Aarhus Universistet
Matematisk Institut
Datalogisk Afdeling

An implementation of a persistent store for C++

Peter Hübel
Jens T. Thorsen

December 1992

Contents

Introduction....1
Motivation....1
1. Persistence in general....2
    1.1. Important topics....3
    1.2. Initialization of the persistent store....3
    1.3. Retrieval of persistent objects....3
    1.4. Persistent types....4
    1.5. Indicating persistence....4
    1.6. Locating persistent objects....4
    1.7. Manipulation of persistent objects....5
2. A few persistent systems....6
    2.1. BETA....6
    2.2. Arjuna....7
    2.3. Ontos....7
3. Persistence and the C++ language....9
    3.1. Object layout....9
    3.2. Object identity....11
    3.3. Component subobjects & arrays....14
    3.4. Run-time type information....15
    3.5. Instance creation....16
4. Implementation....18
    4.1. Overview....18
    4.2. Basic ideas....18
    4.3. Preprocessing....20
    4.4. The persistent base class....20
    4.5. Object identity....21
    4.6. Run-time type information....22
        4.6.1. Prototype objects....23
        4.6.2. The TypeInfo class....26
    4.7. The Class Manager....27
        4.7.1. Class identifiers....27
        4.7.2. Generating new instances....27
    4.8. The Object Manager....27
        4.8.1. Naming persistent roots....28
        4.8.2. Storing objects....29
        4.8.3. Object retrieval & proxy objects....29
        4.8.4. Object retrieval & proxy objects....31
        4.8.5. Object caching....31
        4.8.6. The repository....32
    4.9. Limitations....32
5. Conclusion....33
6. Future work....33
References....34
Appendix A: Source code....36

Introduction

Persistence is the ability to let objects survive the program execution in which they were created. Objects are kept on secondary storage and may later be read into volatile memory and take part in other program executions.

This paper describes the design and the implementation of a library and a set of tools for managing persistent objects in C++. The system uses the languages inheritance mechanisms for providing persistence to objects of user-defined classes. A form of *deep persistence* is used, i.e. when an object is stored or retrieved, all objects reachable from it via references are stored respectively retrieved simultaneously. User programs can control the retrieval of objects, though, in cases where it is more adequate to load only parts of a larger structure.

Making very few assumptions below the language level, the library is very portable and the system has been implemented and tested on both UNIX and MS-DOS platforms without any changes to the source code.

The system does not support all aspects of the C++ language, in particular it is not possible to have persistent instances of classes using multiple inheritance or containing reference variables.[1] Other aspects like arrays and component objects are not supported in the current version.

Motivation

The work presented in this paper constitutes an offshoot from the ESPRIT Project 5303 EuroCoOp [ECO-JT-91-2] at Aarhus University. The basic constructs of the system which implement run-time type information and object management were designed in connection with the project in the task of providing a C++ interface for the object-oriented database interface (OODBI) developed in the project.

Part of the project, stated as an additional requirement in [ECO-JT-91-2], is to investigate the possibilities of obtaining language independence in the OODBI, making it possible to exchange data (objects) between applications written in different languages.

For this purpose a Common Object-Oriented Language (COOL) has been designed, which includes features common to most object-oriented programming languages (OOPL). COOL serves mainly as a data-definition language for specifying the interface of a class, whereas the implementation of methods must be done in an OOPL. The OODBI is designed with a BETA language [KMMPN91] interface, and currently a C++ interface is under preparation by the authors. A COOL to C++ translator has already been implemented, using the Mjølner BETA meta-programming system.

---

[1] Note that the word *reference* has a special meaning in the C++ language. It is not just an address - it is an address guaranteed to refer to an object, i.e. it can't be null. In other languages and as a general concept the term means the equivalent of a *pointer* in C++. To avoid confusion, we will use the terms *reference variable* or *reference member* for the C++ construct.

1. Persistence in general

Persistence is the ability for data to exist beyond the termination of the program which created them. In recent years researchers have been working upon integrating some of the facilities from the database world with traditional programming language features into persistent programming languages. Especially the occurrence of new data extensive applications (graphics systems, CAD, CAM, CASE, Expert Systems, and Office Information Systems) characterised by their need to store and retrieve large amounts of shared, structured data has resulted in this research area.

Some existing languages, amongst them ML [Harper], Smalltalk [GR80], and several LISP dialects, allow the user to save the objects of the program by writing the current state of the heap to disk. Upon restart of the system, the " image" of the heap can be loaded into the main memory and the computation may continue. However, there are some disadvantages using this method:

The amount of data allowed to be stored is limited by the size of the main memory [Cockshott]. The languages assume the whole collection of data to be loaded into main memory at the start of a session, so the amount of data must never exceed the amount of RAM allocated to the heap.

Also, it is inadequate to store everything placed in the heap – it is a waste of storage and may take a considerable time. The user should be able to store only the data of importance as well as to reload data from previous sessions only when necessary. Most persistent programming languages provide these features.

Database Systems are primarily used to store and manipulate a large and long-lived amount of data. They support features like persistence, data sharing, and protection against hardware and software failures (recovery) while conventional programming languages, on the other hand, always have emphasised upon definition and manipulation of data structures existing only during the execution of the program in which they are created. The data can be divided into two classes [Hughes91]:

- *Volatile data:* Data residing in the main (volatile) memory, created and manipulated by the programming language.
- *Persistent data:* Data residing in the secondary (persistent) store, which has to be transferred into the main memory before it can be accessed and manipulated

The clear separation between volatile and persistent data results in a major programming effort and overhead of code. Due to the type-structures in traditional databases to be extremely poor a great amount of code (estimated to be nearly 30% of an application) has to be produced for converting the data from the database into the internal formats of the program.

Many of the existing databases can be accessed and manipulated by the use of a data manipulation language (DML), e.g. SQL, QBE, and QUEL. The interaction with a database can take place in two ways [Nikhil]: Either by the use of an interactive DML, or by the use of an embedded DML, which means insertions of database operations within an existing programming language, e.g. C and PL/1.

Interactive DMLs can be very useful, but they typically lack the computational completeness to express non-data-manipulation parts, e.g. procedural abstraction, complex mathematical operations, etc.

Although some useful functionality is provided in embedded DMLs, mismatch between the query language and the programming language can provide some difficulties, e.g. no effective type check can be done. In order to get some effective use of the system, the programmer has to know all the eccentricities of both languages.

The problems arisen at the interface between the DML and the application language from the difference of type systems and programming paradigms are termed *impedance mismatch* [ZM90]. The fully integration of a persistent store into a language would solve the impedance mismatch problem.

1.1. Important topics

When trying to define persistence for some arbitrary language/system a number of important issues have to be considered:

- Initialization of the persistent store
- Retrieval of persistent objects
- Persistent types
- Indicating persistence
- Locating persistent objects
- Manipulation of persistent objects

Each of these will be discussed in the following sections.

1.2. Initialization of the persistent store

Before a persistent store can be accessed, some kind of initialization procedure has to establish the connection between the application and the store. The programmer can explicitly do the opening of a store or the connection can implicitly be established by the run-time system at start-up. If a system is capable of manipulating multiple persistent stores, a hybrid method can be obtained by letting the run-time system establish the connection to a default storage and then provide the user with the ability to explicitly open another storage, if requested.

1.3. Retrieval of persistent objects

When the connection is established the communication between the persistent store and the application can begin. Due to the great amount of data, persistent values must be stored in persistent memory even during program execution and only data required by the program should be kept in the main memory. Two approaches exist for loading data into main memory:

- *Static (or explicit) approach:* The language provides some primitive operations to specify that a value must be transferred [CM].

- *Dynamic approach:* The system transfers values into main memory when needed, just as the pages in a virtual memory system, e.g. the language Galileo [Albano] or the ObjectStore system [Lamb91]. In case of space shortage, some values can be selected to be moved back into the persistent memory.

1.4. Persistent types

In the development of a persistent programming language/system it has to be considered whether the ability of persistence should be *orthogonal* to typing, i.e. persistence is provided for all types of the system, an approach pioneered by PS-ALGOL [Nikhil], or persistence should be a property of a data type (*type dependent persistence*). Regarding the latter case persistence can be restricted to either instances of a certain type or a fixed set of built-in types, e.g. the type `database` in Pascal/R [Nikhil]. Type orthogonal persistence implies that persistent and volatile data should be accessed and manipulated in the same manner, i.e. the same code is used to operate on both kind of data (*manipulation transparency*). Although systems having a uniform treatment of all the objects whether they are persistent or volatile pay a run-time expense[2] it is widely accepted that the requirements for future persistent languages should include type orthogonal persistence. The uniform treatment makes it much easier to use and many programming errors are thereby avoided.

