GObject tutorial for beginners

Toshio Sekiya

abstract

The GitHub page of this tutorial is available. Click here.

About this tutorial GObject is the base system for the GTK library, the current version of which is four. GTK provides GUI on Linux and is used by GNOME desktop system and many applications. See GTK 4 tutorial. One of the problem to understand GTK 4 is the difficulty of the GObject. This tutorial is useful for those who learns GTK 4. And the readers of this tutorial should read GTK 4 tutorial because GTK is the only application of GObject so far.

GObject API Reference offers everything necessary for GObject. The contents of this tutorial are not beyond the documentation. It just shows examples and how to write GObject programs. But I believe it is useful for the beginners who feels difficulty to learn the GObject system. Readers should refer to the GObject documentation when learning this tutorial.

Generating GFM, HTML and PDF The table of contents are at the end of this file and you can see all the tutorials through the link. However, you can make GFM, HTML or PDF by the following steps. GFM is 'GitHub Flavored Markdown', which is used in the document files in the GitHub repository.

1. You need Linux operating system, ruby, rake, pandoc and LaTeX system.
2. download the GObject-tutorial repository and uncompress the files.
3. change your current directory to the top directory of the files.
4. type `rake` to produce GFM files. The files are generated under `gfm` directory.
5. type `rake html` to produce HTML files. The files are generated under `docs` directory.
6. type `rake pdf` to produce a PDF file. The file is generated under `latex` directory.

This system is the same as the one in the `GTK 4 tutorial` repository. There's a document `Readme_for_developers.md` in `gfm` directory in it. It describes the details.

Contribution If you have any questions, feel free to post an issue. If you find any mistakes in the tutorial, post an issue or pull-request. When you give a pull-request, correct the source files, which are under the 'src' directory, and run `rake` and `rake html`. Then GFM and HTML files are automatically updated.

Contents

1 Prerequisites and License

1.1 Prerequisites

1.1.1 Tutorial document

This tutorial is about GObject libraries. It is originally used on Linux with C compiler, but now it is used more widely, on windows and macOS, with Vala, python and so on. However, this tutorial describes only *C programs on Linux.*

If you want to try to compile the examples in the tutorial, you need:

- PC with Linux distribution like Ubuntu, Debian and so on.
- Gcc
- GLib. The version at the time this document is described is 2.76.1. Some example program needs version 2.74 or higher. But they work on the older version if you replace the new function or macro with the old one.
- pkg-config
- meson and ninja

Common Linux distributions has GLib, which is enough for you to compile the examples in this repository.

1.1.2 Tools to make GFM, HTML and PDF files

This repository includes ruby programs. They are used to create Markdown(GFM) files, HTML files, LaTeX files and a PDF file.

You need:

- Linux distribution like Ubuntu.
- Ruby programming language. There are two ways to install it. One is installing the distribution's package. The other is using rbenv and ruby-build. If you want to use the latest version of Ruby, you will need rbenv and ruby-build.
- Rake. It is a gem, which is a library written in ruby. Ruby package includes Rake gem as a standard library so you don't need to install it separately.

1.2 License

2 GObject

2.1 Class and instance

GObject instance is created with `g_object_new` function. GObject has not only instances but also classes.

- A class of GObject is created at the first call of `g_object_new`. And there exists only one GObject class.
- GObject instance is created whenever `g_object_new` is called. So, two or more GObject instances can exist.

In a broad sense, GObject means the object which includes its class and instances. In a narrow sense, GObject is a definition of a C structure.

```
typedef struct _GObject  GObject;
struct  _GObject
{
  GTypeInstance  g_type_instance;

  /*< private >*/
  guint          ref_count;  /* (atomic) */
  GData         *qdata;
};
```

The `g_object_new` function allocates GObject-sized memory, initializes the memory and returns the pointer to the memory. The memory is a GObject instance.

