Persistent Object Storage for C++

Konstantin Knizhnik

March 16, 1999

Contents

1 Introduction

POST++ provides simple and effective storage for application objects. POST++ is based on memory mapping mechanism and shadow pages transactions. POST++ eliminates any overhead on persistent objects access. Moreover POST++ supports work with several storages, storing objects with virtual functions, atomic data file update, provides high performance memory allocator and optional garbage collector for implicit deallocation of memory. POST++ correctly works with C++ classes using multiple inheritance and pointers inside objects.

2 Describing object class

POST++ storage manager needs information about persistent object classes to support garbage collection, relocation of references while loading, and initialization of pointers to virtual tables. Unfortunately C++ language provides no facilities to extract information about class format at runtime. As far as I want to avoid use of some special tools (preprocessors) or some "dirty trick" solutions (extracting information about classes from debugging information), this information should be provided to storage manager by programmer. Such class registration can be done very easy using special macros provided by POST++.

POST++ uses default constructors for initializing object while loading from storage. Programmer should include macro `CLASSINFO(NAME, FIELD_LIST)` in definition of any class, which instances can be saved in the storage. `NAME` corresponds to the name of this class. `FIELD_LIST` describes reference fields of this class. There are three macros defined in file `classinfo.h` for describing references:

`REF(x)`
: Describes single reference field.

`REFS(x)`
: Describes one-dimensional fixed array of references. (i.e. array with constant boundaries).

`VREFS(x)`
: Describes varying one-dimensional array of references. Varying array can be only the last component in class. When you are writing class declaration, you specify array with only one element. The actual number of elements in concrete object instance is specified at object creation time.

List of these macros should be separates by spaces: `REF(a) REF(b) REFS(c)`. Macro `CLASSINFO` defines default constructor (constructor without parameters) and declares class descriptor of this class. Class descriptor is static component of the class with name `self_class`. So class descriptor of the class foo can be accessed by `foo::self_class`. As far as constructors without arguments are called for base classes and components automatically by compiler, you should not worry about calling them explicitly. But do not forget to include `CLASSINFO` macro in definition of any structure, which can be used as component of serialized class. Then you should register your class to be accessible by storage manager. It can be done by macro `REGISTER(NAME)`. Class names are placed in the storage together with objects. Mapping between application and storage classes is established during storage opening. Names of all classes stored in the storage are compared with names of application classes. If some class name is not found within application classes or correspondent application and storage classes have different size, then program assertion will fail.

These rules are illustrated by the following example:

```
struct branch {
    object* obj;
    int key;

    CLASSINFO(branch, REF(obj));
};

class foo : public object {
  protected:
    foo*    next;
    foo*    prev;
    object* arr[10];
    branch  branches[8];
    int     x;
    int     y;
    object* childs[1];
  public:
    CLASSINFO(foo, REF(next) REF(prev) REFS(arr) VREFS(linked));
    foo(int x, int y);
};


REGISTER(1, foo);

main() {
```

```
    storage my_storage("foo.odb");
    if (my_storage.open()) {
        my_root_class* root = (my_root_class*)my_storage.get_root_object();
        if (root == NULL) {
            root = new_in(my_storage, my_root)("some parameters for root");
        }
        ...
        int n_childs = ...;
        size_t varying_size = (n_childs-1)*sizeof(object*);
        // We should subtract 1 from n_childs, because one element is already
        // present in fixed part of class.
        foo* fp = new (foo:self_class, my_storage, varying_size) foo(x, y);
        ...
        my_storage.close();
    }
}
```

