The ESRI(R) SHAPE support Library

(Work in progress—handle with care)
$Id: libshape.w,v 1.20 2006/07/11 14:17:18 tlaronde Exp

Thierry LARONDE

Abstract/Résumé

ESRI(tm) has published a white paper named 'ESRI Shapefile Technical Description', describing the `shape` format and allowing its general use. Due to the size of ESRI and to the public availability of the specifications, the `shape` format is in widespread use. We shall hence support it, which is the very aim of this library.
To keep things clean and tidy, this library is clearly *shape* oriented, that is we do not make many assumptions about KerGIS idiosyncrasies: we coulb have implemented it so that it can be fed directly by KerGIS vector elements (raw geometry or value added creation—areas). But this means that whenever we change something in the vector format, this library too has to be changed, and that we will have to support options (a part of user handling) here where they have nothing to do.
Keep it UNIX(R), keep it simple and to the fact.
The document is in french.

ESRI(tm) a publié un livre blanc intitulé : `ESRI Shapefile Technical Description` décrivant le format `shape` et autorisant l'utilisation de ce format pour les échanges de données. Du fait de la dimension d'ESRI et de la publication du format, ce format est très répandu et se doit donc d'être supporté par KerGIS.
C'est l'objet de cette bibliothèque de fournir les routines nécessaires au support.
Nous aurions pu implémenter directement dans cette bibliothèque le support des éléments vectoriel de KerGIS (géométrie brute, ou valeur ajoutée topologique — les aires). Mais cela signifierait qu'à chaque modification du vectoriel de KerGIS, cette bibliothèque également serait à modifier, et cela signifie en plus que soit nous perdrions des degrés de liberté (par le lien avec des contraintes qui lui sont extérieures), soit que nous serions obligés d'implémenter ici la gestion d'options passées par l'utilisateur, alors que ce n'est pas le lieu.
Conservons donc l'esprit UNIX(R) : faire en sorte que la bibliothèque fasse ce qu'elle a à faire, rien que ça mais bien.

November 26, 2006 at 11:44

1. Licence.

2. Vue d'ensemble.

Le format *shape* consiste en trois fichiers commençant par un même préfixe (limité à 8 caractères et qui doit être en minuscules) et une extension décrivant leur fonction :

`prefix.shp`. le fichier principal contenant la géométrie ;

`prefix.shx`. le fichier d'indexation permettant d'accéder aux enregistrements dans le fichier `.shp` ;

`prefix.dbf`. un fichier d'attributs au format dBASE, la correspondance entre éléments géométriques et attributs étant une bijection établie simplement entre enregistrements du même ordre (au premier élément géométrique est associé le premier enregistrement dBASE, au deuxième : le deuxième, etc.).

L'une des difficultés du format *shape* — par ailleurs simple — est le mélange entre codage little endian et codage big endian.

Si la géométrie est correcte, mais que l'attribut associé n'est pas correct, par exemple le programme d'ESRI `ArcExplorer` n'affichera rien. Nous **devons** donc associer un attribut, éventuellement fictif.

3. Limites du format *shape*.

Le format *shape* n'est pas topologique. Chaque élément est décrit à part et les éléments sont simplement juxtaposés. Cela ne pose pas de problème dans le sens exportation KerGIS vers *shape*, mais cela peut poser problème dans le sens importation d'un *shape* contenant tout et n'importe quoi.

La nature bijective de la relation entre éléments géométriques et attributs, et la nature non topologique du format *shape* conduit à implémenter la **multiplicité** [voir la bibliothèque KerGIS `libcorr`] en démultipliant autant de fois qu'il y a d'attributs un élément géométrique, et en associant un attribut (enregistrement) nul à un élément géométrique sans attributs.

4. Le format *shape* d'ESRI(tm).

Le format `dBASE` est l'objet d'une bibliothèque dédiée, nous ne décrirons donc dans ce qui suit que les données relatives au `.shp` et `.shx`.

Il existe deux versions du **document** décrivant le format : une version datant de 1995, et la version autorisée présente aujourd'hui sur le site d'ESRI qui date de 1998.

La version de 1998 complète la version de 1995 et les fichiers générés à partir de la version 1995 sont compatibles avec la version actuelle. L'inverse n'est par contre pas vrai.

Dans la version 1995, seuls les types `SHP_TYPE_NULL`, `ARC` (qui correspond aujourd'hui à `SHP_TYPE_POLYLINE`), `SHP_TYPE_POLYGON` et `SHP_TYPE_MULTIPOINT` étaient présents.

Autre différence majeure et malencontreuse, dans la version 1995 du document n'était pas précisée la possibilité de mêler `SHP_TYPE_NULL` à tous les autres.

Conclusion, des applications développées à partir de la spécification de 1995, quand bien même ne seraient utilisés que des types décrits à l'époque, partiront vraisemblablement en vrille si des `SHP_TYPE_NULL` sont insérés.

Et le problème est que l'en-tête du fichier ne reflète pas la version : les nombres magiques sont les mêmes...

5. Deux types de données sont utilisées :

integer. 32-bit

double. flottant signé 64-bit en double précision (IEEE 754) qui est défini dans KerGIS (indirectement) dans l'en-tête `types.h` : il suffit d'utiliser *ieee_754_double_kt*.

ce qui nous donne la dimension, mais qui se complique dès lors que certains sont en little endian et d'autres en big endian, et qu'il nous faut convertir entre les différents types en tenant compte de la machine. Ces manipulations sont l'objet de macros dédiées placées dans `ksys/endian.h`.

6. L'interface publique.

L'interface publique est publiée via l'en-tête `shape.h`. L'en-tête `_shape.h` servira quant à lui à la publication des informations privées.

```
⟨ shape.h 6 ⟩ ≡
#ifndef SHAPE_H
#define SHAPE_H
#include "ksys/cdefs.h"
#include "ksys/endian.h"
#include "ksys/types.h"
#include "gis.h"
#include "corr.h"
  ⟨ Errors macros 67 ⟩
  const struct G_Msgs *const SHP_Errors;
  ⟨ Warnings macros 64 ⟩
  const struct G_Msgs *const SHP_Warnings;
  ⟨ Infos macros 61 ⟩
  const struct G_Msgs *const SHP_Infos;
  ⟨ Public macros 7 ⟩
  ⟨ Public structures 10 ⟩
  ⟨ Public prototypes 11 ⟩
#endif  /* SHAPE_H */
```

7. Les types d'éléments géométriques reconnus par *shape*. Ils sont publics car les programmes appelant doivent pouvoir s'y référer.

À noter que le type `ARC` décrit dans la version 1995 a la même valeur — heureusement — que le type `SHP_TYPE_POLYLINE` (et la même structure).

Par contre, dans la spécification de 1995, il n'était pas précisé que le type `SHP_TYPE_NULL` pouvait être mêlé aux autres types.

```
⟨ Public macros 7 ⟩ ≡
#define SHP_TYPE_NULL 0UL
#define SHP_TYPE_POINT 1UL
#define SHP_TYPE_POLYLINE 3UL
#define SHP_TYPE_POLYGON 5UL
#define SHP_TYPE_MULTIPOINT 8UL
#define SHP_TYPE_POINT_Z 11UL
#define SHP_TYPE_POLYLINE_Z 13UL
#define SHP_TYPE_POLYGON_Z 15UL
#define SHP_TYPE_MULTIPOINT_Z 18UL
#define SHP_TYPE_POINT_M 21UL
#define SHP_TYPE_POLYLINE_M 23UL
#define SHP_TYPE_POLYGON_M 25UL
#define SHP_TYPE_MULTIPOINT_M 28UL
#define SHP_TYPE_MULTIPATCH 31UL
```

See also section 8.

This code is used in section 6.

8. Pour le type `SHP_TYPE_MULTIPATCH`, la nature des éléments est spécifiée explicitement (entre autres pour les contours extérieurs et les contours intérieurs des polygones — alors que dans le cas `SHP_TYPE_POLYGON` c'est le sens de description du contour qui devrait, normalement, distinguer un contour externe d'une île... mais il se trouve que certains génèrent des fichiers incorrects).

⟨ Public macros 7 ⟩ +≡
```
#define SHP_NATURE_TRIANGLE_STRIP 0UL
#define SHP_NATURE_TRIANGLE_FAN 1UL
#define SHP_NATURE_OUTER_RING 2UL
#define SHP_NATURE_INNER_RING 3UL
#define SHP_NATURE_FIRST_RING 4UL
#define SHP_NATURE_RING 5UL
```

9. Ouverture/fermeture des fichiers.

L'ouverture des fichiers passe par la `libgis` et plus spécifiquement par *G__open*. Les conventions de cette dernière s'appliquent donc.

On crée une structure *SHP_File* qui servira à échanger les informations avec les applications.

