Everything about nothing

Random notes of what's on my mind. Additional materials you'll find on my homepage.

Sunday, January 3, 2016

Few tips about GObject for C++ programmers

While studying NetworkManager code I got more and more comfortable with OO programming retrofitted into C programming language using GObject library. At first I was confused because it is quite complex and I didn't well understand how it works. Furthermore, the documentation for beginners is lacking and scare. But, the more I learned the more logical it seemed to me. Since I'm well acquainted with C programming language and to some extent to C++ I decided to write about OO model in GObject, NetworkManager and C based on what someone might expect from C++. As usual, this is for my later reference, but also for everyone else wanting to learn how to use or understand GObject. I'll write in a form of tips, or details you should know in order to better understand how it works.

Declaring a class

In C++ language there are two parts of class implementation. First, there is a *class declaration* and then there is a *class definition*. Class declaration goes into a header file (e.g. `Point.h`) and there you'll find something like this:

```
class Point {
}
```

Now, the question is how to do this in C using GObject? The process is a bit more involved due to the characteristics of a C programming language. Anyway, the recipe is the following one. In the header file (e.g. `point.h`) that declares class method you would put the following code:

```
#include <glib-object.h>

#define TYPE_POINT (point_get_type())

typedef struct _PointClass {
    GObjectClass parent_class;
} PointClass;

typedef struct _Point {
    GObject parent_instance;
} Point;

GType point_get_type(void);
```

This code declares two structures, one that will be used by a class (`PointClass`) and the another one for each object of the given class type (`Point`). You would also need to define class and object initialization functions as follows:

```
#include "point.h"

typedef struct _PointPrivate PointPrivate;

G_DEFINE_TYPE (Point, point, G_TYPE_OBJECT);

static void point_class_init (PointClass* klass)
{
}

static void point_init (Point* self)
{
}
```

Finally, you can write a main function (in e.g. `main.c` file) which doesn't do anything, but, nevertheless the class is here. That's how you would declare a class.

The code for this example can be found on GitHub.

Final and derivable classes

Pages

- My Blog's Main Page
- My Homepage @UNIZG
- Information security and Privacy Lab @ FER.UNIZG
- Blog for CISOs
- My Students' Blog

Search This Blog

Search

Labels (subset)

alfresco (5) anonymous (3) arpwatch (4) biseri (9) C (7) centos (38) centos6 (18) computer networks (4) configuration (9) development (5) dns (6) english (204) fedora (41) firefox (7) FreeIPA (5) google (6) history (6) hrvatski (48) installation (9) linux (61) mail (3) netadm (18) network (11) openssh (3) ossec (9) oval (3) ovaldi (5) privatnost (3) problem (13) problems (2) programming (13) redhat (6) security (42) sigurnost (7) software (6) ssh (7) sysadm (84) tcp (5) tips (7) tunneling (7) Unix (6) vmware (22) zimbra (10)

Most Popular last 7 days...

Cracking raw MD5 hashes with John the Ripper

Radno vrijeme ambulante u sklopu Cvjetnog naselja...

Calculating TCP RTO...

Network namespaces and NetworkManager

How to run Firefox in a separate network name space

Few tips about GObject for C++ programmers

Implementing IF, AND, OR, etc. in iptables...

Installing FreeIPA on minimal CentOS installation..

eBay shopping

Getting Libreswan to connect to Cisco ASA 5500

The distinction between the two is that you can not subclass final class, while on the other hand, derivable classes can be sub classed. I don't want to go into discussions why final classes, but only into technical details related to their use in GObject. GObject documentation recommends to use final classes if you can, and only then derivable classes. If you take a look at some other sources, especially for C++, you'll find doubts about benefits of final classes.

So, in C++ (at least from C++11) you would declare final class like this:

```
class Point final {
};
```

Now, if you try to subclass it:

```
class Point2 : public Point {
};
```

You'll receive compiler error due to the final keyword prohibiting class Point to be subclassed:

```
g++ -Wall -std=c++11 main.cpp -o main
In file included from main.cpp:1:0:
Point.h:6:7: error: cannot derive from ‘final’ base ‘Point’ in derived
type ‘Point2’
 class Point2 : public Point {
       ^
```

Note that nothing special has to be done in order for a class to be derivable. Only if you want a class to be final you have to explicitly say so. So, what about GObject and final and derivable classes?

