

THE BASIC ELITE

A Cemetech Production - Cemetech.tk

The Elite Guide to TI-BASIC

Table of Contents

Introduction.....	2
What You Will Need.....	2
How to Use this Guide.....	3
Idioms.....	4
Invisible 'Done'.....	4
Booleans.....	4
DelVar.....	5
Scrollable Lists.....	5
Correct Usage of Pause.....	5
Pause Idioms.....	6
Difficulty Setting with Menu(.....	6
Hidden Run Indicator.....	6
Dynamic List Size Counter.....	6
MirageOS and getKey(.....	7
General Tips and Tricks.....	8
Big Text.....	8
Pixel Stroking.....	8
Closing Punctuation.....	8
Multicommand Lines.....	8
Centering Text.....	9
Homescreen Text Specs.....	9
Graphscreen Text Specs.....	9
Text on Grids.....	10
Using Ans.....	10
Variables to Avoid.....	10
Text Characters to Use and Avoid.....	11
MirageOS and Doors CS Tags.....	11
Consistent Variables.....	11
getKey Scancodes.....	12
Return and Stop.....	13
IS and DS.....	13
XORing.....	13
Friendly Windows.....	13
Indicator Lists.....	14
Setting the rand Seed.....	15

Advanced Programming Strategies.....	16
Avoiding Memory Leaks.....	16
Program Groups.....	17
Trees.....	18
Consistent User Interface.....	18
Optimizing with TI-OS Functions.....	19
Accessing Hidden Characters.....	19
Objects in TI-BASIC.....	19
About The BASIC Elite.....	21

Staff

Co-Editors-in-Chief.....

	Kerm Martian
	Jon Pezzino

Introduction

If you are reading this, you are probably a budding young TI-BASIC programmer looking for some inside tips on how to master your favorite language. There is something about TI-BASIC that draws programmers in like flies to honey. I can tell you that I valued the fact that it can be programmed in a medium that is readily available during school, a time period when one's mind is brimming with ideas waiting to be realized in an ingenious program. The language is relatively simple and easy to learn for first time programmers because its syntax is remotely similar to English. Graphics are extremely easy to use in TI-BASIC, even for the novice programmer.

That having been said, let me say this: TI-BASIC is a weak language on a low-performance platform. It is in no way speedy, efficient, convenient, special, well-documented, or particularly useful outside the scope of mathematical programs. Algorithms that would be considered totally acceptable in C++/Java or any other high-level programming language simply do not work due to speed or memory limitations on the z80-based TI Calculators, of which there are many. There is a stifling lack of support for advanced data structures and objects, something many programmers would be unable to live without. I personally would never program in TI-BASIC were it not for the fact that it is an excellent diversion for the mathematically-oriented intellectual during class time. The situation, however, is not hopeless.

This guide aims to document and reveal many of the henceforth unknown or little-known features of TI-BASIC,

provide strategies for overcoming processor speed and memory limitations, and sketch out overall good programming strategies. You will learn to do things you would never have thought possible in TI-BASIC and amaze your friends with your sheer programming skill. It is my hope that you will find this an enlightening read that will educate you not only in the technicalities of TI-BASIC, but instill on you, the reader, a sense of good programming style and encourage you to continue and expand your programming career.

What You Will Need

This guide assumes you already have a solid knowledge of TI-BASIC and are a fairly competent programmer in general. Knowledge of computer languages such as C++/Java is a plus. As the author, I will be writing from the perspective of someone who is familiar with these languages and as such it would be beneficial for the reader to be in a similar mindset so as to fully comprehend the ideas presented.

This guide also assumes that you own a TI-8x series calculator. These tips have only been tested on a TI-83+SE and TI-84+SE, though I am 100% certain that these will work on a TI-83+ and slightly less confident that they will work on the TI-83 series or lower.

How To Use This Guide

This guide is not intended to be a tutorial and by no means a complete guide to TI-BASIC. Ideally, you will read through this guide and reference it while writing your programs. This guide is organized into 3 sections: Idioms, General Tips & Tricks, and Advanced Programming Strategies.

