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 Introduction to the Examples

```
Introduction to the Examples
****************************

   In this part we shall go through some slightly larger examples than
those in the previous parts.  However, none of them are too big, so they
should still be easy to understand.  The note-worthy parts of each example
are described, and you may even find the odd comment in the code.  Large,
complicated programs benefit hugely from the odd well-placed and
descriptive comment.  This fact can't be stressed enough.

   All the examples will run on a standard Amiga, except for the one which
uses ReadArgs (an AmigaDOS 2.0 function).  It is really worth upgrading
your system to AmigaDOS 2.0 (or above) if you are still using previous
versions.  The ReadArgs example can only hint at the power and
friendliness of the newer system functions.  If you are fortunate enough
to have an A4000 or an accelerated machine, then the timing example will
give better (i.e., quicker) results.

   Supplied with this Guide should be a directory of sources of most of
the examples.  Here's a complete catalogue:

simple.e
     The simple program from the introduction.  See A Simple Program.

while.e
     The slightly complicated WHILE loop.  See WHILE loop.

address.e
     The program which prints the addresses of some variables.  See
     Finding addresses (making pointers).

static.e
     The static data problem.  See Static data.

static2.e
     The first solution to the static data problem.  See Static data.

except.e
```

An exception handler example. See Raising an Exception.

except2.e
Another exception handler example. See Raising an Exception.

static3.e
The second solution to the static data problem, using NEW. See List and typed list allocation.

float.e
The floating-point example program. See Floating-Point Functions.

bintree.e
The binary tree example. See Binary Trees.

tree.e
The tree and integer_tree classes, as a module. See Inheritance in E.

tree-use.e
A program to use the integer_tree class. See Inheritance in E.

set.e
The simple, inefficient set class, as a module. See Data-Hiding in E.

set-use.e
A program to use the set class. See Data-Hiding in E.

csv-estr.e
The CSV reading program using E-strings. See String Handling and I-O.

csv-norm.e
The CSV reading program using normal strings. See String Handling and I-O.

csv-buff.e
The CSV reading program using normal strings and a large buffer. See String Handling and I-O.

csv.e
The CSV reading program using normal strings, a large buffer, and an exception handler. See String Handling and I-O.

timing.e
The timing example. See Timing Expressions.

args.e
The argument parsing example for any AmigaDOS. See Any AmigaDOS.

args20.e
The argument parsing example for any AmigaDOS 2.0 and above. See AmigaDOS 2.0 (and above).

gadgets.e
The gadgets example. See Gadgets.

idcmp.e
The IDCMP and gadgets example. See IDCMP Messages.

```
graphics.e
     The graphics example.  See Graphics.

screens.e
     The screens example, without an exception handler.  See Screens.

screens2.e
     The screens example again, but this time with an exception handler.
     See Screens.

dragon.e
     The dragon curve recursion example.  See Recursion Example.
```

1.2 String Handling and I-O

```
String Handling and I/O
***********************

   This chapter shows how to use normal strings and E-strings, and also
how to read data from a file.  The programs use a number of the string
functions and make effective (but different) use of memory where possible.
The key points to understand are:

   * The difference between normal strings and E-strings.

   * The two methods of reading data from a file (line-by-line or all at
     once).

   * The necessary allocation of memory for E-strings.

   * The unnecessary, but advisable, deallocation of the E-string memory
     once it is no longer needed.  The deallocation could be left to the
     automatic deallocation at the end of the program, but that would
     waste an increasing amount of memory whilst the program was running.
     If the input data was large then memory could easily be exhausted.

   * The way in which sections of an E-string (or a normal string, for
     that matter) can easily be turned into normal strings.

   * The way exception handlers can tidy up programs.

   The problem to solve is reading of a CSV (comma separated variables)
file, which is a standard format file for databases and spreadsheets.  The
format is very simple: each record is a line (i.e., terminated with a
line-feed) and each field in a record is separated by a comma.  To make
this example a lot simpler, we will forbid a field to contain a comma
(normally this would require the field to be quoted).  So, a typical input
file would look like this:

     Field1,Field2,Field3
     10,19,-3
     fred,barney,wilma
     ,,last
```

```
first,,
```

In this example all records have three fields, as is well illustrated by the first line (i.e., the first record). The last two records may seem a bit strange, but they just show how fields can be blank. In the last record all but the first field are blank, and in the previous record all but the last are blank.