Two macros are used in GObject to create classes for the purpose of creating final and derivable classes. To define final class use `G_DECLARE_FINAL_TYPE` macro. You should modify header file from the previous example like this:

```
#include <glib-object.h>

#define TYPE_POINT (point_get_type())
G_DECLARE_FINAL_TYPE(Point, point, , POINT, GObject)

typedef struct _Point {
    GObject parent_instance;
} Point;
```

Note that there is no definition of type `PointClass`! The reason is that the macro `G_DECLARE_FINAL_TYPE` declares it. Also, `point_get_type()` isn't declared because `G_DECLARE_FINAL_TYPE` declares it. That's basically it for final class.

To declare derivable class, macro G_DECLARE_DERIVABLE_TYPE is used. In this case, as in the previous, you should only change header file which should look like this:

```
#include <glib-object.h>

#define TYPE_POINT (point_get_type())
G_DECLARE_DERIVABLE_TYPE(Point, point, , POINT, GObject)

typedef struct _PointClass {
    GObjectClass parent_instance;
} PointClass;
```

This time, `PointClass` type is defined explicitly while `Point` is defined by `G_DECLARE_DERIVABLE_TYPE` macro.

The code for this example can be found on GitHub.

Instantiate the object of a given class

The next thing is how would you instantiate an object of some class. For sample, to instantiate an object of a class written in C++ from the previous section in the main function you would do the following:

```
Point* point = new Point();
```

To do the same thing in C/GObject combo, you again have to do a bit more work. Actually, you have to define allocator for a class, i.e. something equivalent to a `new` keyword in C++. The way to create a new object of a given type is:

```
Point *point = someclass_new();
```

So, you need to define the function `someclass_new()`. Our will be very simple one:

```
Point* point_new(void)
{
        return g_object_new(TYPE_POINT, NULL);
}
```

Place it at the end of the file `point.c` and in the main function create an object of the Point class by calling the function `point_new()`. As a final note, instantiating final or derivable class is the same.

The code for this example can be found on GitHub.

Instantiate a singleton object

One feature used by NetworkManager a lot are singleton objects. Namely, some objects control system wide resources and thus there is a need to have a single object control a single resource. For example, main component of a NetworkManager is the object NM_TYPE_MANAGER and it is necessary to have only one such object. There is also a single object for a configuration held in the object/class NM_TYPE_CONFIG.

So, how is singleton object created? NetworkManager has some infrastructure that makes this task really simple. You start with a regular class/object. Then in the C file with an object implementation you should call the following two macros/functions:

```
NM_DEFINE_SINGLETON_INSTANCE (NMNetnsController);
NM_DEFINE_SINGLETON_REGISTER (NMNetnsController);
```

Then, in function that allocates an object, you have a variable singleton_instance that should be NULL until the object is created and then it should contain a pointer to a singleton object. Additionally, after creating object, you should call function `nm_singleton_instance_register()`.

You are tasked with taking care that no new object of the given type is created, i.e. you can shoot yourself in the foot in case you don't take care to check the content of the variable `singleton_instance`. `singleton_instance` is a file global variable declared by the macros shown above.

Private attributes

The next step is to add private attributes to our class. Let's suppose that we need to add x and y coordinates. In C++ we would modify the class declaration in the following way:

```
class Point {
private:
        int x, y;
}
```

And that's basically it for C++ version. In case of C/GObject, you would define struct for private data in C file (`point.c`) with the following content:

```
typedef struct _PointPrivate PointPrivate;

struct _PointPrivate {
        int x, y;
};
```

First, observe that the structure is placed in C file, not in the header file. The reason is that this is private/internal structure so no users of a class should know the content of the given structure. Furthermore, keep in mode that this structure isn't required to be called as shown in the example by the GObject system, you can call it whatever you want, but it is a good practice and strongly suggested to add `Private` suffix to the object name for the readability reasons.

Private part for the object should be allocated when the class is initialized. To achieve that add the following line in the class initialization function, i.e. in `point_class_init()` function:

```
g_type_class_add_private (klass, sizeof (PointPrivate));
```

Before finishing with private attributes, there is one more thing we need to discuss, and that is how to access private part of the object. For that purpose it is good to declare the following macro:

```
#define POINT_GET_PRIVATE(object)      \
          (G_TYPE_INSTANCE_GET_PRIVATE((object), TYPE_POINT,
PointPrivate))
```

This macro takes the pointer to an object and returns pointer to the private data of a given object. Note that the last argument to G_TYPE_INSTANCE_GET_PRIVATE is type (structure definition) of a private data. So, when you have a method that accesses the object's private data you would, at the start of the method, call the mentioned macro to obtain pointer to private data structure and then you would access it as usual.