Section 1: Idioms

An idiom is a piece of code that programmers use over and over again, from program to program. Idioms may be speed up your program, make your code more understandable, or just impose arbitrary standards on your programming style. Idioms also provide you with a library of commonly used functions so that you do not have to figure out the best way to perform basic functions from scratch. Make it a habit to utilize idioms in your program.

Invisible 'Done'

The annoying and unprofessional looking 'Done' that appears once your program has ended can easily be eliminated by inserting the following code at the very end of your program:

```
:ClrHome  
:Output(1,1, ''
```

This program has to do with the fact that output remains after the program has executed and thus the null character (note that there is not a space after the quotation mark) overwrites the Done.

Booleans

One interesting aspect of the TI-OS is that boolean values are represented using 0 and 1 rather than true and false as is the usual case. There are a variety of conventions that utilize this aspect of TI-BASIC. Here are two of the most common:

Conditional Increment without If

Variables can be easily incremented without the use of separate If and assignment statements. Take a peek at the following code segment where 'A' is incremented by 'I' when <condition> is satisfied.

```
:A+I(<condition>→A
```

Remember that <condition> requires one of the test or logical operators (=, >, <, or, and, etc.). One common use of this idiom is when the condition is dependent on a getKey call and A is a variable representing a coordinate of object that is moved around using the arrow keys. This convention is much more elegant than multiple if statements (comparable to a++ in higher-level languages), not to mention that it saves a byte or two by eliminating an If and newline in favor of a parenthese.

Never Use '=' for Tests of Boolean Values

You will often write programs that use variables whose values will strictly be 1 or 0 (used as indicators/booleans/flags/etc.). Remember that you can

save precious bytes of memory by eliminating the use of '=' or '!=' (not equal). To test if boolean A is true replace this:

```
:If A=1
```

with this:

```
:If A
```

The same can be applied for a false test:

```
:If A=0
```

turns into:

```
:If not(A
```

This may seem trivial in terms of memory saved, but saving two bytes per conditional test can really add up to big savings in memory in large programs. In addition, If A is true regardless of the value of A, as long as it is positive and nonzero.

Favor DelVar over '0--><var>'

This is not an extremely important tactic, but it can help if your program is constantly resetting values to 0. DelVar saves a byte when several vars are deleted at once because multiple DelVars can be placed on one line with no special treatment:

```
:DelVar A DelVar B DelVar C
```

Surprisingly enough, the above is legitimate code and its example should be followed when appropriate. You can also place any other command on the same line as DelVar to save even more memory:

```
:DelVar A Disp B
```

Note that you cannot use DelVar on elements of a list or matrix.

Create Scrollable Lists with Ease

If you ever need to display a list to the user, you would ideally output the values on the graph screen on a nice little table. However, if the list is short or not incredibly important, or you are just slapping a program together in a minimal amount of time, you can easily create a scrollable list with this piece of code, which displays L1 in a scrollable format:

```
:Pause L1
```

Use Pause <var> Effectively

Yet another handy use of Pause is its ability to display a value as well as pause at the same time. For instance turn code like this:

```
:Disp A
```

```
:Pause
```

Into this:

```
:Pause A
```

Note that only one string or variable can be displayed at once using this method; no commas to separate arguments allowed!

A Simple Pause Idiom

This is a simple idiom that will pause your program until the user presses any key on the calculator, as opposed to 'Pause' which will only respond to ENTER:

```
:Repeat getKey  
:End
```

Set Difficulty Levels using Menu()

There is a very neat idiom for setting the difficulty level of your game using Menu(. (Assume your difficulty is represented by 'L.')

```
:1→L  
:Menu(''SET  
DIFFICULTY'', 'EASY'', E, 'MEDIUM'', M, 'HARD'', H  
:Lb1 H  
:L+5→L  
:Lb1 M  
:L+5→L  
:Lb1 E  
:<rest of program>
```

As you can see, this is much better than manually setting the level and then skipping to a convergent label for all levels because it avoids having to use Goto; the harder the level the higher up the program 'starts' and just keeps going down the hierarchy.

