Table of Contents

How-To Migrate to Gaudi v21

Introduction

This twiki page is meant to provide instructions and, to some extent, the documentation for the changes between Gaudi v19/v20 and Gaudi v21.

Interfaces

Rationale

To extend a base component with the implementation of new interfaces, we have to provide an implementation of the method `IInterface::queryInterface`, which is almost always the same. A typical case looks like:

```
class MySpecialization: public MyBase, virtual public IMyFeature {
public:
  /// Constructor
  MySpecialization();
  /// Destructor
  virtual ~MySpecialization();
  /// Implementation from IInterface
  virtual StatusCode queryInterface(const InterfaceID &riid, void **ppvInterface);
  /// Implementation from IMyFeature
  virtual StatusCode myMethod();
};
MySpecialization::MySpecialization():MyBase() {
  // ... do something
}
MySpecialization::~MySpecialization() {
  // ... do something
}
StatusCode MySpecialization::myMethod() {
  // ... do something
  return StatusCode::SUCCESS;
}
StatusCode MySpecialization::queryInterface(const InterfaceID &riid, void **ppvInterface) {
  if (ppvInterface == 0) return StatusCode::FAILURE;
  if (IMyFeature::interfaceId() == riid)  { // sometimes is IID_IMyFeature
    *ppvInterface = (IMyFeature*)this;
  }
  else  {
    return BaseClass::queryInterface(riid, ppvInterface);
  }
  addRef();
  return StatusCode::SUCCESS;
}
```

The implementation of `queryInterface` is (usually) a switch-like chain of `if` statements falling back on the base-class implementation. The implementation of such a method can be automated, in fact the `AlgTool` base class has a generic implementation based on a run-time list of implemented interfaces, so, when the base class is `AlgTool`, the example above becomes:

```
class MySpecialization: public AlgTool, virtual public IMyFeature {
public:
  /// Constructor
  MySpecialization();
  /// Destructor
  virtual ~MySpecialization();
  /// Implementation from IMyFeature
```

```
  virtual StatusCode myMethod();
};
MySpecialization::MySpecialization():AlgTool() {
  declareInterface<IMyFeature>(this);
}
MySpecialization::~MySpecialization() {
  // ... do something
}
StatusCode MySpecialization::myMethod() {
  // ... do something
  return StatusCode::SUCCESS;
}
```

The code for `AlgTool` specializations is shorter and less error-prone. The only draw-back that has not been removed is that the declaration of the implemented interfaces happens in two places: the `.h` file and the `.cpp` one, so you have to edit the two files in a consistent way.

Another limitation of the current interface implementation is that it is not possible to have interfaces extending other interfaces. It is not a technical limitation (C++ allows it, of course), but it breaks the design principles of the framework. Each interface is identified by a numerical id (usually the a hash of the name) and a version (major and minor) that allow to tell if the component that we loaded from a library is compatible with the version of the interface that we used to compile our code. When an interface inherits from another and the base interface is changed, one should modify the version numbers of the derived interface, but it is simply impossible to keep track of those kind of chains.

Using some specially crafted templated helper classes, a dedicated `InterfaceId` class and some "Boost::mpl" (Meta Programming Library) constructs, it is possible to make the compiler generate all the mechanical code that is needed, simplifying the maintenance of the components and allowing non-trivial interface hierarchies.

Implementation

The implementation of the new interfaces infrastructure is based on the templated classes:

`implements#<...>`
: Used to write a class that does not have a concrete base class.

`extends#<BASE, ...>`
: Used to write a class that inherits from a concrete base class.

`extend_interfaces#<...>`
: Used to declare an interface that inherits from other interfaces.

Due to the absence of variadic template arguments in C++, the classes have different names depending on the number of interfaces passed as templates, e.g.:

- `implements1`
- `extends2<AlgTool,ISpecial1,ISpecial2>`
- `extend_interfaces1`

The other key element of the new infrastructure is the preprocessor macro `DeclareInterfaceID`, which expands to some common code.

Interface

An interface declaration looks like:

```
class GAUDI_API IIncidentListener: virtual public IInterface {
public:
  /// InterfaceID
  DeclareInterfaceID(IIncidentListener,2,0);
  /// Inform that a new incident has occurred
```

```
  virtual void handle(const Incident&) = 0;
};
```

Line 1 make the interface extend `IInterface` (one can extend other interfaces, like `INamed`). Line 4 declare the interface name and version.

Component base class

```
class GAUDI_API Algorithm: virtual public implements3<IAlgorithm, IProperty, IStateful> {
public:
  // ... no queryInterface
};
```

Line 1 shows how to use the the `implements#<...>` helper to implements 3 interfaces.

Extension of a component

```
class GAUDI_API ConversionSvc: public extends2<Service, IConversionSvc, IAddressCreator> {
public:
  // ... no queryInterface
};
// Constructor
ConversionSvc::ConversionSvc(...): base_class(...) {}
```

The first template argument of `extends#<...>` is the component base class, for the rest, it works exactly as `implements#<...>`.

