Persistent Pointer Factory

User's Guide

www.codefarms.com

Code Farms Inc.
7214 Jock Trail
Richmond, Ontario, Canada
K0A 2Z0

Telephone: (613) 838-4829
Email: info@codefarms.com

August, 1998

INTRODUCTION

PPF helps you to build custom databases, which are fast and can handle large volume of data efficiently even on an ordinary PC. PPF is small (30kB object file), and the resulting databases are often by an order of magnitude faster than ready-made commercial databases.

PPF gives you three smart pointer classes:

PersistString - equivalent of char* for variable length text strings;

PersistPtr<T> - equivalent of T* for lightweight class T which does not use inheritance;

PersistVptr<T> - equivalent of T* for class T which uses inheritance and virtual functions.

If you replace all pointers in your program by these smart pointers, all objects in your program become automatically persistent (stored on disk).

Note the conceptual difference of this approach from the persistent objects based on serialization. Such objects reside in memory, and you store this memory image - usually at the end of the run. The objects must be loaded back to memory when you want to use them again. This is a slow process, and does not permit you to handle large data sets efficiently.

PPF takes a different approach. It keeps the data on disk, and pages it to memory only when needed. You can retrieve individual objects or small subsets, have fast access to the data, and since you have full control over the paging, you can tune up the performance of your database. Objects of some classes may always be in memory, while other class may never have more than one object in memory. The size of your disk space is the only limit on how much data you can store. PPF also provides a transparent management and reuse of the data space, including free lists.

Other advantages of PPF:

- No need to code serialization functions, which is a tedious and error-prone task. A
- Your data set may far exceed both the RAM and the virtual memory space.
- Free lists by-pass expensive allocation and initialization of C++ objects.
- When you stop your program, most of your data is already on disk; only a few pages have to move from memory to disk, which is very fast. This also helps in crash recovery.
- The smart pointers are space efficient. The storage on disk typically takes less space than your objects in memory. The size of PersistPtr is the same as sizeof(char*), PersistVptr takes twice as much.

FIRST EXAMPLE

The use of PPF is so simple that, except for this short instruction, no documentation is required. Just to give you the first feeling how PPF works, consider the following example:

```
class B {
    int b;
    ...
};
class A {
    int a;
    A *aPtr;
    B *bPtr;
    ...
};
```

If you tag your classes with the PersistClass(..) statement, and replace all pointers by the smart pointer PersistPtr<...>, your classes will become automatically persistent:

```
class B {
    PersistClass(B);
```

```
    int b;
    ...
};
class A {
    PersistClass(A);
    int i;
    PersistPtr<A> aPtr;
    PersistPtr<B> bPtr;
    ...
};
```

There are several minor things we still have to add - they will be explained shortly.

LIMITATION

When calling a function, you can pass PersistPtr<> either as a copy, or as a reference:

```
void foo1(PersistPtr<A> a){ // copy of 'a' is used inside foo1()
    if(a->i)...
}

void foo2(PersistPtr<A>& a){ // address of 'a' is used inside foo2()
    a->i = 0;
}
```

Function calls are restricted by the following rules:

(1) When passing a persistent object to a function, pass the PersistPtr<>, not the object.

(2) The function parameters must not be expressions using operators -> . & or * on persistent objects.

Examples:

```
class A {
    PersistClass(A);
    PersistPtr<A> next;
    int id;
    ...
};

void foo3(A a){ // NOT allowed, violation of rule (1)
    a.i = 0;
    ...
}

void foo4(int *i){ // OK, appears not to deal with persistent objects
    *i = 0;
    ...
}

int k;
PersistPtr<A> ap,tmpPtr;

ap = new A;
foo1(ap); // OK
foo2(ap); // OK
foo2(ap->next); // violation of rule (2), may crash
```

```
foo4(&(ap->id)); // violation of rule (2), may update id in a wrong object
```

Correct uses:

```
tmpPtr = ap->next;
foo2(tmpPtr);
ap->next = tmpPtr;

k = a->i;
foo3(&k);
a->i = k;
```

LIGHTWEIGHT CLASSES AND HEAVY-WEIGHT CLASSES

The previous example assumed that classes A and B are "light-weight", which means that they do not use inheritance. Smart pointers which point to normal (or "heavy-weight") objects must be smarter. They keep 8 bytes of information compared to 4 bytes required by smart pointers for light-weight objects. In many applications, most of the classes fall into the light-weight category, and using the 8-byte pointers is a waste of space. For this reason, PPF provides two sets of classes and pointers:

PersistClass(T), PersistPtr<T> ... for light-weight class T;

PersistVclass(T), PersistVptr<T> ... for heavy-weight class T.