The code for this example can be found on GitHub.

Public methods

After we added private attributes we want to add public methods that will allow us to get and set the values for the given private attributes. So, in C++ we would modify Point class definition as follows:

```
class Point {
private:
        int x, y;
public:
        void setValue(int x, int y);
        int getx(void);
        int gety(void);
};
```

The public methods should be defined in a C++ file (i.e. `Point.cpp`):

```
#include "Point.h"

void Point::setValue(int x, int y)
{
        this->x = x;
        this->y = y;
}

int Point::getx(void)
{
        return this->x;
}

int Point::gety(void)
{
        return this->y;
}
```

And, obviously, the methods would be used in a following way in the main function:

```
point->setValue(1, 1);
std::cout << "x=" << point->getx() << ", y=" << point-
>gety() << std::endl;
```

Now, to achieve the same with GObject system several changes to the existing C code are necessary. First, we'll define a helper macro in a C file (`point.c`) that we will use to obtain private data of an object of class Point:

```
#define POINT_GET_PRIVATE(object)      \
          (G_TYPE_INSTANCE_GET_PRIVATE((object), TYPE_POINT,
PointPrivate))
```

Next, public methods are just global functions with a prefix of a object name and with the first argument being the pointer to an object of a given class. In our case, we have three public methods each with the following prototype:

```
void point_set_value(Point *point, int x, int y);
int point_get_x(Point *point);
int point_get_y(Point *point);
```

Those declaration should go into the header file (`point.h`) because they are accessible to any user of object Point, while their definitions should go into the C file (`point.c`):

```
void point_set_value(Point* point, int x, int y)
{
        PointPrivate* priv = POINT_GET_PRIVATE(point);

        priv->x = x;
        priv->y = y;
}

int point_get_x(Point* point)
{
        PointPrivate* priv = POINT_GET_PRIVATE(point);

        return priv->x;
}

int point_get_y(Point* point)
{
        PointPrivate* priv = POINT_GET_PRIVATE(point);

        return priv->y;
}
```

Finally, you should use those public methods after object of class Point is allocated in the main function. Here is the example:

```
point_set_value(point, 1, 1);
printf("x=%d, y=%d\n", point_get_x(point), point_get_y(point));
```

And that's it. The code for this example can be found in GitHub repository.

Virtual methods

The methods implemented using GObject in the previous section are non-virtual public methods. They are easy to implement, but the least flexible. So, to implement public virtual methods in GObject we need to:

1. In class structure (`PointClass`) add function pointer for each virtual method we need.
2. In header file declare dispatcher methods that will be called by the user of the class.
3. Define dispatcher methods in the source file.
4. Define implementation of virtual methods in the source file.
5. Initialize function pointers with the implementation of virtual methods.

Here are the steps in more details with the concrete example of Point class. Before continuing, note that virtual methods are possible only in derivable classes, not in final!

Step 1 is to add function pointer fields in the class structure. That means that you class structure definition has to have the following format:

```
typedef struct _PointClass
{
        GObjectClass parent_class;
        void (*set_value) (Point *self, int x, int y);
        int (*get_x) (Point *self);
        int (*get_y) (Point *self);
} PointClass;
```

Note the bold parts, those are function pointer that will point to implementation of virtual methods. Each method takes, as the first argument, pointer to the object it has to act upon.

Step 2 is to declare dispatcher methods. Those are very similar to non-virtual public methods in appearance, i.e. the name consists of a object name prefix and the function name. Again, this is recommended, not something that is mandated by GObject system. Anyway, in our case those are the lines you have to add to header file:

```
void point_set_value(Point* self, int x, int y);
int point_get_x(Point* self);
int point_get_y(Point* self);
```

Step 3 is to define dispatcher methods in the source file. I'll do it only for one method, the rest are the same. So, in case of `point_set_value()` it would look like this:

```
void point_set_value(Point* self, int x, int y)
{
 PointClass *klass;
 klass = _POINT_GET_CLASS(self);
 klass->set_value(self, x, y);
}
```

As you can see, what the method does is it access class definition, and then invokes appropriate function through its pointer. I have to stress here two things:

1. I skipped assertions, i.e. if the object/classes are of right type!
2. Note the leading underscore in _POINT_GET_CLASS. It is the consequence of skipping module name when calling G_DECLARE_DERIVABLE_TYPE macro, i.e. I skipped third argument!

