Vroom/C++ : user interface

E. Abécassis

1.0 Presentation

Vroom is an object-oriented database which offers persistence to C++ objects. The programmers develop C++ applications and Vroom tools ensure persistency on their objects. Specific constructors and destructors are used to manage persistency. Extensions to C++ are offered to customize this language for database applications. The most important extensions are transaction management, index, dynamic collections and clustering.

A Vroom object is just like a classical C++ object but its identity is not acheive through pointer but through logical identifier. Those identifiers allow the programmers to access objects across different sessions. Other particularities of Vroom objects will be detailed later.

2.0 Vroom installation

2.1 Environnement variables

There are four environnement variables in Vroom. They must be set before any use of Vroom.

- VROOM_DIR : absolute path for Vroom installation directory

When you load Vroom from your installation tape, you have a directory ‘VROOM_VER’. The absolute path for this directory must be set in your variable VROOM_DIR.

- SWBHOME : absolute path for Software Bus installation directory

One of the sub-directory of VROOM_DIR is the Software Bus directory. Software Bus (Swb) is the communication tool used by Vroom. You must set your variable SWBHOME with the absolute path of the Swb directory.

- VROOM_BASE : name of the current database

When you want to launch client, server or Vroom tools, they always need a base name. If it is not given, they take VROOM_BASE instead. So be sure to correctly set this variable.

- VROOM_SWAP : Swap directory for Vroom client

Your Vroom clients use a special swap area. This swap file is created as a temporary file in your VROOM_SWAP directory. Be sure that this directory is local to your client host.

Moreover, you must modify two of your default environnement variables :

```
setenv PATH ${PATH}:${VROOM_DIR}/bin:${SWBHOME}/sun4/bin

setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${SWBHOME}/sun4/lib
```

2.2 Software bus installation

To connect your clients and your server, Vroom needs a communication tool. This tool is the Swb.

1. Choose one machine as the communication server. launch locator as background task and then, cdb as background task too. To verify the correct installation, use the command ‘swbs’. This command informs you of the current services on the network. If you have an error message, verify your environnement variables and redo.
2. Launch locator on every machine on your network where you want to use a client or a server.

2.3 Create your database

The first step to build a Vroom/C++ application is to initialise a base. There is a command “vr_init” which creates a new base. This base will store the database schema and persistent objects.

A base is defined by :

- name : use to connect a client to a server which manages this base
- default volumes directory : it is the directory where the default files for the database are created. One default volume is created at the initialisation. This file contents objects created without location specification (cf. cluster)
- system directory : it is the directory where Vroom stores system files.
- log file : it is the directory where Vroom creates log file when needed.
- schema directory : it is the directory where Vroom will generate files for your schema.

All of this information are in a system file named “repert.sys”. This file must be in the directory specified by the environnement variable “VROOM_DIR” .

The exact syntax of the repert.sys file is :

```
<basename>.system     : <system_directory>
<basename>.volume     : <default_vol_directory>
<basename>.log        : <log_file>
<basename>.schema     : <schema_directory>
```

example of file “repert.sys” :

```
testbase.system       : /PROJECT/VROOM_BASE/system_dir
testbase.volume       : /PROJECT/VROOM_BASE/base_dir
testbase.log          : /PROJECT/VROOM_BASE/log_file
testbase.schema       : /PROJECT/VROOM_BASE/schema_dir
```

When your file “repert.sys” is created, you can create the base with the command :

```
vr_init [-b <basename>]
```

If you don’t use the -b parameter, vr_init use VROOM_BASE as the current database.

vr_init can also be used to reinitialise a database.

NB : You will have better performance if you declare volumes and system files on a disk local to you server host.

3.0 Administration tools

3.1 Commands

- vr_init [+s/-s] -b <basename>

Initialises a new base corresponding to <basename> in file repert.sys. If the base already exists, it returns error code -1. You must have no server on <basename> to use vr_init

- vr_admin -b <basename> <schemafile>

Creates classes described in file <schemafile> in the base <basename>. It generates include files for each class described in the schema and an include file <schema>.h. You must have a server on <basename> to use vr_admin

- vr_shell -b <basename> [<filename>]

Allows interactive construction of volumes and clusters. It is also the only way to set a class in a cluster. You must have a server on <basename> to use vr_shell

- vr_server [-b <basename>]

Launchs the vroom server on base <basename>

3.2 vr_init

When you have declared your base in 'repert.sys' file, you can use the command vr_init to create your database. When you have already a database and you want to clean it, you can use this command. In this case, you have two choices :

1. "+s' : Only delete the data.
2. "-s" : Delete all data, volumes, clusters and classes

The default is "-s".

3.3 vr_admin

When your database is initialised, you must create a schema. A schema contents the description of your classes.

