Working With Object Stores of Events Using ptool

Robert Grossman
University of Illinois at Chicago, Chicago, USA

Abstract

The purpose of these notes is to give an expository treatment of how to use a persistent object manager called ptool to analyze high energy physics data. Traditionally, the analysis of scientific data has been file based. On the other hand, scientific data often has a rich semantic complexity, which mirrors objects more closely than files. The goal of ptool was to replace file based analysis of scientific data with object based access in a scalable fashion. Ptool was designed to scale as the amount of data scales by interfacing object stores to hierarchical storage systems; and as the complexity of the query scales by providing the ability to query objects in parallel.

1 Introduction

The purpose of these notes is to give an expository treatment of how to use a persistent object manager called ptool to analyze high energy physics data. Traditionally, the analysis of scientific data has been file based. On the other hand, scientific data often has a rich semantic complexity, which mirrors objects more closely than files. The goal of ptool was to replace file based analysis of scientific data with object based access in a scalable fashion. Ptool was designed:

• to scale as the amount of data scales by interfacing object stores to hierarchical storage systems;

• to scale as the complexity of the query scales by providing the ability to query objects in parallel.

Another important design goal was to provide the ability to create, update and access persistent objects in a tool based framework [4] rather than a framework based upon object oriented databases. In particular, ptool was designed to provide high performance and low overhead. This was done by trading performance and overhead for functionality. Ptool only provides persistence for complex objects—any other functionality, such as concurrency or transactions must be layered on top of ptool.

The idea of using databases and object stores to analyze high energy physics data began with [18]. Ptool was initially developed for aeronautic applications [5], [8], [6]. A proposal for analyzing HEP data using object stores was described in [1], which led to the PASS Project. The PASS Project is a five year research and development project supported by the US Department of Energy under the High Performance Computing and Communications Program with the goal of developing new technology to analyze HEP data using database computing. The project is presently in its second year. As part of the PASS Project, ptool has been applied to the analysis of HEP data viewed as collections of persistent events [3] and [7].

The current notes are an expository treatment of the use of ptool to analyze HEP data. The research explained in these expository notes is the work of many individuals. This research is described in the reports [13] and [14]. These notes describe just one part

---

1) This research was supported in part by NASA grant NAG2-513, DOE grant DE-FG02-92ER25133, and NSF grants IRI 9224605 and CDA-9303433. For more information, contact Robert Grossman, Laboratory for Advanced Computing, m/c 249, University of Illinois at Chicago, 815 South Morgan Street, Chicago, IL 60607, grossman@math.uic.edu.
of the PASS Project. The PASS Project has also analyzed HEP data using both relational databases and object oriented databases, as well as analyzed the differences between these approaches [9].

These notes are organized as follows: Section 2 is a brief introduction to object stores; Section 3 describes the functional requirements for our problem; Section 4 describes the basic concepts of the system; Section 5 describes the physical data model; and Section 6 describes the architecture and implementation. There is also a rather lengthy appendix which contains program fragments with some basic examples. The purpose of the appendix is to give enough information so that someone familiar with UNIX can create, populate and access persistent stores of events.

Ptool was designed by the present author and Xiao Qin [10]. D. Valsamis and W. Xu also made important contributions to ptool. The eventstore [13] and [14] was designed and developed by the present author, D. Lifka, D. Malon, E. May, L. Price, X. Qin, and D. Valsamis.

The examples in these notes were run with ptool, version 0.6 release 2 on a Sun Sparcstation 1.

2 Object stores

We begin with a quick review of some basic definitions, which we discuss more fully below. An object is an abstract data type together with some functions, or methods, for creating, accessing, and modifying it. In the context of object stores, an element of the abstract data type is called an attribute of the object. An object is called persistent if it continues to exist once the process which creates it dies; otherwise, the object is called transient. An object manager is a process which can create and manage persistent collections of objects, called stores. An object oriented database is an object manager with additional functionality, such as transactions, back up, and recovery.

The underlying thesis of these notes is that for many applications file based access to scientific data should be replaced with object based access using an object manager. This provides several advantages:

- With object-based data access, scientific and engineering applications can easily exploit specialized application specific algorithms for storing and accessing data in order to gain higher performance. In other words, rather than accessing unstructured data using the file system, one accesses the relevant structured objects by querying a persistent object store.
- By using an object store instead of an object oriented database, one can usually obtain higher performance and greater scalability. This is because the additional functionality of an object oriented database involves trading performance for functionality. With a persistent object store, one can incrementally add functionality as required for a specific application.
- By using an object store, code for the application specific analysis of data can be modularly separated from more generic code to store and access data.

Although partly misleading, it is helpful to consider the analogies in Table 1 between file systems and object stores and the analogies in Table 2 between relational databases and object oriented databases.

