beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 22, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Types

```
Types
*****
```

We've already met the LONG type and found that this was the normal type for variables (see Variable types). The types INT and LIST were also mentioned. Learning how to use types in an effective and readable way is very important. The type of a variable (as well as its name) can give clues to the reader about how or for what it is used. There are also more fundamental reasons for needing types, e.g., to logically group data using objects (see OBJECT Type).

This is a very large chapter and you might like to take it slowly. One of the most important things to get to grips with is pointers. Concentrate on trying to understand these as they play a large part in any kind of system programming.

1.2 LONG Type

```
LONG Type
=========
```

The LONG type is the most important type because it is the default type and by far the most common type. It can be used to store a variety of data, including memory addresses, as we shall see.

1.3 Default type

Default type

LONG is the default type of variables. It is a 32-bit type, meaning that 32-bits of memory (RAM) are used to store the data for each variable of this type and the data can take (integer) values in the range -2,147,483,648 to 2,147,483,647. Variables default to being LONG typed, but they can also be explicitly declared as LONG:

```
DEF x:LONG, y

PROC fred(p:LONG, q, r:LONG)
  DEF zed:LONG
  statements
ENDPROC
```

The global variable x, procedure parameters p and r, and local variable zed have all been declared to be LONG values. The declarations are very similar to the kinds we've seen before, except that the variables have :LONG after their name in the declaration. This is the way the type of a variable is given. Note that the global variable y and the procedure parameter q are also LONG, since they do not have a type specified and LONG is the default type for variables.

1.4 Memory addresses

Memory addresses

There's a very good reason why LONG is the normal type. A 32-bit (integer) value can be used as a memory address. Therefore we can store the address (or location) of data in a variable (the variable is then called a pointer). The variable would then not contain the value of the data but a way of finding the data. Once the data location is known the data can be read or even altered! The next section covers pointers and addresses in more detail. See PTR Type.

1.5 PTR Type

PTR Type
========

The PTR type is used to hold memory addresses. Variables which have a PTR type are called pointers (since they store memory addresses, as mentioned in the previous section). This section describes, in detail, addresses, pointers and the PTR type.

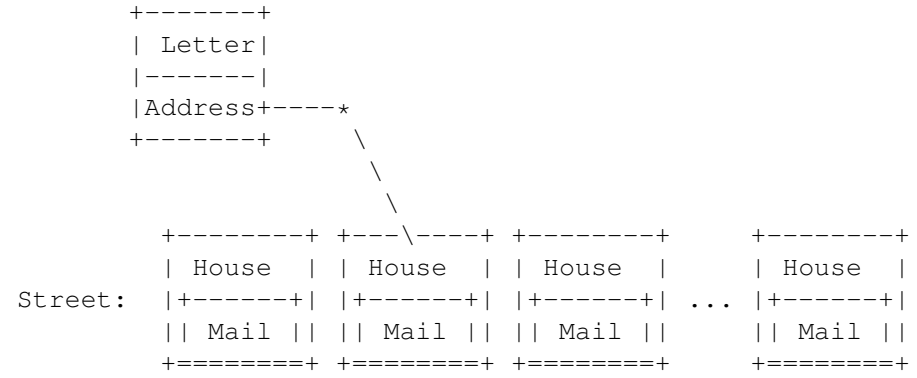
1.6 Addresses

Addresses

Every piece of data a program uses is stored somewhere in the computer's memory,and this includes the data contained in variables. So, when you assign one to the variable x you are actually storing one in the location reserved for x in the computer's memory. A location in memory is known as a memory address, and this is just a 32-bit number (so can be stored in a LONG variable). If you know the location of a variable's data then you can read the value and you can also change it.

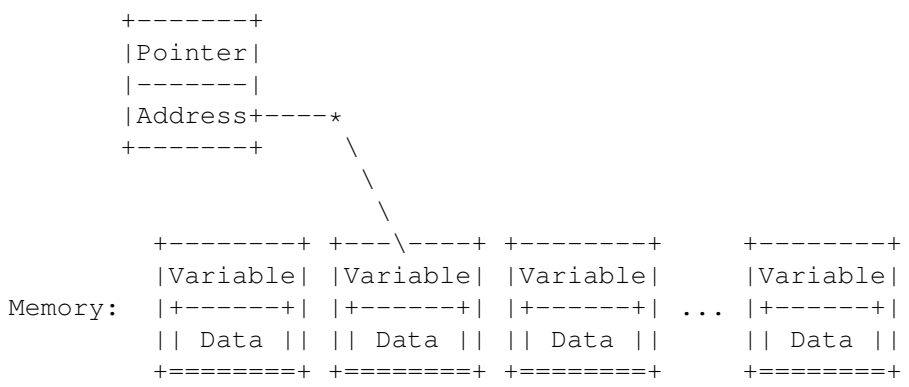
To understand memory addresses, a good analogy is to think of memory as a road or street, each memory location as a post-box on a house, and each piece of data as a letter. If you were a postman you would need to know where to put your letters, and this information is given by the address of the post-box. As time goes by, each post-box is filled with different letters. This is like the value in a memory location (or variable) changing. To change the letters stored in your post-box, you tell your friends your address and they can send letters in and fill it. This is like letting some part of a program change your data by giving it the address of the data.

The next two diagrams illustrate this analogy. A letter contains an address which points to a particular house (or lot of mail) on a street.



A pointer contains an address which points to a variable (or data) in

memory.



1.7 Pointers

```
Pointers
--------
```

Variables which contain memory addresses are called pointers. As we saw in the previous section, we can store memory addresses in LONG variables. However, we then don't know the type of the data stored at those addresses. If it is important (or useful) to know this then the PTR type (or, more accurately, one of the many PTR types) should be used.

```
    DEF p:PTR TO LONG, i:PTR TO INT,
        cptr:PTR TO CHAR, gptr:PTR TO gadget
```

The values stored in each of p, cptr, i and gptr are LONG since they are memory addresses. However, the data at the address stored in p is taken to be LONG (a 32-bit value), that at cptr is CHAR (an 8-bit value), that at i is INT (a 16-bit value), and that at gptr is gadget, which is an object (see OBJECT Type).

Since pointers are just data like any other LONG variable, the value of the pointer is somewhere in memory. This means it has an address, so you can have a pointer which is actually pointing to another pointer! This is one of the reasons pointers can be quite difficult to think about, and misunderstanding them is often the cause of big problems with programs.