At the beginning, your base is empty. You must, at least, create one class. You create a a set of classes and later add new classes in your base by using the same command :

```
vr_admin -b <basename> <filename>.sch
```

The file "filename.sch" contents the description of your classes. The first part of the description is the class definition. A class definition looks like C++ except that you must follow some rules (cf class description). If you need collections of instances, you must declare them in the description file.

example of a description class file “employee.sch” :

```
class Employee : public Person {
public :
     int               salary ; // C++ int
     oz_Dept           dp ; // ref to a persistent department
     oz_list(Project)  proj_l ; // must be declared

     void increase_salary(int percent) ;
     void migrate(oz_Department) ;
} ;

classdef SeqX(Employee) ;
classdef ListX(Employee) ;
```

vr_admin generates few files :

- <filename>.h : contains all the declarations of your file <filename>.sch.
- oz_<filename>.h : contains all the declarations of references and collections.
- g_<filename>.cc : contains the C++ code to manage persistence for your classes.
- oz_<classname>.h : contains the declaration of the reference for class <classname>.
- oz_<listname>.h : contains the declaration of the list <listname>.
- oz_<seqname>.h : contains the declaration of the sequence <seqname>.
- oz_<dictname>.h : contains the declaration of the dictionary <dictname>.
- <basename>.cc : This file will be unique for a database.

3.4 Vroom shell

It is a command shell to manage the organisation of your database. Only few functionnalities are avaible today.

1. Volumes and clusters

```
create volume <volname> [in file <filename>]
move volume <volname> in file <filename>
delete volume <volname>
delete volumes
display volumes [<volname>*]
create cluster <clustername> [in volume <volname>]
delete cluster <clustername>
delete clusters
display clusters [<clustername>*]
```

Volumes are useful to distribute your base across different file systems. However, your interest is to localise volumes of a database close to the server which manages them.

When you create a volume, you must create a cluster into. When you want to store instances of a class in a specific volume, you just have to declare the class in this cluster.

Clusters are used to group objects in the same set of pages. For example, if you want to create a class Person and a class Vehicle and you know that you will work often on Person and Vehicle at the same time, VROOM offers better performance if you clusterise these two classes in the same cluster. When VROOM will load a page from the disk, it only contains Vehicle and Person and not useless objects.

Creation of a volume without file specification is created in your default volumes directory declared in your database.

Creation of a cluster without volume specification is created in your default volume.

2. Classes

```
move classes <classname> in cluster <clustername>
display classes [<classname>*]
```

The vr_shell allows the administrator to organize the localisation of the classes. When you create a class with vr_admin, every instances of this class are created in default cluster in default volume. If you want that new instances will be created in a specific cluster, you can use the command ‘move classes’ to do it.

3. Names

```
display names [<name>*]
delete name <name>
delete names
```

The external root of your database is the named object. To inspect or delete such names you can use vr_shell.

4.0 Description of a class

Each class must have a description. This description contents C++ definition and declaration of collections.

4.1 cluster of a class

For every classes of your schema, you can specify a cluster at any time. This cluster must exist before the link edition of your application. Then, every time you will create an instance of this class, it storage space will be allocated in the set of pages of the specified cluster. The purpose of this mechanism, is to group in the same disk pages, the objects which are used together.

4.2 Definition of a class

In the description file, you must give the complete definition of the class. The Vroom/C++ definition is just like C++ definition except for certain particularities :

- Attributes could be one of the following types
  - char
  - unsigned char
  - int
  - unsigned int
  - short
  - unsigned short
  - long
  - unsigned long
  - float
  - double
  - enum type
  - pointer
  - StringX
  - ByteX
  - <persistent object>
  - Reference to SeqX(<persitent object>)
  - Reference to ListX(<persistent object>)
  - Reference to DictX(<persistent object>)
  - Reference to SeqX(<int>)
  - Reference to ListX(<int>)
  - Reference to SeqX(<float>)
  - Reference to ListX(<float>)
  - Reference to SeqX(<StringX>)

- Reference to ListX(<StringX>)
- C++ array of authorised type

- Overloading of new and delete are not allowed
- Use of pointer on persistent classes is prohibited
- The size of an instance of a Vroom class must be less to 3900 bytes
- Basic types (int, char, float, StringX...) cannot be used in variable arguments functions (ex: printf).
- C++ Array of basic types cannot be used as a contiguous memory space.

4.3 collections

If you need collections of class instances in your application, you must declare them. For example, if you have a department object, with an attribute which is a set of employees you must declare the set of Employee in Employee description file.

