beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 24, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Common Problems

```
Common Problems
***************

   If you are new to programming or the Amiga E language then you might
appreciate some help locating problems (or bugs) in your programs.  This
Appendix details some of the most common mistakes people make.


  Assignment and Copying
  Pointers and Memory Allocation
  String and List Misuse
  Initialising Data
  Freeing Resources
  Pointers and Dereferencing
  Mathematics Functions
  Signed and Unsigned Values
```

1.2 Assignment and Copying

```
Assignment and Copying
======================

   This is probably the most common problem encountered by people who are
used to languages like BASIC.  Strings, lists, arrays and objects cannot
be initialised using an assignment statement: data must be copied.  Unlike
BASIC, this kind of data is represented by a pointer (see PTR Type), so
only the pointer would be copied by an assignment statement, not the data
it points to.  The following examples all copy a pointer rather than the
data, and so the memory for the data is shared (and this is probably not
what was intended).

       DEF s[30]:STRING, t[30]:STRING,
           l[10]:LIST, m[10]:LIST,
           x:myobj, y:myobj,
           a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

     /* You probably don't want to do any of these */
       s:='Some text in a string'
       l:=[-6,4,-9]
       x:=[1,2,3]:myobj
       a:=[1,-3,8,7]:INT

       t:=s
       m:=l
       y:=x
       b:=a
```

All the declarations allocate memory for the appropriate data. The first four assignments replace the pointers to this memory with pointers to some statically allocated memory. The memory allocated by the declarations is probably now unreachable, because the only pointers to it have been over-written. BASIC programmers might expect, say, the assignment to s to have copied the string into the memory allocated for s by its declaration, but this is not the case (only the pointer to the string is copied).

For the E-string, s, and E-list, l, there is another, disastrous side-effect. The assignment to s, for example, means that s will point to a normal string, not an E-string. So, s can no longer be used with any of the E-string functions. The same considerations apply to the E-list, l, as well.

The final four assignments also copy only the pointers. This means that s and t will point to exactly the same memory. So they will represent exactly the same string, and any change to one of them (by a StrAdd, for example) will appear to change both (of course, only one lump of memory is being changed, but there are two references to it). This is called memory sharing, and is only a problem if you didn't intend to do it!

To get the result that a BASIC programmer might have intended you need to copy the appropriate data. For E-strings and E-lists the functions to use are, respectively, StrCopy and ListCopy. All other data must be copied using a function like CopyMem (an Amiga system function from the Exec library). (Normal strings can be copied using AstrCopy built-in function, see the 'Reference Manual'.) Here's the revised forms of the above assignments:

```
       DEF s[30]:STRING, t[30]:STRING,
           l[10]:LIST, m[10]:LIST,
           x:myobj, y:myobj,
           a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

       StrCopy(s, 'Some text in a string')  /* Defaults to ALL */
       ListCopy(l, [-6,4,-9])               /* Defaults to ALL */
       CopyMem([1,2,3]:myobj, x, SIZEOF myobj)
       CopyMem([1,-3,8,7]:INT, a, 4*SIZEOF INT)

       StrCopy(t, s)   /* Defaults to ALL */
       ListCopy(m, l)  /* Defaults to ALL */
       CopyMem(x, y, SIZEOF myobj)
       CopyMem(a, b, 4*SIZEOF INT)
```

Notice that you need to supply the size (in bytes) of the data being copied when you use CopyMem. The parameters are also given in a slightly different order to the E-string and E-list copying functions (i.e., the source must be the first parameter and the destination the second). The CopyMem function does a byte-by-byte copy, something like this:

```
     PROC copymem(src, dest, size)
       DEF i
       FOR i:=1 TO size DO dest[]++:=src[]++
     ENDPROC
```

Of course, you can use string constants and lists to give initialised arrays, but in this case you should be initialising an appropriately typed pointer. You must also be careful not to run into a static data problem (see Static data).

```
       DEF s:PTR TO CHAR, l:PTR TO LONG, x:PTR TO myobj, a:PTR TO INT
       s:='Some text in a string'
       l:=[-6,4,-9]
       x:=[1,2,3]:myobj
       a:=[1,-3,8,7]:INT
```

1.3 Pointers and Memory Allocation

Pointers and Memory Allocation
==============================

Another common error is to declare a pointer (usually a pointer to an object) and then use it without the memory for the target data being allocated.

```
     /* You don't want to do this */
       DEF p:PTR TO object
       p.element:=99
```

There are two ways of correcting this: either dynamically allocate the memory using NEW or, more simply, let an appropriate declaration allocate it. See Memory Allocation.

```
       DEF p:PTR TO object
       NEW p
       p.element:=99

       DEF p:object
       p.element:=99
```

1.4 String and List Misuse

String and List Misuse
======================

Some of the string functions can only be used with E-strings. Generally, these are the ones that might extend the string. If you use a normal string instead you can run into some serious (but subtle) problems. Commonly misused functions are ReadStr, MidStr and RightStr. Similar problems can arise by using a list when an E-list is required by a list function.