3 Functional Requirements

Consider the following characteristics of the off-line analysis of HEP data viewed as object stores of events:

- Large numbers of complex objects must be accessed.
- Data analysis consists of numerically intensive queries, which are normally expressed in Fortran or a similar language.
- Although large numbers of objects are examined in a query, a typical query touches only a small percentage of the data in each object, say 0.1%. Different queries touch
<table>
<thead>
<tr>
<th>file system</th>
<th>object store</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory</td>
<td>set of sets</td>
</tr>
<tr>
<td>file</td>
<td>set of objects</td>
</tr>
<tr>
<td>record</td>
<td>object</td>
</tr>
<tr>
<td>field</td>
<td>attribute</td>
</tr>
<tr>
<td>distributed file system</td>
<td>distributed object store</td>
</tr>
<tr>
<td>hierarchical file system</td>
<td>multi-level object store</td>
</tr>
</tbody>
</table>

Table 1: Some analogies between file systems and object stores.

<table>
<thead>
<tr>
<th>relational database</th>
<th>object oriented database</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>store of objects</td>
</tr>
<tr>
<td>table</td>
<td>set of objects</td>
</tr>
<tr>
<td>tuple</td>
<td>object</td>
</tr>
<tr>
<td>structured query language</td>
<td>object query language</td>
</tr>
</tbody>
</table>

Table 2: Some analogies between relational and object oriented databases.

different portions of the data.

- The data is historical in the sense that it is written once and rarely updated.

Other applications which share these characteristics include: the analysis of seismic data, the analysis of consumer marketing data, and anomaly detection. Collectively, these applications may be viewed as data mining applications. Each is looking for an application specific needle in an application specific haystack.

Ptool was designed for these types of applications. The intention was to support
1. access to large numbers of complex objects distributed on a variety of storage media;
2. and numerically intensive queries.

These requirements are noteworthy in what they omit: there is no direct support for transactions, back up, or recovery. The decision was made to trade performance for functionality in these areas, since the data is primarily historical.

4 Concepts

In this section, we describe a reference model for computing with distributed stores of objects. As throughout the notes, this is an expository treatment of research described elsewhere. In particular, this section is adapted from [11] and [17].

The reference model uses just three primary concepts:

- Objects. The data is assumed to be organized into objects.
- Processes. Computation consists of communicating processes which act upon objects.
- Nodes. Objects and processes are distributed among nodes in a network, with two nodes being able to exchange objects in case there is a path connecting them.

In addition, the following three concepts, which are defined in terms of the basic concepts, are also important:

- Persistent Objects. An object is called persistent if it exists independently of the process which creates it. Otherwise an object is called transient. In other words, transient objects basically reside in memory, while persistent objects also reside on disk or other permanent media.
- Brokers. Processes may make requests of other processes, called brokers, to take certain actions on objects.
- Folios. For efficiency, objects are collected into physical units called folios, which the system manages. Components of the system respond to distributed requests.
for objects and folios. In turn, folios may themselves consist of smaller physical
units, called subfolios or segments.

We now elaborate on these concepts.

As already mentioned above, an object is an abstract data type together with some
functions, or methods, for creating, accessing, and modifying it. From the viewpoint of
object stores, an element of the abstract data type is called an attribute of the object.
Objects which share the same attributes and methods form classes. We assume that each
persistent object is assigned a unique id, called a persistent object id and that each object
belongs to one persistent store or store.

An object manager creates, updates, and accesses persistent objects. Object oriented
databases, in addition to providing persistence, also provide additional functionality such
as transactions, back up, and recovery.

It is convenient to organize objects into logical units called collections. Collections
may contain the actual objects themselves or pointers to the objects. There are many
types of collections. For example, unordered collections without duplicates are called sets;
ordered collections are called lists. A collection itself is an object. A store is a top level
collection which for this reason is handled somewhat differently by the object manager.

Since collections are themselves objects, functions may be applied to collections. If
the result is another collection, the function is called a query. Notice that this definition of
query includes queries that select objects for inclusion into the output collection, queries
that compute new objects from the input objects, and queries which both select objects
and compute new objects. When new objects are produced by a query, this additional
data is sometimes called derived data.

Nodes are the abstractions of the different physical devices in a distributed comput-
ing environment. Nodes abstract physical memory, virtual memory, disks, tapes, work-
stations, disk arrays, clusters of workstations, etc. For example, brokers can move folios
from node to node, abstracting the movement of folios from workstation to workstation,
from tape to disk, and from disk to disk.

5 Physical Data Model

The object manager must ultimately pass objects to a storage manager in order
for the objects to be persistent. There are several possibilities: the storage manager
may be part of the object manager itself, the storage manager may be accessed through an
API, the storage manager may be accessed through a file system and its API, or the
storage manager may be accesses through a hierarchical storage system and its API.
There are several versions of ptool available. Ptool32 accesses the storage system through
a file system API, while ptool64 accesses the storage system through a hierarchical storage
system API. Although the API's are essentially the same, providing an additional level of caching when interfacing to a hierarchical storage system provides better performance.

For this reason, ptool32 uses single level caching as in Diagram 1, while ptool64 uses a
multi-level caching as in Diagram 2.

The current version of ptool64 (version 0.6) interfaces to a hierarchical file system
through the abstraction of a bitfile, as described in the IEEE Hierarchical Storage Sys-

tem Reference Model, version 4 [2]. It is a current area of research to design the most
appropriate interface between object managers, file systems, and storage systems. Later
versions of ptool plan to provide direct interfaces to storage systems, as in version 5 of
Storage System Reference Model [20].

