{{lastupdated_at}} by {{lastupdated_by}}

GContainer

Usage

The abstract GContainer class is used to defined - as much as possible - the common interface for container classes. As the exact type of object varies from container class to container class, not all methods can be defined at the level of the GContainer class. See action #767 for the reasons behind this.

Base class

GContainer is the interface class for container classes in GammaLib. The class is an abstract base class which defines the methods that all GammaLib container classes are required to implement. It derives from the GBase base class. Below the interface definition:

```
class GContainer : public GBase {
public:
   virtual ~GContainer(void) {}
   virtual int  size(void) const = 0;
   virtual bool isempty(void) const = 0;
   virtual void remove(const int& index) = 0;
   virtual void reserve(const int& num) = 0;
};
```

The methods have the following meaning: * size() returns the number of elements in the container * isempty() checks whether the container is empty (contains 0 elements) * remove() removes object with specified index from the container * reserve() reserves space for num objects in the container

The reserve() method is particularly useful for containers that use the std::vector class for implementing the object list.

Object containers

Object container are containers that hold a list of instances of a given class. In addition to the methods required by GContainer, the following methods shall be implemented by object containers:

```
GObject&       operator[](const int& index);
const GObject& operator[](const int& index) const;
void           append(const GObject& object);
void           insert(const int& index, const GObject& object);
void           extend(const GContainer& container);
```

The operator[] of object containers returns references to the objects. This allows accessing and setting of objects in an object container.

The append() method appends an object at the end of the container, while the insert() method inserts an object **before** a specific index (similar to std::vector). Object containers will always clone the object that should be appended or inserted, hence object containers will hold deep copies of the object provided in the argument.

The extend() method extends the container with the content of another container.

Pointer containers

Pointer containers are objects that hold a list of pointers to instances of a given class. One can distinguish two types of pointer containers: those that have their own internal memory management and those that simply store pointer which will be managed outside the class.

Pointer containers with internal memory management

Pointer containers with internal memory management deal with the allocation and deallocation of container elements. These containers will always clone the object that should be appended or inserted, hence object containers will hold deep copies of the objects, similar to object containers. In particular, pointer containers should assure that all pointers it holds are valid.

Pointer containers with internal memory management will be used instead of object containers when the container should be able to hold different derived classes of a given base class. An example of such a container is the GModels class. Pointer containers with internal memory management shall implement in addition to the GContainer methods the following methods:

```
GObject*       operator[](const int& index);
const GObject* operator[](const int& index) const;
GObject*       set(const int& index, const GObject& object);
GObject*       append(const GObject& object);
GObject*       insert(const int& index, const GObject& object);
void           extend(const GContainer& container);
```

The operator[] of pointer containers allows to access the elements of the container. Note that operator[] of a pointer container with internal memory management **shall** always return a pointer, which explicitly prevents of assigning of elements using the operator[]. The reason behind this policy is that code slicing it thus avoided (see #516). Code slicing occurs in situations where you assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is "sliced" away.

To prevent code slicing, a set method shall be implemented that handles the element assignment. Note that all objects are passed by reference, hence there is no way that a NULL pointer can be assigned to a container element. The class **shall** assure that pointers are always valid. This implies that **pointers returned by the operator[] do not need to be checked against NULL values.**

For example, the set() method should use the object's clone() method to store a pointer to a deep copy of the object in the container (this is also a good illustration of the need for the clone() method):

```
GModel* GModels::set(const int& index, const GModel& model)
{
   // Compile option: raise exception if index is out of range
   #if defined(G_RANGE_CHECK)
   if (index < 0 || index >= size()) {
      throw GException::out_of_range(G_SET1, index, 0, size()-1);
   }
   #endif

   // Delete any existing model
   if (m_models[index] != NULL) delete m_models[index];

   // Assign new model by cloning
   m_models[index] = model.clone();

   // Set parameter pointers
   set_pointers();

   // Return pointer
   return m_models[index];
}
```

The set(), append() and insert() methods **shall** return a pointer to the object that has been cloned (see above). Very often this pointer may not be used, but having the pointer is convenient for manipulating the object after appending it to the container. Note that the returned pointer is non-const, so that the content of the object can indeed be manipulated.

Pointer containers with external memory management

Pointer containers with external memory management will not deal with the allocation and deallocation of container elements. An example of such a class is the GOptimizerPars class.

See the Change Request #689 that deals with a possible modification of the GXmlNode interface.

Further notes

Containers containing named objects should have a contains() method to check for existence of a specific named object in the container, e.g.

```
GModels models;
if (models.contains("Crab"))
    std::cout << "We have the Crab!";
```