MASTER IN COMPUTER SCIENCE

A contribution to C++ database development methodology

Lambers, Pascal

Award date:
1998

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX,

NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

A contribution to C++ database development methodology

Pascal Lambers

Mémoire présenté en vue de l'obtention du grade de

Licencié en Informatique

Année Académique 1997 - 1998

For the accomplishment of this thesis, I would like to thank:

- my promotor, Mr J.L.-Hainaut, who has always been available to steer me into the right direction;
- Mr V. Englebert, for his inexhaustible technical assistance concerning C++, database models and Voyager 2;
- Kirsten Bruyneels for her linguistic assistance on rereading this essay;
- to my parents, who have been paying my education for years but whom I hope to release from this burden before long;
- to all the people from the DB-MAIN team I have besieged with questions on the few occasions I did not find J.L.-Hainaut or V. Englebert.

ABSTRACT

This work is dedicated to the development of a methodology to design C++ object oriented databases. Starting out from a conceptual entity-relationship model, the investigation leads towards the definition of the C++ logical and physical models. For both schemata, particular attention is paid to the transformations that have to be performed to obtain them and the support the DB-MAIN CASE tool offers to do so. Then, the structures appearing in the physical model are translated into C++ code. The last chapters present a CASE tool that was developed to generate the C++ code corresponding to the formerly defined physical model. Finally, the generated code and the corresponding instructions are presented.

RESUME

Ce travail est consacré à la recherche d'une méthodologie de développement d'une base de données C++. A partir d'un schéma conceptuel entité-association, la recherche permet de définir des modèles C++ logique et physique. Pour les deux schémas, beaucoup d'importance a été accordée aux transformations qui doivent être faites pour les obtenir et au support que l'outil CASE appelé DB-MAIN offre pour le faire. Ensuite, les structures qui apparaissent dans le modèle physique sont traduites en code C++. Les derniers chapitres sont consacrés à la création d'un outil CASE pour générer le code C++ correspondant au modèle physique défini précédemment. Finalement, le code généré est présenté avec son mode d'emploi.

CONTENTS

I. Introduction 1
II. Context reminder 3
II.1. Database Development Methodology 3
II.2. The Conceptual Model 6
II.2.1. A General Approach of the Conceptual Model 6
II.2.2. Note on the Conceptual Model with regard to C++ 9
II.3. The Logical Model 9
II.3.1. The General Abstract Model 9
II.3.2. The C++ Logical Object Oriented Schema 9
II.4. Computer Aided Software Engineering 11
III. Declarative Structures in C++ Classes 13
III.1. Entity Types 13
III.2. Attributes and Attribute Types 14
IV. Logical Design of an Object Oriented Schema 17
IV.1. Analysis: 17
IV.2. A closer Look at the Transformations 17
IV.3. Transformation Plan 21
V. Physical Design of a C++ Object Oriented Schema 23
V.1. The Physical Schema 23
V.2. Translation of the Physical Schema 23
V.2.1. Object Collections 23
V.2.2. Identifiers 25
V.2.3. Object Attributes 26
V.2.4. Cardinality Constraints 30
V.2.5. Types 31
VI. The DB-MAIN CASE Tool 33
VI.1. Assistance for Simple transformations: 33
VI.2. The Assist Menu 34
VI.3. Other useful Functions 36
VII. The C++ Generator 37
VII.1. A Restricted Model 37
VII.1.1. Overview of the Structures the C++ Generator can handle: 37
VII.1.2. Extra restrictions on the C++ object oriented logical model: 38
VII.1.3. Analysis Script to Validate the Schema. 39
VII.2. Chronological Overview of the Operations Performed by the Programme 40
VII.3. Some Technical Details of the C++ Generator. 41
VIII. Case Study 43
VIII.1. The Generated C++ Code 43
VIII.2. An Example 47
VIII.2.1. Conceptual design 47
VIII.2.2. Logical design 48
VIII.2.3. Physical design 49
VIII.2.4. Generating code 50
IX. Conclusion 53
X. Bibliography 55

Appendices:

I. The Source Code of the C++ Generator (Lg.: Voyager 2) 59
II. The Generated C++ Code. Example 1 85
II.1. Physcial Schema 85
II.2. The C++ Code Corresponding to the Schema 86
III. The Generated C++ Code. Example 2 93
III.1. Physical Schema 93
III.2. The C++ Code Corresponding to the Schema 94

Graphical presentations:

Figure II.1 From conceptual to physical schemata. 5
Figure II.2 Entity types: subtypes and supertypes 6
Figure II.3 Relationship types 7
Figure II.4: Entity type CLIENT. 7
Figure II.5: Identifiers. 8
Figure II.6 The inverse constraint 8
Figure II.7 An object oriented logical schema. 11
Figure IV.1 A closed chain of mandatory object attributes 19
Figure IV.2 An interrupted chain of object attributes. 19
Figure IV.3 Detecting circular chains of object attributes 19
Figure IV.4 A functional relationship type to be transformed. 20
Figure IV.5 Transforming object attributes 20
Figure IV.6 Transforming object attributes 21
Figure V.1 Influence of object attributes on constructors 27
Figure V.2 Object attribute with cardinality [1-N] 30
Figure VII.1 Primary and secondary identifiers 38
Figure VIII.1 Conceptual schema of a team 47
Figure VIII.2 Logical schema of the Team. 48
Figure VIII.3 Physical schema of the team. 49

I. Introduction

This thesis has to be situated in the context of forward database design, a discipline that includes, among others, the development of conceptual, logical and physical models, which are methodological steps on the way towards the actual implementation of a database. Good database design requires a stable methodology, which will be partly dependent on the database management system used and on the language it supports. Whereas the conceptual model is implementation independent, the logical and physical models depend on the database management system used. As a consequence, the database development methodology has to be partly adapted to the used database management system.

Although C++ is a powerful device to develop object oriented applications, one does not have to be all too familiar with the language to be able to find out that, as far as database design is concerned, C++ makes a very poor environment. Whereas SQL offers devices such as identifiers, access keys, foreign keys, obligatory fields, to name just a few of its treasures, C++ offers non of these. But maybe, C++ offers other structures, unknown to more frequently used database management systems, that can come in handy to develop a database... After all, there might be a way to design databases in C++ and to develop a corresponding methodology.

This thesis is a contribution to a C++ database development methodology. It does not only try to develop a one to design databases in C++, which is itself derived from a general database developing methodology, but also investigates by which tools this development process can be supported. After having explored the existing support tools and having demonstrated how these existing tools can be expanded, for instance by means of validation scripts, I developed a tool to generate the C++ code. This code corresponds to a formerly proposed physical model.

The structure of this thesis is as follows: The first chapter is a context reminder, a general approach to database design methodology, in which special attention is paid to object oriented database design and in which at the same time a first effort is made to define a C++ object oriented logical model. The second chapter investigates which structures, contained by the model defined in the former chapter, can be directly translated into C++ and, during this process, the structures that do not exist in C++ become evident as well. On the basis of this knowledge, the design methodology is fully developed in the next two chapters. Chapter three is dedicated to the translation of the conceptual structures into logical ones and in the next chapter, code, or rather pseudo-code to facilitate the lecture of the chapter, is developed for the structures that cannot be immediately translated into C++.

The next step consists of investigating by means of which tools the newly developed methodology can be supported. This is taken care of in the next two chapters. Chapter five investigates on the one hand in which way the DB-MAIN CASE tool can provide existing help and, on the other hand it demonstrates how the user can use the tool to develop even more specific support according to his/her own specific needs. In chapter six, a tool is developed to generate the C++ code corresponding to a physical model. This was by far the most time consuming part of the thesis. Although the programme that generates the code cannot yet completely handle the object oriented models elaborated in the preceding chapters, it can generate code for most of its structures. Still, a few extra constraints need to be imposed on the formerly defined models and this is taken care of at the beginning of this chapter. To make things more easy, an extra tool was designed to validate the simplified models.

The last chapter is a case study, based on the methodology developed in the former chapter. It also investigates the code generated by the generator. The source code of the C++ generator can be found in annexe 1, and in the following annexes some physical schemata are exposed, accompanied by the corresponding C++ code generated by the C++ generator.

II. Context reminder

This first chapter comprises four parts. In the first part, the reader is reminded of the principles of a good database development methodology and the quality criteria that should be aimed for[1]. The second and the third parts treat the conceptual and a logical schemata, and the last part offers a general definition for CASE tools.

II.1. Database Development Methodology

Good database development consists of the following phases:

- studies of the users' requirements

  During this first phase the client explains what he expects, wants and needs. In other words, he expresses the necessities and requirements of the database he wants to see developed, and which will be written down in text form. In this way, a semi-formal textual description is obtained.
  Important criteria of quality are completeness, correctness, readability. Moreover, the text should not contain any redundancies, nor contradictions.

- conceptual design

  The textual description taken care of in the first phase consists of factual sentences describing the application domain (i.e. the problem to solve or the system to describe) in terms of its concepts, properties and organising rules. In other words, such a text can be interpreted as a linguistic expression of the future conceptual schema. Therefore, in this phase, the text will be decomposed, and somewhat reworked, in order to obtain a list of elementary propositions that are easy to interpret and to translate into entity-relationship constructs. These elementary propositions are then interpreted and translated into entity-relationship constructs. The users' requirements of the future database are now formalised into a conceptual schema, of which the most important property should be *correctness*. This criterion will be fulfilled if the conceptual schema expresses all the meaning included in the starting text in an elegant way. A more detailed presentation of the conceptual model can be found in II.3.

  The textual description may include some flaws, such as redundancies and conflicting information, and can lack some important information as well. Consequently, the conceptual analyst may be obliged to ask people from the application domain for additional information. Of course, many other sources of information can be used to contribute to the conceptual analysis, such as administrative and legal documents, observation of working procedures, forms and other documents, screen layouts and printed reports, existing files and databases, existing programs, etc. Some sources require more advanced techniques and methods (reverse engineering for instance) but they are beyond the scope of this thesis.

[1] Appropriate techniques to realise the mentioned quality criteria can be found in [HAINAUT, 96a] and [HAINAUT, 96b].

Important quality standards for the development of a conceptual schema are normalisation, clarity, minimality[2] and the respect of modelling standards.

- logical design

The conceptual schema, designed in the former phase, is an *abstract* piece of art and is not *operational* or, in other words, it is independent of the Data Management System[3] (DMS) that will finally be used to implement the database. Logical design consists of designing an operational expression[4] of all the specifications included in the conceptual schema. A logical schema is semantically equivalent to the conceptual schema it is derived from but dependent on the model of a family of DMS. If, for example, the DMS is a relational one, a relational logical schema will be designed. A more detailed description of a logical schema will be presented in II.3.

Two important quality criteria to be taken into consideration are time and space efficiency. It should be noted that these criteria are often not complementary: often the price to pay for time efficiency is a loss of space efficiency and vice versa.

- physical design

Whereas the logical schema depends on the model of a family of D(B)MS, a physical schema is compliant with a specific D(B)MS of this family. For instance, starting out with a relational logical schema, which is SQL compliant, different physical schemata can be designed, such as ORACLE 7, DB2 or SYBASE. Example:

[2] I.e. no redundancies.

[3] Data Management Systems (DMS) can either be Database Management Systems (DBMS) or plain programming languages. There are different kinds of Database Management Systems. ORACLE, SYBASE, DB2 and SQL-Server, for instance are relational DBMS which understand some kind of dialects of the SQL language. IMS, IDMS, DATACOM/DB, TOTALK and IMAGE are examples of non relational DBMS. Also plain programming languages offer data structure management systems, generally called File Management Systems. To avoid having to distinguish these categories, one can talk about Data Management Systems, or DMS.

[4] I.e., an expression that can be processed or operated by software.

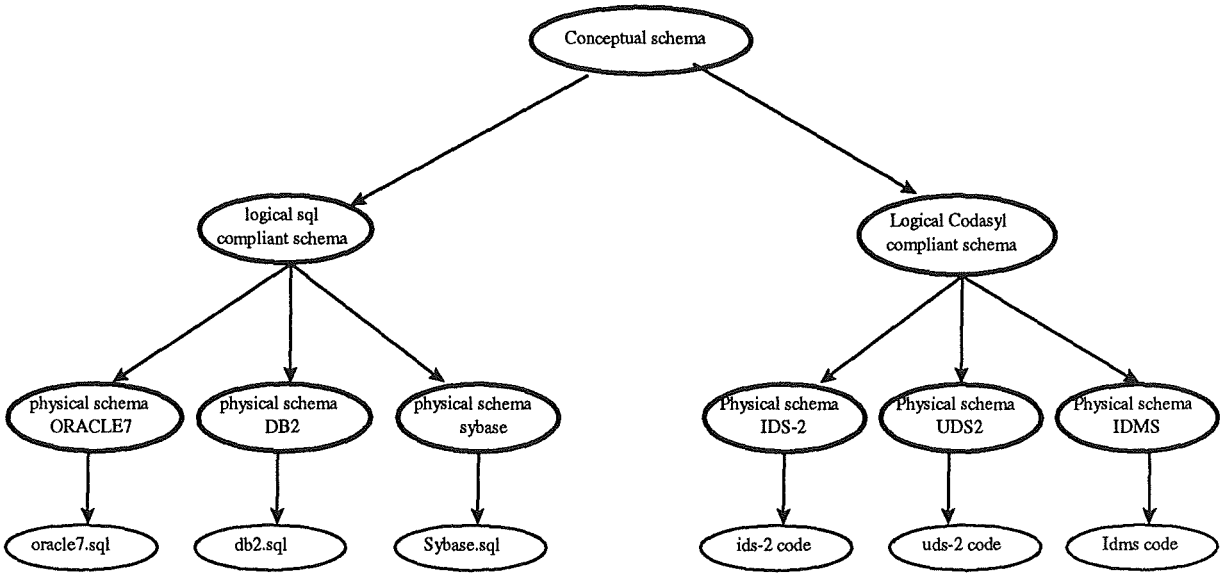


Figure II.1 From conceptual to physical schemata.

Physical design comprises two phases. The first one consists in developing an abstract physical schema by augmenting the logical schema with technical specifications. Access keys (indices) are added to the schema, the designer must specify which files (collections) are available and in which file(s) the rows of each table will be stored, the in the D(B)MS invalid names, reserved words for instance, are replaced by valid ones, technical descriptions are added where necessary and redundant access keys are taken away. Through the second phase, the physical schema is translated into the Data Definition Language of the D(B)MS.

Important quality criteria to take into account are time and space efficiency. The available tuning features of the D(B)MS should be fully exploited, and the designer should not lose sight of the strengths and weaknesses of the host programming language. Finally, when the physical schema is translated into the Data Definition Language, good programming standards should be aimed for.

- implementation
  During this phase, the database is put on the computer and made to work.

- exploitation procedures
  Once the database implemented, certain functions and procedures are necessary to exploit it. For instance, one has to write functions and procedures that are capable of modifying data, update data, etc.

- utilisation
  The database is being used...

- maintenance and evolution
  ... and has to be maintained. Databases that have been developed according to a rigid database development methodology and whose documentation such as texts, conceptual, logical, physical schemata, etc. have well been conserved, will be a lot easier and less time consuming to be maintained. The maintenance of databases of which all or most of the documentation has been lost, can be a very slow and cumbersome procedure. Nevertheless, thanks to reverse engineering there is a way of redocumenting, reeningeering, maintaining, etc. even those databases.

II.2. The Conceptual Model

II.2.1. A General Approach of the Conceptual Model

The conceptual schema discussed in this work is the Entity Relationship conceptual schema. It comprises the following concepts:

Entity types
An entity type represents a class of concrete or abstract real-world entities, such as computers, books, dreams and orders. It can also be used to model more computer oriented structures such as record types, tables, segments, classes, etc. Instances of entity types are called entities. The graphical representation of an entity type is a rectangle, as illustrated in Figure II.1.

An entity type can be a subtype of one or more other entity types, called its supertypes. A subtype inherits all the properties of its supertype. The collection of subtypes of an entity type E is declared **total** (symbol **T**) if each E belongs to at least one subtype. This collection is declared **disjoint** (symbol **D**) if each E belongs to at most one of its subtypes. If the collection is both total and disjoint, it forms a **partition** (symbol **P**) of entity type E. In Figure II.2, entity types MAN and WOMAN form a partition of entity type PERSON.

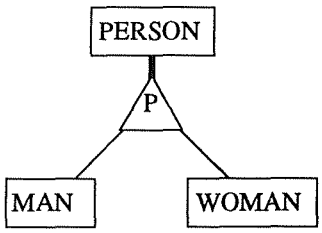


Figure II.2 Entity types: subtypes and supertypes

Relationship types
A relationship type is defined by a correspondence between two or more entities. Each entity type that enters into a relationship with a relationship type plays a specific role. If a relationship type creates a correspondence between two entity types, it is called **binary**, if it does so between three or more of them, it is called **N-ary**. A relationship type with at least 2 roles taken by the same entity type is called recursive or **cyclic**.

Each role Ro of an relationship type Re is characterised by its name, which is not obligatory, and its cardinality [i-j], a constraint stating that each entity assuming role Ro must enter at least i and at most j times into a relationship with Re. Generally, i is 0 or 1 and j 1 or N but as long as $i \leq j$, $i \geq 0$ and $j>0$, they can take any value. A role can be taken by more than one entity type. If this is the case, it is called a **multi-ET** role. The graphical representation of a relationship type is a hexagon, as illustrated in Figure II.3, in which a CLIENT orders from 0 up to N instances of PRODUCT and a PRODUCT can be ordered by 0 to N instances of client.

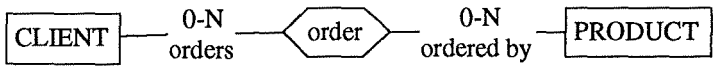


Figure II.3 Relationship types

Attributes

An attribute is a quality or characteristic of an entity type or a relationship type. It can take one or more values and groups of values. A value is a symbol used to represent an elementary fact, often in the form of a character string. A few examples are 'John', 'Deeds', 'County Road, '7', etc. Possible domain values are: boolean, numeric, char, etc. Graphically, an attribute is placed in the lower part of the rectangle and hexagon, representing an entity type and a relationship type respectively.

As illustrated in Figure II.4, an attribute has a cardinality [i-j]. The default cardinality is [1-1] and can be omitted. Attributes can be single valued or multivalued, optional or mandatory and atomic or compound. An attribute is optional if i=0 , it is single valued if j=1 and as soon as j>1, it is multivalued. If an attribute comprises other component attributes itself, it is compound, otherwise atomic. When the domain of an attribute is an entity type, it is called an object attribute, which will be further discussed in II.3. In entity type CLIENT, represented below, the attribute *name* is single valued and mandatory, *first name* is optional and multivalued, *address* is mandatory, compound and single valued and *comp* is an optional object attribute.

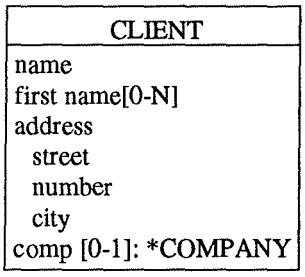


Figure II.4: Entity type CLIENT.

Integrity constraints

An integrity constraint is a property, which cannot be expressed by means of the basic concepts of the schema, which the data corresponding to this schema should comply to. There are many integrity constraints such as cardinality constraints, which have already been discussed, inclusion constraints, exclusion constraints, identifiers, etc. Except for the identifier and the inverse constraints, no special attention will be paid to them, because they are not taken into account in the other chapters of this thesis.

An **identifier** makes it possible to identify unambiguously one and only one instance of an entity type or a relationship type. The value or combination of values taken by an identifier must be unique. The main identifier of a parent object is called the *primary identifier*, all its other identifiers are called *secondary identifiers*. A parent object can only have one primary identifier. The identifier of an entity type can consist of:

- one or several of its attributes;
- at least one role assumed by this entity type;
- a group formed by one or more of its attributes and one or more of the roles it assumes.

In Figure II.5 below, entity type CLIENT is identified by attribute *number*, which means that two entities can not have the same values for this attribute.

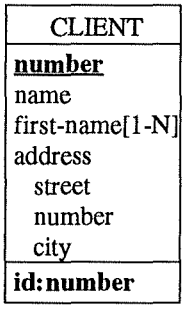


Figure II.5: Identifiers.

An **inverse constraint** can be asserted between two object attributes, expressing that each is the inverse of the other [DB-MAIN, 97]. For example, in Figure II.6, *orders* of CUSTOMER and *owner* of ORDER are declared inverse object attributes. If c denotes the Owner of ORDER entity o, then c must belong to the *orders* value set of CUSTOMER c.

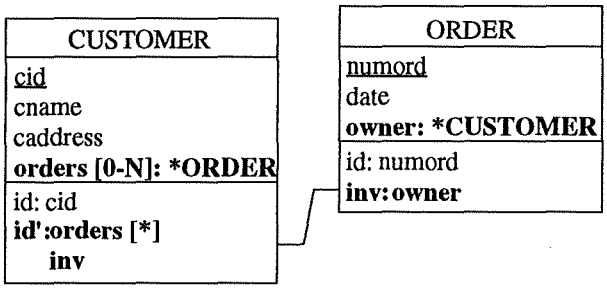


Figure II.6 The inverse constraint.

II.2.2. Note on the Conceptual Model with regard to C++

In C++, the collection of subtypes of an entity type is always disjoint: any instance of a superclass belongs to at most one instance of the classes that inherit from this class. This can be considered as an exception to the rule that a conceptual model is independent of the database management system.

II.3. The Logical Model

This chapter consists of two parts. In the first part, the General Abstract Model (GAM) is presented. The GAM contains all the general characteristics of a logical model. The second part is an application of this GAM: the C++ logical object oriented schema.

II.3.1. The General Abstract Model

By means of the GAM, the general characteristics of a logical model are presented. Firstly, in a logical model, multivalued attributes are extended, i.e. the way in which their values are structured is determined. The possibilities are the following:

- set: unstructured collection of distinct elements;
- bag: unstructured collection of not necessarily distinct elements;
- list: sequenced collection of not necessarily distinct elements;
- unique list: sequenced collection of distinct elements;
- array: sequenced collection of cells that each contain an element;
- unique array: sequenced collection of cells that each contain a distinct element.

