A Preprocessor Approach to Persistent C++

Cem Evrendilek, Asuman Dogac and Tolga Gesli

Software Research and Development Center
Scientific and Technical Research Council of Türkiye
E-Mail:asuman@vm.cc.metu.edu.tr

EXTENDED ABSTRACT

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 to lose their manipulation and access semantics since the objects with different declarations may have the same storage representation. In this work persistence is added to C++ in DOS environment through a preprocessor and a class library developed in C++, such that the access and manipulation semantics of objects are preserved. The new language is called C**. The disk management of objects declared as persistent are automatically handled by the system through a virtual memory management emulation. Persistency is implemented as a storage class that is completely orthogonal to type. In other words, persistency is a property of objects, not their classes. Language changes are kept to a minimum, thus among the existing persistent 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. Upward compatibility with C++ is preserved. The hybrid object identifier (OID) mechanism implemented in C** enables dynamic clustering and reduction in the object table size. Although there are several other persistent C++ implementations, the implementation technique of C** is original in that it provides the user with transparent type modifications and uses operator overloading extensively in realizing persistency. To the best of our knowledge C** is the first persistent C++ implementation on DOS with persistence as a storage class.

INTRODUCTION

There are certain principles of persistence that should govern language design as stated in [AB 87]:

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. This is referred to 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 the 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 movement of data by the programmer are called persistent programming languages. Discovering engineering techniques to support persistence for arbitrarily complex structures is a difficult task [AB 87].

In [Atw 90] 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). C** is implemented in DOS environment as a preprocessor to C++. To the best of our knowledge C** is the first persistent C++ implementation on DOS with persistence as a storage class. There are other persistent C++ implementations [LLOW 91],[CDRS 89], [RC 89], [SCD 90], [D 91], [AG 89] however these implementations either do not support SC approach or do not have the advantages cited above. The detailed comparison of these systems with C** is provided in [EDG 94].

The overall system works as follows. A program written in 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. Note that DOS places restriction on the amount of main memory available to an application making it impossible to deal with large objects with sizes greater than a segment. To alleviate this problem a virtual memory emulation has been built into the C** Storage System [Say 92].

The contributions of C** are as follows:

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

2. C** requires the minimum coding effort to develop an application in persistent C++. The only thing necessary is to declare 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 [RC 89], classes prefixed with db 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 counterparts. In order to carry an existing application

to O++ [AG 89], it is necessary, first of all, to define pointers to persistent objects since the only way to reference persistent objects is through the use of 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. When porting an application developed in C++ to O_2 environment, "import from C++" utility 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.

3. 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 also 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.

4. 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 O_2 for example, when a C++ object that has data members with more than one level of pointer indirections is to be made persistent, such data members have to be modelled using list or set structures supplied by O_2.

5. The hybrid OID mechanism implemented in C** enables both dynamic clustering and reduction in the object table size.

6. Objects of any complexity with arbitrary level of pointer indirections to any type of object is supported.

7. Another advantage of using this preprocessing approach is to be able use standart compilers.

IMPLEMENTATION OF C**

Some Considerations about C++

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 [Atw 90].

C++ allows methods and some operators to be overloaded. This presents the ability to use the same methods and operators with different semantics for different data types. We have chosen method and operator overloading to access the persistent and the transient objects uniformly using the same operators and methods. Overloading has a problem at the implementation level: Not every operator can be overloaded and also compile time information about classes of which an persistent instance is created, 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 (=), and subscript operator ([]). However :?, ::, . operators can not be overloaded in C++. Scope resolution operator and :? is handled by the class parser. Dot operator (.) is handled by the preprocessor by replacing it with -> context sensitively.

It should also be noted that scope rules of C++ do not apply for such an object since the scope of a persistent object is the whole application regardless of in which block it is declared.

The Design Philosophy

In order to obtain a prototype rapidly, we have decided to base the design on overloading and this resulted in the preprocessor approach.

The physical representation chosen for persistent objects is similar to their memory representations in that the notion of pointers is preserved.