In the work of providing persistence to C++ most implementations use the type dependent persistence approach requiring the user-defined classes to be derived from a special " persistence supporting" base class.

1.5. Indicating persistence

The manner an object is shown to be persistent in an application can be done in a variety of ways [ZM90]. When the language is dealing with orthogonal persistence, persistence is a property to an instance, so the language must provide a run-time method for indicating the objects that are persistent.

One technique is termed *persistence by reachability* and implies the existence of a persistent root object. All objects reachable via references from this root are by default persistent objects, and those not reachable are volatile. If the language provides support for garbage collection and the persistent root, and the garbage-collection root is the same, all addressable objects at the termination of the program will persist.

1.6. Locating persistent objects

As a consequence of the persistent store holding a large number of objects the system must support some kind of object identity for producing an effective way of retrieving objects. Identity is a property of an object which distinguishes it from all other objects. Inspired by the use of primary keys as the basic addressing mechanism in relational

---

[2] A given pointer reference might refer to a persistent object, so the system is required to test whether a given pointer refers to a persistent object already stored in the main memory or an object which has to be loaded from the storage.

databases the use of tables and index files seems to be a suitable solution to this problem. However, it can be difficult to find a unique key among the attributes of a given object and user-defined keys can also provide some integrity problems in the persistent store. For instance if an objects key value is changed it has to be ensured that objects referring to the changed object have their references updated otherwise they might refer to a nonexistent object (*dangling references*).

A better way of representing object identity, ensuring referential integrity, is termed *identity through surrogates* [Hughes91]. Surrogates are system-generated, globally unique identifiers that are unique with respect to all the surrogates that ever existed in the persistent store. This means that even if an object is deleted from the store its identifier will never be reused by a new object. At the user level user-defined keys can still be used in identifying objects and a mapping between the key and the surrogate will ensure retrieval of the right object from the persistent store. In the persistent store all object references are based on surrogates so the problems mentioned earlier, arising from changes in a given objects attributes, are avoided.

1.7. Manipulation of persistent objects

To minimize mismatch persistent data should be manipulated in the same way as volatile data. This means that functions used for operating upon volatile data also should be used to operate upon persistent data which is the case in type orthogonal persistence. For a language where persistence is a property of types this implies that the programmer should be able to construct functions for persistent data in the same manner as for volatile data.

2. A few persistent systems

Using the framework given in the previous section, we briefly describe some systems offering persistence. Lately many such systems have emerged, but we limit ourselves to a few of them. Regarding systems providing persistence for an already existing language we are not interested in those requiring a special compiler, e.g. the ObjectStore database system [Lamb91]. The systems presented here were mainly chosen for their availability and for the different approaches they use. Although the systems may contain other special features, e.g. concurrency, sharing of objects, recovery and query capabilities, we will restrict ourselves to the study of persistence. Since the OODBI of the EuroCoOp project is implemented in the BETA language using its built-in persistence, a short description of this aspect of BETA is included.

2.1. BETA

The BETA language [KMMPN91] is an object-oriented programming language developed in the joint Nordic *Mjølner* research project. The language supports (deep) persistence orthogonal to types [Mjølner91] based on the ideas proposed in [Agesen89]. Persistent objects are stored in a flat representation in a persistent store, i.e. a file. The representation of the persistent store is provided by the BETA class `persistentStore`. It is possible to have more than one persistent store, but at the present time it is not possible to have references between objects stored in different persistent stores. The class `persistentStore` provides operations for opening and closing persistent stores and for storing and retrieving of persistent objects. It also provides an operation `initFragment` providing the persistent store with references to the library class patterns used to instantiate the objects that are to be retrieved[3]. When an object is made persistent all the objects reachable via references from this object will also become persistent automatically.

- *Initialization:* The connection to the store is established explicitly by use of the function `open`.
- *Retrieval:* An object is retrieved explicitly by the use of the function `get`.
- *Types:* The language supports type orthogonal persistence.
- *Indicating:* Persistence by reachability. The explicit storing of an object makes it become persistent root.
- *Locating:* Objects are stored under a user-defined name. All objects are named by a globally unique identifier (GID) for internally use.
- *Manipulation:* Persistence being orthogonal to types, all objects are manipulated in the same manner.

[3] This operation will be superfluous in the next release.

2.2. Arjuna

Arjuna is a research software system developed at The University of Newcastle upon Tyne [Arjuna] for constructing distributed, fault-tolerant applications. As a base for distribution, naturally the system has means of making object persistent. The system is implemented in C++ and is freely available.

Persistence in Arjuna is achieved through the inheritance mechanism of C++ (multiple inheritance is not supported, though). The " persistent" classes are derived from an abstract base class `StateManager` which manages the state of persistent objects, i.e. whether they are active or passive, and provides them with all the basic support mechanisms for persistence.

The hardware components of the system are workstations (nodes) connected by a network. Every node consists of a (stable) object store, where the persistent objects normally resides. An object is made active by loading the passive state into the home nodes primary memory. In distributed systems operations upon an object sited in a remote node is typically invoked via remote procedure calls. Arjuna's RPC Module[4] takes care of such calls by creating a server process at the remote host that contains the object.

- *Initialization:* The connection to the stable store is established implicitly by the run-time system.
- *Retrieval:* The activation of a persistent object is obtained explicitly.
- *Types:* The class `StateManager` defines persistent behaviour, so instances of its derived classes may be stored in the system. Though this means that it is possible to design an arbitrary class and provide it with persistence is it still type dependent persistence that is provided.
- *Locating:* Internally to the system all naming of persistent objects is performed using unique identifiers but Arjuna also provides a name server allowing persistent objects to be registered and reached via user-defined names.
- *Manipulation:* After the activation of an object, it can be manipulated in the same manner as volatile objects. Arjuna's RPC Module makes it transparent to the user whether the object is sited in the home node or in a remote node.

2.3. ONTOS

ONTOS is a commercial multiuser, distributed object database developed by Ontologic, Inc[5]. The system provides persistence for C++ objects. It supports the multiple inheritance feature of C++ but does not support persistence for objects inheriting from virtual base classes[6]. The system consists of two parts: A C++ class library and a number of free functions. The class library provides persistent classes which can be specialized by users, classes for managing references to other persistent objects etc. The free functions

---

[4] Restrictions are imposed on 'persistent' classes due to the RPC system.

[5] The information in this section is mainly taken [ONTOS91a] and [ONTOS91b].

[6] As we shall see later on, there are good reasons for this restriction.

constitute the interface between the C++ application and the database, including functions for opening and closing of a logical database. Persistence in ONTOS is achieved by inheriting form the persistent classes. The class `Object` is an abstract base class defining attributes and functions that are required for supporting persistence, so objects must be derived from this in order to become persistent. References to other (possibly persistence) objects can be maintained by means of the class `Reference`, which works as a kind of abstract reference to objects.

- *Initialization:* The connection to the logical database is established in an explicit way by the use of the function `OC_open()`.
- *Retrieval:* Persistent objects can be activated in both explicit and implicit ways. The function `OC_lookup()` is an example of the explicit way and the *abstract* reference mechanism provides the implicit method for activating objects.
- *Types:* The class `Object` defines persistent behaviour, so only instances of its derived classes can be stored in the database. As mentioned earlier (section 2.2) is it possible to design an arbitrary class and provide it with persistence, but note that this does not mean type orthogonal persistence is obtained.
- *Locating:* ONTOS provides means for assigning names to persistent objects that can serve as entry points into the database. Apart from instances of the *primitive* classes of ONTOS; that is classes representing `int`, `char`, `char*`, etc., all persistent objects are assigned a system generated, globally identifier that is used internally.
- *Manipulation:* Persistent objects can be manipulated in the same manner as volatile objects.

3. Persistence and the C++ language

In this section we discuss some of the language features that somehow are related to persistence, in particular the issues of obtaining object layout, object identity and run-time type information (RTTI). It also serves as a rationale for many of the decisions we made during the design process. The discussion is largely based on the reference manuals [Stroustrup91] and [ES90]. References starting with [§] are to sections in [ES90]. We assume familiarity with the basic object-oriented features of C++ such as *inheritance, access control, constructors* and *overloading*.

3.1. Object layout

Insight on the internal structure of objects is essential to the process of designing a persistent store. Clearly, the way data is organized in a computers memory is a low level issue. Nevertheless C++, being famous for " assembler" features like pointer arithmetics and bitwise operators, encourages an investigation of what knowledge can be obtained by " pure" language constructs.

