The Glib Object system v0.10.0

Mathieu Lacage <mathieu@gnome.org>

The Glib Object system v0.10.0

by Mathieu Lacage

If you want to help on this document, feel free to send me a patch to the original xml which is available on http://www.gnome.org/~mathieu/.

While I am always happy to answer questions about this document, I am not always the best person these questions should be asked to. In doubt, always mail gtk-devel-list@gnome.org. You can register to this mailing list on http://mail.gnome.org.

I'd like to thank Owen Taylor, Tim Janik and especially James Henstridge for their help on IRC, when I was trying to figure out the why of some of GObject aspects.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

GObject, and its lower-level type system, GType, are used by GTK+ and most Gnome libraries to provide:

- object-oriented C-based APIs and
- automatic transparent API bindings to other compiled or interpreted languages.

A lot of programmers are used to work with compiled-only or dynamically interpreted-only languages and do not understand the challenges associated with cross-language interoperability. This introduction tries to provide an insight into these challenges. describes briefly the solution choosen by GLib.

The following chapters go into greater detail into how GType and GObject work and how you can use them as a C programmer. I personally find it useful to keep in mind that allowing access to C objects from other interpreted languages was one of the major design goals: this can often explain the sometimes rather convoluted APIs and features present in this library.

Data types and programming

One could say (I have seen such definitions used in some textbooks on programming language theory) that a programming language is merely a way to create data types and manipulate them. Most languages provide a number of language-native types and a few primitives to create more complex types based on these primitive types.

In C, the language provides types such as *char*, *long*, *pointer*. During compilation of C code, the compiler maps these language types to the compiler's target architecture machine types. If you are using a C interpreter (I have never seen one myself but it is possible :), the interpreter (the program which interprets the source code and executes it) maps the language types to the machine types of the target machine at runtime, during the program execution (or just before execution if it uses a Just In Time compiler engine).

Perl and Python which are interpreted languages do not really provide type definitions similar to those used by C. Perl and Python programmers manipulate variables and the type of the variables is decided only upon the first assignment or upon the first use which forces a type on the variable. The interpreter also often provides a lot of automatic conversions from one type to the other. For example, in Perl, a variable which holds an integer can be automatically converted to a string given the required context:

```
my $tmp = 10;
print "this is an integer converted to a string:" . $tmp . "\n";
```

Of course, it is also often possible to explicitely specify conversions when the default conversions provided by the language are not intuitive.

Exporting a C API

C APIs are defined by a set of functions and global variables which are usually exported from a binary. C functions have an arbitrary number of arguments and one return value. Each function is thus uniquely identified by the function name and the set of C types which describe the function arguments and return value. The global variables exported by the API are similarly identified by their name and their type.

A C API is thus merely defined by a set of names to which a set of types are associated. If you know the function calling convention and the mapping of the C types to the machine types used by the platform you are on, you can resolve the name of each function to find where the code associated to this function is located in memory, and then construct a valid argument list for the function. Finally, all you have to do is triger a call to the target C function with the argument list.

For the sake of discussion, here is a sample C function and the associated 32 bit x86 assembly code generated by gcc on my linux box:

```
static void function_foo (int foo)
{}

int main (int argc, char *argv[])
{

        function_foo (10);

        return 0;
}

push   $0xa
call   0x80482f4 <function_foo>
```

The assembly code shown above is pretty straightforward: the first instruction pushes the hexadecimal value 0xa (decimal value 10) as a 32 bit integer on the stack and calls `function_foo`. As you can see, C function calls are implemented by gcc by native function calls (this is probably the fastest implementation possible).

Now, let's say we want to call the C function `function_foo` from a python program. To do this, the python interpreter needs to:

- Find where the function is located. This means probably find the binary generated by the C compiler which exports this functions.
- Load the code of the function in executable memory.
- Convert the python parameters to C-compatible parameters before calling the function.
- Call the function with the right calling convention
- Convert the return values of the C function to python-compatible variables to return them to the python code.

The process described above is pretty complex and there are a lot of ways to make it entirely automatic and transparent to the C and the Python programmers:

- The first solution is to write by hand a lot of glue code, once for each function exported or imported, which does the python to C parameter conversion and the C to python return value conversion. This glue code is then linked with the interpreter which allows python programs to call a python functions which delegates the work to the C function.
- Another nicer solution is to automatically generate the glue code, once for each function exported or imported, with a special compiler which reads the original function signature.
- The solution used by GLib is to use the GType library which holds at runtime a description of all the objects manipulated by the programmer. This so-called *dynamic type*1 library is then used by special generic glue code to automatically convert function parameters and function calling conventions between different runtime domains.

The greatest advantage of the solution implemented by GType is that the glue code sitting at the runtime domain boundaries is written once: the figure below states this more clearly.

Figure 1.1.

Currently, there exist at least Python and Perl generic glue code which makes it possible to use C objects written with GType directly in Python or Perl, with a minimum amount of work: there is no need to generate huge amounts of glue code either automatically or by hand.

Although that goal was arguably laudable, its pursuit has had a major influence on the whole GType/GObject

1 There are numerous different implementations of dynamic type systems: all C++ compilers have one, Java and .NET have one too. A dynamic type system allows you to get information about every instantiated object at runtime. It can be implemented by a process-specific database: every new object created registers the characteristics of its associated type in the type system. It can also be implemented by introspection interfaces. The common point between all these different type systems and implementations is that they all allow you to query for object metadata at runtime.

library. C programmers are likely to be puzzled at the complexity of the features exposed in the following chapters if they forget that the GType/GObject library was not only designed to offer OO-like features to C programmers but also transparent cross-langage interoperability.

Chapter 2. The Glib Dynamic Type System

A type, as manipulated by the Glib type system, is much more generic than what is usually understood as an Object type. It is best explained by looking at the structure and the functions used to register new types in the type system.

```
typedef struct _GTypeInfo               GTypeInfo;
struct _GTypeInfo
{
  /* interface types, classed types, instantiated types */
  guint16                class_size;

  GBaseInitFunc          base_init;
  GBaseFinalizeFunc      base_finalize;

  /* classed types, instantiated types */
  GClassInitFunc         class_init;
  GClassFinalizeFunc     class_finalize;
  gconstpointer          class_data;

  /* instantiated types */
  guint16                instance_size;
  guint16                n_preallocs;
  GInstanceInitFunc      instance_init;

  /* value handling */
  const GTypeValueTable *value_table;
};
GType g_type_register_static (GType             parent_type,
                              const gchar      *type_name,
                              const GTypeInfo  *info,
                              GTypeFlags        flags);
GType g_type_register_fundamental (GType                       type_id,
                                   const gchar                *type_name,
                                   const GTypeInfo            *info,
                                   const GTypeFundamentalInfo *finfo,
                                   GTypeFlags                  flags);
```

`g_type_register_static` and `g_type_register_fundamental` are the C functions, defined in `gtype.h` and implemented in `gtype.c` which you should use to register a new type in the program's type system. It is not likely you will ever need to use `g_type_register_fundamental` (you have to be Tim Janik to do that) but in case you want to, the last chapter explains how to create new fundamental types. 2

Fundamental types are top-level types which do not derive from any other type while other non-fundamental types derive from other types. Upon initialization by `g_type_init`, the type system not only initializes its internal data structures but it also registers a number of core types: some of these are fundamental types. Others are types derived from these fundamental types.

Fundamental and non-Fundamental types are defined by:

- class size: the class_size field in GTypeInfo.
- class initialization functions (C++ constructor): the base_init and class_init fields in GTypeInfo.
- class destruction functions (C++ destructor): the base_finalize and class_finalize fields in GTypeInfo.

2 Please, note that there exist another registration function: the `g_type_register_dynamic`. We will not discuss this function here since its use is very similar to the `_static` version.

- instance size (C++ parameter to new): the instance_size field in GTypeInfo.
- instanciation policy (C++ type of new operator): the n_preallocs field in GTypeInfo.
- copy functions (C++ copy operators): the value_table field in GTypeInfo.
- XXX: GTypeFlags.

Fundamental types are also defined by a set of GTypeFundamentalFlags which are stored in a GTypeFundamentalInfo. Non-Fundamental types are furthermore defined by the type of their parent which is passed as the parent_type parameter to `g_type_register_static` and `g_type_register_dynamic`.

Copy functions

The major common point between *all* glib types (fundamental and non-fundamental, classed and non-classed, instantiable and non-instantiable) is that they can all be manipulated through a single API to copy/assign them.

The GValue structure is used as an abstract container for all of these types. Its simplistic API (defined in `gobject/gvalue.h`) can be used to invoke the value_table functions registered during type registration: for example `g_value_copy` copies the content of a GValue to another GValue. This is similar to a C++ assignment which invokes the C++ copy operator to modify the default bit-by-bit copy semantics of C++/C structures/classes.

The following code shows shows you can copy around a 64 bit integer, as well as a GObject instance pointer (sample code for this is located in the source tarball for this document in `sample/gtype/test.c`):

```
static void test_int (void)
{
  GValue a_value = {0, };
  GValue b_value = {0, };
  guint64 a, b;

  a = 0xdeadbeaf;

  g_value_init (&a_value, G_TYPE_UINT64);
  g_value_set_uint64 (&a_value, a);

  g_value_init (&b_value, G_TYPE_UINT64);
  g_value_copy (&a_value, &b_value);

  b = g_value_get_uint64 (&b_value);

  if (a == b) {
    g_print ("Yay !! 10 lines of code to copy around a uint64.\n");
  } else {
    g_print ("Are you sure this is not a Z80 ?\n");
  }
}

static void test_object (void)
{
  GObject *obj;
  GValue obj_vala = {0, };
  GValue obj_valb = {0, };
  obj = g_object_new (MAMAN_BAR_TYPE, NULL);

  g_value_init (&obj_vala, MAMAN_BAR_TYPE);
  g_value_set_object (&obj_vala, obj);

  g_value_init (&obj_valb, G_TYPE_OBJECT);

  /* g_value_copy's semantics for G_TYPE_OBJECT types is to copy the reference.
     This function thus calls g_object_ref.
     It is interesting to note that the assignment works here because
     MAMAN_BAR_TYPE is a G_TYPE_OBJECT.
```

```
   */
  g_value_copy (&obj_vala, &obj_valb);

  g_object_unref (G_OBJECT (obj));
  g_object_unref (G_OBJECT (obj));
}
```

The important point about the above code is that the exact semantic of the copy calls is undefined since they depend on the implementation of the copy function. Certain copy functions might decide to allocate a new chunk of memory and then to copy the data from the source to the destination. Others might want to simply increment the reference count of the instance and copy the reference to the new GValue.

The value_table used to specify these assignment functions is defined in `gtype.h` and is thoroughly described in the API documentation provided with GObject (for once ;-) which is why we will not detail its exact semantics.

```
typedef struct _GTypeValueTable         GTypeValueTable;
struct _GTypeValueTable
{
  void     (*value_init)         (GValue       *value);
  void     (*value_free)         (GValue       *value);
  void     (*value_copy)         (const GValue *src_value,
                                  GValue       *dest_value);
  /* varargs functionality (optional) */
  gpointer (*value_peek_pointer) (const GValue *value);
  gchar     *collect_format;
  gchar*   (*collect_value)      (GValue       *value,
                                  guint         n_collect_values,
                                  GTypeCValue  *collect_values,
                                  guint         collect_flags);
  gchar     *lcopy_format;
  gchar*   (*lcopy_value)        (const GValue *value,
                                  guint         n_collect_values,
                                  GTypeCValue  *collect_values,
                                  guint         collect_flags);
};
```

Interestingly, it is also very unlikely you will ever need to specify a value_table during type registration because these value_tables are inherited from the parent types for non-fundamental types which means that unless you want to write a fundamental type (not a great idea !), you will not need to provide a new value_table since you will inherit the value_table structure from your parent type.

Conventions

There are a number of conventions users are expected to follow when creating new types which are to be exported in a header file:

- Use the `object_method` pattern for function names: to invoke the method named foo on an instance of object type bar, call `bar_foo`.
- Use prefixing to avoid namespace conflicts with other projects. If your library (or application) is named *Maman*, prefix all your function names with *maman_*. For example: `maman_object_method`.
- Create a macro named `PREFIX_OBJECT_TYPE` which always returns the Gtype for the associated object type. For an object of type *Bar* in a libray prefixed by *maman*, use: `MAMAN_BAR_TYPE`. It is common although not a convention to implement this macro using either a global static variable or a function named `prefix_object_get_type`. We will follow the function pattern wherever possible in this document.
- Create a macro named `PREFIX_OBJECT (obj)` which returns a pointer of type PrefixObject. This macro is used to enforce static type safety by doing explicit casts wherever needed. It also enforces dynamic type safety by doing runtime checks. It is possible to disable the dynamic type checks in production builds (see http://developer.gnome.org/doc/API/2.0/glib/glib-building.html). For example, we would create `MA-`

MAN_BAR (obj) to keep the previous example.

- If the type is classed, create a macro named PREFIX_OBJECT_CLASS (klass). This macro is strictly equivalent to the previous casting macro: it does static casting with dynamic type checking of class structures. It is expected to return a pointer to a class structure of type PrefixObjectClass. Again, an example is: MAMAN_BAR_CLASS.
- Create a macro named PREFIX_IS_BAR (obj): this macro is expected to return a gboolean which indicates whether or not the input object instance pointer of type BAR.
- If the type is classed, create a macro named PREFIX_IS_OBJECT_CLASS (klass) which, as above, returns a boolean if the input class pointer is a pointer to a class of type OBJECT.
- If the type is classed, create a macro named PREFIX_OBJECT_GET_CLASS (obj) which returns the class pointer associated to an instance of a given type. This macro is used for static and dynamic type safety purposes (just like the previous casting macros).

