Howard's OBJECTS

Howard Richoux
6721 Shamrock Rd.
Lincoln, NE 68506-2821
(402) 488-5867
CSERVE 76350,1772

Introduction

I am a medium old-fashioned programmer. I started 25 years ago, and programmed
than I like to remember. I learned not to use GOTOs, but somewhere in the 80s I
techniques and concentrated on simply making things that worked.

Turbo Pascal is the best all-around language I have ever used. When they introd
monolithic program, and made it practical to re-use code. The concept of Object
"virtual methods" it lost me.

My use of objects is both primitive and extremely useful. I focus on the abilit

making complex functions appear to be simple. One of the things that was always
instance of whatever. It was straight-forward to code, say, the first use of a
always involved global variables which had to be used simultaneously to track tw
object allowed simultaneous use almost trivially.

An object is also the best place to hide truely ugly code. We all have it. The
with cleanup later (hopefully). I hope that the ugliest of the code is gone be
first time, but if not, I apologize in advance.

Personal Bias

Code is personal, and reflects the biases of the author. I have a few, and I wi

1. Efficient code is nice, but not terribly important. I try not to
   waste CPU cycles, but hardware gets better by a factor of 2 each
   year, and I improve by maybe 10%. The choice is obvious.

2. Code size and EXE size is a little more important, but again, not
   critical. With all the configuration and output support routines,
   my "Hello world." size is around 15k. A seriously useful utility
   can be written in 100 lines or so of non-library code and take
   >25k. To me, this is good. I have done what I can to keep the
   size down, but I think the situation is about optimum now. I welcome
   improvements down at the lowest level because they give the
   greatest returns, but thorough testing is in order.

3. I use minimum comments in the code. My preference is to name
   routines and variables sensibly, and only document the subtle
   points. Comments on code lines tend to hide structure.

4. My development system is a 486DX2-66, 16Mb ram, 450Mb disk and
   LaserJet IV printer. I have access to smaller systems, but my
   smallest is a 386-40. Obviously, this affects my code. Most
   specifically are instances of sending LJ escape sequences to
   the printer. I hope to cut out and make optional, all printer
   specific strings, but the first release may still have some.

On To Objects

The Objects which I use most are as follows:

1. STR_object - consists of a Pascal string (1-255 chars) allocated
   from the Heap. Can be stored or fetched. Not too exciting.

2. STRA_object - an array of STR_objects, with the strings and pointers
   all on the Heap. This feels a lot like a text file in memory.
   Can store and fetch randomly or sequentially. Can be searched
   and sorted. Useful as a memory image of a text file or a
   list of things like file names. Can be saved to disk or loaded
   from disk. Looks like:

```
Index      STRING
  1        xxxxxxxxx
  2        yyyyy
  3        zzzzz
  ...
```

I have not fully examined the limits. I suspect either the index or the strings or something is limited to 64k. I have chosen 10,000 as a compiled in limit. When initting the object an actual allocation limit less than that is chosen for each instance. I have loaded a 173,900 byte text file (3674 lines) into a STRA. It used slightly over 200kbytes of heap when the initialization was set to 3700. Empty array initialization averages 12 bytes per string, but some is recovered in use. All-in-all, this is an extremely useful construct, especially in some of its descendant forms.

3. HOLD_object - descendent of STRA_object. Adds and array of longints which travel with each string.

```
Index      STRING       NUM
  1        xxxxxxxxx    123456
  2        yyyyy             1
  3        zzzzz        999999
  ...
```

If the string array is sorted, the num value moves along with it, making the list good for things like pointers to sections in a text file (HELP_object, SORTSECT).

4. FILE_object - an encapsulation of the binary file I/O, blockread, blockwrite code. I haven't used this too much.

5. TFILE_object - an encapsulation of TEXT file I/O. This is central to virtually every program I write. I find it much more friendly than straight Pascal code. Important here is the implementation of some PD code for random access - TEXTSEEK.

6. OUT_object - I have been aiming towards this for years. I wanted:
   1. Hide the bookkeeping for list output, headers, footers line numbers, page numbers.
   2. Have a program be able to interchangeably, but intelligently switch between output to the CRT, PRINTER or a TEXT file. This is things like pausing when output is to the CRT, form feeds to the printer, and saving output to text files.
   3. Control these options with run time parameters, and hide virtually all of it from the program which uses it.