Hidden Run Indicator

You can make your measly BASIC program look like a high-quality ASM program by eliminating the normally inaccessible runtime indicator in the upper-right corner of the screen with this line of code:

```
:Text(-1,0,90,' '<space>
```

This trick utilizes the large font to overwrite an area of the screen that is normally reserved by the TI-OS.

No Counter Needed for Dynamically-Sized Lists

One feature of many higher level languages is the ability to use data structures known as array lists or vectors. These are essentially identical to the Lists provided by the TI-OS, except that they are dynamically sized and thus elements can be added without worrying about what index to add them at. With a little bit of know-how, these data structures can be mimicked in TI-BASIC. If you wish to add variable 'A' to list LIST,

which is of indeterminate size, the following call will eliminate the need to keep track of LIST's size using a separate variable:

```
:A→LLIST(1+dim(LLIST
```

This idiom takes advantage of the fact that the size of a list can be increased simply by adding an element at its next nonexistent index, which is automatically created by TI-OS when such a call is made.

MirageOS and getKey

One annoying quirk of MirageOS when running BASIC programs is its tendency to register 2nd or Enter (whichever was pressed to run the program) as the first getKey in the BASIC program (something to do with key buffers, I suspect.) Make sure that you clear the getKey buffer with a getKey command after your mirage tag if your program depends on detecting 2nd or Enter through the use of getKey.

```
::'MY PROGRAM  
:getKey
```

Section 2: General Tips and Tricks

This section will cover some good syntactical tips and tricks as well as some of the subtle yet simple dos and don'ts of TI-BASIC.

Big Text On the Graph Screen (TI-83+/84+ only)

This trick allows you to output home screen-sized font to the graph screen in the same manner that you can output small text on the graph screen. All you need to do is provide a -1 in the text argument before the row/column arguments:

```
:Text(-1,<row>,<column>,<string or value>
```

This call will function like any other text call, so feel free to append multiple strings or values to the argument as you normally would. Just remember that the text will be much bigger than normal and so you will not be able to fit as many letters per pixel as with small text.

Change the Pixel Stroke

This is an undocumented feature to draw a thicker pixel with the Pt-On command:

```
:Pt-On(X,Y,n
```

When n is 1, the Pt-On command acts as usual. When n is 2, an open box with its center at (X,Y) is drawn. When n is 3, a 3x3 cross is drawn with its center at (X,Y).

(Almost) Never Use Closing Parentheses or Quotes

This is a fairly nit-picky memory-saving trick, but as mentioned before, every byte counts in TI-BASIC. As a rule, you almost never need closing parentheses or quotes, such as in the following:

```
:A(B-C)
```

Should really be:

```
:A(B-C
```

There are some exceptions to this rule, however, such as when such expressions are in a list of arguments of another expression or function

```
:Text(Ø,Ø,L1(A),<string>
```

If the bolded parentheses were not there, the TI-OS would attempt to parse <string> as an argument for the index of L1, causing an error.

Never Put Multiple Commands on One Line Separated by Colons

Clever programmers may have noticed that if you put a colon after a command or statement, you can fit multiple commands on one line. DO NOT DO THIS! Multiple commands on one line will

make your program difficult to read and modify. Furthermore, using such a convention will make it more likely that you will accidentally delete multiple lines of your code with one wayward press of the Clear button. You will lose far more code using colons than newlines if you accidentally press Clear because Clear ignores colons, deleting all code in its path until it finds a newline. One of the most common error that programmers make is with If <condition>:Then statements:

```
:If <condition>:Then
```

This is a no-no. Instead simply put the 'Then' on a new line:

```
:If <condition>
```

```
:Then
```

Centering Text Quickly and Easily

Centering the text on your title screen can often mean the difference between your program looking like that of a sloppy amateur or a professional TI-BASIC guru. Luckily, centering text is INCREDIBLY easy if you just keep the following method in mind:

1)Count the number of letters in the string you want to center.