So, any user of a class will call those public methods which will dispatch the call to the real method.

In **step 4** we have to define implementation of virtual methods in the source file. Again, I'll show it only for a single method. Also, you'll note that the implementation is the same as for non-virtual public methods. Only we have to call them differently so that compiler can differentiate between different function. I decided to use prefix `objectname_private_`, but it is my choice. I didn't manage to find something recommended by GObject. So, here it goes for the function to set value:

```
void point_private_set_value(Point* self, int x, int y)
{
 PointPrivate *priv = POINT_GET_PRIVATE(self);
 priv->x = x;
 priv->y = y;
}
```

Finally, **step 5** is to initialize pointers to dispatchers which in turn will call real methods. This has to be done by modifying definition of PointClass structure (or class structure in general). Here is how this structure should look like:

```
static void
point_class_init (PointClass* klass)
```

```
{
 g_type_class_add_private (klass, sizeof (PointPrivate));
 klass->set_value = point_private_set_value;
 klass->get_y = point_private_get_y;
 klass->get_x = point_private_get_x;
}
```

Note that we have the part that adds private data for the objects. The bold parts are the ones that initialize virtual functions.

Anyway, that's it for virtual function. You can compile the code on GitHub.

Inheritence and subclassing

Inheritance allows for specialization of some class without touching that class internals. It is one of very important OO mechanisms and as such it is also supported in GObject and used by, e.g. NetworkManager.

So, suppose we want to define 3D point by basing its implementation on Point class while in the same time adding only *z* coordinate. In C++ we would declare new class and say explicitly that we inherit base class Point:

```
#include "Point.h"

class Point3D: public Point
{
private:
        int z;
public:
        void setValue(int x, int y, int z);
        int getz(void);
}
```

As you can see we include declaration of 2D point and then we add third coordinate as well as new methods, one to retrieve the third coordinate and another one to be able to set all three coordinates. There is also definition (i.e. implementation) of a new class which is placed in `Point3D.cpp` file:

```
#include "Point3D.h"

void Point3D::setValue(int x, int y, int z)
{
        Point::setValue(x, y);
        this->z = z;
}

int Point3D::getz(void)
{
        return this->z;
}
```

Note how the base class method is called, by prefixing the method name with the base class name.

As usual, the question is how this is done in C using GObject library. First, GObject make distinction between final and derivable classes, as we already saw above. Base class must be derivable, while subclass might be either final or derivable.

So, we are going to define derivable class `Point` as follows:

```
#include <glib-object.h>

#define TYPE_POINT (point_get_type())
G_DECLARE_DERIVABLE_TYPE(Point, point, , POINT, GObject)

typedef struct _PointClass {
        GObjectClass parent_class;
} PointClass;

Point* point_new(void);

void point_set_value(Point *point, int x, int y);
int point_get_x(Point *point);
int point_get_y(Point *point);
```

This is almost the same as original Point declaration we had. The implementation part isn't changed. Now, Point3D is declared as follows:

```
#include <glib-object.h>
#include "point.h"

#define TYPE_POINT3D (point3d_get_type())
```

```
G_DECLARE_FINAL_TYPE(Point3D, point3d, , POINT3d, Point)

typedef struct _Point3D {
        Point parent_instance;
} Point3D;

Point3D* point3d_new(void);

void point3d_set_value(Point3D* point, int x, int y, int z);
int point3d_get_z(Point3D* point);
```

Note the last parameter in `G_DECLARE_FINAL_TYPE` is `Point`. Basically, that's the base class and up until now we had there `GObject` from which all object descend.

Defining and implementing interfaces

Interfaces in OO languages are used as a form of a contract between different components. Some languages, as Java for example, have direct support for defining interfaces, while others, like C++, don't have and instead use some indirect methods. Here I would like to show how interfaces are implemented in GObject and for that purpose I'll declare interface that will be implemented by Point class used so far for examples. We'll start with C++, as usual.

To define interface in C++ abstract classes are used. Abstract class can not be instantiated because it has at least one pure virtual method that needs to be implemented by subclass. So, here is how interface for Point class would look like in C++:

```
class PointInterface {
public:
 virtual void setValue(int x, int y) = 0;
 virtual int getx(void) = 0;
 virtual int gety(void) = 0;
};
```

As you can see, the pure virtual method is declared by having "equal zero" addition. Now, just one more change is necessary and that is to declare that Point class implements this interface/abstract class. To do that only a single change is necessary, i.e.:

```
class Point : public PointInterface { ... }
```

In other words, `Point` inherits `PointInterface`.