The order of members in an object is implementation dependent. C++ does not even guarantee the order in which storage is allocated for derived classes [§10.1c]. Obtaining the location of a data member in an object could of course be done by simple pointer arithmetics, subtracting addresses, but this could in general not be done until run-time. Fortunately C++ provides a language construct which can denote a specific member in an arbitrary object of a class: Pointers to members [§8.2.3]. A pointer to member (ptm) can be applied to an object to inspect the member it refers to. By using explicit casts, a ptm can be applied even to instances of base classes – a fact that might turn out to be quite useful in the design of a general scheme for accessing the interior of objects. Consider the following example:

```
class Base {
private:
  int bprv;
public:
  int bpub;
};

class Derived : public Base {
private:
  int dprv;
public:
  int dpub;
};

int Base::*ptbm = &Base::bpub; // pointer to base member
int Derived::*ptdm = &Derived::dpub; // pointer to derived member

ptbm = (int Base::*)ptdm; // legal but dangerous
ptdm = (int Derived::*)&Base::bpub; // legal and harmless
```

It is easy to see that when dereferencing `ptbm` on a `Base` object the result will be undefined.[7] However, since the layout information for a certain class customarily only is used in connection with instances of the exact class, this should impose no problems.

The usual access rules are effective so initializing a pointer to a private member has to take place in the scope of the class or in some friendly neighbourhood[8]. There are several problems though.

For once it is not possible to refer to members declared as `private` in base classes. `Base::bprv` is inaccessible anywhere but in the scope of class `Base`. Hence there is no direct method for class `Derived` to obtain information about the layout of its own instances, unless `Derived` were declared as a friend class of `Base`. However, it seems like a harebrained idea to declare a list of friends for a class which actually comprises all classes derived from it. We might as well have declared `Base::bprv` as a `protected` member to grant access to it from derived classes, and thereby totally spoil the (quite reasonable) idea of having private members!

A further problem with pointers to members is that it is impossible to have a ptm to a reference member. It is not even possible to have a normal pointer to a reference variable[9] so pointer arithmetic could not solve the problem either. Reference variables impose additional problems for persistence as will be shown later. For now we ignore them and investigate the ptm approach further.

One method to obtain the complete layout for a class is easily implemented – the work could be done in the body of a function which is declared `friend` of a base class. Friendship is inherited, so the function will have access to the entire scope of all descendants of that class and can obtain information about every member in every class. It is an undesirable solution, though, since it imposes the restriction that all classes should be known at once – the function has to be rewritten every time the layout for an additional class is wanted. Even when neglecting reference variables it does not seem possible to obtain a satisfactory solution.

The most common layout is to arrange the members in order of appearance [§10.1c], so that the members defined in a class are clustered and are placed after members inherited from base classes. Obviously this is also the easiest and most efficient implementation. Assuming that reality reflects this (it does so with all compilers we have used), one could let each class deliver the layout for the part it provides, i.e. of the members defined in the class itself, and somehow infer the complete layout from the parts. This is easily done for single inheritance, since the offsets calculated for the inherited parts are valid in all derived classes too.

Multiple inheritance [§10.1, §10.2c] imposes only a minor problem, though we feel that things get too implementation dependent beyond this point. The offsets calculated in

---

[7] There is no way to check whether a given ptm can be succesfully dereferenced on a given object. Although considered with the problem on down-casting, the proposal of RTTI for C++ in [SL92] does not deal with pointers to members.

[8] Classes or functions declared as `friend` will obtain full access to private parts of a class.

[9] Reference variables are a second order data type in C++, they serve mainly as an " alias" for objects.

the individual parts have to be interpreted differently every time a class is inherited further down in a hierarchy[10]. Thus the offset of the inherited parts must be calculated each time a class is multiply inherited somewhere and stored along with the information on the ordinary members. The language has no constructs to support this directly. It can however be accomplished by pointer arithmetics, as illustrated in Fig.1. A discussion of offsets can be found in [§10.3c].

```
class A {
...
};

class B {
...
};

class Example : public A : public B {
public:
  int get_offset_of_A_part();
  // other declarations
  ...
};

int Example::get_offset_of_A_part()
{
  A* thisA = this; // implicit conversion - thisA points to A part
  int offset = this - thisA; // pointer subtractions
  return offset;
}
```

Fig.1: Calculating the offset of an inherited part

The use of virtual base classes [§10.1, §10.5c] would make this approach unbearably complex. In addition to the above, the calculated offsets are only valid for the class but not for further classes derived from it.

One notices that in many systems dealing with persistence in C++ multiple inheritance is not supported at all, whereas system capable of handling this feature require applications to be developed with certain compilers due to the implementation dependency.

3.2. Object identity

Since objects may persist beyond the lifetime of the program that created them, they can be part of many different program executions. Even when they occur in programs other than the original, we would like to view the different instances as being the same object. Clearly *identity by surrogates* [Hughes91] is the best approach, hence we need some way to associate each object with a unique object identifier (OID).

---

[10] We use the common visualization depicting derived classes “ below” subclasses.

Noticing the fact that no two objects coexisting in volatile memory share the same address leads to a possible implementation: A table containing (OID, address) pairs could be used to keep track of the identity of objects. Naturally the OIDs have to be stored along with the objects in the persistent store.

When dealing with part objects this approach would fail, though, since it leads to an unacceptable asymmetry. Consider the example of Fig.2.

```
class Aggregate {
public:
  // Component
  Part one;
  // Other declarations
  ...
  // Another component
  Part two;
} foo;

if (&foo.one == &foo) {
  // Can't distinguish aggregate and component
  ...
}
```

Fig.2: Component subobjects

A component placed at the beginning of an object typically has the same address as the object itself. Hence it cannot have an identity of its own whereas other components (residing on higher addresses) can. The problem vanishes though when all classes inherit from the same base class, since it will at least contain a pointer to the virtual function table. Anyhow, this approach would cause a significant overhead in determining the identity of an object.

The simplest way to implement object identity is by declaring a member in some common base class and let it represent the OID of an object.

The immutability of OIDs is quite easy to accomplish. If the OID is inherited from a common base class then declaring the member `private` in the base class will make the OIDs inaccessible by users.

Regarding uniqueness, we have found the copy semantics of C++ to be a rather useful feature. In his critique of C++ [Sakkinen92] the author blames the language for not providing a better way to express that copying is not applicable to instances of a certain class[11]. He states that we can't 'sensibly make a copy of a "person" object in any system that handles persons as individuals' – a remark heard quite often in object-oriented research. The reason given is that if the "real world" cannot provide two identical

[11] It can be done by declaring the copy constructor and assignment operator `private`, and thus not accesible for "normal" users.

instances of something, the model we create on the computer should not be able to do so either.

But consider a pack of puppies, let's say Alsatians where it is really hard to tell one from the other. To overcome the problem their owner gives them different names and different collars. It is noticeable that the only way to distinguish them is by properties *not* inherent to the concept of a dog (though often associated with it). The " real world" also gets confused when two people have the same name, a problem which sometimes is overcome by inventing nicknames. In the area of distributed systems the concept of copying is crucial for replicating objects on different sites. Here even the " identity" of an object has to be copied, though it somehow has to be possible to distinguish a copy from the " original" .

What we really seem to be afraid of is not copying – it is the *loss of identity*. Since object-oriented programming to a very large extent is based on *reference by identity*, it seems a bit odd that it should be necessary to prohibit copying for some concepts. Copying is a mere transfer of values and it is *de facto* in the world of object-orientedness that identity should not be based on such. Ideally identity should be transparent, and hence the semantics of a copy operation should be to transfer only the user-defined parts of an object, retaining its identity.

In C++ the default copy semantics have been memberwise since Release 2.0[12], so objects can actually be copied *without* loss of identity, simply by declaring a default constructor, a copy constructor and assignment operator for the identifying part of the object. These special member functions can take appropriate actions, e.g. assign a new identifier in the default and copy constructor but retain the old identifier in the assignment, to ensure uniqueness of the identifying part. With this behaviour defined for a class `OID`, consider the following example:

```
class Unique_Object {
  OID unique_id;
public:
  int value;
  Unique_Object(int v) { value = v; };
};

Unique_Object x(1); // x has value 1
Unique_Object y(2); // y has value 2
Unique_Object z = x; // z gets the value 1 from x
y = z; // y gets the value 1 from z
```

Due to the default constructor of `OID` the `unique_id` member for each object created has a unique value. The use of `x` to create `z` will call the (default) copy constructor for `Unique_Object`, which in turn calls the copy constructor for `OID`, assigning a new unique value to the `unique_id` of `z`. Now even if the copy constructor for `Unique_Object` had been defined to override the default semantics, the default constructor for `OID` would be invoked leading to the same result. The assignment of `z` to `y` will eventually invoke the assignment operator of class `OID`, leaving the `unique_id` of `y` unchanged.