You have3 different types of collections :

- sequence : a dynamic array

```
classdef SeqX(<classname>)
```

- list : An ordinate collection with insertion and suppression capabilities

```
classdef ListX(<classname>)
```

- Dictionary : An ordinate collection indexed by a user defined key.

```
classdef DictX(<classname>, <attr_method_key>, <key_type>)
```

By default, you have SeqX, DictX, ListX of int, float, StringX, ByteX.

```
ex :    oz_ListX(Person) l_pers = new ListX(Person) ;
```

4.4 An complete example

```
ex :
// class description for Person
        class Person {
        public :
            int             age ; // C++ int
            oz_Person       live_with ; //Ref to persistent person
            oz_ListX(Person) l_child ; // list of person
        public :
            void new_child(oz_Person) ;
        } ;
        classdef ListX(Person) ;

// class description for Employee

        class Emp : public Person {
        public :
            int               salary ; // C++ int
            oz_Dept           dp ; // ref to a persistent department
            oz_ListX(Project) proj_l ; // must have been declared

            void increase_salary(int percent) ;
            void migrate(oz_Dept) ;
        } ;
        classdef ListX(Emp) ;
        classdef SeqX(Emp) ;
```

5.0 Vroom types

5.1 Basic types

To manage consistency, VROOM must detect updates during the execution. To do that, it overload every C++ basic types and replaces them by corresponding classes which can detect updates. However, it will be transparent for the user thanks to the implicit conversion system of C++.

A special case is pointer. Pointers have no sense in a persistent environment. That is why VROOM ignores them and don't try to catch update on them. You can freely modify pointer attribute without consistency and concurrency management. So your pointers have only transaction life.

The conclusion is that you can use almost every basic C++ types and VROOM will put control on them. To do that, VROOM does some conversion on your declaration files :

Table of correspondance between C++ types and Vroom types. "Vroom C++ type" is the UNIX/C++ type chosen to store the data :

your C++ type	**Vroom C++ type**	**Vroom type**
char	char	oz_char
unsigned char	unsigned char	oz_uchar
int	long	oz_long
unsigned int	unsigned long	oz_ulong
long	long	oz_long
unsigned long	unsigned long	oz_ulong
short	short	oz_short
unsigned short	unsigned short	oz_ushort
float	float	oz_float
double	double	oz_double
long double	long double	oz_ldouble
pointer	pointer	pointer

example :

```
class Person {
    int    age ; // Persistent data
    Car*   car ; // Temporary data
    void   move() ;
} ;
```

```
class Person {
    oz_long   age ;
    Car*      car ;
    void      move() ;
} ;
```

Note that because VROOM already uses conversion system, some of your own conversions could need two level of casting to work.

```
ex :
        void* ptr = (void*)emp->salary ; // error !!!
        void* ptr = (void*)(int)emp->salary ; // OK
```

5.2 Enum type

You can declare attributes of a class as enum type. For that, you must use a typedef for this enum type and specify in your class that the typedef is an enum. Vroom will encapsulates the attribute in a nested class. This class has conversion functions to enable use of this attribute as an enum.

```
class ... {
     ...
     EnumX(<enum_type>)   <attribut_name> ;
     ...
} ;
```

```
ex :
     typedef enum {SMALL, MEDIUM, LARGE} proj_size ;

     class Project {
          oz_dept           dept ; // ref to a persitent department
          oz_emp            leader ; // ref to a persistent Employee
          EnumX(proj_size) size ;
     } ;
```

5.3 Nested objects

You can declare an attribute to be an instance of a class. This class must be declared as a persistent class and must be in your schema declaration. This object will be local and no persistent reference are possible on it. You can access to it only through its father.

5.4 Reference to persistent object

One of the most important feature of Vroom/C++ is the possibility to use persistent pointers on your objects. Your objects will be persistent. So you cannot use standard C++ pointers which have a sense only during a session. Instead of pointer, you can use reference on persistent object.

For every class declared in your schema, Vroom generates another class which is a smart pointer on your class. You can use instances of smart pointer class just as classical C++ pointer. These pointers are intelligent and can always find the referenced object.