1.8 Indirect types

```
Indirect types
--------------
```

In the previous example we saw INT and CHAR used as the destination types of pointers, and these are the 16- and 8-bit equivalents (respectively) of the LONG type. However, unlike LONG these types cannot be used directly to declare global or local variables, or procedure

parameters. They can only be used in constructing types (for instance with PTR TO). The following declarations are therefore illegal, and it might be nice to try compiling a little program with such a declaration, just to see the error message the E compiler gives.

```
/* This program fragment contains illegal declarations */
DEF c:CHAR, i:INT

/* This program fragment contains illegal declarations */
PROC fred(a:INT, b:CHAR)
  DEF x:INT
  statements
ENDPROC
```

This is not much of a limitation because you can store INT or CHAR values in LONG variables if you really need to. However, it does mean there's a nice, simple rule: every direct value in E is a 32-bit quantity, either a LONG or a pointer. In fact, LONG is actually short-hand for PTR TO CHAR, so you can use LONG values like they were actually PTR TO CHAR values.

1.9 Finding addresses (making pointers)

```
Finding addresses (making pointers)
-----------------------------------
```

If a program knows the address of a variable it can directly read or alter the value stored in the variable. To obtain the address of a simple variable you use { and } around the variable name. The address of non-simple variables (e.g., objects and arrays) can be found much more easily (see the appropriate section), and in fact you will very rarely need to use {var }. However, if you understand how to explicitly make pointers with {var } and use the pointers to get to data, then you'll understand the way pointers are used for the non-simple types much more quickly.

Addresses can be stored in a variable, passed to a procedure or whatever (they're just 32-bit values). Try out the following program:

```
DEF x

PROC main()
  fred(2)
ENDPROC

PROC fred(y)
  DEF z
  WriteF('x is at address \d\n', {x})
  WriteF('y is at address \d\n', {y})
  WriteF('z is at address \d\n', {z})
  WriteF('fred is at address \d\n', {fred})
ENDPROC
```

Notice that you can also find the address of a procedure using { and }.

This is the memory location of the code the procedure represents, although it is not something we need concern ourselves with any further in this Guide. Here's the output from one execution of this program (don't expect your output to be exactly the same, though):

```
x is at address 3758280
y is at address 3758264
z is at address 3758252
fred is at address 3732878
```

This is an interesting program to run several times under different circumstances. You should see that sometimes the numbers for the addresses change. Running the program when another is multi-tasking (and eating memory) should produce the best changes, whereas running it consecutively (in one CLI) should produce the smallest (if any) changes. This gives you a glimpse at the complex memory handling of the Amiga and the E compiler.

1.10 Extracting data (dereferencing pointers)

```
Extracting data (dereferencing pointers)
----------------------------------------
```

If you have an address stored in a variable (i.e., a pointer) you can extract the data using the ^ operator. This is called dereferencing the pointer. The ^ operator should only really be used when {var } has been used to obtain an address. To this end, LONG values are read and written when dereferencing pointers in this way. For pointers to non-simple types (e.g., objects and arrays), dereferencing is achieved in much more readable ways (see the appropriate section for details), and this operator is not used. In fact, ^var is seldom used in programs, but is useful for explaining how pointers work, especially in conjunction with {var }.

Using pointers can remove the scope restriction on local variables, i.e., they can be altered from outside the procedure for which they are local. Whilst this kind of use is not generally advised, it makes for a good example which shows the power of pointers. For example, the following program changes the value of the local variable x for the procedure fred from within the procedure barney. It can do this only because fred passes a pointer to x as a parameter to barney.

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG
  x:=33
  p:={x}
  barney(p)
  WriteF('x is now \d\n', x)
ENDPROC
```

```
PROC barney(ptr:PTR TO LONG)
  DEF val
  val:=^ptr
  ^ptr:=val-6
ENDPROC
```

Here’s what you can expect it to generate as output:

```
x is now 27
```

Notice that the ^ operator (i.e., dereferencing) is quite versatile. In the first assignment of the procedure barney it is used (with the pointer ptr) to get the value stored in the local variable x, and in the second it is used to change this variable’s value. In either case, dereferencing makes the pointer behave exactly as if you’d written the variable to which it points. To emphasise this, we can remove the barney procedure, like we did above (see Style Reuse and Readability):

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG, val
  x:=33
  p:={x}
  val:=x
  x:=val-6
  WriteF('x is now \d\n', x)
ENDPROC
```

Everywhere the barney procedure used ^ptr we’ve written x (because we are now in the procedure for which x is local). We’ve also eliminated the ptr variable (the parameter to the barney procedure), since it was only used with the ^ operator.

To make things clear the fred and barney example is deliberately ‘wordy’. The val and p variables are unnecessary, and the pointer types could be abbreviated to LONG or even omitted, for the reasons outlined above (see LONG Type). This is the compact form of the example:

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=33
  barney({x})
  WriteF('x is now \d\n', x)
ENDPROC

PROC barney(ptr)
  ^ptr:=^ptr-6
ENDPROC
```

By far the most common use of pointers is to address (or reference)

large structures of data. It would be extremely expensive (in terms of CPU time) to pass large amounts of data from procedure to procedure, so addresses to such data are passed instead (and, as we know, these are just 32-bit values). The Amiga system functions (such as ones for creating windows) require a lot of structured data, so if you plan to do any real programming you are going to have to understand and use pointers.

As we have seen, if you have a pointer to some data you can easily read the data, but you can just as easily alter it. If you want to write code that is clear and understandable, you have an implicit responsibility to not use pointers to alter data that you didn't ought to. For instance, if a procedure is passed a pointer which it then uses to change the data being pointed to, then it ought to be well documented (using comments) exactly what changes it makes.

1.11 Procedure parameters

```
Procedure parameters
--------------------
```

Only local and global variables have the luxury of a large choice of types. Procedure parameters can only be LONG or PTR TO type. This is not really a big limitation as we shall see in the later sections.

1.12 ARRAY Type

```
ARRAY Type
==========
```

Quite often, the data used by a program needs to be ordered in some way, primarily so that it can be accessed easily. E provides a way to achieve such simple ordering: the ARRAY type. This type (in its various forms) is common to most computer languages.

1.13 Tables of data

