ABSTRACT
Traditional programming languages such as C++ only provides operations for manipulating and storing transient data. These are data dependent on the activation of the program. Explicit file manipulation is needed in order to support persistent data.

PEPC++, persistent programming in C++, deals with extending the C++ programming language to support persistent objects. The paper presents two new constructs, persist and get, and the necessary operations provided for the manipulation of persistent objects. The persist keyword specifies that an object is persistent; while the get construct allows a PEPC++ program to use previously defined persistent objects in other PEPC++ programs.

Keywords
Programming languages, persistence, objects

1. INTRODUCTION
The long-term storage of data has been of concern to programming language designers for some time [1]. Even though most traditional programming languages provide facilities for the manipulation of data, they do not deal with objects whose lifetime extends beyond the activation of the program. Persistent objects are one facet of the many faceted object-oriented programming paradigms, but one that is not and has never been supported by the C++ programming language [4]. The design of PEPC++ concentrated on providing persistence to an object-oriented programming language such as C++.

According to [3], persistence of data is the length of time for which the data exists and is usable. The spectrum of persistence exists and is categorized by

- transient results in expression evaluation
- local variables in procedure activation
- own variables, global variables and heap items whose extents are different from their scope
- data that exist between executions of a program
- data that exist between various versions of a program
- data that outlive the program

Traditional programming languages usually address only the first three kinds of object persistence; persistence of the last three kinds is typically the domain of database technology [3]. PEPC++ focused on making persistent objects in C++.

2. EXTENDING C++ WITH PERSISTENCE
PEPC++ is a language extension of C++ to provide object persistence. Using the language constructs of PEPC++ allows the programmer to specify which objects are persistent. The state of a persistent object is automatically stored. In the next activation of the defining application, the previously saved state will be made available automatically. In short, the lifetime of a persistent object is not dependent on the lifetime of the program that created it. Furthermore, PEPC++ facilitates sharing of the persistent objects among different PEPC++ applications.

The use of files in C++ is one way of storing transient data. However, the programmer has to explicitly specify using the file handling routine that the variable or set of variables have to be saved and retrieved explicitly. The intricacies of saving and loading are to be handled by the programmer. The use of PEPC++ eliminates the need for the programmer to explicitly store the state of persistent objects into files without sacrificing the semantics of the original language. In PEPC++, the details of reading and writing the state of persistent objects will be handled implicitly. This is made possible by using new constructs to specify persistence of objects.

2.1 Persistent Object Definition
The "persist" construct is used in the declaration of a persistent object. A persistent object is an object whose lifetime exceeds the activation of the application that created it. Listing 1 presents the syntax for the persistent construct.

```
persist <className> <objectName>;
```

Listing 1. Syntax for persistent object declaration
where:

- `<className>` the class definition
- `<objectName>` name of the object to persist

### 2.2 Accessing Persistent Objects

The scope of the persistent object determines its visibility and usability. Only objects that are visible to the current referencing environment can be accessed and manipulated. Therefore, persistent objects used in an application, which are not locally declared, should be made visible to the currently using application.

Note that only globally declared persistent objects are accessible to other subprograms. Locally declared persistent objects can only be accessed by the defining subprogram. This is due to the scoping rules followed by the C++ programming language.

Current PEPC++ programs can access previously declared persistent objects in other PEPC++ applications through the `get` construct. The syntax for the `get` construct is shown in Listing 2.

```cpp
get <className> <objectName>;
```

**Listing 2. Accessing persistent objects declared in other PEPC++ applications**

where:

- `<className>` class definition of the object
- `<objectName>` name of the previously declared persistent object

Listing 3 illustrates a sample PEPC++ code.

```cpp
persist pglobS1;
void getData()
{
    persist Stack S1;
    int j;
    S1.push(30);
    {
        persist Stack S2;
        int i;
        S2.push(10);
        i = S2.pop();
    };
    pglobS1.push(20);
j = S1.pop();
}
void main()
{
    pglobS1.push(5);
}
```

**Listing 3. PEPC++ sample program**

In the above illustration, two persistent objects were defined: `pglobS1`, which is a global persistent object, and `S2`, a local persistent object. It was assumed that `Stack` is a user-defined class.

### 3. IMPLEMENTING PEPC++

The PEPC++ application is a C++ program with persistent constructs. Two constructs were added to the C++ program to provide persistence to objects. Figure 1 illustrates how the PEPC++ code is manipulated.