If you don't want to bother with the two different kinds of classes, use the heavy-weight option all the time.

Let's expand the example above by deriving class B from class C. Class A is still leight-weight:

```
class C {
    PersistVclass(C);
    int c;
    ...
};
class B : public C {
    PersistVclass(B);
    int b;
    ...
};
class A {
    PersistClass(A);
    int i;
    PersistPtr<A> aPtr;
    PersistVptr<C> bPtr;
    ...
};
```

FILE ORGANIZATION

The entire PPF is very small (object file about 30kB), and comes in only 3 files:

(1) factory.h - provides templates for the PersistPtr<>, PersistVptr<>, and definitions of PersistString and PersistFactory.

(2) pointer.cpp (pointer.cc) - implementation of PersistPtr<> and PersistVptr<>

(3) factory.obj (factory.o) - all the remaining functions.

If you keep keep the usual file organization (files x.h and x.cpp for class X), you have to use the following includes:

```
FILE x.h:
    #include "factory.h"
    class X {
        PersistClass(X); // of PersistVclass(X)
        ...
    };
    #include "pointer.cpp" // always include when smart pointers are used

FILE x.cpp:
    #include "x.h"
    PersistImplement(X);  // same for leight-weight and heavy-weight
    ... implementation of X methods....
```

PersistImplement(X) hides the implementation of the methods that have been added to class X under PersistClass(X), and which are necessary for the management of the persistency. This is a simple macro - only 2 lines of code, and comes file factory.h. When linking your program, you have to link to factory.obj.

PLATFORM AND COMPILER

If you received a binary version of PPF, it is already customized to the platform and compiler you ordered.

If you received the full source of PPF (multi-platform), the source is set for DOS or Windows (Win95, Win98, WinNT) and the Microsoft VC++ compiler. If you run in a different environment, edit file factory.h, search for the text string "DOS", and comment or un-comment #define statements depending on your environment.

PERSISTENT POINTERS

PPF pointers have most of the functionality you would expect from a normal pointer. The following list of functions applies to both PersistPtr<T> and to PersistVptr<T>:

```
    PersistPtr();           // constructor which initializes the pointer
                            // to what is NULL for the regular pointer
    PersistPtr(T* realPtr); // initialized by a normal pointer
    PersistPtr(const PersistPtr& rhs); // initialized by a smart pointer
    ~PersistPtr();          // delete the pointer without deleting the object
                            // to which it points
    PersistPtr& operator=(const PersistPtr& rhs); // normal assignment
    PersistPtr& operator=(long rhs); // interpreting properly a long index
    int operator==(const PersistPtr& rhs); // pointers are equal if the point
                             // to the same object
    int operator!=(const PersistPtr& rhs); // same for not-equal
    T* operator->() const;   // operator -> works as usual
    T& operator*() const;    // operator * returns the object to which the
                             // pointer points
    T& operator[](long incr);// indexing into an array works as usual
    long getIndex();         // the index is really the disk address
    void setNull();          // sets the pointer to the equivalent of NULL
    PersistPtr<T>& newArr(long size);    // equivalent of    T::new T[]()
    void delObj();                       // equivalent of T::delete()
    void delArr();                       // equivalent of T::delete[]()
    void getRoot();          // gets the root object of this class
    void setRoot();          // sets the object to be the root of this class.
```

For each class, you can assign one object as its 'root', and retrieve this root on the subsequent runs. This provides a convenient hook or hooks for your data structures:

```
    px.setRoot();   // assigns px to be the root of its class
    px.getRoot();   // sets px to the root of its class
```

