

**beginner**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> beginner		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		December 6, 2024	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>beginner</b>	<b>1</b>
1.1	Memory Allocation . . . . .	1
1.2	Static Allocation . . . . .	1
1.3	Deallocation of Memory . . . . .	2
1.4	Dynamic Allocation . . . . .	3
1.5	NEW and END Operators . . . . .	6
1.6	Object and simple typed allocation . . . . .	6
1.7	Array allocation . . . . .	8
1.8	List and typed list allocation . . . . .	8
1.9	OOP object allocation . . . . .	10

# Chapter 1

## beginner

### 1.1 Memory Allocation

Memory Allocation

\*\*\*\*\*

When a program is running memory is being used in various different ways. In order to use any memory it must first be allocated, which is simply a way of marking memory as being 'in use'. This is to prevent the same piece of memory being used for different data storage (e.g., by different programs), and so helps prevent corruption of the data stored there. There are two general ways in which memory can be allocated: dynamically and statically.

Static Allocation

Deallocation of Memory

Dynamic Allocation

NEW and END Operators

### 1.2 Static Allocation

Static Allocation

=====

Statically allocated memory is memory allocated by the program for variables and static data like string constants, lists and typed lists (see Static data). Every variable in a program requires some memory in which to store its value. Variables declared to be of type ARRAY, LIST, STRING or any object require two lots of memory: one to hold the value of the pointer and one to hold the large amount of data (e.g., the elements in an ARRAY). In fact, such declarations are merely PTR TO type declarations together with an initialisation of the pointer to the address of some (statically) allocated memory to hold the data. The following example shows very similar declarations, with the difference being that in the second case (using PTR) only memory to hold the pointer values is allocated. The first case also allocates memory to hold the appropriate

---

size of array, object and E-string.

```
DEF a[20]:ARRAY,    m:myobj,        s[10]:STRING
```

```
DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
```

The pointers in the second case are not initialised by the declaration and, therefore, they are not valid pointers. This means that they should not be dereferenced in any way, until they have been initialised to the address of some allocated memory. This usually involves dynamic allocation of memory (see Dynamic Allocation).

## 1.3 Deallocation of Memory

Deallocation of Memory

=====

When memory is allocated it is, conceptually, marked as being 'in use'. This means that this piece of memory cannot be allocated again, so a different piece will be allocated (if any is available) when the program wants to allocate some more. In this way, variables are allocated different pieces of memory, and so their values can be distinct. But there is only a certain amount of memory available, and if it could not be marked as 'not in use' again it would soon run out (and the program would come to a nasty end). This is what deallocation does: it marks previously allocated memory as being 'not in use' and so makes it available for allocation again. However, memory should be deallocated only when it is actually no longer in use, and this is where things get a bit complicated.

Memory is such a vital resource in every computer that it is important to use as little of it as necessary and to deallocate it whenever possible. This is why a programming language like E handles most of the memory allocation for variables. The memory allocated for variables can be automatically deallocated when it is no longer possible for the program to use that variable. However, this automatic deallocation is not useful for global variables, since they can be used from any procedure and so can be deallocated only when the program terminates. A procedure's local variables, on the other hand, are allocated when the procedure is called but cannot be used after the procedure returns. They can, therefore, be deallocated when the procedure returns.

Pointers, as always, can cause big problems. The following example shows why you need to be careful when using pointers as the return value of a procedure.

```
/* This is an example of what *NOT* to do */
PROC fullname(first, last)
  DEF full[40]:STRING
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full
```

```
PROC main()
  WriteF('Name is \s\n', fullname('Fred', 'Flintstone'))
ENDPROC
```

On first sight this seems fine, and, in fact, it may even work correctly if you run it once or twice (but be careful: it could crash your machine). The problem is that the procedure `fullname` returns the value of the local variable `full`, which is a pointer to some statically allocated memory for the E-string and this memory will be deallocated when the procedure returns. This means that the return value of any call to `fullname` is the address of recently deallocated memory, so it is invalid to dereference it. But the call to `WriteF` does just that: it dereferences the result of `fullname` in order to print the E-string it points to. This is a very common problem, because it is such an easy thing to do. The fact that it may, on many occasions, appear to work makes it much harder to find, too. The solution, in this case, is to use dynamic allocation (see [Dynamic Allocation](#)).