3 Allocating and deallocating objects in storage

POST++ provides special memory allocator for managing storage memory. This allocator uses two different approaches: for allocating small and large objects. All storage memory is divided into pages (which size is independent from operating system page size and in current implementation of POST++ is 512 bytes). Small objects are those objects, which size is less or equal to 256 bytes (page size/2). These objects are allocated using fixed block chains. Each chain contains the list of blocks with the same size. Sizes of allocated objects are aligned at 8-byte boundary. The optimal number of fixed block chains for objects with size not greater than 256 is 14 (number of different equipartitions of the page). Before each object POST++ allocates object header, which contains class identifier of the object and object size. As far as size of header is exactly 8 bytes and in C++ size of object is always greater than 0, block chain with size 8 can be eliminated. Allocation and deallocation of small object usually is very fast: it requires only one remove/insert operation from L1 list. If the chain is empty and we are attempting to allocate new object, then new page is allocated and used for storing objects of this size (page is divided into the blocks, which are appended to the chain). Space for large object (with size greater than 256 bytes) is allocated from free page list. Size of large objects is aligned on page boundary. POST++ uses first feed, random position algorithm for maintaining list of free pages (all free segments of pages are sorted by their address and special pointer is used to follow current position in this list). Implementation of memory manager can be found in file `storage.cxx`

It is up to the programmer whether to use explicit or implicit memory deallocation. Explicit memory deallocation is faster (especially for small objects) but implicit deallocation (garbage collection) is more reliable. In POST++ mark and sweep garbage collection scheme is used. There is special object in the storage: root object. Garbage collector first marks all objects accessible from the root object (i.e. it is possible to reach the object starting from the root object, and navigating through references). Then all objects that are not marked during first stage of GC will be deallocated. Garbage collection can be made during loading objects from the file (if you pass `do_garbage_collection` attribute to `storage::open()` method). It is also possible to explicitly invoke garbage collection during program execution by calling `storage::do_mark_and_sweep()` method. But be sure that there are no program variable pointed to objects inaccessible from the root objects (these objects will be deallocated by GC).

Because of multiple inheritance C++ classes can have non zero offset within object and references inside object are possible. That is why we have to use special technic to access object header. POST++ maintains page allocation bitmap each bit of which corresponds to the page in the storage. If some large object is allocated at several pages, then bits corresponding to all pages occupied by this object except first one will be set to 1. All other pages have correspondent bits in bitmap cleared. To find start address of the object, we first align pointer value on the page size. Then POST++ finds page in bitmap that contains beginning of the object (this page should have zero bit in bitmap). Then we extract information about the object size from object header placed at the beginning of this page. If size is greater than half of page size then we have already found object descriptor: it is at the beginning of the page. Otherwise we calculate fixed block size used for this page and round down offset of pointer within this page to block size. This scheme of header location is used by garbage collector, operator delete defined in object class and by methods extracting information from the object header about object size and class.

In POST++ special overloaded new method is provided for allocation of objects in the storage. This method takes as extra parameters class descriptor of created object, storage in which object should be created and, optionally, size of varying part of the object instance. Macro `new_in(STORAGE, CLASS)` provides "syntatic sugar" for persistent object creation. Persistent object can be deleted by redefined operator `delete`.

4 Persistent object protocol

All classes of persistent objects in POST++ should be derived from object class defined in `object.h`. This class contains no variables and provides methods for object allocation/deallocation and obtaining information about object class and size at runtime. It is possible to use object class as one of multiple bases of inheritance (order of bases is not significant). Each persistent class should have constructor which is used by POST++ system (see section Describing object class). That means that you should not use constructor without parameters for normal object initialization. If your class constructor even has no meaningful parameters, you should add dummy one to distinguish your constructor with constructor created by macro `CLASSINFO`.

To access objects in persistent storage programmer needs some kind of root object from which each other object in storage can be accessed by normal C pointers. POST++ storage provides two methods allowing you to specify and obtain reference to the root object:

```
void    set_root_object(object* obj);
object* get_root_object();
```

When you create new storage `get_root_object()` returns `NULL`. You should create root object and store reference to it by `set_root_object()` method. Next time you are opening storage, root object can be retrieved by `get_root_object()`.