String constants and normal lists are static data, so you shouldn't try to alter their contents unless you know what you're doing (see Static data).

1.5 Initialising Data

Initialising Data
=================

Probably one of the most common mistakes that even seasoned programmers make is to forget to initialise variables (especially pointers). The rules in the 'Reference Manual' state which declarations initialise variables to zero values, but it is often wise to make even these explicit (using initialised declarations). Variable initialisation becomes even more important when using automatic exceptions.

1.6 Freeing Resources

Freeing Resources
=================

Unlike a Unix operating system, the Amiga operating system requires the programmer to release or free any resources used by a program. In practice, this means that all windows, screens, libraries, etc., that are successfully opened must be closed before the program terminates. Amiga E provides some help, though: the four most commonly used libraries (Dos, Exec, Graphics and Intuition) are opened before the start of an E program and closed at the end (or when CleanUp is called). Also, memory allocated using any of List, String, New, NEW, NewR, NewM and FastNew is automatically freed at the end of a program.

1.7 Pointers and Dereferencing

Pointers and Dereferencing
==========================

C programmers may think that the ^var and {var } expressions are the direct equivalent of C's &var and *var expressions. However, in E dereferencing is normally achieved using array and object element selection, and pointers to large amounts of data (like E-strings or objects) are made by declarations. This means that the ^var and {var } expressions are rarely used, whilst var[] is very common.

1.8 Mathematics Functions

Mathematics Functions
=====================

The standard mathematical operators / and * do not use full 32-bit values in their calculations, as noted previously (see Maths and logic functions). A common problem is to forget this and use them where the values will exceed the 16-bit limit. A typical example is the position calculations used with proportional gadgets. See Signed and Unsigned Values.

1.9 Signed and Unsigned Values

Signed and Unsigned Values
==========================

This is a quite advanced topic, but might be the cause of some strange bugs in your programs. Basically, E does not have a way of differentiating signed and unsigned values from, say, the LONG type. That is, all values from the 32-bit, LONG type are considered to be signed values, so the range of values is from -2,147,483,648 to 2,147,483,647. If the values from this type were taken to be unsigned then no negative values would be allowed but more positive values would be possible (i.e., the range of values would be from zero to 4,294,967,295). This distinction would also affect the mathematical operators.

In practice, though, it is not the LONG type that can cause problems. Instead, it is the 16-bit, INT type, which again is considered to be signed. This means that the range of values is -32,768 to 32,767. However, the Amiga system objects contain a number of 16-bit, INT elements which are actually interpreted as unsigned, ranging from zero to 65,535. A prominent example is the proportional gadget which forms a part of a scroll-bar on a window (for example, a drawer window on Workbench). This works with unsigned 16-bit values, which is at odds with the INT type in E. These values are commonly used in calculations to determine the position of something displayed in a window, and if the INT type is used without taking into account this signed/unsigned problem the results can be quite wrong. Luckily it is very simple to convert the signed INT values into unsigned values if they are part of some expression, since the value of any expression is taken from the LONG type (and unsigned INT values fit well within the range of even signed LONG values).

```
     PROC unsigned_int(x) IS x AND $FFFF
```

The function unsigned_int, above, is very specific to the way the Amiga handles values internally, so to understand how it works is beyond the scope of this Guide. It should be used wherever an unsigned 16-bit value is stored in an INT element of, say, an Amiga system object. For example, the position of the top of a (vertical) proportional gadget as a percentage (zero to one hundred) of its size can be calculated like this:

```
       /* propinfo is from the module 'intuition/intuition' */
       DEF gad:PTR TO propinfo, pct
       /* Set up gad... */
       /* Calculate percentage (MAXPOT is from 'intuition/intuition') */
       pct:=Div(Mul(100,unsigned_int(gad.vertpot)),MAXPOT)
```

Notice that the full 32-bit functions Div and Mul need to be used since the arithmetic may be well over the normal 16-bits used in the / and * operators.