The implementation of these macros is pretty straightforward: a number of simple-to-use macros are provided in gtype.h. For the example we used above, we would write the following trivial code to declare the macros:

```
#define MAMAN_BAR_TYPE           (maman_bar_get_type ())
#define MAMAN_BAR(obj)           (G_TYPE_CHECK_INSTANCE_CAST ((obj), MAMAN_BAR_TYPE, Ma
#define MAMAN_BAR_CLASS(klass)   (G_TYPE_CHECK_CLASS_CAST ((klass), MAMAN_BAR_TYPE, Mam
#define MAMAN_IS_BAR(obj)   (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MAMAN_BAR_TYPE))
#define MAMAN_IS_BAR_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), MAMAN_BAR_TYPE))
#define MAMAN_BAR_GET_CLASS(obj)  (G_TYPE_INSTANCE_GET_CLASS ((obj), MAMAN_BAR_TYPE, Ma
```

Note

Stick to the naming klass as class is a registered c++ keyword.

The following code shows how to implement the maman_bar_get_type function:

```
GType maman_bar_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      /* You fill this structure. */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBarType",
                                   &info, 0);
  }
  return type;
}
```

Non-Instantiable non-classed fundamental types

A lot of types are not instantiable by the type system and do not have a class. Most of these types are fundamental trivial types such as *gchar*, registered in g_value_types_init (in gvaluetypes.c).

To register such a type in the type system, you just need to fill the GTypeInfo structure with zeros since these types are also most of the time fundamental:

```
  GTypeInfo info = {
    0,                          /* class_size */
    NULL,                  /* base_init */
    NULL,                  /* base_destroy */
    NULL,                  /* class_init */
    NULL,                  /* class_destroy */
    NULL,                 /* class_data */
    0,                         /* instance_size */
    0,                         /* n_preallocs */
    NULL,                 /* instance_init */
    NULL,                 /* value_table */
  };
  static const GTypeValueTable value_table = {
    value_init_long0,         /* value_init */
    NULL,                 /* value_free */
    value_copy_long0,         /* value_copy */
    NULL,                 /* value_peek_pointer */
    "i",                 /* collect_format */
    value_collect_int, /* collect_value */
    "p",                 /* lcopy_format */
    value_lcopy_char,         /* lcopy_value */
  };
  info.value_table = &value_table;
  type = g_type_register_fundamental (G_TYPE_CHAR, "gchar", &info, &finfo, 0);
```

Having non-instantiable types might seem a bit useless: what good is a type if you cannot instanciate an instance of that type ? Most of these types are used in conjunction with GValues: a GValue is initialized with an integer or a string and it is passed around by using the registered type's value_table. GValues (and by extension these trivial fundamental types) are most useful when used in conjunction with object properties and signals.

Instantiable classed types: objects

Types which are registered with a class and are declared instantiable are what most closely resembles an *object*. Although GObjects (detailed in Chapter 4, *GObject: what brings everything together.*) are the most well known type of instantiable classed types, other kinds of similar objects used as the base of an inheritance hierarchy have been externally developped and they are all built on the fundamental features described below.

For example, the code below shows how you could register such a fundamental object type in the type system:

```
typedef struct {
  GObject parent;
  /* instance members */
  int field_a;
} MamanBar;

struct _MamanBarClass {
  GObjectClass parent;
  /* class members */
  void (*do_action_public_virtual) (MamanBar *self, guint8 i);

  void (*do_action_public_pure_virtual) (MamanBar *self, guint8 i);
};

#define MAMAN_BAR_TYPE (maman_bar_get_type ())

GType
maman_bar_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBarClass),
      NULL,           /* base_init */
      NULL,           /* base_finalize */
      (GClassInitFunc) foo_class_init,
      NULL,           /* class_finalize */
      NULL,           /* class_data */
      sizeof (MamanBar),
      0,              /* n_preallocs */
      (GInstanceInitFunc) NULL /* instance_init */
    };
    type = g_type_register_fundamental (G_TYPE_OBJECT,
                                        "BarType",
                                        &info, 0);
  }
  return type;
}
```

Upon the first call to `maman_bar_get_type`, the type named *BarType* will be registered in the type system as inheriting from the type *G_TYPE_OBJECT*.

Every object must define two structures: its class structure and its instance structure. All class structures must contain as first member a GTypeClass structure. All instance structures must contain as first member a GTypeInstance structure. The declaration of these C types, coming from `gtype.h` is shown below:

```
struct _GTypeClass
{
  GType g_type;
};
struct _GTypeInstance
{
  GTypeClass *g_class;
};
```

These constraints allow the type system to make sure that every object instance (identified by a pointer to the object's instance structure) contains in its first bytes a pointer to the object's class structure.

This relationship is best explained by an example: let's take object B which inherits from object A:

```
/* A definitions */
typedef struct {
  GTypeInstance parent;
  int field_a;
  int field_b;
} A;
typedef struct {
  GTypeClass parent_class;
  void (*method_a) (void);
  void (*method_b) (void);
} AClass;

/* B definitions. */
typedef struct {
  A parent;
  int field_c;
  int field_d;
} B;
typedef struct {
  AClass parent_class;
  void (*method_c) (void);
  void (*method_d) (void);
} BClass;
```

The C standard mandates that the first field of a C structure is stored starting in the first byte of the buffer used to hold the structure's fields in memory. This means that the first field of an instance of an object B is A's first field which in turn is GTypeInstance's first field which in turn is g_class, a pointer to B's class structure.

Thanks to these simple conditions, it is possible to detect the type of every object instance by doing:

```
B *b;
b->parent.parent.g_class->g_type
```

or, more quickly:

```
B *b;
((GTypeInstance*)b)->g_class->g_type
```

Instanciation of these types can be done with `g_type_create_instance`:

```
GTypeInstance* g_type_create_instance (GType          type);
void           g_type_free_instance   (GTypeInstance *instance);
```

`g_type_create_instance` will lookup the type information structure associated to the type requested. Then, the instance size and instanciation policy (if the n_preallocs field is set to a non-zero value, the type system allocates the object's instance structures in chunks rather than mallocing for every instance) declared by the user are used to get a buffer to hold the object's instance structure.

If this is the first instance of the object ever created, the type system must create a class structure: it allocates a buffer to hold the object's class structure and initializes it. It first copies the parent's class structure over this structure (if there is no parent, it initializes it to zero). It then invokes the base_class_initialization functions (GBaseInitFunc) from topmost fundamental object to bottom-most most derived object. The object's class_init (GClassInitFunc) function is invoked afterwards to complete initialization of the class structure. Finally, the object's interfaces are initialized (we will discuss interface initialization in more detail later). 3

Once the type system has a pointer to an initialized class structure, it sets the object's instance class pointer to the object's class structure and invokes the object's instance_init (GInstanceInitFunc)functions, from top-most fundamental type to bottom-most most derived type.

Object instance destruction through `g_type_free_instance` is very simple: the instance structure is returned to the instance pool if there is one and if this was the last living instance of the object, the class is destroyed.

Class destruction 4 (the concept of destruction is sometimes partly refered to as finalization in Gtype) is the symmetric process of the initialization: interfaces are destroyed first. Then, the most derived class_finalize (ClassFinalizeFunc) function is invoked. The base_class_finalize (GBaseFinalizeFunc) functions are Finally invoked from bottom-most most-derived type to top-most fundamental type and the class structure is freed.

As many readers have now understood it, the base initialization/finalization process is very similar to the C++ Constructor/Destructor paradigm. The practical details are quite different though and it is important not to get confused by the superficial similarities. Typically, what most users have grown to know as a C++ constructor (that is, a list of object methods invoked on the object instance once for each type of the inheritance hierachy) does not exist in GType and must be built on top of the facilities offered by GType. Similarly, GTypes have no instance destruction mechanism. It is the user's responsibility to implement correct destruction semantics on top of the existing GType code. (this is what GObject does. See Chapter 4, *GObject: what brings everything together.*)

For example, if the object B which derives from A is instantiated, GType will only invoke the instance_init callback of object B while a C++ runtime will invoke the constructor of the object type A first and then of the object type B. Furthermore, the C++ code equivalent to the base_init and class_init callbacks of GType is usually not needed because C++ cannot really create object types at runtime.

The instanciation/finalization process can be summarized as follows:

Table 2.1. GType Instantiation/Finalization

Invocation time	**Function Invoked**	**Function's parameters**
First call to `g_type_creat`	type's base_init function	On the inheritance tree of classes from fundamental type to target type. base_init is invoked once for each class structure.

3 The class initialization process is entirely implemented in `type_class_init_Wm` in `gtype.c`.
4It is implemented in `type_data_finalize_class_U` (in `gtype.c`.

Invocation time	Function Invoked	Function's parameters
`e_instance` for target type		
First call to `g_type_create_instance` for target type	target type's class_init function	On target type's class structure
First call to `g_type_create_instance` for target type	interface initialization, see the section called “Interface Initialization”	
Each call to `g_type_create_instance` for target type	target type's instance_init function	On object's instance
Last call to `g_type_free_instance` for target type	interface destruction, see the section called “Interface Destruction”	
Last call to `g_type_free_instance` for target type	target type's class_finalize function	On target type's class structure
Last call to `g_type_free_instance` for target type	type's base_finalize function	On the inheritance tree of classes from fundamental type to target type. base_init is invoked once for each class structure.

Non-instantiable classed types: Interfaces.

GType's Interfaces are very similar to Java's interfaces. To declare one of these you have to register a non-instantiable classed type which derives from GTypeInterface. The following piece of code declares such an interface.

```
#define MAMAN_IBAZ_TYPE             (maman_ibaz_get_type ())
#define MAMAN_IBAZ(obj)             (G_TYPE_CHECK_INSTANCE_CAST ((obj), MAMAN_IBAZ_TYPE
#define MAMAN_IBAZ_CLASS(vtable)    (G_TYPE_CHECK_CLASS_CAST ((vtable), MAMAN_IBAZ_TYPE
#define MAMAN_IS_IBAZ(obj)          (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MAMAN_IBAZ_TYPE
#define MAMAN_IS_IBAZ_CLASS(vtable) (G_TYPE_CHECK_CLASS_TYPE ((vtable), MAMAN_IBAZ_TYPE
#define MAMAN_IBAZ_GET_CLASS(inst)  (G_TYPE_INSTANCE_GET_INTERFACE ((inst), MAMAN_IBAZ_

typedef struct _MamanIbaz MamanIbaz; /* dummy object */
typedef struct _MamanIbazClass MamanIbazClass;

struct _MamanIbazClass {
  GTypeInterface parent;

  void (*do_action) (MamanIbaz *self);
};

GType maman_ibaz_get_type (void);

void maman_ibaz_do_action (MamanIbaz *self);
```

The interface function, `maman_ibaz_do_action` is implemented in a pretty simple way:

```
void maman_ibaz_do_action (MamanIbaz *self)
{
  MAMAN_IBAZ_GET_CLASS (self)->do_action (self);
}
```

`maman_ibaz_get_gtype` registers a type named *MamanIBaz* which inherits from G_TYPE_INTERFACE. All interfaces must be children of G_TYPE_INTERFACE in the inheritance tree.

An interface is defined by only one structure which must contain as first member a GTypeInterface structure. The interface structure is expected to contain the function pointers of the interface methods. It is good style to define helper functions for each of the interface methods which simply call the interface' method directly: `maman_ibaz_do_action` is one of these.

Once an interface type is registered, you must register implementations for these interfaces. The function named `maman_baz_get_type` registers a new GType named MamanBaz which inherits from GObject and which implements the interface MamanIBaz.

```
static void maman_baz_do_action (MamanIbaz *self)
{
  g_print ("Baz implementation of IBaz interface Action.\n");
}


static void
baz_interface_init (gpointer         g_iface,
                    gpointer         iface_data)
{
  MamanIbazClass *klass = (MamanIbazClass *)g_iface;
  klass->do_action = maman_baz_do_action;
}

GType
maman_baz_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBazClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      NULL,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBaz),
      0,      /* n_preallocs */
      NULL    /* instance_init */
    };
    static const GInterfaceInfo ibaz_info = {
      (GInterfaceInitFunc) baz_interface_init,    /* interface_init */
      NULL,               /* interface_finalize */
      NULL          /* interface_data */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBazType",
                                   &info, 0);
    g_type_add_interface_static (type,
                                 MAMAN_IBAZ_TYPE,
                                 &ibaz_info);
  }
  return type;
}
```

`g_type_add_interface_static` records in the type system that a given type implements also FooInterface (`foo_interface_get_type` returns the type of FooInterface). The GInterfaceInfo structure holds information about the implementation of the interface:

```
struct _GInterfaceInfo
{
  GInterfaceInitFunc     interface_init;
  GInterfaceFinalizeFunc interface_finalize;
  gpointer               interface_data;
};
```

Interface Initialization

When an instantiable classed type which registered an interface implementation is created for the first time, its class structure is initialized following the process described in the section called “Instantiable classed types: objects”. Once the class structure is initialized,the function `type_class_init_Wm` (implemented in `gtype.c`) initializes the interface implementations associated with that type by calling `type_iface_vtable_init_Wm` for each interface.

First a memory buffer is allocated to hold the interface structure. The parent's interface structure is then copied over to the new interface structure (the parent interface is already initialized at that point). If there is no parent interface, the interface structure is initialized with zeros. The g_type and the g_instance_type fields are then initialized: g_type is set to the type of the most-derived interface and g_instance_type is set to the type of the most derived type which implements this interface.