In the same way, the class of GObject is memory allocated by `g_object_new` and its structure is defined with GObjectClass. The following is extracted from `gobject.h`. But you don't need to know the details of the structure now.

```
struct  _GObjectClass
{
  GTypeClass   g_type_class;

  /*< private >*/
  GSList      *construct_properties;

  /*< public >*/
  /* seldom overridden */
  GObject*   (*constructor)     (GType                  type,
                                  guint                  n_construct_properties,
                                  GObjectConstructParam *construct_properties);
  /* overridable methods */
  void       (*set_property)        (GObject        *object,
                                          guint           property_id,
                                          const GValue   *value,
                                          GParamSpec     *pspec);
  void       (*get_property)        (GObject        *object,
                                          guint           property_id,
                                          GValue         *value,
                                          GParamSpec     *pspec);
  void       (*dispose)         (GObject        *object);
  void       (*finalize)        (GObject        *object);
  /* seldom overridden */
  void       (*dispatch_properties_changed) (GObject      *object,
                             guint      n_pspecs,
                             GParamSpec  **pspecs);
  /* signals */
  void       (*notify)          (GObject    *object,
                         GParamSpec *pspec);

  /* called when done constructing */
  void       (*constructed)     (GObject    *object);

  /*< private >*/
  gsize     flags;

  gsize         n_construct_properties;

  gpointer pspecs;
  gsize n_pspecs;

  /* padding */
  gpointer  pdummy[3];
};
```

The programs for GObject are included in GLib source files. You can download the GLib source files from GNOME download page.

There are sample programs in src/misc directory in the GObject tutorial repository. You can compile them by:

```
$ cd src/misc
$ meson setup _build
$ ninja -C _build
```

One of the programs is `example1.c`. Its code is as follows.

```
 1 #include <glib-object.h>
 2 
 3 int
 4 main (void)
 5 {
 6   GObject *instance1 = g_object_new (G_TYPE_OBJECT, NULL);
 7   GObject *instance2 = g_object_new (G_TYPE_OBJECT, NULL);
 8   g_print ("The␣address␣of␣instance1␣is␣%p\n", instance1);
 9   g_print ("The␣address␣of␣instance2␣is␣%p\n", instance2);
10 
11   GObjectClass *class1 = G_OBJECT_GET_CLASS (instance1);
12   GObjectClass *class2 = G_OBJECT_GET_CLASS (instance2);
13 
14   g_print ("The␣address␣of␣the␣class␣of␣instance1␣is␣%p\n", class1);
15   g_print ("The␣address␣of␣the␣class␣of␣instance2␣is␣%p\n", class2);
16   g_print ("Class␣Name:␣%s\n", G_OBJECT_CLASS_NAME (class1));
17 
18   g_object_unref (instance1);
19   g_object_unref (instance2);
20 }
```

- 5-6: `instance1` and `instance2` are pointers that points GObject instances. `class1` and `class2` points a class of the instances.
- 8-11: A function `g_object_new` creates a GObject instance. GObject instance is a chunk of memory which has GObject structure (`struct _GObject`). The argument `G_TYPE_OBJECT` is the type of GObject. This type is different from C language type like `char` or `int`. There is *Type System* which is a base system of GObject system. Every data type such as GObject must be registered to the type system. The type system has series of functions for the registration. If one of the functions is called, then the type system determines `GType` type value for the object and returns it to the caller. `GType` is an unsigned long integer on my computer but it depends on the hardware. `g_object_new` allocates GObject-sized memory and returns the pointer to the top address of the memory. After the creation, this program displays the addresses of instances.
- 13-16: A macro `G_OBJECT_GET_CLASS` returns the pointer to the class of the argument. Therefore, `class1` points the class of `instance1` and `class2` points the class of `instance2` respectively. The addresses of the two classes are displayed.
- 18-19: `g_object_unref` will be explained in the next subsection. It destroys the instances and the memory is freed.

Now, execute it.

```
$ cd src/misc; _build/example1
```

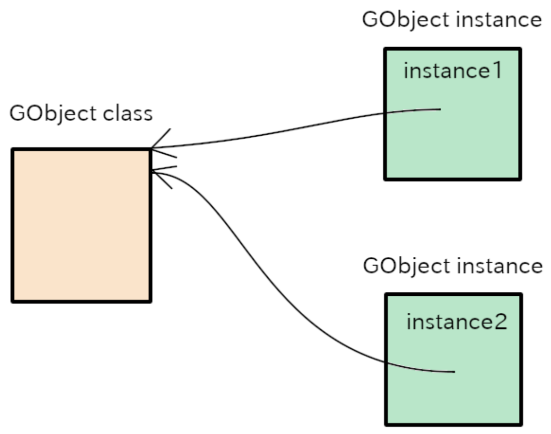

Figure 1: Class and Instance

```
The address of instance1 is 0x55895eaf7ad0
The address of instance2 is 0x55895eaf7af0
The address of the class of instance1 is 0x55895eaf7880
The address of the class of instance2 is 0x55895eaf7880
```

The locations of two instances `instance1` and `instance2` are different. Each instance has its own memory. The locations of two classes `class1` and `class2` are the same. Two GObject instances share the same class.