You can use smart pointer for an attribute of a class or like a pointer variable in your code. The name of the smart_pointer is oz_<classname>. The generated class is :

```
class oz_<type> {
     // constructors
     oz_<type>() ; // empty constructor
     oz_<type>(oz_<type>&) ; // copy constructor
     oz_<type>(<type>*) ; // copy constructor
     oz_<type>(StringX) ; // constructor for named object (ref)

     // pointer assigment
     oz_<type>&  operator=(oz_<type>&) ;
     oz_<type>&  operator=(<type>*) ;

     // pointer operators
     <type>*     operator->() ;
     <type>*     operator()() ;
     <type>&     operator*() ;

     // equal and diff
     int         operator==(oz_<type>&) ;
     int         operator==(<type>*) ;
     int         operator!=(oz_<type>&) ;
     int         operator!=(<type>*) ;

     // delete object
     void        destroy() ;
} ;
```

```
ex :
     class Project {
          vr_dept          dept ; // ref to a persitent department
          vr_emp           leader ; // ref to a persistent Employee
          int              team_size ;
          enum(proj_size)  size ;
     } ;

     class oz_Project {
          // constructors
          oz_Project() ;
          oz_Project(oz_Project&) ;
          oz_Project(Project*) ;

          // pointer assigment
          oz_Project& operator=(oz_Project&) ;
          oz_Project& operator=(Project*) ;

          // pointer operators
          Project*    operator->() ;
          Project*    operator()() ;
          Project&    operator*() ;

          // equal and diff
          int         operator==(oz_Project&) ;
          int         operator==(Project*) ;
          int         operator!=(oz_Project&) ;
          int         operator!=(Project*) ;

          // delete object
          void        destroy() ;
     } ;

     void foo(oz_emp new_lead, int new_team_s) {
          oz_Project proj = new Project ; // Create a new project
          proj->leader = new_lead ; // copy reference
          proj->team_size = new_team_s ; // copy value
          if (proj->team_size < 10) {
               proj->size = SMALL ; // copy value
          } else {
          if (proj->team_size < 20) {
               proj->size = MEDIUM ;
          } else {
```

```
        proj->size = LARGE ;
        } }
    } ;
```

Note that you can use a smart pointer just like a pointer if you use a smart_pointer object or a reference to a smart pointer object. If you use a pointer to a smart pointer, you must first access to the value of the smart pointer by *. That's a reason why it is better to always use smart pointer object and not pointer to a smart pointer :

```
ex :
    oz_Project* proj = new oz_Project ; // create a smart pointer
    *proj = new Project ;   // create a persistent Project
    (*proj)->size = SMALL;  //access to persitent Project
```

NB : C++ ATT 2.1 cannot manage Vroom oz_pointer in few cases :
- 'for' condition
- 'while' condition
- 'if' with two or more conditions

In such case you must do it in two steps

```
oz_Dept dept= new Dept ;
int i =0 ;
for (; i<dept->size;i++) {
   <statements>
}
```

```
oz_Dept dept= new Dept ;
int i = 0 ;
int size = dept->size ;
for (;i<size;i++) {
   <statements>
}
```

5.5 Variable string 'StringX'

Vroom offers a type to manage variable strings. This type has a size limit of 3900 bytes The interface of this type is :

```
class StringX {
public :
    // constructors
    StringX() ; // Construction of an empty stringX
    StringX(char *) ; // Copy constructor
    StringX(StringX&) ; // Copy constructor

    // assignement (value copy)
    StringX&    operator=(char *) ;
    StringX&    operator=(StringX&) ;
```

```
    // direct assignement to avoid previous concatenation
    void assign(char*) ;
    void assign(char*, char*) ;
    void assign(char*, char*, char*) ;
    void assign(char*, char*, char*, char*) ;
    void assign(char*, char*, char*, char*, char*) ;
    void assign(char*, char*, char*, char*, char*, char*) ;

    // append
    StringX&    operator+=(StringX& ) ;
    StringX&    operator+=(char*) ;

    // length of the string
    int         strlen() ;

    // copy content of the string into char *
    // the user must free the allocated char*
    char*        strdup() ;

    // accessing to the ieme element of the string
    char&       operator[](int i) ;

    // get a substring
    StringX     operator()(int index, int count) ;

    // comparing 2 strings by value
    int operator==(char *);
    int operator==(StringX&);
    int operator!=(char *);
    int operator!=(StringX&);
    int operator<(char *);
    int operator<(StringX&);
    int operator<=(char *);
    int operator<=(StringX&);
    int operator>(char *);
    int operator>(StringX&);
    int operator>=(char *);
    int operator>=(StringX&);

    // friend operator
    friend StringX      operator+(StringX&, StringX&) ; // concat
    friend ostream&     operator<<(ostream&,StringX&);
    friend istream&     operator>>(istream&,StringX&);
} ;
```

5.6 Variable bytestring 'ByteX'

Vroom offers a type to manage variable bytestrings. It is a container to store long byte string in your Vroom database. You can store bytestring up to 2 Go. The interface of this type is :

```
class ByteX {
public :
      // constructors
      ByteX() ;
      ByteX(int len) ;
      ByteX(ByteX&) ;

      // assignement (value copy)
      ByteX&        operator=(ByteX&) ;
      ByteX&        assign(char*, int len) ;

      // append
      ByteX&        operator+=(ByteX& ) ;
      ByteX&        append(char*, int len) ;

      // length of the bytestring
      int           len() ;

      // copy content of the byte string into [from] s
      // first and count indicate the substring which is copied
      char*         copyto(char* s, int first,int count) ;
      ByteX&        copyto(ByteX& s, int first,int count) ;
      char*         copyfrom(char* s, int first, int count) ;
      ByteX&        copyfrom(ByteX& s, int first, int count) ;

      // accessing to the ieme element of the bytestring
      char&         operator[](int i) ;

      // comparing 2 bytestrings
      int operator==(ByteX&);
      int operator!=(ByteX&);
      int compare(char*, int len) ;
} ;
```

5.7 vroom collections

Vroom offers generic collections. To make use of a collection of objects, you must declare the class in your schema except for basic types collection which are defined in your default environnement. In the description file, you must declare the collections :

```
classdef ListX(<class_name>) [listname] ;
classdef SeqX(<class_name>) [seqname] ;
classdef DictX(<class_name>, <key>, <key_type>) [dictname] ;
```

If you don't specify a name for your collection, you can use coltype(<classname>) in your code.

