PERSISTENT C++ IN DOS ENVIRONMENT: C*

A Master's Thesis

Presented By

Cem EVRENDİLEK

to

the Computer Engineering Department

of Middle East Technical University

in Partial Fulfillment for the Degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

MIDDLE EAST TECHNICAL UNIVERSITY

ANKARA

September, 1992

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Alpay ANKARA
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Neşe YALABIK
Chairman of the Department

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Asuman DOĞAÇ
Supervisor

Examining Commitee in Charge:

Prof. Dr. Asuman DOĞAÇ

Assoc. Prof. Dr. Sinan NEFTÇİ

Asst. Prof. Dr. Göktürk ÜÇOLUK

Asst. Prof. Dr. Halit OĞUZTÜZÜN

Dr. Aral EGE

ABSTRACT

PERSISTENT C++ IN DOS ENVIRONMENT:C*

EVRENDİLEK, Cem

M.S. in Computer Engineering

Supervisor: Prof. Dr. Asuman DOĞAÇ

September, 1992, 143 pages

In conventional object oriented programming languages, objects are transient, that is they are destroyed upon program termination. Storing objects using explicit file access methods may cause objects having pointer fields to lose their manipulation and access semantics since the objects with different declarations may have the same storage representation. In this thesis persistence is added to C++ in DOS environment through a preprocessor and a class library such that the access and manipulation semantics of objects are preserved. The disk management of objects declared as persistent are automatically handled by the system through virtual memory management emulation. Persistence is implemented as a storage class that is completely orthogonal to type. In other words , persistency is a property of objects, not their classes. Among the existing C++ implementations,

C* requires the minimum coding effort. Furthermore objects of any complexity with arbitrary level of pointer indirections to any type of object is supported. As a result, objects are stored on disk as they are represented in memory. Language changes are kept to a minimum and upward compatibility with C++ is preserved. Both logical and physical object identifiers are supported to provide the ability to optimize the system performance.

Keywords : Persistence, object-oriented programming languages, disk management, virtual memory management.

Science Code: 619.02.02

ÖZ

DOS ORTAMI İÇİN KALICI C++ : C*

EVRENDİLEK, Cem

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı

Tez Yöneticisi : Prof. Dr. Asuman DOĞAÇ

Eylül 1992, 143 sayfa

Nesneye yönelik programlama dillerinde, nesneler geçicidir, yani bir programın sona ermesiyle yokolurlar. Nesneleri kütük erişim komutlarıyla saklamak, göstergeç alanlı nesnelerin işleme ve erişim mantıklarının kaybolmasına sebep olur, çünkü farklı tanımlı nesneler aynı depolama yapısına sahip olabilirler. Bu tezde C++'a DOS ortamında, bir önişleyici ve sınıf kütüphanesi yardımıyla, nesnelerin erişim ve işleme mantıklarını bozmayacak şekilde kalıcılık kazandırılmıştır. Kalıcı olarak tanımlanan nesnelerin disk yönetimi, görüntüsel hafıza yönetimi emulasyonuyla, sistem tarafından otomatik olarak sağlanır. Kalıcılık tiplerden tamamen bağımsız bir depo sınıfı olarak gerçekleştirilmiş, diğer bir deyişle kalıcılık nesnelerin sınıflarının değil, nesnelerin kendilerinin bir özelliği olarak gerçekleştirilmiştir. Varolan C++ gerçekleştirimleri arasında, C* en az kodlama

gayreti gerektirir. Ayrıca, herhangi bir tip ve karmaşıklıkta, istenilen sayıda yöneltimli göstergeçlere sahip nesnelere izin verilmektedir. Bunun bir sonucu olarak, nesneler diskte ana bellekte temsil edildikleri şekilde depo edilirler. Dildeki değişiklikler en azda tutulmuş ve C++ ile yukarı uyumluluk gözetilmiştir. Sisteme performansını iyileştirme yeteneği verebilmek amacıyla hem mantıksal hem de fiziksel nesne tanımlayıcılar desteklenmiştir.

Anahtar Kelimeler: Kalıcılık, nesneye yönelik programlama dili, disk yönetimi, görüntüsel hafıza yönetimi.

Bilim Dalı Sayısal Kodu: 619.02.02

ACKNOWLEDGEMENTS

I am grateful to many individuals for the cooperation, support and the encouragement they gave me while preparing this thesis. First of all, I would be honoured to present my special thanks to my supervisor Prof. Dr. Asuman Doğaç without whose help this work could have never been possible.

I am grateful to my friends, Mehmet Altınel, İlker Durusoy, Ç.Ertan Özkan, Yüksel Saygın, Tolga Geşli, İsfendiar Haghverdi, Oralp Tekuzman and Ayhan Ergül for their valuable comments on the work accomplished.

Finally, I want to thank to my family and Meral Gürdal for their support and patience during this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

There are certain principles of persistence that should govern language design as stated in [1]:

1. Persistence should be a property of arbitrary values and not limited to certain types.

2. All values should have the same rights to persistence.

3. While a value persists, so should its description(type).

The first two of these principles state that a value's persistence should be independent of its type, stated as orthogonality of persistence to type.The third principle asserts , it should not be possible to store a value as one type and subsequently read it as another. Additionally transfers between stores should be inferred from the operations on data. Therefore the languages providing longevity for values of all types without requiring explicit organization or data movement by the programmer are called persistent programming languages [1]. Discovering engineering techniques to support an arbitrary persistent structure is a difficult task.

In a persistent language variables declared as persistent retain their values from one program run to the next. Persistency has been added to C++ in several systems [2], [3], [4], [5], [6], [7]. There are basically two approaches to adding persistence to object-oriented programming languages. In one approach data is moved between memory and disk through explicit commands to read and to write. In the other approach, for the objects defined as persistent, no explicit read or write statements are necessary.

In implementing C* we have taken the second approach where no explicit read or write statements are necessary for persistent objects. The other characteristics of C* are that persistence is completely orthogonal to type and operator overloading is used for persistent objects.

C* is developed as a part of the MOODS Project under development at the Software Research and Development Center of TÜBİTAK at the Middle East Technical University (METU).

The goal of MOODS (METU Object Oriented Database Management System) is to build an object oriented database management system (OODBMS) to support engineering applications. For the time being two prototype systems are being developed, one on personal computers in DOS environment, and the other in UNIX environment.

Two approaches are being followed in the development of OODBMSs:

a. Starting with a relational DBMS and adding to it object oriented features(e.g. POSTGRES).

b. Changing an object oriented programming language so that it becomes an OODBMS (e.g. GEMSTONE, O2).

The second approach is being followed the MOODS Project. C++ is chosen as the language to be enhanced since it is the most widely accepted object oriented programming language. The first step of the enhancement process is adding persistence to the selected language. So, new statements (e.g. declaration of persistent variables, opening and closing databases, accessing to the elements of a variable) are added to C++ [8].

The overall system works as follows. A program written in persistent C++ is passed through a precompilation phase, in which the statements related to persistence are converted into their corresponding storage system calls. Then, the precompiled program is compiled and linked with C* Storage System [9] by the Borland C++ version 3.0. Note that DOS places restriction on the amount of main memory available to an application making it impossible to deal with large objects greater than available memory. For this reason, we have implemented the virtual memory emulation through the C* Storage System.

The contributions of C* are as follows:

1. C* requires the minimum coding effort to develop an application in persistent C++. The only thing necessary is to declare the persistent objects as persistent. The advantage of minimum coding effort becomes more apparent when porting an existing application to a persistent C++ environment. In order to port an existing C++ application to E [6], db type classes need to be defined for persistent objects. Furthermore, it is necessary to find out all persistent to transient pointer assignments, and replace the definition of those transient pointers with db counter parts. In order to carry an existing application to O++ [2] , it is necessary, first of all, define pointers to persistent objects since the only way to reference persistent objects is through pointers. Then it is necessary to find out and replace all occurrences of persistent object names with references. Furthermore it is necessary to change the pointer fields of types that have persistent and transient instances, to dual pointers. In O2 when porting an application developed in C++ to O2 environment, "import from C++" is used. The effect of this utility is to generate read and write methods for each class and also to generate the corresponding persistent pointer class. Then the user must replace declarations of pointers by persistent pointer declarations where necessary.

In C*, the code is preprocessed to perform necessary modifications transparently to the user. This gives the system the power of detecting the transient counterparts of types of persistent objects, or their data members which are declared as pointers, so that pointer swizzling upon dereference can be performed through the overloaded operators. Apart from being easy to implement, this approach gives the user a uniform view of data types since there is no syntactic difference between pointer types. Operationally there are two kinds of pointers, namely, persistent and transient pointers. In the system, since it is allowed to use persistent and transient pointers within the same expression, the operators with operands from different storage classes, i.e., persistent or transient, are also overloaded. This approach while providing the user a uniform view of pointer types and thus keeping the language changes at the absolute minimum, prevents the unnecessary runtime checks on transient pointers. The pointer swizzling is performed within the buffer manager which is invoked by the overloaded operators operating on both persistent pointers and transient pointers used with persistent objects.

2. In C* objects do not lose their manipulation and access semantics because the objects are stored on disk as they are stored in memory and the information captured about their structures is preserved. In C* when a call is issued to the buffer manager, all the information relating to the size and type of the object, and the next operation on it are known.

3. C* does not force the user to use some form of collections like sets, lists, tuples, etc. in storing persistent objects, rather objects of any complexity with arbitrary level of pointer indirections to any type of object is supported. In O2, when a C++ object that has data members with more one than level of pointer

indirections is to be made persistent, such data members have to be modelled using list or set structures supplied by O2.

4. Although there are several other persistent C++ implementations, the implementation technique of C* is original in that it provides user transparent type modifications and uses operator overloading extensively in realizing persistency .

5. Both logical and physical object identifiers are supported. Logical object identifiers provide for dynamic clustering and physical object identifiers provide for fast access until the first object move. Supporting both type of object identifiers provide the user with the ability to implement the needed functionality and to have control in clustering to achieve higher system performance.

C* is implemented in DOS environment as a preprocessor to Borland C++ Version 3.0. To the best of our knowledge this is the first persistent C++ implementation on DOS.

The work is described in five chapters.

In CHAPTER II the previous work on adding persistency to programming languages is summarized, with special emphasis on persistent C++

implementations. The systems summarized are E, O++, O2, Pascal/R, PS-Algol, and ObjectStore. In CHAPTER III, the implementation details of C* is presented.

The examples to clarify the implementation details are given in CHAPTER IV. Finally, CHAPTER V contains the future work and the conclusions.

CHAPTER II

PREVIOUS WORK

Some persistent programming languages will be summarized in this section. Among those some have object oriented features, and some do not.

Pascal/R [10] attempts to integrate the general-purpose programming language Pascal with the relation data type as closely as possible. Consequently Pascal/R takes advantage of the similarity in structure between the tuples of a relation and the Pascal record type. In Pascal/R the tuple is identified with the record, and the relation type is introduced as a constructor similar to the existing set type. The definition of a relation type requires two parameters: the type of the record which defines the structure of the tuple of the relation, and the subset of that record's fields that constitute the primary key.

Databases are defined by means of the "database " constructor which is parameterized in a manner similar to the record type, except that all its components must be of type relation. A database variable may then be declared in the program and its persistence is insured by including it in the program heading. Only variables of type database may persist. Variables of type relation declared outside of a database constructor are considered volatile.

The structure of the database as expressed in Pascal/R is identical to its relational schema hence solving the impedence mismatch problem between the programming languages and the database systems. Pascal/R provides specialized operations for accessing and manipulating relations which are based on the tuple-oriented relational calculus.

In PS-algol, persistence is orthogonal to type [11]. Unlike Pascal/R, PS-algol does not involve the addition of new data structures to a language, but rather provides persistence as a property applicable to all existing data structures. Languages like Pascal/R go some way towards alleviating the programmer's burden to transfer data between the database storage system and the program. PS-algol adopts the principle of data type completeness, thus all data types have the property of persistence.

The language is strongly typed , since the access to a field of a structure, with the construct pointer-expression (field-name) is checked for type consistency. This can be achieved because the field names must be unique in the current scope.

In [12] two approaches to adding persistence to C++ are discussed. In the first approach persistence is defined at the level of the language as a storage class (SC) and in the second approach persistence is defined on top of the language, through inheritance from a virtual base class (VB).

In VB approach, there is a base class that defines virtual functions for creating, deleting, and referencing persistent objects. Any class that has persistent instances includes this class in its list of superclasses. The implementor of each such class must then supply the actual code to implement each of these virtual functions.

SC approach is simpler, faster, has lower resident memory requirements, and is upward compatible with C++. VB approach is more complex, takes more resident memory, is slower at dereference time, and places restrictions on the way the programmer can use C++ in dealing with persistent objects[12].

The disadvantages of VB approach over SC approach are as follows [12]:

1. Complexity. The programmer has to write construction, destruction, fetch and -> functions for each class defined in her application.

2. If a normal C++ object A refers to an object B as the value of a member named parent, an attempt to get from A to B using the expression A->parent will be compiled into single instruction that dereferences a virtual memory pointer using the hardware MMU. If A is a persistent object in a VB-system, the same reference would be expanded into a code sequence that follows the initial pointer to a surrogate object, checks the surrogate to see whether the object B has already been mapped into virtual memory, and if so goes there; if

not, calls a fetch method defined on the class of B; this method or a type-specific method that overrides it, will locate the object on secondary storage, allocate storage for its virtual memory representation, and construct that representation from the on-disk representation of the object. This reduces the performance of the system.

3. In VB approach, C++ full upward compatibility is lost.

The storage overhead for objects in VB approach is also high. In a system designed to work with CFRONT, each instance of a class that inherits from the base class will carry a two-word Virtual Table (Vtbl) entry for each node along every path to the base class. This is a function of the way CFRONT allocates Vtbl entries in the case of multiple inheritance, and virtual functions.

In summary, performance for interactive applications is slow in VB approach. The dereference operator, instead of one virtual memory operation, requires several instructions. The notification of the storage system is also required in VB approach so that the changes can be saved.

ObjectStore uses SC apoproach to persistence. The object identifier (OID) of each persistent object is the physical disk address of the object.

In ObjectStore [5] the attributes of attributes or attributes whose value is determined on demand by a function are not allowed, so there is no syntactic distinction between an attribute in the public portion of a C++ class definition and literal valued data member [12].

In ObjectStore compile-time type-checking of C++ applies to persistent data as well as transient data. It provides looping constructs to iterate over sets. ObjectStore's performance goals demand that once the target object has been retrieved from the database , subsequent references should be just as fast as dereferencing an ordinary pointer in the language. The virtual memory system allows ObjectStore to set the protection for any page of virtual memory to no access, read only, or read/write. All virtual memory mapping and address space manipulation in the application is handled by the operating system under the direction of ObjectStore, using normal system calls. Databases can be stored either of two: within files provided by the operating system's file system, or within partitions of disks, using ObjectStore's own file system.

The collections in ObjectStore are often not known by name. They may be pointed at, or result from the evaluation of an expression. This means that multiple strategies must be generated, with the final selection left until the moment the collection being queried is known, and the query is to be run.

Borland C++ version 3 [13] supports persistency as inherited from a virtual base class for Turbo Vision objects. In this work approximately sixteen bytes of storage is used on the average for each object of a persistent class. There is no buffering effort. Objects are read and written to streams on demand. Therefore each time an object is referenced necessary conversions, checks, and allocations are performed to bring it to memory and back to disk. Since the persistent objects have identifiers as unsigned integers, when pointers are used the alignment problem arises.

In [3], [6], [7], C++ is extended to E programming language. In E which uses SC approach, all primitive types and type constructors of C++ have an analogous db form. The interpreter provides load, store and pointer manipulation functions for persistent objects.

Although persistence is orthogonal to data type, the language is not fully uniform. E uses EXODUS Storage Manager. Each object is named by its physical object identifier(OID). Two basic operations supplied by the storage manager, read and release, provide access to objects. Data requested in a read call is pinned in the buffer pool until it is released.

In E, if a variable is declared to have the persistent storage class,that variable's name is a persistent handle. Each type that can be constructed in C++

has a corresponding db type. An object may be persistent only if it is of db type. Because every object in E is typed, the compiler can statically distinguish references to db type objects from references to C++ objects. E recognizes two kinds of pointers , normal C++ pointers and pointers to db types. The latter is in the form of OID, offset pair called DBREF. Implementation of E is an extended version of the AT&T C++ compiler. The majority of the work in implementing E lies in extending the source code for cfront into an E-to-C translator called efront.

In E db types are used to enable transient pointers to point to persistent objects. Otherwise it would not be possible to trace transient pointers because they may be pointing to persistent objects .

O2 Object-Oriented Database System described in [4] can also be used as a persistent C++ environment. In O2, when an object is made persistent, so are its component objects (recursively), freeing the programmer from performing this task explicitly. Physical object identifiers are used for persistent objects. When an O2 object is exported to C++, O2 provides two methods and a twin class called O2_k for each generated class k, to read or write C++ objects from or into database. The read method is transparently called when accessing the data for the first time, while the write method is transparently called at least commit time.

Note that when objects have fields declared such as char * a; then such fields of objects can grow dynamically. Thus it becomes impossible to bring the objects containing such fields in memory at once since the data fields pointed to by such objects might have grown and thus forward marked. This necessitates the buffer manager to keep track of these operations for O2 pointers.

If an application is developed in C++, O2 can be used to render some objects of C++ classes persistent. C++ utility "import from" is provided for this purpose, parses a C++ class and generates two new methods for it: O2_read and O2_write. The C++ programmer then must adapt the C++ application by replacing declarations of pointers by declarations of persistent pointers when necessary.

CO_2 is a strongly typed language: every object is an instance of a type and all message expressions are statically type-checked. A type has a structure which is either basic, tuple or set. CO_2 is restricted to single inheritance. Instances of persistent types may be persistent or not. Thus only persistent names are type names and only handle on the persistent space is through the type names. The instances of a type can only be manipulated using the methods associated to this type. Methods of persistent types are separated and compiled exclusively.

In O++ [2], C++ is made persistent as a storage class. O++ extends C++ by providing facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

Persistent storage operators pnew and pdelete are used instead of the heap operators new and delete. Components of persistent objects are referenced like the components of volatile objects. It is stated in [2] that having only ordinary pointers has a problem associated with it: If a class is used to build a linked data structure, then the same class cannot be used to create the data structure both in volatile memory and in persistent store and functions taking as arguments pointers to either volatile or persistent objects cannot be written. This is due to the fact that at run time, it should be detected whether a pointer is pointing to a volatile or persistent object.

O++ uses three pointers: pointers to persistent objects, pointers to volatile objects (ordinary C or C++ pointers), and dual pointers.

In [2] two other alternatives for pointers are also considered: only one type of pointers that can point to both volatile and persistent objects, and two types of pointers, an ordinary pointer that can only point to volatile objects and another that can only point either to a volatile or a persistent object. The latter is called dual pointer. The first approach has the advantages that all code will work for both volatile and persistent objects, types are strictly orthogonal to persistence. The disadvantage on the other hand is that a run-time check must be made to determine whether a pointer refers to a volatile or a persistent object. This check imposes a run-time penalty for transient objects. The two pointer alternative is abandoned on the grounds that the separate pointers for persistent objects leads to better program readability.