10. La structure *SHP_File* va servir à l'échange de données entre KerGIS et Shape. Elle doit contenir des informations sur :

0. un *handler*, initialisé par la bibliothèque, qui servira à identifier les fichiers shape concernés ;
1. la nature des éléments présents dans le fichier ;
2. les données concernant l'ensemble des éléments ;
3. la structure des enregistrements associés (description des champs).

À l'ouverture des trois fichiers composant le format *shape*, un *handler* est retourné qui identifie une structure interne, privée, servant à la manipulation des fichiers.

Dans les appels suivants à la bibliothèque (insertion, suppression, recherche), c'est ce *handler* qui doit être communiqué.

⟨ Public structures 10 ⟩ ≡

```
struct SHP_File {
  int handler;      /* this will be set on return by SHP_open */
  char *prefix;     /* prefix filename */
  char *mapset;     /* where to put elements in gisdatabase */
  unsigned long type;      /* type of elements */
  unsigned long nb_elts;   /* total number of elements */
  double box[8];    /* bounding box: w, s, e, n, d, h, Mmin, Mmax */
  struct C_Data *Data;     /* Description of the Data (attributes) */
};
```

See also section 14.

This code is used in section 6.

11. La description du fichier à ouvrir sera faite via une structure **SHP_File**.

La fonction renseigne le *handler* correspondant à la structure interne affectée à la gestion du groupe de fichiers. Dans le cas d'une création, les valeurs inscrites dans l'en-tête sont celles passées via la structure **SHP_File**. En particulier, les valeurs de la boîte englobante sont prises telles quelles.

Les raisons pour lesquelles nous ne mettons pas de valeurs par défaut sont multiples.

D'une part, lors de la génération de `SHP_TYPE_NULL`, la boîte englobante ne serait pas modifiée (puisqu'il n'existe pas de géométrie associée), et le fichier pourrait alors avoir une extension théorique insensée.

D'autre part, il est problématique de supposer que dans les cas où les Z et M ne sont pas utilisés, d'avoir des minima et maxima qui se croisent (le plus simple pour assurer la mise à jour étant précisément de mettre le maximum dans le minimum et vice versa pour rendre toute comparaison future valide) ne posera pas de problèmes à l'application, c'est-à-dire qu'elle ignorera ces valeurs. Cela n'étant pas précisé dans la norme, nous n'en faisons pas l'hypothèse.

Suivant nos conventions, 0 est renvoyé en cas de succès, une erreur distincte de 0 en cas d'échec.

Dans tous les cas, elle génère une erreur si la création est demandée alors que le fichier existe déjà.

⟨ Public prototypes 11 ⟩ ≡

```
int SHP_open(struct SHP_File *Shape, int mode);
```

See also sections 12, 15, and 16.

This code is used in section 6.

12. La fermeture est on ne peut plus simple : on se contente de fermer les descripteurs. Il serait souhaitable de faire des vérifications de consistance à l'avenir avant la fermeture (mais cette vérification peut être faite en réouvrant en lecture les fichiers).

Au retour de la fonction, les structures appartenant au handler ont été libérées.

⟨ Public prototypes 11 ⟩ +≡

```
int SHP_close(struct SHP_File *Shape);
```

13. Manipuler les éléments.

Nous devons pouvoir lire et écrire des éléments, c'est-à-dire essentiellement des géométries, les attributs étant gérés via la `libcorr`.

L'utilisation principale dans KerGIS de la présente bibliothèque est l'import et l'export. Nous allons cependant permettre la **lecture** aléatoire d'un élément. L'élément à lire doit donc être indiqué par son numéro.

14. La structure *SHP_Elt* va servir à l'échange de données entre KerGIS et Shape. Elle doit contenir toutes les informations présentes dans les éléments *shape*, même si pour l'heure, certains types d'éléments ne sont pas supportés.

1. la nature de l'élément ;
2. la description géométrique de l'élément (boîte englobante, sommets etc.) ;
3. les attributs associés.

⟨ Public structures 10 ⟩ +≡

```
struct SHP_Elt {
  unsigned long type;      /* type of element */
  unsigned long id;        /* number id of element to read or written */
  double box[8];           /* w, s, e, n, d, h, Mmin, Mmax */
  unsigned long nb_parts;  /* number of points structure */
  unsigned long *parts;    /* index to first point of part in points */
  unsigned long *part_types;  /* types of parts */
  unsigned long nb_points;  /* total number of points */
  double *X;               /* vector of X coordinates */
  double *Y;               /* vector of Y coordinates */
  double *Z;               /* ignored at the moment */
  double *M;               /* ignored at the moment */
  struct C_Image *Image;   /* attributes */
};
```

15. Dès lors que la structure **SHP_Elt** est donnée, il suffit d'indiquer où nous devons l'ajouter, c'est-à-dire de passer également le **SHP_File**.

⟨ Public prototypes 11 ⟩ +≡

```
int SHP_write_elt(struct SHP_File *Shape, struct SHP_Elt *Elt);
```

16. Lors de la lecture d'un élément, les pointeurs dans la structure *Elt* doivent être valides (ou Λ), car les structures sont réallouées (*realloc*).

⟨ Public prototypes 11 ⟩ +≡

```
int SHP_read_elt(const struct SHP_File *Shape, struct SHP_Elt *Elt);
```

17. Implementation.

L'en-tête privé reprend classiquement — c'est le minimum — l'en-tête public et ajoute ce qui n'est pas supposé être manipulé par les tiers.

```
⟨ _shape.h 17 ⟩ ≡
#ifndef _SHAPE_H
#define _SHAPE_H
#include <assert.h>     /* debugging */
#include <errno.h>
#include <string.h>     /* memcpy */
#include "ksys/endian.h"
#include "shape.h"
  ⟨ Private macros 19 ⟩
  ⟨ Private prototypes 29 ⟩
  ⟨ Private structures 18 ⟩
#endif      /* _SHAPE_H */
```

18. Ouvrir/fermer les fichiers.

Comme plusieurs fichiers peuvent être ouverts (plusieurs groupes de trois fichiers), un vecteur de structures sera globalement accessible aux routines internes.

```
⟨ Private structures 18 ⟩ ≡
  struct _SHP_File {
    int flags;
    unsigned long records_written;      /* this number were written since header update */
    unsigned long shp_length;      /* this */
    unsigned long shx_length;      /* xor this in their respective header */
    int mode;      /* open mode, one of G_O_RDONLY, G_O_WRONLY, G_O_RDWR */
    int fd[2];      /* .shp, .shx */
    struct C_Corr Corr_dbf;      /* dbf */
  };
#ifdef HIC
#define EXTERN
#else
#define EXTERN extern
#endif      /* we define a maximum number of group of files to handle */
#define _MAX_HANDLERS 5      /* 5 * 3 ≡ 15 < minimumvalueof OPEN_MAX */
      EXTERN
          int _SHP_nb_handlers;      /* nb of handlers actually in use */
          EXTERN
              struct _SHP_File *_SHP_Handlers[_MAX_HANDLERS];
#define _HANDLER (_SHP_Handlers[Shape→handler])
#define _SHP_HANDLER→fd [0]
#define _SHX_HANDLER→fd [1]
#define _CORR_DBF_HANDLER→Corr_dbf
#define _MODE_HANDLER→mode      /* flags */
#define _MODIFIED °1
```

This code is used in section 17.

19. Comme nous allons à chaque fois vérifier l'intégrité du *handler*, nous définissons une macro l'effectuant.

```
⟨ Private macros 19 ⟩ ≡
#define _CHECK_HANDLER if (Shape→handler < 0 ∨ Shape→handler >
      _MAX_HANDLERS ∨ _SHP_Handlers[Shape→handler] ≡ Λ) return SHP_EBADHANDLER
```

See also sections 27 and 28.

This code is used in section 17.

20. Nous serons plus souple en importation qu'en création : nous tenterons de récupérer des shapes plus ou moins conformes, mais nous devons créer des shapes strictement conformes.