Finally, the interface' most-derived `base_init` function and then the implementation's `interface_init` function are invoked. It is important to understand that if there are multiple implementations of an interface the `base_init` and `interface_init` functions will be invoked once for each implementation initialized.

It is thus common for base_init functions to hold a local static boolean variable which makes sure that the interface type is initialized only once even if there are multiple implementations of the interface:

```
static void
maman_ibaz_base_init (gpointer g_class)
{
  static gboolean initialized = FALSE;

  if (!initialized) {
    /* create interface signals here. */
    initialized = TRUE;
  }
}
```

If you have found the stuff about interface hairy, you are right: it is hairy but there is not much I can do about it. What I can do is summarize what you need to know about interfaces:

The above process can be summarized as follows:

Table 2.2. `Interface Initialization`

Invocation time	Function Invoked	Function's parameters
First call to `g_type_create_instance` for type implementing interface	interface' base_init function	On interface' vtable
First call to `g_type_create_instance` for type implementing interface	interface' interface_init function	On interface' vtable

It is highly unlikely (ie: I do not know of *anyone* who actually used it) you will ever need other more fancy things such as the ones described in the following section (the section called “Interface Destruction”).

Interface Destruction

When the last instance of an instantiable type which registered an interface implementation is destroyed, the interface's implementations associated to the type are destroyed by `type_iface_vtable_finalize_Wm` (in `gtype.c`).

`type_iface_vtable_finalize_Wm` invokes first the implementation's `interface_finalize` function and then the interface's most-derived `base_finalize` function.

Again, it is important to understand, as in the section called “Interface Initialization”, that both `interface_finalize` and `base_finalize` are invoked exactly once for the destruction of each implementation of an interface. Thus, if you were to use one of these functions, you would need to use a static integer variable which would hold the number of instances of implementations of an interface such that the interface's class is destroyed only once (when the integer variable reaches zero).

The above process can be summarized as follows:

Table 2.3. `Interface Finalization`

Invocation time	Function Invoked	Function's parameters
Last call to `g_type_free_instance` for type implementing interface	interface' interface_finalize function	On interface' vtable
Last call to `g_type_free_in-` f	interface' base_finalize function	On interface' vtable

Invocation time	**Function Invoked**	**Function's parameters**
or type implementing interface		

Now that you have read this section, you can forget about it. Please, forget it *as soon as possible*.

Chapter 3. Signals

Closures

Closures are central to the concept of asynchronous signal delivery which is widely used throughout GTK+ and Gnome applications. A Closure is an abstraction, a generic representation of a callback. It is a small structure which contains three objects:

- a function pointer (the callback itself) whose prototype looks like:

  ```
  return_type function_callback (... , gpointer user_data);
  ```

- the user_data pointer which is passed to the callback upon invocation of the closure
- a function pointer which represents the destructor of the closure: whenever the closure's refcount reaches zero, this function will be called before the closure structure is freed.

The GClosure structure represents the common functionality of all closure implementations: there exist a different Closure implementation for each separate runtime which wants to use the GObject type system. 5 The GObject library provides a simple GCClosure type which is a specific implementation of closures to be used with C/ C++ callbacks.

A GClosure provides simple services:

- Invocation (`g_closure_invoke`): this is what closures were created for: they hide the details of callback invocation from the callback invocator.
- Notification: the closure notifies listeners of certain events such as closure invocation, closure invalidation and closure finalization. Listeners can be registered with `g_closure_add_finalize_notifier` (finalization notification), `g_closure_add_invalidate_notifier` (invalidation notification) and `g_closure_add_marshal_guards` (invocation notification). There exist symmetric de-registration functions for finalization and invalidation events (`g_closure_remove_finalize_notifier` and `g_closure_remove_invalidate_notifier`) but not for the invocation process. 6

C Closures

If you are using C or C++ to connect a callback to a given event, you will either use the simple GCClosures which have a pretty minimal API or the even simpler `g_signal_connect` functions (which will be presented a bit later :).

```
GClosure* g_cclosure_new (GCallback        callback_func,
                          gpointer         user_data,
                          GClosureNotify   destroy_data);
GClosure* g_cclosure_new_swap (GCallback        callback_func,
                               gpointer         user_data,
                               GClosureNotify   destroy_data);
GClosure* g_signal_type_cclosure_new (GType  itype,
                                      guint  struct_offset);
```

5 In Practice, Closures sit at the boundary of language runtimes: if you are writing python code and one of your Python callback receives a signal from one of GTK+ widgets, the C code in GTK+ needs to execute your Python code. The Closure invoked by the GTK+ object invokes the Python callback: it behaves as a normal C object for GTK+ and as a normal Python object for python code.
6 Closures are refcounted and notify listeners of their destruction in a two-stage process: the invalidation notifiers are invoked before the finalization notifiers.

`g_cclosure_new` will create a new closure which can invoke the user-provided callback_func with the user-provided user_data as last parameter. When the closure is finalized (second stage of the destruction process), it will invoke the destroy_data function if the user has supplied one.

`g_cclosure_new_swap` will create a new closure which can invoke the user-provided callback_func with the user-provided user_data as first parameter (instead of being the last parameter as with `g_cclosure_new`). When the closure is finalized (second stage of the destruction process), it will invoke the destroy_data function if the user has supplied one.

non-C closures (for the fearless).

As was explained above, Closures hide the details of callback invocation. In C, callback invocation is just like function invocation: it is a matter of creating the correct stack frame for the called function and executing a *call* assembly instruction.

C closure marshallers transform the array of GValues which represent the parameters to the target function into a C-style function parameter list, invoke the user-supplied C function with this new parameter list, get the return value of the function, transform it into a GValue and return this GValue to the marshaller caller.

The following code implements a simple marshaller in C for a C function which takes an integer as first parameter and returns void.

```
g_cclosure_marshal_VOID__INT (GClosure     *closure,
                              GValue       *return_value,
                              guint         n_param_values,
                              const GValue *param_values,
                              gpointer      invocation_hint,
                              gpointer      marshal_data)
{
  typedef void (*GMarshalFunc_VOID__INT) (gpointer     data1,
                                          gint         arg_1,
                                          gpointer     data2);
  register GMarshalFunc_VOID__INT callback;
  register GCClosure *cc = (GCClosure*) closure;
  register gpointer data1, data2;

  g_return_if_fail (n_param_values == 2);

  data1 = g_value_peek_pointer (param_values + 0);
  data2 = closure->data;

  callback = (GMarshalFunc_VOID__INT) (marshal_data ? marshal_data : cc->callback);

  callback (data1,
            g_marshal_value_peek_int (param_values + 1),
            data2);
}
```

Of course, there exist other kinds of marshallers. For example, James Henstridge wrote a generic Python marshaller which is used by all python Closures (a python closure is used to have python-based callback be invoked by the closure invocation process). This python marshaller transforms the input GValue list representing the function parameters into a Python tupple which is the equivalent structure in python (you can look in `pyg_closure_marshal` in `pygtype.c` in the *pygtk* module in Gnome cvs server).

Signals

GObject's signals have nothing to do with standard UNIX signals: they connect arbitrary application-specific events with any number of listeners. For example, in GTK+, every user event (keystroke or mouse move) is received from the X server and generates a GTK+ event under the form of a signal emission on a given object instance.

Each signal is registered in the type system together with the type on which it can be emitted: users of the type are said to *connect* to the signal on a given type instance when they register a closure to be invoked upon the signal emission. Users can also emit the signal by themselves or stop the emission of the signal from within one of the closures connected to the signal.

When a signal is emitted on a given type instance, all the closures connected to this signal on this type instance will be invoked. All the closures connected to such a signal represent callbacks whose signature looks like:

```
return_type function_callback (gpointer instance, ... , gpointer user_data);
```

Signal registration

To register a new signal on an existing type, we can use any of `g_signal_newv`, `g_signal_new_valist` or `g_signal_new` functions:

```
guint                 g_signal_newv         (const gchar        *signal_name,
                                             GType               itype,
                                             GSignalFlags        signal_flags,
                                             GClosure           *class_closure,
                                             GSignalAccumulator  accumulator,
                                             gpointer            accu_data,
                                             GSignalCMarshaller  c_marshaller,
                                             GType               return_type,
                                             guint               n_params,
                                             GType              *param_types);
```

The number of parameters to these functions is a bit intimidating but they are relatively simple:

- signal_name: is a string which can be used to uniquely identify a given signal.
- itype: is the instance type on which this signal can be emitted.
- signal_flags: partly defines the order in which closures which were connected to the signal are invoked.
- class_closure: this is the default closure for the signal: if it is not NULL upon the signal emission, it will be invoked upon this emission of the signal. The moment where this closure is invoked compared to other closures connected to that signal depends partly on the signal_flags.
- accumulator: this is a function pointer which is invoked after each closure has been invoked. If it returns FALSE, signal emission is stopped. If it returns TRUE, signal emission proceeds normally. It is also used to compute the return value of the signal based on the return value of all the invoked closures.
- accumulator_data: this pointer will be passed down to each invocation of the accumulator during emission.
- c_marshaller: this is the default C marshaller for any closure which is connected to this signal.
- return_type: this is the type of the return value of the signal.
- n_params: this is the number of parameters this signal takes.
- param_types: this is an array of GTypes which indicate the type of each parameter of the signal. The length of this array is indicated by n_params.

As you can see from the above definition, a signal is basically a description of the closures which can be connected to this signal and a description of the order in which the closures connected to this signal will be invoked.

Signal connection

If you want to connect to a signal with a closure, you have three possibilities:

- You can register a class closure at signal registration: this is a system-wide operation. i.e.: the class_closure will be invoked during each emission of a given signal on all the instances of the type which supports that signal.
- You can use `g_signal_override_class_closure` which overrides the class_closure of a given type. It is possible to call this function only on a derived type of the type on which the signal was registered. This function is of use only to language bindings.
- You can register a closure with the `g_signal_connect` family of functions. This is an instance-specific operation: the closure will be invoked only during emission of a given signal on a given instance.

It is also possible to connect a different kind of callback on a given signal: emission hooks are invoked whenever a given signal is emitted whatever the instance on which it is emitted. Emission hooks are used for example to get all mouse_clicked emissions in an application to be able to emit the small mouse click sound. Emission hooks are connected with `g_signal_add_emission_hook` and removed with `g_signal_remove_emission_hook`.

Signal emission

Signal emission is done through the use of the `g_signal_emit` family of functions.

```
void                    g_signal_emitv          (const GValue         *instance_and_params,
                                                  guint                 signal_id,
                                                  GQuark                detail,
                                                  GValue               *return_value);
```

- The instance_and_params array of GValues contains the list of input parameters to the signal. The first element of the array is the instance pointer on which to invoke the signal. The following elements of the array contain the list of parameters to the signal.
- signal_id identifies the signal to invoke.
- detail identifies the specific detail of the signal to invoke. A detail is a kind of magic token/argument which is passed around during signal emission and which is used by closures connected to the signal to filter out unwanted signal emissions. In most cases, you can safely set this value to zero. See the section called “The detail argument” for more details about this parameter.
- return_value holds the return value of the last closure invoked during emission if no accumulator was specified. If an accumulator was specified during signal creation, this accumulator is used to calculate the return_value as a function of the return values of all the closures invoked during emission. 7 If no closure is invoked during emission, the return_value is nonetheless initialized to zero/null.

Internally, the GValue array is passed to the emission function proper, `signal_emit_unlocked_R` (implemented in `gsignal.c`). Signal emission can be decomposed in 5 steps:

- *RUN_FIRST*: if the G_SIGNAL_RUN_FIRST flag was used during signal registration and if there exist a class_closure for this signal, the class_closure is invoked. Jump to *EMISSION_HOOK* state.
- *EMISSION_HOOK*: if any emission hook was added to the signal, they are invoked from first to last added. Accumulate return values and jump to *HANDLER_RUN_FIRST* state.
- *HANDLER_RUN_FIRST*: if any closure were connected with the `g_signal_connect` family of functions, and if they are not blocked (with the `g_signal_handler_block` family of functions) they are run here, from first to last connected. Jump to *RUN_LAST* state.
- *RUN_LAST*: if the G_SIGNAL_RUN_LAST flag was set during registration and if a class_closure was set, it is invoked here. Jump to *HANDLER_RUN_LAST* state.

7 James (again!!) gives a few non-trivial examples of accumulators: “ For instance, you may have an accumulator that ignores NULL returns from closures, and only accumulates the non-NULL ones. Another accumulator may try to return the list of values returned by the closures. ”

- *HANDLER_RUN_LAST*: if any closure were connected with the `g_signal_connect_after` family of functions, if they were not invoked during HANDLER_RUN_FIRST and if they are not blocked, they are run here, from first to last connected. Jump to *RUN_CLEANUP* state.
- *RUN_CLEANUP*: if the G_SIGNAL_RUN_CLEANUP flag was set during registration and if a class_closure was set, it is invoked here. Signal emission is completed here.

If, at any point during emission (except in RUN_CLEANUP state), one of the closures or emission hook stops the signal emission with `g_signal_stop`, emission jumps to CLEANUP state.

If, at any point during emission, one of the closures or emission hook emits the same signal on the same instance, emission is restarted from the RUN_FIRST state.

The accumulator function is invoked in all states, after invocation of each closure (except in EMISSION_HOOK and CLEANUP). It accumulates the closure return value into the signal return value and returns TRUE or FALSE. If, at any point, it does not return TRUE, emission jumps to CLEANUP state.

If no accumulator function was provided, the value returned by the last handler run will be returned by `g_signal_emit`.