CHAPTER III

IMPLEMENTATION OF C*

3.1 Some Considerations about C++

C++ has the computational power of C and also the following object-oriented characteristics: multiple inheritance, encapsulation, overloading, late binding. The assignment operation may carry different semantics for object sharing in different programming languages. From the point of assignment operation, this is termed as having different copy semantics. The copy semantics of C++ is different from that of Smalltalk. As an example if a list list1 is concatenated with a list list2 in Smalltalk to obtain a list list3, whenever list list1 changes so does list3. However in C++, list3 stays the same when list1 changes.

C++'s notion of an object is somewhat weaker than the formal semantic models used by object oriented databases.There is a formal distinction between a relationship and an object-valued C++ pointer. The deletion semantics of the two are different. A pointer takes as its value an address. A relationship takes as its value an object. In the case of a relationship, if you delete the object referred to, a subsequent attempt to cross the relationship will cause an exception to be raised. Relationships are always bidirectionally traversable. This is not true with a pointer [12].

C++ allows methods and operators to be overloaded. This presents the ability to use the same methods and operators with different semantics for different data types. Since we did not have access to C++ compiler source code, to develop a prototype rapidly, we have chosen method and operator overloading to access the persistent and the transient objects uniformly using the same operators and methods. Overloading has two problems at the implementation level:

A performance degradation occurs because for each overloaded operator it is necessary to find out the correct code to be executed. This performance degradation can be prevented by pasting the necessary code at compile time which can be achieved by modifying the C++ compiler.

And secondly, not every operator can be overloaded and also compile time information is necessary for overloaded operators to perform correctly at run time. We overcame this problem by writing a preprocessor augmented with a very simple parser, called Class Parser.

It is necessary to overload arithmetic operators (+, -, ...), reference operator (&), dereference operator (*), component operators (->, .), scope resolution operator (::), assignment operator (=), subscript operator ([]), if else operator (:?). However :?, ::, . operators can not be overloaded in C++. Scope resolution operator (::) is handled by the class parser through flattening such that the inherited attributes are carried down the class hierarchy as explained in section 3. 9. Dot operator (.) is handled by the preprocessor by replacing it with -> context sensitively. :? is not used in our implementation.

3.2. General Structure of the System

The general structure of the implementation is as shown in Figure 3.1. The source file in C++ programming language augmented with our keywords is given to the Preprocessor. The Preprocessor detects the class declarations and passes them to the Class Parser. The Class Parser in turn determines offsets and structures of fields, inheritance topology, access specifiers (public, protected, private) for the fields and methods of the persistent objects. For each class that has a persistent instance, a modified copy of this class is generated by the preprocessor. Thus the detection of the manipulations of the persistent objects in the expressions are left to the compiler itself due to the overloadings defined in a file called overloader. Environment files created by the preprocessor store the necessary information for persistent objects to be used during preprocessing and at run time by the system.

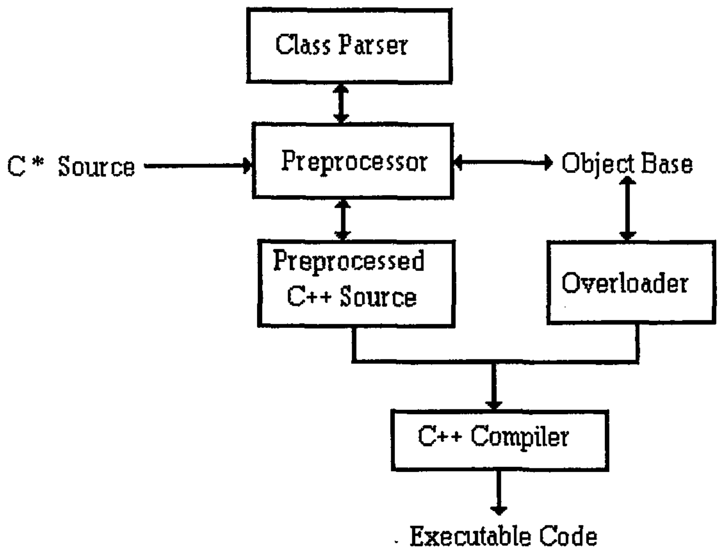

Figure 3.1. General Overview of Implementation

3. 3. The Design Philosophy

The first design decision is to implement the persistency whether as a storage class or as a virtual base class. In virtual base class approach, any class that has persistent objects includes this class in its list of superclasses, and inherits the requisite operations. The implementor of each such method must then supply the actual code to implement each of these virtual functions. This approach, although is easier to implement, has several drawbacks [12]. Thus we have chosen to implement persistency as a storage class.

Another design decision is to use either logical object identifiers or physical disk addresses as object oriented identifiers (OID). The physical OID mechanism as in [14], [15] addresses objects directly through their VMAs and stores forward markers in the original place of the objects when the VMA of the object changes. The object access with physical OID is faster until the first object move. Another drawback of this technique is that dynamic clustering is impossible [16]. Furthermore, physical OID cannot be allocated without object space allocation. The disadvantages of a logical OID are that it introduces a level of indirection in accessing the objects and object table may grow so large that it does not fit into memory.

A hybrid OID mechanism has been implemented in C* where persistent objects have logical OIDs whereas values are referenced directly by their physical addresses. This approach is motivated by observing the fact that clustering becomes critical for the performance of object- oriented databases for engineering applications. Thus frequently accessed objects are modelled as persistent objects and in this way accessed through object-table enabling dynamic clustering. Less frequently accessed objects are accessed through physical object identifiers to keep the object table size reasonable. Objects that are accessed through physical object identifiers are termed as persistent values. Note that persistent value terminology in C* is different than that of O2 where persistent values have copy semantics.

To summarize, this approach has the following advantages :

1. The size of object table is reduced since persistent value references

do not go through object table.

2. Persistent objects may be accessed faster since they are not forward marked.

3. Dynamic clustering of persistent objects are possible. Thus when clustering of certain objects are essential, they can be declared as persistent objects not as values. This feature provides a level of control over clustering.

3. 4. Object Base

The environment files, i.e., the object table, the oid-generator, the structure file, the method file, the free space file, and the object store constitute the object base.

Organized collections of arbitrary types of semantically related persistent objects and persistent values are grouped in storage spaces called object stores.

3. 4. 1 Environment Files

1. OID File

For each object base, there is a corresponding OID file. This file gives the unique logical object identifier for each persistent instance related to an object base.

2. TAB File

This file stores the object table. An object table is maintained in order to keep the associations between the logical object identifiers and the physical addresses of objects. The object table is organized as a linear hash table.

The structure of the object table is given in figure 3.2.

LOID is a unique identifier for the persistent object stored in the corresponding object store. Page Number and Slot Number together gives the byte offset of this object in the object store. Reference Count denotes the number of objects pointing to this object. This value is used for deciding whether an object is to be deleted or not when an unpersist command is issued. If this count is greater than zero, this command has no effect. We have chosen to implement it this way because otherwise a performance degradation would occur due to the fact that an extra memory reference is necessary for each persistent object access.

LOID	Page Number	Slot Number	Reference Count
...	...	...	...

Figure 3.2. Object Table

3. DBF File

This is object store file. It may consist of arbitrary number of objects of arbitrary types. The representation of objects in this file are very similar to that of their memory representation. So the fields of objects on disk are stored consecutively. The size of a persistent object is fixed. However the persistent values pointed to by a field of a persistent object may grow or shrink dynamically. So a length field is stored with persistent values. The portion of the object store corresponding to an arbitrary object can be viewed as in Figure 3.5. Since persistent values are not referenced through the object table, if they grow out of available free space at their current location, they are forward marked. Such an object representation preserves the access semantics. Consider the following example whose memory representation is given in Figure 3.3.

```
class Class{
public:
   char **names;
   int avg;
} Class_of_92;
```

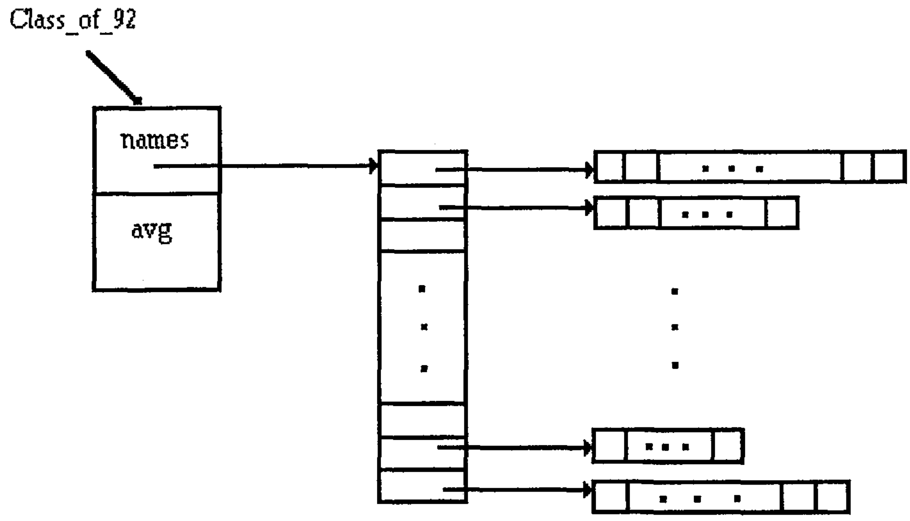

Figure 3.3. Memory Representation of Class_of_92

In writing this object to disk, a C++ programmer must maintain the number of students in the class and also the length information for the name of each student and may store this data consecutively as shown in Figure 3.4. C* relieves the programmer from this burden, persistent declared objects are automatically stored by the system as shown in Figure 3.5. Thus access semantics is preserved.

4. FRE File

This file is for keeping track of free space in object store. The free list kept in this file is updated on demand and compaction is performed dynamically. Free list searching algorithm may be specified parametrically.

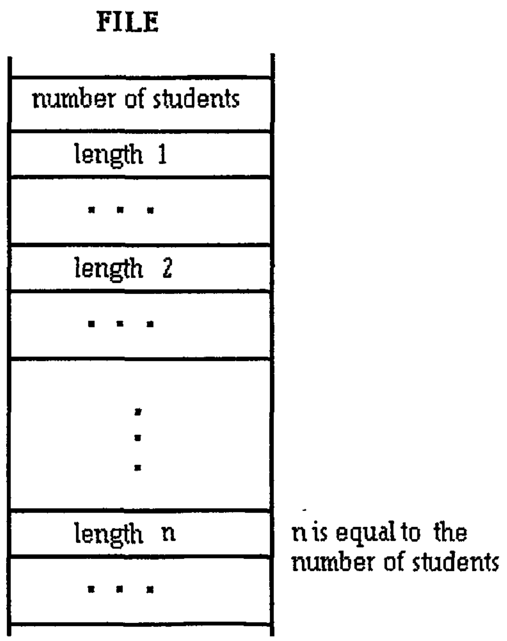

Figure 3.4. A Possible File Representation of Class_of_92

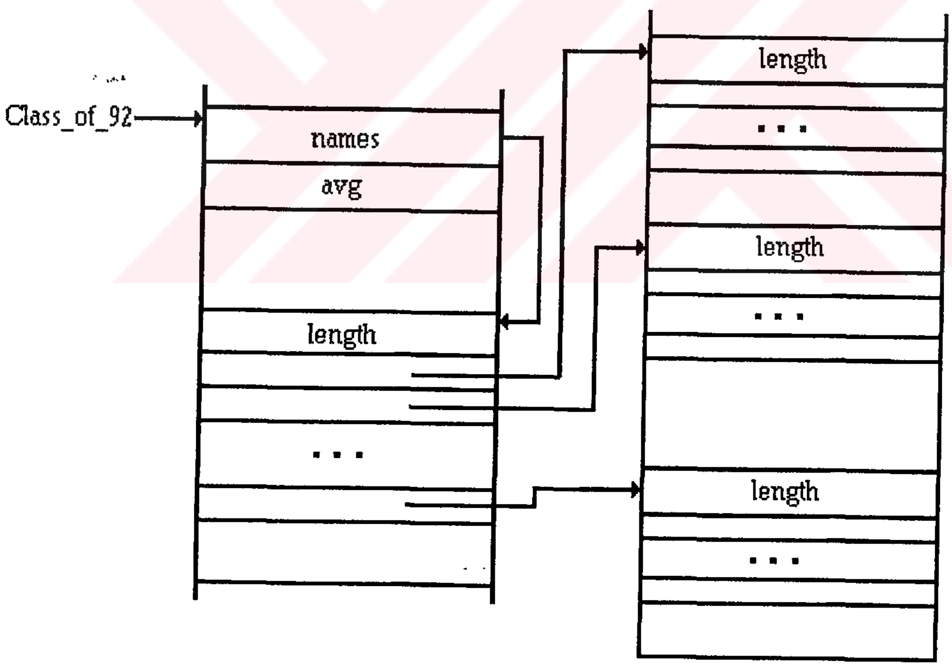

Figure 3.5 . C* Object Representation of Class_of_92

5. STR File

This file is for keeping the parsed structure of types of persistent objects.

6. FUN File

It keeps the methods related to an object base.

7. PVD File

In order to keep the association between persistent variable names and the object identifiers they correspond to, C* Storage System uses a linear hash index on variable names.

The key of the hash table contains the scope information in addition to the variable names.Program name, function name, and block indicator are used in the representation of scope information. So variables having the same name in different parts of a program do not conflict with each other.

3. 5. Forward Marking in C*

Forward marking is used in storage systems that use physical addresses to access the data. Forward marking handles the relocation of objects and avoids dangling pointers and thus facilitates object sharing.

When a persistent value grows too big to fit the available free space following its current location, it may be moved to another location, and a marker is left in its place. The marker will be the address of the new location. If the persistent value, again outgrows its new home, there is no need to leave another marker behind as the only reference to its current physical location is in the original marker. It is this marker that is updated to reflect the new location of the persistent value. Since a persistent value is referenced only by its original physical address it can be placed anywhere within the object store due to forward marking. With forward marking two disk accesses are required in the worst case.

In C*, access semantics captured for multilevel pointer indirections are preserved on disk so that each such indirection is represented as a different persistent-value address, avoiding the relocation of the whole persistent value when the size of some part of it grows. In such a case it is sufficient to forward mark only the portion of the data at the related pointer indirection level.

To clarify the point consider the Class_of_92 example given in Figure 3.5. If a name of a student gets longer, and needs to be relocated, it is necessary to relocate only the name of this student. However if the storage structure were as in Figure 3.4, the whole Class_of_92 data would have to be moved.

3. 6. Storage Management in C*

C* Storage System [9] is composed of the following parts:

1. Buffer manager.

2. Disk manager.

1 . Buffer Manager

In order to facilitate the exchange of data between disk and main memory, the database and buffer are divided into pages of equal size.

Buffer table keeps information about each page in the buffer pool. Normally, a page can be checked whether it is already in memory by searching through this table. In order to improve the search time for a specific page, a hash index is coupled with the buffer table. Chaining is used for resolving collisions. There is no need to use linear hashing since the size of the buffer table is fixed. The structures of the hash table, the buffer table and the user descriptors table are given in Figures 3.6, 3.7 and 3.8 respectively. In Figure 3.6 File Id corresponds to the identifier of a file. This field is required since there may be more than one file open at the same time, thus pages from different files may exist in memory at the same time. Page No is the page number of the page within the corresponding file. Buffer Frame gives the page number in the buffer pool in which the page is located. Lastly, Next field is the pointer to another hash entry. This field is null unless a collision occurs. Instead of adding a new hash table, the buffer table could be organized as a hash table. But, in such a case its size would be larger, and this would affect the performance of direct search. The user descriptors table keeps information about which pages are allocated to an object pointer, the file identifier of the object, and the page and buffer numbers of the object.

In Figure 3.7 File Id and Page No are used for the same reason as discussed above. Time Stamp is the

File Id	Page No	Buffer Frame	Next
...	...	...	...

Figure 3.6. Hash Table

time of the last access to the page. This field is used by the page replacement algorithm. Dirty flag indicates whether the page has been updated or not. If it is updated, it must be written back before being swapped out. Finally, hint shows the buffered page's level of importance. A hint is used to keep an important page in the buffer pool that might otherwise be easily swapped out due to the LRU strategy. There is a corresponding entry in the buffer table for each buffer page. The first record of the buffer table gives information about the first buffer page, the second one for the second buffer page and so on. The lock field indicates whether the page is locked or not. Locked pages are not swapped out unless they are unlocked.

File Id	Page No	Time Stamp	Dirty	Hint	lock
...	...	...	...	...	...

Figure 3.7. Buffer Table

The user descriptors table is illustrated in Figure 3.8. The user may fetch all or part of a small or a large object. The information about this is maintained in the user descriptors table. If the object to be retrieved is a large one then a chunk which is composed of a minimum number of contiguous buffer pages is allocated. P.Cnt and B.Cnt fields show the number of disk pages and buffer pages respectively. Most of the time they have different values since an object is represented as a contiguous byte stream in memory while a large object is represented as a B^+ tree where data is stored in the leaves that are not completely full. Offset field gives the offset of the object within the page. O.Ptr. is a pointer to the object in the buffer pool. PageInfo stores the page number and byte count on the page. This field is used when a portion of a large object is retrieved.

FileId	BufNo	size	P.Cnt	B.Cnt	O.Ptr.	Offset	PageInfo.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

Figure 3.8. User Descriptors Table

In C* Storage System, a variation of LRU is LRU with hints [14] is used as a page replacement policy. Hints are used to indicate the importance level of pages. A low hint means that the buffer page is not important and if it is swapped out the performance will not be affected. The low hint is specified for pages that are randomly accessed. A mid hint implies that the page has some importance and should not be swapped out if there are less important pages available for replacement. A page with a high hint is replaced only as a last resort. B^+-tree root pages receive high hints. When choosing a page for replacement, the buffer manager looks for the buffer page with the lowest hint and the oldest timestamp among those pages that are not locked. Timestamp is maintained in a global variable, and it is incremented with every reference to that variable. The variable is referenced each time a new timestamp is required. In addition to the hint mechanism, page locking is available in C* Storage System. When a user is returned a pointer to an object, the buffer page on which the object resides is automatically locked. Freeing a lock on a page is the user's responsibility.

2. Disk Manager

C* Storage System is designed to work with multiple databases. These databases are actually operating system files (DOS files), and are manipulated as databases. Hence by giving the appropriate path, a database can be created or opened at any disk drive and in any directory.

A database file is divided into extents where the first extent keeps information about the database (e.g. current extent number, free list of the current extent, boundary value for the object table). Initially a database consists of two extents, the first one for saving database information and the second one for storing objects. When space is exhausted in the second extent, a new extent is added to the database.

An extent is composed of a predefined number of pages. The first few pages are reserved for storing the free space array of the extent. This array has the same number of entries as the number of pages in an extent. And each entry shows the free space inside the corresponding page. Note that the first few elements of the free space array have the value of zero, since the first few pages are reserved for storing the free space array and they have no free space.

In C* Storage System there are two types of pages : data pages and slotted pages. A data page is a leaf node of a large object that is represented as a B^+ tree and contains only data, whereas a slotted page has a slot array and a data part.