```
Tables of data
--------------
```

Data can be grouped together in many different ways, but probably the

most common and straight-forward way is to make a table. In a table the data is ordered either vertically or horizontally, but the important thing is the relative positioning of the elements. The E view of this kind of ordered data is the ARRAY type. An array is just a fixed sized collection of data in order. The size of an array is important and this is fixed when it is declared. The following illustrates array declarations:

```
DEF a[132]:ARRAY,
    table[21]:ARRAY OF LONG,
    ints[3]:ARRAY OF INT,
    objs[54]:ARRAY OF myobject
```

The size of the array is given in the square brackets ([and]). The type of the elements in the array defaults to CHAR, but this can be given explicitly using the OF keyword and the type name. However, only LONG, INT, CHAR and object types are allowed (LONG can hold pointer values so this isn't much of a limitation). Object types are described below (see OBJECT Type).

As mentioned above, procedure parameters cannot be arrays (see Procedure parameters). We will overcome this apparent limitation soon (see Array procedure parameters).

1.14 Accessing array data

Accessing array data

To access a particular element in an array you use square brackets again, this time specifying the index (or position) of the element you want. Indices start at zero for the first element of the array, one for the second element and, in general, (n-1) for the n-th element. This may seem strange at first, but it's the way most computer languages do it! We will see a reason why this makes sense soon (see Array pointers).

```
DEF a[10]:ARRAY

PROC main()
  DEF i
  FOR i:=0 TO 9
    a[i]:=i*i
  ENDFOR
  WriteF('The 7th element of the array a is \d\n', a[6])
  a[a[2]]:=10
  WriteF('The array is now:\n')
  FOR i:=0 TO 9
    WriteF(' a[\d] = \d\n', i, a[i])
  ENDFOR
ENDPROC
```

This should all seem very straight-forward although one of the lines looks a bit complicated. Try to work out what happens to the array after the assignment immediately following the first WriteF. In this assignment the

index comes from a value stored in the array itself! Be careful when doing complicated things like this, though: make sure you don't try to read data from or write data to elements beyond the end of the array. In our example there are only ten elements in the array a, so it wouldn't be sensible to talk about the eleventh element. The program could have checked that the value stored at a[2] was a number between zero and nine before trying to access that array element, but it wasn't necessary in this case. Here's the output this example should generate:

```
The 7th element of the array a is 36
The array is now:
 a[0] = 0
 a[1] = 1
 a[2] = 4
 a[3] = 9
 a[4] = 10
 a[5] = 25
 a[6] = 36
 a[7] = 49
 a[8] = 64
 a[9] = 81
```

If you do try to write to a non-existent array element strange things can happen. This may be practically unnoticeable (like corrupting some other data), but if you're really unlucky you might crash your computer. The moral is: stay within the bounds of the array.

A short-hand for the first element of an array (i.e., the one with an index of zero) is to omit the index and write only the square brackets. Therefore, a[] is the same as a[0].

1.15 Array pointers

```
Array pointers
--------------
```

When you declare an array, the address of the (beginning of the) array is given by the variable name without square brackets. Consider the following program:

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF ptr:PTR TO INT, i
  FOR i:=0 TO 9
    a[i]:=i
  ENDFOR
  ptr:=a
  ptr++
  ptr[]:=22
  FOR i:=0 TO 9
    WriteF('a[\d] is \d\n', i, a[i])
  ENDFOR
ENDPROC
```

Here's the output from it:

```
a[0] is 0
a[1] is 22
a[2] is 2
a[3] is 3
a[4] is 4
a[5] is 5
a[6] is 6
a[7] is 7
a[8] is 8
a[9] is 9
```

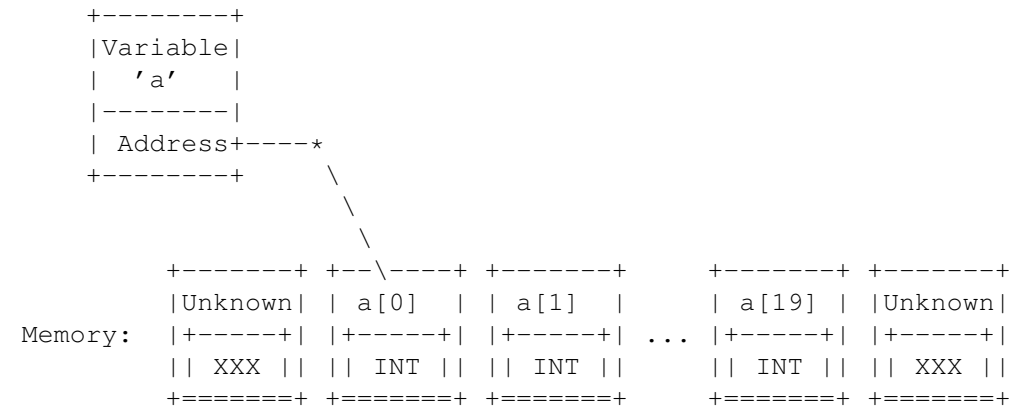
You should notice that the second element of the array has been changed using the pointer. The assignment to ptr initialises it to point to the start of the array a, and then the ptr++ statement increments ptr to point to the next element of the array. It is vital that ptr is declared as PTR TO INT since the array is an ARRAY OF INT. The [] is used to dereference ptr and therefore 22 is stored in the second element of the array. In fact, the ptr can be used in exactly the same way as an array, so ptr[1] would be the next (or third element) of the array a. Also, since ptr points to the second element of a, negative values may legitimately be used as the index, and ptr[-1] is the first element of a.

In fact, the following declarations are identical except the first reserves an appropriate amount of memory for the array whereas the second relies on you having done this somewhere else in the program.

```
DEF a[20]:ARRAY OF INT

DEF a:PTR TO INT
```

The following diagram is similar to the diagrams given earlier (see Addresses). It is an illustration of an array, a, which was declared to be an array of twenty INTs.



As you can see, the variable a is a pointer to the reserved chunk of memory which contains the array elements. Parts of memory that aren't between a[0] and a[19] are marked as 'Unknown' because they are not part of the array. This memory should therefore not be accessed using the array a.