Then you can use this collection type in your application, as variable or as class attribute. To use collection in your code, you address theim by smart pointer. For that purpose, Vroom generates oz_colname for each of your collections. Collections are new constructors generated by Vroom and are not real C++ classes. So you cannot derived collections into subclasses of collection.

When you use operator on collection, you can apply theim directly to the smart pointer. If you use methods, you call theim by the classical '->'.

```
ex :
      class Person {
      public :
           int              age ; // C++ int
           StringX          name ;
           oz_Person        live_with ; //Ref to persistent person
           oz_ListX(Person) l_friend ; //Ref to a list of person

           Person(StringX name) ;
           void new_child(oz_Person) ;
      }

      void foo(oz_person& father) {
           ...
           oz_SeqX(Person) s_pers = new SeqX(Person) ; // create a col
           s_pers += new Person("john") ; // add element
           s_pers += new Person("marie") ;
           father->l_friend = s_pers ; // copy reference, not value
           ...
      } ;
```

Here is the description of the interface of the different collections :

5.7.1 List

To define a list, you must declare in a schema file :

```
classdef ListX(<classname>) [listname] ;
```

```
class oz_ListX(type) {
public :
      // Constructors
      oz_ListX(type)() ; // create an reference to a list
      oz_ListX(type)(oz_ListX(type)&) ; // copy constructor
      oz_ListX(type)(ListX(type)*) ; // copy constructor
      oz_ListX(type)(StringX) ; // constructor for named object (ref)

      // comparing two lists by identity
      int operator==(oz_ListX(type)&) ;
      int operator!=(oz_ListX(type)&) ;

      // append at the end of the list
      oz_ListX(type)   operator+=(oz_ListX(type)&) ;
      oz_ListX(type)   operator+=(oz_type&) ;

      // concatenation. Create a temporary list as result.
      oz_ListX(type)   operator+(oz_ListX(type)) ;

      // access to the ieme element
      oz_type&         operator[](int) ;
}

class ListX(type) {
public :
      // comparing two lists by value
      int              equal(oz_ListX(type)&) ;
      int              diff(oz_ListX(type)&) ;

      // value copy
      void             copyto(oz_ListX(type)&) ;
      void             copyfrom(oz_ListX(type)&) ;

      // insert an element after position pos
      // if pos == COL_TAIL the element is inserted at the end
      // if pos == COL_HEAD the element is inserted at the beginning
      void             insert(oz_type&, int pos) ;
```

```
        // remove an element
        void             remove(int pos) ;
        // test if an object is in collection, return the position or -1
        int              member(oz_type&) ;

        // cardinality
        int              count() ;

        // empty list
        void             clear() ;

        // apply a method of the element class on every elements
        void             apply(void (type::*func)()) ;

        // apply a method of the element class with boolean result
        // insert every element with true result in a temporary list
        oz_ListX(type)   select(int (type::*func)()) ;
    } ;
```

5.7.2 Sequence

To define a sequence, you must declare in a schema file :

```
        classdef SeqX(<classname>) [seqname] ;
```

```
class oz_SeqX(type) {
public :
        // Constructors
        oz_SeqX(type)() ; // create an reference to a sequence
        oz_SeqX(type)(oz_SeqX(type)&) ; // copy constructor
        oz_SeqX(type)(SeqX(type)*) ; // copy constructor
        oz_SeqX(type)(StringX) ; // constructor for named object (ref)

        // comparing two sequences by identity
        int operator==(oz_SeqX(type)&) ;
        int operator!=(oz_SeqX(type)&) ;

        // append at the end of the sequence
        oz_SeqX(type)    operator+=(oz_SeqX(type)&) ;
        oz_SeqX(type)    operator+=(oz_type&) ;

        // concatenation. Create a temporary sequence as result.
        oz_SeqX(type)    operator+(oz_SeqX(type)) ;
```

```
      // access to the ieme element
      oz_type&         operator[](int) ;
}

class SeqX(type) {
public :
      // comparing two sequences by value
      int              equal(oz_SeqX(type)&) ;
      int              diff(oz_SeqX(type)&) ;

      // value copy
      void             copyto(oz_SeqX(type)&) ;
      void             copyfrom(oz_SeqX(type)&) ;

      // test if an object is in collection. return the position or -1
      int              member(oz_type&) ;

      // cardinality
      int              count() ;

      // empty sequence
      void             clear() ;

      // apply a method of the element class on every elements
      void             apply(void (type::*func)()) ;

      // apply a method of the element class with boolean result
      // insert every element with true result in a temporary seq
      oz_SeqX(type)    select(int (type::*func)()) ;
} ;
```