The *detail* argument

All the functions related to signal emission or signal connection have a parameter named the *detail*. Sometimes, this parameter is hidden by the API but it is always there, under one form or another.

Of the three main connection functions, only one has an explicit detail parameter as a GQuark 8:

```
gulong  g_signal_connect_closure_by_id      (gpointer           instance,
                                               guint           signal_id,
                                               GQuark           detail,
                                               GClosure           *closure,
                                               gboolean            after);
```

The two other functions hide the detail parameter in the signal name identification:

```
gulong  g_signal_connect_closure       (gpointer           instance,
                                               const gchar       *detailed_signal,
                                               GClosure          *closure,
                                               gboolean            after);
gulong  g_signal_connect_data              (gpointer          instance,
                                               const gchar  *detailed_signal,
                                               GCallback   c_handler,
                                               gpointer           data,
                                               GClosureNotify   destroy_data,
                                               GConnectFlags   connect_flags);
```

Their detailed_signal parameter is a string which identifies the name of the signal to connect to. However, the format of this string is structured to look like *signal_name::detail_name*. Connecting to the signal named *notify::cursor_position* will actually connect to the signal named *notify* with the *cursor_position* name. Internally, the detail string is transformed to a GQuark if it is present.

Of the four main signal emission functions, three have an explicit detail parameter as a GQuark again:

```
void                  g_signal_emitv        (const GValue       *instance_and_params,
                                              guint               signal_id,
                                              GQuark              detail,
                                              GValue             *return_value);
void                  g_signal_emit_valist  (gpointer             instance,
```

8A GQuark is an integer which uniquely represents a string. It is possible to transform back and forth between the integer and string representations with the functions `g_quark_from_string` and `g_quark_to_string`.

```
                                             guint                signal_id,
                                             GQuark               detail,
                                             va_list              var_args);
void                   g_signal_emit        (gpointer             instance,
                                             guint                signal_id,
                                             GQuark               detail,
                                             ...);
```

The fourth function hides it in its signal name parameter:

```
void                   g_signal_emit_by_name (gpointer            instance,
                                              const gchar        *detailed_signal,
                                              ...);
```

The format of the detailed_signal parameter is exactly the same as the format used by the `g_signal_connect` functions: *signal_name::detail_name*.

If a detail is provided by the user to the emission function, it is used during emission to match against the closures which also provide a detail. The closures which provided a detail will not be invoked (even though they are connected to a signal which is being emitted) if their detail does not match the detail provided by the user.

This completely optional filtering mechanism is mainly used as an optimization for signals which are often emitted for many different reasons: the clients can filter out which events they are interested into before the closure's marshalling code runs. For example, this is used extensively by the *notify* signal of GObject: whenever a property is modified on a GObject, instead of just emitting the *notify* signal, GObject associates as a detail to this signal emission the name of the property modified. This allows clients who wish to be notified of changes to only one property to filter most events before receiving them.

As a simple rule, users can and should set the detail parameter to zero: this will disable completely this optional filtering.

Chapter 4. GObject: what brings everything together.

The two previous chapters discussed the details of Glib's Dynamic Type System and its signal control system. The GObject library also contains an implementation for a base fundamental type named GObject.

GObject is a fundamental classed instantiable type. It implements:

- Memory management with reference counting
- Construction/Destruction of instances
- Generic per-object properties with set/get function pairs
- Easy use of signals

All the GTK+ objects and all of the objects in Gnome libraries which use the glib type system inherit from GObject which is why it is important to understand the details of how it works.

Object instanciation

The `g_object_new` family of functions can be used to instantiate any GType which inherits from the GObject base type. All these functions make sure the class and instance structures have been correctly initialized by glib's type system and then invoke at one point or another the constructor class method which is used to:

- Allocate and clear memory through `g_type_create_instance`,
- Initialize the object' instance with the construction properties.

Although one can expect all class and instance members (except the fields pointing to the parents) to be set to zero, some consider it good practice to explicitly set them.

Objects which inherit from GObject are allowed to override this constructor class method: they should however chain to their parent constructor method before doing so:

```
  GObject*   (*constructor)     (GType                  type,
                                 guint                  n_construct_properties,
                                 GObjectConstructParam *construct_properties);
```

The example below shows how MamanBar overrides the parent's constructor:

```
#define MAMAN_TYPE_BAR            (maman_bar_get_type ())
#define MAMAN_BAR(obj)            (G_TYPE_CHECK_INSTANCE_CAST ((obj), MAMAN_TYPE_BAR, Ma
#define MAMAN_BAR_CLASS(klass)    (G_TYPE_CHECK_CLASS_CAST ((klass), MAMAN_TYPE_BAR, Mam
#define MAMAN_IS_BAR(obj)    (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MAMAN_TYPE_BAR))
#define MAMAN_IS_BAR_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass), MAMAN_TYPE_BAR))
#define MAMAN_BAR_GET_CLASS(obj)  (G_TYPE_INSTANCE_GET_CLASS ((obj), MAMAN_TYPE_BAR, Ma

typedef struct _MamanBar MamanBar;
typedef struct _MamanBarClass MamanBarClass;

struct _MamanBar {
  GObject parent;
  /* instance members */
};

struct _MamanBarClass {
  GObjectClass parent;

  /* class members */
};

/* used by MAMAN_TYPE_BAR */
GType maman_bar_get_type (void);

static GObject *
maman_bar_constructor (GType                  type,
                       guint                  n_construct_properties,
                       GObjectConstructParam *construct_properties)
{
  GObject *obj;

  {
    /* Invoke parent constructor. */
    MamanBarClass *klass;
    GObjectClass *parent_class;
    klass = MAMAN_BAR_CLASS (g_type_class_peek (MAMAN_TYPE_BAR));
    parent_class = G_OBJECT_CLASS (g_type_class_peek_parent (klass));
    obj = parent_class->constructor (type,
                                     n_construct_properties,
                                     construct_properties);
  }

  /* do stuff. */

  return obj;
}

static void
maman_bar_instance_init (GTypeInstance   *instance,
                         gpointer         g_class)
{
  MamanBar *self = (MamanBar *)instance;
  /* do stuff */
}

static void
maman_bar_class_init (gpointer g_class,
                      gpointer g_class_data)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);
  MamanBarClass *klass = MAMAN_BAR_CLASS (g_class);

  gobject_class->constructor = maman_bar_constructor;
}

GType maman_bar_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBarClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      maman_bar_class_init,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBar),
      0,      /* n_preallocs */
      maman_bar_instance_init    /* instance_init */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBarType",
                                   &info, 0);
  }
  return type;
```

```
}
```

If the user instantiates an object MamanBar with:

```
MamanBar *bar = g_object_new (MAMAN_TYPE_BAR, NULL);
```

If this is the first instantiation of such an object, the `maman_b_class_init` function will be invoked after any `maman_b_base_class_init` function. This will make sure the class structure of this new object is correctly initialized. Here, `maman_bar_class_init` is expected to override the object's class methods and setup the class' own methods. In the example above, the constructor method is the only overridden method: it is set to `maman_bar_constructor`.

Once `g_object_new` has obtained a reference to an initialized class structure, it invokes its constructor method to create an instance of the new object. Since it has just been overridden by `maman_bar_class_init` to `maman_bar_constructor`, the latter is called and, because it was implemented correctly, it chains up to its parent's constructor. The problem here is how we can find the parent constructor. An approach (used in GTK+ source code) would be to save the original constructor in a static variable from `maman_bar_class_init` and then to re-use it from `maman_bar_constructor`. This is clearly possible and very simple but I was told it was not nice and the prefered way is to use the `g_type_class_peek` and `g_type_class_peek_parent` functions.

Finally, at one point or another, `g_object_constructor` is invoked by the last constructor in the chain. This function allocates the object's instance' buffer through `g_type_create_instance` which means that the instance_init function is invoked at this point if one was registered. After instance_init returns, the object is fully initialized and should be ready to answer any user-request. When `g_type_create_instance` returns, `g_object_constructor` sets the construction properties (ie: the properties which were given to `g_object_new`) and returns to the user's constructor which is then allowed to do useful instance initialization...

The process described above might seem a bit complicated (it *is* actually overly complicated in my opinion..) but it can be summarized easily by the table below which lists the functions invoked by `g_object_new` and their order of invocation.

The array below lists the functions invoked by `g_object_new` and their order of invocation:

Table 4.1. `g_object_new`

Invocation time	Function Invoked	Function's parameters
First call to `g_object_new` for target type	target type's base_init function	On the inheritance tree of classes from fundamental type to target type. base_init is invoked once for each class structure.
First call to `g_object_new` for target type	target type's class_init function	On target type's class structure
First call to `g_object_new` for target type	interface' base_init function	On interface' vtable
First call to `g_obje`	interface' interface_init	On interface' vtable

Invocation time	**Function Invoked**	**Function's parameters**
`ct_new` for target type	function	
Each call to `g_object_new` for target type	target type's class constructor method: GObjectClass->constructor	On object's instance
Each call to `g_object_new` for target type	type's instance_init function	On the inheritance tree of classes from fundamental type to target type. the instance_init provided for each type is invoked once for each instance structure.

Readers should feel concerned about one little twist in the order in which functions are invoked: while, technically, the class' constructor method is called *before* the GType's instance_init function (since `g_type_create_instance` which calls instance_init is called by `g_object_constructor` which is the top-level class constructor method and to which users are expected to chain to), the user's code which runs in a user-provided constructor will always run *after* GType's instance_init function since the user-provided constructor *must* (you've been warned) chain up *before* doing anything useful.

Object memory management

The memory-management API for GObjects is a bit complicated but the idea behind it is pretty simple: the goal is to provide a flexible model based on reference counting which can be integrated in applications which use or require different memory management models (such as garbage collection, aso...). The methods which are used to manipulate this reference count are described below.

```
/*
  Refcounting
*/
gpointer    g_object_ref                          (gpointer        object);
void        g_object_unref                        (gpointer        object);

/*
  Weak References
*/
typedef void (*GWeakNotify)         (gpointer      data,
                                         GObject      *where_the_object_was);
void     g_object_weak_ref                   (GObject       *object,
                                               GWeakNotify     notify,
                                               gpointer        data);
void     g_object_weak_unref                   (GObject       *object,
                                               GWeakNotify     notify,
                                               gpointer        data);
void        g_object_add_weak_pointer          (GObject        *object,
                                               gpointer       *weak_pointer_location);
void        g_object_remove_weak_pointer       (GObject        *object,
                                               gpointer       *weak_pointer_location);
/*
  Cycle handling
*/
void        g_object_run_dispose       (GObject       *object);
```

Reference count

The functions `g_object_ref`/`g_object_unref` respectively increase and decrease the reference count. None of these function is thread-safe. The reference count is, unsurprisingly, initialized to one by `g_object_new` which means that the caller is currenly the sole owner of the newly-created reference. When the reference count reaches zero, that is, when `g_object_unref` is called by the last client holding a reference to the object, the *dispose* and the *finalize* class methods are invoked.

Finally, after *finalize* is invoked, `g_type_free_instance` is called to free the object instance. Depending on the memory allocation policy decided when the type was registered (through one of the `g_type_register_*` functions), the object's instance memory will be freed or returned to the object pool for this type. Once the object has been freed, if it was the last instance of the type, the type's class will be destroyed as described in the section called “Instantiable classed types: objects” and the section called “Non-instantiable classed types: Interfaces.”.

The table below summarizes the destruction process of a GObject:

Table 4.2. `g_object_unref`

Invocation time	**Function Invoked**	**Function's parameters**
Last call to `g_object_unref` for an instance of target type	target type's dispose class function	GObject instance
Last call to `g_object_unref` for an instance of target type	target type's finalize class function	GObject instance
Last call to `g_object_unref` for the last instance of target type	interface' interface_finalize function	On interface' vtable
Last call to `g_object_unref`for the last instance of target type	interface' base_finalize function	On interface' vtable
Last call to `g_object_unr`	target type's class_finalize func-	On target type's class structure

Invocation time	Function Invoked	Function's parameters
`ef` for the last instance of target type	tion	
Last call to `g_object_unref` for the last instance of target type	type's base_finalize function	On the inheritance tree of classes from fundamental type to target type. base_init is invoked once for each class structure.

Weak References

Weak References are used to monitor object finalization: `g_object_weak_ref` adds a monitoring callback which does not hold a reference to the object but which is invoked when the object runs its dispose method. As such, each weak ref can be invoked more than once upon object finalization (since dispose can run more than once during object finalization).

`g_object_weak_unref` can be used to remove a monitoring callback from the object.

Weak References are also used to implement `g_object_add_weak_pointer` and `g_object_remove_weak_pointer`. These functions add a weak reference to the object they are applied to which makes sure to nullify the pointer given by the user when object is finalized.

Reference counts and cycles

Note: the following section was inspired by James Henstridge. I guess this means that all praise and all curses will be directly forwarded to him.

GObject's memory management model was designed to be easily integrated in existing code using garbage collection. This is why the destruction process is split in two phases: the first phase, executed in the dispose handler is supposed to release all references to other member objects. The second phase, executed by the finalize handler is supposed to complete the object's destruction process. Object methods should be able to run without program error (that is, without segfault :) in-between the two phases.

This two-step destruction process is very useful to break reference counting cycles. While the detection of the cycles is up to the external code, once the cycles have been detected, the external code can invoke `g_object_dispose` which will indeed break any existing cycles since it will run the dispose handler associated to the object and thus release all references to other objects.