Associated with a large storage object data structure are algorithms to search for a range of bytes, to insert a sequence of bytes at a given point in the object, to append a sequence of bytes to the end of the object, and to delete a sequence of bytes from a given point in the object.

3. 7. Preprocessor

3. 7. 1 C* Clauses

1. object_base <pob-name> ;

This statement declares a handle for an object base. This logical handle is then used for associating a physical path name through open and create statements. This clause also provides a potential measure for the number of different object bases to be manipulated in a program.

2. create <pob-name> (pathname)

This statement creates and opens a new object base with the object base handle pob-name and an associated pathname and makes the necessary initializations for the environment files.

3. open <pob-name> (pathname)

This statement opens an existing object base with the associated pathname, and it also assigns to pob-name a real object base handle.

Create and open statements are also used to bind the logical pob-name to physical pathname. As a result of these operations, the preprocessor generates files to keep the information about the current object base.

4. close <pob-name>

This clause closes the object base associated with handle pob-name.

5. persistent <pob-name> [class-name] variable-name

This clause declares a persistent object pointed to by the variable- name. class-name is optional in this clause. If it is not given, preprocessor will expect the variable-name to be previously bounded to a class-name. In case object base has been created previously, the necessary predeclarations are obtained by the preprocessor from the object base.

6. unpersist <pob-name> variable-name

Each object table entry contains a reference count field denoting the number of persistent objects pointing to this object (see section 3. 4. 1). This clause, after checking Reference Count field, deallocates the space for the persistent object pointed by variable-name from the object base with object base handle pob-name.

7. new_instance <db-name> [class-name] variable-name

This statement allocates space for the object pointed by a variable name. The objects with the same variable name which are created through the new_instance statement are considered as the elements of an implicit set.

8. access <pob-name> [class-name] variable-name seq-no

This statement accesses a persistent object in the object base whose handle is given in pob-name. Different persistent objects may share the same variable-name. The objects sharing the same variable-name are treated as a list. seq-no provides an index to this list. All the variable names related to an object base are stored as a table indexed on the variable name.

9. delete_function <db-name> class-name::function-name(par-list)

This statement deletes a method of a class from the object base with the associated database handle db-name. The formal parameter list par- list should exactly match with its stored definition.

3. 7. 2 Translations Performed by the Preprocessor

In our implementation there are persistent values and persistent objects. They both may be arbitrarily complex. So it is necessary to find out a way to handle them at run time without a detailed parsing process. Our solution is to modify those persistent instances such that they attain different types from transient instances of the same class at run time. Thus it is possible to overload the operators of persistent objects.

In fact what is accomplished is to provide a general list structure which contains either a set of object identifiers or persistent values at the leaves without losing the type information related to them.

The preprocessor collects the set of types of objects declared as persistent into a set called SPOT(Set of Persistent Object Types), and passes this to the class parser together with the declarations. Class parser in turn creates the copies of the declarations in set SPOT and parses them. The preprocessor modifies the types and the types of fields within them as described below:

A) The modification rules for fields of types in SPOT

Atomic data types of C++ such as int, char, float, ... have their persistent counterpart classes pint, pchar, pfloat,...respectively defined in the overloader. Class parser modifies the copies of the types in SPOT such that atomic data types are replaced by their counterparts.

rule 1: int, float, char, ... -> pint,pfloat,pchar, ...

The pointer fields of types in SPOT pointing to atomic types are modified in a different way. For

atomic_type *... name;

the field declaration is translated into

patomic_typeptrn name;

where n is the number of pointer indirections. For example,

char ** a;

declaration becomes

pcharptr2 a;

Overloaded operators for such classes are automatically generated and inserted into the resulting C++ code for n greater than or equal to 2.

For fields containing class, structure, or union class-keys, their subfields which conform to the above mentioned cases are modified till no such fields are left.

rule 2: int *.... = p int ptr n

float * ... = p float ptr n

char *.... = p charptr n

rule 3:

Each field of all persistent objects is checked against the types of other persistent objects in SPOT.

Assume a field X has a structured (struct - class - union) type.

case 1:

X does not contain pointer indirections ,i.e., it is just a name.

case 1.1 X is in SPOT

case 1.2 X is not in SPOT

In both of the cases X is a component of this object and can not be used to share other values. So space is allocated within the body of the object for the fields of X.

case 2:

X is a pointer of order n. Find all types w*... in SPOT and choose the one with the minimum number of asterisks, denoted by m.

case 2.1 X in SPOT

Suppose X is w *... where the number of asterisks is n and there is a type in SPOT in the form w *... where the number of asterisks is m.

case 2.1.1 n >= m

For this case X becomes OID"n-m" < OID"m" <w> > where OID"k" <z> has been implemented as;

OID"k" <z> = { set of OID"k-1" <z> , k>1
set of OID's of type <z> , k=1

OID0 <z> is <z> itself.

The problem with this template notation is that, when n=2*m, the interior class becomes equal to the exterior one, and results in the infinite recursive replacement of the same type definition, which is an error. To handle this situation, the interior type is replaced with another template type STR"m" <w>, the structure type. The definition of STR"k" <z> is;

$$\text{STR"k"} <z> = \begin{cases} \text{set of STR"k-1"} <z>, & k>1 \\ \text{set of type} <z>, & k=1 \end{cases}$$

case 2.1.2 When n < m, no modifications are necessary concerning OID. The declaration becomes STR"n", in other words X is replaced by STR"n". In this case, it is obvious that there are no objects in the object table that can be encapsulated in the definition of X.

To clarify the point, consider the following example.

```
Class X{
        w*a;
};
w**b;
w c;
```

In this case, field a is declared as OID1<w> a, when SPOT={X, w**, w}.

case 2.2 If X is not in SPOT, case 2.1.2 holds.

There is a restriction in current implementation of C* for the types of persistent objects in SPOT: The types in SPOT are simple class name identifiers. This restriction forces the complex object types to be encapsulated.

Thus the type of each persistent object is modified to have fields of the form:

OID"m" <STR"n" <w> >

or

STR"k" <z>

or

p"type"ptrk

or

p"type" which are called fundamental persistent types.

To clarify the point, the following example is provided.

```
Class Emp {
};
```

```
Class Dept {
Emp *employees;
};

Dept dept1;
Emp emp1, emp2;
persistent dept1, emp1;
```

For the above example SPOT is {Dept, Emp}. In modifying the employees field of class Dept, it is observed that this field is a pointer of order one to class Emp, i.e., n=1. So a type of the form Emp*... with minimum number of pointer indirections is searched for in SPOT. It turns out that Emp is in SPOT, with m=0. Thus this configuration fits to case 2.1.1. Since n-m=1, modified type of field employees is OID1 <Emp>.

In C++, employees field of dept1 is a pointer to a memory area that may contain multiple number of objects of type Emp. In our implementation employees field points to a set of object oriented identifiers as explained above. The reason for this is that although C++ does not support object identifiers as a data type, we do so. Thus a persistent object refers to other persistent objects via their logical object identifiers. For this reason, in our system, physical implementation of such pointer fields is different from that of C++ in that they point to a set objects which are not consecutive. But the system provides mechanisms to access them as if they were consecutive through overloading.

As another example consider the following:

```
class A {
    ...
    B *x;
    ...
};

class B {
    ...
    C *y;
    ...
};

class C {
    ...
    Z z;
    ....
};

A a;
B b;
C c;
```

For these declarations, the memory configuration of a, b and c is given in Figure 3.9.

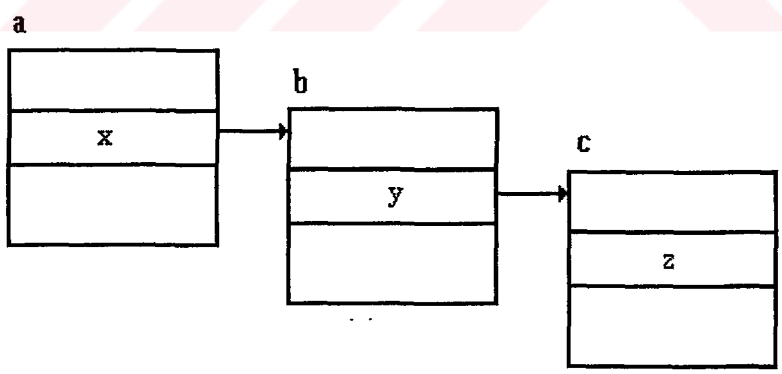

Figure 3.9. Memory Representation

When a, b, and c are made persistent, the disk representation is as shown in Figure 3.10.

It is clear from the examples that in modifying the types of persistent objects the template facility of C++ is used. The template facility provided in C++ is used for associating the pointers with their type information in case of recursive or mutually recursive type definitions. To clarify the discussion, consider the example below :

```
template < class T>
class oid {
public :
...
 T type;
.... };
template < class T>
class x {
public :
...
  oid < T > data;
... };
x < x < LOGICAL_OID > > pobj ;
```

is used for

```
class x {
public  :
...
x* data ;
...
};
x pobj;
```

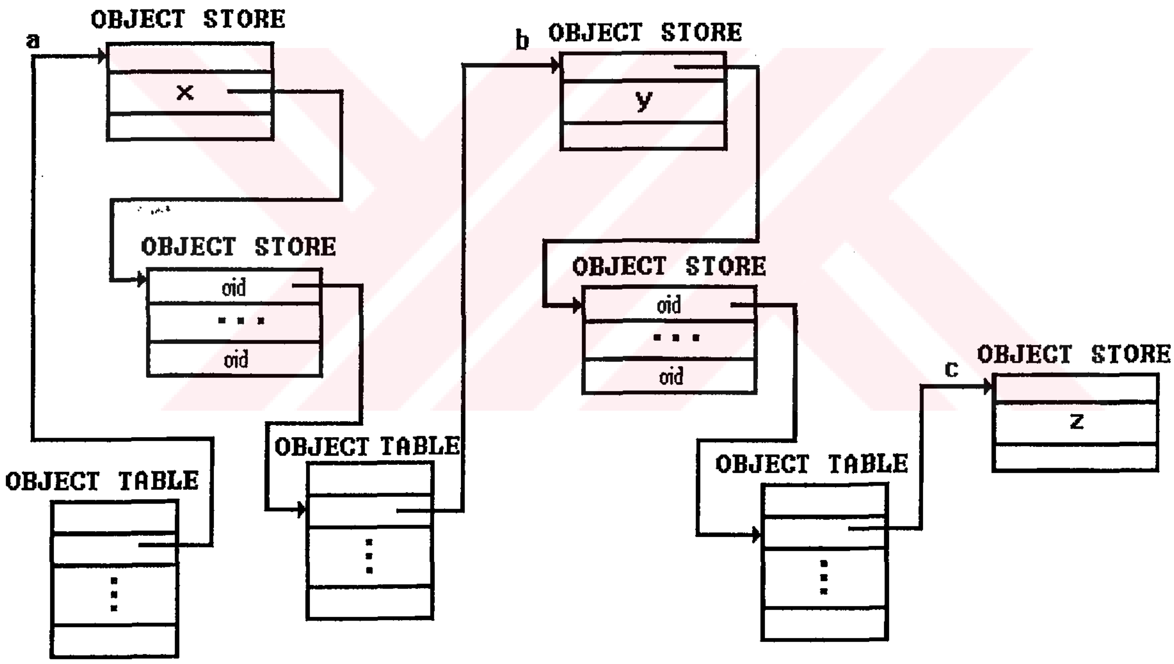

Figure 3.10. Object Store Representation

In case of a persistent object pobj of type x, an expression in the form of (*(pobj.data)).data may be handled if:

i- the dot after the pobj is detected and replaced with the overloadable operator -> to be defined in the derived class which is explained in the next section,

ii- * operator is overloaded in the class oid to return the properly type casted object.

In the example above, LOGICAL_OID is a class requiring an extra object table access.

The dot "." operator should be replaced when the expression to which it is applied needs a proper type cast or it should set a global variable to ensure that the expression to the right of the dot operator to be evaluated correctly. This is the case when the expression to the left of the dot operator have an entry in the object table, i.e., when it is a persistent object name or it is a variable name whose type has been replaced by a STR template type by the preprocessor.

Another example for the modifications performed by the preprocessor is given below:

```
class AlcoholicBeverage {
  public:
    char *type;
    float degree;
};
```

```
class Wine:public AlcoholicBeverage {
   public:
     char brand[20];
     int  year;
  };
class Dinner {
   public:
     char *plate_name;
     Wine *w;
     int price;
  };
```

For this code segment, the following declarations are generated in case a persistent instance of object of class Dinner is defined.

```
class Wine___ {
   public:
     pcharptr1 type;
     pfloat degree;
     pcharptr1 brand;
     pint  year;
   ...
  };
```

```
class Dinner___ {
  public:
    pcharptr1 plate_name;
   STR1 < Wine___ > w;
    int price;
};
...
CSTARO___ <Dinner___> myDinner;
```

In the above code segment Dinner___ is an example to a class template (also called a generic class or class generator) which allows to define a pattern for class definitions. We use this notion to declare persistent objects.

3. 8. Overloader

Overloader is a header module which is included in the preprocessed C++ source with the include compiler directive. In this header module, methods and operators of the fundamental persistent types are overloaded. Hence the evaluation of expressions involving persistent objects which are built upon these types are handled properly. Note that it is C++ compiler's responsibility to call the relevant methods and operators.

In the overloader, a class called CSTARO___ is defined which inherits both from the template class and the class vma___ as shown in Figure 3.11. CSTARO___ is used in defining persistent objects by the preprocessor. The template class for each persistent object declaration is the name of the modified class. This class is created with the translations performed by the preprocessor as explained in Section 3. 7. 2.

For the following C* code segment;

```
Class Emp {
..
};

Class Dept {
...
Emp *employees;
...
};

Dept dept1;
Emp emp1, emp2;
persistent <company> dept1, emp1;
```

the following two C++ statements add persistence to the emp1 and dept1 transient objects.

CSTARO___<Emp___> emp1;

CSTARO___<Dept___> dept1;

Each persistent object has a unique object identifier hidden from the user inherited from the class vma___. When a persistent object is referenced, this identifier is used in locating the object on disk by accessing the object table.

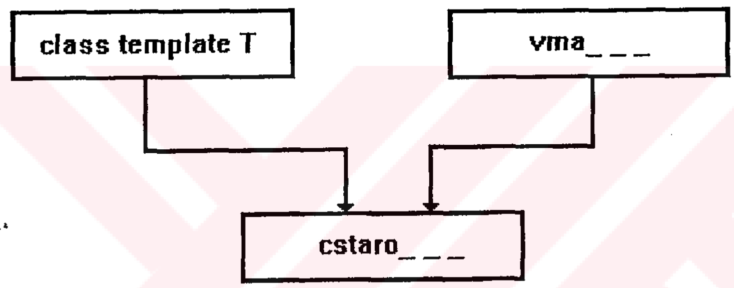

Figure 3.11. Persistent Object Inheritance Topology

The class CSTARO___ is implemented as follows:

template < class T >

class cstaro___ : public T, public vma___ {

public :

```
   cstaro___ () {
           ...
           }
cstaro___ * operator -> () {
                    ...
                    }
cstaro___ operator * () {
                    ...
                    }
};
```

The constructor of the class cstaro initializes the size of the object, and records it in the object table. Most importantly it delays the construction of the fields inherited from template T till the physical location of the object is determined and then calls the constructor of this template causing its data members to be constructed.

Since the size of the normalized objects stays constant, their space is allocated in the related object base at compile time and those persistent objects are given unique logical object-identifiers which are maintained in the object-table. Automatically allocating space for pointer fields of such persistent objects relieves the user from compulsary use of malloc. Constructors of persistent objects are modified by the preprocessor to do this task according to the information obtained from the class parser.

When a transient object is defined, its space in memory is allocated during compile time. Thus when the constructors are called they just initialize the static data members. However since for a persistent object its space can be allocated only in run-time, constructors of these objects are responsible for this task. This is accomplished as follows: When a persistent object is created, its logical object identifier and the corresponding file offset is determined before the fields of this object get their file offsets. As a consequence, the constructor of the flattenned object should be called before the constructors of its fields. Thus the constructor calling sequence of transient objects does not fit to the purposes of that of persistent ones. The constructor calling sequence is controlled through a global variable and explicit calls are issued to the constructors whenever necessary.

In an expression involving persistent objects, the overloaded operators *, and -> of class CSTARO___ return an object casted to the proper type after setting the global object identifier and the related object table which may be used in evaluating this expression.

Through these hierarchy of inheritance, overloading, and modifications, persistent objects start to imitate their transient counterparts in the expressions and make the necessary changes in the buffer area or directly in their physical locations.

The fundamental classes defined in the overloader all have the mainfield data member at offset zero which is used to keep the data itself.

The data structure for persistent data type pint, for example, is as follows:

```
class pint {
public:
  int mainfield;
  long int offset;
  int value_offset;
  ...
};
```

Another data member is the offset field. Since the status of the objects are determined by the contents of the data members, it is necessary to keep track of the relative offsets of the fields within persistent objects to properly access those fields. This task is achieved by modifying the constructors of persistent objects through the information obtained from the class parser such that all the fields of a persistent object get their offset initialized transparently to the user. The assignments to the offset fields of the data members are performed at the very beginning of the constructors. If the class of the persistent object has no user defined constructor, a constructor is generated for it. As an example ,

```
class student___ {
public:
  pint grade; // int grade
  pcharptr1  name ; // char * name
  ...
  student___ () { grade.offset=0;
                name.offset = (sizeof(int));
                name.value_offset=0;
                ...
               }
...
};
```

The value_offset field is used for pointer fields to avoid dangling pointers. As an example consider the fragment given below:

```
class Atype {
public:
```

```
    ...
    char * data;
    ...
};
...
Atype c1,c2;
persistent c1,c2;
...
c1.data= c2.data+5;
```

If the persistent value pointed to by the data field of persistent object c2 is forward marked, it should be possible for the object c1 to access the new location in a consistent manner. In this case, for the uniform treatment of persistent objects, the data field of c1 should point to the beginning of the area pointed to by c2.data to be informed of the forward marking process. It is the value_offset field for pointer data members which enables persistent value sharing. For the above example the value_offset field of the member "data" of persistent object c1 is assigned the increment 5 and accessing c1.data becomes the process of checking for the forward mark , then accessing the data area pointed to by the pointer field with an increment of the value contained in the value_offset field.

3.9. Class Parser

To be able to modify and create necessary type declarations for the persistent objects, the set of class declarations collected by the preprocessor is sent to the class parser.