Secondly, access mechanisms (or access keys) are added. In a relational, i.e. SQL based model, foreign keys are added to enforce that the values of one or more columns of one table correspond to the values of the identifier of another. It is also determined where the data are going to be stored. Strongly related data could for instance be grouped and stored in the same file. In this way, data that often have to be accessed at the same time are found in the same file. Finally, the different entities are stored in a certain way, sometimes ordered, sometimes not. In a relational based model, for example, there is no order between the different rows of a table.

II.3.2. The C++ Logical Object Oriented Schema

The C++ logical object oriented schema is just one possible application of the GAM. Other possible applications are the logical SQL schema, the logical CODASYL schema, the logical COBOL schema, etc.

In a C++ logical object oriented schema,

the following structures and constraints can be expressed:

- entity types without attributes
- atomic and compound attributes
- object attributes
- single valued and multivalued attributes
- optional and mandatory attributes
- identifiers consisting of one or several attributes
- multivalued identifiers
- inverse constraints

- (foreign keys)

and the following structures cannot be expressed:

- collections of non disjoint subtypes
- roles
- relationship types without attributes
- relationship types with attributes

Remarks:

1. In object oriented database design, foreign keys should be avoided. The reason is that physically, in C++, foreign keys become either copies of or pointers to values of one or several data members[5] forming an identifier of an object from another class. Object attributes are translated into pointers to other objects and from there on, any data member of those objects can be accessed. Thus, only one pointer has to be stored to be able to access any data member of the object pointed to. Therefore, object attributes are a lot more efficient and economical. Moreover, object oriented databases without foreign keys are more flexible since each foreign key must correspond to an identifier whereas there is not the case for pointers pointing to objects. Consequently, relationship types should rather be transformed into object attributes than into foreign keys.
2. A distinction was made between relationship types without attributes and relationship types with attributes because both objects will be subjected to a different transformation to obtain a logical schema from a conceptual schema.

The way in which the structures that cannot be expressed in the logical model, are transformed will be discussed in chapter IV, *Logical Design of an Object Oriented Schema*, in which I'll pay special attention to object attributes and to some conditions related to them. Chapter VI, *The DB-MAIN CASE Tool*, discusses how the DB-MAIN CASE tool can be a trustworthy assistant to perform transformations.

[5] In C++, attributes become *data members*.

Example of an (C++) object oriented logical model [DB-MAIN, 97]:

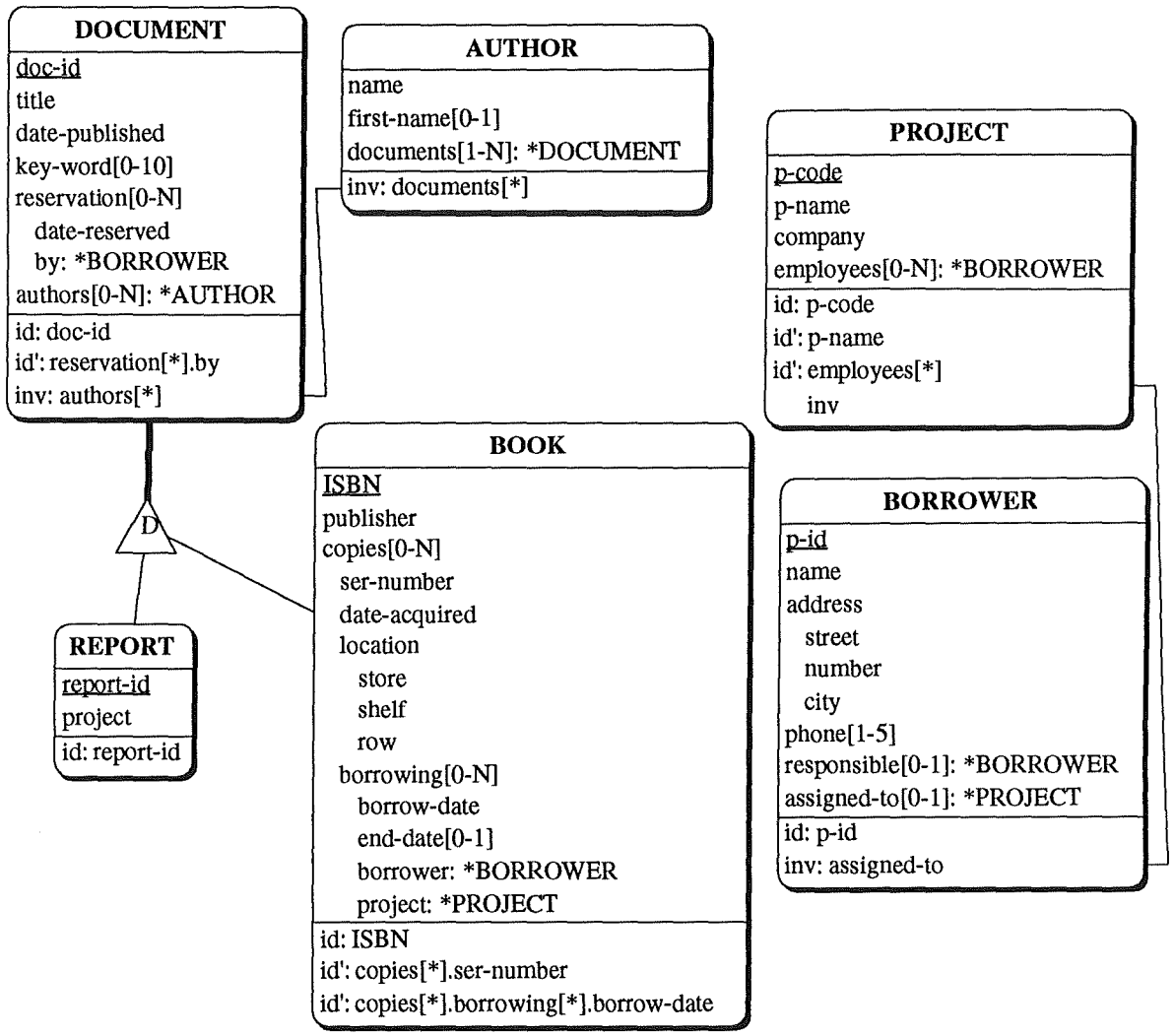


Figure II.7 An object oriented logical schema.

II.4. Computer Aided Software Engineering

Computer aided Software Engineering (CASE) tools provide automated support for many of the system analysis and design methods available to the information systems developer. They provide an environment to draw and redraw diagrams, to generate system documentation, to generate code structures and database schemata, etc.

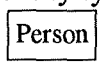
Besides the DB-MAIN CASE tool, an example that is worthwhile being mentioned is Rational Rose. This tool can be used to apply the Unified Modelling Language (UML) [Muller, 97], and goes with a C++ generator.

III. Declarative Structures in C++ Classes

Compared to SQL for instance, C++ is a very poor language to implement a database. It offers very few declarative structures or, in other words, very few structures found in physical models that can be directly and implicitly expressed. This chapter gives an overview of the declarative structures that do exist in C++. For each structure, the corresponding code will be represented together with a few indispensable comments concerning language specific characteristics.

III.1. Entity Types

Entity types are translated into classes.
eg: entity type 'Person'



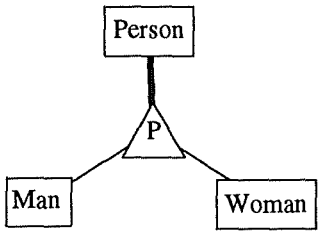
becomes the following C++ class:

```
class Person {
  ...
};
```

Subtypes and **Supertypes** are translated into subclasses and superclasses. As I have already mentioned, the collection of subtypes is always disjoint in C++. As a consequence, if a collection of subtypes is total, it automatically forms a partition of its supertype. However, it is possible to make a distinction between total and non total collections of subtypes. One can create a total collection of subtypes by declaring a virtual data member of the superclass to be a *pure virtual function.* A virtual function is 'made pure' by the initializer = 0 [Stroustrup, 97]. A class with one ore more pure virtual functions is an *abstract class*, and no objects of that class can be created.

eg:

The following schema, in which entity types MAN and WOMAN form a partition of entity type PERSON,



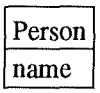
is translated into C++ by three classes. In class Person, member function *f()* is declared to be a pure virtual function, so that no more objects of class Person can be created. In classes Man and Woman, the *f()* member functions are overridden, so that they are no longer virtual and by means of ' *: public Person*', it is expressed that classes Man and Woman are subclasses that inherit publicly from class Person.

```
class Person  {
...
virtual void f ( ) = 0 ;
..
};

class Man : public Person {
...
virtual void f ( );
...
};

class Woman : public Person {
...
virtual void f ( );
...
};
```

III.2. Attributes and Attribute Types

Simple attributes are translated into data members.

eg: attribute *name* of type *char* and with length *30* of entity type *person*



becomes the following data member

```
class Person {
  ...
  char name[31];
  ...
};
```

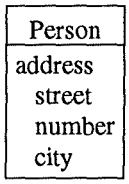
C++ provides member functions, equivalent to methods in Delphi, to access and then set, get or modify the values of data members. They are not represented in this example. Since strings are null terminated, in C++, the length of the data member is obtained by adding 1 to the length of the corresponding attribute. Data members are always optional and to make them mandatory, the programmer has to add additional code. This holds for multivalued attributes as well.

Attributes can be of different **types**. The following types can be immediately translated in C++ types:

TYPE	C++
Boolean	bool
Char	char
Compound(*)	struct
Float	float
Numeric, Integer	int (for small numbers, <32768.) double (for larger numbers) float (if there are decimal digits)

(*) Compound attributes are translated into structures, which are equivalent to records in Delphi and many other programming languages.

Eg: The following compound attribute *address* of entity type *Person*



becomes in C++

```
class Person {
   ...
   struct Taddress {
    char street;
    int number;
    char city;
   } address;
   ...
};
```

Similar to 'street', which is a data member of type 'char', address is a structure of type 'Taddress'.

Collection types: Although one can make a distinction between sets, bags, lists, unique lists, arrays and unique arrays, as explained on page 9, only arrays can be directly implemented in C++. There are three different situations to be discussed:

1. Arrays of simple types (integers, characters, floats,...)
   To create an array of 0 to 5 integers, for example, one writes in C++:
   ```
   int myint[5];
   ```

2. Arrays of compound types. Example of an array of 0 to 5 five addresses of type t_address:

   ```
   struct t_address {
    char street;
    int number;
    char city;
   } Address[5];
   ```

3. Arrays of strings cannot be immediately translated into C++. The only way to implement them is by means of arrays of pointers.
   Example:
   ```
   char * s[5];
   s[1] = new char[25];
   ```
   s is an array of five pointers to characters. By means of 'new', enough space is reserved for the pointer to point to the first character of a string of 25 characters.

For other non declarative structures and constraints such as identifiers, object collections, object attributes, object relationships and cardinality constraints, etc., the programmer has to write additional code. In chapter V, *Physical Design of a C++ Object Oriented Schema*, you find the algorithms and pseudo-code, which the programmer can develop to implement non declarative structures and constraints.

IV. Logical Design of an Object Oriented Schema

This chapter discusses the transformations that have to be performed to transform a conceptual entity relationship schema into a C++ logical object oriented schema. Ideally, the transformations used during the logical design phase should all be reversible and semantically equivalent but unfortunately, this is not the case.

As has been concluded on page 9, conceptual schemata contain three structures that cannot be expressed in the C++ object oriented logical schemata: collections of non disjoint subtypes, (multi-ET) roles and relationship types.

IV.1. Analysis:

1. Collections of non disjoint subtypes. It is not possible for an entity type E to belong to more than one of its subtypes. Unfortunately, a semantically equivalent transformation does not exist here. If, in the conceptual model, the collection of subtypes is total and non disjoint (**T**), it has to be transformed into a collection that forms partition of its supertype (**P**) and if it is non total and non disjoint (-), it will have to be transformed in a non total and disjoint one (**D**).
   Another possibility consists of suppressing the superclass and adding its attributes to all its subtypes. An entity that was to belong to several of its subtypes will now exist several times in the database. This causes redundancy.

2. Multi-ET roles, i.e. roles that are taken by more than one entity type, prevent relationship types from being transformed in into object attributes, a transformation which is described in the next step. Therefore, they must be transformed into relationship types.

3. Relationship types.
   - Transformation of non functional relationship types into entity types.
     Non functional relationship types are N-ary relationship types, relationship types with attributes or binary, many to many relationship types.
   - Transformation of functional relationship types into object attributes.
     Functional relationship types are binary, one to many relationship types without attributes.

IV.2. A closer Look at the Transformations

In the next paragraphs, I will give a detailed analysis of each transformation. Later, in chapter VI, I will investigate to which extent the DB-MAIN CASE tool can assist the user through this transformation process.

Transforming non disjoint collections of entity types:
Passing from disjoint collections of entity types to disjoint (**D**) ones, is not really a transformation properly speaking. In fact, by doing this one merely changes the semantics of the schema.

Transforming multi-ET roles into relationship types:
When a multi-ET role is transformed into a relationship type, the relationship type holding this role is split up into two equivalent ones with the same cardinalities, namely the cardinality of the multi-ET role and the cardinality of the original role at the other side.

Transforming non functional relationship types:
When a non functional relationship type R is transformed into an entity type E, each one of the original roles is transformed into a *one to many* or sometimes *one to one* relationship type. The basic propagation rules of this transformation are:

- Cardinality propagation:
  For each role with cardinality [I-J] in R, there is a new relationship type with cardinalities [I-J], [1-1]. The [1-1] cardinality is placed at the side of the newly created entity type E.
- Attribute propagation:
  The attributes of R are associated to the entity type E obtained by the transformation.
- Identifier propagation:
  The identifiers of R are translated for entity type E as follows: each role is replaced by the corresponding role of the new relationship type, and each attribute is kept unchanged.

Remark:
As has been said before, the above discussed transformation is reversible. A relationship type R that has been transformed into an entity type E can be retransformed into a relationship type and if both these transformations are performed, the original situation will be recovered. The conditions on the entity type E to be transformed into a relationship type are:

- E takes at least two roles;
- the cardinalities of these roles must be [1-1] for the obvious raison that each E entity must be linked to one and only one entity of the other side, in the same way the (future) corresponding relationship will be made of one and only one entity of each kind;
- all these roles belong to distinct relationship types, because otherwise one of the relationship types would be cyclic, and it would not be possible to replace it by a role.
- E has at least one implicit or an explicit identifier. The raison is that all relationships of the same kind are distinct, and cannot be made of the same entities and attribute values.

Transforming functional relationship types:
The transformation of functional relationship types, i.e. binary, one to many relationship types without attributes, to object attributes is undoubtedly the most critical phase of this logical design process. There are two problems concerning this transformation process. In the first place, this complexity directly results from the decision I made to create/ generate a non transaction-oriented database. The difference between this and a transaction-oriented database and why the use of object attributes is limited as a consequence of the choice not to work with a transaction-oriented C++ database, are explained in V.3, for the physical representation of object attributes cannot be left out of this discussion. In the second place, this transformation process is complex because it imposes certain choices to be made.

1. Restrictions on object attributes:
In the C++ object oriented logical schema I propose, it is not possible to have a circular chain of **mandatory** object attributes, i.e. a closed chain of mandatory object attributes in such a way that the type of the one corresponds to the next class in the chain.

Example:

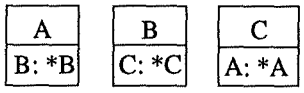


Figure IV.1 A closed chain of mandatory object attributes.

In this logical schema, class A contains an object attribute to class B, class B one to class C and to close the chain, class C one to class A.

As will be discussed in more detail in V.3., the raison of this restriction is that, in non transaction-oriented databases, an A object cannot be created before its corresponding B object, which cannot be created before its corresponding C object, which itself cannot be created before its corresponding A object, etc.

As soon as this chain of mandatory object attributes is interrupted by making at least one of them **optional**, also a non transaction-oriented database can deal with the situation. In the example below, one can first create an instance of B, then one of A, of which the object attribute B points to the newly created class B, and then one of C, of which the object attribute A corresponds to the newly created class C.

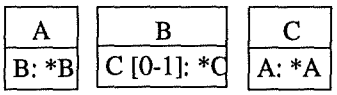


Figure IV.2 An interrupted chain of object attributes.

However, it is far more easy to detect these circular chains before the relationship types have been transformed, as demonstrated in the example below:

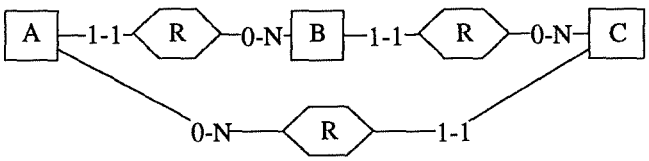


Figure IV.3 Detecting circular chains of object attributes

2. How to transform and which choices are to be made:
Before a relationship type can be transformed into an object attribute, two preconditions have to be fulfilled:
- the relationship type is binary and without attributes;
- both roles are mono-ET[1].

The designer has to decide which entity type will contain the object attribute. It can be placed in one entity type or in each entity type (with an inverse constraint between the two object-attributes).

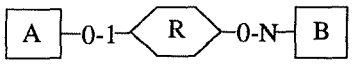


Figure IV.4 A functional relationship type to be transformed.

In the schema represented above, the object attribute obtained by the transformation can be placed in class A, in class B or in both classes with an inverse constraint between the two object-attributes.

Graphical representation of the three situations:

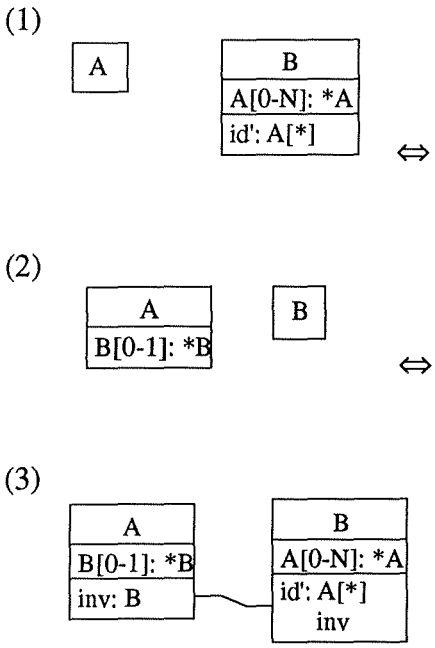


Figure IV.5 Transforming object attributes

[1] I.e. roles that are taken by only one entity type.

In (1), for each instance of B, the 0-N associated A classes can be found directly: they correspond to the values of object attribute A. For each instance of A, to find back the associated instance of B, the database must progressively search all the object attributes of type A of every B instance.
In (2), for each instance of A, the associated B class can be found directly since it corresponds to the value of object attribute B. However, to find the A classes corresponding to an instance of B, it is necessary to search object attribute B of every A instance.
In(3), the database can immediately find the corresponding class(es) for both classes. Once more, the inverse constraint between the two object attributes expresses that each one is the inverse of the other.

Although the three possibilities are semantically equivalent, they have a repercussion on database optimisation. If access time is more important than keeping the size of the database minimal, then the object attribute should be placed in both entity types. If size is more important than access time, the object attribute should be placed in the entity type from which the majority of the accesses towards the corresponding entity types are made.

However, in one situation, there is only a choice between two options: if the minimum cardinality of one role is 0 and that of the other >0. In this situation, the object attribute corresponding to the relationship type to be transformed cannot be placed in the entity type that plays the role with the cardinality >0. The following example makes clear why:

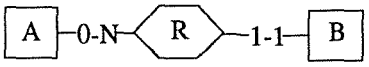


Figure IV.6 Transforming object attributes.

If, in Figure IV.6, R were transformed into an object attribute ObjA contained by A, ObjA could not be obligatory since because the role 0-N it is possible that some instances of a are related to 0 instances of B. However the role 1-1 should be translated into an obligatory attribute.

IV.3. Transformation Plan

In conclusion of this chapter, I present the following transformation plan to obtain a C++ object oriented logical schema:

1. Collections of non disjoint subtypes
   - To be transformed into collections of disjoint subtypes. This is a non semantically equivalent transformation.
2. Multi-ET roles
   - To be transformed into relationship types.
3. Non functional relationship types
   - To be transformed into entity types.
4. Functional relationship types
   - Firstly, the minimum cardinality of object attributes has to be adapted in such a way that there are no circular chains of mandatory object attributes, then
   - functional relationship types are to be transformed into object attributes.

V. Physical Design of a C++ Object Oriented Schema

The first part of this chapter is dedicated to the adaptations of the logical schema that are necessary to obtain a physical schema. The rest of the chapter analyses how non C++ declarative structures are translated into C++ code.

V.1. The Physical Schema

To obtain a physical schema, the last elements that need to be taken care of are names and technical descriptions. The names of classes, attributes, etc. are directly derived from conceptual names. They do not necessarily satisfy the naming conventions of C++, which are as follows: names must always start with a letter and may contain letters, digits and underscores (_). Neither can names be words reserved for the target language. Although the length of names is not restricted, some C++ compilers will only distinguish two names if they there is a difference among the first 32 characters. For some elements in the schema, it can be a good idea to add a technical description, i.e. technical comments and recommendations that can be useful to the programmer or the database manager.

V.2. Translation of the Physical Schema

Many important constructs and integrity constraints that are necessary to express essential properties of the application domain to be described, can not directly be expressed in C++. To implement them, the programmer has to develop code him/herself. In this chapter, five such constructs and integrity constraints are discussed: object collections, identifiers, object attributes, cardinality constraints, and some attribute types. For each of them, I'll propose algorithms and pseudo code, which are one way to implement them in C++ and which correspond to the code generated by the C++ generator discussed in chapter VII. Together with the declarative structures discussed in chapter III, these five constructs are the ingredients of a physical schema.

V.2.1. Object Collections

To express the properties of the above-mentioned C++ logical schema, two kinds of object collection are necessary: collections of objects from one class, representing an entity type, and collections of values of a multivalued attribute.

Collections of objects (class instances):
Collections of objects that collect all the instances of a class can be simulated by lists. The main advantage is that the number of objects that can be added to a list is undetermined. Each newly created object of class C is added to the list of objects from its class. The performance of the database can be augmented if this list takes the form of a 'double chain' of pointers to make it possible to navigate back AND forth. Unique lists can be obtained by checking whether the identifier constraint (cf. infra) is respected before adding an object to the list.

