Glib

- General purpose utility library for implementing non graphical features
- Can be used independently
- Cross platform interface
- Vast array of data types supported (Linked lists, hash tables, trees, Strings etc)
- Utility functions – file manipulation, internationalization, warnings, debug flags etc

GObject

- Originally GtkObject class (part of GTK+ 1)
- Allow easy access to C objects from other programming languages
- Object oriented interface in C
- GType - Supports single inheritance
- Supports nonclassed data types
- GValue, GObject

Class

- Classes are data structures
- Variables of a class type – instances
- Contains various data fields (attributes) – property
- Has several functions – methods
- Object – data structure in memory that conforms to the class.
- Process of creation – instantiation using constructor

Declaring a Gobject

- Consists of two structures
  - Instance structure – holds the class attributes and is the basis for an object in memory
  - Class structure – prototypes for certain methods and all signals that the object can provide

  -
    ```
    typedef struct _myObject
    {
      GObject parent_instance;
      guint num;
    } MyObject;
    ```

Declaring a Gobject (contd)

```
typedef struct _myObjectClass
{
  GObjectClass parent_class;
  void (*some_signal) (MyObject *obj);
} MyObjectClass;
```

Utility macros

- #define TYPE_MY_OBJECT (myobject_get_type())
- #define MYOBJECT(object) G_TYPE_CHECK_INSTANCE_CAST((object), TYPE_MYOBJECT, MyObject))
- #define MYOBJECT_CLASS(klass) G_TYPE_CHECK_CLASS_CAST((klass), TYPE_MYOBJECT, MyObjectClass))
- #define IS_MYOBJECT(object) (G_TYPE_CHECK_INSTANCE_TYPE((object), TYPE_MYOBJECT))
- #define IS_MYOBJECT_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE((klass), TYPE_MYOBJECT))
- #define MYOBJECT_GET_CLASS(object) (G_TYPE_INSTANCE_GET_CLASS((object), TYPE_MYOBJECT, MyObjectClass))

Instance Type Identifier

```
GType myobject_get_type(void)
{
    static GType myobject_type = 0;
    if (!myobject_type)
    {
        static const GTypeInfo myobject_info = {
            sizeof(MyObjectClass),          /* class structure size */
            NULL,                           /* base class initializer */
            NULL,                           /* base class finalizer */
            (GClassInitFunc)myobject_class_init,    /* class initializer */
            NULL,                           /* class finalizer */
            NULL,                           /* class data */
            sizeof(MyObject),               /* instance structure size */
            16,                             /* preallocated instances */
            NULL,                           /* instance initializer */
            NULL                            /* function table */
        };
        myobject_type = g_type_register_static(
                G_TYPE_OBJECT,              /* parent class */
                "MyObject",                 /* type name */
                &myobject_info,             /* GTypeInfo struct (above) */
                0);                         /* flags */
    }
    return myobject_type;
}
```

Base Class : GObject

- G_TYPE_OBJECT returns GObject's type identifier. Don't confuse this with G_OBJECT_TYPE.
- G_OBJECT(object) casts object to the GObject instance structure.
- G_OBJECT_CLASS(klass) casts an object class structure klass to the GObjectClass class structure.
- G_IS_OBJECT(object) returns TRUE if the object parameter is an instance of a GObject. This should return TRUE for any object that you define with GObject, unless you're very daring and decide to make your own base object.
- G_IS_OBJECT_CLASS(klass) returns TRUE if klass is a class structure. It should return TRUE for any class structure within the GObject system.
- G_OBJECT_GET_CLASS(object) returns the class structure (GObjectClass) corresponding to any instance structure.

Base Class Methods

```
typedef struct {
  GTypeClass   g_type_class;
  /* seldomly overidden */
  GObject*   (*constructor)     (GType type, guint
n_construct_properties, GObjectConstructParam
*construct_properties);
  /* overridable methods */
  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);
```

Base Class Methods - Contd

```
/* seldomly overidden */
void       (*dispatch_properties_changed) (GObject      *object,
guint    n_pspecs,
GParamSpec  **pspecs);
/* signals */
void       (*notify)                 (GObject *object,
GParamSpec *pspec);
/* called when done constructing */
void       (*constructed)            (GObject *object);
} GObjectClass;
```

Methods

- Methods usually do not appear in class structure. Instead, method prototypes usually appear somewhere soon after the class structure.
- • A method's name should reflect the class name (for example, media_*() for Media).
- • A method's first parameter is always an object (a structure of the instance class). Any remaining parameters are up to you.
- • In public methods, always check that the first parameter is actually a valid object of the method's class.
- • In addition, cast this object parameter after you do the check, because the object you get could be in a subclass.
- • Be careful about setting an object's attributes. Standard GTK+/GNOME practice dictates that all attributes are properties; use that system for setting attributes.

Methods (Contd)

-
  ```
  void my_object_print_num(MyObject *obj)
  {
    MyObject* myobj;
    g_return_if_fail(IS_MY_OBJECT(obj));
    myobj = MY_OBJECT(obj);
    g_print(”MyObject number: %d\n”, myobj->num);
  }
  ```

Properties

- System to set and retrieve data on GObject instances
- Have names and descriptions – self documenting
- Using this system helps to employ object design tool - Glade

Declaring parameters for properties

- GParamSpec
- Use one of the g_param_spec functions