Attentive readers might now have understood one of the rules about the dispose handler we stated a bit sooner: the dispose handler can be invoked multiple times. Let's say we have a reference count cycle: object A references B which itself references object A. Let's say we have detected the cycle and we want to destroy the two objects. One way to do this would be to invoke `g_object_dispose` on one of the objects.

If object A releases all its references to all objects, this means it releases its reference to object B. If object B was not owned by anyone else, this is its last reference count which means this last unref runs B's dispose handler which, in turn, releases B's reference on object A. If this is A's last reference count, this last unref runs A's dispose handler which is running for the second time before A's finalize handler is invoked !

The above example, which might seem a bit contrived can really happen if your GObject's are being by language bindings. I would thus suggest the rules stated above for object destruction are closely followed. Otherwise, *Bad Bad Things* will happen.

Object properties

One of GObject's nice features is its generic get/set mechanism for object properties. When an object is instanciated, the object's class_init handler should be used to register the object's properties with `g_object_class_install_property` (implemented in `gobject.c`).

The best way to understand how object properties work is by looking at a real example on how it is used:

```
/************************************************/
/* Implementation                               */
/************************************************/

enum {
  MAMAN_BAR_CONSTRUCT_NAME = 1,
  MAMAN_BAR_PAPA_NUMBER,
};

static void
maman_bar_instance_init (GTypeInstance   *instance,
                         gpointer         g_class)
{
  MamanBar *self = (MamanBar *)instance;
}


static void
maman_bar_set_property (GObject      *object,
                        guint         property_id,
                        const GValue *value,
                        GParamSpec   *pspec)
{
  MamanBar *self = (MamanBar *) object;

  switch (property_id) {
  case MAMAN_BAR_CONSTRUCT_NAME: {
    g_free (self->private->name);
    self->private->name = g_value_dup_string (value);
    g_print ("maman: %s\n",self->private->name);
  }
    break;
  case MAMAN_BAR_PAPA_NUMBER: {
    self->private->papa_number = g_value_get_uchar (value);
    g_print ("papa: %u\n",self->private->papa_number);
  }
    break;
  default:
    /* We don't have any other property... */
    G_OBJECT_WARN_INVALID_PROPERTY_ID(object,property_id,pspec);
    break;
  }
}

static void
maman_bar_get_property (GObject      *object,
                        guint         property_id,
                        GValue       *value,
                        GParamSpec   *pspec)
{
  MamanBar *self = (MamanBar *) object;

  switch (property_id) {
  case MAMAN_BAR_CONSTRUCT_NAME: {
    g_value_set_string (value, self->private->name);
  }
    break;
  case MAMAN_BAR_PAPA_NUMBER: {
    g_value_set_uchar (value, self->private->papa_number);
  }
    break;
  default:
    /* We don't have any other property... */
    G_OBJECT_WARN_INVALID_PROPERTY_ID(object,property_id,pspec);
    break;
  }
}

static void
maman_bar_class_init (gpointer g_class,
                      gpointer g_class_data)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);
  MamanBarClass *klass = MAMAN_BAR_CLASS (g_class);
  GParamSpec *pspec;

  gobject_class->set_property = maman_bar_set_property;
  gobject_class->get_property = maman_bar_get_property;

  pspec = g_param_spec_string ("maman-name",
                               "Maman construct prop",
                               "Set maman's name",
                               "no-name-set" /* default value */,
                               G_PARAM_CONSTRUCT_ONLY | G_PARAM_READWRITE);
  g_object_class_install_property (gobject_class,
                                   MAMAN_BAR_CONSTRUCT_NAME,
                                   pspec);

  pspec = g_param_spec_uchar ("papa-number",
                              "Number of current Papa",
                              "Set/Get papa's number",
                              0  /* minimum value */,
                              10 /* maximum value */,
                              2  /* default value */,
                              G_PARAM_READWRITE);
  g_object_class_install_property (gobject_class,
                                   MAMAN_BAR_PAPA_NUMBER,
                                   pspec);
}

/************************************************/
/* Use                                          */
/************************************************/

GObject *bar;
GValue val = {0,};
bar = g_object_new (MAMAN_TYPE_SUBBAR, NULL);
g_value_init (&val, G_TYPE_CHAR);
g_value_set_char (&val, 11);
g_object_set_property (G_OBJECT (bar), "papa-number", &val);
```

The client code just above looks simple but a lot of things happen under the hood:

`g_object_set_property` first ensures a property with this name was registered in bar's class_init handler. If so, it calls `object_set_property` which first walks the class hierarchy, from bottom, most derived type, to top, fundamental type to find the class which registered that property. It then tries to convert the user-provided GValue into a GValue whose type if that of the associated property.

If the user provides a signed char GValue, as is shown here, and if the object's property was registered as an unsigned int, `g_value_transform` will try to transform the input signed char into an unsigned int. Of course, the success of the transformation depends on the availability of the required transform function. In practice, there will almost always be a transformation 9 which matches and conversion will be caried out if needed.

After transformation, the GValue is validated by `g_param_value_validate` which makes sure the user's data stored in the GValue matches the characteristics specified by the property's GParamSpec. Here, the

9Its behaviour might not be what you expect but it is up to you to actually avoid relying on these transformations.

GParamSpec we provided in class_init has a validation function which makes sure that the GValue contains a value which respects the minimum and maximum bounds of the GParamSpec. In the example above, the client's GValue does not respect these constraints (it is set to 11, while the maximum is 10). As such, the `g_object_set_property` function will return with an error.

If the user's GValue had been set to a valid value, `object_set_property` would have proceeded with calling the object's set_property class method. Here, since our implementation of Foo did override this method, the code path would jump to `foo_set_property` after having retrieved from the GParamSpec the *param_id* 10 which had been stored by `g_object_class_install_property`.

Once the property has been set by the object's set_property class method, the code path returns to `g_object_set_property` which calls `g_object_notify_queue_thaw`. This function makes sure that the "notify" signal is emitted on the object's instance with the changed property as parameter unless notifications were frozen by `g_object_freeze_notify`.

`g_object_thaw_notify` can be used to re-enable notification of property modifications through the "notify" signal. It is important to remember that even if properties are changed while property change notification is frozen, the "notify" signal will be emitted once for each of these changed properties as soon as the property change notification is thawn: no property change is lost for the "notify" signal. Signal can only be delayed by the notification freezing mechanism.

Accessing multiple properties at once

It is interesting to note that the `g_object_set` and `g_object_set_valist` (vararg version) functions can be used to set multiple properties at once. The client code shown above can then be re-written as:

```
MamanBar *foo;
foo = /* */;
g_object_set (G_OBJECT (foo),
              "papa-number", 2,
              "maman-name", "test",
              NULL);
```

The code above will trigger one notify signal emission for each property modified.

Of course, the _get versions are also available: `g_object_get` and `g_object_get_valist` (vararg version) can be used to get numerous properties at once.

Really attentive readers now understand how `g_object_new`, `g_object_newv` and `g_object_new_valist` work: they parse the user-provided variable number of parameters and invoke `g_object_set` on each pair of parameters only after the object has been successfully constructed. Of course, the "notify" signal will be emitted for each property set.

10 It should be noted that the param_id used here need only to uniquely identify each GParamSpec within the FooClass such that the switch used in the set and get methods actually works. Of course, this locally-unique integer is purely an optimization: it would have been possible to use a set of *if (strcmp (a, b) == 0) {} else if (strcmp (a, b) == 0) {}* statements.

Chapter 5. How To ?

This chapter tries to answer the real-life questions of users and presents the most common scenario use-cases I could come up with. The use-cases are presented from most likely to less likely.

How To define and implement a new GObject ?

Clearly, this is one of the most common question people ask: they just want to crank code and implement a subclass of a GObject. Sometimes because they want to create their own class hierarchy, sometimes because they want to subclass one of GTK+'s widget. This chapter will focus on the implementation of a subtype of GObject. The sample source code associated to this section can be found in the documentation's source tarball, in the `sample/gobject` directory:

- `maman-bar.{h|c}`: this is the source for a object which derives from GObject and which shows how to declare different types of methods on the object.
- `maman-subbar.{h|c}`: this is the source for a object which derives from MamanBar and which shows how to override some of its parent's methods.
- `maman-foo.{h|c}`: this is the source for an object which derives from GObject and which declares a signal.
- `test.c`: this is the main source which instantiates an instance of type and exercises their API.

Boilerplate header code

The first step before writing the code for your GObject is to write the type's header which contains the needed type, function and macro definitions. Each of these elements is nothing but a convention which is followed not only by GTK+'s code but also by most users of GObject. If you feel the need not to obey the rules stated below, think about it twice:

- If your users are a bit accustomed to GTK+ code or any Glib code, they will be a bit surprised and getting used to the conventions you decided upon will take time (money) and will make them grumpy (not a good thing)
- You must assess the fact that these conventions might have been designed by both smart and experienced people: maybe they were at least partly right. Try to put your ego aside.

Pick a name convention for your headers and source code and stick to it:

- use a dash to separate the prefix from the typename: `maman-bar.h` and `maman-bar.c` (this is the convention used by Nautilus and most Gnome libraries).
- use an underscore to separate the prefix from the typename: `maman_bar.h` and `maman_bar.c`.
- Do not separate the prefix from the typename: `mamanbar.h` and `mamanbar.c`. (this is the convention used by GTK+)

I personally like the first solution better: it makes reading file names easier for those with poor eyesight like me.

When you need some private (internal) declarations in several (sub)classes, you can define them in a private header file which is often named by appending the *private* keyword to the public header name. For example, one could use `maman-bar-private.h`, `maman_bar_private.h` or `mamanbarprivate.h`. Typically, such private header files are not installed.

The basic conventions for any header which exposes a GType are described in the section called “Conventions”.

Most GObject-based code also obeys onf of the following conventions: pick one and stick to it.

- If you want to declare a type named bar with prefix maman, name the type instance `MamanBar` and its class `MamanBarClass` (name is case-sensitive). It is customary to declare them with code similar to the following:

```
/*
 * Copyright/Licensing information.
 */

#ifndef MAMAN_BAR_H
#define MAMAN_BAR_H

/*
 * Potentially, include other headers on which this header depends.
 */


/*
 * Type macros.
 */

typedef struct _MamanBar MamanBar;
typedef struct _MamanBarClass MamanBarClass;

struct _MamanBar {
  GObject parent;
  /* instance members */
};

struct _MamanBarClass {
  GObjectClass parent;
  /* class members */
};

/* used by MAMAN_BAR_TYPE */
GType maman_bar_get_type (void);

/*
 * Method definitions.
 */

#endif
```

- Most GTK+ types declare their private fields in the public header with a /* private */ comment, relying on their user's intelligence not to try to play with these fields. Fields not marked private are considered public by default. The /* protected */ comment (same semantics as those of C++) is also used, mainly in the GType library, in code written by Tim Janik.

```
struct _MamanBar {
  GObject parent;

  /* private */
  int hsize;
};
```

- All of Nautilus code and a lot of Gnome libraries use private indirection members, as described by Herb Sutter in his Pimpl articles (see Compilation Firewalls [http://www.gotw.ca/gotw/024.htm] and The Fast Pimpl Idiom [http://www.gotw.ca/gotw/028.htm] : he summarizes the different issues better than I will).

```
typedef struct _MamanBarPrivate MamanBarPrivate;
struct _MamanBar {
  GObject parent;
  /* private */
  MamanBarPrivate *priv;
};
```

Note

Do not call this `private`, as that is a registered c++ keyword.

The private structure is then defined in the .c file, instantiated in the object's `init` function and destroyed in the object's `finalize` function.

```
static void maman_bar_finalize(GObject *object) {
  MamanBar *self = MAMAN_BAR (object);
  ...
  g_free (self->priv);
}

static void maman_bar_init(GTypeInstance *instance, gpointer g_class) {
  MamanBar *self = MAMAN_BAR (instance);
  self->priv = g_new0(MamanBarPrivate,1);
  ...
}
```

- A similar alternative, available since Glib version 2.4, is to define a private structure in the .c file, declare it as a private structure in `class_init` using `g_type_class_add_private` and declare a macro to allow convenient access to this structure. A private structure will then be attached to each newly created object by the GObject system. You dont need to free or allocate the private structure, only the objects or pointers that it may contain.

```
typedef struct _MamanBarPrivate MamanBarPrivate;

struct _MamanBarPrivate {
  int private_field;
};

#define MAMAN_BAR_GET_PRIVATE(o) (G_TYPE_INSTANCE_GET_PRIVATE ((o), MAMAN_BAR_TYPE, M

static void
maman_bar_class_init (MamanBarClass *klass)
{
  ...
  g_type_class_add_private (klass, sizeof (MamanBarPrivate));
  ...
}

static int
maman_bar_get_private_field (MamanBar *self)
{
  MamanBarPrivate *priv = MAMAN_BAR_GET_PRIVATE (self);

  return priv->private_field;
}
```

Finally, there are different header include conventions. Again, pick one and stick to it. I personally use indifferently any of the two, depending on the codebase I work on: the rule is consistency.

- Some people add at the top of their headers a number of #include directives to pull in all the headers needed to compile client code. This allows client code to simply #include "maman-bar.h".

- Other do not #include anything and expect the client to #include themselves the headers they need before including your header. This speeds up compilation because it minimizes the amount of pre-processor work. This can be used in conjunction with the re-declaration of certain unused types in the client code to minimize compile-time dependencies and thus speed up compilation.