Each object of class C contains a pointer 'first' to the first object of the list, a pointer 'previous' to the object preceding it in the list, a pointer 'next' to the object placed right behind it in the list and the member functions to get, set and modify the values of these three data members[1].

```
class Person {0
 ...
 C * next;
 C * previous;
 C * first;
 corresponding member functions
 ...
};
```

Whenever an object O of class C is created, its constructor will execute the following lines:

```
1. next ← NULL;                                          (1)
2. previous ← former_object_in_list;                     (2)
3. IF previous-exits THEN previous_object.next ← this_object
                  ELSE first ← this object
4. former_object ← this_object
```

Comments: O is added as the last object and therefore the value of 'next' becomes NULL (1). The global variable 'former_object_in_list' is assigned to 'previous' (2). If there is a previous object in the list, its data member 'next' will be assigned a pointer to object O.

When, later on, O is destroyed, its destructor will execute the following algorithm:

```
IF next_exits THEN
    if previous_exists then                              (1)
      previous_object.next ← next;
      next_object.next.previous ← previous;
    else                                                 (2)
      next_object.previous ← NULL;
      first ← next;
 ELSE
    if previous_exists then                              (3)
       previous_object.next ← NULL;
       former_object ← previous;
    else                                                 (4)
       former_object ← NULL;
       first ← NULL;
```

Comments: Before being able to restore the list without O, the algorithm has to detect O's position in the list. O can be situated in the middle (1), at the end (2), in the beginning (3) of the list or O can be the only object in the list (4).

[1] To simplify the pseudo code the member functions are left away. In real C++ code, *previous_object.next ← next*, for instance, will become *get_previous()->next = next;*

Collections of values of multivalued attributes:

1. Lists and unique lists of attributes can be implemented in the same way as lists and unique lists of objects. In chapter VIII, another way of implementing them will be proposed.
2. Arrays have been discussed in chapter III. Unique arrays can be obtained by checking code equivalent to the one discussed in V.2.2.
3. The algorithms to simulate bags and sets will not be discussed.

V.2.2. Identifiers

An identifier is an attribute, or a set of attributes, that designates a property which is unique for each instance of a class. Since in C++ this uniqueness cannot be declared, it must be procedurally enforced. Before a new instance can be added to the collection of already existing objects of a certain class, a procedure has to compare the identifier of this instance with the identifiers of all the other instances to check its uniqueness. Only in case of uniqueness, the new instance can actually be created. Ideally, this 'checking procedure' is called by the object's constructor, who'll refuse to construct the object if the identifier constraint is not respected. The following algorithm demonstrates what such a 'checking procedure' would look like:

A new object 'NO', of which attr1 and attr2 form the primary identifier and attr3 is a secondary identifier, can only be created if the following function returns 'true':

```
Function boolean Check_identifier (attr1: Tattr1, attr2: Tattr2, attr3:
                                   Tattr3)
for each object in Get_list_of_objects() do                    (1)
  IF
    O.attr1 = NO.attr1 and O.attr2 = NO.attr2                  (2)
    or                                                         (3)
    O.attr3 = NO.attr3
  THEN return false
return true;
```

Remarks:

(1) In fact, Get_list_of_objects( ) returns a list of pointers to objects, but here it is far more easy to represent them as objects.

(2) Between the comparisons of different attributes that form one identifier, the algorithm uses logical operator 'and': as soon as one attribute value is unique, the identifier constraint is respected.

(3) Between the different identifiers (primary and secondary identifiers) logical operator 'or' is used: in order to be able to create an object, *all* its identifier constraints have to be verified.

The constructor of an object start as follows:

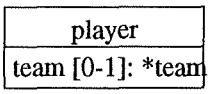
```
if Check_identifier(...) then
 destroy_this_object()                    (1)
 throw an exception
else
  (continue construction)
```

Remark: (1) Destroy_this_object comes down to the object's calling its own destructor.

V.2.3. Object Attributes

As a consequence of what was concluded in *II.3.2.,The C++ Logical Object Oriented Schema*, no attention will be paid to foreign keys and this discussion will exclusively be dedicated to object attributes. In this third part of the chapter, I will first present the physical representation of object attributes, then I'll explain why object attributes can be a problem for non transaction-oriented databases and finally I'll give a detailed analysis of the algorithms that must be implemented in C++ to comply with the integrity constraints at all times.

The physical representation of object attributes:
An entity type containing object attributes, will be translated by a class containing data members holding the address of an object of the corresponding type. From now on, data members corresponding to object attributes will be referred to as *member classes*. Example:



becomes in C++:

```
class player {
 ...
 team * ptr_to_team;
 player(..., team * ateam, ...);
 ...
};
```

When a class contains a member class of type C, its constructor will contain an argument which corresponds to this member class. When the constructor is called, the corresponding parameter, will be a pointer or a list of pointers to the object(s) of class C with which object 0 enters into relationship. If the object attribute is mandatory, this pointer is obligatory and cannot be at NULL to indicate that the pointer points to nothing, but if the object attribute is optional, this pointer can be set at NULL, for example, if the object it points to has not (yet) been created.

Object attributes and non transaction-oriented databases
In a transaction-oriented database, it is possible temporarily not to respect the database's integrity constraints, namely from the beginning until the end of a transaction. In this way, it becomes a lot easier to add, change and remove data to and from the database. For example, in order to change the identifying 'membership_id' of a class 'person', the old membership_id has to be taken away before it can be replaced and at that very moment the identity constraint is not respected.

On the contrary, in a non transaction-oriented database, the integrity constraints have to be fulfilled at all times. Therefore, if the physical schema contains *circular chains of* ***mandatory*** *object attributes* (cf. IV.2), in the end, the constructors of all these classes will contain a mandatory (list of) pointer(s) to the object(s) of the class they enter into a relationship with. Since it is not possible to put one of these pointers temporarily at NULL, because one of the objects has not (yet) been created, it is never possible to create even the first object of the chain. Destructors will pose the same problem.

In order to create a transaction-oriented database, software should be added to batch the data until all the objects have been created before these data can actually be added to the database. Of course, the code necessary to implement the database will become a lot more complex.

Guaranteeing permanent validity of the database with respect to object attributes:
In this part of the chapter, I shall present the different cardinalities that can be attributed to object attributes and for each one of them, discuss the functions and procedures the C++ database should be provided with to take care of the integrity constraints.

1. *Supposition: the database contains only optional object attributes with cardinality [0-1].*
For each such attribute, contained by class C, class C should be equipped as follows:
C's constructor: to each member class contained by class C must correspond one formal argument. When an object O of class C is created, the corresponding parameters will be pointers which will be assigned by the constructor to the corresponding member classes. Since the minimum cardinality is 0, the value passed can always be NULL.

Example:

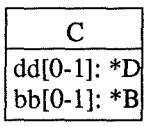


Figure V.1 Influence of object attributes on constructors

corresponds to:

```
//Class
class C {...
  B * bb;
  D * dd;
  C(..., B * mybb, D * mydd, ...);
...};

//constructor
C::C(..., B * mybb, D * mydd, ...) {...
  bb=mybb;
  dd=mydd;
...}
```

C's destructor:
When object O is destroyed, it is necessary that all the pointers to it contained by other objects, are set at NULL. Since it is known which classes contain member classes of type C, one possible solution consists in making C's destructor call a function that accesses to every single object of all the classes with member classes of type C in order to find pointers to the object it is called by, i.e. O, and set them at NULL. This is an easy solution, and only a few lines of code suffice to implement it in C++. By using a *template*, it is even possible to write this function only one time for each class C and use it to delete pointers to objects of **any** class. This can work on condition that all the member classes of one particular type share the same name, regardless of the class they belong to. In the example below, for instance, it is expected that all member classes pointing to objects of type C are called 'cptr'.

```
Example:
template <class T>
function boolean Delete_Pointers_In_foreign_objects
                                        (T * ptr, C * this)      (1)(2)
|while (ptr)
| |if ptr->cptr = = this
| | | ptr->cptr=NULL
| | | ptr= T::get_next();
```

Remarks:
(1) *ptr* will be a pointer to the first object of the list of objects from its class.
(2) Every destructor contains a pointer named *this* to the object it is called by.

Unfortunately, this solution is a very slow one for large databases: to find one pointer, the function would have to pass through the whole list of objects of a certain class. It would be much more efficient if each object stored the pointers to those objects containing a pointer to itself. To make this work, each object O of class C has to be provided with one data member for each class that contains a member class of type C. It would then be up to the constructor of each object to provide pointers to those objects and store them. There are two possible situations. In the first place, when an object O is created, its constructor receives pointers to other objects that need a pointer to O. These pointers can allready be stored by the constructor

of O. Later, when other objects are created that also contain a pointer to O, the constructor of those objects must provide O with a pointer to themselves.

2. Supposition: the database contains object attributes with cardinality [1-1]:
Managing object attributes with cardinality [1-1] is very similar to managing object attributes with cardinality [0-1]. There are two big differences. Firstly, the [1-1] cardinality is translated into C++ by **compulsory** pointers. The **constructor** of an object O will have to do the same work as for object attributes with cardinality [0-1] except for one thing: in the very first place, it has to check whether the objects pointed to by these compulsory pointers do actually (already) exist. If this is not the case, O cannot be created and its constructor must call its destructor. This can only work on condition that the user or the intermediate programmer puts pointers to objects that do not yet exist at NULL. Then the constructor will simply work as follows:

```
class::class(...) {
 if PtrToCompulsoryObject= = NULL then {
  delete this;
  throw ErrorMessage;
 }
...
}
```

The main drawback to this system is that if the user forgets to put compulsory pointers at NULL, the system will not work, because as soon as a pointer is declared, it contains a random value different from NULL.

Just as in the case of cardinality [0-1], O's **destructor** has to find all the objects that contain a compulsory pointer to the object that has called it. However, because of the [1-1] cardinality the member classes are mandatory now and cannot be set at NULL. Consequently, O's destructor has to call the destructors of the objects containing a pointer to the object is was called by, i.e. O. This must be done because objects that contain a compulsory pointer to an object O can no longer exist when O is destroyed or the integrity constraints of the database would be violated. This can be done as in the example below:

Supposition: object 0 is destroyed and objptr1 and objptr2 are two pointers to objects that contain a compulsory pointer to O.

```
class::~class() {
 delete objptr1;                (1)
 delete objptr2;
 ...
}
```

Remark: (1) *delete objptr1* will call the destructor of the object pointed to by objptr1. This destructor could itself call destructors of other objects, etc. As has been mentioned before, this would not work for circular chains of mandatory attributes, because then each destructor would call another destructor in an endless loop. This could be resolved by a system putting flags in objects to make sure the loop is interrupted.

3. Supposition: the database contains object attributes with cardinality [0-N] and [1-N]:
The system would work similarly to what has been presented in the two preceding cases. However, due to the maximum cardinality N, it becomes indispensable to work with lists of pointers. An object attribute with maximum cardinality N, will be translated by a list of pointers to other objects. For the **constructor** of an object, this does not make a lot of difference. It receives lists of pointers and if the minimum cardinality is 1, it has to check whether the length of the lists is superior to zero. For the **destructor**, the system becomes a bit more complex, as is demonstrated on the basis of the situation schematised in Figure V.2.

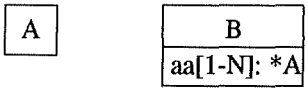


Figure V.2 Object attribute with cardinality [1-N]

If an instance I of class A is destroyed, its destructor must search in the list of instances of class B for pointers to I. Each instance can contain a list of one or more pointers to instances of A, in its data member *aa*. If I's destructor finds that this list contains a pointer to I, it has to remove this pointer from the list. However, if this pointer is the only one contained by the list, I's destructor has to destroy the object containing this pointer because, due to the [1-N] cardinality, it cannot exist without possessing a pointer to at least one instance of A.

V.2.4. Cardinality Constraints

In C++, as soon as a variable of any type is declared, the memory reserved to place the values of these variables will always contain some random information of the corresponding type. A boolean variable, for instance, will always be either at *true* or at *false*. As a consequence, even though an attribute is mandatory, i.e. when its minimum cardinality is 1, the corresponding data member will still be optional because it is impossible for a constructor to determine whether its arguments have been assigned a value to or whether they merely contain random values.

However, there is a way to make a distinction between mandatory and optional attributes anyway: by means of pointers, which can be set at NULL, if they point to nothing. To make this work, it becomes necessary to convert every data member of an object into a pointer to this data member. For example, instead of a data member of type int, a data member is needed, that points to a value of type int. If the C++ database is implemented in this way, the user, programmer or program that provides the data to be stored in the database have to respect

only one convention: if there is no value for a variable, it must be set at NULL. A pointer named *my_int_ptr* pointing to an integer is declared as follows:

```
int * my_int_ptr;
```

V.2.5. Types

In the DB-MAIN CASE tool, which will be further discussed in chapter VI, there are, three types that cannot be immediately translated into C++, namely Date, VarChar and Object Type.

Date:
This type can be converted either in char[11], i.e. an array of 11 characters, or in double. The advantage of saving a date as an array of eleven characters is that is can be saved as 'dd/mm/jjjj' or 'dd-mm-jjjj', i.e. ten digits, two separation characters such as /,-, etc. and one extra character because C++ strings are null-terminated. The disadvantage is that it is impossible to perform mathematical operations on strings, which could for instance be used to calculate the number of days before an order is expired. Therefore, it is better to translate this type into 'double' in order to store a date as a 32-bit value.

Varchar:
Although C++, does not directly provide a type such as 'varchar', it is not impossible to implement this type as demonstrated in the following example:

```
char * s;
s = new char[n];
```

s is a pointer to a array of characters. When memory is allocated to s by means of 'new', n specifies the number of characters the array will be able to contain. Consequently, n can be determined at run time.

Unfortunately, this only possible way of implementing Db-Main's Varchar is rather cumbersome. To store a string, *John* for instance, an algorithm would have to be written to do the following operations:

```
n = strlen("John") + 1;                    (1)
s = new char[n];
s[0] = 'J';
s[1] = 'o';
s[3]='h';
s[4]='n';
s[5]='\0';
Remark: (1) John = four characters + '\0'
```

Object type:
As has already been demonstrated earlier in this chapter, object types are translated into C++ by the name of the class and are not any different from other user defined types. The only inconvenience is that if the type of a data member is a class, this class must have been declared before. If a type is used before its declaration, the compiler will give an error message. If two classes have each other as *member class*, it will become necessary to declare at least one of the classes first and define them later. Example.:

```
class Person;        //declaration of classes
class Team;

class Person {       //definition of classes
   ...
   Team: ptr_to_myteam
   ...
};

class Team {
   ...
   Person: ptr_to_mypers[0-11]
   ...
};
```

VI. The DB-MAIN CASE Tool

The DB-Main case tool is one of the many results of the DB-Main programme, which is a research, development and technology transfer programme developed by the Computer Science Institute of the University of Namur. It is dedicated to database applications engineering, and more specifically to the reverse engineering, re-engineering, migration, integration, maintenance and evolution of such systems.

The DB-Main CASE tool can be used either as a tool set to support system engineering, or as a CASE tool development environment (i.e. a CASE tool factory). The tool offers basic support for most forward and reverse engineering activities. In this chapter, I analyse how it can support the object oriented database developing methodology by means of transformation scripts, analysis scripts and some other useful functions. I will not give an exhaustive list of all the existing functions, but just focus on the ones that are particularly useful for the proposed object oriented database design. A complete list of all the functions available in the tool can be found in [DB-MAIN,95] and [DBMAIN,97].

VI.1. Assistance for Simple transformations:

The table below presents the functions that can be used to transform a conceptual schema into a C++ object oriented logical schema:

Action:	Command:
Making the collection of subtypes disjoint	Click on the subtype and mark the corresponding **Disjoint** check box.
Transforming multi-et roles	Click on the multi-ET role to be transformed and execute the command **Transform\role\multi-ET-> rel-types**
Transforming non functional relationship types into entity types	Click on the relationship type to be transformed and execute the command **Transform\Rel-type\ ->Entity type**
Transforming functional relationship types into object attributes	If necessary, adapt the cardinalities (to avoid circular chains of mandatory attributes) by double clicking on the concerned cardinality. Click on the relationship type(s) to be transformed and execute the command **Transform\Rel-type\->Object type**

The table below presents some functions that can be used to translate the now obtained schema into a C++ object oriented physical schema:

Action:	Command:
Name processing (a list of C++'s reserved keywords can be found in Borland C++'s online help.	**Transform\Name processing...** To replace hyphens (-) by underscores (_): **Add patterns, search for -, replace by _**

For each of the transformations above, when you execute one of the commands the dialogue box that will appear on screen will guide you through the rest of the process.

VI.2. The Assist Menu

Apart from these loose transformations, the DB-MAIN tool offers much more sophisticated help, which can be found in the Assist menu (Assist). This menu comprises a series of expert assistants dedicated to specific classes of problems. Two of these experts are particularly useful with respect to this thesis.

Assist\Global transformation:
The global transformation assistant carries out selected actions on selected objects in order to solve structural problems. For each outstanding class of constructs (the **problem**), the assistant proposes one or several transformations that replace them by equivalent constructs (the **solution**). The user can build, save and reuse customised transformation scripts dedicated to specific complex problems. A script is a sequence of operations that can be performed by an assistant. [DBMAIN,97]

After having made all collections of subtypes disjoint and having made sure there will not be any circular chains of object attributes once the schema transformed, the following script will suffice to obtain a C++ logical object oriented schema:

script:
Split rel-types with Multi-ET roles
Complex rel-types into entity types
Binary rel-types without attributes into objects

To obtain a C++ object oriented physical schema, one more line should be added to the script. This can be done by selecting Name Processing in the Global Transformations box and consequently by clicking on 'Add'. The script is completed by one line, namely
Name processing.

Remark: As has been explained in Chapter V, there are always two or three equivalent ways to transform relationship types into object attributes. The choice is up to the user in function of the C++ code he/she finally wants to generate. However, the assistant will not leave this choice up to the user, but will always transform objects according to the third possibility, i.e. by adding an inverse constraint.

Assist\Schema analysis:
The analysis assistant is able to detect specified structural problems of any complexity in the current schema. A structural pattern is defined by an object type and properties which the objects have to satisfy. Every time a rule is defined, it can be added to the text box 'rules' in order to form a script, which can be created, saved, reloaded and reused by the user. The assistant can be used in two ways, namely to validate the current schema, and to search the schema for specified objects. [DB-MAIN,97]

validation (press button **validate**)
The assistant searches the current schema for objects that do not satisfy the rules specified in the script. These objects are presented in a diagnostic window which can be used as a notepad. When a diagnostic message is selected, the assistant makes the offending object current in the schema.

search (press button **search**)
The assistant searches the current schema for all the objects that satisfy the rules specified in the script.

The following set of rules can be used to validate a C++ logical object oriented schema:

DISJOINT_in_ISA(no)	(meaning: disjoint ISA)
ALL_RT(no)	(meaning: no relationship types)
ALL_ROLE(no)	(meaning: no roles)
ALL_REF(no)	(meaning: no reference keys, but object attributes instead)
ALL_KEY(no)	(meaning: no access keys)

Remark: this script does not detect circular chains of object attributes as they have been defined in chapter IV.

The following rules can be added to the script to validate a C++ physical object oriented schema in which there are no more hyphens (-):
NO_CHARS_in_LIST_NAMES(-)
NONE_in_LIST_NAMES(this, if, while)

VI.3. Other useful Functions

Using generators:
The DB-MAIN CASE tool allows the user to generate code corresponding to a physical schema. Physical schemata can be translated into different languages such as SQL, CODASYL, C++, COBOL, etc. This is done by calling a corresponding program, written in Voyager2. The Voyager 2 development tool consists of a compiler named comp_v2.exe. This compiler accepts a Voyager 2 programme (file *.v2) and produces a precompiled file with the extension **.oxo**. Programs written in Voyager 2 can be called by executing the commands:

File\Execute Voyager...
File\Continue Voyager...
File\Rerun Voyager...
by clicking on the corresponding buttons on the toolbar for special mouse and key actions.

VII. The C++ Generator

The C++ generator, of which the source code can be found in Appendix 1, has been written in Voyager 2. Its name is genercpp.oxo. It generates the C++ code corresponding to the physical object oriented C++ schema discussed in chapter V. Unfortunately, the C++ generator cannot (yet) handle all the structures allowed in the formerly discussed model. Therefore, I have to introduce some extra restrictions on the logical and physical models. I also wrote an analysis script, which is included on the diskette, that can serve to validate a logical object oriented C++ schema once it is entirely adapted to the capabilities of the programme.

VII.1. A Restricted Model

VII.1.1. Overview of the Structures the C++ Generator can handle:

- Entity types:
  Entity types can comprise 0 to N (object) attributes, a primary identifier, (a) secondary identifier(s), 0 to N subtypes and 0 to N supertypes. Multiple inheritance is not supported. Once more, IS-A relations must always have the disjoint attribute.

- Attributes:
  - Attributes are always optional, i.e. the minimum cardinality is always 0.
  - The cardinality of single valued, atomic attributes must be 0-1. The deepness of attributes is unlimited, i.e. they can be 'wrapped' into an endless number of compound ones.
  - The cardinality of multivalued, atomic attributes must be 0-N. They are always implemented as lists.
  - The cardinality of single valued compound attributes must be 0-x, with $x < +\infty$ because they are always implemented as arrays. Idem dito for the cardinality of multivalued, compound attributes.
  - The cardinality of **object attributes** is always 0-1 or 0-N, in which case they are implemented as lists. They can belong to compound attributes and their deepness is unlimited. They can belong to one or more compound and multivalued attributes, although this would probably not make much sense.
  - Atomic attributes can be of the following types and are translated into specified C++ types:
    - char (+ specified length): translated into char [length]
    - varchar: translated into *char* [maxlength]
    - numeric: translated into *int* if the length of the number is strictly inferior to five digits and the number of decimal digits equals zero, translated into *double* if the length of the number is superior or equal to five digits. If the number of decimal digits does not equal zero, it is translated into *float.*
    - date: translated into *double*
    - float: translated into *float*
    - object attribute: translated by a *pointer to an object* of the corresponding class
    - boolean: translated into *bool*

Example:

CompoundMultivalAttr [0-3]
ObjectAttr [0-N]
CompoundMultivalAttr [0-6]
SinglevalCompoundAttr [0-2]
AtomicMultivalAttr [0-N]
ObjectAttr [0-1]
SinglevalAtomicAttr [0-1]
SinglevalAtomicAttr [0-1]

- Identifiers:
  - Identifiers can comprise one to many single valued, atomic attributes, which can be part of single valued, compound attributes. Identifiers cannot consist of groups.
  - The deepness of single valued, atomic attributes is unlimited.
  - Identifiers can be primary or secondary[1].