So now we know the format of the file to be read. To operate on a file we must first open it using the Open function (from the dos.library), and to read the lines from the file we will use the ReadStr (built-in) function. There will be four versions of a program to read a CSV file: two of which read data line-by-line and two which read all the file at once. Of the two which read line-by-line, one manipulates the read lines as E-strings and the other uses normal strings. The use of normal strings is arguably more advanced than the use of E-strings, since cunning tricks are employed to make effective use of memory. However, the programs are not meant to show that E-strings are better than normal strings (or vice versa), rather they are meant to show how to use strings properly.

```
/* A suitably large size for the record buffer */
CONST BUFFERSIZE=512

PROC main()
  DEF filehandle, status, buffer[BUFFERSIZE]:STRING, filename
  filename:='datafile'
  IF filehandle:=Open(filename, OLDFILE)
    REPEAT
      status:=ReadStr(filehandle, buffer)
      /* This is the way to check ReadStr() actually read something */
      IF buffer[] OR (status<>-1) THEN process_record(buffer)
    UNTIL status=-1
    /* If Open() succeeded then we must Close() the file */
    Close(filehandle)
  ELSE
    WriteF('Error: Failed to open "\s"\n', filename)
  ENDIF
ENDPROC

PROC process_record(line)
  DEF i=1, start=0, end, len, s
  /* Show the whole line being processed */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Find the index of a comma after the start index */
    end:=InStr(line, ',', start)
    /* Length is end index minus start index */
    len:=(IF end<>-1 THEN end ELSE EstrLen(line))-start
    IF len>0
      /* Allocate an E-string of the correct length */
      IF s:=String(len)
        /* Copy the portion of the line to the E-string s */
        MidStr(s, line, start, len)
        /* At this point we could do something useful... */
        WriteF('\t\d) "\s"\n', i, s)
        /* We've finished with the E-string so deallocate it */
        DisposeLink(s)
```

```
        ELSE
          /* It's a non-fatal error if the String() call fails */
          WriteF('\t\d) Memory exhausted! (len=\d)\n', len)
        ENDIF
      ELSE
        WriteF('\t\d) Empty Field\n', i)
      ENDIF
      /* The new start is after the end we found */
      start:=end+1
      INC i
    /* Once a comma is not found we've finished */
    UNTIL end=-1
  ENDPROC
```

There are a couple of points worth noting about this program:

* A large E-string, buffer, is used to hold each line before it is processed. If a record exceeds the size of this E-string then ReadStr will only read a partial record, and the next ReadStr will read some more this record. However, the program considers each call to ReadStr to read a whole record, so it will get the records slightly wrong in this case. This is a limitation of the program and it should be documented so that users know to constrain themselves to datafiles without long lines.

* The file name is 'hard-wired' to be datafile. A more flexible program would allow this to be passed as an argument (see Argument Parsing).

* ReadStr may return -1 to indicate an error (usually when the end of the file has been reached), but the E-string read so far may still be valid. The check on the E-string and error value is the proper way of deciding whether ReadStr actually read anything from the file.

* Look carefully at the manipulation of the string indexes start and end, and the calculation of the length of a portion of a string.

* MidStr is used to copy a field from a record, so an E-string must be used to hold the field.

* The E-string s is only valid between the successful allocation by string and the DisposeLink. It would be incorrect to try to, for instance, print it at any other point. On the other hand, a more complicated program may want to store up all the data, and so it may be inappropriate to deallocate the E-string at this point. In this case, the pointer to the E-string could be stored and it might be valid for the rest of the program.

* The allocation using String is very closely followed by deallocation using DisposeLink. This suggests that a single E-string could be allocated and used repeatedly (like buffer is), due to the simple nature of this example.

To change this to use normal strings (in a very memory efficient way), we need to alter only the process_record procedure. Some note-worthy differences are:

* Small parts of the E-string buffer are turned into normal strings by terminating them with NIL when necessary. This involves changing a comma that is found.

* No new memory is allocated, rather the buffer memory is reused (as described above). This is fine for this example, although if the fields were needed after a record had been processed they would need to be copied, since the contents of buffer are changed by ReadStr.

```
PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Show the whole line being processed */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Find the index of a comma after the start index */
    end:=InStr(line, ',', start)
    /* If a comma was found then terminate with a NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Point to the start of the field */
    s:=line+start
    IF s[]
      /* At this point we could do something useful... */
      WriteF('\t\d) "\s"\n', i, s)
    ELSE
      WriteF('\t\d) Empty Field\n', i)
    ENDIF
    /* The new start is after the end we found */
    start:=end+1
    INC i
  /* Once a comma is not found we've finished */
  UNTIL end=-1
ENDPROC
```

The next two versions of the program are basically the same: they both read the whole file into one large, dynamically allocated buffer and then process the data. The second of the two versions also uses exceptions to make the program much more readable. The differences from the above version which uses normal strings are:

* The main procedure calculates the length of the data in the file and then uses New to dynamically allocate some memory to hold it.