C'est le cas par exemple pour le *prefix* qui doit comporter au maximum 8 caractères en minuscules, être suivi d'un point et d'un suffixe normalisé en minuscules également. Comme, compte tenu de certains systèmes de fichiers ou de certaines habitudes, en particulier la casse n'est pas respectée, on prendra ce qui vient en entrée. Par contre, à l'export on normalisera le *prefix*.

```
⟨ open.c 20 ⟩ ≡
#include <stdlib.h>
#include <string.h>
#include <unistd.h>      /* getpid() */
#include <sys/types.h>
#include <sys/stat.h>
#include "_shape.h"
  int SHP_open(struct SHP_File *Shape, int mode)
  {
    int status ← G_SUCCESS;      /* return value */
    char *filename;
    size_t prefix_len;      /* length of prefix */      /* Initializations */
    Shape→handler ← −1;
    ⟨ Check prefix for compliance if writing and set filename 21 ⟩
    ⟨ Allocate and init new _SHP_File; goto end if no handler left or problem 25 ⟩
    ⟨ Open the three files or goto end 22 ⟩
    ⟨ Check or init shp and shx files 26 ⟩      /* already done for dbf via libcorr */
  end: return status;
  }
```

21. La description du format *shape* indique que le préfixe comprend au maximum 8 lettres, et qu'il doit être en minuscules (pour les systèmes de fichiers prenant en compte la casse). Nous normalisons donc et avertissons l'utilisateur si nous tronquons le nom mais uniquement dans le cas où nous créons le shape (nous visons la canonisation). Si nous importons, passons...

Au nom de fichier nous devons ajouter le point et le suffixe (3 lettres), plus le `'\0'` terminal soit donc 5 **char**.

```
⟨ Check prefix for compliance if writing and set filename 21 ⟩ ≡
  if (mode ≡ G_O_WRONLY) {      /* creation */
    if (strlen(Shape→prefix) > 8) {
      G_msg(SHP_WPREFIXMAX, SHP_Warnings, G_WARNING, Shape→prefix);
      Shape→prefix[8] ← '\0';
    }
    G_tolcase(Shape→prefix);
  }
  prefix_len ← strlen(Shape→prefix);
  if ((filename ← (char *) malloc(prefix_len + 5)) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    SHP_close(Shape);
    goto end;
  }
  strcpy(filename, Shape→prefix);
```

This code is used in section 20.

22. L'ouverture des trois fichiers se fait via *G_open*, qui crée les éléments de la base de données `GISDATABASE` s'ils n'existent pas, et échoue si les fichiers demandés en création existent.

G_open **doit** être utilisé car la politique pour l'accès partagé au fichier sera implémentée via cette fonction.

Si tout le monde respectait la norme édictée par ESRI, les noms de fichiers seraient en minuscules. Ils le seront en création (dont nous sommes responsables), mais en lecture, nous sommes obligés d'être plus souples pour éviter à l'utilisateur final de devoir constamment normaliser les noms de fichiers avant de nous les transmettre. D'où la gymnastique sur les suffixes, le décalage de 3 basculant des minuscules aux majuscules. Cela ne doit être fait que pour le premier fichier et la première fois. La valeur de *shift* nous permet de dépister si cela a déjà été fait.

```
⟨ Open the three files or goto end 22 ⟩ ≡
  {
    int i;
    char *suffixes[6] ← {".shp", ".shx", ".dbf", ".SHP", ".SHX", ".DBF"};
    int shift ← 0;        /* start in normalized mode, i.e. lowercase */
    for (i ← 0; i < 3; i++) {
      filename[prefix_len] ← '\0';        /* truncate to prefix */
      strcat(filename, suffixes[i + shift]);        /* append suffix */
      if (i ≠ 2) {
        if ((_SHP_Handlers[Shape→handler]→fd[i] ← G_open(X_SHAPE_SUBELT, filename, Shape→mapset,
            mode)) < 0) {
          if (i ≡ 0 ∧ ¬shift ∧ mode ≠ G_O_WRONLY) {        /* retry first time in uppercase */
            shift ← 3;
            −−i;
            continue;
          }
          status ← errno ? errno ≪ G_ERROR_SHIFT : G_EOPEN;
          SHP_close(Shape);
          goto end;
        }
      }
      else {        /* dbf */
        ⟨ Open DBF or goto end 23 ⟩
      }
    }
  }
```

This code is used in section 20.

23. L'ouverture du fichier DBF se fait via via la `libcorr`, en spécifiant simplement le *driver*, et en complétant *cluster*, *dbase* et *table* pour obtenir la création ou l'ouverture du fichier dans le sous-élément `X_SHAPE_SUBELT`.

```
⟨ Open DBF or goto end 23 ⟩ ≡
  _CORR_DBF.handler ← −1;      /* if we have to close before opening */
  _CORR_DBF.name ← Λ;      /* anonymous: no head file left in gisdatabase */
  _CORR_DBF.Head.driver ← "dbf";
  _CORR_DBF.Head.partition ← Λ;
  _CORR_DBF.Head.cluster ← Λ;
  _CORR_DBF.Head.dbase ← X_SHAPE_SUBELT;
  _CORR_DBF.Head.table ← filename;
  _CORR_DBF.Head.encoding ← Λ;      /* DBF will set or get */
  _CORR_DBF.Head.datum_group ← Λ;
  _CORR_DBF.Head.datum_group_name ← Λ;
  _CORR_DBF.Head.group_empty_as ← Λ;
  _CORR_DBF.Head.group_all_as ← Λ;
  _CORR_DBF.Head.null_as ← Λ;
  _CORR_DBF.Head.separator ← '\0';      /* let DBF set default; unused */
  _CORR_DBF.Head.chunk_size ← C_MAX_NB_ATTS;      /* used only on reading */
  if (mode ≡ G_O_WRONLY)      /* copy Data passed */
    _CORR_DBF.Data ← *Shape→Data;
  if ((status ← C_open(&(_CORR_DBF), mode))) {
    SHP_close(Shape);
    goto end;
  }
  if (mode ≠ G_O_WRONLY)      /* copy Data set */
    *Shape→Data ← _CORR_DBF.Data;
```

This code is used in section 22.

24. Comme *filename* sera encore utilisé par le soutier dBASE, on pensera à désallouer la mémoire après la fermeture du fichier.

```
⟨ Free allocated strings 24 ⟩ ≡
  free(_CORR_DBF.Head.table);
```

This code is used in section 60.

25. S'il n'y a pas de handler disponible, on peut arrêter tout de suite. Sinon on alloue une nouvelle structure qui sera complétée par la suite.

```
⟨ Allocate and init new _SHP_File; goto end if no handler left or problem 25 ⟩ ≡
  if (_SHP_nb_handlers ≥ _MAX_HANDLERS) {
    status ← SHP_EOPENMAX;
    goto end;
  }
  for (Shape→handler ← 0; Shape→handler < _MAX_HANDLERS ∧ _SHP_Handlers[Shape→handler] ≠ Λ;
       ++Shape→handler) ;      /* calloc ensures default values of 0 */
  errno ← 0;
  if ((_SHP_Handlers[Shape→handler] ← (struct _SHP_File *) calloc(1, sizeof(struct _SHP_File))) ≡ Λ) {
    Shape→handler ← −1;
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
  ++_SHP_nb_handlers;
  _MODE ← mode;      /* save opening mode */
  _SHP ← −1;      /* used by SHP_close if we need to abort before finishing */
  _SHX ← −1;
  _CORR_DBF.handler ← −1;
```

This code is used in section 20.

26. Quel est donc le format du fichier, et que devons-nous vérifier ? De facto, nous vérifions le fichier `shp` et le `dbf`. La gestion du `.dbf` est de la responsabilité de la bibliothèque dédiée, décrivons donc le format du fichier shape contenant la géométrie.

```
⟨ Check or init shp and shx files 26 ⟩ ≡
  {
    struct stat buf;
    errno ← 0;
    if (fstat(_SHP, &buf) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (buf.st_size) status ← _SHP_read_header(Shape);
    else if (_MODE ≡ G_O_RDONLY) status ← SHP_EEMPTY;
    else {
      _HANDLER→flags |= _MODIFIED;
      _HANDLER→shp_length ← HEADER_LENGTH_W;
      _HANDLER→shx_length ← HEADER_LENGTH_W;
      status ← _SHP_write_header(Shape);
    }
    if (status) {
      SHP_close(Shape);
      goto end;      /* already error, bail out */
    }
  }
```

This code is used in section 20.

27. Le fichier *shape* comprend un en-tête de longueur fixe, suivi d'enregistrements comprenant un en-tête de longueur fixe et des données de longueur variable.

Quand nous parlons d'écrire l'en-tête, nous invoquons l'en-tête général.

Nous allons faire également beaucoup de conversions en devant écrire un flux. Nous écrivons dans un buffer, les noms : *buf* , *offset* doivent être considérés comme réservés, et pour utiliser les macros, bien les lire ! Elles incrémentent offset en fonction de ce qui a été inscrit.

Il existe une différence entre les `GET_ *` et les `PUT_ *` : on récupère une valeur à l'offset indiqué dans l'invocation d'une macro `GET_ *` ; on place la valeur indiquée à l'offset courant (qui sera automatiquement incrémenté) avec une macro `PUT_ *`.

Les structures utilisées par le code sont dans un format "naturel" pour la machine les traitant, et on effectue les conversions entre le stockage sur disque (tailles fixées), et la manipulation en mémoire (tailles variables).