---

[12] Earlier versions of the language used bitwise copying.

There are cases known when the identifying part of an object has to be duplicated, e.g. in multi-user object-oriented database systems where several (read only) copies of an object may coexist during a transaction. For this purpose a special member function could be provided, overriding the default behaviour.

It is in our opinion that the copy semantics of C++ provide a very good mechanism to implement identity though surrogates, though the consequences must be considered when using the suggested implementation. Using objects as value-parameters in a procedure call will create (temporary) copies and hence the object inside the procedure has another identity. This can be avoided by using call-by-reference (which is more efficient for large objects anyway).

3.3. Component subobjects & arrays

An object can be part of an other object as shown in the example Fig.2 on page 12. In [Sakkinen92] the term *component subobject* is used to distinguish it from parts inherited from base classes (*superclass subobjects*). The problem of identifying component subobjects goes further here since there is no way to decide if an object is part of an enclosing object or if it is a “ free” instance of its own. Using the classes of Fig.2 and given an instance of `Aggregate` and a pointer to a `Part` object, it *is* possible to determine whether the latter is part `one` of the first:

```
Aggregate a;
Part* p;

if (p = &(a.one)) {
}
```

But to decide if `p` is part of any object can not be accomplished without keeping track of all objects. The problem is that there is no link to the enclosing object as there is in other languages allowing component subobjects, e.g. BETA [KMMPN91]. The question is whether a component subobject should have an identity of its own or not. It seems certainly as the right choice, and in object-oriented database systems it is common to provide support for composite objects, e.g. the Orion system. On the other hand it makes no sense to talk about identity of an object which never exists on its own.

Somewhat the same problems occur with arrays in C++. Due to its heritage from the C language, arrays in C++ are not “ real” objects but merely a collection of them. Given a pointer to an object, it is not possible to decide whether the object referred to is an element in an array or not. Since the name of an array denotes a pointer to the first element, the “ identity” of an array would be the same as of the first element. This is a severe loss of information.

An easy way to overcome this is using *templates*[13] [§14] for creating array objects of any type, e.g. the declaration `Array<Part> myparts(200);` would create an array of 200 `part` objects, each with an identity of their own.

---

[13] Templates are the C++ notion of generic types. They are well suited for creating homogenous collections of objects.

3.4. Run-time type information

Somebody once stated that ‘there are no objects in C++’. This is true in the sense of objects being purely conceptual. C++ knows only of addresses, i.e. pointers and reference variables, but given an address it is not always possible to decide whether it denotes an object or not. This *type loss* has been discussed in [Sakkinen92] and is mainly due to the languages heritage from C.

With the increasing need for object-oriented databases the demand for RTTI has also emerged within the C++ community. Stroustrup and Lenkov [SL92] have recently prepared a proposal for the ANSI/ISO C++ committee, defining RTTI as a part of the language.

Until now there have been several approaches on providing RTTI to C++, mostly in the form of class libraries, which all essentially are based on a virtual function approach, resembling the one described in [Stroustrup91]. There are two possibilities for tagging type information to object of a class:

[1] Using a data member (possibly a pointer) to reflect the type.

[2] Declaring a virtual function for the class, returning something which denotes the type.

Both approaches require all classes to be derived from a single root, since objects in C++ like in all statically typed object oriented languages are restricted to *inheritance polymorhism*, i.e. polymorphism is only applicable among instances of classes belonging to the same branch of an inheritance hierarchy. In most languages this is no problem, since there exists a unique class, e.g. `Object`, which is ancestor of all other classes. C++ classes, on the other hand, are organized in “ forests” – there is no common base class. The generic pointer `void*` represents *no* type and can therefore not be used as a polymorphic base.

The second approach is clearly preferable, since it requires no special initialization of the instances to reflect their type. Had the type of an object been represented by a data member, there would essentially be two solutions: Either the member must be accessible to all derived classes by declaring it `protected` and non-`const`, which would have the unpleasant effect of making it modifiable. The alternative would be to propagate the initializing value all the way up to the constructor of the base class, since the language does only permit initialization of immediate base classes and members not inherited [§12.6.2].

We may notice that most implementations actually *have* type information for classes containing virtual functions: The reference in an object to the virtual functions table of the class clearly indicates its type. Knowing the details of the implementation one could use this as a basis for a RTTI system. This has been noted in the proposal [SL92].

This proposal (which may very well be accepted as part of the standard) is based on a language extension, where an operator `typeid()` returns an object of a special class `Type_info` which identifies the type of the object. It also suggests adding run-time checked type casts to the language. A few points are worthwile mentioning in this context:

- The proposal restricts RTTI to be available only for classes containing virtual functions, hence it will impose no overhead and give an easy implementation, using the existing idea with virtual function tables.
- The implementation of class `Type_info` is not defined as part of the language – only a few operations for dealing with equality are declared in the interface.
- Instances of class `Type_info` are ordered – but the ordering does not reflect any inheritance relationship among classes. It is not even guaranteed that the ordering is the same from one program execution to another.

3.5. Instance creation

When retrieving objects from secondary storage there must be means to create new instances of the proper class. In C++ an object often embodies more than just data members. This is equally true for most languages supporting virtual functions because any implementation somehow has to know which function to call for a particular object. Thus it is not sufficient simply to allocate a chunk of memory and declare it as an object.

C++ has no genuine " create" operator for objects. Instance creation involves two steps:

- Memory allocation – either on the stack or by calling the `new` operator.
- Initialization of members – this takes place in the constructors.

Apart from that the compiler does initialize a pointer to the virtual table to indicate the type of the object, but this step is beyond the control of the programmer. As a consequence it is not possible to obtain the address of a constructor [§12.1]. Even if it were we still would lack something. This leaves us with only two methods for creating objects dynamically (see Fig.3 on page 17):

[1] Wrap a call to `new` in a function and returning the created object. The constructor to use could be either a default or a special purpose constructor, designed just for this purpose to minimize the costs[14]. This technique can be enhanced by overloading the new operator making it possible to call the constructor on an already created object [§12.1, p.266].

[2] Take an existing object, allocate an appropriate chunk of memory and make a raw copy. An ugly approach, but it has been tested to work all right. Still, the object to clone has to be created in a usual manner.

In order to complicate things both methods have a severe deficiency though. Objects containing reference members cannot have default constructors since reference variables *must* be initialized [§8.4.3]. Creating objects containing reference members by either method may imply a considerable amount of overhead since it would be necessary to instantiate a number of dummy objects to fulfil this requirement.

---

[14] A bad designed class could have extensive actions taking place in the default constructor, e.g. retrieving initial information though a WAN.

```
Base* BaseGenerator1()
{
  return new Base();
}

Base* BaseGenerator2()
{
  static Base clonable; // prototypical object
  char* chunk = new char[sizeof(Base)]; // allocate memory
  memcpy(chunk, &clonable, sizeof(Base)); // copy the bytes
  return (Base*)chunk;
}
```

Fig.3: Generating Objects

4. Implementation

4.1. Overview

This section describes our implementation of a persistent store for C++. First we describe the basic ideas of the implementation and the changes necessary for an arbitrary class to support persistence. Section 4.4 describes the persistent base class. Section 4.5 is on object identity. Section 4.6 describes the representation of run time type information. Note that we unify the concepts of RTTI and object layout in viewing the latter as being part of the type information. In sections 4.7 and 4.8 the basic components of our implementation are presented: The Class Manager, managing the run time type information for the persistent classes, and the Object Manager, which transfers objects to and from the persistent store. Finally we discuss the limitations of our approach and suggest some solutions to overcome them.

An schematic overview of the systems components is depicted in Fig.4.