* The read data is terminated with a NIL so that it can safely be treated as a (very long) normal string.

* The process_buffer procedure splits the read data up into lots of normal strings, one for each line of data.

```
PROC main()
  DEF buffer, filehandle, len, filename
  filename:='datafile'
  /* Get the length of data in the file */
  IF 0<(len:=FileLength(filename))
    /* Allocate just enough room for the data + a terminating NIL */
    IF buffer:=New(len+1)
      IF filehandle:=Open(filename, OLDFILE)
        /* Read whole file, checking amount read */
```

```
        IF len=Read(filehandle, buffer, len)
          /* Terminate buffer with a NIL just in case... */
          buffer[len]:=NIL
          process_buffer(buffer, len)
        ELSE
          WriteF('Error: File reading error\n')
        ENDIF
        /* If Open() succeeded then we must Close() the file */
        Close(filehandle)
      ELSE
        WriteF('Error: Failed to open "\s"\n', filename)
      ENDIF
      /* Deallocate buffer (not really necessary in this example) */
      Dispose(buffer)
    ELSE
      WriteF('Error: Insufficient memory to load file\n')
    ENDIF
  ELSE
    WriteF('Error: "\s" is an empty file\n', filename)
  ENDIF
ENDPROC

/* buffer is like a normal string since it's NIL-terminated */
PROC process_buffer(buffer, len)
  DEF start=0, end
  REPEAT
    /* Find the index of a linefeed after the start index */
    end:=InStr(buffer, '\n', start)
    /* If a linefeed was found then terminate with a NIL */
    IF end<>-1 THEN buffer[end]:=NIL
    process_record(buffer+start)
    start:=end+1
  /* We've finished if at the end or no more linefeeds */
  UNTIL (start>=len) OR (end=-1)
ENDPROC

PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Show the whole line being processed */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Find the index of a comma after the start index */
    end:=InStr(line, ',', start)
    /* If a comma was found then terminate with a NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Point to the start of the field */
    s:=line+start
    IF s[]
      /* At this point we could do something useful... */
      WriteF('\t\d) "\s"\n', i, s)
    ELSE
      WriteF('\t\d) Empty Field\n', i)
    ENDIF
    /* The new start is after the end we found */
    start:=end+1
    INC i
  /* Once a comma is not found we've finished */
```

```
    UNTIL end=-1
  ENDPROC
```

The program is now quite messy, with many error cases in the main procedure. We can very simply change this by using an exception handler and a few automatic exceptions.

```
  /* Some constants for exceptions (ERR_NONE is zero: no error) */
  ENUM ERR_NONE, ERR_LEN, ERR_NEW, ERR_OPEN, ERR_READ

  /* Make some exceptions automatic */
  RAISE ERR_LEN  IF FileLength()<=0,
        ERR_NEW  IF New()=NIL,
        ERR_OPEN IF Open()=NIL

  PROC main() HANDLE
    /* Note the careful initialisation of buffer and filehandle */
    DEF buffer=NIL, filehandle=NIL, len, filename
    filename:='datafile'
    /* Get the length of data in the file */
    len:=FileLength(filename)
    /* Allocate just enough room for the data + a terminating NIL */
    buffer:=New(len+1)
    filehandle:=Open(filename, OLDFILE)
    /* Read whole file, checking amount read */
    IF len<>Read(filehandle, buffer, len) THEN Raise(ERR_READ)
    /* Terminate buffer with a NIL just in case... */
    buffer[len]:=NIL
    process_buffer(buffer, len)
  EXCEPT DO
    /* Both of these are safe thanks to the initialisations */
    IF buffer THEN Dispose(buffer)
    IF filehandle THEN Close(filehandle)
    /* Report error (if there was one) */
    SELECT exception
    CASE ERR_LEN;   WriteF('Error: "\s" is an empty file\n', filename)
    CASE ERR_NEW;   WriteF('Error: Insufficient memory to load file\n')
    CASE ERR_OPEN;  WriteF('Error: Failed to open "\s"\n', filename)
    CASE ERR_READ;  WriteF('Error: File reading error\n')
    ENDSELECT
  ENDPROC
```

The code is now much clearer, and the majority of errors can be caught automatically. Notice that the exception handler is called even if the program succeeds (thanks to the DO after the EXCEPT). This is because when the program terminates it needs to deallocate the resources it allocated in every case (successful or otherwise), so the code is the same. Conditional deallocation (of the buffer, for example) is made safe by an appropriate initialisation.