As mentioned, to increase performance objects are gathered into physical units
called folios and subfolios or segments. Objects may span one or more segments, and even
one or more folios. We assume that each folio has a folio id attached to it, and each
segment has a segment id attached to it. Briefly in ptool64, there is a cache manager,
called a segment cache manager, which manages segments and another, called a network
cache manager or folio cache manager, which manages folios. The folio cache manager
Figure 1: The design of ptool32. Ptool32 provides uniform access to a persistent space of objects. The Persistent Object Manager provides persistence for complex objects by mapping physical extents of virtual memory called segments. When a segment that is not currently mapped is needed, the Persistent Object Manager generates a fault to the Segment Cache Manager. If the segment is available in the local segment cache, it is mapped into virtual memory. Otherwise a fault is generated, and the segment is obtained from the appropriate data file. This figure is from [10].
extracts folios from bitfiles, while the segment cache manager extracts segments from folios. Bitfiles are managed by the hierarchical storage system. See Diagram 2. In ptool32, there is simply a segment cache manager which interfaces directly to the file system, as illustrated in 1.

Persistent objects must be stored on some permanent media; because of this, each persistent object is associated with a physical address, or physical id. For example, a persistent object may have a virtual memory address or byte location within a file associated to it in this way. Note that this physical address may change. For example, this happens if the object is cached or migrated. In addition, each object has a logical object id associated to it. By assumption, this does not change, despite any changes to the object. There are several possibilities:

1. The pid is always the physical address just described.
2. The pid is always the logical id and a table is maintained between the physical addresses and the logical ids.
3. The physical address can be computed from the logical id, and perhaps some auxiliary information.
4. The pid is sometimes the physical address and sometimes the logical id; tables are maintained as necessary. The pid is said to swizzle between the two.

Ptool uses the third possibility.

6 Architecture and Implementation

In this section, we describe the architecture of ptool32 and ptool64. Ptool64 provides a 64 bit persistent address space, while ptool32 provides a 32 bit persistent address space. The main difference in architecture is that ptool64 uses multi-level caching and migration, while ptool32 uses single level caching and migration. In the current implementation (version 0.6), there are additional differences arising from the fact that the languages, compilers, and operating systems used do not support 64 bit pointers. A future release of ptool is planned which does exploit 64 bit systems. We now describe the various components of the system: the Persistent Object Manager, the Segment Cache Manager, and the Folio Cache Manager.

The Persistent Object Manager is responsible for creating, storing, and accessing complex persistent objects. The Persistent Object Manager is implemented using virtual memory mapping techniques [19]. The Persistent Object Manager uses the UNIX function mmap to map identified disk regions to virtual memory. When objects in this identified region are referenced, the virtual memory system is responsible for moving them in and out of memory. The Persistent Object Manager divides the portion of virtual memory it uses into several slots. The segments or subfolios managed by the Persistent Object Manager are moved into one of the slots when needed. If a referenced object is contained in a segment which is already mapped, then the referenced object is returned. If not, the Persistent Object Manager generates a fault to the Segment Cache Manager.

The Segment Cache Manager maintains a cache of segments. If the requested segment is available in this cache, it is returned to the Persistent Object Manager, which frees a slot of virtual memory it manages, and maps the segment into the slot. On the other hand, if the segment is not available, a fault is generated. In ptool32, this fault causes ptool32 to retrieve the required segment from the appropriate data file. See Figure 1.

In ptool64, an additional level of caching is used. The segment fault causes the Folio Cache Manager to search the cache of folios it maintains. If the segment is available in one of these folios, it is extracted and returned to the Segment Cache Manager. If not, the Folio Cache Manager generates a fault to the hierarchical storage system which obtains the required file and extracts the folio. See Figures 2, 3.

In applications, ptool32 is usually used on a single workstation with an attached local disk and one level of caching suffices. While ptool64 is usually used in a networked
Figure 2: The design of ptool64: View 1. Ptool64 provides uniform, scalable access to a persistent space of objects. The Persistent Object Manager provides persistence for complex objects by mapping physical extents of virtual memory called segments. When a segment that is not currently mapped is needed, the Persistent Object Manager generates a fault to the Segment Cache Manager. If the segment is available in the local segment cache, it is mapped into virtual memory. Otherwise a fault is generated and passed to the Folio Cache Manager. The segment may be available in the Folio Cache Manager's segment cache; if not, a fault is generated to the hierarchical storage system to retrieve the corresponding physical collection of segments called a folio which contains the needed segment. Pre-emptive prefetching is used to improve performance. This figure is from [12].
Figure 3: The design of ptool64: View 2. The multi-level caching algorithm implemented in version 0.6 of ptool64 is designed to exploit the structure of a heterogeneous computing network with a high performance connection to hierarchical storage. The Network Cache Manager is assumed to transport folios over a high bandwidth channel between the hierarchical storage system and a node on the network. Segment Cache Managers run on other nodes and transport segments over the network at a lower bandwidth. We assume that the Persistent Object Managers which run on each node have a high bandwidth means of transporting objects between virtual memory and the local disk. This figure is from [12].
environment with local and remote disks and other storage media. Our experience has shown the advantages of multi-level caching in these environments [12].