Boilerplate code

In your code, the first step is to #include the needed headers: depending on your header include strategy, this can be as simple as #include "maman-bar.h" or as complicated as tens of #include lines ending with #include "maman-bar.h":

```
/*
 * Copyright information
 */

#include "maman-bar.h"

/* If you use Pimpls, include the private structure
 * definition here. Some people create a maman-bar-private.h header
 * which is included by the maman-bar.c file and which contains the
 * definition for this private structure.
 */
struct _MamanBarPrivate {
  int member_1;
  /* stuff */
};

/*
 * forward definitions
 */
```

Implement `maman_bar_get_type` and make sure the code compiles:

```
GType
maman_bar_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBarClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      NULL,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBar),
      0,      /* n_preallocs */
      NULL    /* instance_init */
      };
      type = g_type_register_static (G_TYPE_OBJECT,
                                     "MamanBarType",
                                     &info, 0);
    }
    return type;
}
```

Object Construction

People often get confused when trying to construct their GObjects because of the sheer number of different ways to hook into the objects's construction process: it is difficult to figure which is the *correct*, recommended way.

Table 4.1, “g_object_new” shows what user-provided functions are invoked during object instanciation and in which order they are invoked. A user looking for the equivalent of the simple C++ constructor function should use the instance_init method. It will be invoked after all the parent's instance_init functions have been invoked. It cannot take arbitrary construction parameters (as in C++) but if your object needs arbitrary parameters to complete initialization, you can use construction properties.

Construction properties will be set only after all instance_init functions have run. No object reference will be returned to the client of `g_object_new>` until all the construction properties have been set.

As such, I would recommend writing the following code first:

```
static void
maman_bar_init (GTypeInstance   *instance,
                gpointer         g_class)
{
  MamanBar *self = (MamanBar *)instance;
  self->private = g_new0 (MamanBarPrivate, 1);

  /* initialize all public and private members to reasonable default values. */
  /* If you need specific consruction properties to complete initialization,
   * delay initialization completion until the property is set.
   */
}
```

And make sure that you set `maman_bar_init` as the type's instance_init function in `maman_bar_get_type`. Make sure the code builds and runs: create an instance of the object and make sure `maman_bar_init` is called (add a `g_print` call in it).

Now, if you need special construction properties, install the properties in the class_init function, override the set and get methods and implement the get and set methods as described in the section called “Object properties”. Make sure that these properties use a construct only GParamSpec by setting the param spec's flag field to G_PARAM_CONSTRUCT_ONLY: this helps GType ensure that these properties are not set again later by malicious user code.

```
static void
bar_class_init (MamanBarClass *klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
  GParamSpec *maman_param_spec;

  gobject_class->set_property = bar_set_property;
  gobject_class->get_property = bar_get_property;

  maman_param_spec = g_param_spec_string ("maman",
                                          "Maman construct prop",
                                          "Set maman's name",
                                          "no-name-set" /* default value */,
                                          G_PARAM_CONSTRUCT_ONLY |G_PARAM_READWRITE);

  g_object_class_install_property (gobject_class,
                                   PROP_MAMAN,
                                   maman_param_spec);
}
```

If you need this, make sure you can build and run code similar to the code shown above. Make sure your construct properties can set correctly during construction, make sure you cannot set them afterwards and make sure that if your users do not call `g_object_new` with the required construction properties, these will be initialized with the default values.

I consider good taste to halt program execution if a construction property is set its default value. This allows you to catch client code which does not give a reasonable value to the construction properties. Of course, you are free to disagree but you should have a good reason to do so.

Some people sometimes need to construct their object but only after the construction properties have been set.

This is possible through the use of the constructor class method as described in the section called “Object instanciation”. However, I have yet to see *any* reasonable use of this feature. As such, to initialize your object instances, use by default the base_init function and construction properties.

Object Destruction

Again, it is often difficult to figure out which mechanism to use to hook into the object's destruction process: when the last `g_object_unref` function call is made, a lot of things happen as described in Table 4.2, “g_object_unref”.

The destruction process of your object must be split is two different phases: you must override both the dispose and the finalize class methods.

```
struct _MamanBarPrivate {
  gboolean dispose_has_run;
};

static GObjectClass parent_class = NULL;

static void
bar_dispose (GObject *obj)
{
  MamanBar *self = (MamanBar *)obj;

  if (self->private->dispose_has_run) {
   /* If dispose did already run, return. */
    return;
  }
  /* Make sure dispose does not run twice. */
  object->private->dispose_has_run = TRUE;

  /*
   * In dispose, you are supposed to free all types referenced from this
   * object which might themselves hold a reference to self. Generally,
   * the most simple solution is to unref all members on which you own a
   * reference.
   */

   /* Chain up to the parent class */
   G_OBJECT_CLASS (parent_class)->dispose (obj);
}

static void
bar_finalize (GObject *obj)
{
  MamanBar *self = (MamanBar *)obj;

  /*
   * Here, complete object destruction.
   * You might not need to do much...
   */
  g_free (self->private);

   /* Chain up to the parent class */
   G_OBJECT_CLASS (parent_class)->finalize (obj);
}

static void
bar_class_init (BarClass *klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);

  gobject_class->dispose = bar_dispose;
  gobject_class->finalize = bar_finalize;
}
```

```
static void
maman_bar_init (GTypeInstance   *instance,
                gpointer         g_class)
{
  MamanBar *self = (MamanBar *)instance;
  self->private = g_new0 (MamanBarPrivate, 1);
  self->private->dispose_has_run = FALSE;

  parent_class = g_type_class_peek_parent (klass);
}
```

Add similar code to your GObject, make sure the code still builds and runs: dispose and finalize must be called during the last unref. It is possible that object methods might be invoked after dispose is run and before finalize runs. GObject does not consider this to be a program error: you must gracefully detect this and neither crash nor warn the user. To do this, you need something like the following code at the start of each object method, to make sure the object's data is still valid before manipulating it:

```
if (self->private->dispose_has_run) {
  /* Dispose has run. Data is not valid anymore. */
  return;
}
```

Object methods

Just as with C++, there are many different ways to define object methods and extend them: the following list and sections draw on C++ vocabulary. (Readers are expected to know basic C++ buzzwords. Those who have not had to write C++ code recently can refer to e.g. http://www.cplusplus.com/doc/tutorial/ to refresh their memories.)

- non-virtual public methods,
- virtual public methods and
- virtual private methods

Non-virtual public methods

These are the simplest: you want to provide a simple method which can act on your object. All you need to do is to provide a function prototype in the header and an implementation of that prototype in the source file.

```
/* declaration in the header. */
void maman_bar_do_action (MamanBar *self, /* parameters */);
/* implementation in the source file */
void maman_bar_do_action (MamanBar *self, /* parameters */)
{
  /* do stuff here. */
}
```

There is really nothing scary about this.

Virtual public methods

This is the preferred way to create polymorphic GObjects. All you need to do is to define the common method and its class function in the public header, implement the common method in the source file and re-implement the class function in each object which inherits from you.

```
/* declaration in maman-bar.h. */
struct _MamanBarClass {
  GObjectClass parent;

  /* stuff */
  void (*do_action) (MamanBar *self, /* parameters */);
};
void maman_bar_do_action (MamanBar *self, /* parameters */);
/* implementation in maman-bar.c */
void maman_bar_do_action (MamanBar *self, /* parameters */)
{
  MAMAN_BAR_GET_CLASS (self)->do_action (self, /* parameters */);
}
```

The code above simply redirects the do_action call to the relevant class function. Some users, concerned about performance, do not provide the `maman_bar_do_action` wrapper function and require users to de-reference the class pointer themselves. This is not such a great idea in terms of encapsulation and makes it difficult to change the object's implementation afterwards, should this be needed.

Other users, also concerned by performance issues, declare the `maman_bar_do_action` function inline in the header file. This, however, makes it difficult to change the object's implementation later (although easier than requiring users to directly de-reference the class function) and is often difficult to write in a portable way (the *inline* keyword is not part of the C standard).

In doubt, unless a user shows you hard numbers about the performance cost of the function call, just `maman_bar_do_action` in the source file.

Please, note that it is possible for you to provide a default implementation for this class method in the object's class_init function: initialize the klass->do_action field to a pointer to the actual implementation. You can also make this class method pure virtual by initializing the klass->do_action field to NULL:

```
static void
maman_bar_real_do_action_two (MamanBar *self, /* parameters */)
{
  /* Default implementation for the virtual method. */
}

static void
maman_bar_class_init (BarClass *klass)
{
  /* pure virtual method: mandates implementation in children. */
  klass->do_action_one = NULL;
  /* merely virtual method. */
  klass->do_action_two = maman_bar_real_do_action_two;
}

void maman_bar_do_action_one (MamanBar *self, /* parameters */)
{
  MAMAN_BAR_GET_CLASS (self)->do_action_one (self, /* parameters */);
}
void maman_bar_do_action_two (MamanBar *self, /* parameters */)
{
  MAMAN_BAR_GET_CLASS (self)->do_action_two (self, /* parameters */);
}
```

Virtual private Methods

These are very similar to Virtual Public methods. They just don't have a public function to call the function directly. The header file contains only a declaration of the class function:

```
/* declaration in maman-bar.h. */
struct _MamanBarClass {
  GObjectClass parent;
  /* stuff */
  void (*helper_do_specific_action) (MamanBar *self, /* parameters */);
};
void maman_bar_do_any_action (MamanBar *self, /* parameters */);
```

These class functions are often used to delegate part of the job to child classes:

```
/* this accessor function is static: it is not exported outside of this file. */
static void
maman_bar_do_specific_action (MamanBar *self, /* parameters */)
{
  MAMAN_BAR_GET_CLASS (self)->do_specific_action (self, /* parameters */);
}

void maman_bar_do_any_action (MamanBar *self, /* parameters */)
{
  /* random code here */

  /*
   * Try to execute the requested action. Maybe the requested action cannot be implemen
   * here. So, we delegate its implementation to the child class:
   */
  maman_bar_do_specific_action (self, /* parameters */);

  /* other random code here */
}
```

Again, it is possible to provide a default implementation for this private virtual class function:

```
static void
maman_bar_class_init (MamanBarClass *klass)
{
  /* pure virtual method: mandates implementation in children. */
  klass->do_specific_action_one = NULL;
  /* merely virtual method. */
  klass->do_specific_action_two = maman_bar_real_do_specific_action_two;
}
```

Children can then implement the subclass with code such as:

```
static void
maman_bar_subtype_class_init (MamanBarSubTypeClass *klass)
{
  MamanBarClass *bar_class = MAMAN_BAR_CLASS (klass);
  /* implement pure virtual class function. */
  bar_class->do_specific_action_one = maman_bar_subtype_do_specific_action_one;
}
```

Chaining up

Chaining up is often loosely defined by the following set of conditions:

- Parent class A defines a public virtual method named `foo` and provides a default implementation.
- Child class B re-implements method `foo`.
- In the method B::foo, the child class B calls its parent class method A::foo.

There are many uses to this idiom:

- You need to change the behaviour of a class without modifying its code. You create a subclass to inherit its implementation, re-implement a public virtual method to modify the behaviour slightly and chain up to ensure that the previous behaviour is not really modifed, just extended.
- You are lazy, you have access to the source code of the parent class but you don't want to modify it to add method calls to new specialized method calls: it is faster to hack the child class to chain up than to modify the parent to call down.
- You need to implement the Chain Of Responsability pattern: each object of the inheritance tree chains up to its parent (typically, at the begining or the end of the method) to ensure that they each handler is run in turn.

I am personally not really convinced any of the last two uses are really a good idea but since this programming idiom is often used, this section attemps to explain how to implement it.

To explicitely chain up to the implementation of the virtual method in the parent class, you first need a handle to the original parent class structure. This pointer can then be used to access the original class function pointer and invoke it directly. [11]

The function `g_type_class_peek_parent` is used to access the original parent class structure. Its input is a pointer to the class of the derived object and it returns a pointer to the original parent class structure. The code below shows how you could use it:

```
static void
b_method_to_call (B *obj, int a)
{
  BClass *klass;
  AClass *parent_class;
  klass = B_GET_CLASS (obj);
  parent_class = g_type_class_peek_parent (klass);

  /* do stuff before chain up */
  parent_class->method_to_call (obj, a);
  /* do stuff after chain up */
}
```

A lot of people who use this idiom in GTK+ store the parent class structure pointer in a global static variable to avoid the costly call to `g_type_class_peek_parent` for each function call. Typically, the class_init callback initializes the global static variable. `gtk/gtkhscale.c` does this.

How To define and implement Interfaces ?

How To define Interfaces ?

The bulk of interface definition has already been shown in the section called “Non-instantiable classed types: Interfaces.” but I feel it is needed to show exactly how to create an interface. The sample source code associated to this section can be found in the documentation's source tarball, in the `sample/interface/maman-ibaz.{h|c}` file.