If you feel like a small exercise, try to write a similar program but this time using the tools/file module which comes in the standard Amiga E distribution. Of course, you'll first need to read the accompanying documentation, but you should find that this module makes file interaction very simple.

1.3 Timing Expressions

Timing Expressions
****

You may recall the outline of a timing procedure in Part Two (see Evaluation). This chapter gives the complete version of this example. The information missing from the outline was how to determine the system time and use this to calculate the time taken by calls to Eval. So the things to notice about this example are:

* Use of the Amiga system function DateStamp (from the dos.library). (You really need the 'Rom Kernel Reference Manuals' and the 'AmigaDOS Manual' to understand the system functions.)

* Use of the module dos/dos to include the definitions of the object datestamp and the constant TICKS_PER_SECOND. (There are fifty ticks per second.)

* Use of the repeat procedure to do Eval a decent number of times for each expression (so that some time is taken up by the calls!).

* The timing of the evaluation of 0, to calculate the overhead of the procedure calls and loop. This value is stored in the variable offset the first time the test procedure is called. The expression 0 should take a negligible amount of time, so the number of ticks timed is actually the time taken by the procedure calls and loop calculations. Subtracting this time from the other times gives a fair view of how long the expressions take, relative to one another. (Thanks to Wouter for this offset idea.)

* Use of Forbid and Permit to turn off multi-tasking temporarily, making the CPU calculate only the expressions (rather than dealing with screen output, other programs, etc.).

* Use of CtrlC and CleanUp to allow the user to stop the program if it gets too boring...

* Use of the option LARGE (using OPT) to produce an executable that uses the large data and code model. This seems to help make the timings less susceptible variations due to, for instance, optimisations, and so better for comparison. See the 'Reference Manual' for more details.

Also supplied are some example outputs. The first was from an A1200 with 2MB Chip RAM and 4MB Fast RAM. The second was from an A500Plus with 2MB Chip RAM. Both used the constant LOTS_OF_TIMES as 500,000, but you might need to increase this number to compare, for instance, an A4000/040 to an A4000/030. However, 500,000 gives a pretty long wait for results on the A500.

```
MODULE 'dos/dos'

CONST TICKS_PER_MINUTE=TICKS_PER_SECOND*60, LOTS_OF_TIMES=500000

DEF x, y, offset
```

```
PROC fred(n)
  DEF i
  i:=n+x
ENDPROC

/* Repeat evaluation of an expression */
PROC repeat(exp)
  DEF i
  FOR i:=0 TO LOTS_OF_TIMES
    Eval(exp)  /* Evaluate the expression */
  ENDFOR
ENDPROC

/* Time an expression, and set-up offset if not done already */
PROC test(exp, message)
  DEF t
  IF offset=0 THEN offset:=time(`0)  /* Calculate offset */
  t:=time(exp)
  WriteF('\s:\t\d ticks\n', message, t-offset)
ENDPROC

/* Time the repeated calls, and calculate number of ticks */
PROC time(x)
  DEF ds1:datestamp, ds2:datestamp
  Forbid()
  DateStamp(ds1)
  repeat(x)
  DateStamp(ds2)
  Permit()
  IF CtrlC() THEN CleanUp(1)
ENDPROC ((ds2.minute-ds1.minute)*TICKS_PER_MINUTE)+ds2.tick-ds1.tick

PROC main()
  x:=9999
  y:=1717
  test(`x+y,     'Addition')
  test(`y-x,     'Subtraction')
  test(`x*y,     'Multiplication')
  test(`x/y,     'Division')
  test(`x OR y,  'Bitwise OR')
  test(`x AND y, 'Bitwise AND')
  test(`x=y,     'Equality')
  test(`x<y,     'Less than')
  test(`x<=y,    'Less than or equal')
  test(`y:=1,    'Assignment of 1')
  test(`y:=x,    'Assignment of x')
  test(`y++,     'Increment')
  test(`IF FALSE THEN y ELSE x, 'IF FALSE')
  test(`IF TRUE THEN y ELSE x,  'IF TRUE')
  test(`IF x THEN y ELSE x,     'IF x')
  test(`fred(2),  'fred(2)')
ENDPROC
```

