GammaLib - Action #287

Feature # 119 (Closed): Perform automatic dynamic typecasting in Python interface

Investigate possible solutions for type casting in SWIG

07/08/2012 10:14 PM - Knödlseder Jürgen

Status:	Closed	**Start date:**	03/09/2012
Priority:	Normal	**Due date:**	
Assigned To:	Knödlseder Jürgen	**% Done:**	100%
Category:		**Estimated time:**	16.00 hours
Target version:	00-09-00		

Description

GammaLib uses extensively derived classes to implement specific instances of a general quantity. For example, GObservation defines a general observation, and GCTAObservation implements an observation with the CTA telescope.

When we now iterate in Python over observations (for example the observations that are stored in an GObservations container), we always see the base class type GObservation instead of the derived class type.

To circumvent this problem, specific functions for typecasting are implemented for now. This makes that Python code rather complex. We would like to investigate whether it is not possible to do automatic type casting in SWIG.

In this task we investigate possible solutions. For this purpose, create a simple base class and a derived class, and see whether you can tweak the swig interface file so that a pointer to the derived class is returned.

Related issues:

Blocks GammaLib - Action # 288: Implement solution for derived classes of GOb...	**Closed**	**03/09/2012**

History

#1 - 07/08/2012 10:33 PM - Knödlseder Jürgen

- File casting.py added

- Description updated

The file attachment:casting.py provides a small example that illustrates the type casting problem in GammaLib. The output of the file is

```
<class 'gammalib.obs.GObservation'>
<class 'gammalib.obs.GObservation'>
<class 'gammalib.obs.GObservation'>
```

but we would like to see

```
<class 'gammalib.obs.GLATObservation'>
<class 'gammalib.obs.GCTAObservation'>
<class 'gammalib.obs.GMWLObservation'>
```

#2 - 07/28/2012 12:53 AM - Knödlseder Jürgen

- Target version deleted (Stage Jean-Baptiste Cayrou)

#3 - 09/01/2012 03:52 AM - Knödlseder Jürgen

- Assigned To deleted (Anonymous)

#4 - 09/09/2014 11:25 PM - Knödlseder Jürgen

Here some interesting links:

- http://swig.10945.n7.nabble.com/Status-of-SWIG-TypeDynamicCast-td11400.html

Is this code useful?

```
+   {
+       swig_type_info *ty = SWIG_TypeDynamicCast(SWIGTYPE_p_IDOM_Node, (void **) &result);
+       ST(argvi) = sv_newmortal();
+       SWIG_MakePtr(ST(argvi++), (void *) result, ty);
+   }
```

#5 - 09/09/2014 11:31 PM - Knödlseder Jürgen

As example, adding

```
%import(module="gammalib.cta") "../inst/cta/pyext/cta.i";
```

to obs.i makes all CTA classes available to the obs module, i.e.

```
/* -------- TYPES TABLE (BEGIN) -------- */

#define SWIGTYPE_p_GBase swig_types[0]
#define SWIGTYPE_p_GCTAAeff swig_types[1]
#define SWIGTYPE_p_GCTAAeff2D swig_types[2]
#define SWIGTYPE_p_GCTAAeffArf swig_types[3]
#define SWIGTYPE_p_GCTAAeffPerfTable swig_types[4]
...
```

This could be done at the GammaLib configuration level, yet would not allow to compile external modules.

#6 - 09/10/2014 12:01 AM - Knödlseder Jürgen

Some information about type sharing from the swig manual:

15.3 The SWIG runtime code

Many of SWIG's target languages generate a set of functions commonly known as the "SWIG runtime." These functions are primarily related to the runtime type system which checks pointer types and performs other tasks such as proper casting of pointer values in C++. As a general rule, the statically typed target languages, such as Java, use the language's built in static type checking and have no need for a SWIG runtime. All the dynamically typed / interpreted languages rely on the SWIG runtime.

The runtime functions are private to each SWIG-generated module. That is, the runtime functions are declared with "static" linkage and are visible only to the wrapper functions defined in that module. The only problem with this approach is that when more than one SWIG module is used in the same application, those modules often need to share type information. This is especially true for C++ programs where SWIG must collect and share information about inheritance relationships that cross module boundaries.

To solve the problem of sharing information across modules, a pointer to the type information is stored in a global variable in the target language namespace. During module initialization, type information is loaded into the global data structure of type information from all modules.

There are a few trade offs with this approach. This type information is global across all SWIG modules loaded, and can cause type conflicts between modules that were not designed to work together. To solve this approach, the SWIG runtime code uses a define SWIG_TYPE_TABLE to provide a unique type table. This behavior can be enabled when compiling the generated _wrap.cxx or _wrap.c file by adding -DSWIG_TYPE_TABLE=myprojectname to the command line argument.

Then, only modules compiled with SWIG_TYPE_TABLE set to myprojectname will share type information. So if your project has three modules, all three should be compiled with -DSWIG_TYPE_TABLE=myprojectname, and then these three modules will share type information. But any other project's types will not interfere or clash with the types in your module.

Another issue relating to the global type table is thread safety. If two modules try and load at the same time, the type information can become corrupt. SWIG currently does not provide any locking, and if you use threads, you must make sure that modules are loaded serially. Be careful if you use threads and the automatic module loading that some scripting languages provide. One solution is to load all modules before spawning any threads, or use SWIG_TYPE_TABLE to separate type tables so they do not clash with each other.