Example:

Demo
attr1[0-1] attr11[0-1] attr111[0-1] attr112 [0-1] attr2[0-1] attr3[0-1] attr31[0-1] attr32[0-1]
id': attr1.attr11.attr111 id': attr1.attr11.attr112 attr2 attr3.attr31

Figure VII.1 Primary and secondary identifiers

VII.1.2. Extra restrictions on the C++ object oriented logical model:

With regard to the C++ object oriented logical model defined in II.3, there are some restrictions, namely the schema cannot take:
- any (object) attribute with any minimum cardinality > 0.
- single valued, atomic attributes with maximum cardinality x, with 1< x <N.
- compound attributes implemented as lists.

[1] Since the minimum cardinality of attributes is always 0, identifiers are in fact always secondary. Still, the programme would not be troubled by primary identifiers.

- identifiers comprising multivalued attributes or compound attributes. However, for single valued, compound attributes consisting exclusively of single valued, atomic attributes, it suffices to declare all its atomic attributes identifier to obtain the same effect.
- identifiers comprising multivalued, compound attributes or attributes being part of such attributes. However, this would be very unhappily chosen identifiers anyway.
- although some of these structures have already been mentioned in II.3. the C++ generator cannot take referential keys, access keys, relationship types or collections

VII.1.3. Analysis Script to Validate the Schema.

The following analysis script can serve to detect those objects that do not comply with the logical schema proposed in this chapter:

ALL_COLL(no)	meaning: no collections(*)
not DISJOINT_in_ISA(no)	ISA must always be disjoint(*)
ALL_RT(no)	no relationship types(*)
ALL_ROLE(no)	no roles(*)
MIN_CARD_of_ATT(0 0)	minimum cardinality is always 0(*)
GROUP_per_GROUP(0 0)	groups cannot comprise groups
ALL_KEY(no)	no access keys(*)
ALL_REF(no)	no reference keys(*)

V2_CONSTRAINT_on_ATT E:\DBMAIN\genercpp.OXO max_rep_si_attr_0_or_N(!)
meaning: this function is included in the C++ generator and verifies whether the maximum cardinality of atomic attributes is either 1 or N.

V2_CONSTRAINT_on_ATT E:\DBMAIN\genercpp.OXO Max_rep_co_attr_not_N(!)
meaning: this function is included in the C++ generator and verifies whether the maximum cardinality of compound attributes is different from N.

V2_CONSTRAINT_on_ATT E:\DBMAIN\genercpp.OXO Check_all_card_for_id(!)
meaning: this function is included in the C++ generator and verifies whether the maximum cardinality of compound attributes containing simple attributes that are (part of) an identifier, is 1.

(*) If this object makes part of the schema from which code is generated, the C++ generator will just deny it and still generate a programme that can be compiled.
(!) In the script 'validgen.ana', the directory E:\DBMAIN\Cplusp10.OXO should be made conform to the place where the C++ generator is stored on the computer.

To validate the physical schema, two more rules can be added:

NO_CHARS_in_LIST_NAMES(-)	meaning: no hyphens
NONE_in_LIST_NAMES(this, if, while[2])	meaning: the names of the schema, entity types, rel-types, attributes, roles and groups aren't in the list <list>.

VII.2. Chronological Overview of the Operations Performed by the Programme

The C++ generator 'writes' the C++ programme in the following order:

- include statements
- an enumeration holding one value for each class
- forward class declarations
- for each class: class definition (any class is generated before any of its subtypes)
    - public data members
    - private data members (and declarations + definitions of inline member functions if the class has one or more identifier(s)) .
    - declarations of public data members

- global variables
- initialisation of static variables
- for each class: definition of its non-inline member functions:
    - constructor
        - call to function that checks identifier(s)
        - code to assign values to data members
        - code to insert object into list of objects of its class
        - code to initialise lists and iterators related to data members corresponding to multivalued attributes with cardinality 0-N[3].
        - code to manage relations defined by object attributes

    - destructor
        - destruction of lists and iterators
        - removal of object from its list
        - code to keep cardinality constraints concerning relations defined by object attributes intact.

    - get_first() to get the first object of the list
    - get_next() to get the next object in the list
    - get_previous() to get the previous object in the list
    - what() to find which class a pointer is pointing (in case of inheritance)
    - find_id() to detect whether the identifier constraint has been respected for a new object

[2] An exhaustive list of all C++'s reserved words can be found in Borland C++'s online help.
[3] Why this is done after having assigned values to the data members will be explained in the next chapter.

- function delete_one_class(...) to delete all the objects of a specified class (by means of a template).
- function send_error_message(...) to send a message corresponding to the detected error.
- empty main programme, that reminds the user to work with the C++ exception handling mechanism.

VII.3. Some Technical Details of the C++ Generator.

The C++ generator, named GenerCpp.oxo is a programme written in Voyager 2, which is an imperative language with original characteristics such as a primitive type *list* with garbage collection and declarative requests to the predefined repository of the DB-MAIN tool [ENGLEBERT, 96]. The source code of the programme is represented in appendix I.

GenerCpp makes a list of all the entity types contained by the logical schema it has to generate code from. This is done by a 'depth first search' algorithm to make sure that any supertype always precedes all its subtypes in the list. Classes are generated in compliance with the order in which they figure into this list. If subtypes were generated before their supertypes, the generated programme would not compile. The programme will walk through this list time and again to make sure that for each block of code, the classes will always appear in the same order.

Many functions and procedures of the programme needed to be recursive. This was inevitable because, in the logical and physical schemata, a supertype can contain an unlimited number of subtypes and vice versa, the 'deepness' of an attribute contained by x compound attributes is not restricted, an identifier can comprise x attributes, etc. This flexibility in the schemata is only possible if the programme treats certain objects recursively.

Since the same information has to be retrieved from the schema at different moments, the programme is chopped up into numerous functions and procedures according to this criterion. For instance, whether a class has subtypes has to be known at five different places in the programme. This is taken care of by a boolean function called *CheckIdYesNo(...)*.

VIII. Case Study

This final chapter comprises two parts. Whereas in chapters three and five, C++ code and pseudo-code were suggested for each structure from a rather theoretical point of view, the first part of this chapter will give an overview of the code that is actually generated by the C++ generator, taking into account some practical considerations. The second part is a practical application of the methodology elaborated in the previous chapters, stretching from conceptual design to code generation. The chapter does not include entirely generated programmes, which can be found in the appendices.

VIII.1. The Generated C++ Code

The code is discussed according to the order in which it is generated, i.e. as it is listed in the previous chapter.

- include statements:
There are three include statements:
#include <string.h>, #include<stdio.h>,#include<iostream.h>,
and when at least one attribute has cardinality 0-N, a fourth include statement is generated:
#include "..\lists\tlist.h", which is a call to a library necessary to implement lists, with a relative path name.

- an enumeration holding one value for each class:
This value will be composed by the concatenation of "w_" and the name of the class and will be used by virtual member function *What()*.

- forward class declarations:
As has been said before, declaring the classes before defining them is necessary as soon as there are *class members*. The name of a class is formed by the concatenation of "c"+ the name of the corresponding entity type in the physical schema. Entity type *Person*, for instance, becomes class *cPerson*. This permits the user to create classes inheriting from the generated class listening to the name of the corresponding entity type. By putting the derived classes in other .h and .cpp files, the user can change the physical schema later on and regenerate C++ code, which will then be completely compatible to the files containing the derived classes added by the user.

-class definition
- If the class is a derived one, the name of its base class will be indicated. Classes always inherit publicly from their base class.
- public data members:
  All data members corresponding to attributes in the physical schema are public. They are all wrapped into a structure of which the name is formed as follows: <*class name*>_buf and the corresponding type will be named t_c<*class name*>_buf. Since no methods are generated to access them, they have to be accessed directly. For compound attributes, a structure will be generated bearing the name of the attribute and with as type name t_<*attribute name*>.

Multivalued attributes with cardinality [0-N] are implemented as lists of elements of the corresponding type. For each list, an iterator is generated to walk through it. The code will be:

Tlist <*attribute type*> * <*attribute name*>; (list)
Titer <*attribute type*> * i_<*attribute name*>; (iterator)

Multivalued compound attributes, with cardinality 0-X (X<N), are implemented as arrays the length of which is the maximum cardinality of the corresponding attribute.

- private data members and inline member functions:
To be able to insert each object in a list of objects of its class, each object contains three private data members: *next*, *previous* and *first*. The last one is a static data member, to avoid having to generate a global variable instead. They contain pointers to the next, previous and first class respectively.

If the object has one or more identifiers, the member function *find_id(...)* will be generated. The function has one argument for each data member belonging to at least one identifier (primary or secondary).

Also only in the case of at least one identifier, two more private boolean data members are generated: *init0_N_attr* and *inserted_into_list*, accompanied by the four member functions to get and to set their values. They are necessary for the following reason: when the constructor initialises the lists of an object and inserts the object into the list of objects of its class, the destructor has to do the reverse operation. If the identifier constraint is not respected, the object will be destroyed before these two operations are performed and the destructor cannot destroy what has not been initialised before. Therefore, the constructor will put these data members at *true* if the work has been done and at *false* if not. The destructor will only act when they are at *true*.

- declarations of public data members:
The remaining member functions are the constructor, the destructor, the static member function *get_first()*, *get_next()*, *get_previous()* and the virtual member function *what()*.

Member function *What()* returns a value of the formerly defined enumeration. It can be used to find out to which class of objects a pointer points to. For example:
if (mypointer->What() = = w_Person) { ... }
The member function is declared virtual to allow each derived class to provide its own version. Moreover, since the function is useless for abstract base classes, which correspond to ISA-relations with the attribute **Total**, in such cases, it is declared *pure virtual* by the initialiser *=0* and no objects of that abstract class can be created. When *What()* is declared in a derived class, this class will no longer be abstract.

- global variables:
For each class, a pointer named *<class name>_former* points to the last object that has been created of this class. When this global variable is declared, it is set at NULL. It is used by the constructor to get a hold of a pointer to the previously created object of the same class.

- initialisation of static variables:
The pointer *first* of each class is set at NULL before the first object of the class is created.

- definition of the constructor:
The constructor of a base class takes only one argument: a structure of type *t_<class name>_buf* which serves as buffer and contains all the data members of that class. The constructor of an inherited class takes one argument for each of its base classes up along the hierarchy. The constructor always uses reference parameters to avoid a senseless consumption of time and space.

Depending on what follows, some local variables might be generated. Then if, the object has at least one identifier, data members *init0_N_attr* and *inserted_into_list* are set at *false*, before a call is made to *find_id(...)* . If this function detects a double identifier, it will call the object's destructor and throw an error message, otherwise the creation of the object continues.

The function *memcpy(...)*, pre-existing in C++, assigns the values to the data members controlled by the user. It also takes reference parameters. Thanks to this function the whole buffer is assigned at once.

Finally, the object is inserted into the list of objects of its class, and the lists corresponding to attributes with cardinality O-N are initialised, whereby, in case of identifiers *inserted_into_list* and *init0_N_attr* are immediately set at *true*.

- definition of the destructor:
Depending on the work the destructor of a class will have to do, some local variables might be generated. Then, in case of one or more identifiers, the destructor tests whether *inserted_into_list* is at *true* and if so, deletes the lists and their corresponding iterators. The same holds for the next step: in case of one or more identifiers, the destructor tests whether *inserted_into_list* is at *true* and if this is the case, it takes the object out if its list, otherwise this is done right away. Finally, the destructor, who knows which objects of which classes may contain pointers to the object by which it is called, searches all those objects for pointers to this object and either sets those pointers at NULL, or, if it is a list of pointers, it removes the pointer from this list. In this way, the destructor keeps intact the cardinality constraints concerning relations defined by object attributes.

- definition of member functions *get_first( ), get_next( ), get_previous*
These functions merely return the first, next and previous object respectively.

- **definition of member function *what( )***
  As has been said before, this function returns a value of the previously defined enumeration. The only particularity of it is that, it is only defined outside of its class if it is not pure virtual (cf. supra.) , i.e. if it does not belong to an abstract class.

- **definition of member function *find_id(..)***
  This function takes as argument all the data members that are part of an identifier. However, if a data member belongs to more than one identifier, it only appears one time as an argument. Then, the function will walk through the list of objects of its class and check for each identifier if the object to be created has unique values.

- **definition of stand alone function *delete_one_class(S *aptr)***
  The function is preceded by the statement *template <class S>*. In this way, the function can take a pointer to the first object of a list of any class and delete all of them.

- **definition of stand alone function *send_error_message(int catchint)***
  This function calls an error message appropriate to the integer it receives. Right now, its only usable message is the one sent when a double identifier is detected. It functions according to C++'s exception handling mechanism.

- **definition of the main programme**
  The main programme is exclusively generated to make the programme compilable. It also suggests the user to use the C++ exception handling mechanism when creating objects.

  To create a new object, the user is suggested to work as demonstrated in the example:

```
cPerson::t_cPerson_buf pers1 = {100, "Smith", "John"};              (1)

try {
 cPerson * first_pers = new cPerson(pers1);                         (2)
 for (i=1; i<3; i++) {                                              (3)
  first_pers->cPerson_buf.luckynumbers->
  AddLast(new int (i));
 }
}

catch(int i) {                                                      (4)
  end_error_message(i);
}
```

(1) The values are passed to the structure (*struct*) functioning as buffer.
(2) The object is created.
(3) The values are added to the list corresponding to one of the object's data members by the method *AddLast(...)*. The user does not have to worry about the initialisation and the destruction of any list. Furthermore, such lists can be treated by means of the following self-evident functions: *First( ), Last( ), Current( ), AddFirst(cell *c), AddLast(cell *c), Remove( )* and *Length( )*.

(4) If a double identifier is detected, an error message is sent by the constructor, and caught right here.

VIII.2. An Example

This example shows how the formerly developed methodology can be applied. It stretches from conceptual design to code generation and supposes the designer uses the DB-MAIN tool to guide him/her through this process.

VIII.2.1. Conceptual design

The conceptual schema represented in Figure VIII.1. presents the following real-world: a Team, consisting of soccer players, which is identified by its number and which also has a name, comprises 0 to N instances of Member. Each Member is a Person, identified by the combination of his name and first name. A Person can have 0 to N nicknames and 1 to 5 addresses (street, number, city and for each address 0 to 5 telephone numbers). Each member has an identifying number and it is also indicated whether he has already received his salary for the current month or not. Each member has a car, which is identified by its number plate, is of a certain make and can be shared by 0 to N members.

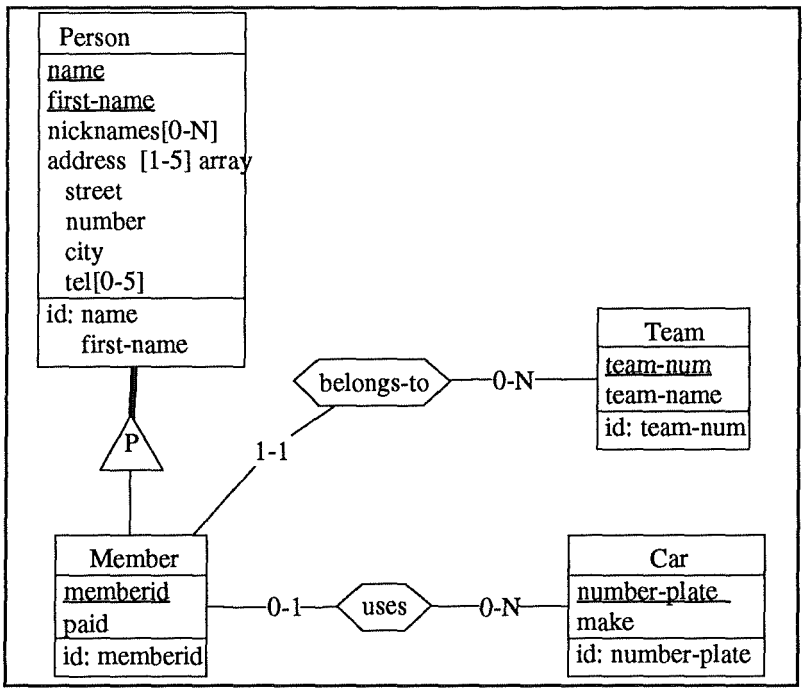


Figure VIII.1 Conceptual schema of a team.

VIII.2.2. Logical design

In order to conserve the schema developed in the former paragraph, it suffices to execute the command **Product/Copy schema**. A new version name has to be specified, for instance 'logical', because, in this way, the schema expresses its main characteristic. The schema must be transformed into a C++ logical object oriented one, according to the transformation plan presented on page 21. Since there are neither collections of none disjoint subtypes, nor multi-ET roles or non functional relationship types, only functional relationship types need being transformed. This process can be assisted by the assist menu: **Assist/Global transformations**. The script comprises only one line: **"binary rel-types without att. into objects"**. Unfortunately, when this transformation is performed, the schema still has a few characteristics that **genercpp.oxo** cannot deal with. Therefore, the schema has to undergo three more transformations: firstly, the inverse constraints have to be taken away and the minimum cardinality of all the attributes needs to become one[1], and secondly, the maximum cardinality of atomic attributes must be either 1 or N. To take away the inverse constraints, the designer must select the corresponding groups and delete them by pushing on the delete button. Since the maximum cardinality of attribute *tel* must be either 1 or N, the designer is obliged to set it at N, and in this way it will be implemented as an unlimited list of telephone numbers. The final C++ logical object oriented schema is represented in **Figure VIII.2**.

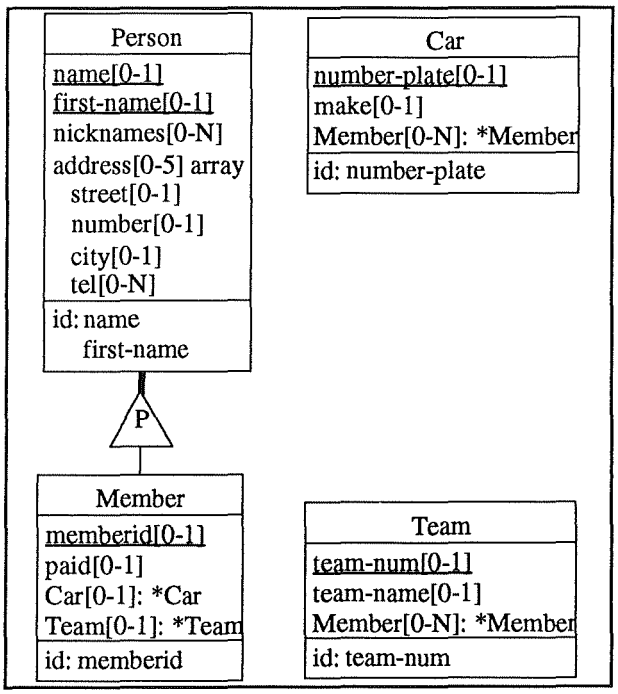


Figure VIII.2 Logical schema of the Team.

[1] In fact, if the designer does not perform these transformations, genercpp.oxo will still generate exactly the same code: it just denies the inverse constraint and supposes the minimum cardinality is zero.

VIII.2.3. Physical design

In the first place, the designer should execute the command **Product/Copy schema** and specify a new version name, for instance 'physical'. To obtain the C++ object oriented physical schema, it suffices to replaces all hyphens by underscores since none of the names coincide with a reserved word. Once this is done, the designer can test the validity of the schema by loading the validation script **validgen.ana** by means of the command **Assist/Schema analysis/load**. When the script is loaded, it must be assured that the path towards genercpp.oxo held by the last three functions is the correct one. Now, it suffices to click on **OK** to check whether the script is valid. If this is the case, the message **'the schema verifies all rules'** will appear on screen. The validated physical model is presented in Figure VIII.3.

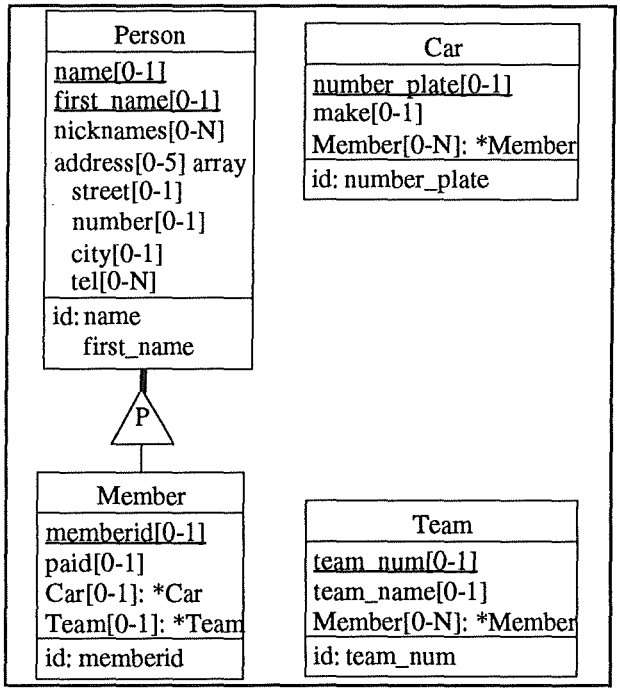


Figure VIII.3 Physical schema of the team.

VIII.2.4. Generating code

By executing the command **File/Execute Voyager**, the designer can call **genercpp.oxo** to generate the code corresponding to the physical schema. The generated code is completely represented in appendix III. This paragraph will only highlight the most important parts of it.

To begin with, entity type Person is translated by class *cPerson,* which is a base class:

```
class cPerson {
public:
  struct t_cPerson_buf {
    char name[25];
    char first_name[25];
    Tlist <char> * nicknames;
    Titer <char> * i_nicknames;
    struct t_address {
      char street[31];
      int number;
      char city[17];
      Tlist <double> * tel;
      Titer <double> * i_tel;
    } address[5];
  } cPerson_buf;
private:
  cPerson* next;
  cPerson* previous;
  static cPerson* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cPerson * find_id(char name[25],char first_name[25]);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
public:
  cPerson(t_cPerson_buf &cPerson_bu);
  ~cPerson();
  static cPerson * get_first();
  cPerson * get_next();
  cPerson * get_previous();
  virtual w_class what() = 0;
};
```

All the data members corresponding to attributes of entity type Person, are wrapped into one structure, named *cPerson_buf.* For both *name* and *first_name*, space for 25 characters will be reserved. *Nicknames* corresponds to a multivalued attribute with cardinality 0-N and consequently, it is translated by a data member that can hold a list accompanied by its iterator to walk through it, called *i_nicknames. Address* is a structure itself, because it corresponds to a compound attribute. It subsumes *number*, an integer because the number was specified to take strictly less then five digits, *city*, *tel*, and *i_tel. Tel* can hold a list of doubles, because it was specified in the physical schema that the number would be composed of 10 digits. Since *address* corresponds even to a compound **multivalued** attribute with cardinality [0-5], its length is specified between square brackets: [5].