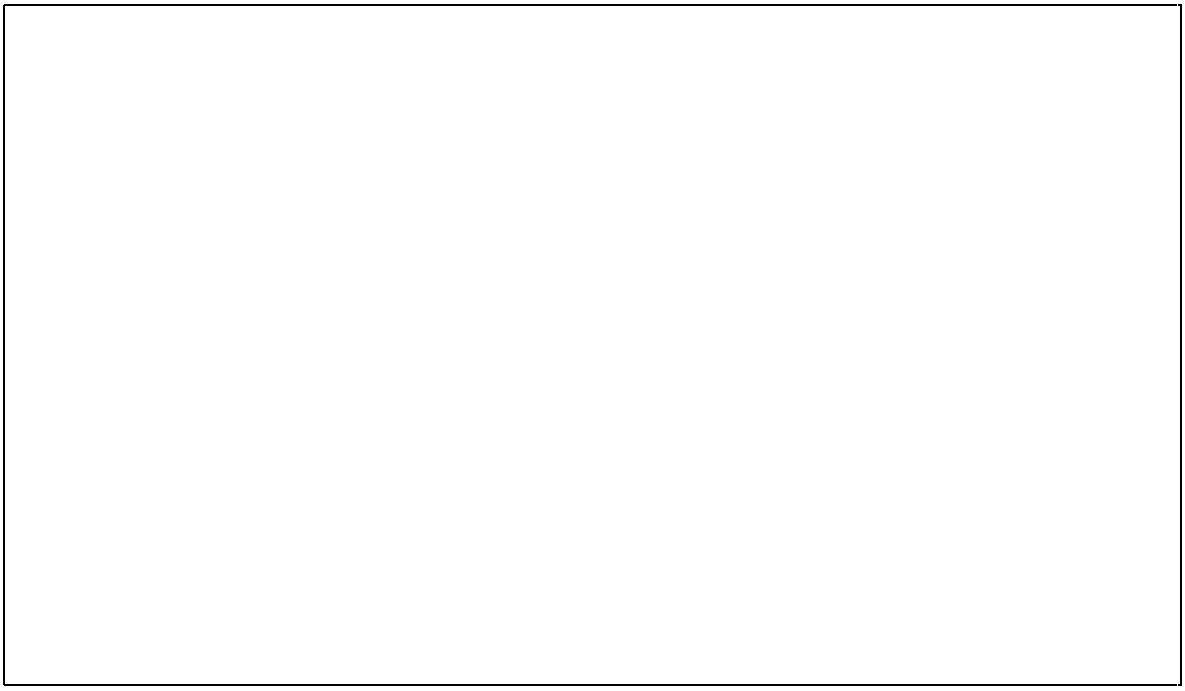

Fig.4: The components of the implementation

4.2. Basic ideas

Persistence for a class is obtained by inheritance from a common base class `COOL`[15] and by specialization of a virtual function `memberinfo()` returning an object which can be used to identify the class of an object.

---

[15] The name " COOL" was originally chosen to reflect that classes for use with the C++/OODBI interface were defined in the COOL language [ECO-JT-91-2].

Run-time type information is implemented by a structure reflecting the inheritance among persistent classes and with an interface inspired by a proposal currently presented to the ANSI/ISO C++ committee [SL92].

```
// Original header file

class Person {
private:
  char* name;
  int ssn;
  Person* spouse;
public:
  Person() {};
  void marry(Person* p) { spouse = p; p->spouse = this; };
};
```

```
// Preprocessed header file

class Person : public COOL {
private:
  virtual const TypeInfo& memberinfo() const;
private:
  char* name;
  int ssn;
  Person* spouse;
public:
  Person() {};
  void marry(Person* p) { spouse = p; p->spouse = this; };
};
```

```
// Generated RTTI source file

static int PersonTrigger = ClassManager::trig(Person());

static COOL* PersonGenerator(COOL* into)
{
  return (into ? new(into) Person() : new Person(); };
}

const TypeInfo& Person::memberinfo()
{
  static COOL_PT array[] = {
    COOL_PT("spouse", (COOL_POINTER)&Person::spouse),
    COOL_PT("name", (COOL_TEXT)&Person::name),
    COOL_PT("ssn", (COOL_INT)&Person::ssn),
  }
  static TypeInfo ti("Person", array, 3, PersonGenerator);
  return ti;
}
```

Fig.5: An example of preprocessing

Objects are “passive” in the sense that all the functionality for transferring persistent objects to and from stable storage is located in a special object called the Object Manager. References to objects not in volatile memory are represented by *proxy objects*. In the current implementation these are plain objects of the right type, but it makes no sense to access them since their values have not been retrieved from the persistent store.

The implementation uses persistence by reachability and objects can be stored as persistent roots and associated with names. The programmers is entirely responsible for ensuring that objects of “persistent” classes do not refer to volatile objects when being stored. This will most likely cause a run-time error.

4.3. Preprocessing

For an arbitrary class to support persistence a number of requirements must be fulfilled:

- The class must have class `COOL` as an ancestor.
- It must implement the inherited virtual function `memberinfo()` providing RTTI[16].
- A function for generating new instances of the class[17] must be implemented.

Furthermore there must be a call to a “trigger” function with a (temporary) instance of the class to notify the Class Manager of its existence.

A preprocessor taking care of all those things has been implemented. Given a normal header file with class definitions it generates a header file ready to include in user programs, as well as a C++ file containing type information and initialization functions which automatically will make the type information known to the Class Manager. An example class and the result of preprocessing it is shown in Fig.5 on page 19.

4.4. The persistent base class

The class `COOL` which acts as the common base for all persistent classes is shown in Fig.6 on page 21. Its two main purposes are to declare the virtual function `memberinfo()` and to provide its descendants with a unique identifier. Naturally friendship is granted to the Object Manager and the Object Cache – they have to know about the interior of any object.

Furthermore it declares a flag for use by the Object Manager. Any (potentially) persistent object can be one of three states when residing in volatile memory:

- *proxy* when the attributes of the object have not been retrieved yet (this will be explained in detail later on).
- *ready* when the object can be used like a normal (transient) object.
- *busy* when the object is being processed by the object manager. This serves merely as a flag for avoiding recursion when traversing the references from the object.

---

[16] Our implementation of RTTI includes a representation of the object layout.

[17] The current implementation requires the class to have a default constructor.

```
class COOL {
  friend class ObjMgr;
  friend class ObjectCache;
  friend const TypeInfo& typeid(const COOL&);
private:
  // Type information
  virtual const TypeInfo& memberinfo() const = 0; // pure virtual
  // Flag for traversal of closure
  enum { proxy=0, ready=1, busy=2 } state;
  // Unique identifier
  OID oid;
public:
  COOL() : state(ready) {};
  virtual ~COOL();
};
```

Fig.6: The common base class

The class is abstract due to the declaration of a *pure* virtual function[18]. This construct corresponds to the "subclassresponsibility" feature in Smalltalk [GR80] and means the class does not implement the function itself. Hence it makes no sense (and is disallowed) to create any instances of the class.

4.5. Object identity

In our implementation, as in most systems, object identity is supported by assigning a global unique object identifier (OID) to each object. For this purpose we supply a special class for object identifiers (class `OID`) which implements them as integers. Modelling OIDs as a separate concept rather than putting the integer directly into the object has at least two advantages. They become a separate type which the compiler can check statically and we can ensure their uniqueness as discussed earlier.

To ensure consistency of the system OIDs are never reused, even when objects are removed from the persistent store. If an object can't be found on retrieval (it might have been deleted by accident or lost by a system failure) the error surely will be noticed. But in the case where another object has taken its place there is a severe inconsistency which might be hard to detect. There is a problem, of course, in that there is an upper limit on the number of objects, that can be created. On the other hand, assuming 64 bit to represent an OID, we can create 2^{64} (approximately 16×10^{18}) objects. Given that one year is roughly 3×10^7 seconds, this seems more than enough to outlive current technology.[19]

Ideally an object should have an identity from the time of instantiation, but programs in C++ use value semantics more often than not. Passing objects around by value instantiates a great number of temporary objects which could lead to a significant "waste" of

---

[18] The C++ syntax " `= 0`" can be thought of as the pointer to the virtual function being **null**.

[19] In the current implementation OIDs are represented by 32 bits, though. Still, it permits storing one object per second in average for about 250 years.

identifiers. Hence we choose not to assign identifiers to objects before they enter the Object Manager and thereby become persistent. At instantiation time the default constructor for class `OID` will initialize objects to have an invalid (zero) OID.

```
class OID {
  friend class ObjMgr;
  friend class Buffer;
private:
  // Object ID represented as unsigned long
  unsigned long oid;
  // Constructors and type conversion used by friends
  OID(unsigned long o) : oid(o) {};
  operator unsigned long() const { return oid; };
public:
  // Outside can only create invalid OIDs
  OID() : oid(0) {};
  // Copy ctor prevents duplication of OIDs (ensures uniqueness)
  OID(OID&) : oid(0) {};
  // Assignment prevents duplication of OIDs (ensures uniqueness)
  OID& operator=(const OID&) { return *this; };
  // Calculation of hash value
  unsigned hash(unsigned seed) const {
    return unsigned(oid % (unsigned long)seed);
  };
  // Equality test
  int operator==(const OID& other) const { return oid == other.oid; };
  // Validity test
  int ok() const { return oid > 0L; };
};
```

Fig.7: The OID class

Declaring the Object Manager and class `Buffer` as friends makes the internal representation available to them. This way we can manipulate OIDs on a low level and still retain the desired semantics for unique identifiers as discussed in section 3.2.