THE STRING CLASS

Class PersistString is not a template, and it manages variable-length strings on disk. All strings are stored on the same disk (file string.ppf).

```
class PersistString {
public:
    // allocation
    PersistString(); // equivalent of NULL pointer
    PersistString(long sz); // assigns string of length sz
    PersistString(char *s); // creates space and copies NULL ending s into it
    ~PersistString(); // destroys the pointer, not the string
    void delString(); // deletes the string, sets the pointer to NULL

    // pager is controlled differently for each class
    static void startPager(long pgSz,long maxPgs,long totMem);
    static void closePager();

    // operators
    PersistString& operator=(const PersistString& rhs);
    PersistString& operator=(long rhs);
    PersistString& operator=(PersistString* realPtr);
    int operator==(const PersistString& rhs);
    int operator!=(const PersistString& rhs);
    char* operator->();
    char operator[](long incr);
    void setNull();
    char *getPtr(long ind); // moves string from a given address to memory
    char *getPtr();       // moves this string to memory and returns pointer
    long getInd(char *s); // special use: get disk address for a memory pointer

    // manipulating strings (this group will be significantly expanded)
    int cmp(const PersistString& s); // compare two strings
    size_t size();                   // get the size of the string
    long allocSize();  // get the allocated size, which may be bigger
```

METHODS ADDED TO YOUR CLASS

Macros PersistClass(T) and PersistVclass(T) add the following methods to your classes:

```
    void* operator new(size_t size); // overloads operator new, so you can
                                     // allocate new objects as usual
    static void startPager(long pgSz,long maxPgs,long totMem);
          // starts the pager for class T with the following parameters:
          // pgSz=page size, maxPgs=max.number of pages in memory at any time,
          // totMem=estimate of the total space for the objects of this class
    static void closePager(); // close the pager and move all data to disk.
    T(); // default constructor - provided only for PersistVclass-es.
```

Note that totMem is an estimate of the total size which will be needed for the objects of this class. This is not a critical value, but if you provide a good guess, you will improve the performance by avoiding unnecessary reallocation of some internal arrays.

The page size must be at least sizeof(PersistHeader)=16 bytes. For text strings (see later), the page size must be at least the size of the

longest string you will ever use. This is not an estimate; if you exceed this limit, you will get an error message about an unseccessful attempt to access a string which is too long.

If you want all the objects of a certain class in memory, the X::startPager() call must use pageSize which includes the space of one or several objects to accomodate the Header. Two examples:

(A) 500 objects of class X, sizeof(X)==12 will require 2 objects to accomodate the 16 byte Header. In order to keep all X instances in memory during the program run, use X::startPager(n,1,n); where n>=502*12.

(B) 400 objects of class X, sizeof(x)==28 will require 1 object to accomodate the Header. In order to keep all X instances in memory during the program run, use X::startPager(n,1,n); where n>=401*28.

EXAMPLE OF THE MAIN PROGRAM

```
#include "factory.h"
#include "c.h"
#include "b.h"
#include "a.h"

int main() {
    ...
    A::startPager(1024,1000,100000000);
    B::startPager(40,2,1000);
    C::startPager(1000000,1,1000000);
    PersistStart;

    ... body of the main program ...

    A::closePager();
    B::closePager();
    C::closePager();
    return 0;
}
```

Note the selected configuration of disk paging:

Class A will have up to 1000 pages in memory, each 1024 bytes in size. The total space estimated for objects of class A is 100MB, but this is not a hard limit.

Class B has only small pages (40 bytes), and at most 2 pages in memory at any given time. The total space occupied by objects of class B is estimated at 1kB.

Class C has large pages (100kB), only 1 page in memory, and the total size of the data 100kB. This means that if the estimate of the total size was correct, all C objects will be in memory all the time, without any paging. If the estimate proves to be low, this class will start swapping memory to disk.

The important thing about controlling the paging is that, except for class PersistString, the paging parameters can be different for different program runs. For example, if you have a computer with a lot of memory, this computer can use many large pages. Another computer can later use the same data with a reduced memory requirement. Both computers, of course, will depend on the same disk storage of the data.

The management of the strings is, internally, more complex, because it manages strings (and their free lists) for a wide variety of sizes. For this reason, the paging parameters must be identical for subsequent program runs. If they are not the same, PPF will print an error message. All strings automatically allocated within the boundary of a single page. The selected page size must be at least the size of the largest string you plan to use. This is a hard limit: If you ask for a larger string, PPF will refuse to allocate it, and will issue an error.

You can close and restart the pager of any class several times (even using different paging parameters each time), but only one call to PersistStart is required after the initial opening of all pagers (calls to startPager) - see test1.cpp.