Hint: In practice application classes used to be changed during program development and support. Unfortunately POST++ due to its simplicity provides no facilities for automatic object conversion (see for example lazy object update scheme in GOODS), So to avoid problems with adding new fields to the objects, I can recommend you to reserve some free space in objects for future use. This is especially significant for root object, because it is first candidate for adding new components. You should also avoid reverse references to the root object. If no other object has reference to the root objects, then root object can be simply changed (by means of `set_root_object` method) to instance of new class. POST++ storage provides methods for setting and retrieving storage version identifier. This identifier can be used by application for updating objects in the storage depending on the storage and the application versions.

5 Storage constructor

file description	**when used**	**suffix**
temporary file with new storage image	used in non-transaction mode to store new image of storage	`.tmp`
transaction log file	used in transaction mode to saved shadow pages	`.log`
saved copy of storage file	used only in Windows-95 for renaming temporary file	`.sav`

Two other parameters of storage constructor have default values. First of them `max_file_size` specifies limitation of storage file extension. If storage file is larger than `storage::max_file_size` then it will not be truncated but further extends are not possible. If `max_file_size` is greater than the file size, then behavior depends on storage opening mode. In transaction mode, file is mapped on memory with read-write protection. Windows-NT/95 extends in this case size of the file till `max_file_size`. The file size will be truncated by `storage::close()` method to the boundary of last object allocated in the storage. In Windows it is necessary to have at least `storage::max_file_size` free bytes on disk to successfully open storage in read-write mode even if you are not going to add new objects in the storage.

The last parameter of storage constructor is `max_locked_objects`, This parameter is used only in transaction mode to provide buffering of shadow pages writes to the transaction log file. To provide data consistency POST++ should guaranty that shadow page will be saved in the transaction log file before modified page will be flushed on disk. POST++ use one of two approaches: synchronous log writes (`max_locked_objects == 0`) and buffered writes with locking of pages in memory. By locking page in the memory, we can guaranty that it will not be swapped out on disk before transaction log buffers. Shadow pages are written to the transaction log file in asynchronous mode (with operating system cashing enabled). When number of locked pages exceeds `max_locked_pages`, log file buffers are flushed on disk and all locked pages are unlocked. Such approach can significantly increase transaction performance (up to 5 times under NT). But unfortunately different operating systems use different approaches to locking pages in memory.

- Windows 95 doesn't support it at all.
- In Windows NT every process can lock its pages, but total number of locked pages should not exceed process working set quota. By default process can't lock more than 30 pages. If you specify `max_locked_pages` parameter greater than 30, than POST++ will try to extend process working set to feet your requirement. But my experiments show that difference in performance with 30 and 60 locked pages is very negligible.
- In Unix only superuser can lock pages in memory. That is why file constructor checks if process has enough permissions to use lock operations. So if you specify `max_locked_pages` parameter greater than 0, then decision whether to use synchronous or asynchronous writes to the transaction log file will be taken at moment of storage class creation. If you want to use benefits of memory locking mechanism (2-5 times, depending on type of transaction), you should change owner of your application to root and grant set-user-ID permission: `chmod +s application`.

6 Opening storage

POST++ uses memory mapping mechanism for accessing data from the file. Two different approaches are used in POST++ to provide storage data consistency. First and more advanced is based on transaction mechanism using shadow pages to provide storage recovery after fault and transaction rollback. Before write shadow page creation algorithm is used. This algorithm is implemented in the following way: all mapped on file pages are set to readonly protection. Any write access to such page will cause access violation exception. This exception is handled by special handler, which change page protection to read-write and place copy of this page in transaction log file (log file name is combined from the original data file name and suffix `.log`). All following write accesses to this page will not cause page faults. Storage method `commit()` flushes all modified pages on disk and truncates the log file. `storage::commit()` method is implicitly called by `storage::close()`. If fault happened before `storage::commit()` operation, all changes will be undone by coping modified pages from transaction log to the storage data file. Also all changes can be undone explicitly by `storage::rollback()` method. To choose transaction based model of data file access, specify `storage::use_transaction_log` attribute for `storage::open()` method.