4.6. Run-time type information

We merge the concepts of RTTI and object layout by viewing the latter as being part of the first. The RTTI proposal by Stroustrup and Lenkov [SL92] suggests that different kinds of extented type information (as the member layout of objects) should be implemented independently of each other – mainly to avoid overhead for users who only need minimal RTTI. In this implementation though, we maintain a tight coupling between the basic type information (giving the ability to compare objects by means of their `Type_info` objects) and the extended information (giving the ablity to apply the obtained information to objects) for efficiency reasons.

Ideally we would like any implementation of RTTI to fulfil the following requirements:

[1] Type information for a class should be shared among all instances of that class,

[2] it should not be able to modify it by users,

[3] it should be extensible in a sense to allow new (derived) classes being added to the system without recompiling any of the existing definitions and without having to store explicit knowledge about the type information of their base classes,

[4] the implementation should be portable and not rely on any specific object layout or compiler, and

[5] it should impose no or minimal added cost (in time or space) for objects that will not become persistent.

Every class implements a virtual function `const TypeInfo& memberinfo()`. The function returns an object that identifies the type of the object for which it is called. Inspired by [SL92], the system also provides a global function `const TypeInfo& typeid(COOL&)` which can be used in tests like:

```
Base* b;
Derived* d;
...

if (typeid(*b) == typeid(*d)) {
  // Call derived function
}
```

The functions are declared `const` to reflect that the type information obtained may not be modified.

4.6.1. Prototype objects

The term "prototype object" originates from the implementation of the BETA language. Here each class has associated a prototype object[20] with it, describing its layout [Madsen88].

We implement a prototype object as an array of class `COOL_PT`[21], henceforth referred to as a "prototype array" to avoid confusion with the BETA construct. It consists of a number of member entries, each containing information about a member – its name[22] and type, as well as the offset of the member within an object of the defining class. Our implementation is based on the pointers to member (ptm) approach rather than calculating the offsets by pointer arithmetics using the `this` pointer. The latter technique implies an overhead of both space and time, as it would be necessary to generate code for performing these calculations.

Given a feature like ptm in the `C++` language it is possible to avoid nasty implementation dependent things such as pointer arithmetics. By minimizing circumvention of the type system we also obtain greater portability. We can rely safely on the fact, that given a ptm of a class `SomeClass`, we can apply it to any object of class `SomeClass` (or any class which is publicly and unambiguously derived from

---

[20] Actually the prototypes are not "real" objects.

[21] An abbreviation for COOL ProtoType.

[22] Actually the name is unnecessary, but we have included it for debugging purposes.

class `SomeClass`) using the ptm operator. Banning the use of multiple inheritance ensures unambiguous derivation.

At one point we have to circumvent type checking by explicit type casts – we deliberately violate contravariance in the initialization of prototype objects. We have specified a set of overloaded constructors for the `COOL_PT` class to keep the `type` enumeration within the class. This way we retain readable source code and avoid conflicts with other libraries, polluting the global name space with nasty things like `#define TEXT char*`. Unfortunately `C++` does not provide an implicit conversion from a ptm of a derived class to a ptm of a base class. It is a proper choice: this would violate contravariance. The problem can only be overcome by an explicit cast. But one can beyond reasonable doubt assume that a ptm of some user defined class is implemented exactly the same way as a ptm of another user defined class. Thus we believe the portability issue [4] is covered by our approach.

```
// Some aliases, the C++ syntax for ptm's is quite unreadable
typedef int COOL::*COOL_INT;            // pointer to int member
typedef double COOL::*COOL_DOUBLE;      // pointer to double member
typedef char* COOL::*COOL_TEXT;         // pointer to char* member
...
typedef COOL* COOL::*COOL_POINTER;      // pointer to COOL* member

// Prototype class
class COOL_PT {
  friend class ClsMgr;
  friend class ObjMgr;
private:
  // Prototype information
  const char* const name; // Name of class (in header) or member
  // Type values
  enum { CHAR=1, INTEGER, LONG, TEXT, POINTER, ... } type;
  union {
    COOL_INT int_member; // Pointers to members
    COOL_DOUBLE double_member;
    COOL_TEXT text_member;
    ...
    COOL_POINTER pointer;
  };
public:
  // Constructors
  COOL_PT(const char* const); // Header constructor
  COOL_PT(const char* const, COOL_INT); // int member
  COOL_PT(const char* const, COOL_DOUBLE); // double member
  ...
  COOL_PT(const char* const, COOL_POINTER); // pointer
};
```

Fig.8: The COOL_PT Class

Actually some C++ implementations (the gnu g++ compiler for instance) have a rather relaxed attitude towards the contravariance problem, and will merely issue a warning

when a pointer to an `int` or `char*` member of a derived class (e.g. class `Node` shown in Fig.9) is used to initialize an equally typed ptm of a base class:

```
short COOL::*tag = &Node::kind; // warning: contravariance violation
char* COOL::*text = &Node::info; // warning: contravariance violation
```

But attempts to initialize e.g. a pointer from a `COOL` object to another `COOL` object with a pointer from a `Node` object to another `Node` object will fail without explicit conversion:

```
Node* Node::*aNode = &Node::next; // ok
COOL* COOL::*aCOOL = &Node::next; // error: incompatible pointer types
COOL* COOL::*anotherCOOL = (COOL_POINTER)&Node::next; // ok
```

In order to simplify things we chose always to supply the cast. We also think it is pleasant to have a description of the member types in the declaration of the prototype array, since it mostly resides in another file than the class declaration. Explicit type casts can be prone to errors, but as prototype declarations are generated automatically and don't have to be modified the risk of applying a wrong cast is minimal.

```
class Node : public COOL {
public:
  short kind;
  char* info;
  Node* next;
};

COOL_PT array[] = {
  COOL_PT("next", (COOL_POINTER)&Node::next),
  COOL_PT("kind", (COOL_SHORT)&Node::kind),
  COOL_PT("info", (COOL_TEXT)&Node::info),
};
```

Fig.9: Prototype array for a sample class

Note that the order of members is not the same in the definition of the prototype array as in the class. Since many operations are chasing pointers, it seems like a natural thing to put them at the start of the array. The layout of the prototype array shown in Fig.9 is illustrated in Fig.10. The compiler is supposed to use the conventional implementation of pointers to members as offset plus one[23] and we assume the base class (`COOL`) part of an object occupies a total of 8 bytes.

Member name	*Type of member*	*Pointer to member*
"next"	POINTER	15
"kind"	SHORT	9
"info"	TEXT	11

Fig.10: Layout of the prototype array

[23] Adding one to the offset will allow the representation of a null pointer to be zero (§8.1.2 in [ES90]).

As discussed in section 3.1, we can only refer to members declared in the class the prototype array is being constructed for. Note that the prototype arrays do not reflect inheritance. Somehow the prototype arrays that comprise to a full description of a class have to be connected. There seem to be two possibilities: One solution could be to tag the name of the base class to the prototype array and let the Class Manager resolve the dependencies. Although (almost) trivial to implement, it seems like a costly approach. Rather one could use an existing construct that somehow reflects the class hierarchy. We construct the prototype arrays in the body of the virtual function `memberinfo()`. This satisfies the requirement that the prototype array for a class *must* be declared within the scope of that class. The prototype arrays are then used for construction of `TypeInfo` objects, which take care of the ordering to reflect the inheritance path. This fulfils the extensibility requirement [3], since we don't require explicit knowledge about the base class.

4.6.2. The TypeInfo class

```
class TypeInfo {
  friend class ClsMgr;
  friend class ObjMgr;
private:
  // Implementation of run-time type information
  const TypeInfo* base; // TypeInfo of base class
  const char* const typename; // name of the class
  const COOL_PT* const array; // pointer to prototype array
  const unsigned short entries; // # of entries in array
  const COOL_GENERATOR* generator; // instance creation function
  const CID cid; // Class IDentifier
private:
  // Prevent users from copying information
  TypeInfo(const TypeInfo&);
  TypeInfo& operator=(const TypeInfo&);
public:
  TypeInfo( const char* const n,
            const COOL_PT* const a, unsigned short e,
            const COOL_GENERATOR* const g,
            const TypeInfo* b = 0 ) :
    name(n), array(a), entries(e),
    generator(g), base(b), cid(ClassManager::manage(*this)) {};
  // Operators to check relationship of classes
  int operator==(const TypeInfo& ti) const { return this == &ti; };
  int operator!=(const TypeInfo& ti) const { return this != &ti; };
  // Inspect order among TypeInfo objects - reflects inheritance
  int derived_from(const TypeInfo& ti) const;
  // Nice to have this for debugging purposes
  const char* name() const { return typename; };
};
```

Fig.11: The TypeInfo Class

The objects constructed in the `memberinfo()` function of each persistent class are unique instances of the class `TypeInfo` shown in Fig.11. Uniqueness means there is only one `TypeInfo` object for each persistent class, residing “ inside” the

`memberinfo()` function of its class. By giving the object static linkage it gets created only once at the time of the first call. The constructor of `TypeInfo` obtains a unique class identifier (CID) by registering the object in the Class Manager.