A  Persistent Objects and ptool32
In this section, we describe how to create and access persistent objects, following [10], using the example of events. For simplicity, we put each event directly in the store. More commonly, events would be gathered together into sets and the sets placed in the store, as we describe in the next section.

There are three main steps to create and access a store of events: first, a scheme for the store is designed, which defines the objects and their attributes; second, a store is created and populated with objects; and third, the store is queried for objects meeting specified criteria.

1. The first step is to define the schema for the store by defining the objects and their attributes. This is done by creating a file, in this case event.h, containing the classes defining the objects, and the member variables defining the attributes. See Figure 4. In this example, a Lepton object consists of a four vector and a charge, and an Event object consists of two Leptons, plus some additional data.

2. The second step is to populate the store. Figure 7 contains the code to do this. It begins with the statement #include "event.h" defining the schema. The next step is to create a store with the statement store a("PsiEvents");. This creates a store with the internal handle a, and the external name PsiEvents. If the store PsiEvents already exists, then this program would append objects to it. The store must also be closed with the statement: a.close(). To make an object persistent, the standard C++ declaration

    Event *e;
    e = new Event;

is replaced by

    Event *e;
    e = new(&a) Event;

In order to access the object later, an entry point must be created for it in the store with the statement:

    a.add(e);

Attributes for persistent objects and transient objects are accessed in the same way:

    e->vertex=(double)drand48()

assigns a random double to the vertex attribute of the Event object e. Finally, note that if a persistent object contains pointers to other persistent objects, then these must also be made persistent, as with

    e->lepton1=ll=new(&a) Lepton

But note that it is not necessary to create an entry point for these, since they are automatically accessed whenever their parent object is accessed.

3. The third step is to query the persistent store of objects. Figure 8 contains a simple program to do this. The schema is included at the beginning: #include "event.h". The store with the external name PsiEvents is opened with the internal handle b using the statement

    store b("PsiEvents");

and later closed with b.close(). To loop through all objects in a store is easy:

    Event *e;
    for (e=(Event*)b.first(); b.more(); b=(Event*)b.next())
Note that this assumes that all objects in the store are of the same type. Also note that the objects must be explicitly cast into objects of the correct type.

To summarize: first, define the classes for the persistent objects; second, populate the store by persistently allocating instances of the class using

```c
store a("PsiEvents");
Event *e;
e = new(&a) Event;
```

third, iterate over the objects in the store and make selections.
#ifndef __EVENT__
#define __EVENT__

#include <iostream.h>
#include "ptool32.h"

enum populationMode { randomPopulation = 0,
        sequentialPopulation = 1 };
class Lepton;

class Event {
public:
    int runNumber;
    int eventNumber;
    double vertex;
    Lepton *lepton1;
    Lepton *lepton2;

    Event(populationMode, store*);
    friend ostream& operator << (ostream& os, Event& l);
};

class Lepton {
public:
    double p[4];
    double charge;

    Lepton(populationMode);
    friend ostream& operator << (ostream& os, Lepton& l);
};

#endif

Figure 4: This figure contains the schema for a very simple event object containing just a few attributes. With ptool, objects are defined using C++ classes and attributes are nothing more than data members of the corresponding classes. This is the simplest of three examples showing how ptool can be used to create eventstores. The first two examples use ptool32 and the third example uses ptool64.

#include <iostream.h>
#include "event.h"

extern "C" {
    void srand(int);
    long rand();
    long random();
    double drand48();
}

Event::Event(populationMode m, store *a) {
    if (m==randomPopulation) {
        Lepton *l1, *l2;
        runNumber=(int)rand()%10000;
        eventNumber=(int)rand()%10000;
        vertex=(double)drand48();

        lepton1=l1=new(a) Lepton(randomPopulation);
        lepton2=l2=new(a) Lepton(randomPopulation);
    }
}

Lepton::Lepton(populationMode m) {
    if (m==randomPopulation) {
        p[0]=(double)drand48();
        p[1]=(double)drand48();
        p[2]=(double)drand48();
        p[3]=(double)drand48();
        charge=(double)drand48();
    }
}

Figure 5: This figure defines the methods for the eventstore. With ptool, objects are defined using C++ classes and methods are nothing more than member functions of the corresponding classes. To simplify the code for this simple example, the values of the attributes are defined randomly.
ostream& operator << (ostream& os, Event& t) {
  os << "Event: \n";
  os << "run# = " << t.runNumber << "\t";
  os << "event# = " << t.eventNumber << "\t";
  os << "vertex = " << t.vertex << "\n";
  os << *(t.lepton1);
  os << *(t.lepton2);
  os << "\n";
  return os;
}

ostream& operator << (ostream& os, Lepton& l) {
  os << "Lepton: \n";
  os << "p0 = " << l.p[0] << "\t";
  os << "p1 = " << l.p[1] << "\t";
  os << "p2 = " << l.p[2] << "\t";
  os << "p3 = " << l.p[3] << "\n";
  os << "t \ t charge = " << l.charge << "\n";
  return os;
}

Figure 6: The methods for a simple eventstore created with ptool32 (continued).