1.16 Point to other elements

```
Point to other elements
-----------------------
```

We saw in the previous section how to increment a pointer so that it points to the next element in the array. Decrementing a pointer p (i.e., making it point to the previous element) is done in a similar way, using the p-- statement. Actually, p++ and p-- are really expressions which denote pointer values. p++ denotes the address stored in p before it is incremented, and p-- denotes the address after p is decremented. Therefore,

```
       addr:=p
       p++
```

does the same as

```
       addr:=p++
```

And

```
       p--
       addr:=p
```

does the same as

```
       addr:=p--
```

The reason why ++ and -- should be used to increment and decrement a pointer is that values from different types occupy different numbers of memory locations. In fact, a single memory location is a byte, and this is eight bits. Therefore, CHAR values occupy a single byte, whereas LONG values take up four bytes (32 bits). If p were a pointer to CHAR and it was pointing to an array (of CHAR) the p+1 memory location would contain the second element of the array (and p+2 the third, etc.). But if p were a pointer to an array of LONG the second element in the array would be at p+4 (and the third at p+8). The locations p, p+1, p+2 and p+3 all make up the LONG value at address p. Having to remember things like this is a pain, and it's a lot less readable than using ++ or --. However, you must remember to declare your pointer with the correct type in order for ++ and -- to work correctly.

1.17 Array procedure parameters

```
Array procedure parameters
--------------------------
```

Since we now know how to get the address of an array we can simulate

passing an array as a procedure parameter by passing the address of the array. For example, the following program uses a procedure to fill in the first x elements of an array with their index numbers.

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF i
  fillin(a, 10)
  FOR i:=0 TO 9
    WriteF('a[\d] is \d\n', i, a[i])
  ENDFOR
ENDPROC

PROC fillin(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[]:=i
    ptr++
  ENDFOR
ENDPROC
```

Here's the output it should generate:

```
a[0] is 0
a[1] is 1
a[2] is 2
a[3] is 3
a[4] is 4
a[5] is 5
a[6] is 6
a[7] is 7
a[8] is 8
a[9] is 9
```

The array a only has ten elements so we shouldn't fill in any more than the first ten elements. Therefore, in the example, the call to the procedure fillin should not have a bigger number than ten as the second parameter. Also, we could treat ptr more like an array (and not use ++), but in this case using ++ is slightly better since we are assigning to each element in turn. The alternative definition of fillin (without using ++) is:

```
PROC fillin2(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[i]:=i
  ENDFOR
ENDPROC
```

Also, yet another version of fillin uses the expression form of ++ in the assignment (see Assignments) and the horizontal form of the FOR loop to give a really compact definition.

```
PROC fillin3(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1 DO ptr[]++:=i
```

```
ENDPROC
```

1.18 OBJECT Type

```
OBJECT Type
===========
```

Objects are the E equivalent of C and Assembly structures, or Pascal records. They are like arrays except the elements are named not numbered, and they can be of different types. So, to find a particular element in an object you use a name instead of an index (number). Objects are also the basis of the OOP features of E (see Object Oriented E).

1.19 Example object

```
Example object
--------------
```

We'll dive straight in with this first example, and define an object and use it. Object definitions are global and must be made before any procedure definitions.

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  DEF a:rec
  a.tag:=1
  a.check:=a
  a.data:=a.tag+(10000*a.tag)
ENDPROC
```

This program doesn't visibly do anything so there isn't much point in compiling it. What it does do, however, is show how a typical object is defined and how elements of an object are selected.

The object being defined in the example is rec, and its elements are defined just like variable declarations (but without a DEF). There can be as many lines of element definitions as you like between the OBJECT and ENDOBJECT lines, and each line can contain any number of elements separated by commas. The elements of the rec object are tag and check (which are LONG), table (which is an array of CHAR with eight elements)

and data (which is also LONG). Every variable of rec object type will have space reserved for each of these elements. The declaration of the (local) variable a therefore reserves enough memory for one rec object.

1.20 Element selection and element types

```
Element selection and element types
-----------------------------------
```

To select elements in an object obj you use obj.name, where name is one of the element names. In the example, the tag element of the rec object a is selected by writing a.tag. The other elements are selected in a similar way.

Just like an array declaration the address of an object obj is stored in the variable obj, and any pointer of type PTR TO objectname can be used just like an object of type objectname. Therefore, in the previous example a is a PTR TO rec.

As the example object shows, the elements of an object can have several different types. In fact, the elements can have any type, including object, pointer to object and array of object. The following example shows how to access some different typed elements.

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

OBJECT bigrec
  data:PTR TO LONG
  subrec:PTR TO rec
  rectable[22]:ARRAY OF rec
ENDOBJECT

PROC main()
  DEF r:rec, b:bigrec, rt:PTR TO rec
  r.table[]:="H"
  b.subrec:=r
  b.subrec.tag:=1
  b.subrec.data:=r.tag+(10000*b.subrec.tag)
  b.subrec.table[1]:="i"
  b.rectable[0].data:=r.tag
  b.rectable[0].table[0]:="A"
  rt:=b.rectable
  rt[].data++:=0
  rt[].table[]--:="B"
ENDPROC
```

The ++ and -- operators apply to first thing in the selection (i.e., rt in both the last two assignments in the example above), and may occur only after all the selections. Notice that object selection and array indexing can be repeated as much as necessary (but only as the types of the

elements allow). As a simple example, consider the third assignment:

```
b.subrec.tag:=1
```

This selects the subrec element from the bigrec object b, and then sets the tag element of this rec object to 1. Now, consider one of the later assignments:

```
b.rectable[0].table[0]:="A"
```

This selects the rectable element from b, which is an array of rec objects. The first element of this array is selected, and then the table element of the rec object is selected. Finally, the first character of the table is set to the character A.

As you can probably tell, it is important to give the elements of objects appropriate types if you want to do multiple selection in this way. However, this is not always possible or the best way of doing some things, so there is a way of giving a different type to pointers (this is called explicit pointer typing--see the 'Reference Manual' for more details).

Here's a quite simple example which uses an array of objects:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  DEF a[10]:ARRAY OF rec, p:PTR TO rec, i
  p:=a
  FOR i:=0 TO 9
    a[i].tag:=i
    p.check++:=i
  ENDFOR
  FOR i:=0 TO 9
    IF a[i].tag<>a[i].check
      WriteF('Whoops, a[\d] went wrong...\n', i)
    ENDIF
  ENDFOR
ENDPROC
```

If you think about it for long enough you'll see that a[0].tag is the same as a.tag. That's because a is a pointer to the first element of the array, and the elements of the array are objects. Therefore, a is a pointer to an object (the first object in the array).