Class declarations are in the form given below :

Clss ::= ClassDecList

ClassDecList ::= ClassDecList ClassDec | e

ClassDec ::= ClassDeclaration SEMI

ClassDeclaration ::= ClassKey ClassName BaseSpec Members

BaseSpec ::= COLUMN BaseList | e

BaseList ::= BaseSpecifier |

BaseList COMMA BaseSpecifier

BaseSpecifier ::= ClassName |

VIRTUAL AccessSpecifiers ClassName |

AccessSpecifier Virtual ClassName

AccessSpecifiers ::= AccessSpecifier | e

Virtual ::= VIRTUAL | e

AccessSpecifier ::= PRIVATE |

PUBLIC |

PROTECTED

ClassKey ::= CLASS |

UNION |

STRUCT

ClassName ::= IDENTIFIER

Members ::= LC MemberList RC | e

MemberList ::= MemberList MemberDec |

MemberDec

MemberDec ::= FuncDec LP Parameters RP SEMI |

TypeSpecifier MemberDecList SEMI |

AccessSpecifier COLUMN MemberDec

FuncDec ::= IDENTIFIER |

TILDA IDENTIFIER

TypeSpecifier ::= SimpleTypeName

SimpleTypeName ::= ClassName |

ClassKey ClassName |

CHAR |

INT |

SHORT |

LONG |

SIGNED |

UNSIGNED |

FLOAT |

DOUBLE |

VOID

MemberDecList ::= Declarator |

MemberDecList COMMA Declarator

Declarator ::= Dname |

PTR Declarator |

REF Declarator |

Declarator LB NUM RB |

LP Parameters RP |

Declarator LP Parameters RP |

LP Declarator RP

Paramaters ::= SimpleTypeName Ptr |

Parameters COMMA SimpleTypeName Ptr | e

Ptr ::= Ptr REF |

Ptr PTR | e

Dname ::= IDENTIFIER

During the parsing process, the detailed structures of types and inheritance topology is stored in a linked list data structure as shown in figure 3.12. The type field of structure Class shows whether the declaration class key is union,struct or class.The type field of structure Field however, is for differentiating between methods and data members. Since the inheritance topology is a directed acyclic graph, topological sort numbers are used to keep track of this hierarchy. If a class in the class hierarchy has a persistent instance, then the classes which are taken into consideration are those that topologically precede the class at hand and have a simple path to that class. This is illustrated in figure 3.13. For the case given in figure 3.13 if a persistent instance of class E is declared, classes A and C are involved in the flattening process which is explained below.

For a derived class with a persistent instance, all the data members and methods of its base classes are carried down which is called as the flattening. Flattening moves the inherited attributes down the class inheritance topology together with the access specifier adjustments as long as there are no name conflicts. In case of name conflicts, this process adjusts the relative offsets of data members accordingly using the scope resolution operator :: through the modification of constructors of derived classes. It should be noted that for each derived class with a persistent instance, the offsets of base class may be initialized differently.

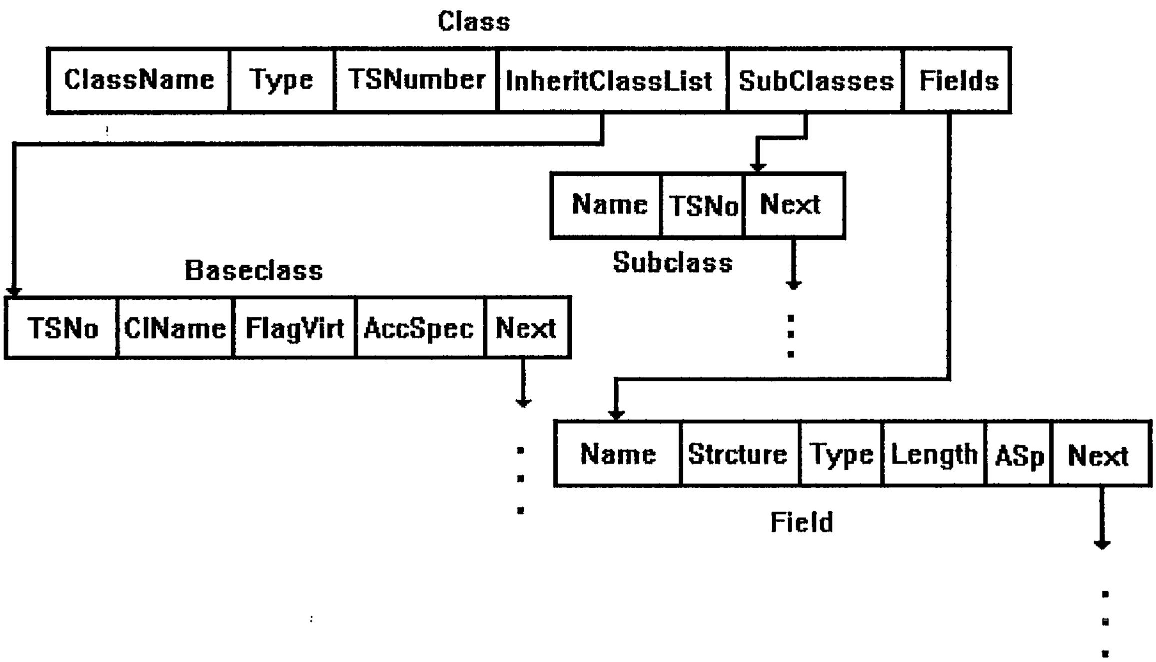

Figure 3.12. Data Structures for Parsing

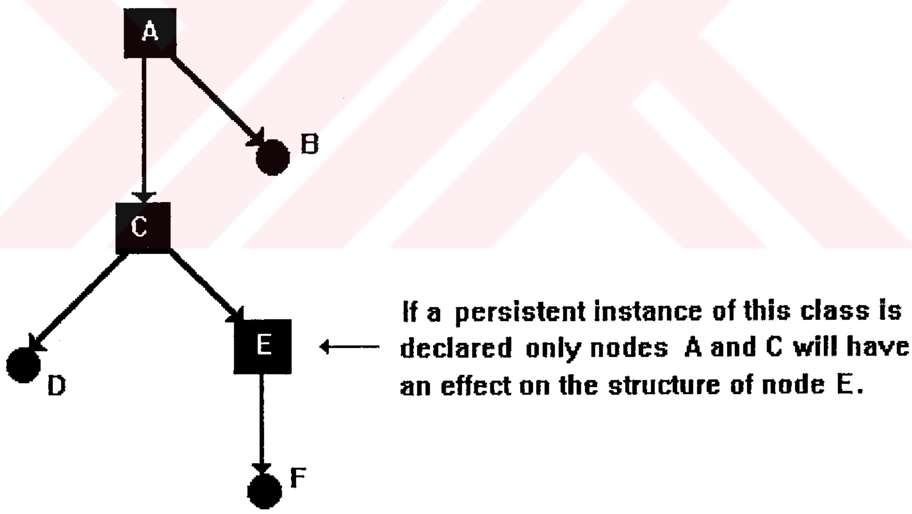

Figure 3.13 . Inheritance Topology

3. 10 Pointer Swizzling in C*

Pointer swizzling is the process of converting the format of pointers while objects migrate to and from main memory and auxiliary storage.

Approaches that have been taken with respect to pointer swizzling can be roughly divided into two groups; those that use memory mapping techniques similar to virtual memory, and those that use software checks, to detect accesses to nonresident objects. In the former approach the basic idea is to allocate virtual memory addresses for pages containing persistent data one step ahead of a program's usage of the pages. When a program first attempts to access a page, a virtual memory page fault occurs. This fault is intercepted by the underlying object manager which then loads the page into its preassigned location in memory. The advantages of this method of swizzling are that programs only see regular virtual memory pointers, allowing accesses to objects to occur at memory speeds. In addition, the same compiled code can be used both for persistent and non-persistent objects. A disadvantage of this approach is that programs may incur unnecessary swizzling and unswizzling overhead [17].

The latter approach as presented in [18] considers an object at-a-time approach to swizzling. All pointers in an unswizzled object are swizzled immediately upon the first use of the object, and the object is then marked as swizzled. One advantage of the object-at-a-time approach is that it should generally perform less unnecessary swizzling and unswizzling work. A disadvantage, however, is that objects that are not referenced by a program directly, can be faulted into memory resulting in unnecessary I/O operations.

Pointer swizzling using software checks may be implemented in several ways.Swizzling pointers upon dereference is one of these. The advantage of this scheme is that pointers that are never referenced are never swizzled. Furthermore since only pointers to referenced objects are used, unnecessary I/O operations are avoided. Swizzling upon dereference fails to swizzle pointers between persistent objects. This is due to the fact that when a pointer is to be dereferenced it has already been copied to a temporary location which is treated as a memory pointer.

Swizzling pointers upon discovery , on the other hand, swizzles pointers when the value of a pointer is copied into a transient pointer variable. This swizzling approach solves the major problem associated with swizzling upon dereference. The disadvantage of this approach is that the object referred may in fact not be needed resulting in unnecessary swizzling and I/O operations.

In the swizzling scheme used by EPVM 2.0 [17], only pointer dereferences are used to fault objects into the cache. Once an object is in the cache, any use of a pointer that refers to the object will cause the pointer to be swizzled.

The comparisons of swizzling strategies [17] reveals that memory-mapped approach is faster than the software approach when operating purely on in-memory data. Another observation in terms of total execution time is that the approach employed by EPVM 2.0 delivered superior performance when compared to the memory-mapped approach when update ratio was high.

In C*, the code is preprocessed to perform necessary modifications transparently to the user. This gives the system the power of detecting the transient counterparts of types of persistent objects, or their data members which are declared as pointers, so that pointer swizzling upon dereference can be performed through the overloaded operators. Apart from being easy to implement, this approach gives the user a uniform view of data types since there is no syntactic difference between pointer types. Operationally there are two kinds of pointers namely, persistent and transient pointers. In the system, since it is allowed to use persistent and transient pointers within the same expression, the operators with operands from different storage classes, i.e., persistent or transient, are also overloaded. This approach while providing the user a uniform view of pointer types and thus keeping the language changes at the absolute minimum, prevents the unnecessary runtime checks on transient pointers. The pointer swizzling is performed within the buffer manager which is invoked by the overloaded operators operating on both persistent pointers and transient pointers used with persistent objects.

CHAPTER IV

EXAMPLES

In order to clarify the implementation details of C*, examples are provided in the following code segments.

Consider the following code fragment.

```
class personnel {
public:
    int age;
};
class student : public personnel {
public:
    int grade;
    char* name;
    student();
 };
student::student () {
 };
 void main() {
```

```
14          student stu;
15          database < cstar > ;
16         persistent < cstar > student stu1;
17         create < cstar > ("demo.dbf");
18         *(stu1.name + 2)='c';
19         stu1.grade = 98;
20         stu.grade=0;
21         close < cstar >;
22  }
```

Lines 1-4 declare the class personnel. This class has an integer valued data member age. Lines 5-10 declare the class student which inherits from the class personnel with an integer valued attribute grade, and a character pointer valued attribute named name. Lines 11-12 shows the constructor of the class student with no functionality. It will be used to demonstrate the modification and usage of the constructors of the persistent objects. Line 14 declares a transient variable stu of class student. Line 15 declares a name that will be bound to the object base at creation time. Line 16 declares a persistent instance of class student, stu1. In line 17 an object base is created with the given path name. In line 18 and 19 expressions involving persistent objects are evaluated. In line 20 an expression involving a transient object of the same class is evaluated. Line 21 closes the object base. As seen from the example, allocations of pointers are not necessary and conventional

C++ expressions are supported allowing address arithmetic and direct assignments.

The above code fragment when processed by the preprocessor generates the code given below.

```
char * FHANDLE = "demo.oid" ;
char * DATA = "demo.dbf" ;
char * OBJTAB = "demo.tab" ;
#define CREAT_OR_OPEN 0
#include "inherit.cpp"
#include "cls.mds"
#include "cnstr.mds"
 class personnel {
 public:
      int age;
 };
 class student___ {
 public:
    pint age ;
    pint grade ;
    pcharptr1 name ;
    student___();
```

```
18 } ;
19 student___ :: student___ () {
20 age.offset=0;
21 grade.offset= sizeof(int);
22 name.offset= sizeof(int)+sizeof(int);
23 name.length = 0;
24 name.pcharptr1::pcharptr1(sizeof(int)+sizeof(int));
25 };
26 void main() {
27         student stu;moods___< student___ > stu1 ;
28         *(stu1->name + 2)='c';
29         stu1->grade = 98;
30         stu.grade=0;
31         fcloseall();
32 }
```

Lines 1-3 declare file names for the files containing the unique object identifiers, object store itself and the object table respectively. In line 4 a constant flag is set by the preprocessor to determine whether a create or an open operation will be executed at run time. Line 5 includes overloader source code. Lines 6-7 include the environment files containing the definition of the original class declaration and its constructor, respectively. Lines 12-18 give the modified class declaration for the persistent object stu1. Lines 19-25 gives the constructor of the class student___ modified to initialize the offsets of the data members of this class. In line 27 the persistent declaration is replaced by a template class declaration. Note that all fields having atomic types such as int, float, char, etc. are modified to have persistent types such as pint, pfloat, pchar, etc., and also the operators for assignment, address arithmetic and dereference for persistent arguments have already been overloaded in the overloader file. In order to overload operators involving pointers a sequence of classes are automatically generated. In the example above, for the name field pcharptr1 and pcharptr0 classes are generated. In lines 29, 30 since component operator,".", can not be overloaded, it is replaced by the dereference operator, "->" . In line 31, close statement is replaced by fcloseall statement.

Another C* code segment and its preprocessed output is given below to illustrate the implicit set concept implemented in C*.

```
class personnel {
public:
    int age;
};
class student : public personnel {
public:
```

```
 int grade;
 char* name;
 student();
};
void main() {
        database < cstar > ;
         persistent < cstar > student stu1;
         open < cstar > ("demo.dbf");
         new_instance<cstar> student stu1;
         access < cstar > student stu1 1 ;
          cout<< stu1.grade;
          close < cstar >;
}
```

```
char * FHANDLE = "demo.oid" ;
char * DATA = "demo.dbf" ;
char * OBJTAB = "demo.tab" ;
#define CREAT_OR_OPEN 1
#include "inherit.cpp"
#include "cls.mds"
#include "cnstr.mds"
class personnel {
```

```
public:
   int age;
};
class student___ {
public:
  pint age ;
  pint grade ;
  pcharptr1 name ;
  student___();
 } ;
student___ ::student___ () {
age.offset=0;
grade.offset= sizeof(int);
name.offset= sizeof(int)+sizeof(int);
name.length = 0;
name.pcharptr1::pcharptr1(sizeof(int)+sizeof(int));
};
void main() {
        moods___< student___ > stu1 ;
       {moods___<student___> stu1;}
        stu1->objid=2;
        stu1->objid=1;
```

```
31      OBJ___=1;
32       cout<< (*(stu1->grade)).mainfield;
33       fcloseall();
34 }
```

In line 28 the statement within curly braces is used to create a new persistent object. In normal C++ context, this declaration does not have any effect. However in C*, this declaration produces a side effect of creating a persistent object through overloading. It should be noted that new_instance clause of the C* creates this code segment.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

C* gives C++ users a great flexibility in DOS environment in that the file processing is automatically and efficiently handled by the system. Furthermore virtual memory management prevents the problem of main memory shortages.

This system is developed as the kernel of an OODBMS which is currently under development.

From the experience we gained from C* implementation is that adding a mechanism to C++ to control the constructor calling sequence will make the language more powerful and flexible. In some cases it is quite possible that a class's constructor will perform some housekeeping process before the constructors of data members of that class or before the constructors of inherited classes are called.

Changing constructor calling sequence and the capability of overloading all operators in C++, will help in the development of system software while keeping the language flexible.

Another suggestion to C++ is to extend the pointer mechanism to be more flexible, i.e. to be able to point objects with different storage classes without degrading performance. Thus handling pointer swizzling through overloading will be much simpler.

The concept of overloading used in the implementation has its uses at different areas. Implementation of virtual arrays, image representation schemes, and search strategies are a few examples where overloading has been proved to be useful for rapid prototyping.

There are certain features of C* which when properly exploited by coupling them with other techniques, will improve system performance. In the following these features are explained.

5. 1 Buffer Management

In C* when a call is issued to the buffer manager, all the information relating to the size and type of the object, and the next operation on it are known. This may increase the efficiency in such a way that buffer manager itself may adjust some fields at arbitrary offsets so that a succeeding dereference operator may directly use a memory address avoiding an extra I/O through the buffer manager. Thus it is possible to use predictive buffer management strategies.

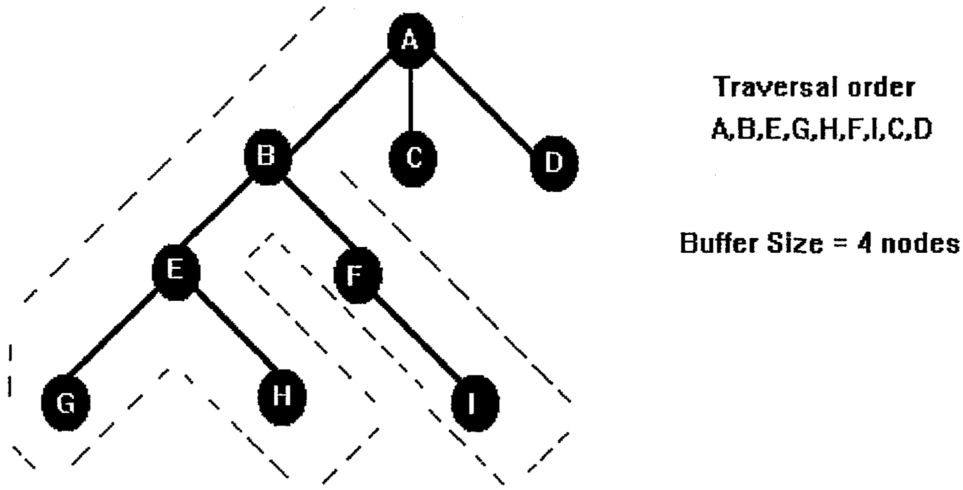

Figure 5.1. The Tree for Depth First Traversal

Another important implication of the previous discussion is that more efficient replacement policies can be implemented. For a possible set of candidates for replacement, those that have high dereference count, i.e., the candidates through which more number of pointers have been dereferenced, may be given high priority. When this approach is used together with the least recently used method for replacement policy, traversal patterns like recursive depth first search cause less faults. As an example, consider the tree shown in figure 5.1. Nodes A to I are visited in depth first order.The traversal order is A, B, E, G, H, F, I, C, D. Let us assume the buffer can accommodate at most four nodes and compare the pure LRU with the proposed replacements policy. For the access pattern generated by the traversal algorithm, the number of page faults for pure LRU, are more than that of the LRU coupled with the heuristic which exhibits a characteristic such that the more the nodes are dereferenced, the lower the chance of being a candidate for replacement. Second approach is more informed of the nature of the access pattern since whether a null pointer has been reached or not is known by the buffer manager.

5. 2 Clustering

Clustering is the process of placing objects physically close on the secondary storage so that the amount of necessary I/O is minimized increasing the throughput of the system. It is an instance of a well-studied graph partitioning problem but it has been stated in [19] that the problem is NP-complete,

One of the important considerations for implementing a clustering strategy is how the description of the expected access patterns for the database is obtained. Maintaining usage statistics, using programmer hints, and performing data-flow analysis of the methods and type hierarchy are examples of techniques for acquiring such information.

The use of logical object identifiers in C* enables dynamic clustering. Additionally, overloading mechanism employed in C* makes it possible to gather statistics on inter object references by storing the captured information.

REFERENCES

[1] M. P., Atkinson, O.P., Buneman, " Types and Persistence in Database Programming Languages", ACM Computing Surveys, 19(2), 126, (1987).