2)If you are displaying on the home screen, keep in mind that there are 16 characters per row, so simply subtract the result from #1 from 16, divide by two, and add that many spaces to your string. (A similar strategy can be applied to using Output (instead of Disp.)

3)If you are displaying on the graph screen, remember that each character is 6 pixels high by 3 across (except spaces) and that there are 95x63 pixels available to display on. Apply a similar strategy as mentioned above.

Screen Display Specifications

It is often handy to know the specifications of the home screen and graph screen.

> Home Screen

- 8 rows by 16 columns
- Output(<row>,<column>,<var>) (row and column start at 1)

> Graph screen

- 96x64 pixels
- 95x63 pixels accessible to programmer
- Text(<row>,<column>,<var>[,<var 2>...<var n>])
(row and column start at 0, so max argument for row is 94 and max argument for column is 62)

Specifications: Text on the Graph Screen

Each character in normal font on the graph screen is 6 pixels high (including the space at top) and 3 pixels across. Each character in the large font (`Text(-1,<row>,<column>,<text>)`) is 7 pixels high and 5 pixels across.

Displaying Text on Grids

Have you ever tried to display text on a grid on the graph screen? Did you ever notice that sometimes letters appear to overwrite grid lines for no reason whatsoever, while next time you run the program, it may not happen at all? Well, don't take chances that this random error will plague your program: eliminate the risk of this aesthetic nightmare by redrawing surrounding grid lines every time you output text to the graph screen. Here is an example from a chess game where a rook is being drawn (assume each grid space is 5x5 pixels):

```
:Text(1,1,"R
:Line(0,0,5,0
:Line(0,-5,5,-5
```

As you can see, the grid lines above and below the space in question are redrawn just in case the error decides to strike. I am not sure what exactly what causes this error to occur, but I can tell you that you will be happy when you do not have to worry about it!

Using the Ans Variable

Many programmers in TI-BASIC find it useful to write 'functions' or programs that take a parameter in the form of one variable and modify it so that the parent program can use that variable as if it had modified it itself. By default, all functions should use the Ans variable as their argument because it makes calling functions easier:

```
:X
:prgmSOMEFUNC
:Disp Ans
```

In this segment, the programmer does not have to worry about which variable to store so that that SOMEFUNC works properly; he simply declares the expression he wants the function to perform on and reaps the rewards in Ans form as well.

Avoid using X, Y, and sometimes n, T, and θ

Whenever the graph screen is accessed, certain variables are accessed and modified by the TI-OS (depending on the graphing mode). These variables are X, Y, n, T, and θ . Avoid using these variables for any sort of data other than a temporary counter when only the home screen is being used. Also note that n

cannot hold negative values because it is used as an index for sequences, which only use positive integers. In addition, Y will often default to 0 when you first use the graphscreen regardless of what you might have set it to; especially do not use it in this circumstance.

Favor Capital Letters over Lowercase

Although the idea of using proper grammar in your program may seem attractive, the memory costs of doing so can be high. Lowercase characters occupy twice the memory of uppercase characters. Think about this: in a text-intensive program, the size of your program could literally double if you use lowercase! That is a sacrifice that can rarely be made except:

- a) in text editors
- b) in programs with a minimum number of menus or overall text
- c) on title screens
- d) in MirageOS tags.

Think long and hard before deciding to use lowercase characters in your program!

Utilize Mirage and Doors CS Tags

In case you didn't know, you can make your TI-BASIC program compatible with MirageOS by changing the first line of the program to:

```
::' <program name/description>
```

You should always take advantage of this technique: it will improve the popularity of your program because it will be able to be accessed along with peoples' other games rather than through the TI-OS and it will be accessible from the Archive, meaning your customers will not lose their program when their RAM clears, meaning they will be able to pass it on to other calculator owners.

Even better, you can make your BASIC programs compatible with two of the major shells at once, Doors CS and MirageOS, by substitution the previous first line to this:

```
::DCS  
:0123456789ABCDEF
```

where 0123456789ABCDEF is a series of 16 hexadecimal numbers representing an 8x8 icon. If you do not know hexadecimal, simply use FF8181818181FF for a generic icon.