Rien ne garantit qu'un **double** sera un *ieee_754_double_kt*. Par contre le C standard garantit qu'un **long** est au moins un tetra, nous utilisons donc le **long** et ce sont les macros qui font la conversion.

```
⟨ Private macros 19 ⟩ +≡
#define PUT_BE_INT(some_int)
  {
    uint32_t an_int;
    an_int ← ttobet((uint32_t)some_int);
    memcpy(buf + offset, &an_int, 4);
    offset += 4;
  }
#define PUT_LE_INT(some_int)
  {
    uint32_t an_int;
    an_int ← ttolet((uint32_t)some_int);
    memcpy(buf + offset, &an_int, 4);
    offset += 4;
  }
#define PUT_LE_DOUBLE(a_double)
  {
    ieee_754_double_ktocta;
    char *po;
    octa ← (ieee_754_double_kt)a_double;
    po ← (char *) &octa;
    otoleo(po);
    memcpy(buf + offset, po, 8);
    offset += 8;
  }
#define GET_BE_INT(offset)(unsigned long)bettot ( * ( ( uint32_t * ) (buf + offset) ) )
#define GET_LE_INT(offset)(unsigned long)lettot ( * ( ( uint32_t * ) (buf + offset) ) )
#define GET_LE_DOUBLE(a_double, offset)
  {
    ieee_754_double_ktocta;
    char *po;
    po ← (char *) &octa;
    memcpy(po, buf + offset, 8);
    leotoo(po);
    a_double ← (double) octa;
  }
```

28. Il suffit donc de lire les HEADER_LENGTH octets et de mettre à jour les informations dans la structure *Header*.

```
⟨ Private macros 19 ⟩ +≡
#define HEADER_LENGTH 100_UL /* in bytes */
#define HEADER_LENGTH_W 50_UL /* in wydes */
#define HEADER_CODE 9994_UL
#define HEADER_VERSION 1000_UL
```

29.

```
⟨ Private prototypes 29 ⟩ ≡
  int _SHP_read_header(struct SHP_File *Shape);
```

See also section 35.

This code is used in section 17.

30.

```
⟨ header.c 30 ⟩ ≡
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "_shape.h"
  int _SHP_read_header(struct SHP_File *Shape)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    char buf[HEADER_LENGTH];
    ⟨ Other _SHP_read_header variables 32 ⟩
    ⟨ Saved shp and shx offsets 37 ⟩
    errno ← 0; /* check mode */
    if (_MODE ≡ G_O_WRONLY) return SHP_EMODE;
    ⟨ Save current offsets and seek to beginning 38 ⟩ /* shp first */
    errno ← 0;
    if (read(_SHP, buf, HEADER_LENGTH) ≠ HEADER_LENGTH) {
      status ← SHP_ESHPREAD;
      goto end;
    }
    if (GET_BE_INT(0) ≠ HEADER_CODE) return SHP_EHEADERCODE;
    if (GET_LE_INT(28) ≠ HEADER_VERSION) return SHP_EHEADERVERSION;
    _HANDLER→shp_length ← GET_BE_INT(24);
    Shape→type ← GET_LE_INT(32);
    for (i ← 0; i < 8; i++) GET_LE_DOUBLE(Shape→box[i], 36 + i * 8) /* shx second */
    errno ← 0;
    if (read(_SHX, buf, HEADER_LENGTH) ≠ HEADER_LENGTH) {
      status ← SHP_ESHXREAD;
      goto end;
    }
    _HANDLER→shx_length ← GET_BE_INT(24);
    ⟨ Check consistency of files; if problem, goto end 31 ⟩
    ⟨ Print header infos 34 ⟩
  end: ⟨ Restore shp and shx offsets 40 ⟩
    return status;
  }
```

See also section 36.

31. Plusieurs vérifications peuvent être faites, en particulier sur la cohérence entre le `_SHP` et le `_SHX`, les longueurs des fichiers et le nombre d'éléments.

```
⟨ Check consistency of files; if problem, goto end 31 ⟩ ≡
  {
    struct stat buf;
    errno ← 0;
    if (fstat(_SHP, &buf) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if ((buf.st_size ≠ _HANDLER→shp_length * 2)) {
      status ← SHP_ESHPSIZE;
      goto end;
    }
    if (fstat(_SHX, &buf) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if ((buf.st_size ≠ _HANDLER→shx_length * 2)) {
      status ← SHP_ESHXSIZE;
      goto end;
    }
    Shape→nb_elts ← (_HANDLER→shx_length − 50)/4;
    ⟨ Scan the shp file to gather information about elements 33 ⟩
  }
```

This code is used in section 30.

32. Parce qu'il peut exister deux versions des fichiers (qui ne sont pas reflétées dans les numéros de version) : une version conforme à la spécification datant de 1995 (ancienne), et une version conforme à la actuelle (mars 1998), on fait une passe pour vérifier que les éléments sont bien à l'adresse indiquée, et pour en découvrir le type.

```
⟨ Other _SHP_read_header variables 32 ⟩ ≡
  struct {
    int valid;
    long nb_elts;
    char *name;
  } types[33] ← {{1, 0_UL, "NULL"}, {1, 0_UL, "POINT"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "POLYLINE (ARC)"},
      {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "POLYGON"}, {0, 0_UL, "UNKNOWN"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL,
      "MULTIPOINT"}, {0, 0_UL, "UNKNOWN"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "POINT_Z"}, {0, 0_UL,
      "UNKNOWN"}, {1, 0_UL, "POLYLINE_Z"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "POLYGON_Z"}, {0, 0_UL,
      "UNKNOWN"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "MULTIPOINT_Z"}, {0, 0_UL, "UNKNOWN"}, {0, 0_UL,
      "UNKNOWN"}, {1, 0_UL, "POINT_M"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "POLYLINE_M"}, {0, 0_UL,
      "UNKNOWN"}, {1, 0_UL, "POLYGON_M"}, {0, 0_UL, "UNKNOWN"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL,
      "MULTIPOINT_M"}, {0, 0_UL, "UNKNOWN"}, {0, 0_UL, "UNKNOWN"}, {1, 0_UL, "MULTIPATCH"}, {0, 0_UL,
      "UNKNOWN"}      /* catch all remainings > 32 */
  };
  int compatibility ← 1;      /* 0 : incompatible 1995; 1 : compatible 1995 */
  int has_nulls ← 0;      /* has mixed nulls -> incompatible 1995 */
  int mixed_types ← 0;
```

This code is used in section 30.

33. Nous avons lu les en-têtes, nous sommes donc positionnés au démarrage des données dans le fichier d'index et le fichier de géométrie. Nous allons lire séquentiellement l'index, puis aller chercher l'élément correspondant dans le fichier de géométrie (rien n'oblige — sinon l'index serait inutile — à ce que les éléments soient présents dans l'ordre dans le fichier `.shp`) et nous contenter d'identifier le type de l'élément.

Nous nous servirons de l'entrée 32 dans le tableau *types* pour, au final, additionner tous les éléments invalides.

Nous devons également tenir compte du fait qu'il peut y avoir 0 éléments, donc autant de types et faire en sorte que ce cas particulier soit traité dans la branche "structure correcte" : si le fichier est vide et que nous sommes parvenus à lire l'en-tête, il est correct !

```
⟨ Scan the shp file to gather information about elements 33 ⟩ ≡
  {
    struct SHP_Elt *Elt;
    int nb_types;
    unsigned long max_type;
    char *buf ← Λ;
    if ((Elt ← (struct SHP_Elt *) malloc(sizeof(struct SHP_Elt))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    for (i ← 1; i ≤ Shape→nb_elts; i++) {
      Elt→id ← i;
      ⟨ Retrieve index entry 54 ⟩
      ⟨ Retrieve elt record 53 ⟩
      Elt→id ← GET_BE_INT(0);
      if (Elt→id ≠ i) {
        types[32].nb_elts++;
        continue;
      }
      Elt→type ← GET_LE_INT(8);
      Elt→type ← Elt→type > 31 ? 32 : Elt→type;
      types[Elt→type].nb_elts++;
    }
    nb_types ← 0;
    max_type ← 0;
    for (i ← 0; i < 32; i++) {
      if (types[i].nb_elts) {
        ++nb_types;
        max_type ← i;
      }
      if (¬types[i].valid) types[32].nb_elts += types[i].nb_elts;
    }
    if (types[SHP_TYPE_NULL].nb_elts) has_nulls ← 1;
    if (max_type > SHP_TYPE_POLYGON ∨ (nb_types ≡ 2 ∧ has_nulls)) compatibility ← 0;
    if (nb_types ≤ 1 ∨ (nb_types ≡ 2 ∧ has_nulls ∧ ¬compatibility)) mixed_types ← 0;
    else mixed_types ← 1;
    free(Elt);
  }
```

This code is used in section 31.

