Subject: [FIXED] FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Fri, 18 Jan 2013 22:03:41 GMT
View Forum Message <> Reply to Message

I am a bit confused about the question whether object ownership of the FairTimeStamp instance passed to FairWriteoutBuffer::FillNewData is transferred to the buffer or not.

I had a look at the implementation of the buffers in the SDS package and found that the digis therein are allocated on the heap but never freed. Therefore I assumed that the ownership has been transferred to the buffer and searched for the corresponding deallocation in the implementation of FairWriteoutBuffer+derived. Actually there is none. Finally, I've ran the TimeBasedSimulation macro in the macros/mvd folder and counted the FairTimeStamp instances at runtime - the number was increasing continuously, as expected.

But if the buffer is not freeing the digits, how can the user know if a digit can be safely deallocated? Wouldn't it make more sense to delete the digits after they have been written to the TClonesArray?

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Sat, 26 Jan 2013 13:31:37 GMT
View Forum Message <> Reply to Message

Another point I do not quite understand is why what you call "active time" is fixed to -1 in the call to Modify (at FairWriteoutBuffer.cxx:186)

std::vector<std::pair<double, FairTimeStamp*> > modifiedData = Modify(std::pair<double, FairTimeStamp*>(currentdeadtime, oldData), std::pair<double, FairTimeStamp*>(-1, data));

In case I do not combine the signals given by both arguments but just use a dead time (aka active time) to emulate occupancy, I need the active time of the new hit in Modify - but it is replaced by -1.

Seems fishy to me, but maybe I just don't understand the deeper meaning of this mysterious number. Looking at the code of the SDS package doesn't clarify things either:

PndSdsDigiStripWriteoutBuffer.cxx:32

```
std::vector<std::pair<double, PndSdsDigiStrip*> >
PndSdsDigiStripWriteoutBuffer::Modify(std::pair<double, PndSdsDigiStrip*> oldData,
std::pair<double, PndSdsDigiStrip*> newData)
{
	std::vector<std::pair<double, PndSdsDigiStrip*> > result;
	std::pair<double, PndSdsDigiStrip*> singleResult;
```

```
if (newData.first > 0)
      singleResult.first = oldData.first + newData.first;
singleResult.second = oldData.second;
```

The if statement makes no sense because newData.first is always -1 as shown above (I don't find any other call to Modify either). The same if statement can be found in PndSdsDigiPixelWriteoutBuffer.cxx:31 and in the Gem package.

Seems to be inconsistent - what do I miss here?

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Mohammad Al-Turany on Sat, 26 Jan 2013 15:33:32 GMT
View Forum Message <> Reply to Message

Hi,

Sorry for the delay in answering this mail, in any case objects added to TClonesArray are added via new with placement, so you do not call the new and delete in the traditional way (That is why it is much faster than STL or TObjectArray). If the objects added to the TClonesArray do not allocate memory, it is enough to call the clear of the array to free the memory, other wise (e.g: your object has a TString or any other object inside it) then you need to call delete.

Normally each Task call the delete (Clear) of the TClonesArray in the finish event method.

Now for the Time based simulation it is more complicated because we use the TClonesArray::AbsorbObjects method which simply move objects from one array to the other, and it really move and not copy them. Now after moving and writing the objects a method FairRootManager::DeleteOldWriteoutBufferData is called to free the memory in the Buffer. In the Task the memory is freed by the Finishevent. in Other words, we have three TClonesArray:

1. The one used internally to read objects from tree
2. The one in the Buffer
3. The one in the task, which is connected to the output tree

Objects are moved from one array to the other by calling TClonesArray::AbsorbObjects, we clean the buffer from the rest and the Task has to clean what it write to the output tree.

regards,

Mohammad

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Sat, 26 Jan 2013 16:39:18 GMT

Thank you for this in-depth answer! But I think that the problem is not related to memory-pooling - it happens outside of the pool (TClonesArray):