#include <string.h>
#include <stdio.h>
#include "ptool32.h"
#include "event.h"

void main()
{
  store a("PsiEvents");
  Event *tmp;

  for (int i=0; i<10; i++) {
    tmp = new(&a) Event(randomPopulation,&a);
    a.add(tmp);
  }
  a.close();
}

Figure 7: This figure shows how to populate a simple eventstore containing 10 events using ptool32.
```cpp
#include <iostream.h>
#include "event.h"

extern "C" {
    void srand(int);
    long rand();
    long random();
    double drand48();
}

Event::Event(populationMode m, store *a) {
    if (m==randomPopulation) {
        Lepton *11, *12;
        runNumber=(int)rand()%10000;
        eventNumber=(int)rand()%10000;
        vertex=(double)drand48();

        lepton1=1i=new(a) Lepton(randomPopulation);
        lepton2=12=new(a) Lepton(randomPopulation);
    }
};

Lepton::Lepton(populationMode m) {
    if (m==randomPopulation) {
        p[0]=(double)drand48();
        p[1]=(double)drand48();
        p[2]=(double)drand48();
        p[3]=(double)drand48();
        charge=(double)drand48();
    }
};
```

Figure 8: This figure shows how to access an eventstore using ptool32 by looping through all the events.
B Persistent Sets

Sets have two fundamental roles in object stores:
1. Sets organize collections of objects, just as directories organize collections of files. Sets may contain other sets of objects, just as directories may contain other directories of files.
2. Queries map one set of objects to another set of objects. By some combination of selecting objects and computing new objects from the input set. From this viewpoint, sets are analogous to tables in relational databases, since relational queries map one or more input tables to an output table.

There are a variety of different ways of constructing sets: in this section, we describe one of the most basic ways in order to introduce the idea. In practice, one would use more elaborate variants, or alternatives such as templates. To define our illustrative set class, one uses an auxiliary class

```cpp
class Link {
public:
    Link() { next = 0; }
    Link *Next() { return next; }
protected:
    Link *next;
};
```

Then to define a set of Events for example, one modifies the class for Event to inherit Link

```cpp
class Event : public Link {
};
```

A Set class can then be defined by simply creating pointers to the head of the set, to the current position in the set, and providing methods to add and delete elements from the set. Such a set class will allow one to create sets of any object which inherit Link. See Figures 9 and 10.

Given the classes Link and Set, persistent sets for the store

```cpp
store a("PsiEvents");
```

can be defined as follows:
1. Define a persistent set in the store a

   ```cpp
   Set *PsiSet;
   PsiSet = new(&a) Set;
   ```

   Note that the first statement declares that PsiSet is a pointer to a Set, while the second statement defines the variable. The definition of a variable allocates storage for it. In this case, the presence of the over loaded new indicates that the storage is persistent.

2. Define a persistent element of the set

   ```cpp
   Link * tmp;
   tmp = new(&a) Event();
   ```

   This gives us a persistent Event which, since it inherits the class Link, can be an element of a persistent set. At this point, although tmp is persistent, it is not part of any set. Furthermore, there is no way to reference it in the store, since no entry point for it has been created.

3. The statement

   ```cpp
   PsiSet->Add(tmp);
   ```

adds the persistent tmp to the set PsiSet.
#ifndef __SET__
#define __SET__

#define print_set_element(objLink, genLink) \
    cout << *( (objLink *)genLink );

class Link {
    friend class Set;
public:
    Link() { next = 0; }
    Link *Next() { return next; }
protected:
    Link *next;
};

class Set {
public:
    Set() { head = tail = current = 0; length = 0; }
    Set& Add( Link* );
    Set& AddFirst( Link* );

    long Length() { return length; } 
    Link* Next();
    Link* First() { return head; } 
    Link* Last() { return tail; }

    void Discard(); 
    void reset() { current = head; }
protected:
    Link *head;
    Link *tail;
    Link *current;
    long length;
};
#endif

Figure 9: The class definition for a very basic container class.
```cpp
#include <iostream.h>
#include "set.h"

Set& Set::AddFirst( Link *newNode ) {
    length++;  
    newNode -> next = head; 
    head = newNode; 
    if ( !tail ) tail = current = newNode; 

    return *this; }

Set& Set::Add( Link *newNode ) {
    length++;  
    newNode -> next = 0; 
    if ( tail ) {
        tail -> next = newNode; 
        tail = newNode; 
        current = ( !current ) ? newNode : current; 
    } else 
        head = tail = current = newNode; 

    return *this; }

void Set::Discard() {
    Link *todelete; 

    while( todelete = Next() ) {
        delete todelete; 
        head = current; 
    }
    head = tail = current = 0; 
    length = 0; }

Link* Set::Next() {
    Link *ret = current; 

    current = ( current ) ? current -> next : head; 
    return ret; }
```

Figure 10: The methods for a very basic container class. The container class is simply a linked list with pointers to the first, last and current elements.
#ifndef __EVENT__
#define __EVENT__