34. Les informations récupérées ont un intérêt.

Il existe une différence entre les messages générées via *G_msg*, qui sont tous redirigés vers *stderr* (ou envoyés par courriel), et des informations qui sont la produit voulu de la fonction : ces messages sont dirigés vers *stdout*.

De plus, alors que les “méta-messages” (des commentaires sur le déroulement du programme) peuvent un jour être traduits, les productions voulues (des informations) sont dans un format fixe et dans une langue fixée : le C.

```
⟨ Print header infos 34 ⟩ ≡
  printf("Header␣informations:\n");
  printf("Bounding␣box:\nXmin:␣%f\nYmin:␣%f\nXmax:␣%f\nYmax:␣%f\nZmin\
      :␣%f\nZmax:␣%f\n""Mmin:␣%f\nMmax:␣%f\n", Shape→box[0], Shape→box[1], Shape→box[2],
      Shape→box[3], Shape→box[4], Shape→box[5], Shape→box[6], Shape→box[7]);
  printf("Nb␣of␣records:␣%ld\n", Shape→nb_elts);
  printf("Repartition␣of␣types:\n");
  for (i ← 0; i < 32; i++) {
    if (types[i].valid) printf("␣␣%15s:␣%10ld\n", types[i].name, types[i].nb_elts);
  }
  if (types[32].nb_elts) printf("Number␣of␣unknowns:␣%ld\n", types[32].nb_elts);
  printf("Number␣of␣incorrect␣elements:␣%ld\n", types[32].nb_elts);
  printf("1995␣compatible:␣%s\n", compatibility ? "YES" : "NO");
  printf("Has␣nulls:␣%s\n", has_nulls ? "YES" : "NO");
  printf("Mixed␣types␣(excluding␣NULLs):␣%s\n", mixed_types ? "YES" : "NO");
```

This code is used in section 30.

35. Initialiser ou écrire l’en-tête : *_SHP_write_header*.

La gestion des offsets est prise en compte dans la routine, qui est typiquement du genre de celle qui ne doit pas être appélée plusieurs fois au sein du même processus !

```
⟨ Private prototypes 29 ⟩ +≡
  int _SHP_write_header(struct SHP_File *Shape);
```

36.

```
⟨ header.c 30 ⟩ +≡
  int _SHP_write_header(struct SHP_File *Shape)
  {
    int status ← G_SUCCESS;
    int i;
    char buf[HEADER_LENGTH];
    off_t offset ← 0;
    ⟨ Saved shp and shx offsets 37 ⟩
    if (¬(_HANDLER→flags & _MODIFIED)) return G_SUCCESS;      /* nothing to be done */
    G_msg(SHP_IHWRITE, SHP_Infos, G_INFO);
    if (_HANDLER→records_written) G_msg(SHP_INBWRITTEN, SHP_Infos, G_INFO, _HANDLER→records_written);
        /* check mode */
    if (_MODE ≡ G_O_RDONLY) return SHP_EMODE;
    ⟨ Save current offsets and seek to beginning 38 ⟩
    PUT_BE_INT(HEADER_CODE);
    for (i ← 0; i < 5; i++) PUT_BE_INT(0);
    PUT_BE_INT(_HANDLER→shp_length);
    PUT_LE_INT(HEADER_VERSION);
    PUT_LE_INT(Shape→type);
    for (i ← 0; i < 8; i++) PUT_LE_DOUBLE(Shape→box[i]);      /* shp first */
    if (write(_SHP, buf, HEADER_LENGTH) ≠ HEADER_LENGTH) {
      status ← SHP_ESHPWRITE;
      goto end;
    }      /* shx */
    offset ← 24;
    PUT_BE_INT(_HANDLER→shx_length);
    if (write(_SHX, buf, HEADER_LENGTH) ≠ HEADER_LENGTH) {
      status ← SHP_ESHXWRITE;
      goto end;
    }      /* flush tracker of records written */
    _HANDLER→records_written ← 0;
    _HANDLER→flags &= ∼_MODIFIED;
  end: ⟨ Restore shp and shx offsets 40 ⟩
    return status;
  }
```

37. Nous pouvons être appelés à lire ou écrire l'en-tête alors que l'offset pointe ailleurs qu'au début du fichier. On sauvegarde donc les offsets, avant de revenir au début.

```
⟨ Saved shp and shx offsets 37 ⟩ ≡
  off_t currpos_shp ← 0, currpos_shx ← 0;      /* current saved */
```

This code is used in sections 30, 36, and 42.

38.

```
⟨ Save current offsets and seek to beginning 38 ⟩ ≡
  currpos_shp ← lseek(_SHP, 0, SEEK_CUR);
  lseek(_SHP, 0, SEEK_SET);
  currpos_shx ← lseek(_SHX, 0, SEEK_CUR);
  lseek(_SHX, 0, SEEK_SET);
```

This code is used in sections 30 and 36.

39.

⟨ Save current offsets and seek to end 39 ⟩ ≡
```
currpos_shp ← lseek(_SHP, 0, SEEK_CUR);
lseek(_SHP, 0, SEEK_END);
currpos_shx ← lseek(_SHX, 0, SEEK_CUR);
lseek(_SHX, 0, SEEK_END);
```
This code is used in section 42.

40.

⟨ Restore shp and shx offsets 40 ⟩ ≡
```
lseek(_SHP, currpos_shp, SEEK_SET);
lseek(_SHX, currpos_shx, SEEK_SET);
```
This code is used in sections 30, 36, and 42.

41. Manipuler les éléments.

Les éléments sont inscrits comme *record* après l'en-tête dans le fichier principal. La structure consiste en un en-tête (de dimension fixe), suivi par le contenu introduit par le type qui détermine la structure du dit contenu.

Pour diverses raisons (en particulier le fait qu'à l'avenir nous pourrions utiliser plusieurs descripteurs suivant les modes, l'un étant tel que les écritures sont toutes à la fin [ce qui permet des accès concurrents]), les modifications de l'en-tête ne sont pas toutes répercutées immédiatement : les valeurs inscrites sur le disque ne sont pas forcément celles en mémoire.

C'est la valeur de *records_written* qui permet de suivre l'état.

42. Écrire un élément : *SHP_write_elt*.

Le traitement dépend du type d'élément. Nous allons donc utiliser un énorme switch.

```
⟨ write.c 42 ⟩ ≡
#include <unistd.h>
#include "_shape.h"
  int SHP_write_elt(struct SHP_File *Shape, struct SHP_Elt *Elt)
  {
    int status ← G_SUCCESS;
    int i;
    unsigned long wlength ← 0;      /* length of content in wydes */
    unsigned long blength;      /* total length in bytes: if wlength is max this won't do! */
    unsigned long record_nb;
    char *buf ← Λ;      /* writing in this buffer before writing to file */
    off_t offset ← 0;      /* will track current position in buffer */
    ⟨ Saved shp and shx offsets 37 ⟩
    _CHECK_HANDLER;      /* only appending at the moment */
    ⟨ Save current offsets and seek to end 39 ⟩      /* check mode */
    if (_MODE ≡ G_O_RDONLY) {
      status ← SHP_EMODE;
      goto end;
    }      /* others are not supported at the moment */
    switch (Elt→type) {
    case SHP_TYPE_NULL: blength ← 12;
      break;
    case SHP_TYPE_POINT: blength ← 28;
      break;
    case SHP_TYPE_POLYLINE: case SHP_TYPE_POLYGON: blength ← 52 + Elt→nb_parts * 4 + Elt→nb_points * 16;
      break;
    default: status ← SHP_ETYPE;
      goto end;
    }
    if (Elt→type ≠ Shape→type ∧ Elt→type ≠ SHP_TYPE_NULL) {
      status ← SHP_ETYPEMIXED;
      goto end;
    }
    errno ← 0;
    if ((buf ← (char *) malloc(blength)) ≡ Λ) {      /* alloc the room */
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }      /* we skip the elt header : it will be completed after */
    offset += 8;
    PUT_LE_INT(Elt→type);
    switch (Elt→type) {
    case SHP_TYPE_NULL: break;      /* the type is already set and it's enough */
    case SHP_TYPE_POINT: ⟨ Point shape 48 ⟩
      break;
    case SHP_TYPE_POLYLINE: case SHP_TYPE_POLYGON: ⟨ Polygon shape 49 ⟩
      break;
    default: status ← SHP_ETYPE;
      goto end;
    }
    if (offset ≠ blength) {
```

```
    G_msg(SHP_WLENGTH, SHP_Warnings, G_WARNING, blength, (unsigned long) offset);
    status ← SHP_ELOST;
    goto end;
  }      /* update wlength */
  wlength ← (offset − 8)/2;
  offset ← 4;
  PUT_BE_INT(wlength);
  ⟨ if Elt→Image→nb_atts_here ≡ 0, create an empty attribute 43 ⟩
  ⟨ Write a nb_atts_here of record 44 ⟩
  if (Elt→type ≠ SHP_TYPE_NULL) ⟨ Adjust global bounding box if necessary 47 ⟩
end: ⟨ Restore shp and shx offsets 40 ⟩
  free(buf);
  return status;
}
```

43. Lorsque la géométrie ne possède pas d'attributs, on lui passe un attribut vide.

```
⟨ if Elt→Image→nb_atts_here ≡ 0, create an empty attribute 43 ⟩ ≡
  if (Elt→Image→nb_atts_here ≡ 0) {
    G_msg(SHP_INOATTS, SHP_Infos, G_INFO, (long) Elt→Image→group);
    C_alloc_atts(_CORR_DBF.Data.nb_datums, Elt→Image, 1);
  }
```

This code is used in section 42.