Choose Consistent Variables

Part of programming is programming consistently: making an overall plan and sticking to it each and every time. This task becomes much easier if you use the same variables for the same tasks over and over again. This way, you will never get confused about which variables to use for what functions

because you will have a template for such. This will minimize debugging time and make it easier to understand your code even if you have not looked for it in a long time. Below are some standard variables and their uses:

- I,J,θ,Z: For(Loops (J and Z are for nested loops)
- A,B,C: General temporary variables
- D,O: Can be used to represent a coordinate pair whenever it is appropriate to think of displaying in terms of 'Down' (D) and 'Over' (O), such as when using the Output (function or Text(function.
- R,C: Can be used to represent a row/column pair
- P,Q: Another utility or secondary pair
- K: a temporary getKey storage variable

You do not have to use this exact template for your programs; the important thing is to choose a template that makes sense to you and stick with it.

Know The getKey Scancodes

You can save yourself time and the unnecessary effort of referencing getKey codes if you remember this simple system: The first digit of the code is the row, the second in the column. Note that in row 10 the number becomes 10<column>. Here are the key codes for the arrow keys, which are probably the most commonly used getKey codes yet also the most confusing to figure out:

Left: 24
Up: 25
Right: 26
Down: 34

Distinguish Between Return and Stop

In functions you may sometimes find it useful to return to the parent program before all of the code has been executed, such as in the case of exceptions or algorithms/searches of indeterminate length. Remember that unless you have a truly exceptional condition, functions should almost never use Stop. Calling Stop will cause the entire program hierarchy to stop, not just the current program, whereas Return will return to the program that called the function in question. Utilizing Return will also allow the program to return to MirageOS and Doors CS rather than crash to the TI-OS.

Never use IS>(or DS>(

These two functions provided by TI-BASIC are essentially mistakes. They are confusing, rarely-used, and archaic. There is no proper or understandable documentation regarding their use, so attempting to use them will most likely cause results you might not expect. Favor other, more well-known idioms for conditional incrementing such as the one provided earlier in this guide.

Understanding xor

Many programmers are unaware of the use of the relative xor operator. Knowledge of this operator can save you many bytes of unnecessary and/or statements. The xor means 'either or'; it will evaluate to true only if one of the expressions is true and the other is false. One example of where xor can be used is in games where a piece or unit can be moved one space forward, backward, or to the side but not diagonally:

```
: 'P and Q are the coordinates of the move whose legality  
is being evaluated, R and C are the coordinates of the  
current unit's space.  
: If abs(P-R)=1 xor abs(Q-C)=1  
: <move there>
```

Use A Friendly Window

The ideal graph screen settings will use a coordinate system that correspond to the pixel coordinate system (so that text and lines can be used similarly and without translation between pixels and points). Personally, I find that the best graph screen has the following settings:

```
Xmin: 0  
Xmax: 94  
Ymin: -62  
Ymax: 0
```

With these settings, all one needs to do to translate between points and pixels is juxtapose x and y and use an opposite sign in front of the y value. This will save valuable time and space that might otherwise be used translating the graphics context from points to pixels if both Text((which accepts pixel coordinates) and Line((which accepts point coordinates) commands are used.

Use 'Indicator Lists' to Check for Saved Data

One of the more difficult challenges in TI-BASIC is devising a method for loading saved data from previous instances of your program. The dilemma is this: implement auto-loading and assume that the user was savvy enough to run the installation program ahead of time so that they will not get an error when they run your program or add a Load option, which is awkward for the user because you will often your program to automatically load old data. I offer a third solution to the saved-data dilemma: so called 'Indicator Lists.' An Indicator List is a list of saved data whose existence the program can verify without causing an error if it does not. There are two ways to implement an Indicator List: the 'false initiation' method or the dim(check method. The false initiation method is quite simple. At the beginning of your program, store a dummy value (usually 0) to the first element of the list you are using to save data then check the dim of the list. If the dim is just 1 (from storing the dummy value), then you know this is the first time the user has run the program and there is no saved data. Otherwise, you know that saved data exists in the list and you may feel free to proceed without fear of crashing your program through the dreaded ERR: UNDEFINED because if the list does not exist, storing to its first element will instantiate it. Here is an example (Note that Lbl 1 would be the point where you know that saved data exists, otherwise the program continues on as if no saved data exists):

```
:0→LDATA(1
:If 1<dim(LDATA
:Goto 1
:<assume no saved data exists>
```