1.21 Amiga system objects

```
Amiga system objects
--------------------
```

There are many different Amiga system objects. For instance, there's

one which contains the information needed to make a gadget (like the 'close' gadget on most windows), and one which contains all the information about a process or task. These objects are vitally important and so are supplied with E in the form of 'modules'. Each module is specific to a certain area of the Amiga system and contains object and other definitions. Modules are discussed in more detail later (see Modules).

1.22 LIST and STRING Types

```
LIST and STRING Types
=====================
```

Arrays are common to many computer languages. However, they can be a bit of a pain because you always need to make sure you haven't run off the end of the array when you're writing to it. This is where the STRING and LIST types come in. STRING is very much like ARRAY OF CHAR and LIST is like ARRAY OF LONG. However, each has a set of E (built-in) functions which safely manipulate variables of these types without exceeding their bounds.

1.23 Normal strings and E-strings

```
Normal strings and E-strings
----------------------------
```

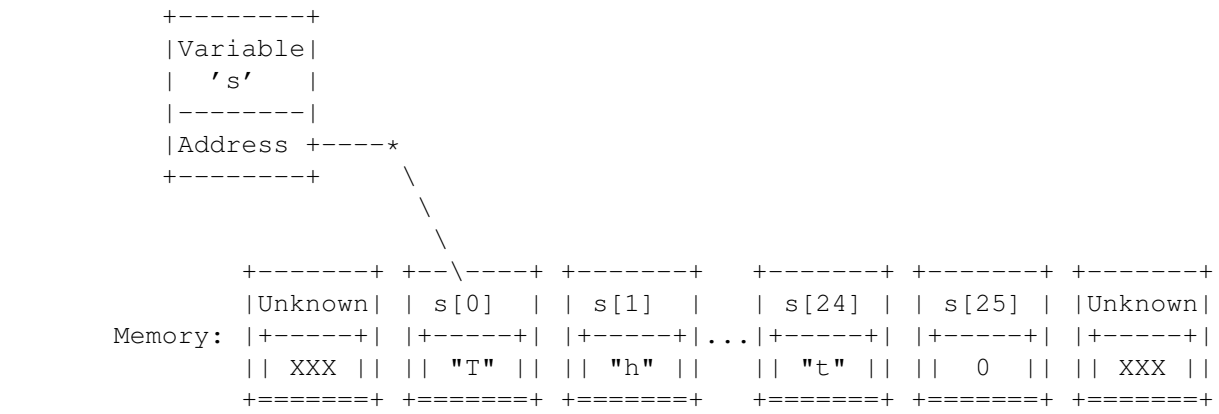
Normal strings are common to most programming languages. They are simply an array of characters, with the end of the string marked by a null character (ASCII zero). We've already met normal strings (see Strings). The ones we used were constant strings contained in ' characters, and they denote pointers to the memory where the string data is stored. Therefore, you can assign a string constant to a pointer (to CHAR), and you've got a ready-filled array with the elements you want (an initialised array.).

```
DEF s:PTR TO CHAR
s:='This is a string constant'
/* Now s[] is T and s[2] is i */
```

Remember that LONG is actually PTR TO CHAR so this code is precisely the same as:

```
DEF s
s:='This is a string constant'
```

The following diagram illustrates the above assignment to s. The first two characters s[0] and s[1]) are T and h, and the last character (before the terminating null) is t. Memory marked as 'Unknown' is not part of the string constant.



E-strings are very similar to normal strings and, in fact, an E-string can be used wherever a normal string can. However, the reverse is not true, so if something requires an E-string you cannot use a normal string instead. The difference between a normal string and an E-string was hinted at in the introduction to this section: E-strings can be safely altered without exceeding their bounds. A normal string is just an array so you need to be careful not to exceed its bounds. However, an E-string knows what its bounds are, and so any of the string manipulation functions can alter them safely.

An E-string (STRING type) variable is declared as in the following example, with the maximum size of the E-string given just like an array declaration.

```
DEF s[30]:STRING
```

As with an array declaration, the variable s is actually a pointer to the string data. To initialise an E-string you need to use the function StrCopy as we shall see.

There are some worked examples in Part Three (see String Handling and I-O) which show how to use normal strings and E-strings.

1.24 String functions

```
String functions
----------------
```

There are a number of useful built-in functions which manipulate strings. Remember that an E-string can be used wherever a normal string can, but normal strings cannot be used where an E-string is required. If

a parameter is marked as string then a normal or E-string can be passed as that parameter, but if it is marked as e-string then only an E-string may be used. Some of these functions have default arguments, which means you don't need to specify some parameters to get the default values (see Default Arguments). (You can, of course, ignore the defaults and always give all parameters.)

String(maxsize)
Allocates memory for an E-string of maximum size maxsize and returns a pointer to the string data. It is used to make space for a new E-string, like a STRING declaration does. The following code fragments are practically equivalent:

```
DEF s[37]:STRING
```

```
DEF s:PTR TO CHAR
s:=String(37)
```

The slight difference is that there may not be enough memory left to hold the E-string when the String function is used. In that case the special value NIL (a constant) is returned. Your program must check that the value returned is not NIL before you use it as an E-string (or dereference it).

The memory for the declaration version, STRING, is allocated when the program is run, so your program won't run if there isn't enough memory. The String version is often called dynamic allocation because it happens only when the program is running; the declaration version has allocation done by the E compiler.

The memory allocated using String can be deallocated using DisposeLink (see System support functions).

StrCmp(string1,string2,length=ALL)
Compares string1 with string2 (they can both be normal or E-strings). Returns TRUE if the first length characters of the strings match, and FALSE otherwise. The length defaults to the special constant ALL which means that the strings must agree on every character. For example, the following comparisons all return TRUE:

```
StrCmp('ABC',  'ABC')
StrCmp('ABC',  'ABC', ALL)
StrCmp('ABCd', 'ABC',    3)
StrCmp('ABCde','ABCxxjs',3)
```

And the following return FALSE (notice the case of the letters):

```
StrCmp('ABC',  'ABc')
StrCmp('ABC',  'ABc', ALL)
StrCmp('ABCd', 'ABC', ALL)
```

StrCopy(e-string,string,length=ALL)
Copies the contents of string to e-string, and also returns a pointer to the resulting E-string (for convenience). Only length characters are copied from the source string, but the special constant ALL can be used to indicate that the whole of the source string is to be copied (and this is the default value for length).