Since *Person* has both an identifier and multivalued attributes it also has the data members and member functions to take care of them: *init0_N_attr, inserted_into_list, find_id(...)*, etc. The last function of the class *What( )* is declared pure virtual, because person is an abstract class, which can have no instances.

Enough has been said about constructors, but I would like to attract the readers attention to the place in the constructor of cPerson where the list *nicknames* is initialised:

```
cPerson::cPerson(t_cPerson_buf &cPerson_bu)
{ ...
 int i;
 // Initialise lists for multivalued attr
 for (i=0; i<5; i++) {
 cPerson_buf.address[i].tel= new Tlist<double>(TLIST_REMOVE_WITH_NEW);
 Person_buf.address[i].i_tel= new iter<double>(cPerson_buf.address[i].tel);
 };
 ...
};//end constructor
```

First an integer *i* is generated to be able to generate a *for* loop (although thisinteger could have been declared within the for loop). Then, lists *tel* and their iterators *i_tel* are initialised, all five of them because they belong to *address*. The destructor will deallocate memory afterwards. *TLIST_REMOVE_WITH_NEW*, indicates that when the list is destroyed, its elements are destroyed as well. For lists of pointers to objects, this will become *T_LIST_NOT_REMOVE*, because these objects have to remain untouched and are merely removed from the list.

Finally, I would like to draw the attention to the definition of the constructor of class *cMember*, which inherits from *cPerson* and starts as follows:

```
cMember::cMember(t_cMember_buf &cMember_bu, t_cPerson_buf cPerson_bu):
                                                    cPerson(cPerson_bu)
{
...
}
```

Since *cMember* has *cPerson* as base class it has two arguments, one structure for *cPerson* and one for the class by which it is called. The rest of the code can be found in appendix III.

IX. Conclusion

In conclusion, I shall make an evaluation of this work, which covers five areas: the suitability of the aid offered by the DB-MAIN CASE tool, the C++ generator, the aptness of the C++ language as a database management system and, finally, the generated code.

The DB-MAIN CASE tool proves to be perfectly suitable for object oriented database design. It appeared perfectly possible to write transformation scripts to help obtain the C++ object oriented logical and physical schemata. However, there is one drawback to the transformation assistant, namely when it transforms relationship types into object attributes. As has been discussed in chapter IV, there are always two or three semantically equivalent ways to transform object attributes. Unfortunately, the assistant does not leave the choice up to the designer but always takes the possibility with the inverse constraint. In fact, this choice should be up to the designer who can do so in compliance with the criteria outlined in chapter IV.

The C++ generator is certainly not yet a complete programme. It could be improved on three levels. It should be able to deal with physical schemata that are semantically richer, and also to generate code that can store data permanently and that provides a user interface. In the following paragraphs, I'll discuss each level in more detail.

As far as the first level is concerned, not being able to deal with quite some integrity constraints is certainly the programme's most important shortcoming. Cardinality constraints in particular are undoubtedly the programme's weak point. The programme cannot yet handle obligatory attributes, nor is it capable of generating single valued atomic attributes with a maximum cardinality different from 1 or N, implementing compound multivalued attributes as list, etc. Furthermore, it cannot generate the inverse constraint, identifiers comprising one or more multivalued attributes, to mention just the most important shortfalls. The most straightforward improvement that can be made to the programme consists of making it capable of handling mandatory attributes having 1 as minimum cardinality. This would be a great step forward in return for the insertion of relatively few lines of code. In addition, it becomes ever more easy to extend the programme because an increasing number of functions and procedures that have been written before, can be reused.

But also other steps to ameliorate the programme should be taken, apart from those concerning the semantics of the physical schema. To begin with, the generated code provides no means to store any data permanently. Instead it can only work with volatile data. This should be taken care of to make the programme really interesting. In a later stage, one could start thinking of adding an interface to use the generated programme. Edit boxes could for instance be provided to retrieve data from the keyboard, with an OK to enable to user to store the data. Dialogue boxes could appear to inform the user of a violation of an integrity constraint, and so on. Of course, every user will use the generated code for a different purpose, but one could think of generating an interface according to the desires the user expresses when the programme is generated.

All these adaptations will lead to the expansion of genercpp.oxo, which was written in Voyager 2. This language is remarkably powerful when it comes to working with lists and the DB-MAIN repository. Unfortunately, it does not allow to split up a programme into different modules and this might become annoying if the generator is to become a very large programme. It is also a pity that there is not yet a debugger for Voyager 2.

A few conclusions can also be drawn concerning the aptness of the C++ as a database management system. Although C++ makes a very poor environment for database design with very few declarative structures, as has been ascertained in chapter three, it does have a few positive characteristics as well. For instance, its inheritance system can be immediately used to translate ISA-relations, except for non disjoint ones, and C++ *structures* are a perfect device to translate compound attributes. For many other structures found in the physical schema, extra code has to be added to implement them, which is a time consuming job. But after all, as soon as some programme has been developed to generate the necessary code, time can no longer be an objection.

Finally, the generated code can use some criticism as well. Its most urgent need is standardisation. At some points, certain ways of coding were chosen to facilitate the creation of programme that generates the code. For this reason, the parameter of the constructor of a base class, for instance, has only one argument, namely a C++ *structure* and the data it contains are all passed to the object by the function memcpy( ). In fact, the constructor should have one argument for each attribute, and the values should be assigned to the object's data members one by one. This way, the user will be able to work with the object's constructor in the way he/she is most familiar with. After all, generating one argument for each data member is not all that difficult, but unfortunately this became only clear once I had written the function that checks the uniqueness of the identifier.

Moreover, the generated code lacks standardisation in yet another area. Namely, it does not have member functions to get, set and modify the values of those data members that correspond to attributes. Such functions are only generated for data members that belong standard to every object, such as *first, next, previous, inserted_into_list*, and so on. In this way, the generated programme does not get too long, especially if it is generated for a physical schema with deeply nested attributes contained by compound multivalued attributes. However, some users who are rather keen on object oriented programming might not like the idea of having to access objects directly. Especially not when sometimes they have to and other times they can not. This is the case, because, for instance, to work with lists generated for multivalued attributes, member functions do have to be used. This is due to the fact that the libraries the programme uses to implement these lists contain member functions such as AddLast() to add elements to the lists, etc.

Certainly, the generated code is has a lot of powerful sides too. It is very compact and its constructors and destructors are smart enough to allocate memory, to deallocate memory, to insert the object into and remove an object from the list of objects from its class, to be considerate of subclasses and relationships with other classes, and so on. But the longer I worked on it, the wider became the range of possibilities I thought of to make the programme ever more powerful!

X. Bibliography

[**AMMERAAL, 95**] Ammeraal L., *"Basiscursus C++"*, Schoonhoven, Academic Service Informatica, 1995.

[**BATINI, 92**] Batini C., Ceri S., Navathe S. B., *"Conceptual Database Design: an Entity-Relationship Approach"*, The Benjamin/Cummings Publishing Company, Redwood City, United States, 1992.

[**BODART, 94**] Bodart F., Pigneur Y., *"Conception assistée des systèmes d'information. Méthode, modèles, outils"*, Masson, Paris, 1994.

[**COBUILD, 93**] *"Collins Cobuild English Usage"*, The University of Birmingham, London, HarperCollins Publishers, 1993.

[**DATE, 95**] "Date C.J., *"An Introduction to Database Systems - Volume I* (6th edition), Addison-Wesley, 1995.

[**DB_MAIN, 95**] *"DB_MAIN Tutorial. Volume 1: Introduction to Database Design"*, DB-Main Project, Institut d'Informatique, FUNDP, Namur, 1997.

[**DB-MAIN, 97**] *"The DB-MAIN Database Engineering CASE tool - Version 3 - Functions Overview"*, DB-Main project, Institut d'Informatique, FUNDP, Namur, 1997.

[**ENGLEBERT, 96**] Englebert V., *"Voyager II, Reference Manual"*, Institut d'informatique, FUNDP, Namur, 1996.

[**HAINAUT, 97**] Hainaut J-L., Englebert V., Hick J-M., Henrard J., Roland D., *"Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study"*, Institut d'informatique, FUNDP, Namur, 1997.

[**HAINAUT, 96a**] Hainaut J-L., *"BASES DE DONNEES - Technologie et Conception. Première Partie: Conception d'une base de données"*, syllabus, Institut d'Informatique, FUNDP, Namur, 1996.

[**HAINAUT, 96b**] Hainaut J-L., *"BASES DE DONNEES - Technologie et Conception. Seconde Partie: Technologie des bases de données"*, syllabus, Institut d'Informatique, FUNDP, Namur, 1996.

[**HAINAUT, 96c**] Hainaut J-L., Roland D., Henrard J., Englebert V., Hick J-M., *"DB-MAIN, A General-purpose CASE Environment for Advanced Database Applications Engineering"*, Institut d'Informatique, FUNDP, Namur, 1996.

[**OXFORD, 95**], *"OXFORD Advanced Learner's Dictionary"*, Oxford, Oxford University Press, 1995.

[**MULLER, 97**], Muller P-A., *"Modélisation objet avec UML"*, Eyrolles, Paris, 1997.

[**STROUSTRUP, 97**] Stroustrup B. *"The C++ Programming Language"*, Addison Wesley, Reading, Massachusetts, 1997.

APPENDICES

I. The Source Code of the C++ Generator (Lg.: Voyager 2) 59
II. The Generated C++ Code. Example 1 85
II.1. Physcial Schema 85
II.2. The C++ Code Corresponding to the Schema 86
III. The Generated C++ Code. Example 2 93
III.1. Physical Schema 93
III.2. The C++ Code Corresponding to the Schema 94

I. The Source Code of the C++ Generator (Lg.: Voyager 2)

Remarks:

(1) The corresponding .oxo file, named **genercpp.oxo** to run the programme is delivered on the diskette in the back cover of the thesis.

