GammaLib - Bug #516

GModels container not updated appropriately

09/20/2012 06:47 PM - Knödlseder Jürgen

Status:	Closed	**Start date:**	09/20/2012
Priority:	Urgent	**Due date:**	
Assigned To:	Knödlseder Jürgen	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	00-07-00		

Description

In the following example, a point source model was appended to the GModels container, and later the point source model was replaced by an extended model. However, the container still returns a pointer to a point source model, although the content corresponds clearly to the extended source model.

```
>>> from gammalib import *
>>> m=GModels()
>>> p=GModelPointSource()
>>> p.name("Point source")
>>> e=GModelExtendedSource()
>>> e.name("Extended source")
>>> m.append(p)
>>> print m
=== GModels ===
 Number of models ..........: 1
 Number of parameters ......: 0
=== GModelPointSource ===
 Name ......................: Point source
 Instruments ...............: all
 Model type ................:
 Number of parameters ......: 0
 Number of spatial par's ...: 0
 Number of spectral par's ..: 0
 Number of temporal par's ..: 0
>>> m[0]=e
>>> print m
=== GModels ===
 Number of models ..........: 1
 Number of parameters ......: 0
=== GModelPointSource ===
 Name ......................: Extended source
 Instruments ...............: all
 Model type ................:
 Number of parameters ......: 0
 Number of spatial par's ...: 0
 Number of spectral par's ..: 0
 Number of temporal par's ..: 0
>>> print type(m[0])
<class 'gammalib.model.GModelPointSource'>
>>> print m[0].type()
PointSource
```

This seems not to be a problem of type casting at the moment of reading, as the object returns a type of "PointSource".

Interestingly, the pointer has however be changed:

```
>>> print m[0].__repr__()
<gammalib.model.GModelPointSource; proxy of <Swig Object of type 'GModelPointSource *' at 0x101a58300> >
>>> m[0]=e
```

```
>>> print m[0].__repr__()
<gammalib.model.GModelPointSource; proxy of <Swig Object of type 'GModelPointSource *' at 0x101a583f0> >
```

Possible, this indicates a problem in the *setitem* method. Some debugging is needed here.

History

#1 - 09/20/2012 10:45 PM - Knödlseder Jürgen

The problem is related to the design of the GModel class and it's derived classes. This is illustrated by the following code example:

```
#include <string>
#include <iostream>

class GModel {
public:
    GModel(void) {}
    virtual ~GModel(void) {}
    virtual std::string type(void) const = 0;
    virtual std::string name(void) const { return m_name; }
    virtual GModel& operator=(const GModel& model)
    {
        if (this != &model) {
            std::cout << "GModel& operator=(const GModel& model)" << std::endl;
            m_name = model.m_name;
        }
    }
    std::string m_name;
};

class GSkyModel : public GModel {
public:
    GSkyModel(void) { m_name="sky"; }
    virtual ~GSkyModel(void) {}
    virtual std::string type(void) const { return "sky"; }
};

class GDataModel : public GModel {
public:
    GDataModel(void) { m_name="data"; }
    virtual ~GDataModel(void) {}
    virtual std::string type(void) const { return "data"; }
};

int main(void)
{
    GModel& sky  = *(new GSkyModel);
    GModel& data = *(new GDataModel);
    std::cout << sky.type() << ":" << sky.name() << ": " << &sky << std::endl;
    if (dynamic_cast<GSkyModel*>(&sky) != NULL)
        std::cout << "is GSkyModel" << std::endl;
    else
        std::cout << "is GDataModel" << std::endl;
    sky = data;
    std::cout << sky.type() << ":" << sky.name() << ": " << &sky << std::endl;
    if (dynamic_cast<GSkyModel*>(&sky) != NULL)
        std::cout << "is GSkyModel" << std::endl;
    else
        std::cout << "is GDataModel" << std::endl;
    return 0;
}
```

This code outputs the following:

```
sky:sky: 0x100100080
is GSkyModel
GModel& operator=(const GModel& model)
sky:data: 0x100100080
is GSkyModel
```

The assignment in line 43 just copy the members of GDataModel to GSkyModel, but the sky object remains of course of type GSkyModel.