[2] R., Agrawal, N., Gehani, "Design of the Persistence and Query ProcessingFacilities in O++: The Rationale ", Data Engineering, 12(3), 387, (1989).

[3] M. Carey, et. al., "Storage Management for Objects in EXODUS", Object-Oriented Concepts, Databases, and Applications, W. Kim and F. Lochovsky, eds., Addison-Wesley, (1990).

[4] O. Deux, et al."The O2 System", Comm. of the ACM, 34(10), 987, (1991).

[5] C. Lamb, et. al.," The ObjectStore Database System", Comm. of the ACM, 34(10), 1024, (1991).

[6] J. Richardson, M. Carey, "Persistence in the E language: Issues and

Implementation", Software-Practice & Experience, 19(7), 826, (1989).

[7] D. Schuh, M. Carey, D. Dewitt, "Persistence in E Revisited", Implementing Persistent Object Bases, 4(5), 42, (1990).

[8] C. Evrendilek, A. Dogac, I. Durusoy, C. Ozkan, Y. Saygin, M. Altinel, "Adding Persistence to C++ in DOS Environment through Virtual Memory management", Proc. of the 7th Intl. Symposium on Computer and Information Sciences, (1992).

[9] Y. Saygin, " MOODS Storage System", M.S. Thesis in Computer Engineering, Middle East Tecnical University, Ankara, (August 1992).

[10] J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", ACM Trans. Database Syst. , 2(3), 126,(1977).

[11] W.P. Cockshott, M.P. Atkinson, K.J. Chilsholm, P.J. Bailey, R. Morrison, "Persistent Object Management System", Software Practice and Experience, 14(1), 457, (1984).

[12] T. Atwood, " Two Approaches to Adding Persistence to C++", Implementing Persistent Object Bases, 7(8), 39, (1990).

[13] ____________, Turbo Vision for C++ User's Guide, Borland, (1991).

[14] H-T. Chou, D.J. DeWitt, H.R. Katz, A.C. Klug, "Design and Implementation of the Wisconsin Storage System", Software Practice and Experience, 15(1), 943, (1985).

[15] M. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System", Proc. of the 12th VLDB, 452, (August 1986).

[16] P. Pucheral, J.M. Thevenin, " The STRECH Extensible Object Manager", ESPRIT Project Technical Report, (1989).

[17] S.J. White, D.J. Dewitt, " A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies", Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1789, (1992).

[18] J.E.B. Moss, " Working with Persistent Objects: To swizzle or Not to Swizzle", COINS Technical Report, (May 1990).

[19] M.M. Tsangaris, J.F. Naughton, " A Stochastic Approach for Clustering in Object Bases", Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 681, (1992).