The only true disadvantage of the false initiation method is that you will need to offset the list indexes of your data by one because the dummy value will otherwise clear the first value of saved data in the list, but this can easily be overcome with a little bit of planning beforehand.

The dim(check method is equally easy to implement. Remember that another way to instantiate a list is to store a number to its dim. If the list already exists, doing so will not modify the data stored in it. Thus, you can safely store

the desired dimension to your list in the beginning of your program and then check to see if the first value is 0 (there is no saved data) or another value (there is saved data). Here is an example:

```
:10→dim(LDATA
:If 0≠LDATA(1
:Goto 1
:<assume no saved data exists>
```

There are a few more cons to the dim(check method, but you will want to use it if it is essential that you need to use the first value of your list data. The dim(check method will not work for lists of unknown size (unless you store 999 to the dim, the maximum size allowed) because you could potentially delete data if the saved list is longer than the dim you store. Another disadvantage of the dim(check method is if the first value of your list could potentially be zero. If such is the case, then the dim(check method will not work unless there is another value in the list which could never be zero were there saved data.

Setting the rand Seed

There is a handy undocumented trick to set the seed for the rand command:

```
:N→rand
```

where N is any real number. Remember that the same sequence of random numbers will be generated (using rand) if N is the same number. This trick can be utilized in encryption keys because every N will generate a unique sequence of random numbers.

Section 3: Advanced Programming Strategies

This section contains advanced strategies regarding programming in TI-BASIC. These are conceptual strategies rather than syntactical tricks; you must keep these tips in mind from the moment you start writing your program in order to utilize the to their fullest potential.

Avoid Memory Leaks

This might seem like a fairly subtle and unimportant programming strategy, but it is vital to a program with lots of iterations. Failure to follow this rule is the cause of so called 'program fatigue' where the program runs slower and slower as time goes on. Failure to follow this rule is also the cause of most of the mysterious 'ERROR:MEMORY' messages you might receive when your program is running.

The rule is based on the principle that every time that the TI-OS detects the beginning of a control structure (While, For(, Repeat, If <condition>:Then) or a program call (subprogram, subroutine, function, whatever you want to call it), it allocates memory space to remind it to search for the End that accompanies the beginning of these control structures. If, however, you exit the loop before its corresponding End has been reached, you will leave that little piece of memory dangling and doomed to clog your RAM until the program exits. The most common way to exit a loop is use of the misunderstood Goto statement (though I suppose horrendous use of the Menu (function could also qualify). An example of poor use of control structures is provided:

```
:While 1
:If getKey=45
:Goto 1
:End
:Lbl 1
```

Notice how the End that complements 'While 1' will never be reached? TI-OS doesn't know that and as such will wait for that End to be reached at some point in time, using up memory. The above example can and should be replaced with something that does not cause a memory leak:

```
:Ø→A
:While not(A
:If getKey=45
:1→A
:End
```

Note that this is not necessarily the best or most elegant solution, but it readily demonstrates the principle of memory-leakage through improper use of control structures.

This rule might also lead you to infer another rule: generally avoid the use of Goto and Lbl. These conventions are inefficient, confusing, obsolete, and ineffective the vast majority of the time and are usually avoided by skilled programmers in any language. Overuse of Goto will create so-called 'spaghetti' code, where reading your code becomes akin to untangling spaghetti because it is so difficult to constantly follow the path that the gotos will take you. Also note that Goto will slow down your program as a whole because the program interpreter has to scan the entire program to look for a label when it encounters Goto. This is much slower than other control structures, which the TI-OS is able to handle much more efficiently.