As above, the first step is to get the header right:

```
#ifndef MAMAN_IBAZ_H
```

[11]The *original* adjective used in this sentence is not innocuous. To fully understand its meaning, you need to recall how class structures are initialized: for each object type, the class structure associated to this object is created by first copying the class structure of its parent type (a simple `memcpy`) and then by invoking the class_init callback on the resulting class structure. Since the class_init callback is responsible for overwriting the class structure with the user re-implementations of the class methods, we cannot merely use the modified copy of the parent class structure stored in our derived instance. We want to get a copy of the class structure of an instance of the parent class.

```
#define MAMAN_IBAZ_H

#include <glib-object.h>

#define MAMAN_TYPE_IBAZ                (maman_ibaz_get_type ())
#define MAMAN_IBAZ(obj)                (G_TYPE_CHECK_INSTANCE_CAST ((obj), MAMAN_TYPE_IBAZ
#define MAMAN_IBAZ_CLASS(vtable)       (G_TYPE_CHECK_CLASS_CAST ((vtable), MAMAN_TYPE_IBAZ
#define MAMAN_IS_IBAZ(obj)             (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MAMAN_TYPE_IBAZ
#define MAMAN_IS_IBAZ_CLASS(vtable) (G_TYPE_CHECK_CLASS_TYPE ((vtable), MAMAN_TYPE_IBAZ
#define MAMAN_IBAZ_GET_CLASS(inst)  (G_TYPE_INSTANCE_GET_INTERFACE ((inst), MAMAN_TYPE_


typedef struct _MamanIbaz MamanIbaz; /* dummy object */
typedef struct _MamanIbazClass MamanIbazClass;

struct _MamanIbazClass {
  GTypeInterface parent;

  void (*do_action) (MamanIbaz *self);
};

GType maman_ibaz_get_type (void);

void maman_ibaz_do_action (MamanIbaz *self);

#endif /*MAMAN_IBAZ_H*/
```

This code is almost exactly similar to the code for a normal GType which derives from a GObject except for a few details:

- The `_GET_CLASS` macro is not implemented with `G_TYPE_INSTANCE_GET_CLASS` but with `G_TYPE_INSTANCE_GET_INTERFACE`.
- The instance type, MamanIbaz is not fully defined: it is used merely as an abstract type which represents an instance of whatever object which implements the interface.

The implementation of the MamanIbaz type itself is trivial:

- `maman_ibaz_get_type` registers the type in the type system.
- `maman_ibaz_base_init` is expected to register the interface's signals if there are any (we will see a bit (later how to use them). Make sure to use a static local boolean variable to make sure not to run the initialization code twice (as described in the section called “Interface Initialization”, `base_init` is run once for each interface implementation instanciation)
- `maman_ibaz_do_action` de-references the class structure to access its associated class function and calls it.

```
static void
maman_ibaz_base_init (gpointer g_class)
{
  static gboolean initialized = FALSE;

  if (!initialized) {
    /* create interface signals here. */
    initialized = TRUE;
  }
}

GType
maman_ibaz_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanIbazClass),
      maman_ibaz_base_init,   /* base_init */
      NULL,   /* base_finalize */
      NULL,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      0,
      0,      /* n_preallocs */
      NULL    /* instance_init */
    };
    type = g_type_register_static (G_TYPE_INTERFACE, "MamanIbaz", &info, 0);
  }
  return type;
}

void maman_ibaz_do_action (MamanIbaz *self)
{
  MAMAN_IBAZ_GET_CLASS (self)->do_action (self);
}
```

How To define and implement an implementation of an Interface ?

Once the interface is defined, implementing it is rather trivial. Source code showing how to do this for the IBaz interface defined in the previous section is located in `sample/interface/maman-baz.{h|c}`.

The first step is to define a normal GType. Here, we have decided to use a GType which derives from GObject. Its name is MamanBaz:

```
#ifndef MAMAN_BAZ_H
#define MAMAN_BAZ_H

#include <glib-object.h>

#define MAMAN_TYPE_BAZ             (maman_baz_get_type ())
#define MAMAN_BAZ(obj)             (G_TYPE_CHECK_INSTANCE_CAST ((obj), MAMAN_TYPE_BAZ,
#define MAMAN_BAZ_CLASS(vtable)    (G_TYPE_CHECK_CLASS_CAST ((vtable), MAMAN_TYPE_BAZ,
#define MAMAN_IS_BAZ(obj)          (G_TYPE_CHECK_INSTANCE_TYPE ((obj), MAMAN_TYPE_BAZ))
#define MAMAN_IS_BAZ_CLASS(vtable) (G_TYPE_CHECK_CLASS_TYPE ((vtable), MAMAN_TYPE_BAZ))
#define MAMAN_BAZ_GET_CLASS(inst)  (G_TYPE_INSTANCE_GET_CLASS ((inst), MAMAN_TYPE_BAZ,


typedef struct _MamanBaz MamanBaz;
typedef struct _MamanBazClass MamanBazClass;

struct _MamanBaz {
  GObject parent;
  int instance_member;
};

struct _MamanBazClass {
  GObjectClass parent;
};

GType maman_baz_get_type (void);


#endif //MAMAN_BAZ_H
```

There is clearly nothing specifically weird or scary about this header: it does not define any weird API or derives from a weird type.

The second step is to implement `maman_baz_get_type`:

```
GType
maman_baz_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBazClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      NULL,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBaz),
      0,      /* n_preallocs */
      baz_instance_init    /* instance_init */
    };
    static const GInterfaceInfo ibaz_info = {
      (GInterfaceInitFunc) baz_interface_init,    /* interface_init */
      NULL,               /* interface_finalize */
      NULL                /* interface_data */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBazType",
                                   &info, 0);
    g_type_add_interface_static (type,
                                 MAMAN_TYPE_IBAZ,
                                 &ibaz_info);
  }
  return type;
}
```

This function is very much like all the similar functions we looked at previously. The only interface-specific code present here is the call to `g_type_add_interface_static` which is used to inform the type system that this just-registered GType also implements the interface `MAMAN_TYPE_IBAZ`.

`baz_interface_init`, the interface initialization function, is also pretty simple:

```
static void baz_do_action (MamanBaz *self)
{
  g_print ("Baz implementation of IBaz interface Action: 0x%x.\n", self->instance_membe
}
static void
baz_interface_init (gpointer   g_iface,
                    gpointer   iface_data)
{
  MamanIbazClass *klass = (MamanIbazClass *)g_iface;
  klass->do_action = (void (*) (MamanIbaz *self))baz_do_action;
}
static void
baz_instance_init (GTypeInstance   *instance,
                   gpointer         g_class)
{
  MamanBaz *self = (MamanBaz *)instance;
  self->instance_member = 0xdeadbeaf;
}
```

`baz_interface_init` merely initializes the interface methods to the implementations defined by MamanBaz: `maman_baz_do_action` does nothing very useful but it could :)

Interface definition prerequisites

To specify that an interface requires the presence of other interfaces when implemented, GObject introduces the concept of *prerequisites*: it is possible to associate a list of prerequisite interfaces to an interface. For example, if object A wishes to implement interface I1, and if interface I1 has a prerequisite on interface I2, A has to implement both I1 and I2.

The mechanism described above is, in practice, very similar to Java's interface I1 extends interface I2. The example below shows the GObject equivalent:

```
  type = g_type_register_static (G_TYPE_INTERFACE, "MamanIbar", &info, 0);
  /* Make the MamanIbar interface require MamanIbaz interface. */
  g_type_interface_add_prerequisite (type, MAMAN_TYPE_IBAZ);
```

The code shown above adds the MamanIbaz interface to the list of prerequisites of MamanIbar while the code below shows how an implementation can implement both interfaces and register their implementations:

```
static void ibar_do_another_action (MamanBar *self)
{
  g_print ("Bar implementation of IBar interface Another Action: 0x%x.\n", self->instan
}

static void
ibar_interface_init (gpointer   g_iface,
                     gpointer   iface_data)
{
  MamanIbarClass *klass = (MamanIbarClass *)g_iface;
  klass->do_another_action = (void (*) (MamanIbar *self))ibar_do_another_action;
}


static void ibaz_do_action (MamanBar *self)
{
  g_print ("Bar implementation of IBaz interface Action: 0x%x.\n", self->instance_membe
}

static void
ibaz_interface_init (gpointer   g_iface,
                    gpointer   iface_data)
{
  MamanIbazClass *klass = (MamanIbazClass *)g_iface;
  klass->do_action = (void (*) (MamanIbaz *self))ibaz_do_action;
}


static void
bar_instance_init (GTypeInstance   *instance,
                   gpointer         g_class)
{
  MamanBar *self = (MamanBar *)instance;
  self->instance_member = 0x666;
}


GType
maman_bar_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBarClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      NULL,   /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBar),
      0,      /* n_preallocs */
      bar_instance_init    /* instance_init */
    };
    static const GInterfaceInfo ibar_info = {
      (GInterfaceInitFunc) ibar_interface_init,   /* interface_init */
      NULL,                                       /* interface_finalize */
            NULL                                        /* interface_data */
    };
    static const GInterfaceInfo ibaz_info = {
      (GInterfaceInitFunc) ibaz_interface_init,   /* interface_init */
      NULL,                                       /* interface_finalize */
      NULL                                        /* interface_data */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBarType",
                                   &info, 0);
    g_type_add_interface_static (type,
                                 MAMAN_TYPE_IBAZ,
                                 &ibaz_info);
    g_type_add_interface_static (type,
                                 MAMAN_TYPE_IBAR,
                                 &ibar_info);
  }
  return type;
}
```

It is very important to notice that the order in which interface implementations are added to the main object is not random: `g_type_interface_static` must be invoked first on the interfaces which have no prerequisites and then on the others.

Complete source code showing how to define the MamanIbar interface which requires MamanIbaz and how to implement the MamanIbar interface is located in `sample/interface/maman-ibar.{h|c}` and `sample/interface/maman-bar.{h|c}`.

Interface Properties

Starting from version 2.4 of glib, gobject interfaces can also have properties. Declaration of the interface properties is similar to declaring the properties of ordinary gobject types as explained in the section called “Object properties”, except that `g_object_interface_install_property` is used to declare the properties instead of `g_object_class_install_property`.

To include a property named 'name' of type string in the maman_ibaz interface example code above, we only need to add one [12] line in the `maman_ibaz_base_init` [13] as shown below:

```
static void
maman_ibaz_base_init (gpointer g_class)
{
  static gboolean initialized = FALSE;

  if (!initialized) {
    /* create interface signals here. */

    g_object_interface_install_property (g_class,
                            g_param_spec_string ("name",
                                    "maman_ibaz_name",
                                    "Name of the MamanIbaz",
                                    "maman",
                                    G_PARAM_READWRITE));
    initialized = TRUE;
  }
}
```

One point worth noting is that the declared property wasn't assigned an integer ID. The reason being that integer IDs of properities are utilized only inside the get and set methods and since interfaces do not implement properties, there is no need to assign integer IDs to interface properties.

[12]That really is one line extended to six for the sake of clarity
[13]The gobject_install_property can also be called from `class_init` but it must not be called after that point.

The story for the implementers of the interface is also quite trivial. An implementer shall declare and define it's properties in the usual way as explained in the section called “Object properties”, except for one small change: it shall declare the properties of the interface it implements using `g_object_class_override_property` instead of `g_object_class_install_property`. The following code snipet shows the modifications needed in the MamanBaz declaration and implementation above:

```
struct _MamanBaz {
  GObject parent;
  gint instance_member;
  gchar *name;         /* placeholder for property */
};

enum
{
  ARG_0,
  ARG_NAME
};

GType
maman_baz_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    static const GTypeInfo info = {
      sizeof (MamanBazClass),
      NULL,   /* base_init */
      NULL,   /* base_finalize */
      baz_class_init, /* class_init */
      NULL,   /* class_finalize */
      NULL,   /* class_data */
      sizeof (MamanBaz),
      0,      /* n_preallocs */
      baz_instance_init    /* instance_init */
    };
    static const GInterfaceInfo ibaz_info = {
      (GInterfaceInitFunc) baz_interface_init,    /* interface_init */
      NULL,               /* interface_finalize */
      NULL                /* interface_data */
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "MamanBazType",
                                   &info, 0);
    g_type_add_interface_static (type,
                                 MAMAN_TYPE_IBAZ,
                                 &ibaz_info);
  }
  return type;
}

static void
maman_baz_class_init (MamanBazClass * klass)
{
  GObjectClass *gobject_class;

  gobject_class = (GObjectClass *) klass;

  parent_class = g_type_class_ref (G_TYPE_OBJECT);

  gobject_class->set_property = maman_baz_set_property;
  gobject_class->get_property = maman_baz_get_property;

  g_object_class_override_property (gobject_class, ARG_NAME, "name");
}

static void
maman_baz_set_property (GObject * object, guint prop_id,
                        const GValue * value, GParamSpec * pspec)
{
  MamanBaz *baz;
  GObject *obj;

  /* it's not null if we got it, but it might not be ours */
  g_return_if_fail (G_IS_MAMAN_BAZ (object));

  baz = MAMAN_BAZ (object);

  switch (prop_id) {
    case ARG_NAME:
      baz->name = g_value_get_string (value);
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
  }
}

static void
maman_baz_get_property (GObject * object, guint prop_id,
                        GValue * value, GParamSpec * pspec)
{
  MamanBaz *baz;

  /* it's not null if we got it, but it might not be ours */
  g_return_if_fail (G_IS_TEXT_PLUGIN (object));

  baz = MAMAN_BAZ (object);

  switch (prop_id) {
    case ARG_NAME:
      g_value_set_string (value, baz->name);
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
  }
}
```

Howto create and use signals

The signal system which was built in GType is pretty complex and flexible: it is possible for its users to connect at runtime any number of callbacks (implemented in any language for which a binding exists) 14 to any signal and to stop the emission of any signal at any state of the signal emission process. This flexibility makes it possible to use GSignal for much more than just emit events which can be received by numerous clients.