2.2 Reference count

GObject instance has its own memory. They are allocated by the system when it is created. If it becomes useless, the memory must be freed. However, how can we determine whether it is useless? GObject system provides reference count to solve the problem.

An instance is created and used by other instance or the main program. That is to say, the instance is referred. If the instance is referred by A and B, then the number of the reference is two. This number is called *reference count*. Let's think about a scenario like this:

- A calls `g_object_new` and owns an instance G. A refers G, so the reference count of G is 1.
- B wants to use G too. B calls `g_object_ref` and increases the reference count by 1. Now the reference count is 2.
- A no longer uses G. A calls `g_object_unref` and decreases the reference count by 1. Now the reference count is 1.
- B no longer uses G. B calls `g_object_unref` and decreases the reference count by 1. Now the reference count is 0.
- Because the reference count is zero, G knows that no one refers to it. G begins finalizing process by itself. G disappears and the memory is freed.

A program `example2.c` is based on the scenario above.

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

static void
show_ref_count (GObject *instance)
{
  if (G_IS_OBJECT (instance))
    {
      g_print ("Reference count is %d.\n", instance->ref_count);
    }
  else
    {
      g_print ("Instance␣is␣not␣a␣GObject.\n");
    }
}

int
main (void)
{
  GObject *instance = g_object_new (G_TYPE_OBJECT, NULL);
  g_print ("Call␣g_object_new.\n");
  show_ref_count (instance);
  g_object_ref (instance);
  g_print ("Call␣g_object_ref.\n");
  show_ref_count (instance);
  g_object_unref (instance);
  g_print ("Call␣g_object_unref.\n");
  show_ref_count (instance);
  g_object_unref (instance);

  g_print ("Call␣g_object_unref.\n");
  g_print ("Now␣the␣reference␣count␣is␣zero␣and␣the␣instance␣is␣destroyed.\n");
  g_print ("The␣instance␣memories␣are␣possibly␣returned␣to␣the␣system.\n");
  g_print ("Therefore,␣the␣access␣to␣the␣same␣address␣may␣cause␣a␣segmentation␣
      error.\n");
}
```

Now execute it.

```
$ cd src/misc; _build/example2
bash: cd: src/misc: No such file or directory
Call g_object_new.
Reference count is 1.
Call g_object_ref.
Reference count is 2.
Call g_object_unref.
Reference count is 1.
Call g_object_unref.
Now the reference count is zero and the instance is destroyed.
The instance memories are possibly returned to the system.
Therefore, the access to the same address may cause a segmentation error.
```

`example2` shows:

- `g_object_new` creates a new GObject instance and sets its reference count to 1.
- `g_object_ref` increases the reference count by 1.
- `g_object_unref` decreases the reference count by 1. If the reference count drops to zero, the instance destroys itself.

2.3 Initialization and destruction process

The actual process of GObject initialization and destruction is very complex. The following is simplified description without details.

Initialization

1. Registers GObject type with the type system. This is done in the GLib initialization process before the function `main` is called. (If the compiler is gcc, then `__attribute__ ((constructor))` is used to qualify the initialization function. Refer to GCC manual.)
2. Allocates memory for GObjectClass and GObject structure.
3. Initializes the GObjectClass structure memory. This memory will be the class of GObject.
4. Initializes the GObject structure memory. This memory will be the instance of GObject.

This initialization process is carried out when `g_object_new` function is called for the first time. At the second and subsequent call for `g_object_new`, it performs only two processes: (1) memory allocation for GObject structure (2) initialization for the memory. `g_object_new` returns the pointer that points the instance (the memory allocated for the GObject structure).

Destruction

1. Destroys GObject instance. The memory for the instance is freed.

GObject type is a static type. Static type never destroys its class. So, even if the destroyed instance is the last instance, the class still remains.

When you write code to define a child object of GObject, It is important to understand the process above. The detailed process will be explained in the later sections.

3 Type system and registration process

GObject is a base object. We don't usually use GObject itself. Because GObject is very simple and not enough to be used by itself in most situations. Instead, we use descendant objects of GObject such as many kinds of GtkWidget. We can rather say such derivability is the most important feature of GObject.

This section describes how to define a child object of GObject.

3.1 Name convention

An example of this section is an object represents a real number. It is not so useful because we have already had double type in C language to represent real numbers. However, I think this example is not so bad to know the technique how to define a child object.

First, you need to know the naming convention. An object name consists of name space and name. For example, "GObject" consists of a name space "G" and a name "Object". "GtkWidget" consists of a name space "Gtk" and a name "Widget". Let the name space be "T" and the name be "Double" of the new object. In this tutorial, we use "T" as a name space for all the objects we make.

TDouble is the object name. It is a child object of GObject. It represents a real number and the type of the number is double. It has some useful functions.

3.2 Define TDoubleClass and TDouble

When we say "type", it can be the type in the type system or C language type. For example, GObject is a type name in the type system. And char, int or double is C language types. When the meaning of the word "type" is clear in the context, we just call it "type". But if it's ambiguous, we call it "C type" or "type in the type system".

TDouble object has the class and instance. The C type of the class is TDoubleClass. Its structure is like this:

