all.

Normally, most mathematical calculations involving decimals will give accurate answers if you use **'double'** precision variables. However, if you need to carry out a sequence of, high-precision calculations on a variable, always be on the lookout for accumulating loss of precision.

In most practical calculations, the 'float' type of variable will give satisfactory accuracy.

String Variables

String variables store text and get their name from the term "a string of characters".
Here is a small snippet of code to illustrate string processing ..

```
def s,t as string
s = "Here are 3 percentage signs - "
t = "%%%"
print s + t
```

Note that strings are enclosed in double quotes. (ie. "This is a string")

When we print the result in the example, the second string value follows the first. Why is this?

The print instruction appears to be adding two strings - in fact, this operation is called "concatenation". It simply appends one string to another.

String variables can hold up to **254 ASCII** characters.

CBasic also has another string type known as an **ISTRING**.

This is an advanced STRING type, that can be defined and accessed like an array.

An ISTRING can be used anywhere a normal string can be used.

The maximum length of an ISTRING is 65535 characters.

Here is an example using an istring variable ..

```
openconsole

def name[29] as istring

name = "Donald Duck"

print name

do:until inkey$ <> ""
closeconsole
```

Defined in this example as name[29], the variable ‘name’ can hold up to 30 characters.

This is because istring variables are similar to 'arrays', and are **zero-based**.

We will leave discussing arrays until later.

You can use the console skeleton program to try the above example out. The name will print as 'Donald Duck'.

Why have two string data types? Obviously if you need to work with long strings, an ‘istring’ type will be needed.

If you are writing small strings to a Binary file (such as football team names), ‘istring’ variables can write smaller records than ‘string’ variables - which use fixed length 255 byte records, however few characters are in use. The reason is that in a binary file, all data is written in raw form, and will occupy the full number of bytes allocated to the variable type.

Otherwise, the string type is very commonly used for most string variables..

Using Strings for User Input

In console-based programs such as our previous examples, you can display a message to the user, wait until he types some data and presses ‘enter’, and then use the information.

```
openconsole

def name$ as string

print "Type your name and press ENTER."
input name$
print "Hello ",name$

do:until inkey$ <> ""
closeconsole
```

Notice the variable ‘name$’ ends in the character ‘$’. This is not actually necessary when using a string variable - it has a historical significance - when simply appending ‘$’ to a variable name, made it a string variable. Nowadays, it is simply a useful aid to recognising string variables in your code.

Use the ‘$’ if you find it helps - otherwise, just use your DEF statement to define string variables.