Lastly, SWIG uses a #define SWIG_RUNTIME_VERSION, located in Lib/swigrun.swg and near the top of every generated module. This number gets incremented when the data structures change, so that SWIG modules generated with different versions can peacefully coexist. So the type structures are separated by the (SWIG_TYPE_TABLE, SWIG_RUNTIME_VERSION) pair, where by default SWIG_TYPE_TABLE is empty. Only modules compiled with the same pair will share type information.

#7 - 09/10/2014 01:13 AM - Knödlseder Jürgen

Some progress. The following code added to the _wrap_GObservations___getitem__ function in obs_wrap.cpp works:

```
 myinfo = SWIG_TypeDynamicCast(SWIGTYPE_p_GObservation, (void **)&result);
 printf("Mangled name ..........: %s\n", myinfo->name);
 printf("Human readable name ...: %s\n", myinfo->str);
 mycast = myinfo->cast;
 while (mycast != 0) {
   printf("Mangled name can cast ......: %s\n", mycast->type->name);
   printf("Human readable name can cast: %s\n", mycast->type->str);
   if (strcmp("_p_GCTAObservation", mycast->type->name) == 0) {
       //myinfo = SWIG_TypeDynamicCast(mycast->type, (void **)&result);
       myinfo = mycast->type;
   }
   mycast = mycast->next;
 }
 resultobj = SWIG_NewPointerObj(SWIG_as_voidptr(result), myinfo, 0 |  0 );
```

It gives:

```
>>> obs=gammalib.GObservations()
>>> run=gammalib.GCTAObservation()
>>> obs.append(run)
<gammalib.obs.GObservation; proxy of <Swig Object of type 'GObservation *' at 0x101fe3840> >
>>> type(obs[0])
Mangled name ..........: _p_GObservation
Human readable name ...: GObservation *
Mangled name can cast ......: _p_GCTAObservation
Human readable name can cast: GCTAObservation *
Mangled name can cast ......: _p_GCOMObservation
Human readable name can cast: GCOMObservation *
Mangled name can cast ......: _p_GObservation
Human readable name can cast: GObservation *
Mangled name can cast ......: _p_GLATObservation
Human readable name can cast: GLATObservation *
Mangled name can cast ......: _p_GMWLObservation
Human readable name can cast: GMWLObservation *
<class 'gammalib.cta.GCTAObservation'>
```

This means that I'm able to cast to GCTAObservation in the obs module without a need to import the CTA module. The swig_type_info structure contains in fact a linked list of all the derived classes.

An easy way to implement this would be to add a method to each derived class that returns the mangled swig name. Alternatively one could use typeid, but this is problematic since there is not standard response for this function.

#8 - 09/10/2014 01:52 AM - Knödlseder Jürgen

- Status changed from New to In Progress

- Assigned To set to Knödlseder Jürgen

- % Done changed from 0 to 30

Making progress on this issue, adding the following code to GObservations.i provides automatic typecasting without any need to no the actual derived classes at the actual moment. The code relies on constructing the SWIG mangled class names from the instrument name. In a future version, GObservation should have a type() method that returns the class name:

```
%typemap(out) GObservation* {
    // Build mangled name
    char classname[80];
    strcpy(classname, "_p_G");
    strcat(classname, result->instrument().c_str());
    strcat(classname, "Observation");

    // Get corresponding swig_type_info structure
    swig_type_info *myinfo = 0;
    swig_cast_info *mycast = 0;
    myinfo = SWIG_TypeDynamicCast(SWIGTYPE_p_GObservation, (void **)&result);
    printf("Mangled name ..........: %s\n", myinfo->name);
    printf("Human readable name ...: %s\n", myinfo->str);
    mycast = myinfo->cast;
    while (mycast != 0) {
        printf("Mangled name can cast ......: %s\n", mycast->type->name);
        printf("Human readable name can cast: %s\n", mycast->type->str);
        if (strcmp(classname, mycast->type->name) == 0) {
            myinfo = mycast->type;
        }
        mycast = mycast->next;
    }
    $result = SWIG_NewPointerObj(SWIG_as_voidptr($1), myinfo, 0 |  0);
}
```

#9 - 09/10/2014 02:25 AM - Knödlseder Jürgen

- % Done changed from 30 to 40

This code does the job, using the newly introduced type() method:

```
%typemap(out) GObservation* {
    char classname[80];
    strcpy(classname, "_p_");
    strcat(classname, result->type().c_str());
    swig_type_info *myinfo = SWIGTYPE_p_GObservation;
    swig_cast_info *mycast = 0;
    //printf("Mangled name ..........: %s\n", myinfo->name);
    //printf("Human readable name ...: %s\n", myinfo->str);
    mycast = myinfo->cast;
    while (mycast != 0) {
        //printf("Mangled name can cast ......: %s\n", mycast->type->name);
        //printf("Human readable name can cast: %s\n", mycast->type->str);
        if (strcmp(classname, mycast->type->name) == 0) {
            myinfo = mycast->type;
            break;
        }
        mycast = mycast->next;
    }
    $result = SWIG_NewPointerObj(SWIG_as_voidptr($1), myinfo, 0 |  0);
}
```

Now have to generalize this over all other derived classes.

#10 - 09/10/2014 12:14 PM - Knödlseder Jürgen

- Status changed from In Progress to Closed

- % Done changed from 40 to 100

- Remaining (hours) changed from 16.0 to 0.0

The investigation is considered to be finished, we now know how to achieve full automatic type casting.

#11 - 09/10/2014 12:15 PM - Knödlseder Jürgen

- Target version set to 00-09-00

Files

casting.py	1.86 KB	07/08/2012	Knödlseder Jürgen