If you're still a bit sceptical that this really is a problem, try the above `fullname` procedure definition with either of these replacement main procedures, but be aware, again, that each one has the potential to crash your machine.

```
/* This might not print the correct string */
PROC main()
  DEF f
    f:=fullname('Fred', 'Flintstone')
    WriteF('Name is \s\n', f)
  ENDPROC

/* This will definitely print g instead of f */
PROC main()
  DEF f, g
    f:=fullname('Fred', 'Flintstone')
    g:=fullname('Barney', 'Rubble')
    WriteF('Name is \s\n', f)
  ENDPROC
```

(The reason why things go wrong is outlined above, but the reasons why each prints what it does is beyond the scope of this Guide.)

## 1.4 Dynamic Allocation

Dynamic Allocation  
=====

Dynamically allocated memory is any memory that is not statically allocated. To allocate memory dynamically you can use the `List` and `String` functions, all flavours of `New`, and the versatile `NEW` operator. But because the memory is dynamically allocated it must be explicitly deallocated when no longer needed. In all the above cases, though, any memory that is still allocated when the program terminates will be deallocated automatically.

---

Another way to allocate memory dynamically is to use the Amiga system functions based on AllocMem. However, these functions require that the memory allocated using them be deallocated (using functions like FreeMem) before the program terminates, or else it will never be deallocated (not until your machine is rebooted, anyway). It is safer, therefore, to try to use the E functions for dynamic allocation whenever possible.

There are many reasons why you might want to use dynamic allocation, and most of them involve initialisation of pointers. For example, the declarations in the section about static allocation can be extended to give initialisations for the pointers declared in the second DEF line (see Static Allocation).

```
DEF a[20]:ARRAY,    m:myobj,        s[10]:STRING

DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
a:=New(20)
m:=New(SIZEOF myobj)
s:=String(20)
```

These are initialisations to dynamically allocated memory, whereas the first line of declarations initialise similar pointers to statically allocated memory. If these sections of code were part of a procedure then, since they would now be local variables, there would be one other, significant difference: the dynamically allocated memory would not automatically be deallocated when the procedure returns, whereas the statically allocated memory would. This means that we can solve the deallocation problem (see Deallocation of Memory).

```
/* This is the correct way of doing it */
PROC fullname(first, last)
  DEF full
  full:=String(40)
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

PROC main()
  DEF f, g
  WriteF('Name is \s\n', fullname('Fred', 'Flintstone'))
  f:=fullname('Fred', 'Flintstone')
  g:=fullname('Barney', 'Rubble')
  WriteF('Name is \s\n', f)
ENDPROC
```

The memory for the E-string pointed to by full is now allocated dynamically, using String, and is not deallocated until the end of the program. This means that it is quite valid to pass the value of full as the result of the procedure fullname, and it is quite valid to dereference the result by printing it using WriteF. However, this has caused one last problem: the memory is not deallocated until the end of the program, so is potentially wasted since it could be used, for example, to hold the results of subsequent calls. Of course, the memory can be deallocated only when the data it stores is no longer required. The following replacement main procedure shows when you might want to deallocate the E-string (using DisposeLink).

```

PROC main()
  DEF f, g
  f:=fullname('Fred', 'Flintstone')
  WriteF('Name is \s, f points to $\h\n', f, f)
/* Try this with and without the next DisposeLink line */
  DisposeLink(f)
  g:=fullname('Barney', 'Rubble')
  WriteF('Name is \s, g points to $\h\n', g, g)
  DisposeLink(g)
ENDPROC

```

If you run this with the `DisposeLink(f)` line you'll probably find that `g` will be a pointer to the same memory as `f`. This is because the call to `DisposeLink` has deallocated the memory pointed to by `f`, so it can be reused to store the E-string pointed to by `g`. If you comment out (or delete) the `DisposeLink` line, then you will find that `f` and `g` always point to different memory.

In some ways it is best to never do any deallocation, because of the problems you can get into if you deallocate memory too early (i.e., before you've finished with the data it contains). Of course, it is safe (but temporarily wasteful) to do this with the E dynamic allocation functions, but it is very wasteful (and wrong) to do this with the Amiga system functions like `AllocMem`.