The 'digits' are usually created on the heap and not in a pool. They are passed as pointers to FairWriteoutBuffer, which stores the pointers in maps. When the data has to be written out, the relating portion of digits is *copied* to a pool via AddNewDataToTClonesArray:

```
void PndSdsDigiStripWriteoutBuffer::AddNewDataToTClonesArray(FairTimeStamp* data)
{
  FairRootManager* ioman = FairRootManager::Instance();
  TClonesArray* myArray = ioman->GetTClonesArray(fBranchName);
  if (fVerbose > 1) std::cout << "Data Inserted: "  <<  *(PndSdsDigiStrip*)(data) << std::endl;
  new ((*myArray)[myArray->GetEntries()]) PndSdsDigiStrip(*(PndSdsDigiStrip*)(data));
}
```

At least I read this as the invocation of a copy-constructor. So the original object on the heap - which was initially passed to the buffer - is still alive and will at some point go out of scope -> memory leak.

For the creation of the objects, see for example PndSdsHybridHitProducer.cxx:370:

```
PndSdsDigiPixel *tempPixel = new PndSdsDigiPixel( fPixelList[iPix].GetMCIndex(),
fInBranchId, fPixelList[iPix].GetSensorID() ,fPixelList[iPix].GetFE(),
                                        fPixelList[iPix].GetCol(), fPixelList[iPix].GetRow(),
                                        smearedCharge, correctedTimeStamp);
//fChargeConverter->GetTimeStamp(point->GetTime(),
charge,fEventHeader->GetEventTime()) );

    if (fTimeOrderedDigi){
            tempPixel->ResetLinks();
            std::vector<int> indices = fPixelList[iPix].GetMCIndex();
            FairEventHeader* evtHeader =
(FairEventHeader*)FairRootManager::Instance()->GetObject("EventHeader.");
            for (int i = 0; i < indices.size(); i++){
              tempPixel->AddLink(FairLink(evtHeader->GetInputFileId(),
evtHeader->GetMCEntryNumber(),  fInBranchId, indices[i]));
            }
            tempPixel->AddLink(FairLink(-1, fEventNr, "EventHeader.", -1));
    }
    fDataBuffer->FillNewData(tempPixel,
fChargeConverter->ChargeToDigiValue(fPixelList[iPix].GetCharge())*6 + EventTime,
point->GetTime()+EventTime);
```

I'd say FairWriteoutBuffer should either delete the digits on the heap after copying them to the TClonesArray (transfer of ownership by policy) or use one of the popular shared_ptr implementations.

As I said, the number of allocated FairTimestamp derived instances was increasing steadily during a digitization run while they were correctly freed in the simulation. This can be checked.

Kind regards,
Oliver

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Mohammad Al-Turany on Sat, 26 Jan 2013 18:37:59 GMT
View Forum Message <> Reply to Message

Hallo Oliver,

Yes, it is true. Thanks for pointing this out. I think the SDS people should correct that, and hopefully all the people who copied this code also!

regards,

Mohammad

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Sat, 26 Jan 2013 21:00:06 GMT
View Forum Message <> Reply to Message

Ok, let's see if I understand you correctly:

The policy is:
The class which derives from FairWriteoutBuffer has to do the housekeeping and is responsible for deleting the digits which are used by the underlying FairWriteoutBuffer implementation. It is guaranteed that a digit can be safely deallocated after it was written to the TClonesArray by AddNewDataToTClonesArray.

I guess this is correct? It would also make sense to me that the base class FairWriteoutBuffer itself deletes the digits - this caused my initial confusion. But your answer implies that the memleak is not considered to be a bug in FairWriteoutBuffer but in the derived class.

I'm happy either way - and because these things are not obvious it would be nice if someone adds a comment about this in FairWriteoutBuffer.h some day

Thanks,
Oliver

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory

leak)
Posted by Tobias Stockmanns on Mon, 28 Jan 2013 13:02:12 GMT
View Forum Message <> Reply to Message

Dear all,

sorry that I did not participate in the discussion so far.

I will have a look into it in the next few days and see if one could add an automatic deletion mechanism into the base classes.

Cheers,

Tobias

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Tobias Stockmanns on Wed, 30 Jan 2013 16:43:20 GMT
View Forum Message <> Reply to Message

Dear Olliver,

thank you for pointing out the memory leak. You are right this is a bug.