Another approach to providing data consistency is based on copy on write mechanism. In this case original file is not affected. Any attempt to modify page that is mapped on the file, cause creation copy of the page, which is allocated from system swap and has read-write access. File is updated only by explicit call of `storage::flush()` method. This method writes data to temporary file (with suffix `.tmp`) and then renames this file to original one. So this operation cause an atomic update of the file (certainly if operating system can guaranty atomicity of `rename()` operation).

Attention: If you are not using transactions, `storage::close()` method doesn't flush data in the file. So if you don't call `storage::flush()` method before `storage::close()` all modifications done since last flush will be lost.

Windows 95 specific: In Windows 95 rename to existing file is not possible, so original file is first saved in file with name with suffix `.sav`. Then temporary file wit suffix `.tmp` is renamed to the original name and finally old copy is removed. So if fault is happened during `flush()` operation and after it you find no storage file, please do not panic, just look for file with name terminated with `.sav` suffix and rename it to the original one.

Hint: I recommend you to use transactions if you are planning to save data during program execution. It is also possible with copy on write approach but it is much more expensive. Also transactions are always preferable if size of storage is large, because creating temporary copy of file will require a lot of disk space and time.

There are several attributes, which can be passed to storage `open()` method:

- `support_virtual_functions`
  This attribute should be set if objects with virtual functions are placed in the storage. If this attribute is not set, POST++ decides that all persistent objects contain references only within storage (to other objects in the storage). So adjustment of references should be done only if base address of data file mapping is changed (this address is stored in the first word of data file and POST++ always tries to map file to the same address to avoid unnecessary reference adjustment). But if object class contains virtual functions, pointer to virtual table is placed inside object. If you recompile your application, address of this table can be changed. POST++ library compares timestamp of executable image with timestamp stored in database created by this application. If these timestamp are not equal, correction of virtual table pointer should be performed. To get application timestamp POST++ should locate executable file image. Unfortunately there is no portable way to find out executable file name in Unix. Under Unix POST++ looks at the value of environment variable `_`, which is set by shell. This approach will not work if process was started not by shell (for example by `system()`) or working directory is changed by `chdir()`. The most portable way is to use file `comptime.cxx`, which should be compiled each time you recompile your application and linked together with storage library. There is no such problem in Windows, where name of executable image can be obtained by Win32 API. While storage opening POST++ compares this timestamp with timestamp stored in the data file and if they are different and `support_virtual_functions` attribute is specified then correction of all objects (by calling default constructor) will be done.

- `read_only`
  By setting this attribute programmer says that he wants only readonly access to the data file. POST++ will create readonly view of the data file and any attempt to change some object in the storage or allocate new one will cause protection violation fault. There is one exception: if it is impossible to map data file to the same address or application is changed and `support_virtual_functions` is specified, then protection of region is temporary changed to copy on write and conversion of loaded objects takes place.

- `use_transaction_log`
  Setting of this attribute force using transactions for all data file updates. Shadow page strategy is used for implementing transactions. Transaction is opened implicitly when storage first modification of storage is done. It is closed explicitly either by `storage::commit()` or by `storage::rollback()` operations. Method `storage::commit()` saves all modified pages on disk and truncates transaction log, method `storage::rollback()` undo all changes made within this transaction.

- `no_file_mapping`
  By default POST++ will map data file to the process virtual memory. Time of opening database is greatly reduced in this case, because pages of the file will be loaded on demand. But if size of database is not so large or all data from the database need to be accessed immediately, then reading file to memory can be preferable to using virtual memory mapping because no extra overhead of handling

page faults takes place in this case. Flag `no_file_mapping`prevents POST++ from mapping file and cause reading it in allocated memory segment.