APPENDIX

PROGRAM LISTINGS

```
/**********************/
/* CLASSPAR\DBDECL.H  */
/**********************/
struct att    {
     char          *Name;
     int            Tip;        /* 1-Var 2-Method */
     char          *Type;
     int            Length;
     int            ASp;
     struct att    *Next;
};
typedef struct att att;

struct Slst {
   char *Name;
   int  TSNo;
   struct Slst *Next;
};
typedef struct Slst Slst;

struct BLst   {
       int    TSNo;
        char *ClName;
        int    FlagVirt;
        int    AccSpec;
        struct BLst *Next;
};

typedef struct BLst  BLst;

struct clstr {
     char   *ClassName;
     int     Tip;        /* 1-Class 2-Union 3-Struct */
     int     TSNumber;
     BLst   *InheritClassList;
     Slst   *SubClasses;
     att    *Fields;
} ;
typedef struct clstr clstr;

struct decl {
     char   *Name;
     int     Tip;
     int     No;
     struct decl   *Next;
};
typedef struct decl decl;
```

```
/***************************/
/*CLASSPAR\YYOUT.H        */
/***************************/

#define _EOI_       0
#define SEMI        1
#define COLUMN      2
#define COMMA       3
#define REF         4
#define LP          5
#define RP          6
#define LC          7
#define RC          8
#define LB          9
#define RB          10
#define PRIVATE     11
#define PROTECTED   12
#define PTR         13
#define VIRTUAL     14
#define PUBLIC      15
#define TILDA       16
#define CLASS       17
#define UNION       18
#define STRUCT      19
#define IDENTIFIER  20
#define NUM         21
#define REALNUM     22
#define INT         23
#define CHAR        24
#define LONG        25
#define FLOAT       26
#define UNSIGNED    27
#define SIGNED      28
#define SHORT       29
#define DOUBLE      30
#define VOID        31
#define EQU         32
```

```
/****************************/
/* CLASSPAR\LEXAN.C         */
/****************************/

#include<stdio.h>
#include<stdlib.h>
#include "yyout.h"
#include<alloc.h>

char filename[100];
extern unsigned char  tlength;
extern char far *text;

FILE *infile;
char far *buffer;
char far *ptr;
int   count;
int   p;

void init(void);
unsigned char  get_token(void);

void init()
{
   buffer = (char*) farmalloc(2048);
   infile = fopen(filename,"rb");
   count = fread(buffer,1,2048,infile);
   p = 2048;
   ptr = buffer;
}

unsigned char get_token(void)
{
   unsigned char cnt;

   while(1) {
       if( p<33) {
          _fmemcpy(buffer,ptr,p);
          count = p + fread(buffer,1,2048-p,infile);
          p = 2048;
          ptr = buffer;
       }
       if(!count) {
          count = fread(buffer,1,2048,infile);
          if (!count)  {
             farfree(buffer);
             fclose(infile);
             return(0);
          }
```

```
            ptr = buffer;
            p=2048;
        }
        switch ( *ptr) {
            case ';'  : count--; p--; ptr++ ; return(SEMI);
            case ':'  : count--; p--; ptr++ ;
return(COLUMN);
            case ','  : count--; p--; ptr++ ;
return(COMMA);
            case '*'  : text=(char far*) farmalloc(2);
                        *text='*'; *(text+1)=0; tlength
=2;
                        count--; p--; ptr++ ;
return(PTR);
            case '&'  : text=(char far*) farmalloc(2);
                        *text='&'; *(text+1)=0;
tlength=2;
                        count--; p--; ptr++ ;
return(REF);
            case '~'  : count--; p--; ptr++ ;
return(TILDA);
            case '='  : count--; p--; ptr++ ; return(EQU);
            case '('  : count--; p--; ptr++ ; return(LP);
            case ')'  : count--; p--; ptr++ ; return(RP);
            case '{'  : count--; p--; ptr++ ; return(LC);
            case '}'  : count--; p--; ptr++ ; return(RC);
            case '['  : count--; p--; ptr++ ; return(LB);
            case ']'  : count--; p--; ptr++ ; return(RB);
        }
        if (!_fstrncmp(ptr,"int",3)) {
            text = (char*)farmalloc(4);
            tlength = 4;
            _fstrcpy(text,"int");
            ptr +=3; count -=3; p -=3;
            return(INT);
        }
        else if (!_fstrncmp(ptr,"char",4)) {
            text = (char*)farmalloc(5);
            tlength = 5;
            _fstrcpy(text,"char");
            ptr +=4;  count -=4; p -=4;
            return(CHAR);
        }
        else if (!_fstrncmp(ptr,"long",4)) {
            text = (char*)farmalloc(5);
            tlength =5;
            _fstrcpy(text,"long");
            ptr +=4; count -=4; p -=4;
            return(LONG);
        }
        else if (!_fstrncmp(ptr,"float",5)) {
            text = (char*)farmalloc(6);
```

```
        tlength =6;
        _fstrcpy(text,"float");
        ptr +=5; count-=5; p -=5;
        return(FLOAT);
    }
    else if (!_fstrncmp(ptr,"unsigned",8)) {
        text = (char*)farmalloc(9);
        tlength =9;
        _fstrcpy(text,"unsigned");
        ptr +=8; count -=8; p -=8;
        return(UNSIGNED);
    }
    else if (!_fstrncmp(ptr,"signed",6)) {
        text = (char*)farmalloc(7);
        tlength =7;
        _fstrcpy(text,"unsigned");
        ptr +=6; count -=6; p -=6;
        return(SIGNED);
    }
    else if (!_fstrncmp(ptr,"short",5)) {
        text = (char*)farmalloc(6);
        tlength =6;
        _fstrcpy(text,"short");
        ptr +=5; count -=5; p -=5;
        return(SHORT);
    }
    else if (!_fstrncmp(ptr,"double",6)) {
        text = (char*)farmalloc(7);
        tlength =7;
        _fstrcpy(text,"double");
        ptr +=6; count -=6; p -=6;
        return(DOUBLE);
    }
    else if (!_fstrncmp(ptr,"void",4)) {
        text = (char*)farmalloc(5);
        tlength =5;
        _fstrcpy(text,"void");
        ptr +=4; count -=4; p -=4;
        return(VOID);
    }
    else if (!_fstrncmp(ptr,"class",5)) {
        text = (char*)farmalloc(6);
        tlength =6;
        _fstrcpy(text,"class");
        ptr +=5; count -=5; p -=5;
        return(CLASS);
    }
    else if (!_fstrncmp(ptr,"struct",6)) {
        text = (char*)farmalloc(7);
        tlength =7;
        _fstrcpy(text,"struct");
        ptr +=6; count -=6; p -=6;
```

```
        return(STRUCT);
    }
    else if (!_fstrncmp(ptr,"union",5)) {
        text = (char*)farmalloc(6);
        tlength =6;
        _fstrcpy(text,"union");
        ptr +=5; count -=5; p -=5;
        return(UNION);
    }
    else if (!_fstrncmp(ptr,"virtual",7)) {
        text = 0;
        ptr +=7; count -=7; p -=7;
        return(VIRTUAL);
    }
    else if (!_fstrncmp(ptr,"private",7)) {
        text = 0;
        ptr +=7; count -=7; p -=7;
        return(PRIVATE);
    }
    else if (!_fstrncmp(ptr,"public",6)) {
        text = 0;
        ptr +=6; count -=6; p -=6;
        return(PUBLIC);
    }
    else if (!_fstrncmp(ptr,"protected",9)) {
        text = 0;
        ptr +=9; count -=9; p -=9;
        return(PROTECTED);
    }
    else if (isalpha(*ptr) ) {
        cnt = 1;
        while (isalnum(*(ptr+cnt)))    cnt++;
        text = (char*)farmalloc(cnt+1);
        tlength =cnt+1;
        _fstrncpy(text,ptr,cnt);
        text[cnt]=0;
        ptr +=cnt; count -=cnt; p -=cnt;
        return(IDENTIFIER);
    }
    else if (isdigit(*ptr) ) {
        cnt = 1;
        while (isdigit(*(ptr+cnt)))    cnt++;
        if (*(ptr+cnt)=='.') {
             cnt++;
             while (isdigit(*(ptr+cnt))) cnt++;
             text = (char*)farmalloc(cnt+1);
             tlength =cnt+1;
             _fstrncpy(text,ptr,cnt);
             text[cnt]=0;
             ptr +=cnt; count -=cnt; p -=cnt;
             return(FLOAT);
        }
```

```
            text = (char*)farmalloc(cnt+1);
            tlength =cnt+1;
            _fstrncpy(text,ptr,cnt);
            text[cnt]=0;
            ptr +=cnt; count -=cnt; p -=cnt;
            return(NUM);
         }
         else if (*ptr == '.') {
            cnt=1;
            while (isdigit(*(ptr+cnt))  cnt++;
            text = (char*)farmalloc(cnt+1);
            tlength =cnt+1;
            _fstrncpy(text,ptr,cnt);
            text[cnt]=0;
            ptr +=cnt; count -=cnt; p -=cnt;
            return(FLOAT);
         }

         else if (*ptr=='-'|| *ptr =='+' ) {
            cnt = 1;
            while (*(ptr+cnt) ==' ') cnt++;
            while (isdigit(*(ptr+cnt))) cnt++;
            if (*(ptr+cnt)=='.') {
                 cnt++;
                 while (isdigit(*(ptr+cnt))) cnt++;
                 text = (char*)farmalloc(cnt+1);
                 tlength =cnt+1;
                 _fstrncpy(text,ptr,cnt);
                 text[cnt]=0;
                 ptr +=cnt; count -=cnt; p -=cnt;
                 return(FLOAT);
            }
            text = (char*)farmalloc(cnt+1);
            tlength =cnt+1;
            _fstrncpy(text,ptr,cnt);
            text[cnt]=0;
            ptr +=cnt; count -=cnt; p -=cnt;
            return(NUM);
         }
         else {
            ptr++;
            count--;
            p--;
         }
     }
}
```

```
/**************************/
/*CLASSPAR\PARSE.C       */
/**************************/

#include "dbdecl.h"
#include "yyout.h"
#include <alloc.h>
#include <stdlib.h>
#include <stdio.h>


char c;
char clname[30];
char far *errstr[100];
extern unsigned char get_token(void);
extern void init(void);
int  declerator();
int  parse();
void write_rout();
int  not_exist(char*,att*);
int  findTSNo(char*);
void AddToList(att*,BLst*);
int BaseList();

unsigned char  tlength;
char far *text,*tempstr,*tempstr2;
unsigned char token,temp,temp2;
int TopNum,Curr,top;

clstr *OneCls,*ClsLst[100];
att   *OneAttr, *Att1;
BLst  *BaseCls, *BCls1;

int parse(void)
{
     TopNum = 0;
     while(token=get_token()){
        switch (token) {
```

```
            case CLASS   : Curr = PRIVATE; break;
            case STRUCT  :
            case UNION   : Curr = PUBLIC;break;
            default : _fstrcpy(errstr," CLASS expected
,invalid token");
                        return -1;
        }
        TopNum++;
        OneCls = (clstr*) farmalloc(sizeof(clstr));
        OneCls->Tip = token;
        OneCls->TSNumber=TopNum;
        OneCls->Fields=0;
        OneCls->InheritClassList=0;
        OneCls->SubClasses=0;
        token=get_token();  /* class name */
        OneCls->ClassName=text;
        ClsLst[TopNum]=OneCls;
        if ((token=get_token())==COLUMN) {
            BaseList();

AddToList(OneCls->Fields,OneCls->InheritClassList);
        } /* if*/
        if ( token ==SEMI) continue; /* class decleration
ends */
        if ( token != LC)  {
            _fstrcpy(errstr," LC expected");
            return -1;
        }
/*  MEMBER DECLERATORS */
        while ((temp=get_token()) != RC ) {
            tempstr = text;
            switch (temp) {
                case TILDA :
                      token = get_token();  /* identifier
*/
                      tempstr = (char far *)
farmalloc(tlength+2);
                      *tempstr = '~';
                      _fstrcpy(tempstr+1,text);
                      tempstr[tlength]=0;
                      farfree(text);
                      if ((token=get_token()) == LP)

while((token=get_token())!=RP);
                      else   {
                              _fstrcpy(errstr," LP
expected");
                              return -1;
                      }
                      if ((token=get_token())!= SEMI ) {
                              sprintf(errstr," SEMI
expected in dec of %s",tempstr);
```

```
                            return -1 ;
                    }
                    OneAttr = (att*)
farmalloc(sizeof(att));
                    OneAttr->Name = tempstr;
                    OneAttr->Type = (char far*)
farmalloc(5);
                    _fstrcpy(OneAttr->Type,"CONS");
                    OneAttr->Tip=2;
                    OneAttr->Next=0;
                    OneAttr->ASp=Curr;
                    OneAttr->Length=1;
                    if (Att1 = OneCls->Fields) {
                            while (Att1->Next)
Att1=Att1->Next;

                            Att1->Next = OneAttr;
                    }
                    else OneCls->Fields= OneAttr;
                    break;
               case INT    :
               case CHAR   :
               case SHORT  :
               case LONG   :
               case SIGNED:
               case UNSIGNED:
               case FLOAT:
               case DOUBLE:
               case VOID :
                    do {
                         /*var name or * or & */
                         temp2 = get_token();
                         tempstr2 = text;
                         declerator();
                    } while (token==COMMA);
                    if (token!= SEMI ) {
                            sprintf(errstr," SEMI
expected in dec of %s",tempstr);
                            return -1 ;
                    }
                    break;
               case PUBLIC:
               case PRIVATE:
               case PROTECTED :
                    Curr = temp;
                    if((temp=get_token())!= COLUMN) {
                            sprintf(errstr," SEMI
expected in dec of %s",tempstr);
                            return -1 ;
                    }
                    break;
               case CLASS:
               case STRUCT:
```

```
                case UNION :
                      continue;
                case IDENTIFIER : /* ya constructor yada
class name */
                      if ((temp2=get_token())==LP) {

while((token=get_token())!=RP);
                              if ((token=get_token())!=
SEMI ) {
                                 sprintf(errstr," SEMI
expected in dec of %s",tempstr);
                                 return -1 ;
                              }
                              OneAttr = (att*)
farmalloc(sizeof(att));
                              OneAttr->Name = tempstr;
                              OneAttr->Type = (char far*)
farmalloc(5);

_fstrcpy(OneAttr->Type,"CONS");
                              OneAttr->Tip=2;
                              OneAttr->Next=0;
                              OneAttr->ASp=Curr;
                              OneAttr->Length=1;
                              if (Att1 = OneCls->Fields)
{
                                   while (Att1->Next)
Att1=Att1->Next;
                                   Att1->Next = OneAttr;
                              }
                              else OneCls->Fields=
OneAttr;
                              break;
                      }
                      else {
                              tempstr2=text;
                              declerator();
                              while (token==COMMA) {
                                   temp2 = get_token();
                                   tempstr2 = text;
                                   declerator();
                              }
                              if (token!= SEMI ) {
                                   sprintf(errstr," SEMI
expected in dec of %s",tempstr);
                                   return -1 ;
                              }
                              break;
                      }
          }
      } /*while RC */
/*  END MEMBER DECLERATIONS */
```

```
/*
AddToList(OneCls->Fields,OneCls->InheritClassList); */
     if ((token=get_token())!=SEMI) {
        sprintf(errstr," SEMI expected in dec of
%s",tempstr);
        return -1 ;
     }

     }    /* while*/
 return (0);
}       /* parse*/

int declerator()
{
   int x2;
   att *Before;
        switch(temp2) {
            case IDENTIFIER :
                  switch (token = get_token()) {
                      case LP :

while((token=get_token())!=RP);
                            token=get_token();
                            OneAttr = (att*)
farmalloc(sizeof(att));
                            OneAttr->Name = tempstr2;
                            OneAttr->Type = tempstr;
                            OneAttr->Tip=2;
                            OneAttr->Next=0;
                            OneAttr->ASp=Curr;
                            OneAttr->Length=1;
                            if (Att1 = OneCls->Fields) {
                                while (Att1->Next)
Att1=Att1->Next;
                                Att1->Next = OneAttr;
                            }
                            else OneCls->Fields= OneAttr;
                            break;
                      case SEMI :
                      case COMMA :
                            OneAttr = (att*)
farmalloc(sizeof(att));
                            OneAttr->Name = tempstr2;
                            OneAttr->Type = tempstr;
                            OneAttr->Tip=1;
                            OneAttr->Next=0;
                            OneAttr->ASp=Curr;
                            OneAttr->Length=1;
/************             if (Att1 = OneCls->Fields) {
                                while
(strcmp(OneAttr->Name,Att1->Name) && Att1->Next)
                                        Att1=Att1->Next;
```

```
                                if
(strcmp(OneAttr->Name,Att1->Name) )
                                         Att1->Next =
OneAttr;

                            }
***********/            if ( Before = OneCls->Fields) {
                                if (
!strcmp(OneAttr->Name,Before->Name))

Before=OneCls->Fields= Before->Next;
                                if ( Before) {
                                    Att1=Before->Next;
                                    if ( Att1) {
                                        while
(strcmp(OneAttr->Name,Att1->Name) && Att1->Next) {
                                            Before=Att1;

Att1=Att1->Next;
                                        }
                                        if
(!strcmp(OneAttr->Name,Att1->Name) ) {

Before->Next=Att1->Next;
                                            Att1=Before;
                                            while
(Att1->Next) Att1=Att1->Next;
                                            Att1->Next =
OneAttr;
                                        }
                                        else Att1->Next =
OneAttr;
                                    }
                                    else  Before->Next=
OneAttr;
                                }
                                else OneCls->Fields=
OneAttr;
                            }
                            else OneCls->Fields= OneAttr;
                            break;
                      case LB :
                            top =1;
                            do {
                                if
((token=get_token())!=NUM) {
                                    sprintf(errstr," NUM
expected in dec of %s",tempstr2);
                                    return -1 ;
                                }
                                top *= atoi(text);
                                if
```

```
((token=get_token())!=RB) {
                                        sprintf(errstr," RB
expected in dec of %s",tempstr2);
                                        return -1 ;
                                    }
                               } while
((token=get_token())==RB);
                               OneAttr = (att*)
farmalloc(sizeof(att));
                               OneAttr->Name = tempstr2;
                               OneAttr->Type = tempstr;
                               OneAttr->Tip=1;
                               OneAttr->Next=0;
                               OneAttr->ASp=Curr;
                               OneAttr->Length=top;
                               if (Att1 = OneCls->Fields) {
                                   while (Att1->Next)
Att1=Att1->Next;
                                   Att1->Next = OneAttr;
                               }
                               else OneCls->Fields= OneAttr;
                               break;
                  }
                  break;
            case PTR:
            case REF :
                  while((temp2=get_token())==PTR ||
temp2==REF) {
                     tempstr2 = (char
far*)farrealloc(tempstr2,strlen(tempstr2)+2);
                     x2 = strlen(tempstr2);
                     tempstr2[x2] =*text ;
                     tempstr2[x2+1] =0 ;
                  }
                  tempstr2 = (char
far*)farrealloc(tempstr2,strlen(tempstr2)+tlength);

_fstrcpy(&(tempstr2[strlen(tempstr2)]),text) ;
                  declerator();
                  break;
        }
}

int BaseList()
{
            do {
                BaseCls = (BLst*)farmalloc(sizeof(BLst));
                switch(token=get_token()) {
                    case IDENTIFIER :
                          BaseCls->ClName=text;
                          BaseCls->TSNo =findTSNo(text);
                          BaseCls->FlagVirt=0;
```

```
                        BaseCls->AccSpec = PUBLIC;
                        break;
                case VIRTUAL:
                        BaseCls->FlagVirt=1;
                        switch(token=get_token()){
                            case IDENTIFIER :
                                  BaseCls->ClName=text;
                                  BaseCls->TSNo
=findTSNo(text);
                                  BaseCls->AccSpec =
PUBLIC;
                                  break;
                            case PUBLIC:
                            case PRIVATE:
                            case PROTECTED :
                                  BaseCls->AccSpec =
token;
                                  token=get_token();
                                  BaseCls->ClName=text;
                                  BaseCls->TSNo
=findTSNo(text);
                                  break;
                            default :
                                  _fstrcpy(errstr,"
invalid token");
                                  return -1;
                        } /*switch*/
                        break;
                case PUBLIC:
                case PRIVATE :
                case PROTECTED :
                        BaseCls->AccSpec = token;
                        switch(token=get_token()){
                            case IDENTIFIER :
                                  BaseCls->ClName=text;
                                  BaseCls->TSNo
=findTSNo(text);
                                  BaseCls->FlagVirt=0;
                                  break;
                            case VIRTUAL :
                                  BaseCls->FlagVirt= 1;
                                  token=get_token();
                                  BaseCls->ClName=text;
                                  BaseCls->TSNo
=findTSNo(text);
                                  break;
                            default :
                                  _fstrcpy(errstr,"
invalid token");
                                  return -1;
                        } /*switch*/
                        break;
```

```
                        default :
                              _fstrcpy(errstr," invalid
token");
                              return -1;
                    } /*switch*/
                    BaseCls->Next=0;
                    if (BCls1 = OneCls->InheritClassList) {
                        while (BCls1->Next)
BCls1=BCls1->Next;
                        BCls1->Next = BaseCls;
                    }
                    else OneCls->InheritClassList = BaseCls;
                } while((token=get_token())== COMMA);

}


void write_rout()
{
  FILE* moodstr,*name_to;
  att   *a1;
  BLst  *b1;
  clstr *p;
  int   idx,i,count;
  idx=1;
//  printf(" Enter the name of required class : \n");
  name_to=fopen("name.mds","rt");
  fscanf(name_to,"%s", clname);
  fclose(name_to);
  printf("%s",clname);
  moodstr=fopen("str.mds","wt");
  while(p = ClsLst[idx]) {
     if (! strcmp(p->ClassName,clname)) {
        a1=p->Fields;
        while(a1) {
           if( a1->Tip ==1)  {
             count = 0;
             while(a1->Name[count]=='*' ||
a1->Name[count]=='&') count++;
             fprintf(moodstr,"%-20s ",a1->Name+count);
             if ( a1->Length > 1)
fprintf(moodstr,"%d*",a1->Length);
             fprintf(moodstr,"sizeof(");
             if (!strcmp(a1->Type,"int")) {
                 fprintf(moodstr,"pint");
                 if ( count )
fprintf(moodstr,"ptr%d",count);
             }
             else if  (!strcmp(a1->Type,"char")) {
                 fprintf(moodstr,"pchar");
                 if ( count )
fprintf(moodstr,"ptr%d",count);
```

```
                }
                else if  (!strcmp(a1->Type,"float")) {
                    fprintf(moodstr,"pfloat");
                    if ( count )
fprintf(moodstr,"ptr%d",count);
                }
                else {
                   for(i=0;i<count;i++)
                       fprintf(moodstr,"%c",a1->Name[i]);
                   fprintf(moodstr,"%s",a1->Type);
                }
                fprintf(moodstr,")\n");
              }
              a1=a1->Next;
           }
        }
        idx++;
     }
     fclose(moodstr);
}


int not_exist(name, list)
        char *name;
        att *list;
{
        att *p;
       p=list;
       while(p){
          if (!strcmp(name,p->Name))
          return(0);
          p=p->Next;
       }
       return(1);
}

att* last_elem(list)
      att* list;
{
        att *p;
        p=list;
        if (!p) return 0;
        while(p->Next)
          p = p->Next;
       return(p);
}

void AddToList(y,xx)
     att    *y;
     BLst   *xx;
{
        BLst   *x;
```

```
      att     *a, *y1, *y2;
      clstr   *p;
