Table of Contents

Data Containers for Track Discussions

Introduction

This page collects together some random thoughts on the various data containers in use in LHCb, as input to further discussions.

Current Containers

Standard Transient Event Store (TES) Containers

The transient event store is not itself a container, but a mechanism for managing data. Data created by one algorithm can be stored on the TES (registered at locations that look a lot like standard unix paths) and then retrieved by other algorithms. Any class can be put on the TES, the only requirement is it must inherit from the DataObject base class. The TES is the standard Gaudi mechanism for managing data. Objects to be given to the TES must be created with new and are passed to the TES via pointer. At the end of each even the standard TES deletes all objects it contains, ready for the next event.

The process of registering and retrieving data from the TES has a small cpu overhead (although in practise it is extremely small, negligible in most cases).

Various standard containers are provided by Gaudi for storing containers of objects (data objects, tracks, clusters etc.) on the TES.

This is the lowest level container in Gaudi. It is designed to contain a list of pointers to data objects. Once an object has been inserted into this container the container owns the object. For instance when the TES deletes an ObjectVector, the container in turn deletes all objects it contains.

The ObjectVector class can be used to store any data object, the only requirement is that it must inherit from the ContainedObject base class.

The KeyedContainer is an extension to the ObjectVector container, which adds a unique key to each object that is stored. This key can then be used as a unique label for that object in the container, for direct random access etc. The key for a given object remains valid even if objects are added or removed from the container. Objects in a KeyedContainer must inherit from the KeyedObject base class.

Consequences of the TES

The most obvious feature of the TES is it's design is based around the use of pointers to data, rather than for instance references to object. This is in fact a key design choice in Gaudi itself. One consequence of this is data objects need to be created on the heap, via `operator new`, and ultimately are removed via `operator delete`. This feature then leads to a long winded discussion of the speed (or lack of) of code that make use use `new` and `delete` a lot. It is true that in early implementations of the L1/HLT tracking this was an important consideration and something needed to be done.

This lead to the development of various schemes to either work around or try and fix the speed problems.

Fast Containers

The first solution was to effectively abandon the use of `new/delete` entirely and to create some fast data containers. The PatDataStore is an example of this approach.

This is a tool used within the Pat tracking framework. It is a Gaudi tool that provides access to certain Pat data containers. This containers contain pointers to objects that are owned by the container. The primary difference between these Pat containers and a standard TES container are

- The containers themselves are not deleted at the end of each event.
- Objects in the containers are not deleted by the Pat containers at the end of each event. Instead at the start of each event the container is 'reset'. After these the objects in the container are still there, but flagged as ready to be reused in the next event.

Pros :-

- The approach used by the PatDataStore is clearly fast.

Cons :-

- The Pat containers have a rather non standard interface. Firstly the user must explicitly reset the container each event, as this is not done for them by the Gaudi framework (note that the TransientFastContainer in the next section solves this problem).
- The user is forced to explicitly call the `setXXX` for each data member of the class, when the object is being reused. The class constructors cannot be used.
- IMHO a bad feature is that (unless I am mistaken, this is based on observation of the code and not a real test, which is to be done) the objects retain knowledge of the previous event. If the user does not call the setXXX method for some data member then the value will NOT be the default value for the class, but the value in the last event... This I believe is a bug waiting to happen.

A more recent development inspired by the PatDataStore containers is the Gaudi TransientFastContainer. This is an official Gaudi container with its own Gaudi FastContainersSvc.

Whilst the methods are differently named, this container is effectively just the PatDataStore containers. Thus, it shares most of the the pros and cons with that container. One noticeable acception though is the FastContainersSvc clears each vector itself at the start of each event, so it is not necessary for the user to do this.

Overload new and delete

The other approach is to address directly the root of the problem and to speed up `new` and `delete`. This can be done since in C++ these are methods like any other, and thus can be overloaded for any class to do whatever you want.

This is effectively the approach I believe used in the Tsa world, where no private memory management is used but simply that provided by the Gaudi framework.

Note that the topic of overloading new and delete is a very well know and discussed problem. Most C++ books will mention it at some point or a quick google search will lead you to many many pages discussing the issue. In addition there exists many well tested and proven solutions to speed things up, such as the `memory pool`. In many ways this is similar to the solution adopted by the PatDataStore, in that a pool of memory is reserved and when `new` is called the system simply takes some memory from the pool. When the object is finally 'deleted' the memory is simply given back to the pool.

In the most recent versions of GOD, all event model classes have been using a memory pool implementation from boost to overload `new` and `delete`.

pros :-

- Solution is totally transparent to users. No changes to user code are needed to benefit and no custom data management tools to maintain.
- Very minimal maintenance, since well prooven reliable solutions already exist in STL and boost.

cons :-

- The benefit from pool based new and delete overloads are primarily for 'small' objects. As the object size increases the difference in speed between this and the standard ::new and :: delete operators rapidly decreases. It is with noting that any event model object must inherit from at least ContainedObject. This alone means the class is not really that small.
- Is it as fast as PatDataStore ?

For really small objects (i.e. a single POD data member) another fast solution exists, the FastClusterContainer. This container uses tricks with unions to create a container with very fast insertion time.

pros :-

- Apparently (I have not used them myself) very fast.
- Interface is very STL like.
- Available in standard TES
- No need to clear by-hand each event

cons :-

- None I can see really... Although I need to think a little more about the implementation. (In particular this container might benefit from pool based allocators - see below).

Other Possible Ideas

The above sections outline the main containers in use right now. If I have missed any please let me know or update this page yourself.

In this section other possible ideas we could consider are discussed. Please fell free to add your own.

Improvements to

The PatDataStore provides a proven fast way of managing data objects. It does though have in my opinion problems with its user interface. The primary one being the inability to use a proper constructor and the potential leakage of data between events. However, it should be possible to improve upon this I think. One possible example is the ROOT TClonesArray class (thanks to Matt for pointing it out) which implements a similar idea but assigns objects to particular memory locations using the placement new method, and is something we could look at (NOTE : I am NOT suggesting using the ROOT TClonesArray directly...)

Custom allocators

One area people tend to forget when using STL containers are the allocators, the object which define how objects are added to the container. Since containers like vector are dynamically sized, they need to be able to request more resources as they grow. The allocator that is used by default is std::allocator e.g. When a user creates a vector of type `std::vector< TYPE >` they are actually creating a `std::vector< TYPE, std::allocator< TYPE > >`. This default allocator simple uses the default `::new` and `::delete` methods to assign memory, so whilst the user might not realize it, they are still using `new` and `delete`.

Newer version of GCC contain possible alternatives to this, for instance a pool based allocator that follows a similar principle to that used in the event model overloading of new and delete. See for example here for more details.

Ideas for how to proceed

It is clear that there are many different implementations of containers that could be used, either already existing or improvements on the current containers. It is also clear that we need a fast solution, the benchmark being set by the current Pat packages. It is also my opinion (you may disagree) that the Pat containers have problems that we should avoid.

It is also clear that a proper comparison of the speed of all methods is needed before further conclusions can really be drawn. By proper I mean using realistic data objects in realistic usage patterns. This can be done either in gaudi itself, or perhaps more conveniently in some standalone test programs. I volunteer to continue looking into this...

Finally, it is really a little premature to consider in too much detail the containers before we have a proper idea on the data objects themselves.

Some hard numbers

Some interesting results from profiling different container implementations can be found here.

ChrisRJones - 02 Apr 2007

This topic: LHCb > ATeamDataContainers
Topic revision: r8 - 2007-05-09 - KurtRinnert