#include <iostream.h>
#include "ptool32.h"
#include "set.h"

enum populationMode { randomPopulation = 0,
  sequentialPopulation = 1 };

class Lepton;
class Jet;

class Event : public Link {
public:
  int runNumber;
  int eventNumber;
  double vertex;
  Lepton *lepton1;
  Lepton *lepton2;
  Set *event_to_jet_set;
  Event(populationMode, store*);
  friend ostream& operator<<(ostream& os, Event& l);
};

class Lepton {
public:
  double p[4];
  double charge;
  Lepton(populationMode);
  friend ostream& operator<<(ostream& os, Lepton& l);
};

class Jet : public Link {
public:
  int jet_num;
  int ntrk;
  double phi_jet;
  double eta_jet;
  double pt;
  double mass_jet;
  Jet(populationMode);
  friend ostream& operator<<(ostream&, Jet&);
};
#endif

Figure 11: The second version of an eventstore. This eventstore is also created using ptool32, but this time an event can contain one or more jets. This figure defines the classes.
```cpp
#include <iostream.h>
#include "event.h"
#include "set.h"

extern "C" {
    void srand(int);
    long rand();
    long random();
    double drand48();
}

Event::Event(populationMode m, store *a) {
    if (m==randomPopulation) {
        Lepton *l1, *l2;
        runNumber=(int)rand()%10000;
        eventNumber=(int)rand()%10000;
        vertex=(double)drand48();

        lepton1=l1=new(a) Lepton(randomPopulation);
        lepton2=l2=new(a) Lepton(randomPopulation);

        Link *tmpjet;
        int numberJets = ((int)random()%3)+1;
        event_to_jet_set = new(a) Set;
        for (int i=0; i<numberJets; i++) {
            tmpjet = new(a) Jet(randomPopulation);
            event_to_jet_set->Add(tmpjet);
        }
    }
};

Lepton::Lepton(populationMode m) {
    if (m==randomPopulation) {
        p[0]=(double)drand48();
        p[1]=(double)drand48();
        p[2]=(double)drand48();
        p[3]=(double)drand48();
        charge=(double)drand48();
    }
};
```

Figure 12: The methods for the event class for the second version of the eventstore. Again, for simplicity, the attributes are given random values. In particular, each event is given 1, 2 or 3 jets, with the number of jets determined randomly.
Jet::Jet(populationMode m) {
    if (m==randomPopulation) {
        jet_num=(int)rand()%10;
        ntrk=(int)rand()%100;
        q_frac=(double)drand48();
        phi_jet=(double)drand48();
        eta_jet=(double)drand48();
        pt=(double)drand48();
        mass_jet=(double)drand48();
    }
}

ostream& operator << (ostream& os, Event& t) {
    os << "Event: \n";
    os << "run# = " << t.runNumber << "\t";
    os << "event# = " << t.eventNumber << "\t";
    os << "vertex = " << t.vertex << "\n";
    os << *(t.lepton1);
    os << *(t.lepton2);
    Jet *tmpjet;
    while (tmpjet = (Jet*) t.event_to_jet_set->Next())
        os << *tmpjet;
    os << "\n";
    return os;
}

ostream& operator << (ostream& os, Lepton& l) {
    os << "Lepton: \n";
    os << "p0 = " << l.p[0] << "\t";
    os << "p1 = " << l.p[1] << "\t";
    os << "p2 = " << l.p[2] << "\t";
    os << "p3 = " << l.p[3] << "\n";
    os << "t \t charge = " << l.charge << "\n";
    return os;
}

ostream& operator << (ostream& os, Jet& l) {
    os << "jet: \n";
    os << "jet_number = " << l.jet_num << "\t";
    os << "ntrk = " << l.ntrk << "\t";
    os << "q_frac = " << l.q_frac << "\t";
    os << "phi_jet= " << l.phi_jet << "\n";
    os << "t \t eta_jet = " << l.eta_jet << "\t";
    os << "pt = " << l.pt << "\t";
    os << "mass_jet = " << l.mass_jet << "\n";
    return os;
}

Figure 13: The methods for the second version of the eventstore (continued).
4. The statement
   
   a.add(PsiSet);

   adds an entry point for the persistent set PsiSet to the store a, allowing subsequent
   access to the persistent set PsiSet in the same way as for any other persistent object.
   In this way, we can populate stores with persistent sets of persistent objects. See
   Figure 14. Consider a store store b("PsiEvents") consisting of sets of Events. Accessing
   the persistent sets of persistent events consists of two steps:

   1. Since the persistent sets are themselves persistent objects, they can be accessing just
      as any other persistent objects:

      ```
      Set *PsiSet;
      for (PsiSet=(Set*)b.first();
           b.more(); PsiSet=(Set*)b.next())
      ```

   2. Given a persistent set, one can follow the pointers next from one Link to the next.
      The method Next() from the class Link does this automatically.