Another common cause of memory-leakage is a tangled program/subprogram hierarchy. Avoid recursive programs (programs that call themselves) because if they recur too much, they will gobble memory up like none other, causing the original program to crash. Also, make certain that programs have an established hierarchy so that when a subprogram is called, it is guaranteed to return to its parent program. For example, if you were to have a program whose purpose was to take in input, use subroutines to display the output, and start all over again, you would want to make certain that the parent program is handling everything, unlike in this diagram:

Parent Program -> Subprogram 1 -> Subprogram 2 ->

Subprogram 3 (displays answer) -> Parent Program...

As you can see in this diagram, Subprogram 3 does not ever call the original parent program, instead minting a new 'instance' of the parent program, dooming the original parent to needlessly gobble up memory. Instead, the diagram should look like this:

Parent Program -> Subprogram 1 -> Subprogram 2 ->

Subprogram 3 (stores answer, then returns, traveling back up the program hierarchy to Parent program)

Use TI-Groups as 'File types'

In your TI-BASIC programming career, you may occasionally write a program in which it is necessary to share data with other calculators. The best way to do this by far is the use of groups. Simply create a group of all data files the program will use and send that group to your friend's calculator. This will save you the effort of having to remember which files you need to send to share data, it will archive your data and thus save it in the event of a RAM clearance, and keeps everything together in one neat package. It also makes it easy to 'open' a file: simply ungroup. Obviously, BASIC programs cannot modify groups, so you will have to re-create the group to change data,

much the same as you would have to load a file into your computer's RAM to modify before writing it to your hard drive.

Iterate through Strings and Use them to Store Data

Often, hard coding rules or superfluous frills (such as names or symbols) can be a tedious task. Rather than go through the hassle of using multiple If statements to define symbols, characters, or names, why not simply store them into a string and access its substrings? This approach is much more friendly when it comes time to change your program; Say you decide you want to represent the player as an 'O' rather than an 'X' or change an enemy from '*' to 'M'. This code is an example of such. The player is represented as an 'X' and there are three enemies, *, V, & M.

```
: 'X*VM → Str1
```

```
: 'Later in the program...
```

```
: Output(<player's Y-coord>, <player's X-coord>, sub(Str1, 1, 1
```

And an example of storing names and recalling through iteration:

```
: 'BILLYJOE FRED BOB JANE → Str1
```

```
: Prompt A
```

```
: Disp sub(Str1, 5A, 1
```

If you cannot figure out what this code does, try putting it in your calculator and entering a number 1-5 when prompted for A. (Notice how every 'friend' has a 5-character-long name, even if it has to be made that long with spaces.)

Trees

Keep in mind that a tree can be stored in an array (list) using the following algorithm:

Node X

Parent Node: Node X / 2

Children: Left: Node 2X, Right: Node 2X + 1

Of course, it is difficult to sort such a tree (or balance, heap/reheap, etc.), but it is very easy to search (big-0 log n) and to add new items such that the tree remains a binary search tree. Consider using a tree in your next data-intensive program. (Note: this is not an in-depth explanation of trees; however, I feel this will only benefit people with prior knowledge of trees and as such don't see fit to explain them here.)

Use a consistent User Interface Format

A consistent UI will make your program much easier to use because the user will not have to figure out what the program is doing each time something new comes up: they will be able to pick up on the pattern of formats and tell on their own. For instance, always use an understandable and consistent format

with the Input function. For example, if you decide the below format is how you will prompt for numbers, then always use this format when prompting for numbers.

```
:Input 'How Many? ',A
```

Here is an example for a good menu format:

```
:Disp '      HELLO','-----
```

Notice how the text is centered and the underlining goes across the entire screen.

