Vala: my new favorite leaky abstraction

Christopher White (CXW / cxw42)
Conference in the Cloud 2020

Vala: GObject's VBA

- Compiles to C
- Wraps GLib+GObject in a friendly syntax
- Bindings for GStreamer, …
- Why?

A simple Vala library

```
namespace My {
    public class Foo {
        public enum Language {
            PERL,
            RAKU,
            VALA,
        }
    }
}
```

9 lines

The generated code

```
/* simple.h generated by valac 0.48.6, the Vala compiler, do not modify */

#ifndef __SIMPLE_H__
#define __SIMPLE_H__

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

G_BEGIN_DECLS

#define MY_TYPE_FOO (my_foo_get_type ())
#define MY_FOO(obj) (G_TYPE_CHECK_INSTANCE_CAST ((obj), MY_TYPE_FOO, MyFoo))
#define MY_FOO_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass), MY_TYPE_FOO, MyFooClass))
#define MY_IS_FOO(obj) (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MY_TYPE_FOO))
#define MY_IS_FOO_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), MY_TYPE_FOO))
#define MY_FOO_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS ((obj), MY_TYPE_FOO, MyFooClass))

typedef struct _MyFoo MyFoo;
typedef struct _MyFooClass MyFooClass;
typedef struct _MyFooPrivate MyFooPrivate;
typedef enum  {
        MY_FOO_LANGUAGE_PERL,
        MY_FOO_LANGUAGE_RAKU,
        MY_FOO_LANGUAGE_VALA
} MyFooLanguage;

#define MY_FOO_TYPE_LANGUAGE (my_foo_language_get_type ())

struct _MyFoo {
        GTypeInstance parent_instance;
        volatile int ref_count;
        MyFooPrivate * priv;
};

struct _MyFooClass {
        GTypeClass parent_class;
        void (*finalize) (MyFoo *self);
};

gpointer my_foo_ref (gpointer instance);
void my_foo_unref (gpointer instance);
GParamSpec* my_param_spec_foo (const gchar* name,
                               const gchar* nick,
                               const gchar* blurb,
                               GType object_type,
                               GParamFlags flags);
void my_value_set_foo (GValue* value,
                       gpointer v_object);
void my_value_take_foo (GValue* value,
                        gpointer v_object);
gpointer my_value_get_foo (const GValue* value);
GType my_foo_get_type (void) G_GNUC_CONST;
G_DEFINE_AUTOPTR_CLEANUP_FUNC (MyFoo, my_foo_unref)
GType my_foo_language_get_type (void) G_GNUC_CONST;
MyFoo* my_foo_new (void);
MyFoo* my_foo_construct (GType object_type);

G_END_DECLS

#endif
```

```
/* simple.c generated by valac 0.48.6, the Vala compiler
 * generated from simple.vala, do not modify */

#include <glib-object.h>
#include <gobject/gvaluecollector.h>
#include "simple.h"

typedef struct _MyParamSpecFoo MyParamSpecFoo;

struct _MyParamSpecFoo {
GParamSpec parent_instance;
};

static gpointer my_foo_parent_class = NULL;

static void my_foo_finalize (MyFoo * obj);
static GType my_foo_get_type_once (void);

static GType
my_foo_language_get_type_once (void)
{
static const GEnumValue values[] = {{MY_FOO_LANGUAGE_PERL, "MY_FOO_LANGUAGE_PERL", "perl"},
{MY_FOO_LANGUAGE_RAKU, "MY_FOO_LANGUAGE_RAKU", "raku"}, {MY_FOO_LANGUAGE_VALA,
"MY_FOO_LANGUAGE_VALA", "vala"}, {0, NULL, NULL}};
GType my_foo_language_type_id;
my_foo_language_type_id = g_enum_register_static ("MyFooLanguage", values);
return my_foo_language_type_id;
}

GType
my_foo_language_get_type (void)
{
static volatile gsize my_foo_language_type_id__volatile = 0;
if (g_once_init_enter (&my_foo_language_type_id__volatile)) {
GType my_foo_language_type_id;
my_foo_language_type_id = my_foo_language_get_type_once ();
g_once_init_leave (&my_foo_language_type_id__volatile, my_foo_language_type_id);
}
return my_foo_language_type_id__volatile;
}

MyFoo*
my_foo_construct (GType object_type)
{
MyFoo* self = NULL;
self = (MyFoo*) g_type_create_instance (object_type);
return self;
}

MyFoo*
my_foo_new (void)
{
return my_foo_construct (MY_TYPE_FOO);
}

static void
my_value_foo_init (GValue* value)
{
value->data[0].v_pointer = NULL;
}
```

```
static void
my_value_foo_free_value (GValue* value)
{
if (value->data[0].v_pointer) {
my_foo_unref (value->data[0].v_pointer);
}
}

static void
my_value_foo_copy_value (const GValue* src_value,
                         GValue* dest_value)
{
if (src_value->data[0].v_pointer) {
dest_value->data[0].v_pointer = my_foo_ref (src_value->data[0].v_pointer);
} else {
dest_value->data[0].v_pointer = NULL;
}
}

static gpointer
my_value_foo_peek_pointer (const GValue* value)
{
return value->data[0].v_pointer;
}

static gchar*
my_value_foo_collect_value (GValue* value,
                            guint n_collect_values,
                            GTypeCValue* collect_values,
                            guint collect_flags)
{
if (collect_values[0].v_pointer) {
MyFoo * object;
object = collect_values[0].v_pointer;
if (object->parent_instance.g_class == NULL) {
return g_strconcat ("invalid unclassed object pointer for value type `", G_VALUE_TYPE_NAME (value), "'",
NULL);
} else if (!g_value_type_compatible (G_TYPE_FROM_INSTANCE (object), G_VALUE_TYPE (value))) {
return g_strconcat ("invalid object type `", g_type_name (G_TYPE_FROM_INSTANCE (object)), "' for value type
`", G_VALUE_TYPE_NAME (value), "'", NULL);
}
value->data[0].v_pointer = my_foo_ref (object);
} else {
value->data[0].v_pointer = NULL;
}
return NULL;
}

static gchar*
my_value_foo_lcopy_value (const GValue* value,
                          guint n_collect_values,
                          GTypeCValue* collect_values,
                          guint collect_flags)
{
MyFoo ** object_p;
object_p = collect_values[0].v_pointer;
if (!object_p) {
return g_strdup_printf ("value location for `%s' passed as NULL", G_VALUE_TYPE_NAME (value));
}
if (!value->data[0].v_pointer) {
*object_p = NULL;
} else if (collect_flags & G_VALUE_NOCOPY_CONTENTS) {
```