      ```
      Event *e;
      while (e=(Event*) PsiSet->Next())
          cout << *e;
      ```

   See Figure 15.
```c
#include <string.h>
#include <stdio.h>

#include "ptool32.h"

#include "event.h"
#include "set.h"

void main()
{
    store a("PsiEvents");

    Link *tmp;
    Set *PsiSet;

    PsiSet = new(&a) Set;
    for (int i=0; i<10; i++) {
        tmp = new(&a) Event(randomPopulation,&a);
        PsiSet->Add(tmp);
    }
    a.add(PsiSet);

    PsiSet = new(&a) Set;
    for (i=0; i<20; i++) {
        tmp = new(&a) Event(randomPopulation,&a);
        PsiSet->Add(tmp);
    }
    a.add(PsiSet);

    a.close();
}
```

Figure 14: In this version of the eventstore, ptool32 is used to populate an eventstore containing two sets of events. The first set contains 10 events and the second set contains 20 events.
```c
#include <stdio.h>  
#include "ptool32.h"

#include "set.h"  
#include "event.h"

void main()
{
  Set *PsiSet;  
  Event *e;  
  store b("PsiEvents");

  int count = 1;
  for( PsiSet=(Set*)b.first(); b.more();
      PsiSet=(Set*)b.next()) {
    cout << "Set" << count++ << "\n";
    PsiSet -> reset();
    while( e =(Event*) PsiSet -> Next() ) {
      cout << *e;
      
    }
  }

  b.close();
}
```

Figure 15: Ptool32 is used to access an eventstore containing one or more sets of events by looping over all such sets. For each set accessed, each event in the set is accessed in turn.
C  Pointers to Persistent Objects and ptool64

Recall that object ids should be unique even across stores. For this reason, many
applications require object ids that are 64 bits or even 96 bits long. This allows enough
room to include an id for the store as well as for the object within the store. For example,
the current implementation of ptool64 uses object ids of the form

    int store_id;  // 16 bits
    int folio_id;  // 8 bits
    int segment_id;  // 8 bits
    int offset_id;  // 16-21 bits
    // 11-16 bits reserved

For compilers which support only 32 bit pointers, special effort must be used to work with
pointers which are 64 bits long. For example, the statements

    Event *e;
    e = new Event;

create a 32 bit pointer e. To define 64 bit pointers some other scheme must be used. There
are many ways to do this.

One simple method is to define a special class consisting of 64 bit persistent pointers,
say pptr. Pointers to objects of a given class can then inherit from the class pptr. For
example, the following code fragment declares ppEvent to be a 64 bit pointer to persistent
Events.

    class ppEvent : public pptr {
        Event* operator->();
        // etc.
    };

Another problem is that with most compilers today, new returns a 32 bit pointer. Not all
compilers support overloading new to return a different structure. Again, there are several
ways of proceeding. One way is to return the 64 bit persistent pointer as an argument to
new. In the following example a 64 bit persistent pointer tmp is first declared and then
defined upon the return of the overloaded new.

    ppEvent tmp;
    new(&a, &tmp) Event(&a);

Since this approach requires that each class have its own persistent pointer class, it
is helpful to automate this definition. Since methods cannot return class definitions, we
use a precompiler directive to do this:

    #define PPTR(X,Y)         
    class Y : public pptr { 
        X* operator->(); 
        // etc.
        Y& operator=(pptr pp) 
        // etc.
    };

To define a persistent 64 bit pointer tmp, an instance of the class ppEvent which points
to persistent Events, is now easy:

    PPTR(Event, ppEvent);
    ppEvent tmp;
    new(&a, &tmp) Event(&a);

To summarize the previous discussion, ptool64 works exactly like ptool32 except
that one defines a class of persistent pointers for each class of persistent objects and uses
these pointers to manipulate persistent objects rather than the standard 32 bit pointers
supported by the compiler. Figures 16-22 contain the ptool64 version of the example in
the last section.
```cpp
#define __EVENT__

#include <iostream.h>
#include "ptool64.h"
#include "pset64.h"

enum populationMode { randomPopulation = 0,
    sequentialPopulation = 1
};

class Event;
class Lepton;
class Jet;

PPTR(Event, ppEvent);
PPTR(Lepton, ppLepton);
PPTR(Jet, ppJet);

class Event {
    public:
        int runNumber;
        int eventNumber;
        double vertex;
        ppLepton lepton1;
        ppLepton lepton2;
        Set64 event_to_jet_set;

        Event() {};
        Event(populationMode, store*);
        Event(ppEvent&, store*);
        friend ostream& operator << (ostream& os, ppEvent& l);
    
}

class Lepton {
    public:
        double p[4];
        double charge;

        Lepton() {};
        Lepton(populationMode);
        Lepton(ppLepton&);
        friend ostream& operator << (ostream& os, ppLepton& l);
    
}
```

Figure 16: This figure contains the class definitions for the third version of the eventstore. This version is created using ptool64, which requires that the persistent objects be accessed with 64 bit persistent pointers. These are defined for each class using the macro PPTR.
class Jet : public Link64 {
public:
    int jet_num;
    int ntrk;
    double q_frac;
    double phi_jet;
    double eta_jet;
    double pt;
    double mass_jet;

    Jet() {};
    Jet(populationMode);
    Jet(ppJet&);
    friend ostream& operator << (ostream&, ppJet& );
};
#endif