Line 6 shows how the constructor of the derived class must be written. `base_class` is a typedef to the actual base class (`extends#<...>`) that allows to avoid to repeat the list of interfaces in more than one place.

Migrating old code

Interfaces

- If the interface inherits from more than one interface, it has to be modified to inherit from `extend_interfacesX<IInterface1,...>` (replacing X with the number of interfaces).
- We must add the call to `DeclareInterfaceID` in the `public` section, close to the declaration of the class.
- We must remove the static function `interfaceID` and all the references to the `IID_` static variable.

Example: IAlgorithm.

Component base class

- Make the class inherit from `implements#<...>`.
- Remove the implementation of `queryInterface`, `addRef` and `release` (unless non-trivial).

Extension of a component

- Make the class inherit from `extends#<...>`.
- Modify the constructor to call the constructor of `base_class`.
- Remove the implementation of `queryInterface`, `addRef` and `release` (unless non-trivial). In case of AlgTool specializations, remove the "declareInterface" lines from the constructor.

Backward compatibility

When Compiled with the macro `GAUDI_V20_COMPAT` defined, the old code will work in most cases. The only cases where it will break are when the `IID_` static constant variables are used directly. Those variables had to be removed to avoid conflicts. Replacing the `IID_` constants with calls to the static method `interfaceID()` will fix the user code in a backward compatible way (i.e. it works on both Gaudi v20 and v21 in compatibility mode).

Symbol visibility

Rationale

Since gcc 4.0 and on Windows it is possible to explicitly declare which are the symbol that should be visible from outside a library. The GCC Wiki has a good page about symbol visibility, so I'll not discuss it here.

Implementation

The implementation is based on the initial proposal by Sebastian Binet and the gcc documentation.

The visibility of symbols is declared using the macros `GAUDI_IMPORT`, `GAUDI_EXPORT`, `GAUDI_LOCAL`, `GAUDI_API` (`GaudiKernel/Kernel.h`):

```
#ifdef _WIN32
#  define GAUDI_IMPORT __declspec(dllimport)
#  define GAUDI_EXPORT __declspec(dllexport)
#  define GAUDI_LOCAL
#else
#  if defined(GAUDI_HASCLASSVISIBILITY)
#    define GAUDI_IMPORT __attribute__((visibility("default")))
#    define GAUDI_EXPORT __attribute__((visibility("default")))
#    define GAUDI_LOCAL  __attribute__((visibility("hidden")))
#  else
#    define GAUDI_IMPORT
#    define GAUDI_EXPORT
#    define GAUDI_LOCAL
#  endif
#endif

// Define GAUDI_API for DLL builds
#ifdef GAUDI_LINKER_LIBRARY
#define GAUDI_API GAUDI_EXPORT
#else
#define GAUDI_API GAUDI_IMPORT
#endif
```

where GAUDI_HASCLASSVISIBILITY is defined for gcc >= 4 and not for CINT. Note that the visibility attributes in gcc make sense only if the source is compiled with the command line flag `-fvisibility=hidden`.

In the linker libraries, the exported classes are declared as `GAUDI_API`:

```
class GAUDI_API Algorithm: virtual public implements3<IAlgorithm, IProperty, IStateful> {
...
```

while the exceptions are declared as `GAUDI_EXPORT`:

```
class GAUDI_EXPORT GaudiException: virtual public std::exception {
...
```

A special action is needed to compile on Windows: an environment variable is used to tell to the batch script that create the library if the symbols have to be exported automatically.

Migrating old code

For component libraries, no operation is needed.

For linker libraries we have to follow some simple rules:

- we need to add `GAUDI_API` to the classes that have at least one method implemented in a .cpp
  - note that it is enough to export only the non-inline methods.
- the classes that need a `dynamic_cast` (DataObject etc.) must be exported as `GAUDI_API` (to use the same virtual table in all the libraries)
- exception classes must be exported as `GAUDI_EXPORT`

To be able to compile adapted code with the old version of Gaudi, one should add definitions for the macros `GAUDI_IMPORT`, `GAUDI_EXPORT`, `GAUDI_LOCAL`, `GAUDI_API` if not defined. E.g.:

```
#if defined(GAUDI_V20_COMPAT) && ! defined(GAUDI_API)
#define GAUDI_API
#define GAUDI_IMPORT
#define GAUDI_EXPORT
#define GAUDI_LOCAL
#endif
```

Backward compatibility

For backward compatibility, the macros are declared empty and the command line option is not added when compiling in backward compatibility mode, so no action is needed in the user code.

-- MarcoClemencic - 19 Feb 2009

This topic: Gaudi > HowToMigrateToGaudi21
Topic revision: r5 - 2009-09-03 - MarcoClemencic