Glib Object System

typeset by packz with TEX in date 7/5/2007

Introduzione

La **Glib Object System** (più comunemente chiamata **GObject**) è una libreria rilasciata sotto **LGPL** the fornisce la possibilità di usare in un ambiente C puro una programmazione ad oggetti, senza contare la facilità di creare "binding" tramite altri linguaggi (quali il python per esempio).

La difficoltà di creare a runtime una libreria che permetta di definire un comportamento **object oriented** (OO) riguarda i metodi relativi alla classe da implementare: in un sistema OO ogni classe contiene delle proprietà (variabili) e dei metodi (funzioni) per agire su queste (o per alcuni metodi particolari chiamati costruttori e distruttori di creare e rispettivamente distruggere l'oggetto); la prima cosa che uno potrebbe fare per implementare un simile sistema in C, potrebbe essere tramite una `struct` usando delle normali variabili per le proprietà e dei puntatori a funzione per i metodi, così facendo si ha il problema ulteriore di spreco dello spazio in memoria: ogni istanza della classe (cioé ogni "creazione" di un elemento di questo tipo) porta alla riallocazione in memoria a delle funzioni che fanno le medesime cose rispetto a quello di un qualunque altro suo "simile"! Per questo si usa una `struct` aggiuntiva in cui viene instanziata una cosidetta **vtable**, una tabella cioè dei metodi che gli oggetti condividono: questa struttura verrà chiamata la **class structure** per differenziarla dalla **instance structure** che rappresenta l'oggetto vero e proprio.

Solo una ultima parola sui nomi che ci si ritrova a dare agli oggetti creati: oltre al nome standard da affibiare all'oggetto è necessario prefissarlo con un nome che identifichi il cosidetto **namespace** che lo distingua da progetti con nomi di classi analoghi: ecco spiegata la caratteristica delle funzioni GTK in cui esiste sempre il prefisso `gtk_`.

Implementazione

Per implementare una classe tramite GObject sono necessarie almeno due strutture: una relativa alla classe vera e propria, l'altra relativa alla **istanza** della classe: grazie alla mia eccessiva fantasia chiamiamo la classe `object` ed il namespace sarà `some`:

```
typedef struct _SomeObject SomeObject;
typedef struct _SomeObjectClass SomeObjectClass;
```

per iniziare questo tutorial implementerò un semplice oggetto con una proprietà chiamata `property` ed un metodo (pubblico) per ottenere il valore di questa proprietà.

```
struct _SomeObject{
  GObject parent;
  /*variabili pubbliche*/
  int property;
};

struct _SomeObjectClass{
  GObjectClass parent_class;
  int (*get)(SomeObject*);
};
```

Precedentemente a questo, come standard, sono necessarie delle macro

```
#define SOME_OBJECT_TYPE           (some_object_get_type())
#define SOME_OBJECT(obj)           (G_TYPE_CHECH_INSTANCE_CAST((obj),SOME_OBJECT_TYPE,SomeObject))
#define SOME_OBJECT_CLASS(klass)   (G_TYPE_CHECK_CLASS_CAST((klass),SOME_OBJECT_TYPE,SomeObjectClass))
#define SOME_IS_OBJECT(obj)        (G_TYPE_CHECK_INSTANCE_TYPE((obj),SOME_OBJECT_TYPE))
#define SOME_IS_OBJECT_CLASS(klass)   (G_TYPE_CHECK_CLASS_TYPE((klass),SOME_OBJECT_TYPE))
#define SOME_OBJECT_GET_CLASS(obj) (G_TYPE_GET_CLASS((obj),SOME_OBJECT_TYPE,SomeObjectClass))
```

dove

`SOME_OBJECT_TYPE` restituisce il `GType`
`SOME_OBJECT` restituisce un puntatore a `SomeObject`
`SOME_OBJECT_CLASS` restituisce un puntatore a `SomeObjectClass`
`SOME_IS_OBJECT` restituisce `TRUE` nel caso l'oggetto a cui è applicato risulta essere della classe `SomeObject`
`SOME_IS_OBJECT_CLASS` restituisce `TRUE` nel caso l'oggetto a cui è applicato risulta essere della classe `SomeObjectClass`
`SOME_OBJECT_GET_CLASS` restituisce il puntatore alla classe a cui appartiene l'oggetto a cui è applicato.

ed il tutto va inserito nel file header tra le macro `G_BEGIN_DECLS` e `G_END_DECLS` che hanno lo scopo di creare automaticamente le righe di codice necessarie per la gestione dell'oggetto.