We have assumed that the life time of a persistent object is the duration from its declaration as persistent till it is explicitly deleted by the programmer. Thus the scope rules of C++ do not apply to persistent objects.

The OID mechanism implemented in C** is such that persistent objects have logical OIDs whereas pointer fields of persistent objects are referenced by their physical addresses. Such fields are termed as persistent values. Thus the size of object table is reduced since persistent value references are not recorded in the object table. This approach is motivated by observing the fact that clustering becomes critical for the performance of object-oriented databases. If the clustering of persistent values becomes essential for the system performance, then a dummy persistent object of the desired type can be declared so that, all such fields automatically get a logical OID. This feature provides a level of control over clustering.

General Structure of the System

The general structure of the implementation is as shown in Figure 1. The source file in C** 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' types. 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 overloaded operators defined in the file called overloader. Environment files, which form the object base, created by the preprocessor store the necessary information for persistent objects to be used during preprocessing and at run time by the system.

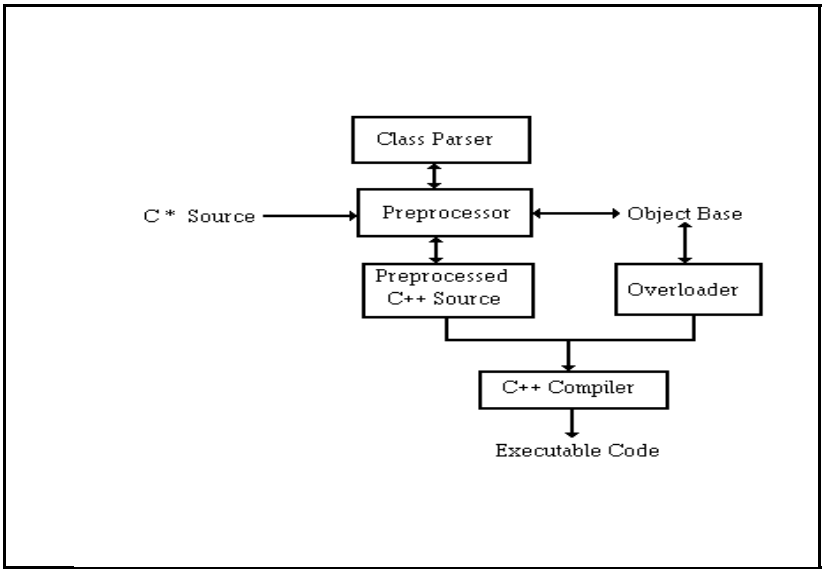

Figure 1. General Overview of Implementation

STORAGE MANAGEMENT IN C**

C** Storage System [Say 92], which is a subset of Exodus Storage Manager [CDRS 89], is composed of a buffer manager, an object table manager, a persistent variable directory manager and a disk manager.

Persistent Value Management in C**

Forward marking is used for managing persistent values. Although this technique is used in storage systems that use physical addresses, in our implementation it handles the relocation of persistent values.

When a persistent value grows too big to fit in the available free space following its current location, it may be moved to another location, and a marker is left in its place. At every place a persistent value starts, there exists a length field. If the length field is negative, the following bytes will give the new address for the persistent value. 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. With forward marking two disk accesses are required in the worst case for persistent values.

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.

PREPROCESSOR

C** Clauses

1. object_base <pob-name> ;

This statement declares a logical handle for an object base. This handle is then used for associating a physical path name through open and create statements to pob-name. 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 associated with the given pathname.

3. open <pob-name> (pathname)

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

Create and open statements bind the logical pob-name to a physical pathname. In this way the system keeps track of the environment files for each 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 declarations may be obtained by the preprocessor from the related 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. 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 if Reference Count field is zero. Otherwise the object is delete marked to denote that an unpersist request has been issued on it. If the reference count drops to zero, the object is deleted if it is delete marked.

7. new_instance <pob-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 if they are the elements of an implicit list(B^+ ISAM).

