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 Exception Handling

```
Exception Handling
******************
```

Often your program has to check the results of functions and do different things if errors have occurred. For instance, if you try to open a window (using OpenW), you may get a NIL pointer returned which shows that the window could not be opened for some reason. In this case you normally can't continue with the program, so you must tidy up and terminate. Tidying up can sometimes involve closing windows, screens and libraries, so sometimes your error cases can make your program cluttered and messy. This is where exceptions come in--an exception is simply an error case, and exception handling is dealing with error cases. The exception handling in E neatly separates error specific code from the real code of your program.

1.2 Procedures with Exception Handlers

```
Procedures with Exception Handlers
==================================
```

A procedure with an exception handler looks like this:

```
PROC fred(params...) HANDLE
  /* Main, real code */
EXCEPT
  /* Error handling code */
ENDPROC
```

This is very similar to a normal procedure, apart from the HANDLE and EXCEPT keywords. The HANDLE keyword means the procedure is going to have an exception handler, and the EXCEPT keyword marks the end of the normal code and the start of the exception handling code. The procedure works just as normal, executing the code in the part before the EXCEPT, but when an error happens you can pass control to the exception handler (i.e., the code after the EXCEPT is executed).

1.3 Raising an Exception

Raising an Exception
==========

When an error occurs (and you want to handle it), you raise an exception using either the Raise or Throw function. You call Raise with a number which identifies the kind of error that occurred. The code in the exception handler is responsible for decoding the number and then doing the appropriate thing. Throw is very similar to Raise, and the following description of Raise also applies to Throw. The difference is that Throw takes a second argument which can be used to pass extra information to a handler (usually a string). The terms 'raising' and 'throwing' an exception can be used interchangeably.

When Raise is called it immediately stops the execution of the current procedure code and passes control to the exception handler of most recent procedure which has a handler (which may be the current procedure). This is a bit complicated, but you can stick to raising exceptions and handling them in the same procedure, as in the next example:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Raise(ERR_MEM)
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF('Not enough memory\n')
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
ENDPROC
```

This uses an exception handler to print a message saying there wasn't enough memory if the call to New returns NIL. The parameter to Raise is stored in the special variable exception in the exception handler part of the code, so if Raise is called with a number other than ERR_MEM a message saying "Unknown exception" will be printed.

Try running this program with a really large BIG_AMOUNT constant, so that the New can't allocate the memory. Notice that the "Got enough

memory" is not printed if Raise is called. That's because the execution of the normal procedure code stops when Raise is called, and control passes to the appropriate exception handler. When the end of the exception handler is reached the procedure is finished, and in this case the program terminates because the procedure was the main procedure.

If Throw is used instead of Raise then, in the handler, the special variable exceptioninfo will contain the value of the second parameter. This can be used in conjunction with exception to provide the handler with more information about the error. Here's the above example re-written to use Throw:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Throw(ERR_MEM, 'Not enough memory\n')
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF(exceptioninfo)
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
ENDPROC
```

An enumeration (using ENUM) is a good way of getting different constants for various exceptions. It's always a good idea to use constants for the parameter to Raise and in the exception handler, because it makes everything a lot more readable: Raise(ERR_MEM) is much clearer than Raise(1). The enumeration starts at one because zero is a special exception: it usually means that no error occurred. This is useful when the handler does the same cleaning up that would normally be done when the program terminates successfully. For this reason there is a special form of EXCEPT which automatically raises a zero exception when the code in the procedure successfully terminates. This is EXCEPT DO, with the DO suggesting to the reader that the exception handler is called even if no error occurs. Also, the argument to the Raise function defaults to zero if it is omitted (see Default Arguments).

So, what happens if you call Raise in a procedure without an exception handler? Well, this is where the real power of the handling mechanism comes to light. In this case, control passes to the exception handler of the most recent procedure with a handler. If none are found then the program terminates. 'Recent' means one of the procedures involved in calling your procedure. So, if the procedure fred calls barney, then when barney is being executed fred is a recent procedure. Because the main procedure is where the program starts it is a recent procedure for every other procedure in the program. This means, in practice:

* If you define fred to be a procedure with an exception handler then any procedures called by fred will have their exceptions handled by the handler in fred if they don't have their own handler.

```
* If you define main to be a procedure with an exception handler then
  any exceptions that are raised will always be dealt with by some
  exception handling code (i.e., the handler of main or some other
  procedure).
```

Here's a more complicated example:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Hello from main\n')
  fred()
  barney()
  WriteF('Goodbye from main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Hello from fred\n')
  Raise(FRED)
  WriteF(' Goodbye from fred\n')
EXCEPT
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
  WriteF('  Hello from barney\n')
  Raise(BARNEY)
  WriteF('  Goodbye from barney\n')
ENDPROC
```

When you run this program you get the following output:

```
Hello from main
 Hello from fred
 Handler fred: 1
  Hello from barney
```

This is because the fred procedure is terminated by the Raise(FRED) call, and the whole program is terminated by the Raise(BARNEY) call (since barney and main do not have handlers).

Now try this:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Hello from main\n')
  fred()
  WriteF('Goodbye from main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Hello from fred\n')
  barney()
  Raise(FRED)
  WriteF(' Goodbye from fred\n')
EXCEPT
```

```
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
  WriteF('  Hello from barney\n')
  Raise(BARNEY)
  WriteF('  Goodbye from barney\n')
ENDPROC
```

When you run this you get the following output:

```
Hello from main
 Hello from fred
  Hello from barney
 Handler fred: 2
Goodbye from main
```

Now the fred procedure calls barney, so main and fred are recent procedures when Raise(BARNEY) is executed, and therefore the fred exception handler is called. When this handler finishes the call to fred in main is finished, so the main procedure is completed and we see the 'Goodbye' message. In the previous program the Raise(BARNEY) call did not get handled and the whole program terminated at that point.