Use Functions Provided by the TI-OS

The TI-OS provides many useful math functions. Oftentimes you will find yourself writing a long and complicated algorithm only to discover that there is a fast, efficient, pre-made function provided by the TI-OS to handle what you are trying to do. For example, you might be trying to calculate the angle of a complex number. Such an algorithm written in TI-BASIC would take approximately 30 bytes and cause considerable speed problems. This situation could easily be remedied with a simple call of the angle(function, saving you time and memory and speeding your program up significantly. Get familiar with the TI 'API' so that you know when to use a pre-made function and when to write your own.

Utilizing Inaccessible Characters

There are a number of characters which you cannot enter on the calculator. Since most BASIC programs use text instead of sprites, a wider of selection of characters to represent objects can greatly improve the aesthetics of your program.

(See

<http://www.ticalc.org/archives/files/fileinfo/350/35050.html>

for an excellent example of this. Notice the player's character and the boss character.) Because you cannot directly type these characters into your program, you should develop a strategy for accessing these characters. One such strategy could be to create a string of all the inaccessible characters on your computer and send it to your calculator, then recall the string into a program that will restore the string when run. Then, when you need an inaccessible character, simply run the above program, and recall the string while editing the program in question requiring the character.

Objects in TI-BASIC

One of the main qualms programmers used to higher level languages (especially Java) will have with BASIC is lack of support for objects. However, you can simulate objects fairly well if you know what you are doing using complex numbers and

lists. A complex number has the following properties (also listed are their equivalents in a higher level language):

- Positivity/Negativity of real number (boolean)
- Integer part of real number (unsigned int)
- Decimal part of real number (unsigned int)
- Positivity/negativity of imaginary number (boolean)
- Integer part of imaginary number (unsigned int)
- Decimal part of imaginary number (unsigned int)

As you can see, one complex number can hold quite a bit of data (six instance fields, to be precise). Now, considering that you are programming on a calculator running at 6 MHz, that is quite a bit of flexibility and power if you know how to unlock it using the `real()`, `imag()`, `iPart()`, and `fracPart()` commands. Think of how many classes you can emulate with six instance fields; there are quite a few! For example, you could have an enemy spaceship that has an x and y coordinate (determined by Integer & Decimal parts of real number), a directional heading (Integer part of imaginary), a health amount (Decimal part of imaginary), a boss indicator (positivity/negativity of real), and some other indicator (positivity/negativity of imaginary). One could create quite a complex and memory-efficient game following this plan. Unfortunately, this approach is not very fast at all. It takes quite a bit of work to compress/decompress a complex number, making this implementation of objects impractical for games that are in real time. There is however, a solution that sacrifices memory-efficiency for more flexibility and speed.

The other way you can implement objects is through parallel lists. In this approach, you create a series of lists, each one representing one instance field. For example, you might create LX, LY, LDIR, and LHEA if you were trying to create a game similar to the one described above. An object is represented by the values of the lists at a specific index. For example, the object with value one would have an x value of LX(1), a y value of LY(1), etc. As you can tell, this approach is much more memory-hogging, but it is worth it if you are more concerned about speed than memory-efficiency. Another advantage of this approach is that you are not limited to the number of instance fields you can have or their type. You simply create a new list for every variable you want to have.

About The BASIC Elite

This guide is brought to you by the Cemetech™ BASIC Elite. The BASIC Elite is an independent project overseen by Cemetech, the goal of which is to promote the production of superior TI-BASIC programs. The BASIC Elite publishes a bi-monthly newsletter consisting of a variety of programming-oriented columns, including reviews of top BASIC programs, spotlights on promising new programmers, and new tips and tricks, along with relevant articles discussing issues in the TI-BASIC community. The BASIC Elite Newsletter is published on ticalc.org by Jonathan Pezzino and Kerm Martian as a 'Miscellaneous Informational Text.' You can also subscribe to our mailing list by visiting the Cemetech Forum and requesting to be added in the BASIC Elite forum under project. Visit the Cemetech website for news on the latest BASIC Elite developments at <http://www.cemetech.net>. You can also contact the authors of this guide through their email, jon_p@sbcglobal.net (Jon) or Kerm.Martian@yahoo.com (Kerm).