![Figure 1. PEPC++ Code Manipulation](image)

The PEPC++ program passes through the PEPC++ preprocessor, before the C++ compiler. The preprocessor is composed of three phases, namely: the scanner, parser, and the code generator. The scanner examines and removes all persistent constructs from the PEPC++ code. The parser checks for the occurrence of persistent constructs. The translation rules specify how persistent constructs will be handled. The code generator handles the actual translation from PEPC++ persistent objects to its corresponding C++ object.
3.1 PEPC++ Translation Rules

Translation rules were designed to map the PEPC++ code to the corresponding C++ code. The PEPC++ code passes through the preprocessor, which generates the actual C++ code using the translation rules defined. The C++ compiler then compiles the translated code. The following are the defined PEPC++ translation rules:

3.1.1 The persist Construct

The persist construct is used to declare persistent objects. The approach used to implement persistent objects involves generating two methods for loading and saving the state of persistent objects. These two methods, namely: PEPC_load() and PEPC_save() are generated and included into the class definition of the persistent object. This approach makes it possible to load and save the state of various objects using the declarations within the class definition. Once these methods are generated, they are inserted within the code to facilitate the binding between the transient object and its persistent copy. The rule for this is specified as follows:

3.1.1.1 Rule Persist

To facilitate the storage and retrieval of persistent objects, the following rules applied:

1. The persist keyword is removed from the declaration, making it a conventional object;
2. At the start of the block containing the persistent declaration (that is, the persistent object becomes in scope), the call to the PEPC_load() method is inserted. For global persistent object declaration, the insertion is done after the main() program header.
3. Before the block containing the persistent declaration ends (that is, before the persistent object becomes out of scope), the call to the PEPC_save() method is inserted. For globally declared persistent objects, the insertion is done before the application terminates.

3.1.1.2 Rule InMethod

The generated definition and implementation of the PEPC_load() and PEPC_save() methods are inserted to class definitions with persistent instances. The generated PEPC_load() and PEPC_save() methods are unique for every class with persistent instances.

3.1.2 The get Construct

The get construct allows a PEPC++ application to access global persistent objects declared in other PEPC++ applications. Imported objects are treated as global objects in the importing applications.

3.1.2.1 Rule get

Globally declared persistent objects are available to other PEPC++ applications. The succeeding statements present the detailed procedure for the translation of the get construct.

1. Check for the persistence of the persistent object;
2. Include the class definition of the persistent object to the importing application;
3. Insert the global declaration of the persistent object in the importing application; and
4. Apply the rules for persist.

3.2 Persistent Object Manipulation

The status of the persistent object storage may change depending on the type of operation performed on the persistent object. To preserve the state of the persistent object, the state must be saved to a secondary storage. The next time the object is in scope, its state must be restored from the secondary storage. The actual call to the load and save method presents another issue: when does the underlying system perform the restore and update operation. It should be noted that the PEPC++ programmer could not explicitly specify when a persistent object is saved or restored. This means that the runtime system must decide when to perform such operations. More importantly, the performance of a PEPC++ program is dependent on the frequency of such operations. Two approaches were considered:

3.2.1 Lazy Approach

Loading is done when the object becomes accessible. Saving is performed just before the object becomes out of scope. Thus, loading and saving is performed only once during the scope of the object. Listing 4 exhibits the lazy approach of loading and saving of persistent objects. Locally declared persistent objects are loaded at the start of the subprogram where they are defined. Before the subprogram terminates, the state of the persistent object is stored. For imported and globally declared persistent objects, the previously saved state is restored once the main program is activated, and stored, before the current application terminates.
### 3.2.2 Eager Approach

Saving is done after every operation, and loading is performed before an expression involving the persistent object is evaluated. Listing 5 displays how the loading and saving of persistent object is performed when the eager approach is followed.

```cpp
persist pglobS1;
void getData()
{
    persist Stack S1;
    int j;
    S1.PEPC_load();
    S1.push(30);
    {
        persist Stack S2;
        int i;
        S2.PEPC_load();
        S2.push(10);
        i = S2.pop();
        S2.PEPC_save();
    }
    pglobS1.push(20);
    j = S1.pop();
    S1.PEPC_save();
}
void main()
{
    pglobS1.PEPC_load();
    pglobS1.push(5);
    pglobS1.PEPC_save();
}
```

Listing 5. Translated PEPC++ program using the eager approach

PEPC++ followed the lazy approach in updating persistent object storage. This would allow load once and save once semantics. For every persistent object declaration, a corresponding load statement is issued. The load statement restores the previously stored state of the persistent object from the persistent object storage. The state of the persistent object is stored before it becomes out of scope. Therefore, for locally declared persistent objects, update is done before the control returns to the caller. For globally declared persistent objects, the update is done before program termination.

### 3.3 The Persistent Object Storage

To allow data to become persistent, it has to be stored in a secondary storage. The persistent object storage is designed to contain all persistent objects. The updated state of all persistent objects for each application will be stored. The persistent object storage is represented by relational databases.

### 4. DISCUSSION

We presented an extension to C++ to support persistent objects. The preprocessor and supporting libraries have been implemented in Visual Basic. The state of persistent objects is stored in a database. Our experience shows that PEPC++ programs are
shorter and more readable than equivalent C++ programs that use files for persistent storage. While the Microsoft Foundation Classes provide serialization facilities, the ANSI C++ language does not provide this explicitly. Our study shows that persistency is a viable approach to improve programmability and reduce complexity of programs that require persistent storage.

REFERENCES