I fixed it now in a way that the data is deleted inside the FairWriteoutBuffer. No additional delete is necessary in your code.

This is not the most elegent way. I would have prefered to leave the deletion of the data to the one who created it but the I would have to call the constructor of the data to create a copy which I cannot do in a base class.

Cheers,

Tobias

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Thu, 31 Jan 2013 13:24:56 GMT
View Forum Message <> Reply to Message

Thank you, Tobias.

Just as an opinion: the most elegant way - from the beginning - would have been to design FairWriteoutBuffer as a template FairWriteoutBuffer<t_digit, t_modify_functor > which is derived from base FairWriteoutBufferBase (<=> your current FairWriteoutBuffer). This way the compiler would autogenerate the boilerplate code which users currently have to implement. The only class the user would have to add is a custom modify functor in case he needs one.

The digit at the user side would be allocated on the stack and passed as const t_digit & (-> no

new, no leak). This would decouple user and framework code so that you never run into these ownership problems.

Nevertheless, it doesn't hurt much to add the classes by hand. I mean ... you don't want to sell this package and scientists aren't too picky when it comes to copy and paste

---

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Tobias Stockmanns on Thu, 31 Jan 2013 14:36:46 GMT
View Forum Message <> Reply to Message

Dear Oliver,

thank you for your suggestion.

This is exactly what I did when I started the project. The problem is the way root handles templates which leads to dependencies between all classes which use the FairWriteoutBuffer. Therefore this solution was ruled out and I had to come up with the inheritance scheme.

Cheers,

Tobias

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Oliver Merle on Thu, 31 Jan 2013 23:58:23 GMT
View Forum Message <> Reply to Message

Just because you mentioned that you would find it more elegant to copy the data via the base class: TObject::Clone() does actually call the constructor of the derived class via the base class.

I would naively expect that FairTimeStamp * my_copy = (FairTimeStamp*)digit->Clone() should do the trick for any derived digit class, but I'm no ROOTer and maybe I am not aware of any pitfalls in the ROOT RTTI (like that template thing).

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Tobias Stockmanns on Fri, 01 Feb 2013 09:08:30 GMT
View Forum Message <> Reply to Message

Again you are right. Yesterday I uploaded a new version of FairWriteoutBuffer which is exactly doing this.

Cheers,

Tobias

Subject: Re: FairWriteoutBuffer::FillNewData and object ownership (memory leak)
Posted by Philipp Mahlberg on Mon, 25 Feb 2013 15:46:22 GMT
View Forum Message <> Reply to Message

Two additional notes concerning the copy creation and deletion of the FairTimeStamp objects in the timebased simulation.

1. From my point of view, the default FairTimeStamp::Modify function still causes a memory leak, as the newly incoming and further on ignored object is not deleted.

2. I am working on a kind of extended timebased simulation for the emc, where I pass interim objects (keeping track of all the needed information) to the event mixing buffers and construct a corresponding Emc waveform right before filling the data to the TClonesArray.

Therefore the interim object contains a pointer to the waveform generator, which will take care of the waveform building process. Since the TObject::Clone() copies the object via streaming, ROOT is constructing new generator objects etc. as well. A simple copy of the address would be sufficient, but the method here creates lots of (useless) objects and brings up a runtime error (ROOT complains about a missing default constructor of an abstract class involved). The latter should be fixed, but I think it is not the main problem though)

As mentioned before, the objects represent an interim state and are not primarily designed to be written into a file, what is done with the final waveforms.

At the moment, I refrain from getting an extra copy of the object by overriding the FillNewData Method.
One could also think of writing a customary streamer which then hopefully just copies the reference address stored in that specific pointer (I am not an expert on this at all, so I don't know weather it will work). But then I could never make use of all of ROOT streaming power to write an object with all its dependencies to file, e.g. for debug reasons. Perhaps it is also possible to distinguish between writing to a file and cloning the object in the streaming function?

Maybe somebody even knows a smarter way how to overcome the problem....