- `fault_tolerant`
  This flag should be used by applications which want to preserve database consistency in case of system or application fault. It is not necessary to specify this flag if `use_transaction_log` flag is used, because consistency will be provided by transaction mechanism in this case. If the flag `use_transaction_log flag` is not specified and the flag `fault_tolerant` flag is set, POST++ will not change original file preserving its consistency. This is achieved either by reading file in memory (if the flag `no_file_mapping` is set) or using copy-on-write pages protection. In last case attempt of modification of mapped on file page will cause creation copy of the page in system swap file. Method `flush()` will save in-memory image of database to temporary file and then rename it to original file using atomic operation. If `fault_tolerant` flag is not specified, POST++ does in-place modification of database pages, providing maximal application performance (because there is no overhead caused by copying modified pages and saving database image in temporary file) As far as modified pages are not flushed to the disk immediately, some changes can be lost as a result of system fault (the worst thing is that some modified pages can be saved to the disk while other not - so database consistency can be violated).

- `do_garbage_collection`
  When this attribute is set POST++ will perform garbage collection in storage during opening. The operation of collecting garbage is combined with reference adjustment. Using garbage collection is always more safer than manual memory deallocation (due to the problem of hanging references), but explicit memory deallocation has less overhead. Garbage collection in POST++ has one more advantage in comparison with explicit deallocation: garbage collector performs utilization of pages used for small objects. If there are no more allocated small objects at the page then garbage collector will include this page in the list of free pages. This is not done for explicit deallocation because free cells for small objects are linked in chain and it is not so easier to remove them from this chain (in case of garbage collector all chains are reconstructed). Even if you are using explicit memory deallocation, I suggest you to do time by time garbage collection to check for reference consistency and absence of memory leaks (`garbage_collection` method returns number of deallocated objects and if you are sure that you have explicitly deallocate all unreachable objects, then this number should be zero). As far as garbage collector modifies all objects in the storage (set mark bit), relink free objects in chains), running GC in transaction mode can be time and disk space consuming operation (all pages from the file will be copied to the transaction log file).

You can specify maximal size for storage files by `file::max_file_size` variable. If the data file size is less than `file::max_file_size` and mode is not `read_only`, then additional `size_of_file - file::max_file_size` bytes of virtual space will be reserved after the file mapping region. When storage size is extended (because of new objects allocation), this pages will be committed (in Windows NT) and used. If size of file is greater than `file::max_file_size` or `read_only` mode is used, then size of mapped region is exactly the same as the file size. Storage extension is not possible in the last case. In Windows I use `GlobalMemoryStatus()` function to obtain information about actually available virtual memory in the system and reduce `file::max_file_size` to this value. Unfortunately I found no portable call in Unix which can be used for the same purpose (`getrlimit` doesn't return actual information about available virtual memory for users process).

The interface to object storage is specified in file `storage.h` and the implementation can be found in `storage.cxx`. Operating system dependent part of mapped on memory file is encapsulated within file class, which definition is in file.h and implementation in `file.cxx`.

7 Installation of POST++

Installation of POST++ is very simple. It is now checked for the following platforms: Digital Unix, Linux, Solaris, Windows NT 4.0, Windows 95. I expect no problems with most of all other new Unix dialects (AIX, HP-UX 10, SCO...). Unfortunately I have no access to these systems. At Windows I compiled POST++ by Microsoft Visual C++ 5.0 and Borland 5.02 compilers. Makefile for Visual C++ is `makefile`, and makefile for Borland C++ is `makefile`.

The only thing that you are needed to use POST++ is library (libstorage.a at Unix and storage.lib at Windows). This library can be produced by just issuing make command. There is special `MAKE.BAT` for Microsoft Visual C++ which invokes `NMAKE` with `makefile.mvc` as input (if your are using Borland either edit this file or invoke Borland make by `make.exe -f makefile.bcc` command).

Installation in Unix can be completed by copying POST++ library and header files to some standard system catalogs. You should set proper values for `INSTALL_LIB_PATH` and `INSTALL_INC_PATH` variables in makefile and execute `make install` command. Default value for `INSTALL_LIB_PATH` is `/usr/local/lib` and for `INSTALL_INC_PATH` is `/usr/local/include`. You can avoid copying of POST++ files in system catalogs by specifying path to POST++ catalog to compiler and linker explicitly.