   After constructing the OUT_object, I found that it added 8k or so to even very simple programs which didn't need all the frills. I then divided it into two levels, 0 & 1 and hid this further into two libraries OUTLIB0 and OUTLIB1. The level 0 object only adds about 4k per program. So a program can start with outlib0 (no headers, footers and word-wrap), and by changing to uses outlib1, can trivially add the headers etc.

7. HELP_object - Needs a better name, but I wrote it to implement HELP files. This is basicly a text file with random access to sections of the file. The sections can be pretty flexibly designated, some constant string at the beginning of the line followed by a name string.

8. DBF_object - Not quite ready for publication. Started out with a bit of PD code for reading DBF files (Gerald Rohr). Worked out most of header, and file structure and encapsulated as DBF_object. Couldn't make heads nor tails of .NDX files, so I used a string array descendent (STRX) to make indexing easy. KEYED_DBF_object handles multiple key files and multiple and partial key fields. I used this as the basis for a group of utilites consolidated into DB.exe to DUMP, CLONE, SORT EXPORT, CREATE, and DDL(show structure). A few small missing pieces yet.

9. BITMAP_object - I haven't used this in a couple of years. It was a large virtual bit-map, using heap in memory and loadable and saveable to disk.

STR_object - 1. String on Heap - D

Here is the interface to the STR_object:

```
{source \hnrlib\hnrobjs.pas(.str_object)}
type  stringptr = ^string;

TYPE  STR_object = OBJECT
            strptr: stringptr;                          { pointer to string on hea
            Procedure   init;                           { gets heap space     }
            Function    store (st: String): boolean;    { Stores the string   }
            Function    fetch: String;                  { Fetches the string  }
            Procedure   dump;                           { debug write         }
            procedure   dispose;                        { releases heap space }
            end;
```

The data for the object consists of a pointer to a string on the heap.

STRA_object - 2. String Array - De

Here is the interface to the STRA_object:

```
{source \hnrlib\hnrobjs.pas(.STRA_object)}
const STRA_BigArrayMax = 15000;
type  STRA_BigArray = array[1..STRA_BigArrayMax] of STR_object;
{type  STRA_BigIndex = array[1..STRA_BigArrayMax] of integer;}

TYPE  STRA_object = OBJECT
            arrayptr    : ^STRA_BigArray;
            arraymax    : integer;
            arrayused   : integer;
            arraysorted : boolean;
            modified    : boolean;
            Procedure init         (max : integer);
            Function  append       (st : string)              : boolean;
            Function  appendpush   (st : string)              : boolean;
            Function  insertstr    (n : integer;st : string)  : boolean;
            Function  deletestr    (n : integer)              : boolean;
            Function  linearfind   (st : string)              : integer;
```

```
            Function  linearsearch (st : string; mode : byte) : integer;
            Function  store        (n : integer; st : string) : boolean;
            Function  fetch        (n : integer) : string;

            Function  fetchString  (n : integer) : string;  {returns nth string
            Function  fetchInteger (n : integer) : integer; {returns nth string
            Function  fetchLongInt (n : integer) : longint; {returns nth string
            Function  fetchreal    (n : integer) : real;    {returns nth string
            Function  fetchboolean (n : integer) : boolean; {returns nth string

            Function  count        : integer; { returns number of slots used }
            Function  sorted       : boolean; { returns whether sorted }
            Function  arraymaxsize : integer; { returns max (from init)}
            Procedure dump;                   { for debugging }
            Procedure clear;                  { empties array }

            Procedure listpage   (f,n,w : integer);           { mini dump for t
            Procedure save       (fname : string);            { to text file }
            Procedure load       (fname : string);            { from text file
            Procedure loadsection(fname,sectiontag,sectionname : string); { fro

            Procedure swap(i,j : integer);                    { for sort }
            Procedure sort;                                   { shell sort}
            Function  binsearchEQ    (st : string) : integer; { if sorted }
            Function  binsearchAPPROX(st : string) : integer; { if sorted }
            Function  binsearchLE    (st : string) : integer; { if sorted }
            Function  binsearchGE    (st : string) : integer; { if sorted }

            Function  find   (st : string) : integer;             { sorted or
            Function  search (st : string; mode : byte) : integer; { sorted or
            Procedure dispose;
            Procedure done;
            end;
```

As you can see, this is rather a "kitchen sink" object. The compiler does a goo
good tools for examining just which functions I have never used. The sort is a

I have been bashing STRA around with some new test code, and it stands up well.
in heap space, it is easy to pass as a parameter between routines - stack friend
object bump into them, there are some irregularities I need to track down.