The generated code

```
                *object_p = value->data[0].v_pointer;
        } else {
                *object_p = my_foo_ref (value->data[0].v_pointer);
        }
        return NULL;
}

GParamSpec*
my_param_spec_foo (const gchar* name,
        const gchar* nick,
        const gchar* blurb,
        GType object_type,
        GParamFlags flags)
{
        MyParamSpecFoo* spec;
        g_return_val_if_fail (g_type_is_a (object_type, MY_TYPE_FOO), NULL);
        spec = g_param_spec_internal (G_TYPE_PARAM_OBJECT, name, nick, blurb, flags);
        G_PARAM_SPEC (spec)->value_type = object_type;
        return G_PARAM_SPEC (spec);
}

gpointer
my_value_get_foo (const GValue* value)
{
        g_return_val_if_fail (G_TYPE_CHECK_VALUE_TYPE (value, MY_TYPE_FOO), NULL);
        return value->data[0].v_pointer;
}

void
my_value_set_foo (GValue* value,
        gpointer v_object)
{
        MyFoo * old;
        g_return_if_fail (G_TYPE_CHECK_VALUE_TYPE (value, MY_TYPE_FOO));
        old = value->data[0].v_pointer;
        if (v_object) {
                g_return_if_fail (G_TYPE_CHECK_INSTANCE_TYPE (v_object, MY_TYPE_FOO));
                g_return_if_fail (g_value_type_compatible (G_TYPE_FROM_INSTANCE (v_object),
G_VALUE_TYPE (value)));
                value->data[0].v_pointer = v_object;
                my_foo_ref (value->data[0].v_pointer);
        } else {
                value->data[0].v_pointer = NULL;
        }
        if (old) {
                my_foo_unref (old);
        }
}

void
my_value_take_foo (GValue* value,
        gpointer v_object)
{
        MyFoo * old;
        g_return_if_fail (G_TYPE_CHECK_VALUE_TYPE (value, MY_TYPE_FOO));
        old = value->data[0].v_pointer;
        if (v_object) {
                g_return_if_fail (G_TYPE_CHECK_INSTANCE_TYPE (v_object, MY_TYPE_FOO));
                g_return_if_fail (g_value_type_compatible (G_TYPE_FROM_INSTANCE (v_object),
G_VALUE_TYPE (value)));
                value->data[0].v_pointer = v_object;
        } else {
                value->data[0].v_pointer = NULL;
        }
        if (old) {
                my_foo_unref (old);
        }
}

static void
my_foo_class_init (MyFooClass * klass,
        gpointer klass_data)
{
        my_foo_parent_class = g_type_class_peek_parent (klass);
        ((MyFooClass *) klass)->finalize = my_foo_finalize;
}

static void
my_foo_instance_init (MyFoo * self,
        gpointer klass)
{
        self->ref_count = 1;
}

static void
my_foo_finalize (MyFoo * obj)
{
        MyFoo * self;
        self = G_TYPE_CHECK_INSTANCE_CAST (obj, MY_TYPE_FOO, MyFoo);
        g_signal_handlers_destroy (self);
}

static GType
my_foo_get_type_once (void)
{
        static const GTypeValueTable g_define_type_value_table = { my_value_foo_init,
my_value_foo_free_value, my_value_foo_copy_value, my_value_foo_peek_pointer, "p",
my_value_foo_collect_value, "p", my_value_foo_lcopy_value };
        static const GTypeInfo g_define_type_info = { sizeof (MyFooClass), (GBaseInitFunc) NULL,
(GBaseFinalizeFunc) NULL, (GClassInitFunc) my_foo_class_init, (GClassFinalizeFunc) NULL, NULL, sizeof
(MyFoo), 0, (GInstanceInitFunc) my_foo_instance_init, &g_define_type_value_table };
        static const GTypeFundamentalInfo g_define_type_fundamental_info =
{ (G_TYPE_FLAG_CLASSED | G_TYPE_FLAG_INSTANTIATABLE | G_TYPE_FLAG_DERIVABLE |
G_TYPE_FLAG_DEEP_DERIVABLE) };
        GType my_foo_type_id;
        my_foo_type_id = g_type_register_fundamental (g_type_fundamental_next (), "MyFoo",
&g_define_type_info, &g_define_type_fundamental_info, 0);
        return my_foo_type_id;
}

GType
my_foo_get_type (void)
{
        static volatile gsize my_foo_type_id__volatile = 0;
        if (g_once_init_enter (&my_foo_type_id__volatile)) {
                GType my_foo_type_id;
                my_foo_type_id = my_foo_get_type_once ();
                g_once_init_leave (&my_foo_type_id__volatile, my_foo_type_id);
        }
        return my_foo_type_id__volatile;
}

gpointer
my_foo_ref (gpointer instance)
{
MyFoo * self;
self = instance;
g_atomic_int_inc (&self->ref_count);
return instance;
}

void
my_foo_unref (gpointer instance)
{
        MyFoo * self;
        self = instance;
        if (g_atomic_int_dec_and_test (&self->ref_count)) {
                MY_FOO_GET_CLASS (self)->finalize (self);
                g_type_free_instance ((GTypeInstance *) self);
        }
}
```