8 POST++ class library

POST++ contains definition of some persistent classes, which can be used in your application and also are good examples of developing classes for POST++. You can see that there are almost no POST specific code in the implementation of these classes. These classes include array, matrix, string, L2-list, hash table, AVL-tree, R-tree, text object. R-tree provides fast access to spatial object (object with spatial coordinates). Text object contains modification of Boyer and Moore algorithm extended to search multiple patterns combined by OR/AND relation. Definition of these classes can be found in following files:

Description	**Interface**	**implementation**
Arrays of scalars and references, matrixes and strings	`array.h`	`array.cxx`
L2-list and AVL-tree	`avltree.h`	`avltree.cxx`
Hash table with collision chains	`hashtab.h`	`hashab.cxx`
R-tree with quadratic method of nodes splitting	`rtree.h`	`rtree.cxx`
T-tree (combination of AVL tree and array)	`ttree.h`	`ttree.cxx`
Text object with modified Boyer and Moore search algorithm	`textobj.h`	`textobj.cxx`

In the article "A study of index structures for main memory database management systems", T.J. Lehman and M.J Carey proposed T-trees as a storage efficient data structure for main memory databases. T-trees are based on AVL trees proposed by Adelson-Velsky and Landis. In this subsection, we provide an overview of T-trees as implemented in POST++.

Like AVL trees, the height of left and right subtrees of a T-tree may differ by at most one. Unlike AVL trees, each node in a T-tree stores multiple key values in a sorted order, rather than a single key value. The left-most and the right-most key value in a node define the range of key values contained in the node. Thus, the left subtree of a node contains only key values less than the left-most key value, while the right subtree contains key values greater than the right-most key value in the node. A key value which is falls between the smallest and largest key values in a node is said to be bounded by that node. Note that keys equal to the smallest or largest key in the node may or may not be considered to be bounded based on whether the index is unique and based on the search condition (e.g. "greater-than" versus "greater-than or equal-to").

A node with both a left and a right child is referred to as an internal node, a node with only one child is referred to as a semi-leaf, and a node with no children is referred to as a leaf. In order to keep occupancy high, every internal node has a minimum number of key values that it must contain (typically $k-2$, if k is the maximum number of keys that can be stored in a node). However, there is no occupancy condition on the leaves or semi-leaves.

Searching for a key value in a T-tree is relatively straightforward. For every node, a check is made to see if the key value is bounded by the left-most and the right-most key value in the node; if this is the case, then the key value is returned if it is contained in the node (else, the key value is not contained in the tree). Otherwise, if the key value is less than the left-most key value, then the left child node is searched; else the right child node is searched. The process is repeated until either the key is found or the node to be searched is null.

Insertions and deletions into the T-tree are a bit more complicated. For insertions, first a variant of the search described above is used to find the node that bounds the key value to be inserted. If such a node exists, then if there is room in the node, the key value is inserted into the node. If there is no room in the node, then the key value is inserted into the node and the left-most key value in the node is inserted into the left subtree of the node (if the left subtree is empty, then a new node is allocated and the left-most key value is inserted into it). If no bounding node is found then let N be the last node encountered by the failed search and proceed as follows: If N has room, the key value is inserted into N; else, it is inserted into a new node that is either the right or left child of N, depending on the key value and the left-most and right-most key values in N.

Deletion of a key value begins by determining the node containing the key value, and the key value is deleted from the node. If deleting the key value results in an empty leaf node, then the node is deleted. If the deletion results in an internal node or semi-leaf containing fewer than the minimum number of key values, then the deficit is made up by moving the largest key in the left subtree into the node, or by merging the node with its right child.

In both insert and delete, allocation/deallocation of a node may cause the tree to become unbalanced and rotations (RR, RL, LL, LR) may need to be performed. (The heights of subtrees in the following description include the effects of the insert or delete.) In the case of an insert, nodes along the path from the newly allocated node to the root are examined until either

1. a node for which the two subtrees have equal heights is found (in this case no rotation needs to be performed), or
2. a node for which the difference in heights between the left and the right subtrees is more than one is found and a single rotation involving the node is performed.