Remember that E-strings are safely manipulated, so the following code fragment results in s becoming More th, since its maximum size is (from its declaration) seven characters.

```
DEF s[7]:STRING
StrCopy(s, 'More than seven characters', ALL)
```

A declaration using STRING (or ARRAY) reserves a small part of memory, and stores a pointer to this memory in the variable being declared. So to get data into this memory you need to copy it there, using StrCopy. If you're familiar with very high-level languages like BASIC you should take care, because you might think you can assign a string to an array or an E-string variable. In E (and languages like C and Assembly) you must explicitly copy data into arrays and E-strings. You should not do the following:

```
/* You don't want to do things like this! */
DEF s[80]:STRING
s:='This is a string constant'
```

This is fairly disastrous: it throws away the pointer to reserved memory that was stored in s and replaces it by a pointer to the string constant. s is then no longer an E-string, and cannot be repaired using SetStr. If you want s to contain the above string you must use StrCopy:

```
DEF s[80]:STRING
StrCopy(s,'This is a string constant')
```

The moral is: remember when you are using pointers to data and when you need to copy data. Also, remember that assignment does not copy large arrays of data, it copies only pointers to data, so if you want to store some data in an ARRAY or STRING type variable you need to copy it there.

StrAdd(e-string,string,length=ALL)

This does the same as StrCopy but the source string is copied onto the end of the destination E-string. The following code fragment results in s becoming This is a string and a half.

```
DEF s[30]:STRING
StrCopy(s, 'This is a string', ALL)
StrAdd(s,  ' and a half')
```

StrLen(string)

Returns the length of string. This assumes that the string is terminated by a null character (i.e., ASCII zero), which is true for any strings made from E-strings and string constants. However, you can make a string constant look short if you use the null character (the special sequence \0) in it. For instance, these calls all return three:

```
StrLen('abc')
StrLen('abc\0def')
```

In fact, most of the string functions assume strings are null-terminated, so you shouldn't use null characters in your strings

unless you really know what you're doing.

For E-strings StrLen is less efficient than the EstrLen function.

EstrLen(e-string)
Returns the length of e-string (remember this can be only an E-string). This is much more efficient than StrLen since E-strings know their length and it doesn't need to search the string for a null character.

StrMax(e-string)
Returns the maximum length of e-string. This is not necessarily the current length of the E-string, rather it is the size used in the declaration with STRING or the call to String.

RightStr(e-string1,e-string2,length)
This is like StrCopy but it copies the right-most characters from e-string2 to e-string1 and both strings must be E-strings. At most length characters are copied, and the special constant ALL cannot be used (to copy all the string you should, of course, use StrCopy). For instance, a value of one for length means the last character of e-string2 is copied to e-string1.

MidStr(e-string,string,index,length=ALL)
Copies the contents of string starting at index (which is an index just like an array index) to e-string. At most length characters are copied, and the special constant ALL can be used if all the remaining characters in string should be copied (this is the default value for length). For example, the following two calls to MidStr result in s becoming four:

```
DEF s[30]:STRING
MidStr(s, 'Just four',      5)
MidStr(s, 'Just four apples', 5, 4)
```

InStr(string1,string2,startindex=0)
Returns the index of the first occurrence of string2 in string1 starting at startindex (in string1). startindex defaults to zero. If string2 could not be found then -1 is returned.

TrimStr(string)
Returns the address of (i.e., a pointer to) the first non-whitespace character in string. For instance, the following code fragment results in s becoming 12345.

```
DEF s:PTR TO CHAR
s:=TrimStr('  \n \t   12345')
```

LowerStr(string)
Converts all uppercase letters in string to lowercase. This change is made in-place, i.e., the contents of the string are directly affected. The string is returned for convenience.

UpperStr(string)
Converts all lowercase letters in string to uppercase. Again, this change is made in-place and the string is returned for convenience.

SetStr(e-string,length)

Sets the length of e-string to length. E-strings know how long they are, so if you alter an E-string (without using an E-string function) and change its size you need to set its length using this function before you can use it as an E-string again. For instance, if you've used an E-string like an array (which you can do) and written characters to it directly you must set its length before you can treat it as anything other than an array:

```
DEF s[10]:STRING
s[0]:="a"     /* Remember that "a" is a character value. */
s[1]:="b"
s[2]:="c"
s[3]:="d"     /* At this point s is just an array of CHAR. */
SetStr(s, 4)  /* Now, s can be used as an E-string again.  */
SetStr(s, 2)  /* s is a bit shorter, but still an E-string.*/
```

Notice that this function can be used to shorten an E-string, but this change is destructive (it cannot easily be reversed to give the original, longer E-string).

Val(string,address=NIL)

What this function does is straight-forward but how you use it is a bit complicated. Basically, it converts string to a LONG integer. Leading whitespace is ignored, and a leading % or $ means that the string denotes a binary or hexadecimal integer (in the same way they do for numeric constants). The decoded integer is returned as the regular return value (see Multiple Return Values). The number of characters of string that were read to make the integer is stored at address, which is usually a variable address (from using {var }), and is also returned as the first optional return value. If address is the special constant NIL (i.e., zero) then this number is not stored (this is the default value for address). You can use this number to calculate the position in the string which was not part of the integer. If an integer could not be decoded from the string then zero is returned as both return values and stored at address.

Follow the comments in this example, and pay special attention to the use of the pointer p.

```
DEF s[30]:STRING, value, chars, p:PTR TO CHAR
StrCopy(s, ' \t \n 10 \t $3F -%0101010')
value, chars:=Val('abcde 10 20')  -> Two return values...
  /* After the above line, value and chars will both be zero */
value:=Val(s, {chars})          -> Use address of chars
  /* Now value will be 10, chars will be 7 */
p:=s+chars
  /* p now points to the space after the 10 in s */
value, chars:=Val(p)
  /* Now value will be $3F (63), chars will be 6 */
p:=p+chars
  /* p now points to the space after the $3F in s */
value, chars:=Val(p)
  /* Now value will be -%0101010 (-42), chars will be 10 */
```

Notice the two different ways of finding the number of characters

```
     read: a multiple-assignment and using the address of a variable.
```

There's a couple of other string functions (ReadStr and StringF) which will be discussed later (see Input and output functions).