1.4 Automatic Exceptions

```
Automatic Exceptions
====================
```

In the previous section we saw an example of raising an exception when a call to New returned NIL. We can re-write this example to use automatic exception raising:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

RAISE ERR_MEM IF New()=NIL

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  WriteF('Got enough memory\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF('Not enough memory\n')
  ELSE
    WriteF('Unknown exception\n')
  ENDIF
ENDPROC
```

The only difference is the removal of the IF which checked the value of block, and the addition of a RAISE part. This RAISE part means that whenever the New function is called in the program, the exception ERR_MEM

will be raised if it returns NIL (i.e., the exception ERR_MEM is automatically raised). This unclutters the program by removing a lot of error checking IF statements.

The precise form of the RAISE part is:

```
RAISE exception  IF function()  compare  value ,
      exception2 IF function2() compare2 value2 ,
      ...
      exceptionN IF functionN() compareN valueN
```

The exception is a constant (or number) which represents the exception to be raised, function is the E built-in or system function to be automatically checked, value is the return value to be checked against, and compare is the method of checking (i.e., =, <>, <, <=, > or >=). This mechanism only exists for built-in or library functions because they would otherwise have no way of raising exceptions. The procedures you define yourself can, of course, use Raise to raise exceptions in a much more flexible way.

1.5 Raise within an Exception Handler

Raise within an Exception Handler
==========

If you call Raise within an exception handler then control passes to the next most recent handler. In this way you can write procedures which have handlers that perform local tidying up. By using Raise at the end of the handler code you can invoke the next layer of tidying up.

As an example we'll use the Amiga system functions AllocMem and FreeMem which are like the built-in function New and Dispose, but the memory allocated by AllocMem must be deallocated (using FreeMem) when it's finished with, before the end of the program.

```
CONST SMALL=100, BIG=123456789

ENUM ERR_MEM=1

RAISE ERR_MEM IF AllocMem()=NIL

PROC main()
  allocate()
ENDPROC

PROC allocate() HANDLE
  DEF mem=NIL
  mem:=AllocMem(SMALL, 0)
  morealloc()
  FreeMem(mem, SMALL)
EXCEPT
  IF mem THEN FreeMem(mem, SMALL)
  WriteF('Handler: deallocating "allocate" local memory\n')
ENDPROC
```

```
PROC morealloc() HANDLE
  DEF more=NIL, andmore=NIL
  more:=AllocMem(SMALL, 0)
  andmore:=AllocMem(BIG, 0)
  WriteF('Allocated all the memory!\n')
  FreeMem(andmore, BIG)
  FreeMem(more, SMALL)
EXCEPT
  IF andmore THEN FreeMem(andmore, BIG)
  IF more THEN FreeMem(more, SMALL)
  WriteF('Handler: deallocating "morealloc" local memory\n')
  Raise(ERR_MEM)
ENDPROC
```

The calls to AllocMem are automatically checked, and if NIL is returned the exception ERR_MEM is raised. The handler in the allocate procedure checks to see if it needs to free the memory pointed to by mem, and the handler in the morealloc checks andmore and more. At the end of the morealloc handler is the call Raise(ERR_MEM). This passes control to the exception handler of the allocate procedure, since allocate called morealloc.

There's a couple of subtle points to notice about this example. Firstly, the memory variables are all initialised to NIL. This is because the automatic exception raising on AllocMem will result in the variables not being assigned if the call returns NIL (i.e., the exception is raised before the assignment takes place), and the handler needs them to be NIL if AllocMem fails. Of course, if AllocMem does not return NIL the assignments work as normal.

Secondly, the IF statements in the handlers check the memory pointer variables do not contain NIL by using their values as truth values. Since NIL is actually zero, a non-NIL pointer will be non-zero, i.e., true in the IF check. This shorthand is often used, and so you should be aware of it.

It is quite common that an exception handler will want to raise the same exception after it has done its processing. The function ReThrow (which has no arguments) can be used for this purpose. It will re-raise the exception, but only if the exception is not zero (since this special value means that no error occurred). If the exception is zero then this function has no effect. In fact, the following code fragments (within a handler) are equivalent:

```
ReThrow()

IF exception THEN Throw(exception, exceptioninfo)
```

There are two examples, in Part Three, of how to use an exception handler to make a program more readable: one deals with using data files (see String Handling and I-O) and the other deals with opening screens and windows (see Screens).