Another benefit of using dynamic allocation is that the size of the arrays, E-lists and E-strings that can be created can be the result of any expression, so is not restricted to constant values. (Remember that the size given on `ARRAY`, `LIST` and `STRING` declarations must be a constant.) This means that the `fullname` procedure can be made more efficient and allocate only the amount of memory it needs for the E-string it creates.

```

PROC fullname(first, last)
  DEF full
  /* The extra +1 is for the added space */
  full:=String(StrLen(first)+StrLen(last)+1)
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

```

However, it may be very complicated or inefficient to calculate the correct size. In these cases, a quick, constant estimate might be better, overall.

The various functions for allocating memory dynamically have corresponding functions for deallocating that memory. The following table shows some of the more common pairings.

Allocation	Deallocation
-----	-----
New	Dispose
NewR	Dispose
List	DisposeLink
String	DisposeLink
NEW	END



FastNew	FastDispose
AllocMem	FreeMem
AllocVec	FreeVec
AllocDosObject	FreeDosObject

NEW and END are versatile and powerful operators, discussed in the following section. The functions beginning with Alloc- are Amiga system functions and are paired with similarly suffixed functions with a Free- prefix. See the 'Rom Kernel Reference Manual' for more details.

## 1.5 NEW and END Operators

NEW and END Operators  
=====

To help deal with dynamic allocation and deallocation of memory there are two, powerful operators, NEW and END. The NEW operator is very versatile, and similar in operation to the New family of built-in functions (see System support functions). The END operator is the deallocating complement of NEW (so it is similar to the Dispose family of built-in functions). The major difference between NEW and the various flavours of New is that NEW allocates memory based on the types of its arguments.

Object and simple typed allocation  
Array allocation  
List and typed list allocation  
OOP object allocation

## 1.6 Object and simple typed allocation

Object and simple typed allocation  
-----

The following sections of code are roughly equivalent and serve to show the function of NEW, and how it is closely related to NewR. (The type can be any object or simple type.)

```
DEF p:PTR TO type
NEW p

DEF p:PTR TO type
p:=NewR(SIZEOF type)
```

Notice that the use of NEW is not like a function call, as there are no parentheses around the parameter p. This is because NEW is an operator rather than a function. It works differently from a function, since it also needs to know the types of its arguments. This means that the declaration of p is very important, since it governs how much memory is

---

allocated by NEW. The version using NewR explicitly gives the amount of memory to be allocated (using the SIZEOF operator), so in this case the declared type of p is not so important for correct allocation.

The next example shows how NEW can be used to initialise several pointers at once. The second section of code is roughly equivalent, but uses NewR. (Remember that the default type of a variable is LONG, which is actually PTR TO CHAR.)

```
DEF p:PTR TO LONG, q:PTR TO myobj, r
NEW p, q, r
```

```
DEF p:PTR TO LONG, q:PTR TO myobj, r
p:=NewR(SIZEOF LONG)
q:=NewR(SIZEOF myobj)
r:=NewR(SIZEOF CHAR)
```

These first two examples have shown the statement form of NEW. There is also an expression form, which has one parameter and returns the address of the newly allocated memory as well as initialising the argument pointer to this address.

```
DEF p:PTR TO myobj, q:PTR TO myobj
q:=NEW p
```

```
DEF p:PTR TO myobj, q:PTR TO myobj
q:=(p:=NewR(SIZEOF type))
```

This may not seem desperately useful, but it's also the way that NEW is used to allocate copies of lists and typed lists (see List and typed list allocation).

To deallocate memory allocated using NEW you use the END statement with the pointers that you want to deallocate. To work properly, END requires that the type of each pointer matches the type used when it was allocated with NEW. Failure to do this will result in an incorrect amount of memory being deallocated, and this can cause many subtle problems in a program. You must also be careful not to deallocate the same memory twice, and to this end the pointers given to END are re-initialised to NIL after the memory they point to is deallocated (it is quite safe to use END with a pointer which is NIL). This does not catch all problems, however, since more than one pointer can point to the same piece of memory, as shown in the example below.

```
DEF p:PTR TO LONG, q:PTR TO LONG
q:=NEW p
p[]:=-24
q[]:=613
END p
/* p is now NIL, but q is now invalid but not NIL */
```

The first assignment initialises q to be the same as p (which is initialised by NEW). Both the next two assignments change the value pointed to by both p and q. The memory allocated to store this value is then deallocated, using END, and this also sets p to NIL. However, the address stored in q is not altered, and still points to the memory that has just been deallocated. This means that q now has a plausible, but

invalid, pointer value. The only thing that can safely be done with `q` is re-initialise it. One of the worst things that could be done is to use it with `END`, which would deallocate the same memory again, and potentially crash your machine. So, in summary, don't deallocate the same pointer value more than once, and keep track of which variables point to the same memory as others.