1.25 Lists and E-lists

```
Lists and E-lists
-----------------
```

Lists are just like strings with LONG elements rather than CHAR elements (so they are very much like ARRAY OF LONG). The list equivalent of an E-string is something called an E-list. It has the same properties as an E-string, except the elements are LONG (so could be pointers). Normal lists are most like string constants, except that the elements can be built from variables and so do not have to be constants. Just as strings are not true E-strings, (normal) lists are not true E-lists.

Lists are written using [and] to delimit comma separated elements. Like string constants a list returns the address of the memory which contains the elements.

For example the following code fragment:

```
DEF list:PTR TO LONG, number
number:=22
list:=[1,2,3,number]
```

is equivalent to:

```
DEF list[4]:ARRAY OF LONG, number
number:=22
list[0]:=1
list[1]:=2
list[2]:=3
list[3]:=number
```

Now, which of these two versions would you rather write? As you can see, lists are pretty useful for making your program easier to write and much easier to read.

E-list variables are like E-string variables and are declared in much the same way. The following code fragment declares lt to be an E-list of maximum size 30. As ever, lt is then a pointer (to LONG), and it points to the memory allocated by the declaration.

```
DEF lt[30]:LIST
```

Lists are most useful for writing tag lists, which are increasingly used in important Amiga system functions. A tag list is a list where the elements are thought of in pairs. The first element of a pair is the tag, and the second is some data for that tag. See the 'Rom Kernel Reference Manual (Libraries)' for more details.

1.26 List functions

```
List functions
--------------

   There are a number of list functions which are very similar to the
string functions (see String functions).  Remember that E-lists are the
list equivalents of E-strings, i.e., they can be altered and extended
safely without exceeding their bounds.  As with E-strings, E-lists are
downwardly compatible with lists.  Therefore, if a function requires a
list as a parameter you can supply a list or an E-list.  But if a function
requires an E-list you cannot use a list in its place.

List(maxsize)
     Allocates memory for an E-list of maximum size maxsize and returns
     a pointer to the list data.  It is used to make space for a new
     E-list, like a LIST declaration does.  The following code fragments
     are (as with String) practically equivalent:

            DEF lt[46]:LIST

            DEF lt:PTR TO LONG
            lt:=List(46)

     You need to check that the return value from List is not NIL before
     you use it as an E-list.  Like String, the memory allocated using
     List is deallocated using DisposeLink (see
     System support functions).

ListCmp(list1,list2,length=ALL)
     Compares list1 with list2 (they can both be normal or E-lists).
     Works just like StrCmp does for E-strings, so, for example, the
     following comparisons all return TRUE:

            ListCmp([1,2,3,4],   [1,2,3,4])
            ListCmp([1,2,3,4],   [1,2,3,7], 3)
            ListCmp([1,2,3,4,5], [1,2,3],   3)

ListCopy(e-list,list,length=ALL)
     Works just like StrCopy, and the following example shows how to
     initialise an E-list:

            DEF lt[7]:LIST, x
            x:=4
            ListCopy(lt, [1,2,3,x])

     As with StrCopy, an E-list cannot be over-filled using ListCopy.

ListAdd(e-list,list,length=ALL)
     Works just like StrAdd, so this next code fragment results in the
     E-list lt becoming the E-list version of [1,2,3,4,5,6,7,8].

            DEF lt[30]:LIST
```

```
            ListCopy(lt, [1,2,3,4])
            ListAdd(lt, [5,6,7,8])

ListLen(list)
     Works just like StrLen, returning the length of list.  There is no
     E-list specific length function.

ListMax(e-list)
     Works just like StrMax, returning the maximum length of the e-list.

SetList(e-list,length)
     Works just like SetStr, setting the length of e-list to length.

ListItem(list,index)
     Returns the element of list at index.  For example, if lt is an
     E-list (so a PTR TO LONG) then ListItem(lt,n) is the same as lt[n].
     This function is most useful when the list is not an E-list; for
     example, the following two code fragments are equivalent:

            WriteF(ListItem(['Fred','Barney','Wilma','Betty'], name))

            DEF lt:PTR TO LONG
            lt:=['Fred','Barney','Wilma','Betty']
            WriteF(lt[name])
```

1.27 Complex types

```
Complex types
-------------

   In E the STRING and LIST types are called complex types.
Complex-typed variables can also be created using the String and List
functions as we've seen in the previous sections.
```

1.28 Typed lists

```
Typed lists
-----------

   Normal lists contain LONG elements, so you can write initialised arrays
of LONG elements.  What about other kinds of array?  Well, that's what
typed lists are for.  You specify the type of the elements of a list
using  :type after the closing ].  The allowable types are CHAR, INT,
LONG and any object type.  There is a subtle difference between a normal,
LONG list and a typed list (even a LONG typed list): only normal lists can
be used with the list functions (see List functions).  For this reason,
the term 'list' tends to refer only to normal lists.

   The following code fragment uses the object rec defined earlier (see
Example object) and gives a couple of examples of typed lists:
```

```
       DEF ints:PTR TO INT, objects:PTR TO rec, p:PTR TO CHAR
       ints:=[1,2,3,4]:INT
       p:='fred'
       objects:=[1,2,p,4,
                 300,301,'barney',303]:rec
```

It is equivalent to:

```
       DEF ints[4]:ARRAY OF INT, objects[2]:ARRAY OF rec, p:PTR TO CHAR
       ints[0]:=1
       ints[1]:=2
       ints[2]:=3
       ints[3]:=4
       p:='fred'
       objects[0].tag:=1
       objects[0].check:=2
       objects[0].table:=p
       objects[0].data:=4
       objects[1].table:='barney'
       objects[1].tag:=300
       objects[1].data:=303
       objects[1].check:=301
```

The last group of assignments to objects[1] have deliberately been shuffled in order to emphasise that the order of the elements in the definition of the object rec is significant. Each of the elements of the list corresponds to an element in the object, and the order of elements in the list corresponds to the order in the object definition. In the example, the (object) list assignment line was broken after the end of the first object (the fourth element) to make it a bit more readable.