That's exactly what happens in the GammaLib classes. When the GModels::operator[] is called, a reference to GModel is returned. If then operator= is used to assign one model to the reference, only the members are copied, but the target reference remains the same. In the above example (that captures the essential elements of the GammaLib implementation), the sky reference remains always a reference to a GSkyModel object.

#2 - 09/20/2012 11:05 PM - Knödlseder Jürgen

A implementation of the main routine that has the expected behavior would be:

```
int main(void)
{
   GModel* sky  = new GSkyModel;
   GModel* data = new GDataModel;
   std::cout << sky->type() << ":" << sky->name() << ": " << sky << std::endl;
   if (dynamic_cast<GSkyModel*>(sky) != NULL)
      std::cout << "is GSkyModel" << std::endl;
   else
      std::cout << "is GDataModel" << std::endl;
   delete sky;
   sky = data;
   std::cout << sky->type() << ":" << sky->name() << ": " << sky << std::endl;
   if (dynamic_cast<GSkyModel*>(sky) != NULL)
      std::cout << "is GSkyModel" << std::endl;
   else
      std::cout << "is GDataModel" << std::endl;
   return 0;
}
```

which results in

```
sky:sky: 0x100100080
is GSkyModel
data:data: 0x1001000b0
is GDataModel
```

The essential difference here is that it handles pointers, not references. The assignment is from one pointer to another. But it requires a delete operation.

I guess it comes down to the question of whether assignments should be allowed between derived classes. Here a discussion on this topic on the web:
http://stackoverflow.com/questions/10868234/how-do-i-prevent-cross-assignment-between-2-classes-derived-from-string

Possible solutions seem to be:

- use explicit to prevent conversion
- declare private constructors and assignment operators for all the the types that one wants to prohibit direct copying from

#3 - 09/20/2012 11:05 PM - Knödlseder Jürgen

- % Done changed from 0 to 20

#4 - 09/21/2012 02:47 AM - Knödlseder Jürgen

It is tempting to replace the reference operator by a pointer operator, yet only the reference operator allows assignment.

On the other hand, it's the assignment that poses problems, so can we really use a reference operator for save assignment here? I would love to code an assignement such as

```
GModels m;
GModelPointSource p;
...
m[0] = p;
```

but can this really be done?

The question is: how can we implement model assignment to the model container class? In Python this is easy as there is the __setitem__ function. But in C++ we have to use a reference operator. The essential method is then the GModel::operator= method that executes the assignment. But this method gives no handle about the manipulation of derived classes. So we're stuck with our assignment operator.

Note that an assignment from a derived class to a base class is not safe, the extra bit from the derived class just get sliced off:

- http://stackoverflow.com/questions/3743093/assign-derived-class-to-base-class
- http://stackoverflow.com/questions/274626/what-is-the-slicing-problem-in-c
- http://en.wikipedia.org/wiki/Object_slicing

The slicing problem is serious because it can result in memory corruption, and it is very difficult to guarantee a program does not suffer from it. To design it out of the language, classes that support inheritance should be accessible by reference only (not by value).

#5 - 09/21/2012 03:08 AM - Knödlseder Jürgen

Got the solution. We need a reference to pointer operator. See the code below:

```
#include <string>
#include <vector>
#include <iostream>

class GModel {
public:
    GModel(void) {}
    virtual ~GModel(void) {}
    virtual std::string type(void) const = 0;
    virtual GModel* clone(void) const = 0;
    virtual std::string name(void) const { return m_name; }
```

```
    virtual GModel& operator=(const GModel& model)
    {
        if (this != &model) {
            std::cout << "GModel& operator=(const GModel& model)" << std::endl;
            m_name = model.m_name;
        }
    }
    std::string m_name;
};

class GModels {
public:
    GModels(void) {}
    virtual ~GModels(void) {}
    virtual void append(const GModel& model) { m_models.push_back(model.clone()); }
    GModel*& operator[](int index) { return (m_models[index]); }
    std::vector<GModel*> m_models;
};

class GSkyModel : public GModel {
public:
    GSkyModel(void) { m_name="sky"; }
    virtual ~GSkyModel(void) {}
    virtual std::string type(void) const { return "sky"; }
    virtual GSkyModel* clone(void) const { return new GSkyModel(*this); }
};

class GDataModel : public GModel {
public:
    GDataModel(void) { m_name="data"; }
    virtual ~GDataModel(void) {}
    virtual std::string type(void) const { return "data"; }
    virtual GDataModel* clone(void) const { return new GDataModel(*this); }
};

int main(void)
{
    GModels    models;
    GSkyModel  sky;
    GDataModel data;
    models.append(sky);
    std::cout << models[0]->type() << ":" << models[0]->name() << std::endl;
    models[0] = &data;
    std::cout << models[0]->type() << ":" << models[0]->name() << std::endl;

    return 0;
}
```