Figure 17: This figure contains additional class definitions. Since set of Jets will be needed, the class Jet inherits the class Link 64.
```cpp
#include <iostream.h>
#include "event.h"
#include "pset64.h"

extern "C" {
    void srand(int);
    long rand();
    long random();
    double drand48();
}

Event::Event(populationMode m, store *a) {
    if (m==randomPopulation) {
        runNumber=(int)rand()%10000;
        eventNumber=(int)rand()%10000;
        vertex=(double)drand48();

        new(a, &lepton1) Lepton(randomPopulation);
        new(a, &lepton2) Lepton(randomPopulation);

        pptr tmp;
        int numberJets = ((int)random()%3)+1;
        for (int i=0; i<numberJets; i++) {
            new(a, &tmp) Jet(randomPopulation);
            event_to_jets_set.Add(tmp);
        }
    }
}

Lepton::Lepton(populationMode m) {
    if (m==randomPopulation) {
        p[0]=(double)drand48();
        p[1]=(double)drand48();
        p[2]=(double)drand48();
        p[3]=(double)drand48();
        charge=(double)drand48();
    }
}
```

Figure 18: This figure and the next two contain the methods for the event class used for the third version of the eventstore, which uses ptool64. This is similar to the methods used for the second version, but somewhat more complicated since persistent objects cannot themselves be accessed, but must be accessed instead through 64 bit persistent pointers.
Jet::Jet(populationMode m) {
    if (m=randomPopulation) {
        jet_num=(int)rand()%10;
        ntrk=(int)rand()%100;
        q_frac=(double)drand48();
        phi_jet=(double)drand48();
        eta_jet=(double)drand48();
        pt=(double)drand48();
        mass_jet=(double)drand48();
    }
}

Event::Event(ppEvent& ppEv, store *a) {
    pptr iterator;
    runNumber=ppEv->runNumber;
    eventNumber=ppEv->eventNumber;
    vertex=ppEv->vertex;
    new(a, &lepton1) Lepton(ppEv->lepton1);
    new(a, &lepton2) Lepton(ppEv->lepton2);
    ppJet tmp; //temp handle new object
    ppJet ppjet; //temp handle for iterating set
    for (ppjet=ppEv->event_to_jet_set.First(iterator);
        ppEv->event_to_jet_set.More(iterator);
        ppjet=ppEv->event_to_jet_set.Next(iterator) ) {
        new(a, &tmp) Jet(ppjet);
        event_to.jet_set.Add(tmp);
    }
}

Lepton::Lepton(ppLepton& 1) {
    p[0]=l->p[0];
    p[1]=l->p[1];
    p[3]=l->p[3];
    charge=l->charge;
}

Jet::Jet(ppJet& j) {
    jet_num=j->jet_num;
    ntrk=j->ntrk;
    q_frac=j->q_frac;
    phi_jet=j->phi_jet;
    eta_jet=j->eta_jet;
    pt=j->pt;
    mass_jet=j->mass_jet;
}

Figure 19: The methods for the third version of the eventstore (continued).
ostream& operator << (ostream& os, ppEvent& t) {
    os << "\n----Event----\n";
    os << "runNum=" << t->runNumber << "\t";
    os << "eventNum=" << t->eventNumber << "\t";
    os << "vertex=" << t->vertex;

    os << t->lepton1;
    os << t->lepton2;

    ppJet ppjet;
    pptr iterator;

    for (ppjet=t->event_to_jet_set.First(iterator);
     t->event_to_jet_set.More(iterator);
     ppjet=t->event_to_jet_set.Next(iterator) ) {
      os << ppjet;
      os << endl;
    }
    return os;
}

ostream& operator << (ostream& os, ppLepton& l) {
    os << "\nLepton: \n";
    os << "p0=" << l->p[0] << "\t";
    os << "p1=" << l->p[1] << "\t";
    os << "p2=" << l->p[2] << "\t";
    os << "p3=" << l->p[3] << "\t";
    os << "charge=" << l->charge;
    return os;
}

ostream& operator << (ostream& os, ppJet& l) {
    os << "\nJet: \n";
    os << "jet_number=" << l->jet_num << "\t";
    os << "ntrk=" << l->ntrk << "\t\t";
    os << "q_frac=" << l->q_frac << "\t";
    os << "phi_jet=" << l->phi_jet << "\n";
    os << "eta_jet=" << l->eta_jet << "\t";
    os << "pt=" << l->pt << "\t";
    os << "mass_jet=" << l->mass_jet;
    return os;
}

Figure 20: The methods for the third version of the eventstore (continued).
```c
#include <string.h>
#include <stdio.h>
#include "ptool64.h"
#include "pset64.h"
#include "event.h"

void main(int argc, char **argv) {
    if (argc<2) {cout <<"USAGE : pop store_name
"; exit(0);}

    store a(argv[1]);

    for (int i=0; i<10; i++) {
        pptr tmp;
        new(&a, &tmp) Event(randomPopulation, &a);
        a.add(tmp);
    }

    a.close();
}
```

Figure 21: An eventstore containing 10 persistent events is populated using ptool64. Notice that a persistent 64 bit pointer to each event is returned through an argument to new.

```c
#include <stdio.h>
#include "ptool64.h"
#include "pset64.h"
#include "event.h"

void main(int argc, char **argv) {
    if (argc<2) {cout <<"USAGE : acc store_name
"; exit(0);}

    store b(argv[1]);

    ppEvent ppEv;
    for (ppEv=b.first(); b.more(); ppEv=b.next()) {
        cout << ppEv;
    }

    b.close();
}
```

Figure 22: In this figure, ptool64 is used to loop through an eventstore of persistent events.
References