```
/* create GParamSpec descriptions for properties */
num_param = g_param_spec_uint("number-id",      /* identifier */
                              "number", /* nickname */
                              "number on my object", /* description */
                              0,                  /* minimum */
                              UINT_MAX,           /* maximum */
                              0,                  /* default */
                              G_PARAM_READWRITE); /* flags */
```

Installing properties

- g_object_class_install_property(class, id, param)
- enum {

  PROP_0,

  PROP_NUM

  };
- myobject_class_init(MyObjectClass *class)

  {

Installing properties (contd)

```
...
GObjectClass *g_object_class;
/* get handle to base object */
g_object_class = G_OBJECT_CLASS(class);
...
g_object_class−>set_property =
my_object_set_property;
g_object_class−>get_property =
my_object_get_property;
```

Installing properties (contd)

```
/* install properties */
g_object_class_install_property(g_object_class,
                                PROP_INV_NR,
                                num_param);
```

Set Property

```
static void my_object_set_property(GObject *object,
                              guint prop_id,
                              const GValue *value,
                              GParamSpec *pspec)
{
  MyObject *obj;
  guint new_nr;
  obj = MY_OBJ(object);
  switch(prop_id)
  {
    case PROP_NUM:
      new_nr = g_value_get_uint(value);
      if (obj−>num != new_nr)
      {
        obj->num = new_nr;
      }
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID(object, prop_id, pspec);
      break;
  }
```

Why Properties?

- Dynamic system – Subclasses can add their own properties easily
- Define behaviour for property change – reaction for an action
- Easy documentation

Using GObject

- Create an object using g_object_new
- /* create an object, setting some properties */

```
my_obj = g_object_new(TYPE_MY_OBJECT,
                      "number", 4,
                      NULL);
```

- Use g_object_set, g_object_get to set, get properties

Signals - GSignal

- Events that happen to an object during the course of the object's life. Means of communication between objects.
- Signal handler
- Marshalling, Accumulator
- Use signal identifiers. Can be stored in an array.

Signals - GSignal

- enum {

  SOME_SIGNAL,

  LAST_SIGNAL

  }

  static my_object_signals[LAST_SIGNAL];

  some_signal(MyObject* obj);

Signals - GSignal

```
void my_object_class_init(MyObjectClass *class)
{
   ...
    class->some_signal = some_signal;
    my_object_signals[SOME_SIGNAL] = g_signal_new(
  ”some_signal”, /* name */
  TYPE_MY_OBJECT,                    /* class type identifier */
  G_SIGNAL_RUN_LAST|G_SIGNAL_DETAILED, /* options */
  G_STRUCT_OFFSET(MediaClass, unpacked), /* handler offset */
  NULL,                         /* accumulator function */
  NULL,                         /* accumulator data */
  g_cclosure_marshal_VOID__VOID, /* marshaller */
  G_TYPE_NONE,                  /* type of return value */
  0);
  ...
}
```

GSignal

- Options – G_SIGNAL_DETAILED, G_SIGNAL_NO_HOOKS, G_SIGNAL_NO_RECURSE, G_SIGNAL_RUN_FIRST, G_SIGNAL_RUN_LAST, G_SIGNAL_RUN_CLEANUP, G_SIGNAL_ACTION

Emitting a signal

- g_signal_emit
  - g_signal_emit(gpointer object, guint signal_id, GQuark detail, ...)
- g_signal_emit_by_name
  - g_signal_emit_by_name(object, name [, parms ..] [, return])

GSignal Stages

1. Default handlers installed with the G_SIGNAL_RUN_FIRST option
2. Emission hooks
3. User−defined handlers installed without the after option
4. Default handlers installed with the G_SIGNAL_RUN_LAST option
5. User−defined handlers installed with the after option
6. Default handlers installed with the G_SIGNAL_RUN_CLEANUP option

Marshallers

- When some code emits a signal, GSignal uses a marshaller to transport a list of parameters to the signal handler and to collect and propagate any return values.
- prefix_RETURNTYPE__PARM1TYPE[_PARM2TYPE_...]
- g_cclosure_marshal_VOID__BOOLEAN
- _my_marshal_INT__VOID
- glib_genmarshal

```
INT:VOID

VOID:OBJECT,INT

UINT:BOOLEAN
```

Signal Accumulator

- Collect and process all return values of signal handlers
- See devhelp
- For propagation, return TRUE else return FALSE

Attaching Handlers to Signals

- g_signal_connect(instance, detailed_signal, c_handler, data)
- Connects a GCallback function to a signal for a particular object.
- The handler will be called before the default handler of the signal.

instance :the instance to connect to.

detailed_signal :a string of the form "signal-name::detail".

c_handler :the GCallback to connect.

data :data to pass to c_handler calls.

Returns :the handler id

Signal Details

- Signal details are further subdivisions of signals. To specify a detail in a signal name, append two colons and
- the detail name (for example, some_signal::number).

Emission hooks

- Applying signal to an identifier as a whole instead of an object
- Refer devhelp
- g_signal_add_emission_hook

Blocking signal handlers

- g_signal_block
- g_signal_unblock