So, how to do this in GObject? Here is the official documentation and we are going to do this for our existing `Point` class.

TBC...

Constructor

In the previous examples we already saw constructors, class constructor (`*_class_init`) and object constructor (`*_init`). Class constructor is called only once, when a first object of a given class is instantiated. Object constructor, on the other hand, is called every time an object of a given class is instantiated. You can easily see this modifying the previous example so that you place `printf()` statements into appropriate constructors (`point_class_init()` for a class constructor and `point_init()` for an object constructor). Then, by creating two objects of a Point class you'll clearly see that the class constructor is called only once, while the object constructor is called as many times as you instantiated objects.

Destructor

Destructor isn't necessary

Object/class initialization and removal

When creating a new object using `g_object_new()` function there is a certain sequence of steps executed. For an object that is first of its class, the sequence is:

1. _class_init()
2. constructor()
3. constructed()
4. _init()

Of those four, the first and fourth, are mandatory and have to be defined in the source. The second and third steps are optional and you define them in class constructor (`_class_init()`) by assigning appropriate pointers to the class interface.

Note that `g_object_new()` accepts an object type that should be created and a sequence of attribute-value pairs terminated with a single NULL. Those attribute-value pairs are used to set properties of an object (installed using `g_object_class_install_property()` function) but only after object constructor finishes. Also, the additional requirement is to define `set_property` class method that will be called to set each property.

For objects of next type only object constructor is called and then properties are initialized.

When object is removed then its destructor is called, if it exists. If this is the last object of its class, then finalize method is called, again, if it exists.

Name spaces

TBD

Literature

If you try to Google for GObject examples or tutorials you'll find some materials from Gnome Web site. But, the truth is that those tutorials leave some open questions and it is really hard to find something else. In the end I managed to find some references that I think are very valuable so here they are:

- Introduction to GObject
- Learning GObject basics
- Avoid G_TYPE_INSTANCE_GET_PRIVATE() in GObjects

Posted by Stjepan Groš (sgros) at 23:34
Labels: english, gobject, networkmanager, programming, tutorial

5 comments:

Pavlo S said...

Very good article. I came from C++ and for me it is so hard so far to build a bridge between OO C++ and Object. Thanks.

February 8, 2017 at 9:37 PM

Anonymous said...

Brilliantly written article with explanation and examples on GitHub.
The article was very helpful for a beginner of GObject.

September 11, 2017 at 9:17 PM

Tomasz Krawczyk said...

Programming in C language is unfortunately too difficult for me and I have no reason to start writing at all. That's why I decided to cooperate with a company that provides ready-made IT solutions that primarily operate in the cloud. You can meet them through https://grapeup.com/about-us/ and certainly everyone interested will find the answer for themselves there.

November 4, 2019 at 8:44 PM

Ayman said...

Thank you, this really helps in explaining GObject and filling the gap between OOP and C programming.

January 14, 2020 at 5:09 AM

Unknown said...

It's a useful article, but instead of repeated "ooh, but this is broken because" examples, it would have been nice to have one, real world, working example. Wrapping a simple c++ object in GObject is pure masochism. No wonder languages like Vala exist so you don't have to deal with this absolute disaster. Every GObject program I've ever used is buggy AF.

April 15, 2020 at 10:38 PM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

About Me

Stjepan Groš (sgros)

scientist, consultant, security specialist, networking guy, system administrator, philosopher ;)

View my complete profile

Blog Archive

- ► 2021 (1)
- ► 2019 (1)
- ► 2018 (2)
- ► 2017 (14)
- ▼ 2016 (13)
  - ► December (1)
  - ► November (2)
  - ► September (1)
  - ► July (1)
  - ► June (1)
  - ► March (2)
  - ► February (2)
  - ▼ January (3)
    - Connections in NetworkManager
    - Few tips about GObject for C++ programmers
    - Processing RA messages in NetworkManager
- ► 2015 (13)
- ► 2014 (19)
- ► 2013 (42)
- ► 2012 (124)
- ► 2011 (59)
- ► 2010 (3)
- ► 2009 (3)
- ► 2008 (8)
- ► 2007 (10)