Here's the output from the A1200:

```
Addition:  22 ticks
```

```
     Subtraction: 22 ticks
     Multiplication:  69 ticks
     Division:  123 ticks
     Bitwise OR:  33 ticks
     Bitwise AND: 27 ticks
     Equality:  44 ticks
     Less than: 43 ticks
     Less than or equal:  70 ticks
     Assignment of 1: 9 ticks
     Assignment of x: 38 ticks
     Increment: 23 ticks
     IF FALSE:  27 ticks
     IF TRUE: 38 ticks
     IF x:  44 ticks
     fred(2): 121 ticks

Compare this to the output from the A500Plus:

     Addition:  118 ticks
     Subtraction: 117 ticks
     Multiplication:  297 ticks
     Division:  643 ticks
     Bitwise OR:  118 ticks
     Bitwise AND: 117 ticks
     Equality:  164 ticks
     Less than: 164 ticks
     Less than or equal:  164 ticks
     Assignment of 1: 60 ticks
     Assignment of x: 102 ticks
     Increment: 134 ticks
     IF FALSE:  118 ticks
     IF TRUE: 164 ticks
     IF x:  193 ticks
     fred(2): 523 ticks

Evidence, if it were needed, that the A1200 is roughly five times faster
than an A500, and that's not using the special 68020 CPU instructions!
```

1.4 Argument Parsing

```
Argument Parsing
****************

   There are two examples in this chapter.  One is for any AmigaDOS and
the other is for AmigaDOS 2.0 and above.  They both illustrate how to
parse the arguments to your program.  If your program is started from the
Shell/CLI the arguments follow the command name on the command line, but
if it was started from Workbench (i.e., you double-clicked on an icon for
the program) then the arguments are those icons that were also selected at
that time (see your Workbench manual for more details).


  Any AmigaDOS
  AmigaDOS 2.0 (and above)
```

1.5 Any AmigaDOS

```
Any AmigaDOS
============
```

This first example works with any AmigaDOS. The first thing that is done is the assignment of wbmessage to a correctly typed pointer. At the same time we can check to see if it is NIL (i.e., whether the program was started from Workbench or not). If it was not started from Workbench the arguments in arg are printed. Otherwise we need to use the fact that wbmessage is really a pointer to a wbstartup object (defined in module workbench/startup), so we can get at the argument list. Then for each argument in the list we need to check the lock supplied with the argument. If it's a proper lock it will be a lock on the directory containing the argument file. The name in the argument is just a filename, not a complete path, so to read the file we need to change the current directory to the lock directory. Once we've got a valid lock and we've changed directory to there, we can find the length of the file (using FileLength) and print it. If there was no lock or the file did not exist, the name of the file and an appropriate error message is printed.

```
MODULE 'workbench/startup'

PROC main()
  DEF startup:PTR TO wbstartup, args:PTR TO wbarg, i, oldlock, len
  IF (startup:=wbmessage)=NIL
    WriteF('Started from Shell/CLI\n   Arguments: "\s"\n', arg)
  ELSE
    WriteF('Started from Workbench\n')
    args:=startup.arglist
    FOR i:=1 TO startup.numargs  /* Loop through the arguments */
      IF args[].lock=NIL
        WriteF('  Argument \d: "\s" (no lock)\n', i, args[].name)
      ELSE
        oldlock:=CurrentDir(args[].lock)
        len:=FileLength(args[].name)  /* Do something with file */
        IF len=-1
          WriteF('  Argument \d: "\s" (file does not exist)\n',
                 i, args[].name)
        ELSE
          WriteF('  Argument \d: "\s", file length is \d bytes\n',
                 i, args[].name, len)
        ENDIF
        CurrentDir(oldlock) /* Important: restore current dir */
      ENDIF
      args++
    ENDFOR
  ENDIF
ENDPROC
```

When you run this program you'll notice a slight difference between arg and the Workbench message: arg does not contain the program name, just the arguments, whereas the first argument in the Workbench argument list is

the program. You can simply ignore the first Workbench argument in the list if you want.