The remaining type, CHAR, is not, in practice, a problem. It is the only unsigned type, with a range of values from zero to 255. There is a fairly simple way to convert these values to signed values (and again this is particular to the way the Amiga stores values internally). One good example of a signed CHAR value is the priority value associated with a node of an Amiga list (i.e., the pri element of an ln object from the module exec/nodes).

```
     PROC signed_char(x) IS IF x<128 THEN x ELSE x-256
```

1.10 Other Information

Other Information
****

This Appendix contains some useful, miscellaneous information.

1.11 Amiga E Versions

Amiga E Versions
================

As I write, the current version of Amiga E is version 3.2e (which is minor update of v3.2a). This edition of the Guide is based primarily on that version, but the majority still applies to the older versions, including the last Public Domain version (v2.1b). Version 3.3 is imminent, and the new 'Reference Manual' will contain details of the new features and changes.

Please note that, as of v3.0a, Amiga E is a commercial product so you must pay a fee to get a version of the full compiler (which will be registered to you). The Public Domain distribution contains only a demonstration version of the compiler, with limited functionality. See the 'Reference Manual' for more details.

1.12 Further Reading

Further Reading
===============

'Amiga E Language Reference'
: Referred to as the 'Reference Manual' in this Guide. This is one of the documents that comes with the Amiga E package, and is essential reading since it was written by Wouter (the author of Amiga E). It contains a lot of extra information.

'Rom Kernel Reference Manual' (Addison-Wesley)
: This is the official Commodore documentation on the Amiga system functions and is a must if you want to use these functions properly. At the time of writing the Third Edition is the most current and it covers the Amiga system functions up to Release 2 (i.e., AmigaDOS 2.04 and KickStart 37). Because there is so much information it comes in three separate volumes: 'Libraries', 'Includes and Autodocs', and 'Devices'. The 'Libraries' volume is probably the most useful as it contains many examples and a lot of tutorial material. However, the examples are written mainly in C (the remainder are in Assembly). To alleviate this problem I have undertaken to re-code them in E, and Part One and Part Two of this effort should be available from Aminet or good PD houses (the archive names will be something like JRH-RKRM-1 and JRH-RKRM-2). (Unfortunately, it seems that the actual manuals may be hard to find since there are now out-of-print.)

'The AmigaDOS Manual' (Bantam Books)
: This is the companion to the 'Rom Kernel Reference Manual' and is the official Commodore book on AmigaDOS (both the AmigaDOS programs and the DOS library functions). Again, the Third Edition is the most current.

Example sources
: Amiga E comes with a large collection of example programs. When you're familiar with the language you should be able to learn quite a bit from these. There are a lot of small, tutorial programs and a few large, complicated programs.

1.13 Amiga E Author

```
Amiga E Author
==============

   In case you didn't know the author and creator of Amiga E is Wouter van
Oortmerssen (or $#%!).  You can reach him by normal mail at the following
address:

     Wouter van Oortmerssen ($#%!)
     Levendaal 87
     2311 JG  Leiden
     HOLLAND

However, he much prefers to chat by E-mail, and you can reach him at the
following addresses:

     Wouter@alf.let.uva.nl (E-programming support)
     Wouter@mars.let.uva.nl (personal)
     Oortmers@gene.fwi.uva.nl (other)

Better still, if your problem or enquiry is of general interest to Amiga E
users you may find it useful joining the Amiga E mailing list.  Wouter
regularly contributes to this list and there are a number of good
programmers who are at hand to help or discuss problems.  To join send a
message to:

     amigae-request@bkhouse.cts.com

Once you're subscribed, you will receive a copy of each message mailed to
the list.  You will also receive a message telling you how you can
contribute (i.e., ask questions!).
```

1.14 Guide Author

```
Guide Author
============

   This Guide was written by Jason Hulance, with a lot of help and
guidance from Wouter.  The original aim was to produce something that
might be a useful introduction to Amiga E for beginners, so that the
language could (rightly) become more widespread.  The hidden agenda was to
free Wouter from such a task so that he could concentrate his efforts on
improving Amiga E.

   You can reach me by normal mail most easily at the following (work)
address:

     Jason R. Hulance
     Formal Systems (Europe) Ltd.
     3 Alfred Street
     Oxford
     OX1 4EH
```

ENGLAND

Alternatively, you can find me on the Amiga E mailing list, or E-mail me directly at one of the following addresses:

jason@fsel.com
m88jrh@ecs.oxford.ac.uk

If you have any changes or additions you'd like to see then I'd be very happy to consider them. Criticism of the text is also welcome, especially if you can suggest a better way of explaining things. I am also keen to hear from people who can highlight areas that are particularly confusing or badly worded!