4.7. The Class Manager

When retrieving objects from the persistent store we must have a way of identifying classes to be able to decide which class to create instances of. Furthermore we have to know how to create instances of a particular class. These issues are handled by a unique object called the Class Manager.

4.7.1. Class identifiers

A simple way to identify classes is by their name. It is not very efficient, though. In this implementation, we use integers for class identifiers (CIDs). CIDs are used to obtain RTTI from the Class Manager. Upon instantiation of a `TypeInfo` object it is handed over to the Class Manager where it is provided with a unique CID for the class it represents. The Class Manager keeps the RTTI for the entire collection of persistent classes in the system.

Assigning a CID to a class could be done in many ways: One could let the programmer assign a number to each class or generate a unique value from the class name. Since we cannot decide on the order in which the type information is registered by the Class Manager assigning consecutive numbers is a bad idea – simply changing the link order of object files could break the system. We chose to maintain a little database in the Class Manager, defining a relation between class name and a CID. This way classes can be added or deleted without invalidating objects already existing in the persistent store (unless, of course, their class definition changes).

4.7.2. Generating new instances

The Class Manager must also know how to create new instances of any (persistent) class. As noted earlier it seems that there is only one “ clean” solution supported by C++. Though having a severe deficiency it is equally good as the “ dirty” one. Until further we have chosen to restrain persistent classes from having reference members and require them to have a default constructor. Currently object generation is implemented by means of a generator function calling the default constructor for its class.

4.8. The Object Manager

The actual work of the system is done in the Object Manager. It takes care of transferring objects to and from the persistent store. To make objects in the persistent store identifiable from user programs, the Object Manager has the ability to associate names (character strings) with objects. Furthermore it keeps track of persistent objects in volatile memory to prevent duplicate instantiation. A schematic overview of the Object Manager is shown in Fig.12 on page 28.

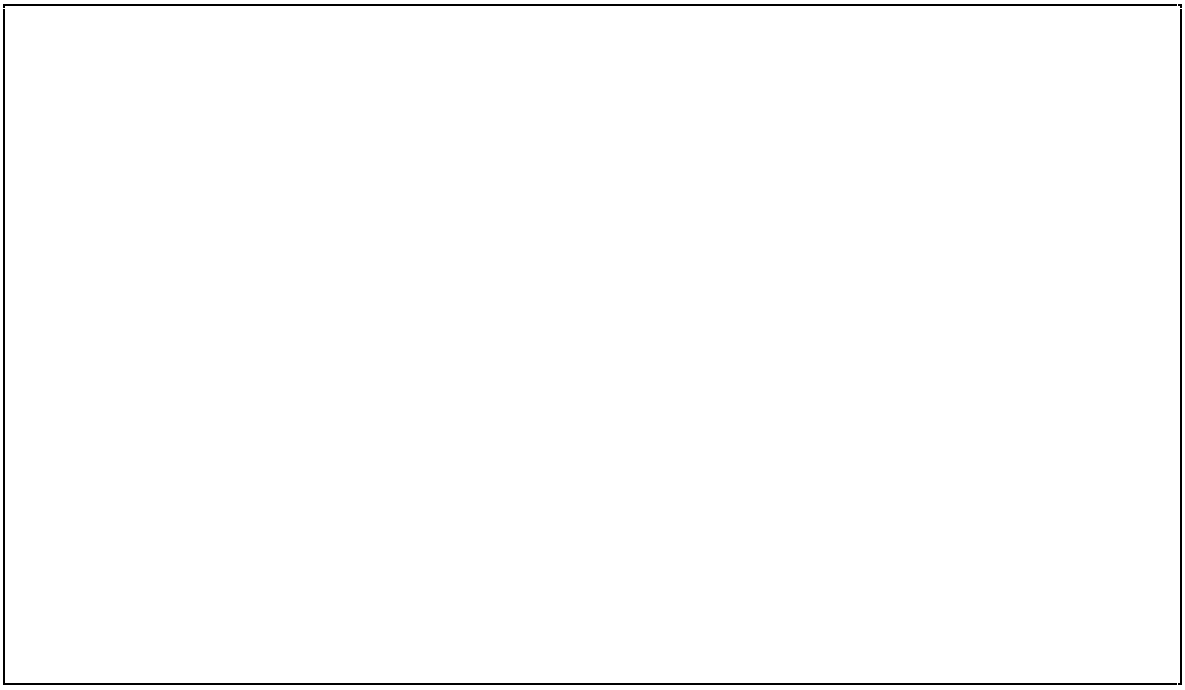

Fig.12: The Object Manager

4.8.1. Naming persistent roots

The public interface of the Object Manager provides functions to store and load objects by means of persistent roots. An object is declared to be a persistent root by associating a name with it. This method of accessing persistent objects originates from PS-ALGOL [Nikhil] where computable names (strings) can be bound to *structures*. Analogue constructs are commonly used in persistent languages. Also, practically all libraries which provide persistence for existing languages have means of association user-supplied names with objects.

For efficiency reasons the relation between names and persistent roots is maintained by an index. Another solution might have been to implement the nameserver as a persistent associative array, store it in the persistent store and retrieve it on system start-up.

A sample program using the example class from Fig.5 on page 19 illustrates the use of names:

```
#include "Person.h"

Person a, b;
a.name = "Mrs. A", b.name = "Mr. B", a.spouse = &b;
a.ssn = 139949132, b.ssn = 242334133, b.spouse = &a;
ObjectManager.putRoot("AandB", a);
...
Person* p;
if (ObjectManager.getRoot("AandB", p)) {
  // Ok, found person
  cout<<p->name<<endl;
}
```

4.8.2. Storing objects

Internally the Object Manager uses OIDs for managing objects. Whenever an object (or a reference to an object) is stored, its OID is inspected to ensure it has a valid identity. An object entering the Object Manager for the first time will have an invalid (zero) OID, and will therefore get a new globally unique OID assigned to it.

In the current implementation, objects are stored immediately when they are transferred to the Object Manager. For performance issues it might be preferable to delay the actual storing until explicitly stated or until program termination. The latter is of course inadequate when objects should be shared concurrently by different programs.

All objects reachable from the object handed to the Object Manager (*the transitive closure* of the object) get stored. The basic algorithm used in this implementation is:

```
procedure save(in object)
{
  if (object is ready) {
    allocate new buffer;
    pack object's OID and CID into buffer
    for (every prototype array in the inheritance path) {
      for (each member in the array) {
        switch (on the type of the member) {
        case POINTER:
          if (the pointer in object is non-zero) {
                pack OID and CID of the reference to into buffer;
                indicate busy;
                save(the object referred to);
                indicate ready;
          } else {
                pack zeros into buffer
          }
        default:
          pack the member into the buffer according to type;
        }
      }
    }
    save buffer with object's oid as index;
  }
}
```

The `state` flag inherited from `COOL` is used to avoid infinite recursion by setting it to `busy` whenever following a pointer. Initially all objects are `ready`, but when retrieved from the persistent store, some of the references may be proxy objects with " garbage" in their attributes. Hence we don't want to save proxy objects and overwrite their persistent representation, so we shall only deal with `ready` objects.

Normal attributes simply get stored, whereas references to other objects are translated from memory addresses to (OID, CID) pairs.

4.8.3. Object retrieval & proxy objects

The reverse mapping has to be performed on retrieval. Since it is impractical, sometimes even impossible, to transfer the entire closure of an objects references into

volatile memory there must be some way to limit the depth to which the structure is loaded. We do this by supplying a parameter to the `getRoot()` function, indicating how many levels by indirection objects should be retrieved.

This leads to an important question: What to do with references to objects not in memory? We have discussed three solutions:

[1] One could simply set unresolved references to a null (or some other unique) pointer. In C++ it is not possible to determine whether an address is valid (i.e. refers to an object) or not. One may test for null or equality with another pointer, but that's it. This lays of course a considerable burden on the programmers, since all references must be tested to check if they are valid – and explicit retrieval must be performed.