1.6 AmigaDOS 2.0 (and above)

AmigaDOS 2.0 (and above)
========================

   This second program can be used as the Shell/CLI part of the previous program to provide much better command line parsing. It can only be used with AmigaDOS 2.0 and above (i.e., OSVERSION which is 37 or more). The template FILE/M used with ReadArgs gives command line parsing similar to C's argv array. The template can be much more interesting than this, but for more details you need the 'AmigaDOS Manual'.

```
     OPT OSVERSION=37

     PROC main()
       DEF templ, rdargs, args=NIL:PTR TO LONG, i
       IF wbmessage=NIL
         WriteF('Started from Shell/CLI\n')
         templ:='FILE/M'
         rdargs:=ReadArgs(templ,{args},NIL)
         IF rdargs
           IF args
             i:=0
             WHILE args[i]  /* Loop through arguments */
               WriteF('   Argument \d: "\s"\n', i, args[i])
               i++
             ENDWHILE
           ENDIF
           FreeArgs(rdargs)
         ENDIF
       ENDIF
     ENDPROC
```

As you can see the result of the ReadArgs call with this template is an array of filenames. The special quoting of filenames is dealt with correctly (i.e., when you use " around a filename that contains spaces). You need to do all this kind of work yourself if you use the arg method.

1.7 Gadgets IDCMP and Graphics

Gadgets, IDCMP and Graphics
****

   There are three examples in this chapter. The first shows how to open a window and put some gadgets on it. The second shows how to decipher Intuition messages that arrive via IDCMP. The third draws things with the graphics functions.

1.8 Gadgets

Gadgets
=======

The following program illustrates how to create a gadget list and use it:

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = 4 * GADGETSIZE

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, next, wptr
  next:=Gadget(buf,  NIL, 1, 0, 10, 30, 50, 'Hello')
  next:=Gadget(next, buf, 2, 3, 70, 30, 50, 'World')
  next:=Gadget(next, buf, 3, 1, 10, 50, 50, 'from')
  next:=Gadget(next, buf, 4, 0, 70, 50, 70, 'gadgets')
  wptr:=OpenW(20,50,200,100, 0, WFLG_ACTIVATE,
              'Gadgets in a window',NIL,1,buf)
  IF wptr         /* Check to see we opened a window */
    Delay(500)    /* Wait a bit */
    CloseW(wptr)  /* Close the window */
  ELSE
    WriteF('Error -- could not open window!')
  ENDIF
ENDPROC
```

Four gadgets are created using an appropriately sized array as the buffer. These gadgets are passed to OpenW (the last parameter). If the window could be opened a small delay is used so that the window is visible before the program closes it and terminates. Delay is an Amiga system function from the DOS library, and Delay(n) waits n/50 seconds. Therefore, the window stays up for 10 seconds, which is enough time to play with the gadgets and see what the different types are. The next example will show a better way of deciding when to terminate the program (using the standard close gadget).

1.9 IDCMP Messages

IDCMP Messages
==============

This next program shows how to use WaitIMessage with a gadget.

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = GADGETSIZE, OURGADGET = 1

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, wptr, class, gad:PTR TO gadget
  Gadget(buf, NIL, OURGADGET, 1, 10, 30, 100, 'Press Me')
  wptr:=OpenW(20,50,200,100,
              IDCMP_CLOSEWINDOW OR IDCMP_GADGETUP,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Gadget message window',NIL,1,buf)
  IF wptr              /* Check to see we opened a window */
    WHILE (class:=WaitIMessage(wptr))<>IDCMP_CLOSEWINDOW
      gad:=MsgIaddr()  /* Our gadget clicked? */
      IF (class=IDCMP_GADGETUP) AND (gad.userdata=OURGADGET)
        TextF(10,60,
              IF gad.flags=0 THEN 'Gadget off ' ELSE 'Gadget on   ')
      ENDIF
    ENDWHILE
    CloseW(wptr)       /* Close the window */
  ELSE
    WriteF('Error -- could not open window!')
  ENDIF
ENDPROC
```

The gadget reports its state when you click on it, using the TextF function (see Graphics functions). The only way to quit the program is using the close gadget of the window. The gadget object is defined in the module intuition/intuition and the iaddr part of the IDCMP message is a pointer to our gadget if the message was a gadget message. The userdata element of the gadget identifies the gadget that was clicked, and the flags element is zero if the boolean gadget is off (unselected) or non-zero if the boolean gadget is on (selected).

1.10 Graphics

Graphics
========

The following program illustrates how to use the various graphics functions.

```
MODULE 'intuition/intuition'

PROC main()
  DEF wptr, i
  wptr:=OpenW(20,50,200,100,IDCMP_CLOSEWINDOW,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Graphics demo window',NIL,1,NIL)
  IF wptr  /* Check to see we opened a window */
    Colour(1,3)
    TextF(20,30,'Hello World')
```

```
        SetTopaz(11)
        TextF(20,60,'Hello World')
        FOR i:=10 TO 150 STEP 8  /* Plot a few points */
          Plot(i,40,2)
        ENDFOR
        Line(160,40,160,70,3)
        Line(160,70,170,40,2)
        Box(10,75,160,85,1)
        WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
        ENDWHILE
        CloseW(wptr)
      ELSE
        WriteF('Error -- could not open window!\n')
      ENDIF
    ENDPROC
```

First of all a small window is opened with a close gadget and activated (so it is the selected window). Clicks on the close gadget will be reported via IDCMP, and this is the only way to quit the program. The graphics functions are used as follows:

* Colour is used to set the foreground colour to pen one and the background colour to pen three. This will make the text nicely highlighted.