5.7.3 Dictionary

To define a dictionary, you must declare in a schema file :

```
      classdef DictX(<classname>,<key>,<key_type>) [name] ;
or    classdef DictX(<classname>,<key>,<key_type>,<multiple>) [name];
```

The key must be an attribute or a methode of the class <classname>.

The key can be of any type which allow ‘<‘ , ‘>’ and ‘==’.

<key_type> is the classname or typename of the key.

multiple : ALL or DISTINCT specify if you can have two or more elements of the dictionary which have the same key.

To declare a dictionary on a class <classname> you must declare first ListX(classname) in your schema.

```
class oz_DictX(type) {
public :
      // Constructors
      oz_DictX(type)() ; // create a reference to a dictionary
      oz_DictX(type)(oz_DictX(type)&) ; // copy constructor
      oz_DictX(type)(DictX(type)*) ; // copy constructor
      oz_DictX(type)(StringX) ; // constructor for named object (ref)

      // comparing two dictionaries by identity
      int operator==(oz_DictX(type)&) ;
      int operator!=(oz_DictX(type)&) ;

      // insert in the dictionary by using the key
      oz_DictX(type)   operator+=(oz_type&) ;

      // access to the ieme element
      oz_type          operator[](int) ;
}

class DictX(type) {
public :
      // comparing two lists by value
      int              equal(oz_DictX(type)&) ;
      int              diff(oz_DictX(type)&) ;

      // value copy
      void             copyto(oz_DictX(type)&) ;
      void             copyfrom(oz_DictX(type)&) ;

      // access to the first element in the dict with key
      oz_type          get(KEY key) ; // if DISTINCT
      oz_ListX(type)   get(KEY key) ; // if ALL

      // Return position of the first element which matches the key
      // You can also address the dictionary as a list
      int              position(KEY key) ;
```

```
    // put an element by using its key
    // if DISTINCT. return 0 if the key already exists else 1
    void             put(oz_type&) ; // if ALL
    int              put(oz_type&) ; // if DISTINCT

    // put an element by using its key.
    // The object is not inserted if there is already an object with
    // same key. In that case, the method return 0 else 1.
    int              puttUniq(oz_type&) ;

    // remove an element
    void             removeElement(oz_type&) ;

    // remove every elements of key 'Key'
    void             removeByKey(KEY key) ;

    // test if an object is in collection. Return the position or -1
    int              member(oz_type&) ;

    // cardinality
    int              count() ;

    // empty list
    void             clear() ;

    // apply a method of the element class on every elements
    void             apply(void (type::*func)()) ;

    // apply a method of the element class with boolean result
    // insert every element with true result in a temporary list
    oz_ListX(type)   select(int (type::*func)()) ;
} ;
```

5.7.4 Iterator

Vroom offers iterator for collections.

```
// iterator for list and sequence
class Iterator(type) {

    // constructor for sequence
```

```
      Iterator(type)(oz_SeqX(int) seq, int beg = 0, int end = -1) ;

      // constructor for list
      Iterator(type)(oz_ListX(int) list, int beg = 0, int end = -1) ;
      // get the current element
      // return NULL oz_pointer if it is the end of the scan
      oz_type& get() ;

      // return the current element and move cursor to the next one.
      // return NULL oz_pointer if it is the end of the scan
      oz_type& next() ;

      // return 0 if the cursor is at the end position else return 1
      int more() ;

      // return the current position of the cursor
      int where() ;
} ;

// iterator for dictionary
class Iterator(DictX(type)) {

      // constructor with absolute positions
      Iterator(DictX(type))(oz_DictX(type), int beg=0, int end=-1) ;

      // constructor with range given by key
      Iterator(DictX(type))(oz_DictX(type), KEY beg, KEY end) ;

      // get the current element
      // return NULL oz_pointer if it is the end of the scan
      oz_type& get() ;

      // return the current element and move cursor to the next one.
      // return NULL oz_pointer if it is the end of the scan
      oz_type& next() ;

      // return 0 if the cursor is at the end position else return 1
      int more() ;

      // return the current position of the cursor
      int where() ;
} ;
```

5.8 C++ array

You can declare C++ array in your class. The type of the array must be an authorized type.

```
ex :
      class Dept {
           StringX               name ;
           oz_Project            tab_proj[10] ;
      } ;
```

5.9 Struct, bitfield, union, typedef

For this version of VROOM, struct, bitfield, union and typedef are forbidden in the declaration of a class.