312 lines

The downsides

The downsides

- Bindings are sometimes broken

The downsides

- Bindings are sometimes broken
- Debugging, single-step:

The downsides

- Bindings are sometimes broken
- Debugging, single-step:
  - the generated C code

The downsides

- Bindings are sometimes broken
- Debugging, single-step:
  - the generated C code
  - A mix of C and Vala!

```
MyFoo*
my_foo_construct (GType object_type)
{
    MyFoo* self = NULL;
#line 2 "simple.vala"
    self = (MyFoo*) g_type_create_instance
(object_type);
#line 2 "simple.vala"
    return self;
#line 93 "simple.c"
}
```

The upsides

The upsides

- Automatic memory management!!!!!!!!!!!!!!!!!!!!

The upsides

- Automatic memory management!!!!!!!!!!!!!!!!!!!!
- Real OO in C...

The upsides

- Automatic memory management!!!!!!!!!!!!!!!!!!!!
- Real OO in C...
- ...with a terser, cleaner syntax
  - Unified syntax for non-virtual methods, virtual methods, non-virtual inherited methods, virtual inherited methods

The upsides

- Automatic memory management!!!!!!!!!!!!!!!!!!!!
- Real OO in C...
- ...with a terser, cleaner syntax
  - Unified syntax for non-virtual methods, virtual methods, non-virtual inherited methods, virtual inherited methods
- Namespacing

The upsides

- Automatic memory management!!!!!!!!!!!!!!!!!!!!
- Real OO in C...
- ...with a terser, cleaner syntax
  - Unified syntax for non-virtual methods, virtual methods, non-virtual inherited methods, virtual inherited methods
- Namespacing
- C ABI, so can be consumed by anything

Colophon and appreciation

- Template: fedora-odometer by Paul W. Frields, bundled by Liam Doherty
  https://github.com/dohliam/libreoffice-impress-templates/tree/master/fedora-slideshow/fedora-odometer
- Fonts: Liberation by Steve Matteson et al.; Anonymous Pro by Mark Simonson

Thank you!

https://metacpan.org/author/CXW

https://devwrench.wordpress.com/

https://github.com/cxw42

https://demozoo.org/sceners/65996/

Happy hacking!