(2) The first three functions in the programme are export functions that are not part of the C++ generator itself. They are called by the validation script to validate the C++ logical object oriented schema.

```
/****************************************
 **          Programme generating  C++        **
 **           Written in Voyager 2             **
 **           by Pascal Lambers août 1998      **
 ****************************************/

string: REMARK="//", file_name, BLANCS="  ";
string: insertion;
file: out;
schema: sch;

/*-------------------------------------------------------------------------------------------------------
  export f. to validate the C++ logical OO schema, for which this progr
  can generate C++ code. DOES NOT MAKE PART OF THE CODE GENERATOR!!
  The max. card. of a si attr must be either 0 or N!
-------------------------------------------------------------------------------------------------------*/
export function integer Max_rep_si_attr_0_or_N (attribute: at, string: car)
{
 if GetType(at) = SI_ATTRIBUTE then {
   if at.max_rep <> 1 and at.max_rep <> N_CARD then { return 0; }
   else { return 1; };
 }
 else { return 1; };
}

/*-------------------------------------------------------------------------------------------------------
  export f. to validate the C++ logical OO schema, for which this progr
  can generate C++ code. DOES NOT MAKE PART OF THE CODE GENERATOR!! The
  max. card. of a co attr can't be N!(because N is implemented as a list)
-------------------------------------------------------------------------------------------------------*/
export function integer Max_rep_co_attr_not_N(attribute: at, string: car)
{
 if GetType(at)= CO_ATTRIBUTE then {
      if at.max_rep = N_CARD then { return 0; }
      else { return 1; };
 }
 else  { return 1; }
}
```

```
/*-------------------------------------------------------------------------------------------
 export f. to validate the C++ logical OO schema, for which this progr
 can generate C++ code. DOES NOT MAKE PART OF THE CODE GENERATOR!! The
 max. card. of the co_attr to which si_attr being part of an identier
 belong, is always 1!
-------------------------------------------------------------------------------------------*/
export function integer Check_all_card_for_id(group: gr, string: st)
 real_component: rc;
 integer: typeint, i;
 si_attribute: siattrib;
 cursor: cur;
 list: idattrlist;
{
 if gr.primary or gr.secondary then {
  for rc in GetListOfComponents(gr) do {
   typeint:= GetType(rc);
   switch (typeint) {
    case SI_ATTRIBUTE: siattrib:=rc;
                       cur:= member(idattrlist, siattrib);
                       if IsVoid(cur) then {
                        AddLast(idattrlist,siattrib);
                       };
    case CO_ATTRIBUTE: ChercheAttrLesPlusLoins(idattrlist, rc);
   }; /*end switch*/
  }; /*end for*/

  for siattrib in idattrlist do { /*idattrlist contains si attr*/
   if CountThisBranch(siattrib,0) > 0 then { return 0; };
  };
 }; /*end if*/
 return 1;
}

/*-------------------------------------------------------------------------
  Proc generating #includes in the beginning of the C++ progr
 -------------------------------------------------------------------------*/
procedure GestionInclude()
 data_object : a;
 list : attlist;
 cursor: ac;
 si_attribute: sa;
 integer: print_yes_no;
{
 printf(out,["#include <string.h>\n"]);
 printf(out,["#include <stdio.h>\n"]);
 printf(out,["#include <iostream.h>\n"]);
 /* If at least one attribute has cardinality 0-N the following
   include has to be genereated: (Relative path name!) */
 attlist:= DATA_OBJECT[a]{@SCH_DATA:[sch] with GetType(a) = SI_ATTRIBUTE};
 attach ac to attlist;
 print_yes_no:= 0;
 while IsNoVoid(ac) and print_yes_no= 0 do {
   sa:= get(ac);
   if sa.min_rep = 0 and sa.max_rep = N_CARD then
    { print_yes_no:=1; };
   ac>>;
 }
```

```
  if print_yes_no=1 then {
      printf(out,["#include \"..\\lists\\tlist.h\"\n"]);
  };
}


/*-------------------------------------------------------
  Proc generating global variables
  ------------------------------------------------------- */
procedure GenererVarGlobales(entity_type: enti)
{
 printf(out,["c",enti.name," * "]);
 printf(out,["c",enti.name,"_former = NULL;\n"]);
}


/*-------------------------------------------------------------------------------
  F. for the creation of a file where the C++ generated progr will be
  stored. Returns an integer: 0 if creation ok, 1 if not.
  -------------------------------------------------------------------------------*/
function integer OuvreFichier()
{
 file_name:=BrowsePrint("Save C++ Generation As ...",
 "CPP Files (*.cpp)|*.CPP|All Files (*.*)|*.*","CPP");
 out:=OpenFile(file_name,_W);
 if IsVoid(out) then {
 return 1;
 }
 else {
     return 0;
     };
}


/*---------------------------------------------------------------
  F. returning a list of all the supertypes of an ET
  ---------------------------------------------------------------*/
function list GiveListDeSuperTypes(entity_type: entity_t)
  entity_type: et;
  cluster: clu;
  sub_type: sub;
{
 return  ENTITY_TYPE[et]{ENTITY_CLU: CLUSTER[clu]{CLU_SUB:
       SUB_TYPE[sub]{@ENTITY_SUB:[entity_t]}}};

}


/*-------------------------------------------------------
  Fonction retourning the supertype of an ET.
  Precondition: the supertype exists!!!!!
  -------------------------------------------------------*/
function entity_type FindSuperType(entity_type: et)
{
 return GetFirst(GiveListDeSuperTypes(et));
 /* Pour l'instant l'héritage multiple n'est pas permis,
    et la liste ne consistera que d'un seul élément  */
}
```

```
/*-------------------------------------------------------------------------
  Recursive F. searching the ET to which a given attr. belongs
  precond.: dobj MUST be owned by an ENTITY_TYPE!
  -------------------------------------------------------------------------*/
function entity_type GetOwningEntityType(attribute: attr)
 list: mylst;
 owner_of_att: ow;
 entity_type: ent;
 attribute: attrib;
 co_attribute: coattr;
{
 mylst:= OWNER_OF_ATT[ow]{OWNER_ATT:[attr]};
 ow:= GetFirst(mylst);
 if GetType(ow)= ENTITY_TYPE then {
      ent:= ow;
      return ent;
 };
 coattr:= ow;
 return GetOwningEntityType(coattr);
}

/*-----------------------------------------------------------
  F. returning the type of an obj attr, i.e. another ET
  precond: objat.type=OBJECT_ATT!!
  -----------------------------------------------------------*/
function entity_type ReturnTypeOfObjAttr(si_attribute: objat)
 data_object: dato;
 entity_type: myent;
{
 dato:= GetFirst( DATA_OBJECT[dato]{DOMAIN:[objat]} );
 myent:=dato;
 return myent;
}


/*-----------------------------------------------------------
  F. giving a corresponding C++ type (as a string)
  for a simple attribute
  -----------------------------------------------------------*/
function string FindCppType(si_attribute: si)
 entity_type: ent;
{ switch (si.type) {
       case DATE_ATT:
            return "double";
       case CHAR_ATT:
            return "char";
       case VARCHAR_ATT:
            return "char";
       case NUM_ATT:
            if si.decim = 0 then {   if si.length>4 then { return "double"; } else { return "int";}  }
            else { return "float"; };
       case FLOAT_ATT:
             return "float";
       case BOOL_ATT:
             return "bool";
```

```
        case OBJECT_ATT:
            ent:= ReturnTypeOfObjAttr(si);
            switch (si.max_rep) {
              case 1     : return "c" + ent.name + " *";
              case N_CARD: return "c" + ent.name;
              otherwise  : return "error";
            };
      };
}


/*-------------------------------------------------------------------------------
  F. returning the size of an attr. of type DATE_ATT, CHAR_ATT,
  or VARCHAR_ATT. P. ex.: (30) pour char(30)
-------------------------------------------------------------------------------*/
function string FindSize(si_attribute: si)
{ switch (si.type) {
   /* une date prend 8+1 caractères p.ex.: 13-02-98 */
   /* +1 because strings en C++ are null terminated */
   /*case DATE_ATT:
     return "["+StrItos(9)+"]";*/
   case CHAR_ATT:
     return "["+StrItos(si.length + 1)+"]";
   otherwise :
     return ""; /*càd retourner une chaîne vide */
   };
}

/* ------------------------------------------------------
 f. that treats simple attributes (of any level)
 ------------------------------------------------------*/
procedure TreatSiAttr(si_attribute: si_a, string: tab)
  si_attribute: sss;
{
 tab:= StrConcat(tab, BLANCS);
 sss:= si_a;
 printf(out,tab);
 if sss.min_rep = 0 and sss.max_rep = N_CARD then
      { printf(out,["Tlist <",FindCppType(sss),"> * ",sss.name,
        ";\n"]);
      }
 else { printf(out, [FindCppType(sss)," ",sss.name, FindSize(sss),";\n"]);
 };
 if sss.min_rep = 0 and sss.max_rep = N_CARD then
      { printf(out,[tab,"Titer <",FindCppType(sss),"> * i_",sss.name,
                    ";\n"]); };
}


/* ------------------------------------------------------
 f. retourning the "header" of a C++ "struct"
 ------------------------------------------------------*/
function string OpenStruct(owner_of_att: ow)
{
 return StrConcat(StrConcat("struct t_",ow.name)," {\n");
}
```

```
/*-------------------------------------------------------------
  fonction closing a C++ "struct". "[max_rep]" of
  "compound attribute" included if <>1 !!
 -------------------------------------------------------------*/
function string CloseStruct(owner_of_att: ow)
string: helpstr;
{
 helpstr:= "} "+ow.name;
 if ow.max_rep <> 1 then
     { helpstr:= helpstr+"["+StrItos(ow.max_rep)+"]"; };
 return  helpstr+";\n";
}

/*-------------------------------------------------------------------------------
  Recursive proc. generating a C++ "struct" for each
  compound attribute of an entity type and a struct "TBuffer"
  containing all attributs of a given ET.
 -------------------------------------------------------------------------------*/
procedure MakeStructures(owner_of_att: o, string: insertion_initiale)
  attribute: at;
  list: l_at;
  cursor: d;
  integer: t;
{
 insertion:= StrConcat(insertion, BLANCS);
 printf(out,[insertion, OpenStruct(o)]);

 l_at:= ATTRIBUTE[at]{@OWNER_ATT:[o]};
 attach d to l_at;
 while IsNoVoid(d) do {
 t:= GetType(get(d));
 switch(t) {
   case SI_ATTRIBUTE: TreatSiAttr(get(d), insertion);
   case CO_ATTRIBUTE: MakeStructures(get(d), insertion);
   otherwise: printf(out, [insertion, BLANCS, "Error!"]);
 };
 d>>;
 };

 printf(out,[insertion, CloseStruct(o)]);
 insertion:= insertion_initiale;
}


/*-------------------------------------------------------------
   Procedure generating a "Forward class declaration"
 -------------------------------------------------------------*/
procedure GenerForwardClassDecl(entity_type: et)
{
  printf(out,["\nclass c",et.name,";"]);
}


/*-------------------------------------------------------------
   F.retourning a list of an ET's SubTypes
 -------------------------------------------------------------*/
function list GiveListDeSubTypes(entity_type: entt)
  entity_type: enti;
```

```
 sub_type: subt;
 cluster: clu;
{
 return ENTITY_TYPE[enti]{ENTITY_SUB:SUB_TYPE[subt]
      {@CLU_SUB: CLUSTER[clu]{@ENTITY_CLU:[entt]}}};

}

/*-----------------------------------------------------------------
  F. returning list of all obj. attr. refering to 'en' but
  not belonging to 'en'.
 -----------------------------------------------------------------*/
function list MakeListOfForeignObjAttrTo(entity_type: en)
 data_object: dataobj;
 list: helplis, returnlist;
 si_attribute: siatt;
{
  helplis:= DATA_OBJECT[dataobj]{@SCH_DATA:[sch]
     with GetType(dataobj)=SI_ATTRIBUTE};
  for siatt in helplis do {
   if siatt.type=OBJECT_ATT and
     GetOwningEntityType(siatt)<>en then {
       if ReturnTypeOfObjAttr(siatt)=en
       then  { AddLast(returnlist,siatt); };
   };
  };
 return returnlist;
}


/*-----------------------------------------------------
  Proc. generating a "class" for each ET
 -----------------------------------------------------*/
procedure GenererClasse(entity_type: ent, list: id_attr_l)
  owner_of_att: own;
  attribute: att;
  list: AttrList;
  cursor: cur;
  integer: t, within_class;
  entity_type: supertype;
{
 printf(out,["\nclass c",ent.name]);

 /* le TE peut avoir des supertypes */
 if Length(GiveListDeSuperTypes(ent)) <> 0 then
    {
     supertype:= FindSuperType(ent);
     printf(out,[": public c", supertype.name]);
    };
 printf(out," {\n");
 printf(out,["public:\n"]);

 /* Partie qui s'occupe des attributs, qui crée des "struct" */
 own:=ent;
 AttrList:= ATTRIBUTE[att]{@OWNER_ATT:[own]};
 attach cur to AttrList;
 printf(out, [BLANCS, "struct "]);
```

```
 printf(out, ["t_c",ent.name,"_buf {\n"]);
  while IsNoVoid(cur) do {
    t:= GetType(get(cur));
    switch(t) {
      case SI_ATTRIBUTE: TreatSiAttr(get(cur), BLANCS);
      case CO_ATTRIBUTE:
         insertion:= BLANCS;
         MakeStructures(get(cur), insertion);
      otherwise: printf(out, [insertion, BLANCS, "error!"]);
    };
    cur>>;
  };

 printf(out,[BLANCS,"} "]);
 printf(out, ["c",ent.name,"_buf;"]);

 /* Les attributs next, previous and first */
 printf(out,["\nprivate:"]);
 printf(out,["\n",BLANCS,"c",ent.name,"* next;"]);
 printf(out,["\n",BLANCS,"c",ent.name,"* previous;"]);
 printf(out,["\n",BLANCS,"static c",ent.name,"* first;"]);
 /*data member & inline member fcts for classes with identifiers*/
 if Length(id_attr_l)>0 then {
  printf(out,["\n",BLANCS,"bool init0_N_attr;"]);
  printf(out,["\n",BLANCS,"bool inserted_into_list;"]);
  GenerCheckIdentifier(ent,1,id_attr_l);
  printf(out,["\n",BLANCS,"void set_init0_N_attr(bool value)"]);
  printf(out," { init0_N_attr=value; }");
  printf(out,["\n",BLANCS,"bool get_init0_N_attr() {"]);
  printf(out," return init0_N_attr; }");
  printf(out,["\n",BLANCS,"void set_inserted_into_list(bool value)"]);
  printf(out," { inserted_into_list=value; }");
  printf(out,["\n",BLANCS,"bool get_inserted_into_list()"]);
  printf(out,[" { return inserted_into_list; }"]);
 };
 /* heads of member functions*/
 printf(out,["\npublic:"]);
 /* constructeur */
 printf(out,["\n",BLANCS,"c",ent.name]);
 within_class:= 1;  /* 1="true" */
 HeadOfConstruct(ent, within_class);
 printf(out,";");
 /* destructeur */
 printf(out,["\n",BLANCS,"~c",ent.name,"();"]);
 printf(out,["\n",BLANCS,"static c",ent.name," * get_first();"]);
 printf(out,["\n",BLANCS,"c",ent.name," * get_next();"]);
 printf(out,["\n",BLANCS,"c",ent.name," * get_previous();"]);

 /* what() */
 GenerWhat(ent, within_class);

 printf(out,"\n};\n"); /* End of class!! */
}
```

```
/*-------------------------------------------------------------------
  Proc. generating a typedef "w_class" containing an
  énumération refering to each class that will be generated
  -------------------------------------------------------------------*/
procedure GenerEnumTypes(list: l_ent)
entity_type: ent;
integer: i;
cursor: c;
string: ESPACES;
{
 ESPACES:= "            ";
 printf(out,"\ntypedef enum {");
 attach c to l_ent;
 i:= 0;
 while IsNoVoid(c) do {
   i:=i+1;
   if i mod 5 = 0 then { printf(out,["\n",ESPACES]); };
   /* aller à la ligne chaque fois après 5 classes */
   ent:= get(c);
   printf(out,["w_",ent.name]);
   c>>;
   if IsNoVoid(c) then { printf(out,", "); };
 };
 printf(out,"} w_class;");
}


/*-------------------------------------------------------------------------
  F. creating a list of all the ET in a schema in a way that
  each superclass precedes its subclasses (otherwise an
 inherited class risks being generated before its 'mother' class
 -------------------------------------------------------------------------*/
function list ConstruireListeTriee()
 data_object: data;
 list: liste_triee;
 entity_type: enti;
{
 for enti in DATA_OBJECT[data] {@SCH_DATA:[sch]
         with GetType(data) = ENTITY_TYPE}
 do {
    if Length(GiveListDeSuperTypes(enti)) = 0 then
      {
       AddLast(liste_triee, enti);
       if Length(GiveListDeSubTypes(enti)) <> 0 then
         {
         SearchDepthFirst(enti, liste_triee);
         };
      };
   }
return liste_triee;
}
```

```
/*-------------------------------------------------------------------------------------
  Recursive proc. walking through the ET of a tree doing 'depth search
  first' As long as an ET has still subtypes, the proc. is called
  again.
  -------------------------------------------------------------------------------------*/
procedure SearchDepthFirst(entity_type: ent, list: list_tri)
 list : liste_sub_ent;
 cursor: cur;
 entity_type: et;
{
 liste_sub_ent:= GiveListDeSubTypes(ent);
 if Length(liste_sub_ent) <> 0 then
   {
    for et in liste_sub_ent do
       {
        AddLast(list_tri, et);
        SearchDepthFirst(et, list_tri);
       }
   };
}

/*-------------------------------------------------------------------------
 Function making a unique list of attributes that are (part of) an
 identif. Hopefully this list is always made in the same order...
 -------------------------------------------------------------------------*/
function list MakeListOfIdAttr(entity_type: ent)
 data_object: dat;
 list: idattrlist;
 group: grr, groupy;
 real_component: rc;
 integer: typeint;
 si_attribute: siattrib;
 cursor: c;
{
 dat:= ent;
 //ClearList(idattrlist);
 for groupy in GROUP[grr]{ @DATA_GR:[dat] with grr.primary or grr.secondary}
 do
 {
  for rc in GetListOfComponents(groupy) do {
   typeint:= GetType(rc);
   switch(typeint){
    case SI_ATTRIBUTE: siattrib:=rc;
                       c:= member(idattrlist, siattrib);
                       if IsVoid(c) then {AddLast(idattrlist,siattrib);};
    case CO_ATTRIBUTE: ChercheAttrLesPlusLoins(idattrlist,rc);
                       //si attr immediately added to idattrlist!

    };
  }
 }
 return idattrlist;
}
```

```
/*-------------------------------------------------------------------
  Proc generating a f. to destroy ALL the objects of
  one class. Can take any class: because of its template!
 -------------------------------------------------------------------*/
procedure GenerGarbageCollector() {
 printf(out,"\n\n//delete all objects of one class");
 printf(out,"\ntemplate <class S>");
 printf(out,"\nvoid delete_one_class(S * aptr) {");
 printf(out,["\n",BLANCS,"while(aptr) {"]);
 printf(out,["\n",BLANCS," delete aptr;"]);
 printf(out,["\n",BLANCS," aptr = S::get_first();"]);
 printf(out,["\n",BLANCS,"}"]);
 printf(out,"\n}");
}


/*-------------------------------------------------------------------
  Proc. generating a f. which sends an error message to the
  screen in case of try, throw, CATCH.
 -------------------------------------------------------------------*/
procedure GenerSendErrorMessage(){
 printf(out,["\n\n//Send error message."]);
 printf(out,["\nvoid send_error_message(int catchint) {"]);
 printf(out,["\n",BLANCS,"if (catchint==11)"]);
 printf(out,[" printf(\"ID is not repected."]);
 printf(out," Object not created!\\n\");");
 printf(out,["\n",BLANCS,"if (catchint==22) { printf(\"Obligatory"]);
 printf(out," value remained empty.\\n\");");
 printf(out,"\n                ");
 printf(out,["printf(\"Object not created!\\n\");\n }"]);
 printf(out,"\n}");
}


/*-------------------------------------------------------------------
  Proc. generating the scripts a C++ schema
  Generates in the file "out" the C++ script of "sch"
 -------------------------------------------------------------------*/
procedure GenererCPP()
 list: l; /* On va garder cette liste de TE pour pouvoir l'utiliser
          plusieurs fois sans perdre l'ordre exacte des TE. */
 list: id_attr_lst;
       /*To have a unique list of attr that are (part of) an
         identif. for a given ET*/
 entity_type: ent;
 integer: within_class;
{
 l:= ConstruireListeTriee();
 /* l sera ch x parcourue dans le même ordre */

 printf(out,[REMARK, "Fichier C++ genere par GenerCpp.exe\n\n"]);
 GestionInclude();
 GenerEnumTypes(l);

 /* Forward declaration of member classes*/
 printf(out, '\n');
 for ent in l do { GenerForwardClassDecl(ent); };
```

```
 /* Generation des classes C++, une "class" pour chaque TE.*/
 printf(out,"\n");
 for ent in 1 do {
  GenererClasse(ent,MakeListOfIdAttr(ent));
 };

 /* Generation des variables globales. */
 printf(out,["\n\n",REMARK,"global variables\n"]);
 for ent in 1 do { GenererVarGlobales(ent); };

 /* Inititilisation des "static members" ("first") */
 printf(out,["\n\n",REMARK,"initiation static members"]);
 for ent in 1 do { InitStaticMember(ent); };

 /* Generation les "member functions". La liste des TE
 parcourue ds le même ordre que ds la boucle précédente */
 printf(out,["\n"]);
 for ent in 1 do {
    printf(out,["\n",REMARK,"member functions of class c",ent.name]);
    GenerConstructeur(ent);
    GenerDestructeur(ent);
    GenerGetFirst(ent);
    GenerGetNext(ent);
    GenerGetPrevious(ent);
    within_class:= 0;
    GenerWhat(ent, within_class);
    id_attr_lst:=MakeListOfIdAttr(ent);
    if Length(id_attr_lst)>0 then
          { GenerCheckIdentifier(ent,within_class,id_attr_lst); };
    printf(out,"\n");
    };

 /* stand alone functions */
  GenerGarbageCollector();
  GenerSendErrorMessage();

 /* Generation du programme principal */
 GenerProgrPrincipal();
}

/* ---------------------------------------------------------------------------
 function returning a list of all the components of a group
  ---------------------------------------------------------------------------*/
function list GetListOfComponents(group: gr)
 real_component: rc;
 component: co;
{
 return REAL_COMPONENT[rc]{REAL_COMP: COMPONENT[co]{ @GR_COMP:[gr]}};
}
```

```
/* ------------------------------------------------------------------
  function killing all elements of a list (leaving 'ghosts')
   ------------------------------------------------------------------*/
procedure ClearList(list: listtoclear)
 cursor: killcursor;
{
 attach killcursor to listtoclear;
 while IsNoVoid(killcursor) do
 { kill(killcursor);
  killcursor>>;
 };
}

/* ----------------------------------------------------------
  procedure that prints the exact arguments of
  procedure 'check id'
   ----------------------------------------------------------*/
procedure PrintFindIdArguments(list :arglist)
 si_attribute: argument;
 integer: i;
 cursor: c;

{
 printf(out,"(");
 i:=0;
 attach c to arglist;
 while IsNoVoid(c) do {
  argument:=get(c);
  i:=i+1;
  if i mod 5 = 0 then { printf(out,["\n","        "]); };
  printf(out,[FindCppType(argument)," ",argument.name,FindSize(argument)]);
  c>>;
  if IsNoVoid(c) then { printf(out,","); };
 }
 printf(out,")");
} /* end procedure*/


/* ------------------------------------------------------------------
  Generation of function checking primary and secondary
  identifiers of each ET. precond.: there are identifiers
   ------------------------------------------------------------------*/
procedure GenerCheckIdentifier(entity_type: myent, integer: in_class,
                               list: my_id_attr_list)
 string: classname, ptrname, D_BLANCS, str;
 group: grbidon, gr;
 data_object: data;
 list: idattrlist, grlist;
 cursor: bidoncursor, gg, cc;
 si_attribute: siattrib;
 real_component: rc;
 integer: typeint;
{

 data:= myent;
 grlist:= GROUP[gr]{@DATA_GR:[data] with gr.primary or gr.secondary};

 D_BLANCS:= BLANCS + BLANCS;
```

```
classname:= "c" + myent.name;
ptrname:= "p" + myent.name;
if in_class=0 then { printf(out,"\n"); };
printf(out,["\n"]);
if in_class=1 then { printf(out,BLANCS); };
printf(out,[classname, " * "]);
if in_class=0 then { printf(out,[classname,"::"]); };
printf(out,"find_id");

PrintFindIdArguments(my_id_attr_list);
if in_class=1 then {
    printf(out,";");
    return;  /*end of this function!*/
};
printf(out,"\n{");
printf(out,["\n ",classname," * ",ptrname,";"]);
printf(out,["\n ",ptrname," = ",classname,"::get_first();"]);
printf(out,["\n while(",ptrname,"){"]);
printf(out,["\n", BLANCS, "if ("]);


/*generate code that checks each group of identifiers*/
attach gg to grlist;
while IsNoVoid(gg) do {
 printf(out,["\n",D_BLANCS,"("]);
  ClearList(idattrlist);
  grbidon:=get(gg); /*grbidon either primary or sec id */
  for rc in GetListOfComponents(grbidon) do {
   typeint:= GetType(rc);
   switch (typeint) {
    case SI_ATTRIBUTE: siattrib:=rc;
                bidoncursor:= member(idattrlist, siattrib);
                if IsVoid(bidoncursor) then {
                  AddLast(idattrlist,siattrib);
                 };
    case CO_ATTRIBUTE: ChercheAttrLesPlusLoins(idattrlist, rc);
   }; /*end switch*/
  }; /*end for*/
  attach cc to idattrlist;
  while IsNoVoid(cc) do {
   siattrib:= get(cc);
   if FindCppType(siattrib)="char" then {
     printf(out, ["strcmp(", ptrname, "->",
              MakeStringOfAttrHierarchy(siattrib,"",1, 105),
            ".",siattrib.name,",",siattrib.name,") == 0"]);
     }
   else {
     printf(out, [ptrname,"->",
            MakeStringOfAttrHierarchy(siattrib,"",1, 105),
            ".",siattrib.name," == ",siattrib.name]);
   }; /* end if*/
   cc>>;
   if IsNoVoid(cc) then { printf(out,[" &&\n",D_BLANCS," "]); };
  };
printf(out,[")",D_BLANCS]);
gg>>;
if IsNoVoid(gg) then { printf(out,["\n",D_BLANCS,"||"]); }
}; /*end while*/
```

```
 printf(out,["\n",D_BLANCS,") { return (",ptrname,
        "); } /*end if statement*/"]);
 printf(out,["\n ", ptrname," = ",ptrname,"->get_next();"]);
 printf(out,["\n } /*end while*/"]);
 printf(out,["\n return(NULL);"]);
 printf(out,"\n}");
} /*end GenerCheckIdentifier*/



/* ---------------------------------------------------------------------
  Recursive f. generating a list with the most profound attr.
  i.e. simple attr., in the hierarchy of decomposable attr.
   ---------------------------------------------------------------------*/
procedure ChercheAttrLesPlusLoins(list: l_simple_att, owner_of_att: ow)
 list: list_bidon;
 attribute: attrib;
 cursor: dd;
 integer: t;
{
 list_bidon:= ATTRIBUTE[attrib]{@OWNER_ATT:[ow]};
 attach dd to list_bidon;
 while IsNoVoid(dd) do {
 t:= GetType(get(dd));
 switch(t) {
  case SI_ATTRIBUTE: AddLast(l_simple_att,get(dd));
  case CO_ATTRIBUTE: ChercheAttrLesPlusLoins(l_simple_att, get(dd));
  };
 dd>>;
 }; /*end while*/
}


/* ---------------------------------------------------------------------------------------
  recursive f. building C++ code to access a 0-N attr, no matter
  how deep the access has to be made. If generbuffername=1 then the
  name of the buffer of the classeis added to the result string
  ---------------------------------------------------------------------------------------*/
function string MakeStringOfAttrHierarchy(attribute: oa, string: sst,
                    integer: generbuffername, integer: integ)
 list: lisst;
 owner_of_att: owner;
 entity_type: ETbidon;
 attribute: attr_bidon;
 string: strbidon;
{
 lisst:= OWNER_OF_ATT[owner]{OWNER_ATT:[oa]}; /*list of 1 element*/
 owner:= GetFirst(lisst);
 if GetType(owner)<>ENTITY_TYPE then {
  attr_bidon:= owner;
  if 1<attr_bidon.max_rep and attr_bidon.max_rep<N_CARD then {
   strbidon:= attr_bidon.name + "["+ StrSetChar("x",0,AscToChar(integ))
          + "]";
   integ:= integ + 1;
   }
  else { strbidon:= attr_bidon.name; };
  sst:= MakeStringOfAttrHierarchy(attr_bidon, sst, generbuffername, integ)
     + "." + strbidon;
```

```
  return sst;
  }
 if generbuffername = 1 then {
  ETbidon:= owner;
  sst:= "c" + ETbidon.name + "_buf" + sst;
  };
 return sst;
}

/*-----------------------------------------------------------------------
 F. testing whether their are attributes of cardinality [0-N]
 returns 1 or 0
 -----------------------------------------------------------------------*/
function integer TestIf0_NAttrAndRemoveOthers(list: llst)
 cursor: d;
 si_attribute: ssiat;

{
 attach d to llst;  /*lst contains only si_attr*/
 while IsNoVoid(d) do {
 ssiat:=get(d);
 if ssiat.min_rep<>0 or ssiat.max_rep<>N_CARD then { kill(d);};
 d>>;
 }; /*only O-N attr are kept in lst*/
 if Length(llst)=0 then { return 0; } else { return 1; };
}

/* -----------------------------------------------------------------------
 recursive proc generating 'for (...' loops for the con-
 structor and the destructor for each multiv attr 0-X, X<N
 -----------------------------------------------------------------------*/
procedure GenerForLoopsMultAttr(attribute: oatt, integer: integ,
                                integer: open_loop)
 list: li;
 owner_of_att: owne;
 attribute: atr;
{
 li:= OWNER_OF_ATT[owne]{OWNER_ATT:[oatt]}; /*list of 1 element*/
 owne:=GetFirst(li);
 if GetType(owne) <> ENTITY_TYPE then {
  atr:= owne; /* because owne.max_rep doesn't exist*/
  if 1<atr.max_rep and atr.max_rep<N_CARD then  {
    switch (open_loop) {
     case 1 :  printf(out,["\n",BLANCS,"for (", AscToChar(integ),
               "=0; ",AscToChar(integ), "<",StrItos(atr.max_rep),
               "; ",AscToChar(integ),"++) {"]);
               integ:= integ + 1;
     case 0 :  printf(out,[BLANCS,"};"]);
    }; /* end switch */
  };
 GenerForLoopsMultAttr(owne, integ, open_loop);
 };
}
```

```
/* ---------------------------------------------------------------------------
  proc counting the number of integers needed to manage loops
  "for (i:=1;...)", for the treatment of the given list of attr.
  ---------------------------------------------------------------------------*/
function integer CountNumOfIntegersNeededFor(list: lst_of_att)
  attribute: attrib;
  integer: respro, resprov;
{
  resprov:= 0;
  for attrib in lst_of_att do {
    respro:= CountThisBranch(attrib, 0);
    if respro > resprov then { resprov:=respro; };
  }
  return resprov;
}

/* ---------------------------------------------------------------------------
  recursive f. counting the number of multiv compound attr
  on its way from a simple attr to its root (a compound attr)
  ---------------------------------------------------------------------------*/
function integer CountThisBranch(attribute: my_attr,
                                 integer: sumofthisbranch)
  list: branchlist;
  owner_of_att: ow;
  attribute: atrbidon;
{
  branchlist:=OWNER_OF_ATT[ow]{OWNER_ATT:[my_attr]}; /*1element in list*/
  ow:=GetFirst(branchlist);
  if GetType(ow) <> ENTITY_TYPE then {
     atrbidon:= ow; /* ow.maxrep doesn't exist */
     if 1<atrbidon.max_rep and atrbidon.max_rep<N_CARD then {
       sumofthisbranch:= CountThisBranch(atrbidon, sumofthisbranch) + 1; }
     return sumofthisbranch;
  };
}

/* ---------------------------------------------------------------------------
  Proc generating a call, made by constructors, to "FindId(...)"
  precond: at least 1 identifier.
  ---------------------------------------------------------------------------*/
procedure GenerCallToProcedureFindId(entity_type: ent)
  list: argumentlst;
  cursor: cur;
  attribute: at;
  integer: i;
{
  argumentlst:= MakeListOfIdAttr(ent);
  printf(out,["\n",BLANCS,"//check identifier(s)"]);
  printf(out,["\n",BLANCS,"if (find_id("]);
  i:=0;
  attach cur to argumentlst;
  while IsNoVoid(cur) do {
    at:=get(cur);
    i:=i+1;
    if i mod 4 = 0 then { printf(out,["\n",BLANCS,BLANCS]); };
    printf(out,["c",ent.name,"_bu",
          MakeStringOfAttrHierarchy(at,"",0,105), ".",at.name]);
    cur>>;
```

```
  if IsNoVoid(cur) then { printf(out,", "); };
 };
 printf(out,["))\n",BLANCS,BLANCS,"{ delete(this); throw 11; }"]);
}


/* ---------------------------------------------------------------------------
  proc printing all the Integer Variables needed for the constructor,
  destructor, etc.
  ---------------------------------------------------------------------------*/
procedure PrintIntegerVariablesNeeded(integer: turnmetoascii,
                                      integer: numofint)
 integer: countme;
{
 if numofint>0 then { printf(out, ["\n", BLANCS, "int "]); }
 else { return; }
 countme:=0;
 while countme < numofint do {
   printf(out,AscToChar(turnmetoascii));
   countme:= countme + 1;
   if countme < numofint then { printf(out,", "); };
   turnmetoascii:= turnmetoascii + 1;
   };
 printf(out,";");
}


/* ---------------------------------------------------------------------------
  proc generating the constructor of a class (head + definition)
  ---------------------------------------------------------------------------*/
procedure GenerConstructeur(entity_type: entti)
  entity_type: supent;
  integer: within_class, ascii_int,numofint, hlpnumofint;
  list: lst, lstforeignobjattr,lst_of_id_attr;
  attribute: at;
  si_attribute: siat;
  cursor: d, al;
  string: str;
{
 lst_of_id_attr:=MakeListOfIdAttr(entti);
 ascii_int:=105;
 numofint:= 0;
 within_class:= 0;
 printf(out,["\nc",entti.name,"::c",entti.name]);
 HeadOfConstruct(entti, within_class);
 printf(out,"\n{");

 /*after two following calls, lst contains simple, 0-N attr*/
 ChercheAttrLesPlusLoins(lst, entti);
 if TestIf0_NAttrAndRemoveOthers(lst) then {
      numofint:= CountNumOfIntegersNeededFor(lst); };

 /*preparations for object attributes*/
 lstforeignobjattr:=MakeListOfForeignObjAttrTo(entti);
 hlpnumofint:= CountNumOfIntegersNeededFor(lstforeignobjattr);

 if hlpnumofint > numofint then { numofint:= hlpnumofint; };
 PrintIntegerVariablesNeeded(ascii_int, numofint);
```

```
 if Length(lst_of_id_attr)>0 then {
  printf(out,["\n",BLANCS,"set_init0_N_attr(false);"]);
  printf(out,["\n",BLANCS,"set_inserted_into_list(false);"]);
  GenerCallToProcedureFindId(entti);
 };

 printf(out,["\n",BLANCS,"//pass values to data members"]);
 printf(out,["\n",BLANCS,"memcpy(&"]);
 printf(out,['c',entti.name,"_buf, &"]);
 printf(out,['c',entti.name,"_bu, sizeof("]);
 printf(out,['c',entti.name,"_bu));"]);

 printf(out,["\n",BLANCS,"//add object to list"]);
 printf(out,["\n",BLANCS,"next = NULL;"]);
 printf(out,["\n",BLANCS,"previous = "]);
 printf(out,['c',entti.name,"_former;"]);
 printf(out,["\n",BLANCS,
        "if (get_previous()) {get_previous()->next = this;}"]);
 printf(out, " else {first = this;};");
 printf(out,[BLANCS,"\n"]);
 printf(out,[BLANCS,'c',entti.name,"_former = this;"]);
 if Length(lst_of_id_attr)>0 then {
   printf(out,["\n",BLANCS,"set_inserted_into_list(true);"]);
 };

/*Generate code to initialize all the 0-N attr lists + iterators*/
 attach al to lst;
 while IsNoVoid(al) do {
  siat:= get(al);
  printf(out, ["\n\n",BLANCS,"// Initialise lists for multivalued attr"]);
  GenerForLoopsMultAttr(siat, ascii_int, 1); /*1= open loops! */
  /*initialization itself*/
  str:= MakeStringOfAttrHierarchy(siat, str,1, ascii_int) + "."
       + siat.name;
  printf(out,["\n", BLANCS,str,"="]);
  printf(out,["\n",BLANCS, BLANCS, "new Tlist<", FindCppType(siat),">"]);
  if siat.type=OBJECT_ATT then
      { printf(out,["(TLIST_NOT_REMOVE);"]); }
  else { printf(out,["(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);"]); };
  str:="";
  str:= MakeStringOfAttrHierarchy(siat, str,1, ascii_int) + ".i_"
       + siat.name;
  printf(out, ["\n",BLANCS,str,"="]);
  printf(out, ["\n",BLANCS, BLANCS, "new Titer<", FindCppType(siat),">"]);
  str:="";
  printf(out,["(", MakeStringOfAttrHierarchy(siat,str,1,ascii_int),
           ".",siat.name,");\n"]);
  GenerForLoopsMultAttr(siat, ascii_int, 0); /*0= close loops! */
  al>>;
 }; /*end while-statement*/
 if Length(lst_of_id_attr)>0 then {
  printf(out,["\n",BLANCS,"set_init0_N_attr(true);"]);
 };
 printf(out,"\n}; //end constructor");
}
```

```
/* ------------------------------------------------------------------------
  proc. generating the parametre of a classe's constructor
  The Boolean in_class makes clear whether we are dealing with
  the head of a constuctor within a class definition or ouside
  of it.
------------------------------------------------------------------------*/
procedure HeadOfConstruct(entity_type: entt, integer: in_class)
 integer: compteur;
 string: ESPACIO;
 entity_type: supertype;
{
 printf(out,["(t_c",entt.name,"_buf &c",entt.name,"_bu"]);
 if Length(GiveListDeSuperTypes(entt)) <> 0 then
   {
    printf(out,",\n");
    if in_class=0 then
     { ESPACIO:=StrBuild( 5 + (2*StrLength(entt.name)) ); }
    else
     { ESPACIO:=StrBuild(2 + StrLength(entt.name)) + BLANCS; }

    ArgumDesSuperClasses(entt, ESPACIO);

    if in_class = 0 then
     {
      ESPACIO:= StrBuild(3 + StrLength(entt.name));
      supertype:= FindSuperType(entt);
      printf(out,["):\n",ESPACIO,"c",supertype.name,"("]);
      compteur:= 0; /*to do "\n" after a certain nbr of arguments*/
      PassArgToBaseConstructor(supertype, ESPACIO, compteur);
     }
   };
 printf(out,')');
}

/* ------------------------------------------------------------------------
  procédure qui s'occupe dans le paramètre d'un constructeur des
  arguments concernant les superclasses d'une classe. Elle se
  déplace récursivement vers la classe souche.
  Précondition: la superclasse de ent existe.
  ------------------------------------------------------------------------*/
procedure ArgumDesSuperClasses(entity_type: ent, string:SPACE)
 entity_type: sup_ent;
{
 sup_ent:= FindSuperType(ent);
 printf(out,[SPACE,"t_c",sup_ent.name,"_buf"]);
 printf(out,[" c",sup_ent.name,"_bu"]);
 if Length(GiveListDeSuperTypes(sup_ent)) <> 0 then
  {
   printf(out, ",\n");
   ArgumDesSuperClasses(sup_ent, SPACE);
  };
}
```

```
/* ------------------------------------------------------------------------
  Recursive proc generating the arguments of a constructor
  of a superclasse, which is called by the constructor of the
  subclass.
  ------------------------------------------------------------------------*/
procedure PassArgToBaseConstructor(entity_type: ent,
                    string: SPACE, integer: i)
{
 printf(out,["c",ent.name,"_bu"]);
 if Length(GiveListDeSuperTypes(ent))<>0 then
    {
     printf(out,", ");
     i:= i + 1;
     if i mod 3 = 0 then /* On va aller à la ligne */
       {
        printf(out,['\n',SPACE]);
       };
     PassArgToBaseConstructor(FindSuperType(ent),SPACE, i);
    };
}

/* ------------------------------------------------------------------------
  proc printing a ptr variable of type 'atype' if at least 1
  of the elements in 'alst', which contains simple attr, is of type
 'atyp'. The name of the variable is 'nameofvar'
------------------------------------------------------------------------*/
procedure PrintPtrVariableOfType(string: atype,string: nameofvar,list: alst)
 si_attribute : attri;
{
 for attri in alst do {
   if FindCppType(attri)=atype then {
      printf(out,["\n",BLANCS,atype," * ",nameofvar,";"]);
      return;
  };
 };
}


/* ------------------------------------------------
  proc. generating the destructor of a class
------------------------------------------------*/
procedure GenerDestructeur(entity_type: entti)
 string: D_BLANCS, T_BLANCS, pointerstr, str;
 list: lstt, lstforeignobjattr,lst_of_id_attr;
 cursor: curs;
 si_attribute: siat, objat;
 integer: ascii_int, numofint,hlpnumofint, printyn;
 entity_type: owningET;
{
 lst_of_id_attr:=MakeListOfIdAttr(entti);
 ascii_int:=105;
 numofint:=0;
 D_BLANCS:= StrConcat(BLANCS,BLANCS);
 T_BLANCS:= StrConcat(BLANCS,D_BLANCS);
 printf(out,["\n\nc",entti.name,"::~c",entti.name,"() {"]);

 /* prep. for attr 0-N. structure: cfr comments for GenerConstructeur!*/
 ChercheAttrLesPlusLoins(lstt, entti);
```

```
if TestIf0_NAttrAndRemoveOthers(lstt) then
  { numofint:= CountNumOfIntegersNeededFor(lstt); };

/*prep for obj attr. with pointers pointing to this entity type*/
lstforeignobjattr:=MakeListOfForeignObjAttrTo(entti);
hlpnumofint:= CountNumOfIntegersNeededFor(lstforeignobjattr);
if hlpnumofint>numofint then { numofint:=hlpnumofint; };

/* print variables */
PrintIntegerVariablesNeeded(ascii_int, numofint);

/*print other variables if necessary*/
if Length(lstt)>0 then {
  //PrintPtrVariableOfType("bool","bp",lstt);
  //PrintPtrVariableOfType("int","ip",lstt);
  //PrintPtrVariableOfType("char","cp",lstt);
  //PrintPtrVariableOfType("float","fp",lstt);
  //PrintPtrVariableOfType("double","dp",lstt);
};

/*Print pointer to entti if it has at lst 1 foreign obj attr 0-N*/
printyn:=0;
for objat in lstforeignobjattr do {
 if objat.max_rep=N_CARD then { printyn:=1; };
};
if printyn=1 then {
 printf(out,["\n",BLANCS,"c",entti.name," * op;"]);
}

/*Print pointer to classes containing a foreig attr pointing to entti */
for objat in lstforeignobjattr do {
 owningET:= GetOwningEntityType(objat);
 printf(out,["\n",BLANCS,"c",owningET.name," * ",owningET.name,"po;"]);
};

/* code to deallocation all lists and iterators if any*/
attach curs to lstt;
if Length(lstt)<> 0 then {
 printf(out,["\n",BLANCS,"//Destroy lists and reiterators"]);
};
if Length(lst_of_id_attr)>0 then {
 printf(out,["\n",BLANCS,"if (get_init0_N_attr()) {"]);
};
while IsNoVoid(curs) do
{
 siat:= get(curs);
 str:="";
 GenerForLoopsMultAttr(siat, ascii_int, 1); /*1= open loops! */
 printf(out,["\n",BLANCS,"delete "]);
 str:= MakeStringOfAttrHierarchy(siat,str,1,ascii_int) + ".i_" + siat.name;
 printf(out,[str,";"]);
 str:="";
 printf(out,["\n",BLANCS,"delete "]);
 str:= MakeStringOfAttrHierarchy(siat,str,1,ascii_int) + "." + siat.name;
 printf(out,[str,";"]);
 GenerForLoopsMultAttr(siat, ascii_int, 0); /*0= close loops! */
 curs>>;
}
```

```
 if Length(lst_of_id_attr)>0 then {
    printf(out,["\n",BLANCS,"} //end if get0_N_..."]);
 };

 printf(out,["\n",BLANCS,"//Take object out of list."]);
 if Length(lst_of_id_attr)>0 then {
  printf(out,["\n",BLANCS,"if (get_inserted_into_list()) {"]);
 };
 printf(out,["\n",BLANCS,"if (get_next()) {"]);
 printf(out,["\n",D_BLANCS,"if (get_previous()) {"]);
 printf(out,["\n",T_BLANCS,"get_previous()->next = next;"]);
 printf(out,["\n",T_BLANCS,"get_next()->previous = previous;"]);
 printf(out,["\n",D_BLANCS,"}"]);
 printf(out,["\n",D_BLANCS,"else {"]);
 printf(out,["\n",T_BLANCS,"get_next()->previous = NULL;"]);
 printf(out,["\n",T_BLANCS,"first = next;"]);
 printf(out,["\n",D_BLANCS,"};"]);
 printf(out,["\n",BLANCS,"}"]);
 printf(out,["\n",BLANCS,"else"]);
 printf(out,["\n",D_BLANCS,"if (get_previous()) {"]);
 printf(out,["\n",T_BLANCS,"get_previous()->next = NULL;"]);
 printf(out,["\n",T_BLANCS]);
 printf(out,['c',entti.name,"_former = previous;"]);
 printf(out,["\n",D_BLANCS,"}"]);
 printf(out,["\n",D_BLANCS,"else {"]);
 printf(out,["\n",T_BLANCS]);
 printf(out,['c',entti.name,"_former = NULL;"]);
 printf(out,["\n",T_BLANCS,"first = NULL;"]);
 printf(out,["\n",BLANCS,"} //end if"]);
 if Length(lst_of_id_attr)>0 then {
    printf(out,["\n",BLANCS,"} //end if get_inserted_..."]);
 };

 if Length(lstforeignobjattr)>0 then {
   printf(out,["\n",BLANCS]);
   printf(out,["//Management of foreign obj attr with pointers to c"]);
   printf(out,entti.name);
 }/*end if*/

for objat in lstforeignobjattr do {
 GenerForLoopsMultAttr(objat, ascii_int, 1);
 owningET:= GetOwningEntityType(objat);
 printf(out,["\n",BLANCS,owningET.name,"po=c",owningET.name,
          "::get_first();"]);
 printf(out,["\n",BLANCS,"while (",owningET.name,"po) {"]);

 switch (objat.max_rep) {
  case 1:
   printf(out,["\n",D_BLANCS,"if (",owningET.name,"po->c",
            owningET.name,"_buf",
            MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".",objat.name," == this) {"]);
   printf(out,["\n",T_BLANCS,owningET.name,"po->c",
            owningET.name,"_buf",
            MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".",objat.name," = NULL;"]);
   printf(out,["\n",D_BLANCS,"}"]);
```

```
  case N_CARD:
    printf(out,["\n",D_BLANCS,"if ("]);
    printf(out,["\n",D_BLANCS,"(",owningET.name,"po->c",owningET.name,
            "_buf",MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".",objat.name,"->Length() != 0) &&"]);
    printf(out,["\n",D_BLANCS," ",owningET.name,"po->c",owningET.name,
            "_buf",MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".",objat.name,"->Member(this))"]);
    printf(out,["\n",D_BLANCS,"{"]);
    printf(out,["\n",D_BLANCS,"for (op=",owningET.name,"po->c",
            owningET.name,"_buf",
            MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".i_",objat.name,"->Init();"]);
    printf(out,["\n",T_BLANCS,"op;"]);
    printf(out,["\n",T_BLANCS,"op=",owningET.name,"po->c",
            owningET.name,"_buf",
            MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
            ".i_",objat.name,"->More() )"]);
    printf(out,["\n",T_BLANCS,"{"]);
    printf(out,["\n",T_BLANCS," if(op == this) {"]);
    printf(out,["\n",T_BLANCS," delete ",owningET.name,"po->c",
             owningET.name,"_buf",
             MakeStringOfAttrHierarchy(objat,"",0,ascii_int),
             ".i_",objat.name,"->Whole();"]);
    printf(out,["\n",T_BLANCS,"} ",REMARK,"end if"]);
    printf(out,["\n",D_BLANCS,"} ",REMARK,"end for"]);
    printf(out,["\n",D_BLANCS,"} ",REMARK,"end if"]);
  otherwise: printf(out,"error! max card must be 1 or N");
  }; /*end switch*/

  printf(out,["\n",D_BLANCS,owningET.name,"po=",
          owningET.name,"po->get_next();"]);
  printf(out,["\n",BLANCS,"} ",REMARK,"end while\n"]);
  GenerForLoopsMultAttr(objat, ascii_int, 0);
 };/*end for*/
 printf(out,"\n}//end of destructor");
} /* fin GenerDestructeur */


/* ------------------------------------------------
  proc. generating the method Get_First() of a class
------------------------------------------------*/
procedure GenerGetFirst(entity_type: entti)
{
 printf(out,["\n\nc",entti.name," * c",entti.name,"::get_first()"]);
 printf(out,["\n",BLANCS,"{ return first; }"]);
}

/* ------------------------------------------------------------------
  proc generating the method Get_Next() of a class
------------------------------------------------------------------*/
procedure GenerGetNext(entity_type: entti)
{
 printf(out,["\n\nc",entti.name," * c",entti.name,"::get_next()"]);
 printf(out,["\n",BLANCS,"{ return next; }"]);
}
```