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. The seq-no provides an index to this list through persistent variable directory. If seq-no is supplied as zero in the statement, the number of objects with name variable-name is returned in the variable seq-no.

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

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

Translations Performed by the Preprocessor

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

Semantically, if a transient value is assigned a persistent object or value, it is lost after program termination. For the case a persistent value or object is assigned a transient value, however, the value is made persistent.

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 all the class declarations. Class parser in turn creates the copies of the class declarations whose names are in SPOT and parses them. The preprocessor modifies the types of the persistent objects and their fields.

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.

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 may not be consecutive. But the system provides mechanisms to access them as if they were consecutive through overloading.

Overloader

Overloader is a header module which is included in the preprocessed C++ source with the include compiler directive. In this module, methods and operators of the fundamental persistent types are overloaded. In this way it becomes C++ compiler's duty to call the relevant methods and operators, when evaluating expressions. If an operator has one or more persistent operands then the overloaded version of this operator first allocates the object in the memory and then evaluates its return value.

In C++ constructors are used for initializing data members of objects. However, in dealing with persistent objects constructors must also allocate space for the objects on disk. Since the way C++ synchronizes the constructors of complex objects does not meet this requirement, the constructor calling sequence of C++ is overridden. The constructor of the class

cstaro___ (the class cstaro___ is responsible for providing persistence to objects) initializes the size of the object, and records it in the object table and 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. In other words, when a persistent object is created, its logical object identifier and the corresponding disk address is determined before the fields of this object get their disk addresses.

If the size of an object stays constant, its space is allocated in the related object base at compile time and such 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 compulsory 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.

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.

During the parsing process, the detailed structures of types and inheritance topology is stored in a linked list data structure. 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.

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. The programs written in C** are obviously shorter than those written in C++ using explicit file commands for programs requiring heavy file I/O, since we need at least the routines immitating the disk management offered by C**. Furthermore virtual memory management prevents the problem of main memory shortages. C** can be obtained free of charge from the authors (by electronic mail asuman@vm.cc.metu.edu.tr). A detailed description of the system is given in [EDG 94].

The experience we gained from C** implementation indicates 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 improve 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.

REFERENCES

[AB 87] Atkinson, M. P., Buneman, O.P., "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, Vol. 19, No. 2, June 1987.

[AG 89] Agrawal, R. and Gehani, N., "Design of the Persistence and Query Processing Facilities in O++: The Rationale", *Data Engineering* Vol. 12, No. 3, Sept. 1989.

[Atw 90] Atwood, T., " Two Approaches to Adding Persistence to C++", *in Implementing Persistent Object Bases*, 1990.

[CDKK 85] Chou, H-T., DeWitt, D. J., Katz, H.R., and Klug, A.C., "Design and Implementation of the Wisconsin Storage System", *Software Practice and Experience*, Vol.15, pp. 943-962, 1985.

[CDRS 89] Carey, M., Dewitt, J. D., Richardson E. J., Shekita J. E., "Storage Management for Objects in EXODUS", *in Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[D 91] Deux, O. et al."The O_2 System", *Comm. of the ACM*, Vol.34, No. 10, Oct. 1991.

[EDG 94] Evrendilek, C., Dogac, A., and Gesli, T., "A Preprocessor Approach to Persistent C++", TUBITAK Software R&D Center, Tech. Rep. 94-2, February 1994.

[LLOW 91] Lamb, C., Landis, G., Orenstein, J. and Weinreb, D.," The ObjectStore Database System", *Comm. of the ACM*, Oct. 1991.

[RC 89] Richardson, J. and Carey, M., "Persistence in the E language: Issues *and Implementation", Software-Practice & Experience*, Vol. 19, Dec. 1989.

[Say 92] Saygin, Y., " MOODS Storage System", *Ms Thesis*, Dept. of Computer Eng., Middle East Technical University, August 1992.

[SCD 90] Schuh, D., Carey, M. and Dewitt D., "Persistence in E Revisited", *in Implementing Persistent Object Bases*, 1990.