44. Si l'élément géométrique était muni de plusieurs attributs, générer autant d'enregistrements que d'attributs.

```
⟨ Write a nb_atts_here of record 44 ⟩ ≡
  record_nb ← Shape→nb_elts;
  for (i ← 0; i < Elt→Image→nb_atts_here; i++) {
    ++record_nb;
    offset ← 0;
    PUT_BE_INT(record_nb);
    if (write(_SHP, buf, (wlength + 4) ∗ 2) ≠ ((wlength + 4) ∗ 2)) {
      status ← SHP_ESHPWRITE;
      goto end;
    }
    ⟨ Update index file or goto end 45 ⟩      /* update header _SHP_File copy. disk file not updated now! */
    Shape→nb_elts ← record_nb;
    Elt→id ← record_nb;      /* return last elt written id */
    _HANDLER→shp_length += wlength + 4;
    _HANDLER→shx_length += 4;
    ++_HANDLER→records_written;
    _HANDLER→flags |= _MODIFIED;
  }
  ⟨ Insert attributes or goto end 46 ⟩
```

This code is used in section 42.

45. Les entrées dans le fichier d'index sont simplement un offset général, et une longueur en wydes. On place le buffer dans un bloc pour recouvrir le *buf* et *offset* générique, ce qui nous permet de continuer d'utiliser les macros.

L'offset du nouvel enregistrement est simplement la fin du fichier avant l'enregistrement en cours. Comme nous ne l'avons pas mise à jour avant de venir ici, il suffit de la recopier.

```
⟨ Update index file or goto end 45 ⟩ ≡
  {
    char buf[8];
    off_t offset ← 0;
    PUT_BE_INT(_HANDLER→shp_length);
    PUT_BE_INT(wlength);
    if (write(_SHX, buf, 8) ≠ 8) {
      status ← SHP_ESHXWRITE;
      goto end;
    }
  }
```

This code is used in section 44.

46. La gestion des attributs est dédiée à la bibliothèque `libcorr`. Donc rien de magique ici... à part se souvenir qu'un attribut n'est inscrit que s'il est marqué comme modifié ou que son identifiant est nul... Dans les autres cas, rien ne se passe !

```
⟨ Insert attributes or goto end 46 ⟩ ≡
  {
    int i;
    for (i ← 0; i < Elt→Image→nb_atts_here; i++) Elt→Image→Atts[i].id ← 0;
  }
  if ((status ← C_set_image(&_CORR_DBF, Elt→Image))) goto end;
```

This code is used in section 44.

47. L'élément ajouté peut très bien sortir de la boîte englobante globale. On vérifie donc et on ajuste si nécessaire.

Comme l'en-tête a été modifié (en mémoire pour l'instant) et qu'il est marqué comme modifié, l'écriture de cet en-tête provoquera également la mise à jour des informations sur la boîte englobante. Il n'y a donc rien de particulier à faire.

```
⟨ Adjust global bounding box if necessary 47 ⟩ ≡
  {
    if (Elt→box[0] < Shape→box[0]) Shape→box[0] ← Elt→box[0];
    if (Elt→box[1] < Shape→box[1]) Shape→box[1] ← Elt→box[1];
    if (Elt→box[2] > Shape→box[2]) Shape→box[2] ← Elt→box[2];
    if (Elt→box[3] > Shape→box[3]) Shape→box[3] ← Elt→box[3];
    if (Elt→type ≡ SHP_TYPE_POINT_Z ∨ Elt→type ≡ SHP_TYPE_MULTIPOINT_Z ∨ Elt→type ≡
          SHP_TYPE_POLYLINE_Z ∨ Elt→type ≡ SHP_TYPE_POLYGON_Z) {
      if (Elt→box[4] < Shape→box[4]) Shape→box[4] ← Elt→box[4];
      if (Elt→box[5] > Shape→box[5]) Shape→box[5] ← Elt→box[5];
    }
    if (Elt→type ≡ SHP_TYPE_POINT_M∨Elt→type ≡ SHP_TYPE_POINT_Z∨Elt→type ≡ SHP_TYPE_MULTIPOINT_M∨
          Elt→type ≡ SHP_TYPE_MULTIPOINT_Z ∨ Elt→type ≡ SHP_TYPE_POLYLINE_M ∨ Elt→type ≡
          SHP_TYPE_POLYLINE_Z ∨ Elt→type ≡ SHP_TYPE_POLYGON_M ∨ Elt→type ≡ SHP_TYPE_POLYGON_Z) {
      if (Elt→box[6] < Shape→box[6]) Shape→box[6] ← Elt→box[6];
      if (Elt→box[7] > Shape→box[7]) Shape→box[7] ← Elt→box[7];
    }
  }
```

This code is used in section 42.

48. Insérer un élément SHP_TYPE_POINT.

Les points consistent simplement en un couple x, y. En provenance du vectoriel de KerGIS, les points sont représentés par une ligne de longueur nulle et de type DOT. Nous n'utilisons donc que le premier couple de coordonnées.

```
⟨ Point shape 48 ⟩ ≡
  PUT_LE_DOUBLE(Elt→X[0]);
  PUT_LE_DOUBLE(Elt→Y[0]);
```

This code is used in section 42.

49. Écrire les polygones.

Les polygones reprennent la même structure que les polylignes, tout simplement parce qu'à l'instar de KerGIS, la principale différence consiste dans la nature des arcs : décrivent-ils un chemin, ou une frontière (une ligne, ou une aire en dimension 2) ? De facto, depuis KerGIS, un arc est une polyligne composée d'un seul tronçon (ce qui n'est pas le cas d'une polyligne dans shape, qui peut comporter plusieurs tronçons non connectés).

```
⟨ Polygon shape 49 ⟩ ≡        /* bounding box */
  for (i ← 0; i < 4; i++) PUT_LE_DOUBLE(Elt→box[i])PUT_LE_INT(Elt→nb_parts);
  PUT_LE_INT(Elt→nb_points);
  for (i ← 0; i < Elt→nb_parts; i++) {
    PUT_LE_INT(Elt→parts[i]);
  }
  for (i ← 0; i < Elt→nb_points; i++) {
    PUT_LE_DOUBLE(Elt→X[i]);
    PUT_LE_DOUBLE(Elt→Y[i]);
  }
```

This code is used in section 42.

50. Lire un élément : *SHP_read_elt*.

Le traitement dépend du type d'élément traité. Nous allons donc utiliser un énorme switch.

Nous sommes plus souples en lecture qu'en écriture : nous tentons d'écrire des fichiers qui respectent à la lettre la spécification, mais nous essayons de récupérer des fichiers qui ne la respectent pas, tant que la structure des éléements est correcte.