Just as a use of `NEW` has a simple (but rough) equivalent using `NewR`, `END` has an equivalent using `Dispose`, as shown by the following sections of code.

```
END p

IF p
  Dispose(p)
  p:=NIL
ENDIF
```

In fact, it's a tiny bit more complicated than that, since OOP objects are allocated and deallocated using `NEW` and `END` (see Object Oriented E).

## 1.7 Array allocation

Array allocation

Arrays can also be allocated using `NEW`, and this works in a very similar way to that outlined in the previous section. The difference is that the size of the array must also be supplied, in both the use of `NEW` and `END`. Of course, the size supplied to `END` must be the same as the size supplied to the appropriate use of `NEW`. All this extra effort also gains you the ability to create an array of a size which is not a constant (unlike variables of type `ARRAY`). This means that the size supplied to `NEW` and `END` can be the result of an arbitrary expression.

```
DEF a:PTR TO LONG, b:PTR TO myobj, s
NEW a[10] /* A dynamic array of LONG */
s:=my_random(20)
NEW b[s] /* A dynamic array of myobj */
/* ...some other code... */
END a[10], b[s]
```

The `my_random` function stands for some arbitrary calculation, to show that `s` does not have to be a constant. This form of `NEW` can also be used as an expression, as before.

## 1.8 List and typed list allocation

List and typed list allocation

-----

---

Lists and typed lists are usually static data, but NEW can be used to create dynamically allocated versions. This form of NEW can be used only as an expression, and it takes the list (or typed list) as its argument and returns the address of the dynamically allocated copy of the list. Deallocation of the memory allocated in this way is a bit more complicated than before, but you can, of course, let it be deallocated automatically at the end of the program.

The following example shows how simple it is to use NEW to cure the static data problem described previously (see Static data). The difference from the original, incorrect program is very subtle.

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=NEW [1, i, i*i]
    /* a[i] is now dynamically allocated */
  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 minor alteration is to prefix the list with NEW, thereby making the list dynamic. This means that each a[i] is now a different list, rather than the same, static list of the original version of the program.

Typed lists are allocated in a similar way, and the following example also shows how to deallocate this memory. Basically, you need to know how long the new array is (i.e., how many elements there are), since a typed list is really just an initialised array. You can then deallocate it like a normal array, remembering to use an appropriately typed pointer. Object-typed lists are restricted (when used with NEW) to an array of at most one object, so is useful only for allocating an initialised object (not really an array). Notice how, in the following code, the pointer q can be treated both as an object and as an array of one object (see Element selection and element types).

```
OBJECT myobj
  x:INT, y:LONG, z:INT
ENDOBJECT

PROC main()
  DEF p:PTR TO INT, q:PTR TO myobj
  p:=NEW [1, 9, 3, 7, 6]:INT
  q:=NEW [1, 2]:myobj
  WriteF('Last element in array p is \d\n', p[4])
  WriteF('Object q is x=\d, y=\d, z=\d\n',
    q.x, q.y, q.z)
  WriteF('Array q is q[0].x=\d, q[0].y=\d, q[0].z=\d\n',
    q[0].x, q[0].y, q[0].z)
  END p[5], q
ENDPROC
```

The dynamically allocated version of an object-typed list differs from the static version in another way: it always has memory allocated for a whole number of objects, so a partially initialised object is padded with zero elements. The static version does not allocate this extra padding, so you must be careful not to access any element beyond those mentioned in the list.

The deallocation of NEW copies of normal lists can, as ever, be left to be done automatically at the end of the program. If you want to deallocate them before this time you must use the function `FastDisposeList`, passing the address of the list as the only argument. You must not use `END` or any other method of deallocation. `FastDisposeList` is the only safe way of deallocating lists allocated using `NEW`.

## 1.9 OOP object allocation

OOP object allocation

-----

Currently, the only way to create OOP objects in E is to use `NEW` and the only safe way to destroy them is to use `END`. This is probably the most common use of `NEW` and `END` and is described in detail later (see Objects in E).