Simple use of signals

The most basic use of signals is to implement simple event notification: for example, if we have a MamanFile object, and if this object has a write method, we might wish to be notified whenever someone uses this method. The code below shows how the user can connect a callback to the write signal. Full code for this simple example is located in `sample/signal/maman-file.{h|c}` and in `sample/signal/test.c`

```
file = g_object_new (MAMAN_FILE_TYPE, NULL);

g_signal_connect (G_OBJECT (file), "write",
                  (GCallback)write_event,
                  NULL);
```

14A python callback can be connected to any signal on any C-based GObject.

```
maman_file_write (file, buffer, 50);
```

The MamanFile signal is registered in the class_init function:

```
klass->write_signal_id =
  g_signal_newv ("write",
                 G_TYPE_FROM_CLASS (g_class),
                 G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
                 NULL /* class closure */,
                 NULL /* accumulator */,
                 NULL /* accu_data */,
                 g_cclosure_marshal_VOID__VOID,
                 G_TYPE_NONE /* return_type */,
                 0     /* n_params */,
                 NULL  /* param_types */);
```

and the signal is emited in `maman_file_write`:

```
void maman_file_write (MamanFile *self, guint8 *buffer, guint32 size)
{
  /* First write data. */
  /* Then, notify user of data written. */
  g_signal_emit (self, MAMAN_FILE_GET_CLASS (self)->write_signal_id,
                 0 /* details */,
                 NULL);
}
```

As shown above, you can safely set the details parameter to zero if you do not know what it can be used for. For a discussion of what you could used it for, see the section called “The detail argument”

The signature of the signal handler in the above example is defined as `g_cclosure_marshal_VOID__VOID`. Its name follows a simple convention which encodes the function parameter and return value types in the function name. Specifically, the value infront of the double underscore is the type of the return value, while the value(s) after the double underscore denote the parameter types. The header `gobject/gmarshal.h` defines a set of commonly needed closures that one can use.

How to provide more flexibility to users ?

The previous implementation does the job but the signal facility of GObject can be used to provide even more flexibility to this file change notification mechanism. One of the key ideas is to make the process of writing data to the file part of the signal emission process to allow users to be notified either before or after the data is written to the file.

To integrate the process of writing the data to the file into the signal emission mechanism, we can register a default class closure for this signal which will be invoked during the signal emission, just like any other user-connected signal handler.

The first step to implement this idea is to change the signature of the signal: we need to pass around the buffer to write and its size. To do this, we use our own marshaller which will be generated through glib's genmarshall tool. We thus create a file named `marshall.list` which contains the following single line:

```
VOID:POINTER,UINT
```

and use the Makefile provided in `sample/signal/Makefile` to generate the file named `maman-file-complex-marshall.c`. This C file is finally included in `maman-file-complex.c`.

Once the marshaller is present, we register the signal and its marshaller in the class_init function of the object MamanFileComplex (full source for this object is included in `sample/signal/maman-file-complex.{h|c}`):

```
GClosure *default_closure;
GType param_types[2];

default_closure = g_cclosure_new (G_CALLBACK (default_write_signal_handler),
                                  (gpointer)0xdeadbeaf /* user_data */,
                                  NULL /* destroy_data */);

param_types[0] = G_TYPE_POINTER;
param_types[1] = G_TYPE_UINT;
klass->write_signal_id =
  g_signal_newv ("write",
                 G_TYPE_FROM_CLASS (g_class),
                 G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
                 default_closure /* class closure */,
                 NULL /* accumulator */,
                 NULL /* accu_data */,
                 maman_file_complex_VOID__POINTER_UINT,
                 G_TYPE_NONE /* return_type */,
                 2     /* n_params */,
                 param_types /* param_types */);
```

The code shown above first creates the closure which contains the code to complete the file write. This closure is registered as the default class_closure of the newly created signal.

Of course, you need to implement completely the code for the default closure since I just provided a skeleton:

```
static void
default_write_signal_handler (GObject *obj, guint8 *buffer, guint size, gpointer user_d
{
  g_assert (user_data == (gpointer)0xdeadbeaf);
  /* Here, we trigger the real file write. */
  g_print ("default signal handler: 0x%x %u\n", buffer, size);
}
```

Finally, the client code must invoke the `maman_file_complex_write` function which triggers the signal emission:

```
void maman_file_complex_write (MamanFileComplex *self, guint8 *buffer, guint size)
{
  /* trigger event */
  g_signal_emit (self,
                 MAMAN_FILE_COMPLEX_GET_CLASS (self)->write_signal_id,
                 0, /* details */
                 buffer, size);
}
```

The client code (as shown in `sample/signal/test.c` and below) can now connect signal handlers before and after the file write is completed: since the default signal handler which does the write itself runs during the RUN_LAST phase of the signal emission, it will run after all handlers connected with `g_signal_connect` and before all handlers connected with `g_signal_connect_after`. If you intent to write a GObject which emits signals, I would thus urge you to create all your signals with the G_SIGNAL_RUN_LAST such that your users have a maximum of flexibility as to when to get the event. Here, we combined it with G_SIGNAL_NO_RECURSE and G_SIGNAL_NO_HOOKS to ensure our users will not try to do really weird things with our GObject. I strongly advise you to do the same unless you really know why (in which case you really know the inner workings of GSignal by heart and you are not reading this).

```
static void complex_write_event_before (GObject *file, guint8 *buffer, guint size, gpoi
{
  g_assert (user_data == NULL);
  g_print ("Complex Write event before: 0x%x, %u\n", buffer, size);
}
```

```
static void complex_write_event_after (GObject *file, guint8 *buffer, guint size, gpoin
{
  g_assert (user_data == NULL);
  g_print ("Complex Write event after: 0x%x, %u\n", buffer, size);
}

static void test_file_complex (void)
{
  guint8 buffer[100];
  GObject *file;

  file = g_object_new (MAMAN_FILE_COMPLEX_TYPE, NULL);

  g_signal_connect (G_OBJECT (file), "write",
                    (GCallback)complex_write_event_before,
                    NULL);

  g_signal_connect_after (G_OBJECT (file), "write",
                          (GCallback)complex_write_event_after,
                          NULL);

  maman_file_complex_write (MAMAN_FILE_COMPLEX (file), buffer, 50);

  g_object_unref (G_OBJECT (file));
}
```

The code above generates the following output on my machine:

```
Complex Write event before: 0xbfffe280, 50
default signal handler: 0xbfffe280 50
Complex Write event after: 0xbfffe280, 50
```

How most people do the same thing with less code

For many historic reasons related to how the ancestor of GObject used to work in GTK+ 1.x versions, there is a much *simpler* 15 way to create a signal with a default handler than to create a closure by hand and to use the `g_signal_newv`.

For example, `g_signal_new` can be used to create a signal which uses a default handler which is stored in the class structure of the object. More specifically, the class structure contains a function pointer which is accessed during signal emission to invoke the default handler and the user is expected to provide to `g_signal_new` the offset from the start of the class structure to the function pointer. 16

The following code shows the declaration of the MamanFileSimple class structure which contains the `write` function pointer.

```
struct _MamanFileSimpleClass {
  GObjectClass parent;

  guint write_signal_id;

  /* signal default handlers */
  void (*write) (MamanFileSimple *self, guint8 *buffer, guint size);
};
```

15I personally think that this method is horribly mind-twisting: it adds a new indirection which unecessarily complicates the overall code path. However, because this method is widely used by all of GTK+ and GObject code, readers need to understand it. The reason why this is done that way in most of GTK+ is related to the fact that the ancestor of GObject did not provide any other way to create a signal with a default handler than this one. Some people have tried to justify that it is done that way because it is better, faster (I am extremly doubtfull about the faster bit. As a matter of fact, the better bit also mystifies me ;-). I have the feeling no one really knows and everyone does it because they copy/pasted code from code which did the same. It is probably better to leave this specific trivia to hacker legends domain...
16I would like to point out here that the reason why the default handler of a signal is named everywhere a class_closure is probably related to the fact that it used to be really a function pointer stored in the class structure.

The `write` function pointer is initialied in the class_init function of the object to `default_write_signal_handler`:

```
static void
maman_file_simple_class_init (gpointer g_class,
                               gpointer g_class_data)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (g_class);
  MamanFileSimpleClass *klass = MAMAN_FILE_SIMPLE_CLASS (g_class);

  klass->write = default_write_signal_handler;
```

Finally, the signal is created with `g_signal_new` in the same class_init function:

```
klass->write_signal_id =
 g_signal_new ("write",
               G_TYPE_FROM_CLASS (g_class),
               G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
               G_STRUCT_OFFSET (MamanFileSimpleClass, write),
               NULL /* accumulator */,
               NULL /* accu_data */,
               maman_file_complex_VOID__POINTER_UINT,
               G_TYPE_NONE /* return_type */,
               2     /* n_params */,
               G_TYPE_POINTER,
               G_TYPE_UINT);
```

Of note, here, is the 4th argument to the function: it is an integer calculated by the `G_STRUCT_OFFSET` macro which indicates the offset of the member *write* from the start of the MamanFileSimpleClass class structure. [17]

While the complete code for this type of default handler looks less clutered as shown in `sample/signal/maman-file-simple.{h|c}`, it contains numerous subtleties. The main subtle point which everyone must be aware of is that the signature of the default handler created that way does not have a user_data argument: `default_write_signal_handler` is different in `sample/signal/maman-file-complex.c` and in `sample/signal/maman-file-simple.c`.

If you have doubts about which method to use, I would advise you to use the second one which involves `g_signal_new` rather than `g_signal_newv`: it is better to write code which looks like the vast majority of other GTK+/Gobject code than to do it your own way. However, now, you know why.

How users can abuse signals (and why some think it is good)

Now that you know how to create signals to which the users can connect easily and at any point in the signal emission process thanks to `g_signal_connect`, `g_signal_connect_after` and G_SIGNAL_RUN_LAST, it is time to look into how your users can and will screw you. This is also interesting to know how you too, can screw other people. This will make you feel good and eleet.

The users can:

- stop the emission of the signal at anytime
- override the default handler of the signal if it is stored as a function pointer in the class structure (which is the prefered way to create a default signal handler, as discussed in the previous section).

In both cases, the original programmer should be as careful as possible to write code which is resistant to the fact that the default handler of the signal might not able to run. This is obviously not the case in the example used in the previous sections since the write to the file depends on whether or not the default handler runs (however, this might be your goal: to allow the user to prevent the file write if he wishes to).

17GSignal uses this offset to create a special wrapper closure which first retrieves the target function pointer before calling it.

If all you want to do is to stop the signal emission from one of the callbacks you connected yourself, you can call `g_signal_stop_by_name`. Its use is very simple which is why I won't detail it further.

If the signal's default handler is just a class function pointer, it is also possible to override it yourself from the class_init function of a type which derives from the parent. That way, when the signal is emitted, the parent class will use the function provided by the child as a signal default handler. Of course, it is also possible (and recommended) to chain up from the child to the parent's default signal handler to ensure the integrity of the parent object.

Overriding a class method and chaining up was demonstrated in the section called “Object methods” which is why I won't bother to show exactly how to do it here again.

Chapter 6. GObject related tools

Debugging reference count problems

The reference counting scheme used by GObject does solve quite a few memory management problems but also introduces new sources of bugs. In large applications, finding the exact spot where a the reference count of an Object is not properly handled can be very difficult. Hopefully, there exist at a too named refdbg/ [http://refdbg.sf.net/] which can be used to automate the task of tracking down the location of invalid code with regard to reference counting. This application intercepts the reference counting calls and tries to detect invalid behavior. It suports a filter-rule mechanism to let you trace only the objects you are interested in and it can be used together with gdb.

Writing API docs

The API documentation for most of the Glib, GObject, GTK+ and GNOME libraries is built with a combination of complex tools. Typically, the part of the documentation which describes the behavior of each function is extracted from the specially-formatted source code comments by a tool named gtk-doc which generates docbook xml and merges this docbook xml with a set of master xml docbook files. These xml docbook files are finally processed with xsltproc (a small program part of the libxslt library) to generate the final html output. Other tools can be used to generate pdf output from the source xml. The following code excerpt shows what these comments look like.

```
/**
 * gtk_widget_freeze_child_notify:
 * @widget: a #GtkWidget
 *
 * Stops emission of "child-notify" signals on @widget. The signals are
 * queued until gtk_widget_thaw_child_notify() is called on @widget.
 *
 * This is the analogue of g_object_freeze_notify() for child properties.
 **/
void
gtk_widget_freeze_child_notify (GtkWidget *widget)
{
...
```

The great thoroughful documentation [http://developer.gnome.org/arch/doc/authors.html] on how to setup and use gtk-doc in your project is provided on the gnome developer website. gtk-doc generates

Chapter 7. GObject Version Changes

GObject was originally part of GTK+ and, as such, was much less generic and was tied to the X windowing system. After versions 1.0.x and 1.2.x of GTK+s, GObject was split, moved to GLib and was made much more generic (it gained flexibility and complexity unfortunatly). The first versions of GLib which included GObject were the unstable releases 1.3.x which lead to the stable GLib 2.0.x releases.

This document is focused on the 2.0.x series even though other stable releases such as 2.2.x were available at that time mainly because all subsequent 2.x releases are supposed to be upward binary and source compatible. Of course, a few new features poped up in GObject after 2.0.x and this chapter tries to give a quick introduction to these new features. If you decide to use them, make sure you understand what this means in terms of library version requirements for your customers.

GLib 2.2.x releases

This version did not add many new features and fixed a few corner-case type bugs. `g_type_interface_prerequisites` was added to improve the introspection APIs.

GLib 2.4.x releases

This version adds a few conveniance features:

- `G_DEFINE_TYPE` macros can be used to easily define new GObjects.
- Interfaces support properties (see the section called “Interface Properties”).
- GType instances can now have a private data structure to hold private data members and implement the pimpl idiom.

GLib 2.6.x releases

This version is under development and temporary snapshots can be found in the Gnome CVS server under the version name 2.5.x.

Chapter 8. GObject performance

Describe all Performance facts of GObject: memory use, speed...