* Text is output in the standard font.

* The font is set to Topaz 11.

* More text is output (probably now in a different font).

* The FOR loop plots a dotted line in pen two.

* A vertical line in pen three is drawn.

* A diagonal line in pen two is drawn. This and the previous line together produce a vee shape.

* A filled box is drawn in pen one.

1.11 Screens

```
Screens
=======
```

This next example uses parts of the previous example, but also opens a custom screen. Basically, it draws coloured lines and boxes in a big window opened on a 16 colour, high resolution screen.

```
    MODULE 'intuition/intuition', 'graphics/view'

    PROC main()
      DEF sptr=NIL, wptr=NIL, i
      sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
```

```
  IF sptr
    wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
                WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
                'Graphics demo window',sptr,$F,NIL)
    IF wptr
      TextF(20,20,'Hello World')
      FOR i:=0 TO 15  /* Draw a line and box in each colour */
        Line(20,30,620,30+(7*i),i)
        Box(10+(40*i),140,30+(40*i),170,1)
        Box(11+(40*i),141,29+(40*i),169,i)
      ENDFOR
      WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
      ENDWHILE
      WriteF('Program finished successfully\n')
    ELSE
      WriteF('Could not open window\n')
    ENDIF
  ELSE
    WriteF('Could not open screen\n')
  ENDIF
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
ENDPROC
```

As you can see, the error-checking IF blocks can make the program hard to read. Here's the same example written with an exception handler:

```
MODULE 'intuition/intuition', 'graphics/view'

ENUM WIN=1, SCRN

RAISE WIN  IF OpenW()=NIL,
      SCRN IF OpenS()=NIL

PROC main() HANDLE
  DEF sptr=NIL, wptr=NIL, i
  sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
  wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Graphics demo window',sptr,$F,NIL)
  TextF(20,20,'Hello World')
  FOR i:=0 TO 15  /* Draw a line and box in each colour */
    Line(20,30,620,30+(7*i),i)
    Box(10+(40*i),140,30+(40*i),170,1)
    Box(11+(40*i),141,29+(40*i),169,i)
  ENDFOR
  WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
  ENDWHILE
EXCEPT DO
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
  SELECT exception
  CASE 0
    WriteF('Program finished successfully\n')
  CASE WIN
    WriteF('Could not open window\n')
  CASE SCRN
```

```
        WriteF('Could not open screen\n')
      ENDSELECT
    ENDPROC
```

It's much easier to see what's going on here. The real part of the program (the bit before the EXCEPT) is no longer cluttered with error checking, and it's easy to see what happens if an error occurs. Notice that if the program successfully finishes it still has to close the screen and window properly, so it's often sensible to use EXCEPT DO to raise a zero exception and deal with all the tidying up in the handler.