In the case of delete, nodes along the path from the de-allocated node's parent to the root are examined until a node is found whose subtrees' heights now differ by one. Furthermore, every time a node whose subtrees' heights differ by more than one is encountered, a rotation is performed. Note that de-allocation of a node may result in multiple rotations.

There are several test programs for testing classes from POST++ persistent class library, which are

Program	**Tested classestes**
`ttree.cxx`	AVL-tree, l2-node, hash table
`testtext.cxx`	text, string
`testspat.cxx`	rectangle, R-tree, hash table
`testperf.cxx`	T-tree insert, find and remove operations

9 How to use POST++

There are some examples of classes and application for POST++. The simplest of them is game "Guess an animal". The algorithm of this game is very simple and the result looks rather impressive (something like artificial intelligence). Moreover this game is very good example, illustrating benefits of persistent object storage. Sources of this game are in file `guess.cxx`. Building of this game is included in default make target. To run it just execute `guess`.

Unix specific: When you will link your Unix application with POST++ library and persisent objects in application contain virtual functions, please do not forget to recompile comptime.cxx file and include it in the linker's list. This file is necessary for POST++ to provide executable file timestamp, which is placed in the storage and used to determine when application is changed and reinitialization of virtual function table pointers in objects is needed. *Attention!* This file should be recompiled each time your are relinking your application. I suggest you to make compiler to call linker for you and include `comptime.cxx` source file in the list of object files for the target of building executable image (see the makefile).

10 Specific of debugging POST++ applications

Information in this section is meaningful only for application using transactions. POST++ uses page protection mechanism to provide creation of shadow page on the original page modification, After storage opening or transaction commit all mapped on file pages are read-only protected. So any attempt to modify contents of the object allocated at this page will cause access violation exception. This exception is handled by special POST++ handler. But if you are using debugger, it will catch this exception first and stop application. If you want to debug your application you should do some preparations:

- In Unix it is enough to tell debugger not to catch `SIGSEGV` signal. For example for GDB it can be done with command: `handle SIGSEGV nostop noprint pass`. If `SIGSEGV` signal is not caused by storage page protection violation, but due to a bug in your program, POST++ exception handler will "understand" that it is not his exception and send SIGABRT signal to the self process, which can be normally catched by debugger.

- In Windows POST++ uses unhandled exception filter to handle storage page protection violations. Unfortunately it is impossible to make Microsoft Debugger to ignore unhandled exceptions. If you are going to debug your application, you should enclose all your program code (`main` or `WinMain` function) with structured exception block. You should always use structured exception handling with Borland C++, because `Unhandled Exception Filter` is not correctly called in Borland. Please use two macros `SEN_TRY` and `SEN_ACCESS_VIOLATION_HANDLER()` to enclose body of `main` (or `WinMain`) function:

```
main() {
    SEN_TRY {
        ...
    } SEN_ACCESS_VIOLATION_HANDLER();
    return 0;
}
```

  Be sure that Debugger behavior for this exception is `Stop if not handled` and not `Stop always` (you can check it in `Debug/Exceptions` menu). In file `testrans.cxx` you can find example of using structured exception handling.

11 Some more information about POST++

POST++ is freeware. It is distributed in hope to be useful. You can do with it anything you want (with no limitation on distributing products using POST++). I will be glad to help you in using POST++ and receive all kind of information (bug reports, suggestions...) about POST++. Freeware status of POST++ doesn't mean lack of support. I promice you to do my best to fix all reported bugs. Also e-mail support is guaranteed. POST++ can be used for various purposes: storing information between session, storing object system in file, snapshots, informational systems...

But if you fill that you need more serious object oriented database for your application supporting concurrency, distribution and transactions, please visit GOODS (Generic Object Oriented Database System) home page, at `http://www.ispras.ru/~knizhnik/goods.html`

Look for the newest version of my homepage	E-mail me about bugs and problems
`http://www.ispras.ru/~knizhnik`	`knizhnik@altavista.net`