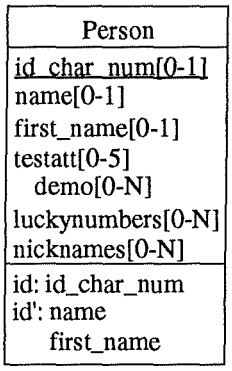
```
/* ----------------------------------------------------------
  procédure générant la méthode Get_Previous() d'une classe
----------------------------------------------------------*/
procedure GenerGetPrevious(entity_type: entti)
{
 printf(out,["\n\nc",entti.name," * c",entti.name,"::get_previous()"]);
 printf(out,["\n",BLANCS,"{ return previous; }"]);
}

/* ----------------------------------------------------------
  proc generating the method What() of a class
----------------------------------------------------------*/
procedure GenerWhat(entity_type: entti, integer: in_class)
  list: l;
  cursor: c;
  entity_type: e;
  sub_type: sub;
  cluster: clu;
  integer: pure_virtual;  /*Boolean*/
{
 /* voire si what is pure_virtual */
 pure_virtual:= 0;
 l:=ENTITY_TYPE[e]{ENTITY_SUB:SUB_TYPE[sub]{@CLU_SUB:CLUSTER[clu]
     {@ENTITY_CLU:[entti]}}};  /* liste des ss_types de entti */
 attach c to l;
 if IsNoVoid(c) then
   {
    clu:=_GetFirst(ENTITY_CLU,entti); /* le clu du premier ss_type */
    if (IsNoVoid(clu.disjoint)) and (IsNoVoid(clu.total)) then
       { pure_virtual:=1; }
    else
      { if (IsNoVoid(clu.total)) and (IsVoid(clu.disjoint)) then {
          if in_class = 1 then /* to avoid having 2x the same message */
            { print("Warning: in C++, subclasses are always disjoint.\n");
              print(["Consequently, the subclasses of c",entti.name,
              " are not only total,\nbut even form a partition!\n"]); };
        };
      };
   };
 if in_class = 1 then
  {
  printf(out,["\n",BLANCS,"virtual w_class what()"]);
  if pure_virtual = 1 then { printf(out," = 0"); };
  printf(out,';');
 }; /* end if statement */


 if (in_class = 0) and (pure_virtual = 0) then
   {
    printf(out,["\n\nw_class c",entti.name,"::what()"]);
    printf(out,["\n",BLANCS,"{ return w_",entti.name,"; }"]);
   };
}
```

```
/* ------------------------------------------------------------
  proc. generating the code that initialises the "static members"
  ------------------------------------------------------------*/
procedure InitStaticMember(entity_type: enti)
{
 printf(out,["\nc",enti.name," * c",enti.name,"::first = NULL;"]);
}


/* ------------------------------------------------------------------------
  Proc. preparing the function "main()" of the C++ programme
  ------------------------------------------------------------------------*/
procedure GenerProgrPrincipal()
{
 printf(out,"\n\nint main()");
 printf(out,"\n{");
 printf(out,["\n",BLANCS,"try {"]);
 printf(out,["\n",BLANCS,"}"]);
 printf(out,["\n",BLANCS,"catch(int i) {"]);
 printf(out,["\n",BLANCS,BLANCS,"send_error_message(i);"]);
 printf(out,["\n",BLANCS,"}"]);
 printf(out,['\n',BLANCS,"return 0;"]);
 printf(out,"\n}");
}


/* -------------------------
  Main programme
-------------------------*/
begin
      if not(OuvreFichier())  then{
            sch:=GetCurrentSchema();
            if not(IsVoid(sch)) then {
                  SetPrintList("","","");
                  GenererCPP();
                  CloseFile(out);
                  print("\nOK!\n");
            };
      };

end
```

II. The Generated C++ Code. Example 1

This appendix comprises a few examples of object oriented physical schemata and their corresponding C++ code generated by the C++ generator. The code is delivered on the diskette in the back cover of this thesis, together with all the C++ files necessary to take care of lists corresponding to multivalued attributes with cardinality 0-N.

II.1. Physcial Schema



Types of the attributes:
id_char_num: numeric
name: char, 14
first_name: char, 29
testatt: compound
demo: numeric
luckynumbers: numeric
nicknames: char

II.2. The C++ Code Corresponding to the Schema

Remark: The code contained by the main programme and by the function *list_pers( )* underneath is not generated by the C++ generator, but has been added to make an executable programme the output of which is represented as well.

The programme tries to create five instances of persons, each of them having a list of lucky numbers. Two of them are not constructed because of a double identifier detected by the constructor. The function *List_pers( )* walks through the list of persons created by the main programme and prints their names, first names and lists of lucky numbers on screen.