      x = xx;
      y1 = last_elem(y);
      while(x) {
         p = ClsLst[x->TSNo];
         a = p->Fields;
         while (a) {
             if ( a->ASp != PRIVATE ) {
            if (( x->AccSpec == PRIVATE)&&( not_exist(
a->Name, y))) {
                 y2 = (att*) malloc(sizeof(att));
                 *y2 = *a;
                 y2->ASp = PRIVATE;
                 y2->Next= 0;
                 if (!y1) OneCls->Fields = y2;
                 else y1->Next = y2;
                 y1 = y2;
            }
            else if((x->AccSpec == PUBLIC)&&( not_exist(
a->Name, y))) {
                y2 = (att*) malloc(sizeof(att));
                *y2 = *a;
                y2->Next= 0;
                 if (!y1) OneCls->Fields = y2;
                 else y1->Next = y2;
                 y1 = y2;
            }
            else if((x->AccSpec==PROTECTED)&&( not_exist(
a->Name, y))) {
                 y2 = (att*) malloc(sizeof(att));
                 *y2 = *a;
                 y2->Next= 0;
                 y2->ASp= PROTECTED;
                 if (!y1) OneCls->Fields = y2;
                 else y1->Next = y2;
                 y1 = y2;
            }
             }
             a = a->Next;
         }
         x = x->Next;
      }
}

int findTSNo( name)
    char *name;
{
   clstr *p;
   int    x=1;

   p = ClsLst[1];
```

```
    while (p) {
        if(!strcmp(p->ClassName,name))
        return(x);
        x++;
        p=ClsLst[x];
        if (x>TopNum) exit(1);
    }
    return 0;
}
```

```
/***************************/
/*CLASSPAR\MAIN.C         */
/***************************/

extern void init(void);
extern void write_rout();
extern int parse();
extern char far *errstr[100];
#include <stdio.h>
#include <string.h>


extern char filename[100];

main()
{
```

```
 /* printf("\nEnter The name of file :\n ");
  scanf("%s",filename);*/
  strcpy(filename,"cls.mds");

  init();
  printf(" %d \n", parse());
  write_rout();
  printf(" \n");
  printf(" %s ",errstr[0]);
  return 1;
}
```

```
/**************************/
/*TOKEN\LEX.CPP          */
/**************************/

#include <sys\stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>
#include <iostream.h>
#include <conio.h>
```

```
#define MBSIZE 2048
#define MTSIZE 256
#define OUR_EOF "|||"
#define CR 10
#define LF 13


int curpos=0,prevpos=0,buflen=0,fhandle;
int isblank,isnewline;
unsigned char *buffer;

void error(int errorno)
{
  switch(errorno) {
      case 1: printf("File open error\n");
              break;
      case 2: printf("File read error\n");
              break;
      case 3: printf(" input file not specified\n");
              break;
      case 5: printf(" Floating point error\n");
              break;
  }
  exit(0);
}

void fillbuf()
{
  if((buflen=read(fhandle,buffer,MBSIZE))<0) error(2);
}

init(char *fname)
{
  buffer =(unsigned char *) malloc(MBSIZE);
  fhandle = open(fname,O_RDWR | O_BINARY);
  curpos=0;prevpos=0;
  if ( fhandle < 0 ) error(1);
  fillbuf();
  return(fhandle);
}

unsigned char * read_char()
{
  static unsigned char ch[4];
  if(curpos<buflen) {
       curpos++;
       ch[0]=buffer[curpos-1];
       ch[1]='\0';
       return(ch);
  }
  if(buflen==MBSIZE) {
       fillbuf();
```

```
        curpos=1;
        prevpos-=MBSIZE;
        ch[0]=buffer[0];
        ch[1]='\0';
        return(ch);
  }
  else {
        strcpy(ch,OUR_EOF);
        return(ch);
  }
}

void unread_char()
{ long loc;
  if (curpos) curpos--;
  else {
            loc = lseek(fhandle,0,1);
            lseek(fhandle,loc-buflen-MBSIZE,0);
            fillbuf();
  }
}

void backtrack(unsigned char *p)
{ int j=0,i=strlen(p);

  for(;j<i;j++) unread_char();
}

void skip_comment()
{ unsigned char *ch, *ch1;
  int flag=1;
  ch = read_char();
  switch(*ch){
     case '/' : ch1=read_char();
                    switch(*ch1){
                        case '*' : while(flag){
                                        ch1=read_char();
                                        if(*ch1=='*'){

ch1=read_char();

if(*ch1=='/') flag=0;
                                        }
                                    } break;
                        case '/' : while(flag){
                                      ch1=read_char();
                                      if(*ch1==CR){

ch1=read_char();

if(*ch1==LF) flag=0;
                                        }
```

```
                                        } break;
                        default    : unread_char();
                    } break;
      default   : unread_char();
   }
}

unsigned char *return_delimeter(unsigned char *token)
{
   unsigned char *ch,*ch1;
   int tokenptr=0;
   ch=read_char();
   switch(*ch){
      case '=': ch1=read_char();
                if(*ch1=='=') strcpy(token,"==");
                else{
                    unread_char();
                    strcpy(token,"=");
                } break;
      case '<': ch1=read_char();
                if(*ch1=='=') strcpy(token,"<=");
                else{
                    unread_char();
                    strcpy(token,"<");
                } break;
      case '>': ch1=read_char();
                if(*ch1=='=') strcpy(token,">=");
                else{
                   unread_char();
                   strcpy(token,">");
                } break;
      case '-': ch1=read_char();
                switch(*ch1){
                   case '='      : strcpy(token,"-=");
                                      break;
                   case '-'      : strcpy(token,"--");
                                      break;
                   case '>'      : strcpy(token,"->");
                                      break;
                   default       : unread_char();
                                      strcpy(token,"-");
                } break;
      case '+': ch1=read_char();
                switch(*ch1){
                   case '='      : strcpy(token,"+=");
                                      break;
                   case '+'      : strcpy(token,"++");
                                      break;
                   default       : unread_char();
                                      strcpy(token,"+");
                } break;
      case '*': ch1=read_char();
```

```
                switch(*ch1){
                    case '='      : strcpy(token,"*=");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"*");
                } break;
        case '/': ch1=read_char();
                switch(*ch1){
                    case '='      : strcpy(token,"/=");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"/");
                } break;
        case '|': ch1=read_char();
                switch(*ch1){
                    case '='      : strcpy(token,"|=");
                                        break;
                    case '|'      : strcpy(token,"||");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"|");
                } break;
        case '&': ch1=read_char();
                switch(*ch1){
                    case '='      : strcpy(token,"&=");
                                        break;
                    case '&'      : strcpy(token,"&&");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"&");
                } break;
        case '%': ch1=read_char();
                switch(*ch1){
                    case '='      : strcpy(token,"%=");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"%");
                } break;
        case '!': ch1=read_char();
                switch(*ch1){
                    case '='      : strcpy(token,"!=");
                                        break;
                    default       : unread_char();
                                        strcpy(token,"!");
                } break;
        case '"': token[0]='"';
                ch1=read_char();
                while(*ch1!='"' &&
strcmp(ch1,OUR_EOF)!=0){
                        token[++tokenptr]=*ch1;
                        ch1=read_char();
                }
```

```
                if(strcmp(ch1,OUR_EOF)==0) error(6);
                token[++tokenptr]='"';
                token[++tokenptr]='\0';
                break;
        case '\'': token[0]='\'';
                ch1=read_char();
                while(*ch1!='\'' &&
strcmp(ch1,OUR_EOF)!=0){
                    token[++tokenptr]=*ch1;
                    ch1=read_char();
                }
                if(strcmp(ch1,OUR_EOF)==0) error(6);
                token[++tokenptr]='\'';
                token[++tokenptr]='\0';
                break;
        case '#':
                while( (*ch == '#') ||
                        (*ch>='a' && *ch<='z') ||
                        (*ch>='A' && *ch<='Z'))
                {
                    token[tokenptr++]=*ch;
                    ch=read_char();
                    /* buralara obur file'a yazmaliyiz
*/
                }
                token[tokenptr]='\0';
                unread_char();
                return(token);
                break;
        case '^': strcpy(token,"^");
                break;
        case ',': strcpy(token,",");
                break;
        case '~': strcpy(token,"~");
                break;
        case '(': strcpy(token,"(");
                break;
        case ')': strcpy(token,")");
                break;
        case '[': strcpy(token,"[");
                break;
        case ']': strcpy(token,"]");
                break;
        case '{': strcpy(token,"{");
                break;
        case '}': strcpy(token,"}");
                break;
        case ';': strcpy(token,";");
                break;
        case '?': strcpy(token,"?");
                break;
        case ':': ch1=read_char();
```

```
                    switch(*ch1){
                       case ':'      : strcpy(token,"::");
                                         break;
                       default       : unread_char();
                                         strcpy(token,":");
                    } break;
      case '.': ch1=read_char();
                    if(*ch1=='.'){
                         ch1=read_char();
                         if(*ch1=='.') strcpy(token,"...");
                         else{
                              unread_char();
                              strcpy(token,"..");
                         }
                    }
                    else{
                         unread_char();
                         strcpy(token,".");
                    } break;
  }
  return(token);
}

unsigned char *get_token()
{
   unsigned char *ch;
   static unsigned char token[MTSIZE];
   int blank_count=0;
   int tokenptr=0;
   prevpos=curpos;
   ch=read_char();
   while(*ch<=32) {
       if( *ch == ' ' ) { isblank=1;blank_count++;}
       else if( *ch == '\n' ) isnewline = 1;
       ch=read_char();
   }
   if(*ch>='0' && *ch<='9'){
        while(*ch>='0' && *ch<='9') {
             token[tokenptr++]=*ch;
             ch=read_char();
        }
        token[tokenptr]='\0';
        unread_char();
        return(token);
   }
   if( (*ch>='a' && *ch<='z') ||  (*ch>='A' &&
*ch<='Z')){
        while((*ch>='a' && *ch<='z') ||  (*ch>='A' &&
*ch<='Z') ||
                (*ch>='0' && *ch<='9') || *ch=='_'){
                       token[tokenptr++]=*ch;
                       ch=read_char();
```

```
                         /* buralara obur file'a yazmaliyiz
*/
                         }
       token[tokenptr]='\0';
       unread_char();
       return(token);
   }
   if(strcmp(ch,OUR_EOF)==0) return(OUR_EOF);
   unread_char();
   return(return_delimeter(token));

}



floating_suffix(unsigned char *token)
{ unsigned char *p;

  p=get_token();
  if(*p=='f' || *p=='F' || *p=='l' || *p=='L')
      strcat(token,p);
  else backtrack(p);
  return(1);
}

sign(unsigned char *token)
{ unsigned char * p;

  p=get_token();
  if((*p=='-' || *p=='+') && strlen(p)==1)
strcat(token,p);
  else backtrack(p);
  return(1);
}

exponent_part(unsigned char *token)
{ unsigned char * p;

  p=get_token();
  if(strlen(p)>1 && (*p=='e' || *p=='E')) {
      strcat(token,p);
      return(1);
  }
  if(strlen(p)==1 && (*p=='e' || *p=='E')) {
     strcat(token,p);
     sign(token);
     p=get_token();
     if(*p>='0' && *p<='9') {
          strcat(token,p);
          return(1);
     } else error(5);
  } else {
```

```
      backtrack(p);
      return(0);
   }
   return(0);
}


fractional_constant(unsigned char * token)
{ unsigned char *p;
   strcpy(token,get_token());
   if(token[0]=='.'){
         p=get_token();
         if(!(*p>='0' && *p<='9')) {
             backtrack(p);
             return(0);
         }
         strcat(token,p);
         return(1);
   }
   p=get_token();
   if(*p=='.'){
       strcat(token,p);
       p=get_token();
       if(*p>='0' && *p<='9') strcat(token,p);
       else backtrack(p);
       return(1);
   } else {
        backtrack(p);
        backtrack(token);
        return(0);
   }
}

floating_constant(unsigned char *token)
{ unsigned char *p;
   unsigned char temp[MTSIZE];

   if(fractional_constant(temp)){
       exponent_part(temp);
       floating_suffix(temp);
       strcpy(token,temp);
       return(1);
   } else {
       strcpy(temp,get_token());
       if(*temp>='0' && *temp<='9')
           if(exponent_part(temp)){
              floating_suffix(temp);
              strcpy(token,temp);
              return(1);
           }else{
              backtrack(temp);
              return(0);
```

```
            }
        else {
            backtrack(temp);
            return(0);
        }
    }
}


integer_suffix(unsigned char *token)
{ unsigned char *p;

  p=get_token();
  if(*p=='u' || *p=='U' || *p=='l' || *p=='L')
      strcat(token,p);
  else backtrack(p);
  return(1);
}

hexadecimal_constant(unsigned char *token)
{ unsigned char *p,*ch;
  int tokenptr;
  strcpy(token,get_token());
  if(strlen(token)==1 && token[0]=='0'){
      p=get_token();
      if(p[0]=='x' || p[0]=='X'){
          strcat(token,p);
          integer_suffix(token);
          return(1);
      } else {
          backtrack(p);
          return(0);
      }
  } else {
      backtrack(token);
      return(0);
  }
}
decimal_constant(unsigned char * token)
{
   strcpy(token,get_token());
   if(*token>='0' && *token<='9')
        integer_suffix(token);
   else {
        backtrack(token);
        return(0);
   }
   return(1);
}

integer_constant(unsigned char *token)
{ unsigned char temp[MTSIZE];
```

```
  if(hexadecimal_constant(temp)) {
      strcpy(token,temp);
      return(1);
  } else
       if(decimal_constant(temp)){
          strcpy(token,temp);
          return(1);
       }
  return(0);
}

unsigned char *get_moken()
{ static unsigned char token[MTSIZE];
  isblank = 0;
  isnewline = 0;
  strcpy(token,get_token());
  if((*token<='9' && *token>='0') || *token=='.') {
     backtrack(token);

     if(floating_constant(token)) {
      //  isblank = 1;
      //  isnewline = 0;
        return(token);}
     else if(integer_constant(token)) {
      //  isblank = 1;
      //  isnewline = 0;
        return(token);}
     else {
      //  isblank = 0;
      //  isnewline = 0;
        return(token);}
  } else { return(token);}

}
  /*
void main() {
unsigned char * to;
fhandle=init("c:\\bc\\staff\\cem\\proto\\demo2.cpp");
while (strcmp(to=get_moken(),OUR_EOF)) {
clrscr();

if (isnewline) {printf("\n");isnewline=0;}
if (isblank) printf(" %s",to);
else printf("%s",to);



}
close(fhandle);
}

*/
```

```
/***************************/
/*INHERIT.CPPP             */
/***************************/

#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <alloc.h>
#include <mem.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>



int it_is_the_first_time=0;
long int OBJ___=1;
FILE* OBJTABLE___;
int SIZ;
int GLBL_LG=0;
long int MAXOBJ___;
long int obj_tab_size;

// routines related to object table

//**********************************



// file name where object table is located




//  OBJECT TABLE STRUCTURES



struct object_entry {

  long int offset;    // the offset of the object in the
database file
  long int object_id; // logical object identifier
  int ref_count ;     // number of objects referencing
```

```
this object

//  char name[15];          // object name if given
//  char path[25];          // the path for the file in
which object is located
//  char *type_path;     // the path for the structure
file of the object's class



} ;



struct object_table {

  long int size; // size of the object table
  struct object_entry otab[100]; // object table itself

} ;




FILE* init_obj_tab(char* name)

// initiliaze object table

{

  int i;
  struct object_table t;
  FILE*  fp;
  char b1[5];
  fp = fopen(name,"w+");
//O_CREAT|O_TEXT,S_IREAD|S_IWRITE

// initialize the entries of the object table

  ltoa(0,b1,10);
  fwrite(b1,sizeof(long int),1,fp);

  for(i=0;i<100;i++) {
     t.otab[i].object_id = 0;
        t.otab[i].offset    = 0;
     t.otab[i].ref_count = 0;

 //  strcpy((char *) t.otab[i].name, "cemo       ");
 //  strcpy((char *) t.otab[i].path,
```

```
"c:\bc\staff\cem\proto    ");
 //  strcpy((char *) t.otab[i].type_path ,
"c:\bc\staff\cem\proto");


  }

      fwrite((void *) t.otab,100*sizeof(struct
object_entry),1,fp);
      return(fp);

}



long int search_ot(long int i,FILE* fp)

{ //position the pointer in the object table file

  char b1[5];
  long int temp;


  fseek(fp,sizeof(long int) + (i-1) * sizeof(struct
object_entry) ,0);
  fread(b1,sizeof(long int),1,fp);
  b1[4]='\0';
  temp= atol(b1) ;
  cout<<"returned file offset for "<<i<<" is
"<<temp<<"\n";
  return(temp);

}


void put_ot(long int oid,FILE* fp)

{ //position the pointer in the object table file

  struct object_table t;
  long int temp;
  char b1[5];
  void * blanks;
  FILE * f;



  fseek(fp,0,0);
  fread(b1,sizeof(long int),1,fp);
  b1[4]='\0';
  obj_tab_size= atol(b1) ;
  ++obj_tab_size;
```

```
  fseek(fp,0,0);
  ltoa(obj_tab_size,b1,10);
  fwrite(b1,sizeof(long int),1,fp);

  f=fopen(DATA,"r+");
  fseek(f,0,SEEK_END);
  temp=ftell(f);



  cout <<" file offset is "<<temp<<"\n";



  t.otab[0].object_id = oid;

 // strcpy((char *) t.otab[i].name, "cemo      ");

 // strcpy((char *) t.otab[i].path,
"c:\bc\staff\cem\proto    ");

 //  strcpy((char *) t.otab[i].type_path ,
"c:\bc\staff\cem\proto");

  t.otab[0].offset    = temp;

  t.otab[0].ref_count = 1;

  fseek(fp,sizeof(long int) + (oid-1) * sizeof(struct
object_entry) ,0);
  ltoa(temp,b1,10);
  fwrite(b1,sizeof(long int),1,fp);
  ltoa(oid,b1,10);
  fwrite(b1,sizeof(long int),1,fp);
  ltoa(1,b1,10);
  fwrite(b1,sizeof(long int),1,fp);
  blanks=malloc(SIZ);
  fseek(f,temp,0);
  memset(blanks,(int) ' ', SIZ);

  cout<<fwrite (blanks,SIZ,1,f)<<"is the no of bytes\n";

  free(blanks);

  cout<<oid<<" was put in . size is "<<
obj_tab_size<<"\n" ;

  fclose(f);

  f=fopen(DATA,"r+");

  fseek(f,0,SEEK_END);
```

```
  temp=ftell(f);

  cout<<"sizeof data file is"<<temp<<"\n";

  fclose(f);


}


//******************************


 void moods_put(long int o,int offset,char *buf,int s)

{

 FILE* temp;

 temp=fopen(DATA,"r+");
 fseek(temp,(o-1)*SIZ+offset,0);
 fwrite(buf,s,1,temp);
 fclose (temp);

}


 void moods_get(long int o,int offset,char *buf,int s)

{

 FILE* temp;

 temp=fopen(DATA,"r+");
 fseek(temp,(o-1)*SIZ+offset,0);
 fread(buf,s,1,temp);
 fclose(temp);

}

//*******************************************
// overloadings
//*******************************************

class pint {

public:
```

```
int mainfield;
long int offset;
int length;

pint operator = (int data) {
                  char buf[sizeof(int)+1];
                  long int off;


                           this->mainfield=data;
                  memcpy(buf, (char *) this,sizeof(int));
                  buf[sizeof(int)]='\0';
                  off=search_ot(OBJ___,OBJTABLE___);

moods_put(1,off+offset,buf,sizeof(int));

                  cout<<" offset is "<<off<< "\n" ;

                  return(*this);

}



pint operator * () {

                  char buf[sizeof( int)+1];
                  long int off;

                  off=search_ot(OBJ___,OBJTABLE___);
                  moods_get(1,off+offset,buf,sizeof(
int));
                  buf[sizeof( int)]='\0';
                  memcpy((char *) this,buf,sizeof(int));
                  return(*this);

}

//ostream& pint::operator << (pint data) ;



};
/*
ostream& pint::operator << (pint data) {
                //  ostream & s;
                 cout << ((*data)).mainfield;
                 return(cout);

}

*/
```

```
class pfloat {

public:

float mainfield;
long int offset;
int length;

pfloat operator = (float data) {
                  char buf[sizeof(float)+1];
                  long int off;

                  this->mainfield=data;
                  memcpy(buf, (char *)
this,sizeof(float));
                  buf[sizeof(float)]='\0';
                  off=search_ot(OBJ___,OBJTABLE___);

moods_put(1,off+offset,buf,sizeof(float));
                  return(*this);

}


pfloat operator * () {
                  char buf[sizeof( float )+1];
                  long int off;

                  off=search_ot(OBJ___,OBJTABLE___);

moods_get(1,off+offset,buf,sizeof(float));
                  buf[sizeof(float)]='\0';
                  memcpy((char *)
this,buf,sizeof(float));
                  return(*this);

}


};



class pchar {

public:

char mainfield;
long int offset;
int length;
```

```
pchar operator = (char data) {
                  char buf[sizeof(char)+1];
                  long int off;

                  off=search_ot(OBJ___,OBJTABLE___);
                  this->mainfield=data;
                  memcpy(buf, (char *)
this,sizeof(char));
                  buf[sizeof(char)]='\0';
                  cout<<"char ";

moods_put(1,off+offset,buf,sizeof(char));
                  return(*this);

}

pchar operator * ();

};


pchar pchar::operator * () {

                  char buf[sizeof(char)+1];
                  long int off;

                  off=search_ot(OBJ___,OBJTABLE___);

moods_get(1,off+offset,buf,sizeof(char));
                  buf[sizeof(char)]='\0';
                  this->mainfield=buf[0];
                  return(*this);
}

//*****************************************************
// pcharptr0 and pcharptr1 are for char * declaration
// they are automatically pasted at compile time

class pcharptr0 {

public:

char mainfield;
long int offset;

pcharptr0 operator = (char data) {

                  char buf[sizeof(char)+1];

                  this->mainfield=data;
                  memcpy(buf, (char *)
```

```
this,sizeof(char));
                    cout<<this->mainfield<<"kkkkkkkk";
                    buf[sizeof(char)]='\0';

moods_put(1,this->offset,buf,sizeof(char));
                    cout<<this->offset<<" is the file
offset \n";
                    return(*this);
}

};


class pcharptr1  {

public:

long int offset;
int length;

pcharptr1 operator + ( int dat ) {

                   GLBL_LG=dat;
               //       this->length=dat;
                   return(*this);
}

pcharptr0 operator * () {

                    long int  temp,off,max_length;
                    char buf1[sizeof(char)+1];
                    char buf2[sizeof(long int)+1];
                    pcharptr0 retch;

                    // find OBJ___'s offset from object
table
                    off=search_ot(OBJ___,OBJTABLE___);
                    cout<<"offset of obj is "<<off<<"\n";
                    // then 1,real__offset+offset+length
field
                    moods_get(1,offset+off,buf2,sizeof(long
int));
                    buf2[sizeof(long int)]='\0';
                    temp=(atol(buf2));
                    cout<<"offset of ptr is "<<temp<<"\n";
                    moods_get(1,temp,buf2,sizeof(long
int));
                    buf2[sizeof(long int)]='\0';
                    max_length=(atol(buf2));
                    cout<<"length is "<<max_length<<"\n";

                    if (max_length<GLBL_LG /*
```

```
this->length*/)
                        {printf("allocation
error\n");exit(1);}
                     else
                        {moods_get(1,temp+GLBL_LG
/*this->length*/+sizeof(long int),buf1,sizeof(char));
                        retch.mainfield=buf1[0];
                        cout<<"*********"<<retch.mainfield;
                        retch.offset=temp+GLBL_LG
/*this->length*/+sizeof(long int);
                        cout<<this->length<<" len
"<<retch.mainfield<<"ula ";

                     }
                     cout<<this->length<<" currrrt leng \n";
                     this->length=0;
                     cout<<this->length<<" currrrt leng \n";
                     GLBL_LG=0;
                     return(retch);

}
pcharptr1() { }
pcharptr1(int order, long int s=16) {

                long int temp , off;
                char buff[5];
                FILE* fp;
                void * blanks;
                // will come as a parameter from
               // the constructor modification
                 this->offset= order ;

                if (it_is_the_first_time) {
                off=search_ot(OBJ___,OBJTABLE___);
                cout <<"construuuuuuu\n"<<"ofs of obj
"<<off<<"\n";
                fp=fopen(DATA,"r+");
                fseek(fp,0,SEEK_END);
                temp=ftell(fp);
                ltoa(temp,buff,10);
                fseek(fp,off+offset,0);
                cout << temp<<" is put to
"<<off<<"+"<<offset<<" for ptr\n";