6.0 Programming with Vroom/C++

6.1 Sessions and Transactions management

- Connection :

```
int vr_connect(char* <basename>) ;
```

In your application, you must connect your process with the server. For this purpose, you must use vr_connect. <basename> is the name of the base on which you want to be connected. It starts a current transaction. The result of vr_connect is 0 if everything is OK. Elsewhere, it returns -1.

- Disconnection :

```
int vr_disconnect() ;
```

Disconnect you from current base and realize a commit on current transaction.

- Commit :

```
int vr_commit() ;
```

Commit the current transaction. Every update is archived on the server and your locks are released. it starts a new current transaction.

- Abort :

```
int vr_abort() ;
```

Abort the current transaction. Every update is discarded and your locks are released. It starts a new current transaction.

6.2 Construction of Vroom/C++ objects

When your base and your schema are ready, you can use the generated file to create an application. Your first need is to create persistent objects. Remember that persistent classes must be accessed only through oz_pointer except in a member function.

To create an object which have a Vroom/C++ class you can :

1. Create a persitent object

```
oz_Person pers1 = new Person ; // create a persistent person
oz_Person pers2 = new(VR_PERS) Person ; // create a persistent person
oz_Person child = new(pers1) Person ; // create a persistent person
```

In the first line, the new operator for class Person creates a persitent object by default. The result is a pointer on the persistent object which cannot be used directly. That is why we declare a oz_Person to receive the pointer. This object “pers” can be used as a normal pointer on a Person object.

In the second line, the effect is the same as previously but the parameter of new makes it explicit.

The third line shows creation of an new object which is persistent and stored physically closed to the object passed as parameter of the new. This creation can improve performance of the system.

2. Create a temporary object

```
ex : oz_Person pers = new(VR_TEMP) Person ; // create a temporary
person
```

The parameter of new indicates that we want a Vroom/C++ temporary object. This parameter is a enum type defined in Vroom include file :

```
typedef enum { VR_TEMP, VR_PERS} VR_Status ;
```

Such an object is just like a persistent object except that it will be automatically deleted at the end of the transaction. At any time before the end of the current transaction, it could be made persistent by calling methods “persistent()” on the object.

```
ex : pers->persistent() ; // now it is persistent
```

3. Create a local object

```
ex :
      void foo() {
           ...
           Person pers ; // local object
           ...
      } ;
```

In that case,’pers’ is not a Vroom object. The object is created by C++ in the current stack and will be deleted at the end of the scope. There is no way to make it persistent.

6.3 Named Objects

To retreive objects from your database, you must have root objects that you can access without an OID (internal object identifier). To do that, Vroom allows to name objects and get objects by its name.

To name object, you have a default method :

```
      void ObjectX::setName(StringX) ;
ex :
      oz_Dpt dpt = new Dpt ;
      dpt->setName(“R&D”) ;
```

To retreive your objects by their name, you can use a special constructor of oz_pointer.

```
oz_type(StringX&)
```

```
ex :
```

```
oz_Dpt dpt("R&D") ;
```

6.4 Concurrency mode

To manage concurrency, Vroom offers a two-phase locking transaction. Your transaction begins with you connection and ended with the call of vr_commit, vr_abort or vr_disconnect. Except for vr_disconnect, the end of a transaction is the beginning of a new one.

During the transaction, every manipulation of objects imply locks. The compatibility of the locks depend of the lock mode of the object.

1. Lock 1 write or N read

In this mode, only one transaction can have write lock on an object. If there is a write lock, no one can access to this object even in read mode. This mode ensures complete isolation of the different transactions and every transaction works on the last commit state of the database.

2. Lock 1 write and N read

In this mode, the system allows only one transaction to have a write lock but, in the same time, N transactions can read the last stable version of the object.

3. Lock N write and N read

In this mode, the system makes no control on locks and the last writer is the winner.

4. Lock depend

When an object has a lock dependance to another, every lock request on the object are transfered to the other object. It is useful for complex object. When you have a set of Vroom/C++ objects linked together which are a unique complex object at conceptual level, you can have one object which is the lock object for the set. Every lock on every objects will be transfered to the lock object and your set of objects will be completly locked or not locked.

6.5 Exception management

To manage errors during the execution of a Vroom session, you can use the Vroom exception management. When an error appends, if the system can detect it, it throw an error. To catch it, you must have declared a try block.

```
try block :
      try {
           <statements>...
      } exception {
           switch (__ERR) {
                        case <error> : <keyword>
                        case...
           }
      }
```

The try block can be declared at every level in your code and can be nested.