[2] The same as [1], but instead of setting unresolved references to some " strange" value, the reference set to an object of the proper class, but the objects attributes are not retrieved from the persistent store. Hence the *reference* is valid, but the *object referred to* is not. We use the term *proxy object* for such objects with invalid attributes.

[3] The same as [2], but instead of allocating an object of the proper class, a special type is used – just containing OID and type of the class. The drawback is that one would have to keep track of all unresolved references to an object and, when the object is loaded, either resolve all unresolved references or just the one dereferenced.

[4] One could keep the OID in the location where the pointer would be normally. This will cause an OS exception on run-time when we dereference the pointer and by catching that exception, we could load the associated object and resolve the OID to a memory address " on the fly" . Again all unresolved references must be kept track of.

The first suggestion is unacceptable. Consider re-saving an object containing an unresolved reference. The only place, where information about the identity of the reference is stored is in the persistent stores representation of the object. Hence we must either keep the persistent form of the object around (which cost memory) or re-read it from the persistent store (which cost time).

The second solution has several advantages over the first. For one thing, when several objects have " unresolved" references to the same object, the references are valid so they do not have to be updated when the object is loaded. Also, since references are valid, the saving algorithm can be kept very simple – just avoid storing proxy objects. It has some shortcomings too, though. If the objects are of great size, a considerable amount of memory is required for " nothing" .

As an alternative one might suggest [3], but this gives some overhead. Also, keeping track of references is in vain, since there is no way to implement a general form of reference count in C++. Consider the following:

```
Person *a, *b;
...
b = a->spouse;
```

If `a->spouse` is an unresolved reference, then we just have introduced one more, namely `b`. But this could only be registered explicitly[24], again burdening the programmer. Unfortunately, in C++ overloaded operators apply only to objects, not to pointers. Techniques have been demonstrated [Stroustrup91] allowing "smart" pointers (objects implementing pointer behaviour) to keep a reference count on them.

We think the best solution is [4] due to its transparent retrieval of objects. The ObjectStore database system uses such a technique [Lamb91] and equal experiments are being made with the BETA persistent store [EuroCoOp].

The current implementation uses [2], but changing it to [4] would only require a minor rewrite. This will be done in near future.

4.8.4. Packing and unpacking objects

When residing in the persistent store, objects are "flattened" or packed as a block of binary data. For this purpose a class `Buffer` has been implemented. Objects are packed and unpacked on a member basis. Using the RTTI of a class, the basic process is something like:

```
for (every prototype array in the inheritance path) {
  for (each member in the array) {
    switch (on the type of the member) {
    case type:
      pack or unpack the member according to type
    ...
    }
  }
}
```

Having the flattening occur in a separate class gives several advantages. It makes it possible to share objects between programs running on different architectures. All that is required is an implementation of `Buffer`, that makes use of the XDR representation.

4.8.5. Object caching

The object cache is implemented as a simple chained hash. Objects are inserted on two events: When a volatile object is made persistent (i.e. handed to the Object Manager) and when an object is loaded from the repository. Objects are automatically removed from the cache when they get deleted[25]. Prior to the load of an object, the cache is checked to determine whether the object already is active in the system. Is this the case, the load will be performed only if the object is a proxy.

Since all persistent objects are reachable from the cache it could be used as a persistent root.

---

[24] This approach can be found in ONTOS

[25] A call in the (virtual) destructor of `class COOL` takes care of that.

4.8.6. The repository

Buffers are stored in the `repository` of the Object Manager. The repository maintains a relation between OIDs and blocks of binary data. It consists of an index and a datafile. The datafile, an instance of the class `BlockStorage`, handles variable length blocks of binary data, using a best-fit algorithm to reuse the “ holes” that occur when objects are deleted.

4.9. Limitations

The implementation does not support full C++ and lacks other features desirable for a persistent store. Especially we must mention the following:

- Multiple inheritance is not supported, though support for non-virtual multiple inheritance would be straightforward to achieve.
- Can't have reference members in objects. This is a feature which is not easily overcome. Although the implementation of reference variables is the same as for pointers, and we thus could access the interiors by “ cheating” , reference members give problems in the area of object generation.
- Support for component subobjects and arrays is not implemented, but should be straightforward to do.
- There is only one Object Manager. There are no real reasons for this design, and having multiple instances could easily be achieved.
- There is no concurrent access to the repository.

5. Conclusion

In this paper we have discussed some features of C++ more or less supporting the design of a persistent store. Using some of these features, we have successfully implemented a persistent store.

Though there is good low level support in C++, features in the language, especially reference variables, are a real obstacle. We have throughout the process found ourselves locked up by features that mutually prevent each other from being circumvented (which sometimes would have been necessary).

In the implementation we have used a persistence-by-subclassing approach. This may seem as a contradiction to our statement earlier in this paper, about the merits of type-orthogonal persistence, but it is not. When we are talking about future persistent programming languages we do not mean already existing languages extended with persistence but in fact a persistent programming language build from scratch. Adding persistence to an already existing language limits one to use constructs of the original language.

6. Future work

The design of the COOL language in the EuroCoOp project will be revisited. Among other things, we would like to get better hold concept of component subobjects, since it is evidently not the same in BETA and C++. One might raise the question if composition by reference, as found in most OOPLs, is powerful enough for design in the target area of the project. Based on the final design of the COOL language, the C++/OODBI will be implemented using some of the techniques presented in this paper.

References

[Agesen89] O. Agesen et al.: *Persistent and Shared Objects in BETA*, Computer Science Department, Aarhus University, 1989, DAIMI IR-89

[Albano] A. Albano et al.: *The Implementation of Galileo's Persistent Values*, Chapter 16 in Atkinson et al. [Atkinson88]

[Arjuna] S. K. Shrivastava et al.: *The Arjuna System Programmers Guide*, Department of Computing Science, Computing Laboratory, The University, Newcastle upon Tyne, 1992

[Atkinson88] M. P. Atkinson et al.: *Data Types and Persistence*, Topics in Information Systems, Springer-Verlag, 1988

[CM] L. Cardelli and D. MacQueen: *Persistence and Type Abstraction*, Chapter 3 in Atkinson et al. [Atkinson88]

[Cockshott] W. P. Cockshott: *Addressing Mechanisms and Persistent Programming*, Chapter 16 in Atkinson et al. [Atkinson88]

[ECO-JT-91-2] J.A. Hem et al.: Workpackage WP5 Task T5.2 Deliverable D5.2: Object Oriented Database Interface, ECO-JT-91-2, Aarhus University, 1992

[ES90] M. A. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[EuroCoOp] Personal communication with members of ESPRIT Project 5303 EuroCoOp team.

[GR80] A. Goldberg and D. Robson: *Smalltalk-80: The language and its implementation*, Addison-Wesley, Reading, MA, 1980

[Harper] R. Harper: *Modules and Persistence in Standard ML*, Chapter 2 in Atkinson et al. [Atkinson88]

[Hughes91] J. G. Hughes: *Object-Oriented Databases*, Prentice Hall, 1991

[KMMPN91] B. B. Kristensen et al.: *Object-Oriented Programming In The BETA Programming Language*, DRAFT, 1991

[Lamb91] C. Lamb et al.: *The Objectstore Database System*, Communications of the ACM, Vol. 34, No. 10, 34-50, 1991

[Madsen88] O. L. Madsen: *Run-time organization for BETA*, Project Mjølner Working Note, 1988

[Mjølner91] *Persistence in BETA*, Mjølner Informatics Report MIA 91-20(0.1)

[Nikhil] R.S. Nikhil: *Functional Databases, Functional Languages*, Chapter 5 in Atkinson et al. [Atkinson88]

[ONTOS91a] ONTOS *Reference Manual*, Ontos Inc., Three Burlington Woods, Burlington, MA 01803, 1991

[ONTOS91b] ONTOS *Developers Guide*, Ontos Inc., Three Burlington Woods, Burlington, MA 01803, 1991

[Sakkinen92] M. Sakkinen: *The Darker Side of C++ Revisited*, in Structured Programming, Vol. 13, No. 4, 155-177, 1992

[SL92] B. Stroustrup and D. Lenkov: *Run-Time Type Identification for C++ (Revised yet again)*, ANSI/X3j16 document x3j16/92-0121, WG21/NO198, 1992

[Stroustrup91] B. Stroustrup: *The C++ Programming Language (Second Edition)*, Addison-Wesley, 1991

[ZM90] S. B. Zdonik and D. Maier: *Fundamentals Of Object-Oriented Databases*, Readings in Object-Oriented Database Systems, 1990

Appendix A: Source code