HOLD_object - 3. String & Longint Array

Here is the interface to the HOLD_object:

```
{source \hnrlib\hnrobjs.pas(.HOLD_object)}
const HOLD_BigIndexMax = 5000;   { find out real limits - hnr 1/94 }

type  HOLD_NumType     = longint;
type  HOLD_NdxType     = integer;
type  HOLD_BigIndex    = array[1..HOLD_BigIndexMax] of HOLD_NumType;


TYPE  HOLD_object = OBJECT(STRA_object)
        ArrNum     : ^HOLD_BigIndex;
        ArrHighVal : HOLD_NumType;
        MaxEntries : HOLD_NdxType;
```

```
        comment    : string[80];

        CONSTRUCTOR init   (n : HOLD_NdxType);
        Function  append   (                st :string;     Num :HOLD_NumType):
        Function  storeN   (n : HOLD_NdxType;    st :string;     Num :HOLD_NumTy
        Function  fetchN   (n : HOLD_NdxType;var st :string; var Num :HOLD_NumTy
        Function  fetchNumN(n : HOLD_NdxType)   : HOLD_NumType;
        Function  fetchStrN(n : HOLD_NdxType)   : string;

        Function  findstr  (st  : string)    : HOLD_NdxType;
        Function  findnum  (Num : HOLD_NumType)   : HOLD_NdxType;

        Function  count                   : HOLD_NdxType;
        Function  HighNum                 : HOLD_NumType;
        Procedure swap     (i,j : HOLD_NdxType);
        Procedure sort;
        Procedure dump;
        Procedure dumpN    (n : HOLD_NdxType);
        Procedure save     (fname : string);
        Procedure load     (fname : string);
        Procedure dispose;
        end;
```

FILE_object - 4. Binary File Encapsulati

Here is the interface to the FILE_object:

{source \hnrlib\hnrobjs.pas(.FILE_object)}

TFILE_object - 5. TEXT File Encapsulatio

Here is the interface to the TFILE_object:

{source \hnrlib\hnrobjs.pas(.TFILE_object)}

```
TYPE  TFILE_object = OBJECT
             Fil      : TEXT;
             filename : string[60];
             opened   : boolean;
             err      : integer;
             linenum  : longint;
             PosCurr  : longint;

             procedure init      (fn : string; create : boolean);
             procedure initAppend(fn : string);
             Procedure open      (fn : string; create : boolean);
             Function  IOResultErrChk : boolean;
             Procedure seek      (l : longint);
             function  currentposition : longint;
             Function  fetchnext(var s : string) : boolean;
             Function  append(s : string) : boolean;
             Procedure clearfile;
             Procedure refreshfile;
```

```
Function  error    : boolean;
Procedure close;
procedure done;
end;
```

OUT_object_0 - 6a. OUTPUT Encapsulation

Here is the interface to the OUT_object (level 0):

{source \hnrlib\hnrobjs.pas(.OUT_object_0)}

OUT_object_1 - 6b. OUTPUT Encapsulation

Here is the interface to the OUT_object (level 1):

{source \hnrlib\hnrobjs.pas(.OUT_object_1)}

HELP_object - 7. Sectioned TEXT File

Here is the interface to the HELP_object:

{source \hnrlib\hnrobjs.pas(.HELP_object)}

DBF_object - 8. DBase DBF File Encapsulat

Here is the interface to the DBF_object:

{source \hnrlib\hnrobjs.pas(.DBF_object)}

The really ugly code is down one or two levels in DBLKstuf and XBASstuf, and best that it remain there. I am not using this heavily at present, but it tests well and is easy to code with. The utility DBPASgen generates a nice shell around the KEYED_DBF_object so that the interface is a Pascal record structure, and it is not necessary go into DBASE world. This topic will get a paper of its own, for more detail. It is included here to show that objects can be pretty high level constructs as well as small building blocks.

BITMAP_object - 9. BitMap 1 - 65000 bit

Here is the interface to the BITMAP_object:

{source \hnrlib\bitstuf.pas(.BITMAP_object)}