1.12 Recursion Example

```
Recursion Example
*****************
```

This next example uses a pair of mutually recursive procedures to draw what is known as a dragon curve (a pretty, space-filling pattern).

```
    MODULE 'intuition/intuition', 'graphics/view'

    /* Screen size, use SIZEY=512 for a PAL screen */
    CONST SIZEX=640, SIZEY=400

    /* Exception values */
    ENUM WIN=1, SCRN, STK, BRK

    /* Directions (DIRECTIONS gives number of directions) */
    ENUM NORTH, EAST, SOUTH, WEST, DIRECTIONS

    RAISE WIN  IF OpenW()=NIL,
          SCRN IF OpenS()=NIL

    /* Start off pointing WEST */
    DEF state=WEST, x, y, t

    /* Face left */
    PROC left()
      state:=Mod(state-1+DIRECTIONS, DIRECTIONS)
    ENDPROC

    /* Move right, changing the state */
    PROC right()
      state:=Mod(state+1, DIRECTIONS)
    ENDPROC

    /* Move in the direction we're facing */
    PROC move()
      SELECT state
      CASE NORTH; draw(0,t)
      CASE EAST;  draw(t,0)
      CASE SOUTH; draw(0,-t)
      CASE WEST;  draw(-t,0)
      ENDSELECT
```

```
ENDPROC

/* Draw and move to specified relative position */
PROC draw(dx, dy)
  /* Check the line will be drawn within the window bounds */
  IF (x>=Abs(dx)) AND (x<=SIZEX-Abs(dx)) AND
     (y>=Abs(dy)) AND (y<=SIZEY-10-Abs(dy))
    Line(x, y, x+dx, y+dy, 2)
  ENDIF
  x:=x+dx
  y:=y+dy
ENDPROC

PROC main() HANDLE
  DEF sptr=NIL, wptr=NIL, i, m
  /* Read arguments:        [m [t [x  [y]]]] */
  /* so you can say: dragon  16              */
  /*             or: dragon  16 1            */
  /*             or: dragon  16 1 450        */
  /*             or: dragon  16 1 450 100    */
  /* m is depth of dragon, t is length of lines */
  /* (x,y) is the start position */
  m:=Val(arg, {i})
  t:=Val(arg:=arg+i, {i})
  x:=Val(arg:=arg+i, {i})
  y:=Val(arg:=arg+i, {i})
  /* If m or t is zero use a more sensible default */
  IF m=0 THEN m:=5
  IF t=0 THEN t:=5
  sptr:=OpenS(SIZEX,SIZEY,4,V_HIRES OR V_LACE,'Dragon Curve Screen')
  wptr:=OpenW(0,10,SIZEX,SIZEY-10,
              IDCMP_CLOSEWINDOW,WFLG_CLOSEGADGET,
              'Dragon Curve Window',sptr,$F,NIL)
  /* Draw the dragon curve */
  dragon(m)
  WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
  ENDWHILE
EXCEPT DO
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
  SELECT exception
  CASE 0
    WriteF('Program finished successfully\n')
  CASE WIN
    WriteF('Could not open window\n')
  CASE SCRN
    WriteF('Could not open screen\n')
  CASE STK
    WriteF('Ran out of stack in recursion\n')
  CASE BRK
    WriteF('User aborted\n')
  ENDSELECT
ENDPROC

/* Draw the dragon curve (with left) */
PROC dragon(m)
  /* Check stack and ctrl-C before recursing */
```

```
    IF FreeStack()<1000 THEN Raise(STK)
    IF CtrlC() THEN Raise(BRK)
    IF m>0
      dragon(m-1)
      left()
      nogard(m-1)
    ELSE
      move()
    ENDIF
  ENDPROC

  /* Draw the dragon curve (with right) */
  PROC nogard(m)
    IF m>0
      dragon(m-1)
      right()
      nogard(m-1)
    ELSE
      move()
    ENDIF
  ENDPROC
```

If you write this to the file dragon.e and compile it to the executable dragon then some good things to try are:

```
dragon 5 9 300 100
dragon 10 4 250 250
dragon 11 3 250 250
dragon 15 1 300 100
dragon 16 1 400 150
```

If you want to understand how the program works you need to study the recursive parts. Here's an overview of the program, outlining the important aspects:

* The constants SIZEX and SIZEY are the width and height (respectively) of the custom screen (and window). As the comment suggests, change SIZEY to 512 if you want a bigger screen and you have a PAL Amiga.

* The state variable holds the current direction (north, south, east or west).

* The left and right procedures turn the current direction to the left and right (respectively) by using some modulo arithmetic trickery.

* The move procedure uses the draw procedure to draw a line (of length t) in the current direction from the current point (stored in x and y).

* The draw procedure draws a line relative to the current point, but only if it fits within the boundaries of the window. The current point is moved to the end of the line (even if it isn't drawn).

* The main procedure reads the command line arguments into the variables m, t, x and y. The depth/size of the dragon is given by m (the first argument) and the length of each line making up the dragon is given by t (the second argument). The starting point is given by

x and y (the final two arguments). The defaults are five for m and t, and zero for x and y.

* The main procedure also opens the screen and window, and sets the dragon drawing.

* The dragon and nogard procedures are very similar, and these are responsible for creating the dragon curve by calling the left, right and move procedures.

* The dragon procedure contains a couple of checks to see if the user has pressed Control-C or if the program has run out of stack space, raising an appropriate exception if necessary. These exceptions are handled by the main procedure.

Notice the use of Val and the exception handling. Also, the important base case of the recursion is when m reaches zero (or becomes negative, but that shouldn't happen). If you start off a big dragon and want to stop it you can press Control-C and the program tidies up nicely. If it has finished drawing you simply click the close gadget on the window.