CREATING AND DESTROYING OBJECTS

If you want to make a class X persistent, register it by placing statement PersistClass(X); or PersistVclass(X); into its definition:

```
class X {
    PersistVclass(X);
    int i;
    ...
public:
    ...
};
```

Statement PersistVclass(X) is a macro which does not increase the size of the X objects; it only provides additional functions for this class. The most important functions are

```
new(size_t sz)  ... is replaced, but it is used as usual
newArr(long sz) ... creates a new array of this class,
delObj()        ... destroys the object and place it on the free list,
delArr()        ... destroys the array and place it on the free list.
```

As you see, you never really destroy objects, you only release them and place them on the free list. Operator delete() should NEVER be called.

Persistent objects must always be referenced through PersistVptr<X> or through PersistPtr<X>, never through normal pointer like X*. Such a pointer referes only to a temporary position in the memory pages, and is likely to change within your program run.

Automatically allocated objects are non-persistent, and ARE destroyed when they come out of scope. For example:

```
int main(){
    PersistVptr<X> px,pa;
    X x;             // non-persistent, automatic
    X *ppx;          // should not appear in your program

    px=new X;        // one persistent object
    pa=X::newArr(7); // array of 7 persistent objects
    ...
    // destroying the objects
    px->delObj();    // equivalent of: delete px;
    pa->delArr();    // equivalent of: delete[] pa;
    ...
    // never do this
    ppx=new X;       // WRONG: pointer in a temporary page
    delete pa;       // WRONG syntax: pa is an object and not pointer
    delete &pa;      // destroys pa but not the persistent object
} // here x is destroyed automatically
```

The allocation of strings is similar

```
    PersistString name;            // pointer to a persistent string
    char *p; static char *s[]={"rb","wb"}; // non-persistent strings

    p=new char[8];                   // non-persistent string 8 bytes long
    name=new PersistString(8);       // persistent string 8 bytes long
    name=new PersistString("first"); // make the text persistent
    name=new PersistString(s[0]);    // another example
```

```
...
delete[] p;        // destroying non-persistent object
name->delString(); // moves string to free list, retains the pointer
delete name;       // destroys the pointer, moves string to free list
```

Notes: An array of T objects must be allocated with PersistPtr::newArr(int size), not with T::new[](). We were not able to overload this operator successfully due to its unpredictable behaviour in VC++.

Also, neither delete() nor delete[]() can be easily overloaded in our case:

```
class A { ... };
PersistPtr<A> p;
p = new A; // this works
delete p; // compiler error
```

The compiler does not know how to interpret the last line. If we write

```
delete &(*p);
```

the compiler understands that we are dealing with class A, but we lose access to the disk address which is critical here.Another possibility is to write

```
delete A::(&p);
```

which is forced to enter A::delete(), and passes to it p which stores the disk address. This syntax is however error prone and ugly. My preference still is to use a separate function:

```
p = new A;
p.delObj();
```

Also note that even though we use PersistPtr as a pointer, it is really and object, which reflects in how we can use it in if() statements:

```
class A { ... };
PersistPtr<A> p;
if(p==NULL)...  // works as expected
if(p!=NULL)...  // works as expected
if(p)  ...      // does not work
if(!p) ...      // does not work
```

DEFAULT CONSTRUCTORS

PPF has to update special fields (virtual function pointers) which are implanted into objects by the C++ compiler. Many programmers are not even aware that these pointers are hidden in their objects.

Default constructors of certain classes are essential for the management of these pointers, and this leads to the following **rules**:

1. **In PPF Ver.1.4 and less:** PersistVptr classes and classes that form members in other classes must have empty default constructors. This implies the necessity to have additional functions (called *init()* below) for initialization.
2. **In PPF Ver.2.0 and higher:** PersistVptr classes and classes that form members in other classes must either have empty default constructors, of these constructors must start with the *PersistConstructor;* statement.

```
// ----------------------
// PPF Ver.1.4 and before
// ----------------------
```

```
class Y {
    PersistClass(Y);
    int k;
    ...
public:
    Y(){}   // must be empty because Y forms a member X::y
    void init(){k=0;}
    ...
};

class X {
    PersistVclass(X);
    int i;
    Y y;
    ...
public:
    X(){}  // must be empty because X is PersistVclass
    void init(){i=0; y.init();} // normally would be in the default constructor
    X(int k){i=k;}              // other constructors coded as usual
    ...
};

PersistVptr px=new X;
px->init();
...
px=new X(7); // the other constructor

// ---------------------
// PPF Ver.2.0 and later
// ---------------------
class Y {
    PersistClass(Y);
    int k;
    ...
public:
    Y(){PersistConstructor; k=0;}  // default constructor
    ...
};

class X {
    PersistVclass(X);
    int i;
    Y y;
    ...
public:
    X(){PersistConstructor; i=0;}  // Y() called for y as default
    X(int k){i=k;} // other constructors coded as usual
    ...
};

PersistVptr px=new X; // no call to init()
...
px=new X(7);          // as before
```