Les erreurs en écriture peuvent se traduire en avertissements en lecture.

```
⟨ read.c 50 ⟩ ≡
#include <unistd.h>
#include "_shape.h"
  int SHP_read_elt(const struct SHP_File *Shape, struct SHP_Elt *Elt)
  {
    int status ← G_SUCCESS;
    int i;
    char *buf ← Λ;      /* reading index/elt in this buffer */
    _CHECK_HANDLER;      /* check mode */
    if (_MODE ≡ G_O_WRONLY) {
      status ← SHP_EMODE;
      goto end;
    }
    if ((¬Elt→id) ∨ Elt→id > Shape→nb_elts) {
      status ← SHP_EBADELTID;
      goto end;
    }
    ⟨ Reset defaults in Elt structure 55 ⟩
    ⟨ Retrieve index entry 54 ⟩
    ⟨ Retrieve elt record 53 ⟩
    Elt→type ← GET_LE_INT(8);
    if (Elt→type ≠ Shape→type ∧ Elt→type ≠ SHP_TYPE_NULL)
      G_msg(SHP_WTYPEMIXED, SHP_Warnings, G_WARNING, Shape→type, Elt→type);
    switch (Elt→type) {
    case SHP_TYPE_NULL: break;      /* done */
    case SHP_TYPE_POINT: case SHP_TYPE_POINT_Z: case SHP_TYPE_POINT_M: ⟨ Read point shape 57 ⟩
      break;
    case SHP_TYPE_MULTIPOINT: case SHP_TYPE_MULTIPOINT_Z: case SHP_TYPE_MULTIPOINT_M:
      ⟨ Read multipoint shape 58 ⟩
      break;
    case SHP_TYPE_POLYLINE: case SHP_TYPE_POLYLINE_Z: case SHP_TYPE_POLYLINE_M:
      case SHP_TYPE_POLYGON: case SHP_TYPE_POLYGON_Z: case SHP_TYPE_POLYGON_M:
      ⟨ Read polygon shape 59 ⟩
      break;
    default: status ← SHP_ETYPE;
      goto end;
    }
    ⟨ Read attribute 51 ⟩
  end: free(buf);
    return status;
  }
```

51. Pour obtenir l'attribut associé, nous réclamons simplement l'image à la `libcorr` en passant l'identifiant de l'élément (le soutier `dbf` ne possède pas de fichier d'indexation pour l'instant, il identifie donc *group* avec *Att*.*id*.

Nous ne tenons donc pas compte de *multiplicity* puisqu'elle doit être égale à 1.

```
⟨ Read attribute 51 ⟩ ≡
  Elt→Image→group ← Elt→id;
  Elt→Image→offset ← 0;
  if ((status ← C_get_image(&_CORR_DBF, Elt→Image))) goto end;
```

This code is used in section 50.

52. À part dans le cas du `SHP_TYPE_NULL` ou des types point unique, tous les éléments multiples ont une bbox définie.

```
⟨ Read bbox 52 ⟩ ≡
  {
    for (i ← 0; i < 4; i++) GET_LE_DOUBLE(Elt→box[i], 12 + i * 8)
  }
```

This code is used in sections 58 and 59.

53. L'index nous indique l'offset et la taille du contenu — auquel il faut donc ajouter l'en-tête — de l'élément en **wydes**. Il suffit donc de se déplacer dans la fichier `_SHP` à l'offset indiqué, d'allouer un tampon suffisant et de lire.

```
⟨ Retrieve elt record 53 ⟩ ≡
  {
    unsigned long length;      /* length of content in bytes */
    if (lseek(_SHP, (off_t)(GET_BE_INT(0) * 2), SEEK_SET) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    length ← (GET_BE_INT(4) + 4) * 2;
    {
      char *p;
      if ((p ← (char *) realloc(buf, length)) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;      /* buf freed at end */
      }
      buf ← p;
    }
    if (read(_SHP, buf, length) ≠ length) {
      status ← SHP_ESHXREAD;
      goto end;
    }
  }
```

This code is used in sections 33 and 50.

54. Récupérer l'entrée correspondant à l'index est on ne peut plus simple.

```
⟨ Retrieve index entry 54 ⟩ ≡
  {
    char *p;
    if ((p ← realloc(buf, 8)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      free(buf);
      goto end;
    }
    buf ← p;
    if (lseek(_SHX, (off_t)(HEADER_LENGTH + (Elt→id − 1) ∗ 8), SEEK_SET) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (read(_SHX, buf, 8) ≠ 8) {
      status ← SHP_ESHXREAD;
      goto end;
    }
  }
```

This code is used in sections 33 and 50.

55. Afin de prévenir tout problème, et pour faire en sorte que même en cas d'erreur, les informations dans la structure aient un sens, nous remettons les compteurs à zéro dès lors que nous allons effectuer les manipulations.

Cela permet qui plus est d'en terminer rapidement dans le cas de la rencontre d'un SHP_TYPE_NULL.

```
⟨ Reset defaults in Elt structure 55 ⟩ ≡
  Elt→nb_parts ← Elt→nb_points ← 0;
  for (i ← 0; i < 8; i++) Elt→box[i] ← 0.0;
```

This code is used in section 50.

56. Il est important que les pointeurs sur les vecteurs dans la structure *Elt* soient valides, car nous réallouons. La procédure inlin si après est commune à tous les types, elle est invoquée à partir du moment où le nombre de points et le nombre de parties ont été renseignés.

```
⟨ Realloc Elt vectors 56 ⟩ ≡
  {
    unsigned long *pi;
    double *pd;
    if (Elt→nb_parts) {
      errno ← 0;
      if ((pi ← (unsigned long *) realloc(Elt→parts, Elt→nb_parts * sizeof(unsigned long))) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      Elt→parts ← pi;
    }
    errno ← 0;
    if ((pd ← (double *) realloc(Elt→X, Elt→nb_points * sizeof(double))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    Elt→X ← pd;
    errno ← 0;
    if ((pd ← (double *) realloc(Elt→Y, Elt→nb_points * sizeof(double))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    Elt→Y ← pd;
    if (Elt→type ≡ SHP_TYPE_POINT_Z ∨ Elt→type ≡ SHP_TYPE_MULTIPOINT_Z ∨ Elt→type ≡
        SHP_TYPE_POLYLINE_Z ∨ Elt→type ≡ SHP_TYPE_POLYGON_Z) {
      errno ← 0;
      if ((pd ← (double *) realloc(Elt→Z, Elt→nb_points * sizeof(double))) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      Elt→Z ← pd;
    }
    if (Elt→type ≡ SHP_TYPE_POINT_M∨Elt→type ≡ SHP_TYPE_POINT_Z∨Elt→type ≡ SHP_TYPE_MULTIPOINT_M∨
        Elt→type ≡ SHP_TYPE_MULTIPOINT_Z ∨ Elt→type ≡ SHP_TYPE_POLYLINE_M ∨ Elt→type ≡
        SHP_TYPE_POLYLINE_Z ∨ Elt→type ≡ SHP_TYPE_POLYGON_M ∨ Elt→type ≡ SHP_TYPE_POLYGON_Z) {
      errno ← 0;
      if ((pd ← (double *) realloc(Elt→M, Elt→nb_points * sizeof(double))) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      Elt→M ← pd;
    }
  }
```

This code is used in sections 57, 58, and 59.

57. Lire les éléments de type `SHP_TYPE_POINT` **et dérivés.**

C'est le cas le plus simple, les altitudes et mesures étant présents ou non suivant le type.

```
⟨ Read point shape 57 ⟩ ≡
  {
    Elt→nb_points ← 1;
    ⟨ Realloc Elt vectors 56 ⟩
    GET_LE_DOUBLE(Elt→X[0], 12)
    GET_LE_DOUBLE(Elt→Y[0], 20)
    if (Elt→type ≡ SHP_TYPE_POINT_M) GET_LE_DOUBLE(Elt→M[0], 28)
    if (Elt→type ≡ SHP_TYPE_POINT_Z) {
      GET_LE_DOUBLE(Elt→Z[0], 28)
      GET_LE_DOUBLE(Elt→M[0], 36)
    }
  }
```

This code is used in section 50.

58. Lire les éléments de type `SHP_TYPE_MULTIPOINT` **et dérivés.**

Le nuage de point est incluse dans une bbox, et il y a simplement un vecteur de coordonnées.

```
⟨ Read multipoint shape 58 ⟩ ≡
  {
    unsigned long delta;
    ⟨ Read bbox 52 ⟩
    Elt→nb_points ← GET_LE_INT(44);
    ⟨ Realloc Elt vectors 56 ⟩
    for (i ← 0; i < Elt→nb_points; i++) {
      GET_LE_DOUBLE(Elt→X[i], 48 + i * 16)
      GET_LE_DOUBLE(Elt→Y[i], 48 + i * 16 + 8)
    }
    if (Elt→type ≡ SHP_TYPE_MULTIPOINT_M) {
      delta ← 48 + Elt→nb_points * 16;
      GET_LE_DOUBLE(Elt→box[6], delta)
      GET_LE_DOUBLE(Elt→box[7], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→M[i], delta + i * 8)
    }
    if (Elt→type ≡ SHP_TYPE_MULTIPOINT_Z) {
      delta ← 48 + Elt→nb_points * 16;
      GET_LE_DOUBLE(Elt→box[4], delta)
      GET_LE_DOUBLE(Elt→box[5], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→Z[i], delta + i * 8)
      delta += Elt→nb_points * 8;
      GET_LE_DOUBLE(Elt→box[6], delta)
      GET_LE_DOUBLE(Elt→box[7], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→M[i], delta + i * 8)
    }
  }
```

This code is used in section 50.