                fwrite(buff,sizeof(long int),1,fp);
                fseek(fp,0,SEEK_END);
                blanks=malloc(s*sizeof(char));
                memset(blanks,(int) ' ', sizeof(char)*s);
                fwrite (blanks,sizeof(char)*s,1,fp);
                free(blanks);
                // initialize the length of the pointer
field
                fseek(fp,temp,0);
```

```
                    ltoa(s-sizeof(long int),buff,10);
                    fwrite(buff,sizeof(long int),1,fp);
                    fclose(fp);
                     }

                   }
};



class vma___ {

public:

FILE * generator;
FILE * objtab;
long int objid;
int dirty;

vma___() {}
vma___(char * gen) {
          char buff[5];
          FILE* temp,*ot;

          temp=fopen(gen,"rt+");
          generator=temp;
          dirty=1;
          fread(buff,sizeof(long int),1,temp);
          fseek(temp,0,0);
          buff[4]='\0';
          objid = atol(buff) ;
          ++objid;
          ltoa(objid,buff,10);
          fwrite(buff,sizeof(long int),1,temp);
          fclose(temp);

             //   used once

             if(!CREAT_OR_OPEN) // means open used
              temp=init_obj_tab(OBJTAB);
             else
              temp= fopen(OBJTAB,"r+") ;
//O_CREAT|O_TEXT,S_IREAD|S_IWRITE
             objtab=temp;
             OBJ___=objid;
             OBJTABLE___=objtab;

        }


} ;
```

```
template <class T>

class moods___:  public  T , public vma___ {

public:

moods___() {

                SIZ=sizeof(T);
                vma___(FHANDLE);
                objid=OBJ___;
                MAXOBJ___=OBJ___;
                objtab=OBJTABLE___;
                put_ot(OBJ___,OBJTABLE___);
                it_is_the_first_time=1;
                T() ;
                cout<<"ooooo"<<grade.offset;
                it_is_the_first_time=0; }


moods___ *  operator -> () {

                   OBJ___=objid;
                   OBJTABLE___=objtab;
                   return ( this) ;}

moods___   operator * () {

                   OBJ___=objid;
                   OBJTABLE___=objtab;
                   return *this;
                   }

};

#define EX_SIZE  20  //bytes may be specified by the user
```

```
/***************************/
/*MCOMP2.CPP              */
/***************************/

#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <ctype.h>
#include "C:\USER\CEM\PROTO\TOKEN\LEX.CPP"
#include <process.h>
#include <stdlib.h>

#define MAXVARSIZE 25

unsigned char * token_ ,*CLvar,*CLname;
int exist_flag;
char  dbn[80];
int PFLAG;
```

```
char DATA[80];


void ConvertToPoint(char *);



// create a new object producer
void datcrat(char * fname)
{
 int i;
 FILE *temp;
 temp=fopen(fname,"wt");
 fwrite("0000",sizeof(long int),1,temp);
 fclose(temp);
}


//  OBJECT TABLE STRUCTURES


struct object_entry {

  long int offset;    // the offset of the object in the
database file
  long int object_id; // logical object identifier
  int ref_count ;     // number of objects referencing
this object

//  char name[15];          // object name if given
//  char path[25];          // the path for the file in
which object is located
//  char *type_path;    // the path for the structure
file of the object's class


} ;


struct object_table {

  long int size; // size of the object table
  struct object_entry otab[100]; // object table itself

} ;
```

```
FILE* init_obj_tab(char* name)

// initiliaze object table

{

  int i;
  struct object_table t;
  FILE*  fp;
  char b1[5];
  fp = fopen(name,"w+");
//O_CREAT|O_TEXT,S_IREAD|S_IWRITE

// initialize the entries of the object table

  ltoa(0,b1,10);
  fwrite(b1,sizeof(long int),1,fp);

  for(i=0;i<100;i++) {
     t.otab[i].object_id = 0;
        t.otab[i].offset     = 0;
     t.otab[i].ref_count = 0;

 //  strcpy((char *) t.otab[i].name, "cemo        ");
 //  strcpy((char *) t.otab[i].path,
"c:\bc\staff\cem\proto    ");
 //  strcpy((char *) t.otab[i].type_path ,
"c:\bc\staff\cem\proto");


  }

      fwrite((void *) t.otab,100*sizeof(struct
object_entry),1,fp);
      return(fp);

}




void keep_org_cons(char *fname)
{
 FILE * temp;
 int parcnt,finished;
 char  current[30],prevtok[30];

 temp=fopen("cnstr.mds","wt");
```

```
 fhandle=init(fname);
 *prevtok=0;
 finished=0;
 parcnt=0;

 while(strcmp(strcpy(current,get_moken()),OUR_EOF)&&
!finished ) {
      if ( !strcmp(prevtok,"::") &&
!strcmp(current,CLname)) {
          fprintf(temp,"%s::%s",CLname,CLname);
          do {
              strcpy(current,get_moken());
              if (!strcmp(current,"{")) {
                parcnt++;
                if (!parcnt) finished=1;
              }
              if (!strcmp(current,"}")) {
                parcnt--;
                if (!parcnt) finished=1;
              }
                if (isnewline) fprintf(temp,"\n");
           if ( isblank ) fprintf(temp," %s",current);
           else fprintf(temp,"%s",current);


          } while  ( (!finished)) ;

      }
      else
       {
         strcpy(prevtok,current);
       }


 }
fclose(temp);
close(fhandle);
}

void ChangeConstructor(char * fname)
{
      char prevtok[80],current[64],temp2[64];
      FILE *temp,*temp3;

      fhandle = init(fname);
      *prevtok=0;
      temp=fopen("temp1.mds","wt");

      while(strcmp(strcpy(current,get_moken()),OUR_EOF) )
{
```

```
          if (isnewline) fprintf(temp,"\n");
          if ( isblank ) fprintf(temp," %s",current);
          else fprintf(temp,"%s",current);

          if ( !strcmp(prevtok,"::") &&
!strcmp(current,CLname)) {
             //append ___ to the CLname
             fprintf(temp,"___ ");
             while(strcmp(strcpy(temp2,get_moken()),"{"))
{

                if (isnewline) {
                                fprintf(temp,"\n");
                           }
                if ( isblank ) {
                               fprintf(temp," %s",temp2);
                          }
                else {
                              fprintf(temp,"%s",temp2);
                          }

             }
                     if (isnewline) {
                                fprintf(temp,"\n");
                           }
                if ( isblank ) {
                               fprintf(temp," %s",temp2);
                          }
                else {
                              fprintf(temp,"%s",temp2);
                          }

             temp3=fopen("ofsets.out","rt");
             while(fgets(prevtok,80,temp3))
fputs(prevtok,temp);
             fclose(temp3);
             *prevtok=0;
          }
          else
             strcpy(prevtok,current);
     }
     fclose(temp);
     close(fhandle);


// a second pass *********** short of time

     fhandle = init("temp1.mds");
     *prevtok=0;
     temp=fopen("temp.mds","wt");
```

```
     while(strcmp(strcpy(current,get_moken()),OUR_EOF) )
{

          // print previous token if not null

     if ( !strcmp(prevtok,CLname) &&
!strcmp(current,"::"));
     else {
          if (isnewline) fprintf(temp,"\n");
          if ( isblank ) fprintf(temp," %s",current);
          else fprintf(temp,"%s",current);
      }

          if ( !strcmp(prevtok,CLname) &&
!strcmp(current,"::")) {
               //append ___ to the CLname
               fprintf(temp,"___ ");
               if (isnewline) fprintf(temp,"\n");
               if ( isblank ) fprintf(temp," %s",current);
               else fprintf(temp,"%s",current);
               *prevtok=0;
          }
          else
               strcpy(prevtok,current);
     }
     fclose(temp);
     close(fhandle);



}


void change_type()
{
  FILE* st,*temp;
  char prevtok[80], temp2[80],current[80];
  int flag;

  st= fopen("str.mds","rt");
  temp= fopen("type.mds","wt");
  *prevtok=0;


  while(strcmp(strcpy(current,get_moken()),OUR_EOF) ) {


          if (isnewline) fprintf(temp,"\n");
          if ( isblank ) fprintf(temp," %s",current);
          else fprintf(temp,"%s",current);
```

```
            if ( !strcmp(prevtok,"class") &&
!strcmp(current,CLname)) {
                //append ___ to the CLname
                fprintf(temp,"___ ");
                // skip till left curly
                while (strcmp(strcpy(temp2,get_moken()),"{"
) ) ;

                if (isnewline) fprintf(temp,"\n");
                if ( isblank ) fprintf(temp," %s",temp2);
                else fprintf(temp,"%s",temp2);
                fprintf(temp,"\npublic:\n" );
                // write the fields; and function prototypes
                // first write the fields
                flag=1;
                while (! feof(st)) {
                char  str1[50],str2[50],*dec,*ptr1,*ptr2;
                int diff;
                if (fscanf(st,"%s",str1)==EOF) flag=0;
                if (fscanf(st,"%s",str2)==EOF) flag=0;
                if ( flag) {
                ptr1=strstr(str2,"(");
                ptr2=strstr(str2,")");
                diff=ptr2-ptr1;
                dec=(char *) malloc(diff*sizeof(char)+1);
                strcpy(dec,"");
                strncat(dec,ptr1+1,diff-1);
                fprintf(temp,"\n%s %s ;",dec,str1);
                  }
                }
                // now paste the function prototypes
                // skip till right curly }******
                    fprintf(temp,"\n%s___();\n",CLname);
                    while
(strcmp(strcpy(temp2,get_moken()),"}" ) )  ;
              fprintf(temp," } ");
              *prevtok=0;
        }
        else
           strcpy(prevtok,current);



  }
 fclose(temp);
 fclose(st);

}


void FindOffset ()
{
```

```
   char *temp1, *temp2,temp3[100],temp4[100],
*temp5,mytemp[80],mytemp1[80];
   FILE *fp,*out;
   int x;

   fhandle=init("str.mds");

   out= fopen("ofsets.out","w");
   strcpy(temp4,get_moken());
   temp1 =(char*) malloc(strlen(temp4)+1);
   strcpy(temp1,temp4);
   //********************************************
   // if the token is pointer then call constructor
   //********************************************

   fprintf(out,"%s.offset=0;\n",temp4);
   sprintf(temp4,"%s",get_moken());
   if (isdigit(*temp4)) {
       strcat(temp4,get_moken());
       strcat(temp4,get_moken());
   }
   strcat(temp4,get_moken());
   strcpy(mytemp,get_moken());
   strcpy(mytemp1,((char *) mytemp+1));
   strcat(temp4,mytemp1);
   strcat(temp4,get_moken());


   temp2 =(char*) malloc(strlen(temp4)+1);
   strcpy(temp2,temp4);
   if(strstr(temp2,"ptr")) {
      fprintf(out,"%s.length = 0;\n",temp1);
         // print out constructor call
         // ilktoken.pxptrn::pxptrn(0);

fprintf(out,"%s.%s::%s(%d);\n",temp1,mytemp,mytemp,0);

   }
   free(temp1);
   while(strcmp(strcpy(temp4,get_moken()),OUR_EOF)) {
          temp1=(char*) malloc(strlen(temp4)+1);
          strcpy(temp1,temp4);
          fprintf(out,"%s.offset= %s;\n",temp1,temp2);
          sprintf(temp3,"%s",get_moken());
          if (isdigit(*temp3)) {
             strcat(temp3,get_moken());
             strcat(temp3,get_moken());
          }
          strcat(temp3,get_moken());
                strcpy(mytemp,get_moken());
                strcpy(mytemp1,((char *) mytemp+1));
                strcat(temp3,mytemp1);
```

```
          strcat(temp3,get_moken());

                if(strstr(temp3,"ptr")) {
            fprintf(out,"%s.length = 0;\n",temp1);

fprintf(out,"%s.%s::%s(%s);\n",temp1,mytemp,mytemp,temp2)
;

                }
          free(temp1);
          temp5=(char
*)malloc(strlen(temp2)+strlen(temp3)+2);
          strcpy(temp5,temp2);
          x =strlen(temp2);
          temp5[x]='+';
          temp5[x+1]=0;
          free(temp2);
          temp2=temp5;
          strcat(temp2,temp3);
   }
   free(temp2);
   fclose(fp);
   fclose(out);
   close(fhandle);
}


void semicolon()
{
  if(strcmp( (token_ = get_moken() ),";")) {
      printf("missing ; ");
      exit(1);
  }
  free(token_);
}

unsigned char * db_name()
{
  unsigned char *dbname;
  if ( !strcmp( (token_ = get_moken()), "<" ) ) {
       free(token_);
       dbname = get_moken(); /* argument 2 */
       if( strcmp( (token_ = get_moken()), ">")) {
             printf("missing > ");
             exit(1);
       }
       free(token_);
   }
   else {
       // skip till semicolon
       while(strcmp(token_,";")) {
```

```
            token_ = get_moken();
            free(token_);
        }
        return(NULL);
    }
    return(dbname);
}

unsigned char * path_name()
{
  unsigned char *pathname;
  if ( !strcmp( (token_ = get_moken()), "(" ) ) {
        free(token_);
        pathname = get_moken(); /* argument 2 */
        if(strcmp( (token_ = get_moken()), ")")) {
              printf("missing ) ");
              exit(1);
        }
        free(token_);
    }
    else {
        free(token_);
        printf("missing (");
        exit(1);
    }
    return(pathname);
}

void main(int argc,char * argv[])

{
  FILE * dest;
  FILE * classes;
  FILE * structure;
  FILE * name_to;
  char * tork;
  int parcnt,first_time,finished;



  clrscr();
 if (argc != 3) { printf( " Filename Expected");
exit(1);}
//  strcpy(argv[1],"demo1.cpp");
  // initializations
  fhandle=init(argv[1]);
  PFLAG=1;
  CLvar=(unsigned char *) malloc(MAXVARSIZE);
  CLname=(unsigned char *) malloc(MAXVARSIZE);

  classes=fopen("cls.mds","wt");
//  classes=fopen("cls.mds","wt");
```

```
  while (abs(stricmp((token_=(unsigned char *)
get_moken()),OUR_EOF))) {
        if( strcmp(token_,"class")== 0) {
            parcnt=0;
            first_time=0;

            if (strcmp(token_,"*")==0)
                 fprintf(classes,"%s",token_);
            else {

                    if (isnewline) fprintf(classes,"\n");
                    if ( isblank ) fprintf(classes,"
%s",token_);

                    else fprintf(classes,"%s",token_);
                  }
            do {
                 token_=get_moken();
                 if (strcmp(token_,"*")==0)
                         fprintf(classes,"%s",token_);
                 else {

                         if (isnewline)
fprintf(classes,"\n");
                         if ( isblank ) fprintf(classes,"
%s",token_);
                         else
fprintf(classes,"%s",token_);
                  }
                 if (strcmp(token_,"{")==0) { ++parcnt;
                          first_time=1;
                 }
                 if (strcmp(token_,"}")==0) { --parcnt;
                 }

            } while (parcnt || (!first_time));
            // skip till the semicolon;

            do {
            token_=get_moken();
            } while (strcmp(token_,";")!=0);
            fprintf(classes,";\n");

        }
  }
  fclose(classes);
  // find persistent variables and classes
  fhandle=init(argv[1]);
  finished=0;
  while ((abs(stricmp((token_=(unsigned char *)
get_moken()),OUR_EOF)))
           &&
```

```
            (! finished)) {

    if (strcmp(token_,"persistent")==0) {
        if (strcmp(token_=get_moken(),"<")==0) {
             get_moken();
             if (strcmp(token_=get_moken(),">")==0) {
              unsigned char * temp;
              unsigned char * var1;
              unsigned char * var2;

              temp = get_moken();
              var1 = (unsigned char *)
malloc(strlen(temp)+1);
              strcpy(var1,temp);

              temp = get_moken();
              var2 = (unsigned char *)
malloc(strlen(temp)+1);
              strcpy(var2,temp);

              if (strcmp(var2,";")==0) {
                  int found;
                  unsigned char * temp;
                  unsigned char * name1;


                  found=0;
                  strcpy (CLvar,var1);
                  // then find the corresponding class
name
                  // *s are excluded
                  close(fhandle);
                  fhandle = init (argv[1]);

                  temp = get_moken();
                  name1 = (unsigned char *)
malloc(strlen(temp)+1);
                  strcpy(name1,temp);

                  while (! found) {

                       if
((strcmp(token_=get_moken(),CLvar)==0)
                           &&

(strcmp(token_=get_moken(),";")==0))
                            {
                               strcpy(CLname,name1);
                               found=1;
                            }
                       name1 = (unsigned char *)
malloc(strlen(token_)+1);
```

```
                    strcpy(name1,token_);

                }

                finished=1;
            }
            else {
                strcpy (CLvar,var2);
                // check for stars

                strcpy (CLname,var1);
                finished=1;
            }


          }
      }
   } // end of main if

  } // end of while
  close (fhandle);

  // parse for inheritance topology
  name_to=fopen("name.mds","wt");
  fprintf (name_to,"%s",CLname);
  fclose(name_to);


spawnl(P_WAIT,"C:\USER\CEM\PERDEMO\CLASSPAR\newest.exe",N
ULL);


  //offset for CLname of CLvar


  // now from str.mds calculate the field offsets;
  // put them in a global array
  // find persistent class name put "p" in front of the
  // atomic fields
  // make persistent variables prefixed by "moods_"
  // replace keywords if new everything is initialized
once
  FindOffset();
  ChangeConstructor(argv[1]);


  //*************** change atomic types ***********
  //                  for the class
  fhandle=init("temp.mds");
  change_type();
  close(fhandle);
```

```
fhandle=init("type.mds");

// first make a pass from the code
// to scan open create and to learn the
// database name
//****************************************
exist_flag=0;
while (strcmp(token_=get_moken(),OUR_EOF)) {
     if (strcmp(token_,"open")==0)

     {
        if (!strcmp(token_=get_moken(),"<"))
           {
               get_moken();//handle
               get_moken();// >
               get_moken();// (
               strcpy(dbn,get_moken());// path
               exist_flag=1;
           }

     }
     else
      if (strcmp(token_,"create")==0)
           {
        if (!strcmp(token_=get_moken(),"<"))
           {
               get_moken();//handle
               get_moken();// >
               get_moken();// (
               strcpy(dbn,get_moken());// path
           }


           }

}

close(fhandle);

dest=fopen("perout.mds","wt");

tork=strstr(dbn,".");
strcpy(tork+1,"oid\"");
fprintf(dest,"char * FHANDLE = %s ;\n",dbn); //objid
strcpy(tork+1,"dbf\"");
strcpy(DATA,dbn);
fprintf(dest,"char * DATA = %s ;\n",dbn);     //
database file
strcpy(tork+1,"tab\"");
fprintf(dest,"char * OBJTAB = %s  ;\n",dbn); // object
table
fprintf(dest,"#define CREAT_OR_OPEN  %s  %d  %s \n","
```

```
",exist_flag,"  \n");
  fprintf(dest,"#include \"inherit.cpp\" \n");

  fprintf(dest,"#include \"cls.mds\" \n");
  keep_org_cons(argv[1]);
  fprintf(dest,"#include \"cnstr.mds\" \n");

  fhandle=init("type.mds");
  while (abs(stricmp((token_=(unsigned char *)
get_moken()),OUR_EOF))) {

      // PERSISTENT
        if (strcmp(token_,"persistent")==0) {
            int per;
            per=0;
            // get persistent variable and its respective
class name
            // if real persistent clause skip till the
semi..
            token_=get_moken();
            if (strcmp(token_,"<")==0) per=1;
            else // write to destination file
             {
               fprintf(dest," persistent",token_);

               if (isnewline) fprintf(dest,"\n");
               if ( isblank ) fprintf(dest," %s",token_);
               else fprintf(dest,"%s",token_);
             }
            while (abs(strcmp(token_,";")) && per)
                token_= get_moken();
                 // search parsed class structures
                 // Let CLname = student and CLvar = st
                 // put moods_ in front of the variable
                fprintf(dest,"moods___< %s___ >  %s ;
\n",CLname,CLvar);
             //      fprintf(dest, "SIZ = sizeof( %s___)
; \n",CLname);
                 // output : moods___<student___> st ;
NewLine
                 // template <T> where t is CLname
          }

          // NEW INSTANCE
          else if (!strcmp(token_,"new_instance")) {
             char *dbname, *classname, *varname ;
             dbname = db_name();
             classname = get_moken();
             if(!strcmp( (token_ = get_moken() ), ";")) {
                 free(token_);
                 varname = classname ;
                 classname = NULL;
```

```
        }
        else {
            varname = get_moken();
            semicolon();
        }
        // INSERT CEM'S CODE HERE
        free(dbname);
        free(classname);
        free(varname);
    }

    // DATABASE
    else if (!strcmp(token_,"database")) {
        char *dbname;
        dbname = db_name();
        semicolon();
        // INSERT CEM'S CODE HERE

        free(dbname);
    }

    // CREATE
    else if (!strcmp(token_,"create")) {
           char dbname[80], *pathname;
           FILE * forfirst;

           strcpy(dbname,db_name());
           pathname = path_name();
           semicolon();
           // INSERT CEM'S CODE HERE
           strcpy(dbname,dbn);
           tork=strstr(dbname,".");
           strcpy(tork+1,"oid");
           datcrat(dbname+1);
           strcpy(dbname,dbn);
           tork=strstr(dbname,".");
           strcpy(tork+1,"tab");
           fclose(init_obj_tab(dbname+1));
                        strcpy(dbname,dbn);
           tork=strstr(dbname,".");
           strcpy(tork+1,"dbf");
               forfirst=fopen(dbname+1,"wt+");
                        fclose(forfirst);
    //     fprintf(dest,"\nOBJTABLE___=init_obj_tab(
OBJTAB );\n");
    //     fprintf(dest,"\nOBJ___=1;\n");

//         fprintf(dest,"\n
           free(dbname);
           free(pathname);
    }
```

```
          // OPEN
          else if (!strcmp(token_,"open")) {
               char *dbname, *pathname;
               dbname = db_name();
               pathname = path_name();
               semicolon();

               // INSERT CEM'S CODE HERE
//*******************************
/*
               fclose(init_obj_tab(dbname+1));
               fclose(fopen(DATA,"wt"));
               fprintf(dest,"\nOBJTABLE___=init_obj_tab(
OBJTAB );\n");
               fprintf(dest,"\nOBJ___=1;\n");
  */
  //*************************************************




               free(dbname);
               free(pathname);
          }

          // CLOSE
          else if (!strcmp(token_,"close")) {
               char *dbname;
               dbname = db_name();
               semicolon();
               // INSERT CEM'S CODE HERE
               if (isnewline) fprintf(dest,"\n");
               if (isblank) fprintf(dest,"
%s","fcloseall();\n");
               else fprintf(dest,"%s","fcloseall();\n");
               free(dbname);
          }

          // UNPERSIST
          else if (!strcmp(token_,"unpersist")) {
               char *dbname,*varname;
               dbname=db_name();
               varname=get_moken();
               semicolon();
               // INSERT CEM'S CODE HERE
               free(dbname);
               free(varname);
          }

          // DELETE_FUNCTION
          else if (!strcmp(token_,"delete_function")) {
```

```
            }

            // ACCESS
            else if (!strcmp(token_,"access")) {
                 char dbname[80],
classname[80],varname[80],seqno[80];
                 strcpy(dbname, db_name());
                 strcpy(classname, get_moken());
                 if(!strcmp((token_ = get_moken()),";")) {
                    strcpy(varname, classname);
                    *classname =0;
                    free(token_);
                 }
                 else {
                    strcpy(varname, token_);
                    strcpy(seqno, get_moken());
                    semicolon();
                 }
                 // INSERT CEM'S CODE HERE
                          // change objid

fprintf(dest,"%s.objid=%s;\n",varname,seqno);

fprintf(dest,"OBJ___=%s;\n",seqno);


                 free(dbname);
                 free(classname);
                 free(varname);
                 free(seqno);
            }
            else // tokens to be directly pasted
            {

                 if (isnewline) fprintf(dest,"\n");
                 if ( isblank ) fprintf(dest," %s",token_);
                 else fprintf(dest,"%s",token_);

            }
        }
  fclose(dest);
  close(fhandle);
  // . becomes -> or ,s becomes (*s)
  if (PFLAG) ConvertToPoint(argv[2]);


}

void ConvertToPoint(char * fname) {
unsigned char *Instr,*prevtok;
FILE *dest;
int toto = 0;
```

```
dest = fopen(fname,"wt");

// Get Moken icin initialization
fhandle=init("perout.mds");
prevtok="\0";
while (strcmp(Instr=get_moken(),OUR_EOF)) {
        if (!strcmp(Instr,".")) {
            if (!strcmp(CLvar,prevtok))
              { fprintf(dest,"%s","->");prevtok="\0"; }
            else
              { fprintf(dest,"%s",".");prevtok="\0";}
        }
        else {
              if (toto==1) { isnewline=1;toto=0;}
              if ( !strcmp(prevtok,"CREAT_OR_OPEN"))
                       { toto=1;isblank=1;isnewline=0;}
              if (isnewline) fprintf(dest,"\n");
              if(*Instr=='#') isblank=0;
              if ( isblank ) fprintf(dest," %s",Instr);
              else fprintf(dest,"%s",Instr);

              prevtok = (char *) malloc(strlen(Instr)+1);
              strcpy(prevtok,Instr);

             }
 }
 fclose(dest);
 close(fhandle);
}
```