If you don't use any *PersistVclass* classes (all your persistent classes are only *PersistClass*), and no class uses members which are instances of another class or, in other words, you only have lightweight classes, then you don't have to worry about these rules. **However, if** *PersistConstructor* **must be used even for a single class, it is recommended that you apply it indiscriminantly to all your classes.**

The reason is that **if you forget about this rule for some class, PPF will malfunction in a hideous way, which is difficult to diagnose**

and debug.

There is no special rule for destructors, they can be used as usual.

HOW THE DATA IS STORED

Objects of each class are stored in one file, using the class name as the file name. For example, class MyBook will be stored in file mybook.ppf. Under Windows or DOS, select class names not longer than 8 characters. Longer names may work, but not always. There is no problem with the size of the class names under UNIX.

Storing objects of each class in its own file improves the organization and performance. If objects of a certain class are not used in your run, the file for that class will not be even opened. Also, if many objects of some class are frequently needed, you can give this class more paging space, and thus to improve the overall performance.

In addition to one *.ppf file for each class, there is also file classes.ppf. In this file, PPF keeps the list of classes used by your program. This list is ordered in certain way, and is critical for the correct operation of PPF.

The necessity of dealing with dozens and perhaps hundreds of files may look like a nuisance, but you can easily combine these files by using pkzip or tar for all *.ppf files, and unpack them later.

AN EFFICIENT IMPLEMENTATION

This entire system is elegant and efficient. Compare this with OODB systems or persistent mechanisms provided by other vendors. When you use PPF, you need only:

```
factory.h   ... some 300 lines of code (without space lines and comments)
pointer.cpp ... some 50 lines of code (without space lines and comments)
factory.obj ... about 30kB
```

The program is written with C++ templates, and is platform independent. It should run under any operating system and/or compiler.

The following numbers demonstrate the efficiency of the disk storage: When running program test1 (regr.bat) under Win95, sizeof(A)=4, sizeof(B)=12, you store 46,530 objects of class A, each 4B, total size of 186,120 bytes. If you allocate A objects as an array, this would be total of 186,120 bytes. If you allocate them individually, the allocator uses internally 4 bytes for each object, and the total memory used is 372,240. The size of the disk file to which the data is paged is 186,288 bytes. This represents only 1/100 of percent overhead over a non-persistent array representation, or about 1/2 of the memory needed for non-persistent, individually allocated objects.

TEST EXAMPLES

Look at the script file regr.bat or regr; they run a regression test of several programs.

Test1 represents a complete, running example with 2 light-weight classes. The example includes a linked list of arrays, plus a simple class-to-class pointer link.

test1.cpp is the main program, and includes the description of the problem.

test1a.h, test1a.cpp implement class A,

test1b.h, test1b.cpp implement class B.

If you check the sizes of files a.ppf and b.ppf, you will see that there is practically no overhead when storing the data on disk.

In Test2, we have a linked list of B objects, where each B object keeps a single A object, plus an array of A objects - each array has a different size. Test2 is similar to test1, except that we have classes A,B,C,and D, where C and D are derived (inherit) from B. We still have a linked list of B objects, but these objects truly are C,D,C,D,...,C,D. The attached A objects and arrays of A's are the same as in test1.

Test3 has a linked list of a light-weight class A, with two strings (name and address) attached to each A object. All three programs (test1,

test2, test3) generate the data internally, and then check the results automatically. When everything is all right, they print message "No errors".

```
test1 1   generates data for test1, checks it, and leaves it on disk.
test1 2   opens the disk files and checks the data.
test2 1   generates data for test2, checks it, and leaves it on disk.
test2 2   opens the disk files and checks the data.
test3 1   generates data for test3, checks it, and leaves it on disk.
test3 2   opens the disk files, releases 1/8 of the strings, and
          they are all removed, replaces them by a different string
          of the same length. It leaves results on disk - both files
          a.ppf and string.ppf should have the same size as originally.
test3 3   opens the disk files, and checks that the new data is in place.
```