59. Lire les éléments de type SHP_TYPE_POLYGON **et dérivés.**

Les arcs sont inclus dans une bbox, et que ce soit pour des polylignes ou des polygones, de facto il s'agit de la même description de plusieurs arcs.

```
⟨ Read polygon shape 59 ⟩ ≡
  {
    unsigned long delta;
    ⟨ Read bbox 52 ⟩
    Elt→nb_parts ← GET_LE_INT(44);
    Elt→nb_points ← GET_LE_INT(48);
    ⟨ Realloc Elt vectors 56 ⟩
    for (i ← 0; i < Elt→nb_parts; i++) Elt→parts[i] ← GET_LE_INT(52 + i * 4);
    delta ← 52 + Elt→nb_parts * 4;
    for (i ← 0; i < Elt→nb_points; i++) {
      GET_LE_DOUBLE(Elt→X[i], delta + i * 16)
      GET_LE_DOUBLE(Elt→Y[i], delta + i * 16 + 8)
    }
    if (Elt→type ≡ SHP_TYPE_POLYLINE_M ∨ Elt→type ≡ SHP_TYPE_POLYGON_M) {
      delta ← 52 + Elt→nb_parts * 4 + Elt→nb_points * 16;
      GET_LE_DOUBLE(Elt→box[6], delta)
      GET_LE_DOUBLE(Elt→box[7], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→M[i], delta + i * 8)
    }
    if (Elt→type ≡ SHP_TYPE_POLYLINE_Z ∨ Elt→type ≡ SHP_TYPE_POLYGON_Z) {
      delta ← 52 + Elt→nb_parts * 4 + Elt→nb_points * 16;
      GET_LE_DOUBLE(Elt→box[4], delta)
      GET_LE_DOUBLE(Elt→box[5], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→Z[i], delta + i * 8)
      delta += Elt→nb_points * 8;
      GET_LE_DOUBLE(Elt→box[6], delta)
      GET_LE_DOUBLE(Elt→box[7], delta + 8)
      delta += 16;
      for (i ← 0; i < Elt→nb_points; i++) GET_LE_DOUBLE(Elt→M[i], delta + i * 8)
    }
  }
```

This code is used in section 50.

60. Clore les fichiers.

Si le *handler* est valide, on clôt un par un les descripteurs, puis on met à jour le vecteur *_SHP_Handlers* en libérant les ressources désormais inutiles.

Beaucoup de choses pourraient être ajoutées ici, ne serait-ce qu'une fermeture et réouverture en lecture seule en vue de vérifier la consistence des informations écrites.

```
⟨ close.c 60 ⟩ ≡
#include <unistd.h>
#define HIC
#include "_shape.h"
  int SHP_close(struct SHP_File *Shape)
  {
    int status ← G_SUCCESS;
    _CHECK_HANDLER;      /* if header changed, flush it to disk */
    status ← _SHP_write_header(Shape);
    errno ← 0;
    if (_SHP ≠ −1) {
      if (close(_SHP) < 0) status ← errno ≪ G_ERROR_SHIFT;
    }
    if (_SHX ≠ −1) {
      if (close(_SHX) < 0) status ← errno ≪ G_ERROR_SHIFT;
    }
    status ← C_close(&_CORR_DBF);
    ⟨ Free allocated strings 24 ⟩
    free(_SHP_Handlers[Shape→handler]);
    _SHP_Handlers[Shape→handler] ← Λ;
    Shape→handler ← −1;
    −− _SHP_nb_handlers;
    return status;
  }
```

61. Gestion des messages d'information.

Suivant nos conventions, tous les messages sont placés dans un vecteur, qu'il suffira alors, un jour, de particulariser pour obtenir la l10n.

```
⟨ Infos macros 61 ⟩ ≡
#define SHP_IHWRITE 64
#define SHP_INBWRITTEN 65
#define SHP_INOATTS 66
#define SHP_IMAX 66
```

This code is used in section 6.

62.

```
⟨ infos.c 62 ⟩ ≡
#include <stdlib.h>      /* at least NULL */
#include "_shape.h"
  ⟨ C lang infos formats 63 ⟩
  static char *_SHP_istrerror(int msgnum);
  static struct G_Msgs _SHP_Infos ← {"libshape", SHP_IMAX, _SHP_istrerror};
  const struct G_Msgs *const SHP_Infos ← &_SHP_Infos;
  char *_SHP_istrerror(int msgnum)
  {
    return infos[msgnum − G_ERROR_MAX − 1];
  }
```

63. Les messages d'information en langue 'C'.

```
⟨ C lang infos formats 63 ⟩ ≡
  static char *infos[] ← {"Writing the header\n", "Nb of elements written: %i\n",
      "group '%ld' has no attributes\n", };
```

This code is used in section 62.

64. Gestion des messages d'avertissement.

Suivant nos conventions, tous les messages sont placés dans un vecteur, qu'il suffira alors, un jour, de particulariser pour obtenir la l10n.

⟨ Warnings macros 64 ⟩ ≡
```
#define SHP_WPREFIXMAX 64
#define SHP_WLENGTH 65
#define SHP_WTYPEMIXED 66
#define SHP_WMAX 66
```
This code is used in section 6.

65.

⟨ warnings.c 65 ⟩ ≡
```
#include <stdlib.h>    /* at least NULL */
#include "_shape.h"
  ⟨ C lang warnings formats 66 ⟩
  static char *_SHP_wstrerror(int msgnum);
  static struct G_Msgs _SHP_Warnings ← {"libshape", SHP_WMAX, _SHP_wstrerror};
  const struct G_Msgs *const SHP_Warnings ← &_SHP_Warnings;
  char *_SHP_wstrerror(int msgnum)
  {
    return warnings[msgnum − G_ERROR_MAX − 1];
  }
```

66. Les messages d'avertissement en langue 'C'.

⟨ C lang warnings formats 66 ⟩ ≡
```
  static char *warnings[] ← {"prefix too long: '%s' truncated to 8 chars\n",
      "expected '%lu' bytes, wrote '%lu'\n",
      "there are incorrect mixed types. Expecting '%#lx', read '%xlx'\n", };
```
This code is used in section 65.

67. Gestion des erreurs.
Suivant nos conventions, tous les messages sont placés dans un vecteur, qu'il suffira alors, un jour, de particulariser pour obtenir la l10n.

```
⟨ Errors macros 67 ⟩ ≡
#define SHP_EOPENMAX 64
#define SHP_EBADHANDLER 65
#define SHP_EHEADERCODE 66
#define SHP_EHEADERVERSION 67
#define SHP_ESHPREAD 68
#define SHP_ESHPWRITE 69
#define SHP_ESHXREAD 70
#define SHP_ESHXWRITE 71
#define SHP_ETYPE 72
#define SHP_ETYPEMIXED 73
#define SHP_EMODE 74
#define SHP_EEMPTY 75
#define SHP_ESHPSIZE 76
#define SHP_ESHXSIZE 77
#define SHP_ELOST 78
#define SHP_EBADELTID 79
#define SHP_EMAX 79
```

This code is used in section 6.

68.

```
⟨ errors.c 68 ⟩ ≡
#include <stdlib.h>      /* at least NULL */
#include "_shape.h"
  ⟨ C lang errors formats 69 ⟩
  static char *_SHP_estrerror(int msgnum);
  static struct G_Msgs _SHP_Errors ← {"libshape", SHP_EMAX, _SHP_estrerror};
  const struct G_Msgs *const SHP_Errors ← &_SHP_Errors;
  char *_SHP_estrerror(int msgnum)
  {
    return errors[msgnum − G_ERROR_MAX − 1];
  }
```

69. Les messages d'erreur en langue 'C'.

```
⟨ C lang errors formats 69 ⟩ ≡
  static char *errors[] ← {"maximum␣of␣shape␣files␣already␣open\n", "bad␣shape␣handler\n",
      "bad␣header␣code\n", "bad␣header␣version\n", "error␣trying␣to␣read␣shp␣file\n",
      "error␣trying␣to␣write␣shp␣file\n", "error␣trying␣to␣read␣shx␣file\n",
      "error␣trying␣to␣write␣shx␣file\n", "shape␣file␣has␣the␣wrong␣type\n",
      "you␣are␣trying␣to␣mix␣different␣types\n",
      "open␣mode␣incompatible␣with␣requested␣operation\n",
      "the␣file␣requested␣for␣reading␣is␣empty!\n",
      "the␣size␣of␣the␣.shp␣file␣is␣incorrect!\n",
      "the␣size␣of␣the␣.shx␣file␣is␣incorrect!\n",
      "I'm␣lost!␣read/write␣length␣don't␣match␣my␣expectations!\n",
      "the␣element␣id␣passed␣is␣incorrect!\n", };
```

This code is used in section 68.

70. Index. Voici une liste des identifiants. Les entrées soulignées indiquent l'emplacement de la définition.