To handle the error, you have few keywords

UP : transfert the error to the upper try block

RETRY : redo the statements of the block try

WAIT : in the case of a lock violation, you can wait that the lock is free

CALL(function) : call the function and return after the exception block.

_ERR can be :

```
VR_kERR_NO_MEM : Problem in memory managment
VR_kERR_LOCK_VIOL : Lock violation
VR_kERR_DEADLOCK : Dead lock in the wait graph
VR_kERR_NULL_OID : Reference to a null object
VR_kERR_STRING : Out of bondaries in string access
VR_kERR_NAME_OBJ : Name already used.
```

6.6 Default methods

Every classes that user defined inherit from ObjectX. This class has few methods available for every subclasses.

```
void ObjectX::setName(StringX&) ;
```

Assign a name to the object. The object can be later accessed by its name with oz_ObjectX constructor ‘oz_ObjectX(StringX&)’

```
void ObjectX::persistent() ;
```

Modify the status of a temporary object. At the end of the transaction, the object will be saved just as if it was created as persistent object.

```
void ObjectX::lock1orN() ;
```

Set lock mode on the object on 1 Write or n Read.

```
void ObjectX::lock1andN() ;
```

Set lock mode on the object on 1 Write and n Read

```
void ObjectX::lockNandN() ;
```

Set lock mode on the object on n Write and n Read

```
VR_LockMode ObjectX::lockMode() ;
```

Return the current lock mode of the object.

```
void ObjectX::lockDepend(oz_ObjectX& obj) ;
```

Every lock request on the receiver will be done on the object ‘obj’. It makes lock dependents of the lock of the object ‘obj’.

7.0 An example

```
// file essai1.sch
#include “oz_Dpt.h”

class Employe {
public :
 StringX name ;
 int age ;
 int salary ;
 oz_Dpt ref_dpt ;

 Employe() ;
 Employe(int _age, int _sal, oz_Dpt dpt) ;
 virtual void display() ;
 void new_year(int increase) ;
} ;

classdef ListX(Employe) ;
classdef DictX(Employe, name, StringX) DictEmpName ;


// file essai2.sch
#include “essai1.h”

class Engineer : public Employe {
 public :
 Engineer() ;
 Engineer(int _age, int _sal, oz_Dpt dpt) :
      Employe(_age, _sal, dpt) {} ;
 virtual void display() ;
} ;

class Leader : public Employe {
 public :
 Leader (int _age, int _sal, oz_Dpt dpt) :
      Employe(_age, _sal, dpt) {} ;
 virtual void display() ;
} ;

class Dpt {
public :
 int                  ident ;
 oz_ListX(Employe)    emp ;

 Dpt(int id) ;
```

```
 void displayAll() ;
} ;


// file essai.cc
#include "stream.h"
#include "essai1.h"
#include "essai2.h"

Employe::Employe(int _age, int _sal, oz_Dpt dpt) {
 this->age = _age ;
 this->salary = _sal ;
 this->ref_dpt = dpt ;
}

void Employe::display() {
 cout << "\tAge : " << this->age << "\n" ;
 cout << "\tSalary : " << this->salary << "\n" ;
 cout << "\tName : " << this->name << "\n" ;
} ;

void Engineer::display() {
 cout << "Engineer" ;
 this->Employe::display() ;
} ;

void Leader::display() {
 cout << "Leader" ;
 this->Employe::display() ;
} ;

void Employe::new_year(int augment) {
 this->age += 1 ;
 this->salary += this->salary / augment ;
}

void Dpt::displayAll() {
 int size = emp->count() ;
 for (int i = 0; i < size; i++) {
      emp[i]->display() ;
 }
}
```

```
void main(int argc, char** argv) {

 oz_connect("test") ;

 oz_Dpt dpt ;

 if ( (argc == 2) && (argv[1][0] == 'c') ){
      dpt = new Dpt ;
      dpt->setName("dtrd") ;
      dpt->ident = 5 ;
      dpt->emp = new ListX(Employe) ;

      dpt->emp += new Employe(0, 1000, dpt) ;
      dpt->emp[0]->name = "John" ;
      dpt->emp[0]->age = 0 ;
      dpt->emp += new Engineer(1, 1000, dpt) ;
      dpt->emp[1]->name = "Steve" ;
      dpt->emp[1]->age = 1 ;
      dpt->emp += new Leader(2, 1000, dpt) ;
      dpt->emp[2]->name = "Alan" ;
      dpt->emp[2]->age = 2 ;
      dpt->emp += new Engineer(3, 1000, dpt) ;
      dpt->emp[3]->name = "Robert" ;
      dpt->emp[3]->age = 3 ;
 } else {
      oz_Dpt temp("dtrd") ;
      dpt = temp ;
 }

 dpt->displayAll() ;

 oz_commit() ;

 dpt->displayAll() ;

 oz_disconnect() ;

} ;
```