C++ code:

```
//Fichier C++ genere par GenerCpp.exe

#include <string.h>
#include <stdio.h>
#include <iostream.h>
#include "..\lists\tlist.h"

typedef enum {w_Person} w_class;

class cPerson;

class cPerson {
public:
  struct t_cPerson_buf {
    int id_char_num;
    char name[15];
    char first_name[30];
    struct t_testatt {
      Tlist <char> * demo;
      Titer <char> * i_demo;
    } testatt[5];
    Tlist <int> * luckynumbers;
    Titer <int> * i_luckynumbers;
    Tlist <char> * nicknames;
    Titer <char> * i_nicknames;
  } cPerson_buf;
private:
  cPerson* next;
  cPerson* previous;
  static cPerson* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cPerson * find_id(int id_char_num,char name[15],char first_name[30]);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
public:
  cPerson(t_cPerson_buf &cPerson_bu);
  ~cPerson();
  static cPerson * get_first();
  cPerson * get_next();
```

```
  cPerson * get_previous();
  virtual w_class what();
 };

 //global variables
 cPerson * cPerson_former = NULL;

 //initiation static members
 cPerson * cPerson::first = NULL;

 //member functions of class cPerson
 cPerson::cPerson(t_cPerson_buf &cPerson_bu)
 {
  int i;
  set_init0_N_attr(false);
  set_inserted_into_list(false);
  //check identifier(s)
  if (find_id(cPerson_bu.id_char_num, cPerson_bu.name, cPerson_bu.first_name))
    { delete(this); throw 11; }
  //pass values to data members
  memcpy(&cPerson_buf, &cPerson_bu, sizeof(cPerson_bu));
  //add object to list
  next = NULL;
  previous = cPerson_former;
  if (get_previous()) {get_previous()->next = this; } else {first = this; };
  cPerson_former = this;
  set_inserted_into_list(true);

  // Initialise lists for multivalued attr
  for (i=0;i<5; i++) {
  cPerson_buf.testatt[i].demo=
    new Tlist<char>(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);
  cPerson_buf.testatt[i].i_demo=
    new Titer<char>(cPerson_buf.testatt[i].demo);
  };

  // Initialise lists for multivalued attr
  cPerson_buf.luckynumbers=
    new Tlist<int>(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);
  cPerson_buf.i_luckynumbers=
    new Titer<int>(cPerson_buf.luckynumbers);


  // Initialise lists for multivalued attr
  cPerson_buf.nicknames=
    new Tlist<char>(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);
  cPerson_buf.i_nicknames=
    new Titer<char>(cPerson_buf.nicknames);
  set_init0_N_attr(true);
 }; //end constructor

cPerson::~cPerson() {
  int i;
  char * cp;
  int * ip;
  int * fp;
```

```
 //Destroy lists and reiterators
 if (get_init0_N_attr()) {
 for (i=0; i<5; i++) {
  delete cPerson_buf.testatt[i].demo;
  delete cPerson_buf.testatt[i].i_demo;
 };

  delete cPerson_buf.luckynumbers;
  delete cPerson_buf.i_luckynumbers;
  delete cPerson_buf.nicknames;
  delete cPerson_buf.i_nicknames;
 } //end if get0_N_...

 //Take object out of list.
 if (get_inserted_into_list()) {
 if (get_next()) {
  if (get_previous()) {
    get_previous()->next = next;
    get_next()->previous = previous;
   }
   else {
    get_next()->previous = NULL;
    first = next;
   };
 }
 else
  if (get_previous()) {
    get_previous()->next = NULL;
    cPerson_former = previous;
   }
   else {
    cPerson_former = NULL;
    first = NULL;
 } //end if
 } //end if get_inserted_...
}//end of destructor

cPerson * cPerson::get_first()
 { return first; }

cPerson * cPerson::get_next()
 { return next; }

cPerson * cPerson::get_previous()
 { return previous; }

w_class cPerson::what()
 { return w_Person; }
```

```
cPerson * cPerson::find_id(int id_char_num,char name[15],char first_name[30])
{
 cPerson * pPerson;
 pPerson = cPerson::get_first();
 while(pPerson){
  if (
    (pPerson->cPerson_buf.id_char_num == id_char_num)
    ||
    (strcmp(pPerson->cPerson_buf.name,name) == 0 &&
     strcmp(pPerson->cPerson_buf.first_name,first_name) == 0)
   ) { return (pPerson); } /*end if statement*/
 pPerson = pPerson->get_next();
 } /*end while*/
 return(NULL);
}


//delete all objects of one class
template <class S>
void delete_one_class(S * aptr) {
  while(aptr) {
   delete aptr;
   printf("\nDelete");
   aptr = S::get_first();
  }
}

//Send error message.
void send_error_message(int catchint) {
  if (catchint==11) printf("ID is not repected. Object not created!\n");
  if (catchint==22) { printf("Obligatory value remained empty.");
                      printf("Object not created!\n");
                      };
}

int main(){
 void list_pers();
 int i;

 cPerson::t_cPerson_buf pers1 = {100, "Simpson", "Tom"};
 try {
   cPerson * first_pers = new cPerson(pers1);
   for (i=1; i<3; i++) {
    first_pers->cPerson_buf.luckynumbers->AddLast(new int (i));
   }
 }
 catch(int i) {
    send_error_message(i);
 }


 cPerson::t_cPerson_buf pers2 = {100, "Garver", "Tessa"};
 try {
  cPerson * sec_pers = new cPerson(pers2);
  for (i=4; i<7; i++) {
   sec_pers->cPerson_buf.luckynumbers->AddLast(new int (i));
  }
 }
```

```
catch(int i) {
   send_error_message(i);
}


cPerson::t_cPerson_buf pers3 = {144, "Merckx", "Axel"};
try {
 cPerson * third_pers = new cPerson(pers3);
 for (i=7; i<9; i++) {
   third_pers->cPerson_buf.luckynumbers->AddLast(new int (i));
 }
}
catch(int i) {
   send_error_message(i);
}


cPerson::t_cPerson_buf pers4 = {384, "Merckx", "Axel"};
try {
 cPerson * fourth_pers = new cPerson(pers4);
 for (i=3; i<9; i++) {
  fourth_pers->cPerson_buf.luckynumbers->AddLast(new int (i));
 }
}
catch(int i) {
   send_error_message(i);
}


cPerson::t_cPerson_buf pers5 = {385, "Gevers", "Gert"};
try {
 cPerson * fifth_pers = new cPerson(pers5);
 for (i=2; i<5; i++) {
   fifth_pers->cPerson_buf.luckynumbers->AddLast(new int (i));
 }
}
catch(int i) {
   send_error_message(i);
}


cout <<"tester cPerson: "<<endl;
list_pers(); printf("\n");

delete_one_class(cPerson::get_first());
list_pers(); printf("\n");

cout << "Press any key and then ENTER!";
int pf;
cin >> pf;

return 0;
}
```

```
//function to print list
void list_pers() {
 int * ip;
 char * cp;
 cPerson * pe;
 pe = cPerson::get_first();
 while (pe) { //the following if-test is only necessary for a base class!
  if (pe->what() == w_Person) {
          printf("%s\t\t\t%s\n",pe->cPerson_buf.name, pe->cPerson_buf.first_name);

   /*Print list 'luckynumebers'*/
   for (ip=pe->cPerson_buf.i_luckynumbers->Init(); ip;
       ip=pe->cPerson_buf.i_luckynumbers->More()) {
    printf("%d\n", *ip);
   };

   /*Print list 'nicknames'*/
   for (cp=pe->cPerson_buf.i_nicknames->Init(); cp;
       cp=pe->cPerson_buf.i_nicknames->More()) {
    printf("%c\n", *cp);
   };

   /*Print list 'testatt[1].demo'*/
   for (cp=pe->cPerson_buf.testatt[1].i_demo->Init(); cp;
      cp=pe->cPerson_buf.testatt[1].i_demo->More()) {
       printf("%c\n", *cp);
   };
  };
 pe = pe->get_next();
 }
}
```

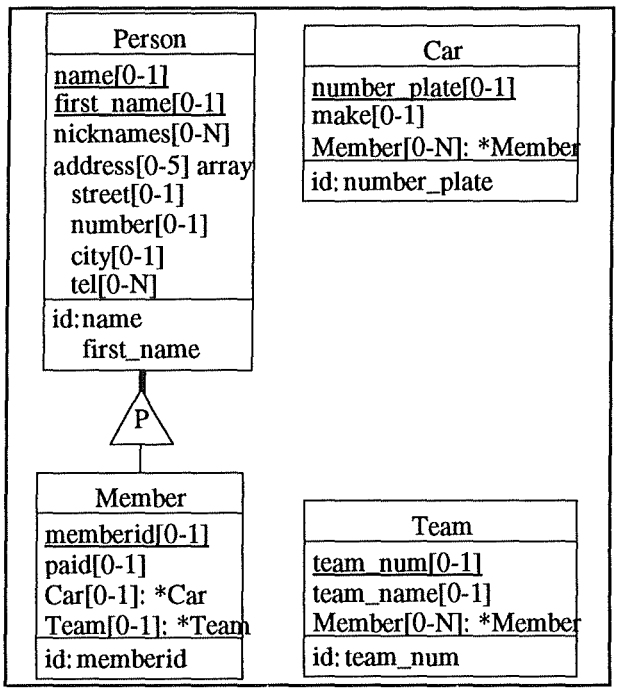
Output of the programme:

```
Id. is not respected! Object not created.
Id. is not respected! Object not created.
Tester cPerson:
Simpson Tom
1
2
Merckx Axel
7
8
Gevers Gert
2
3
4

delete
delete
delete
Press any key and then ENTER!
```

II.3.

III. The Generated C++ Code. Example 2

III.1. Physical Schema



Remark: The code of this schema is represented because it shows how the generator deals with inheritance, with single valued and with multivalued attributes.

III.2. The C++ Code Corresponding to the Schema

```
//Fichier C++ genere par GenerCpp.exe

#include <string.h>
#include <stdio.h>
#include <iostream.h>
#include "..\lists\tlist.h"

typedef enum {w_Car, w_Person, w_Member, w_Team} w_class;

class cCar;
class cPerson;
class cMember;
class cTeam;

class cCar {
public:
  struct t_cCar_buf {
    char number_plate[11];
    char make[24];
    Tlist <cMember> * Member;
    Titer <cMember> * i_Member;
  } cCar_buf;
private:
  cCar* next;
  cCar* previous;
  static cCar* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cCar * find_id(char number_plate[11]);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
public:
  cCar(t_cCar_buf &cCar_bu);
  ~cCar();
  static cCar * get_first();
  cCar * get_next();
  cCar * get_previous();
  virtual w_class what();
};

class cPerson {
public:
  struct t_cPerson_buf {
    char name[25];
    char first_name[25];
    Tlist <char> * nicknames;
    Titer <char> * i_nicknames;
    struct t_address {
      char street[31];
      int number;
      char city[17];
      Tlist <double> * tel;
```

```
    Titer <double> * i_tel;
   } address[5];
  } cPerson_buf;
 private:
  cPerson* next;
  cPerson* previous;
  static cPerson* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cPerson * find_id(char name[25],char first_name[25]);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
 public:
  cPerson(t_cPerson_buf &cPerson_bu);
  ~cPerson();
  static cPerson * get_first();
  cPerson * get_next();
  cPerson * get_previous();
  virtual w_class what() = 0;
};

class cMember: public cPerson {
public:
  struct t_cMember_buf {
   int memberid;
   bool paid;
   cCar * Car;
   cTeam * Team;
  } cMember_buf;
private:
  cMember* next;
  cMember* previous;
  static cMember* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cMember * find_id(int memberid);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
public:
  cMember(t_cMember_buf &cMember_bu,
       t_cPerson_buf cPerson_bu);
  ~cMember();
  static cMember * get_first();
  cMember * get_next();
  cMember * get_previous();
  virtual w_class what();
};
```

```
class cTeam {
public:
  struct t_cTeam_buf {
    int team_num;
    char team_name[31];
    Tlist <cMember> * Member;
    Titer <cMember> * i_Member;
  } cTeam_buf;
private:
  cTeam* next;
  cTeam* previous;
  static cTeam* first;
  bool init0_N_attr;
  bool inserted_into_list;
  cTeam * find_id(int team_num);
  void set_init0_N_attr(bool value) { init0_N_attr=value; }
  bool get_init0_N_attr() { return init0_N_attr; }
  void set_inserted_into_list(bool value) { inserted_into_list=value; }
  bool get_inserted_into_list() { return inserted_into_list; }
public:
  cTeam(t_cTeam_buf &cTeam_bu);
  ~cTeam();
  static cTeam * get_first();
  cTeam * get_next();
  cTeam * get_previous();
  virtual w_class what();
};


//global variables
cCar * cCar_former = NULL;
cPerson * cPerson_former = NULL;
cMember * cMember_former = NULL;
cTeam * cTeam_former = NULL;


//initiation static members
cCar * cCar::first = NULL;
cPerson * cPerson::first = NULL;
cMember * cMember::first = NULL;
cTeam * cTeam::first = NULL;

//member functions of class cCar
cCar::cCar(t_cCar_buf &cCar_bu)
{
  set_init0_N_attr(false);
  set_inserted_into_list(false);
  //check identifier(s)
  if (find_id(cCar_bu.number_plate))
    { delete(this); throw 11; }
  //pass values to data members
  memcpy(&cCar_buf, &cCar_bu, sizeof(cCar_bu));
  //add object to list
  next = NULL;
  previous = cCar_former;
  if (get_previous()) {get_previous()->next = this;} else {first = this;};
  cCar_former = this;
  set_inserted_into_list(true);
```

```
  // Initialise lists for multivalued attr
  cCar_buf.Member=
    new Tlist<cMember>(TLIST_NOT_REMOVE);
  cCar_buf.i_Member=
    new Titer<cMember>(cCar_buf.Member);

  set_init0_N_attr(true);
 }; //end constructor

 cCar::~cCar() {
  cMember * Memberpo;
  //Destroy lists and reiterators
  if (get_init0_N_attr()) {
  delete cCar_buf.i_Member;
  delete cCar_buf.Member;
  } //end if get0_N_...
  //Take object out of list.
  if (get_inserted_into_list()) {
  if (get_next()) {
    if (get_previous()) {
      get_previous()->next = next;
      get_next()->previous = previous;
    }
    else {
      get_next()->previous = NULL;
      first = next;
    };
  }
  else
    if (get_previous()) {
      get_previous()->next = NULL;
      cCar_former = previous;
    }
    else {
      cCar_former = NULL;
      first = NULL;
  } //end if
  } //end if get_inserted_...
  //Management of foreign obj attr with pointers to cCar
  Memberpo=cMember::get_first();
  while (Memberpo) {
    if (Memberpo->cMember_buf.Car == this) {
      Memberpo->cMember_buf.Car = NULL;
    }
    Memberpo=Memberpo->get_next();
  } //end while

 }//end of destructor

 cCar * cCar::get_first()
   { return first; }

 cCar * cCar::get_next()
   { return next; }

 cCar * cCar::get_previous()
   { return previous; }
```

```
w_class cCar::what()
  { return w_Car; }

cCar * cCar::find_id(char number_plate[11])
{
 cCar * pCar;
 pCar = cCar::get_first();
 while(pCar){
  if (
    (strcmp(pCar->cCar_buf.number_plate,number_plate) == 0)
    ) { return (pCar); } /*end if statement*/
 pCar = pCar->get_next();
 } /*end while*/
 return(NULL);
}

//member functions of class cPerson
cPerson::cPerson(t_cPerson_buf &cPerson_bu)
{
  int i;
  set_init0_N_attr(false);
  set_inserted_into_list(false);
  //check identifier(s)
  if (find_id(cPerson_bu.name, cPerson_bu.first_name))
    { delete(this); throw 11; }
  //pass values to data members
  memcpy(&cPerson_buf, &cPerson_bu, sizeof(cPerson_bu));
  //add object to list
  next = NULL;
  previous = cPerson_former;
  if (get_previous()) {get_previous()->next = this;} else {first = this;};
  cPerson_former = this;
  set_inserted_into_list(true);

  // Initialise lists for multivalued attr
  cPerson_buf.nicknames=
    new Tlist<char>(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);
  cPerson_buf.i_nicknames=
    new Titer<char>(cPerson_buf.nicknames);


  // Initialise lists for multivalued attr
  for (i=0; i<5; i++) {
  cPerson_buf.address[i].tel=
    new Tlist<double>(TLIST_REMOVE_WITH_NEW/*TLIST_NOT_REMOVE*/);
  cPerson_buf.address[i].i_tel=
    new Titer<double>(cPerson_buf.address[i].tel);
  };
  set_init0_N_attr(true);
}; //end constructor
```

```
cPerson::~cPerson() {
 int i;
 //Destroy lists and reiterators
 if (get_init0_N_attr()) {
 delete cPerson_buf.i_nicknames;
 delete cPerson_buf.nicknames;
 for (i=0; i<5; i++) {
 delete cPerson_buf.address[i].i_tel;
 delete cPerson_buf.address[i].tel; };
 } //end if get0_N_...
 //Take object out of list.
 if (get_inserted_into_list()) {
 if (get_next()) {
  if (get_previous()) {
   get_previous()->next = next;
   get_next()->previous = previous;
  }
  else {
   get_next()->previous = NULL;
   first = next;
  };
 }
 else
  if (get_previous()) {
   get_previous()->next = NULL;
   cPerson_former = previous;
  }
  else {
   cPerson_former = NULL;
   first = NULL;
 } //end if
 } //end if get_inserted_...
}//end of destructor

cPerson * cPerson::get_first()
 { return first; }

cPerson * cPerson::get_next()
 { return next; }

cPerson * cPerson::get_previous()
 { return previous; }

cPerson * cPerson::find_id(char name[25],char first_name[25])
{
 cPerson * pPerson;
 pPerson = cPerson::get_first();
 while(pPerson){
 if (
  (strcmp(pPerson->cPerson_buf.name,name) == 0 &&
   strcmp(pPerson->cPerson_buf.first_name,first_name) == 0)
  ) { return (pPerson); } /*end if statement*/
 pPerson = pPerson->get_next();
 } /*end while*/
 return(NULL);
}
```

```
//member functions of class cMember
cMember::cMember(t_cMember_buf &cMember_bu,
                 t_cPerson_buf cPerson_bu):
        cPerson(cPerson_bu)
{
 set_init0_N_attr(false);
 set_inserted_into_list(false);
 //check identifier(s)
 if (find_id(cMember_bu.memberid))
   { delete(this); throw 11; }
 //pass values to data members
 memcpy(&cMember_buf, &cMember_bu, sizeof(cMember_bu));
 //add object to list
 next = NULL;
 previous = cMember_former;
 if (get_previous()) {get_previous()->next = this;} else {first = this;};
 cMember_former = this;
 set_inserted_into_list(true);
 set_init0_N_attr(true);
}; //end constructor

cMember::~cMember() {
 cMember * op;
 cCar * Carpo;
 cTeam * Teampo;
 if (get_init0_N_attr()) {
 } //end if get0_N_...
 //Take object out of list.
 if (get_inserted_into_list()) {
 if (get_next()) {
   if (get_previous()) {
     get_previous()->next = next;
     get_next()->previous = previous;
   }
   else {
     get_next()->previous = NULL;
     first = next;
   };
 }
 else
   if (get_previous()) {
     get_previous()->next = NULL;
     cMember_former = previous;
   }
   else {
     cMember_former = NULL;
     first = NULL;
 } //end if
 } //end if get_inserted_...
 //Management of foreign obj attr with pointers to cMember
 Carpo=cCar::get_first();
 while (Carpo) {
   if (
   (Carpo->cCar_buf.Member->Length() != 0) &&
    Carpo->cCar_buf.Member->Member(this))
   {
   for (op=Carpo->cCar_buf.i_Member->Init();
     op;
```

```
      op=Carpo->cCar_buf.i_Member->More() )
      {
       if(op == this) {
       delete Carpo->cCar_buf.i_Member->Whole();
      } //end if
     } //end for
     } //end if
    Carpo=Carpo->get_next();
   } //end while

   Teampo=cTeam::get_first();
   while (Teampo) {
    if (
    (Teampo->cTeam_buf.Member->Length() != 0) &&
     Teampo->cTeam_buf.Member->Member(this))
    {
    for (op=Teampo->cTeam_buf.i_Member->Init();
      op;
      op=Teampo->cTeam_buf.i_Member->More() )
      {
       if(op == this) {
       delete Teampo->cTeam_buf.i_Member->Whole();
      } //end if
     } //end for
     } //end if
    Teampo=Teampo->get_next();
   } //end while

}//end of destructor

cMember * cMember::get_first()
  { return first; }

cMember * cMember::get_next()
  { return next; }

cMember * cMember::get_previous()
  { return previous; }

w_class cMember::what()
  { return w_Member; }

cMember * cMember::find_id(int memberid)
{
 cMember * pMember;
 pMember = cMember::get_first();
 while(pMember){
  if (
    (pMember->cMember_buf.memberid == memberid)
    ) { return (pMember); } /*end if statement*/
 pMember = pMember->get_next();
 } /*end while*/
 return(NULL);
}
```

```
//member functions of class cTeam
cTeam::cTeam(t_cTeam_buf &cTeam_bu)
{
 set_init0_N_attr(false);
 set_inserted_into_list(false);
 //check identifier(s)
 if (find_id(cTeam_bu.team_num))
  { delete(this); throw 11; }
 //pass values to data members
 memcpy(&cTeam_buf, &cTeam_bu, sizeof(cTeam_bu));
 //add object to list
 next = NULL;
 previous = cTeam_former;
 if (get_previous()) {get_previous()->next = this;} else {first = this;};
 cTeam_former = this;
 set_inserted_into_list(true);

 // Initialise lists for multivalued attr
 cTeam_buf.Member=
  new Tlist<cMember>(TLIST_NOT_REMOVE);
 cTeam_buf.i_Member=
  new Titer<cMember>(cTeam_buf.Member);

 set_init0_N_attr(true);
}; //end constructor

cTeam::~cTeam() {
 cMember * Memberpo;
 //Destroy lists and reiterators
 if (get_init0_N_attr()) {
 delete cTeam_buf.i_Member;
 delete cTeam_buf.Member;
 } //end if get0_N_...
 //Take object out of list.
 if (get_inserted_into_list()) {
 if (get_next()) {
  if (get_previous()) {
   get_previous()->next = next;
   get_next()->previous = previous;
  }
  else {
   get_next()->previous = NULL;
   first = next;
  };
 }
 else
  if (get_previous()) {
   get_previous()->next = NULL;
   cTeam_former = previous;
  }
  else {
   cTeam_former = NULL;
   first = NULL;
 } //end if
 } //end if get_inserted_...
```

```
  //Management of foreign obj attr with pointers to cTeam
  Memberpo=cMember::get_first();
  while (Memberpo) {
   if (Memberpo->cMember_buf.Team == this) {
    Memberpo->cMember_buf.Team = NULL;
   }
   Memberpo=Memberpo->get_next();
  } //end while

}//end of destructor

cTeam * cTeam::get_first()
  { return first; }

cTeam * cTeam::get_next()
  { return next; }

cTeam * cTeam::get_previous()
  { return previous; }

w_class cTeam::what()
  { return w_Team; }

cTeam * cTeam::find_id(int team_num)
{
 cTeam * pTeam;
 pTeam = cTeam::get_first();
 while(pTeam){
  if (
    (pTeam->cTeam_buf.team_num == team_num)
    ) { return (pTeam); } /*end if statement*/
 pTeam = pTeam->get_next();
 } /*end while*/
 return(NULL);
}


//delete all objects of one class
template <class S>
void delete_one_class(S * aptr) {
  while(aptr) {
   delete aptr;
   aptr = S::get_first();
  }
}

//Send error message.
void send_error_message(int catchint) {
  if (catchint==11) printf("ID is not repected. Object not created!\n");
  if (catchint==22) { printf("Obligatory value remained empty.\n");
                      printf("Object not created!\n");
  }
}
```

```
int main()
{
 try {
 }
 catch(int i) {
  send_error_message(i);
 }
 return 0;
}
```