Passiamo adesso al file sorgente vero e proprio (il `.c` per capirci) e definiamo l'oggetto su cui vogliamo lavorare tramite la macro

```
G_DEFINE_TYPE(SomeObject,some_object,G_TYPE_OBJECT)
```

dove `SomeObject` definisce il GType dell'oggetto, `some_object` il prefisso con il quale chiamare i metodi ed infine l'ultimo argomento della macro designa da quale classe deve prendere origine il nostro oggetto: in questo caso il tipo fondamentale.

In seguito bisogna definire i metodi ed in particolare quelli relativi all'instanziamento della classe e dell'oggetto tramite le funzioni

```
static void some_object_init(SomeObject* self){
  self->property = 1;
}

static void some_object_class_init(SomeObjectClass* klass){
  klass->get = some_object_get;
}
```

che sono private, la funzione per creare il nuovo oggetto

```
SomeObject* some_object_new(){
  return g_object_new(SOME_OBJECT_TYPE,NULL);
}
```

ed infine il metodo per ottenere la proprietà settata

```
int some_object_get(SomeObject* self){
  return self->property;
}
```

Salvando tutto questo nel file `some_object.c` possiamo compilarlo tramite il comando

```
gcc -Wall -c some_object.c `pkg-config --cflags glib-2.0`
```

per ottenere il file oggetto `some_object.o` da includere nei vostri programmi. Ricordarsi di includere `some_object.h`!

Codice
Siccome abbiamo detto all'inizio che questa libreria è basata sulle funzionalità del Glib Dynamic Type System per cui prima di poter definire qualunque funzione è necessario chiamare `void g_type_init(void)` che appunto inizializza il tutto.

```
#include"some_object.h"
int main(int argc,char* argv[]){
  SomeObject* oggetto_qualsiasi;
  g_type_init();
  oggetto_qualsiasi = some_object_new();
  printf("property: %d\n",some_object_get(oggetto_qualsiasi));
  return 0;
}
```

Per ultima cosa, ma fondamentale, sono necessari i seguenti passi per ottenere codice compilabile:

1 - includere `<glib-object.h>`
2 - includere i file header contenenti le definizioni delle classi create (`some_object.h` nel nostro caso).
3 - mettere correttamente le flags al compilatore: nel caso si usi la versione installata in un sistema avente `pkg-config` bisogna usare `‘pkg-config --libs --cflags gobject-2.0‘`

Cosìse il codice subito sopra lo salviamo nel file `main.c` dobbiamo compilarlo nella seguente maniera per ottenere l'eseguibile

```
gcc main.c some_object.o -o main ‘pkg-config --cflags --libs gobject-2.0‘
```

GObject
Per dovere di cronaca riporto il codice completo della classe base

```
typedef struct {
  GTypeClass   g_type_class;

   /* overridable methods */
   GObject* (*constructor) (GType type,guint n_construct_properties,GObjectConstructParam *construct_properties);
  void (*set_property) (GObject *object, guint property_id, const GValue *value, GParamSpec *pspec);
   void (*get_property) (GObject *object, guint property_id, GValue *value, GParamSpec *pspec);
   void (*dispose) (GObject *object);
   void (*finalize) (GObject *object);

   /* seldomly overidden */
   void (*dispatch_properties_changed) (GObject *object, guint n_pspecs, GParamSpec **pspecs);

   /* signals */
   void (*notify) (GObject  *object, GParamSpec  *pspec);
} GObjectClass;
```

Approfondimenti
A dire la verità la documentazione non è molto esauriente, trovare informazioni intese come guide è molto difficile! personalmente ho imparato di più leggendo il codice sorgente di applicazioni che usano queste

librerie per estendere oggetti preesistenti (come per esempio le **GTK+**) che dalla documentazione ufficiale che si trova sul web.

API reference: http://docs.linux.cz/programming/gnome/developer.gnome.org/doc/API/2.0/gobject/

Extending a GtkDrawingArea (using cairo): http://gnomejournal.org/article/34/writing-a-widget-using-cairo-and-gtk28

GTK+ / Gnome Application Development http://developer.gnome.org/doc/GGAD/ggad.html

Revisioni di questo documento e codice: http://www4.autistici.org/packz/