In line 27 there is the reference to pointer operator. Instead of returning a reference to a value it provides a reference to a pointer. So we assign pointers, no copy of data is made.

In line 54 the data object is now assigned. It's what I want: being able to assign a model. I just have to specify the reference here instead of the value. This makes clear that nothing is really copied, and if we want indeed copy the data object, we would have to use the clone method, i.e.

```
models[0] = data.clone();
```

For the record, the output of this program is

```
sky:sky
data:data
```

#6 - 09/21/2012 03:24 AM - Knödlseder Jürgen

- Status changed from New to Feedback

- % Done changed from 20 to 100

I now implemented the reference to pointer logic in the GModels::operator[] access operator - with success. This can be seen in the following Python test dump:

```
>>> from gammalib import *
>>> m=GModels()
>>> p=GModelPointSource()
>>> p.name("Point source")
>>> e=GModelExtendedSource()
>>> e.name("Extended source")
>>> m.append(p)
>>> print m
=== GModels ===
 Number of models ..........: 1
 Number of parameters ......: 0
=== GModelPointSource ===
 Name ......................: Point source
 Instruments ...............: all
 Model type ................:
 Number of parameters ......: 0
 Number of spatial par's ...: 0
 Number of spectral par's ..: 0
 Number of temporal par's ..: 0
>>> m[0]=e
>>> print m
=== GModels ===
 Number of models ..........: 1
 Number of parameters ......: 0
=== GModelExtendedSource ===
 Name ......................: Extended source
 Instruments ...............: all
 Model type ................:
 Number of parameters ......: 0
 Number of spatial par's ...: 0
 Number of spectral par's ..: 0
 Number of temporal par's ..: 0
>>> print type(m[0])
<class 'gammalib.model.GModelExtendedSource'>
>>> print m[0].type()
ExtendedSource
```

#7 - 09/21/2012 03:32 AM - Knödlseder Jürgen

Just for the record:

I implemented the Python assignment operator as a deep copy of the class. This is to make the assignment analogous to the append() method. Otherwise we would always rely on the fact that the original object does still exist. This is also analogous to the way how Python lists work;

```
>>> a=[]
>>> a.append(78)
>>> print a
[78]
>>> b=13
>>> a[0]=b
>>> print a
[13]
>>> del b
>>> print a
[13]
```

To show that the same behavior is implemented in GModels, continuing the above example we get

```
>>> del e
>>> print m
=== GModels ===
 Number of models .....: 1
 Number of parameters .....: 0
=== GModelExtendedSource ===
 Name .....: Extended source
 Instruments .....: all
 Model type .....:
 Number of parameters .....: 0
 Number of spatial par's ...: 0
 Number of spectral par's ..: 0
 Number of temporal par's ..: 0
```

#8 - 09/21/2012 09:22 AM - Knödlseder Jürgen

Although the above solution is possible and close to the desiderate, it is subject to memory leaks. Assigning a pointer to a preexisting memory cell will overwrite the pointer that lived there before, and the link to the respective memory will be lost.

In conclusion, **it is not possible to write a save method for setting values of a container class holding pointers through an access operator.**

In summary, here the problems that occur:

- accessing the pointers through a reference leads to slicing
- accessing the pointers through a reference to a pointer leads to memory leaks

Thus, a specific set method needs to be implemented for setting of the pointers. This should solve by the way Bug #515.

A section will be added to the coding & design standards document, explaining how this issue should be solved.

#9 - 12/01/2012 09:52 PM - Knödlseder Jürgen

- Status changed from Feedback to Closed