The last object in the list need not be completely defined, so, for instance, the second line of the assignment could have contained only three elements. This makes an object-typed list slightly different from the corresponding array of objects, since an array always defines a whole number of objects. With an object-typed list you must be careful not to access the undefined elements of a partially defined, trailing object.

1.29 Static data

```
Static data
-----------
```

String constants (e.g., fred), lists (e.g., [1,2,3]) and typed lists (e.g., [1,2,3]:INT) are static data. This means that the address of the (initialised) data is fixed when the program is run. Normally you don't need to worry about this, but, for instance, if you want to have a series of lists as initialised arrays you might be tempted to use some kind of loop:

```
     PROC main()
       DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
       FOR i:=0 TO 9
```

```
    a[i]:=[1, i, i*i]
      /* This assignment is probably not what you want! */
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    WriteF('a[\d] is an array at address \d\n', i, p)
    WriteF('  and the second element is \d\n', p[1])
  ENDFOR
ENDPROC
```

The array a is an array of pointers to initialised arrays (which are all three elements long). But, as the comment suggests and the program shows, this probably doesn't do what was intended, since the list is static. That means the address of the list is fixed, so each element of a gets the same address (i.e., the same array). Since i is used in the list, the contents of that part of memory varies slightly as the first FOR loop is processed. But after this loop the contents remain fixed, and the second element of each of the ten arrays is always nine. This is an example of the output that will be generated (the ... represents a number of similar lines):

```
a[0] is an array at address 4021144
  and the second element is 9
a[1] is an array at address 4021144
  and the second element is 9
...
a[9] is an array at address 4021144
  and the second element is 9
```

One solution is to use the dynamic typed-allocation operator NEW (see NEW and END Operators). Another solution is to use the function List and copy the normal list into the new E-list using ListCopy:

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=List(3)
    /* Must check that the allocation succeeded before copying */
    IF a[i]<>NIL THEN ListCopy(a[i], [1, i, i*i], ALL)
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    IF p=NIL
      WriteF('Could not allocate memory for a[\d]\n', i)
    ELSE
      WriteF('a[\d] is an array at address \d\n', i, p)
      WriteF('  and the second element is \d\n', p[1])
    ENDIF
  ENDFOR
ENDPROC
```

The problem is not so bad with string constants, since the contents are fixed. However, if you alter the contents explicitly, you will need to take care not to run into the same problem, as this next example shows.

```
PROC main()
  DEF i, strings[10]:ARRAY OF LONG, s:PTR TO CHAR
```

```
    FOR i:=0 TO 9
      strings[i]:='Hello World\n'
        /* This assignment is probably not what you want! */
    ENDFOR
    s:=strings[4]
    s[5]:="X"
    FOR i:=0 TO 9
      WriteF('strings[\d] is ', i)
      WriteF(strings[i])
    ENDFOR
  ENDPROC
```

This is an example of the output that will be generated (again, the ... represents a number of similar lines)::

```
  strings[0] is HelloXWorld
  strings[1] is HelloXWorld
  ...
  strings[9] is HelloXWorld
```

The solution, once more, is to use dynamic allocation. The functions String and StrCopy should be used in the same way that List and ListCopy were used above.

1.30 Linked Lists

```
Linked Lists
============
```

E-lists and E-strings have a useful extension: they can be used to make linked lists. These are like the lists we've seen already, except the list elements do not occupy a contiguous block of memory. Instead, each element has an extra piece of data: a pointer to the next element in the list. This means that each element can be anywhere in memory. (Normally, the next element of a list is in the next position in memory.) The end of a linked list has been reached when the pointer to the next element is the special value NIL (a constant representing zero). You need to be very careful to check that the pointer is not NIL, or else strange things will happen to your program...

The elements of a linked list are E-lists or E-strings (i.e., the elements are complex typed). So, you can link E-lists to get a 'linked list of E-lists' (or, more simply, a 'list of lists'). Similarly, linking E-strings gives 'linked list of E-strings', or a 'list of strings'. You don't have to stick to these two kinds of linked lists, though: you can use a mixture of E-lists and E-strings in the same linked list. To link one complex typed element to another you use the Link function and to find subsequent elements in a linked list you use the Next or Forward functions.

```
Link(complex1,complex2)
```

Links complex1 to complex2. Both must be an E-list or an E-string, with the exception that complex2 can be the special constant NIL to indicate that complex1 is the end of the linked list. The value complex1 is returned by the function, which isn't

always useful so, usually, calls to Link will be used as statements rather than functions.

The effect of Link is that complex1 will point to complex2 as the next element in the linked list (so complex1 is the head of the list, and complex2 is the tail). For both E-lists and E-strings the pointer to the next element is initially NIL, so you will only need to use Link with a NIL parameter when you want to make a linked list shorter (by losing the tail).

Next(complex)

Returns the pointer to the next element in the linked list. This may be the special constant NIL if complex is the last element in the linked list. Be careful to check that the value isn't NIL before you dereference it! Follow the comments in the example below:

```
DEF s[23]:STRING, t[7]:STRING, lt[41]:LIST, lnk
/* The next two lines set up the linked list "lnk" */
lnk:=Link(lt,t)  /* lnk list starts at lt and is lt->t    */
lnk:=Link(s,lt)  /*   Now it starts at s  and is s->lt->t */
/* The next three lines follow the links in "lnk"  */
lnk:=Next(lnk)   /*   Now it starts at lt and is lt->t    */
lnk:=Next(lnk)   /*   Now it starts at t  and is t        */
lnk:=Next(lnk)   /* Now lnk is NIL so the list has ended  */
```

You may safely call Next with a NIL parameter, and in this case it will return NIL.

Forward(complex,expression)

Returns a pointer to the element which is expression number of links down the linked list complex. If expression is one, then a pointer to the next element is returned (just like using Next). If it's two a pointer to the element after that is returned, and so on.

If expression is greater than the number of links in the list the special value NIL is returned.

Since the link in a linked list is a pointer to the next element you can look through the list only from beginning to end. Technically this is a singly linked list (a doubly linked list would also have a pointer to the previous element in the list, enabling backwards searching through the list).

Linked lists are useful for building lists that can grow quite large. This is because it's much better to have lots of small bits of memory than a large lump. However, you need only worry about these things when you're playing with quite big lists (as a rough guide, ones with over 100,000 elements are big!).