```
typedef struct _TDoubleClass TDoubleClass;
struct _TDoubleClass {
  GObjectClass parent_class;
};
```

_TDoubleClass is a C structure tag name and TDoubleClass is "struct _TDoubleClass". TDoubleClass is a newly created C type.

- Use typedef to define a class type.
- The first member of the structure must be the parent's class structure.

TDoubleClass doesn't need its own member.

The C type of the instance of TDouble is TDouble.

```
typedef struct _TDouble TDouble;
struct _TDouble {
  GObject parent;
  double value;
};
```

This is similar to the structure of the class.

- Use typedef to define an instance type.

- The first member of the structure must be the parent's instance structure.

TDouble has its own member, "value". It is the value of TDouble instance.

The coding convention above needs to be kept all the time.

3.3 Creation process of a child of GObject

The creation process of TDouble type is similar to the one of GObject.

1. Registers TDouble type to the type system.
2. The type system allocates memory for TDoubleClass and TDouble.
3. Initializes TDoubleClass.
4. Initializes TDouble.

3.4 Registration

Usually registration is done by convenient macro such as `G_DECLARE_FINAL_TYPE` and `G_DEFINE_TYPE`. You can use `G_DEFINE_FINAL_TYPE` for a final type class instead of `G_DEFINE_TYPE` since GLib version 2.70. So you don't need to care about registration details. But, in this tutorial, it is important to understand GObject type system, so I want to show you the registration without macro, first.

There are two kinds of types, static and dynamic. Static type doesn't destroy its class even after all the instances have been destroyed. Dynamic type destroys its class when the last instance has been destroyed. The type of GObject is static and its descendant objects' type is also static. The function `g_type_register_static` registers a type of a static object. The following code is extracted from `gtype.h` in the Glib source files.

```
GType
g_type_register_static (GType           parent_type,
                        const gchar     *type_name,
                        const GTypeInfo *info,
                        GTypeFlags      flags);
```

The parameters above are:

- parent_type: Parent type.
- type_name: The name of the type. For example, "TDouble".
- info: Information of the type. `GTypeInfo` structure will be explained below.
- flags: Flag. If the type is abstract type or abstract value type, then set their flag. Otherwise, set it to zero.

Because the type system maintains the parent-child relationship of the type, `g_type_register_static` has a parent type parameter. And the type system also keeps the information of the type. After the registration, `g_type_register_static` returns the type of the new object.

`GTypeInfo` structure is defined as follows.

```
typedef struct _GTypeInfo  GTypeInfo;

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

  GBaseInitFunc          base_init;
  GBaseFinalizeFunc      base_finalize;

  /* interface types, 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;
};
```

This structure needs to be created before the registration.

- class_size: The size of the class. For example, TDouble's class size is `sizeof (TDoubleClass)`.
- base_init, base_finalize: These functions initialize/finalize the dynamic members of the class. In many cases, they aren't necessary, and are assigned NULL. For further information, see GObject API Reference – BaseInitFunc and GObject API Reference – ClassInitFunc.
- class_init: Initializes static members of the class. Assign your class initialization function to `class_init` member. By convention, the name is `<name space>_<name>_class_init`, for example, `t_double_class_init`.
- class_finalize: Finalizes the class. Because descendant type of GObjec is static, it doesn't have a finalize function. Assign NULL to `class_finalize` member.
- class_data: User-supplied data passed to the class init/finalize functions. Usually NULL is assigned.
- instance_size: The size of the instance. For example, TDouble's instance size is `sizeof (TDouble)`.
- n_preallocs: This is ignored. it has been used by the old version of Glib.
- instance_init: Initializes instance members. Assign your instance initialization function to `instance_init` member. By convention, the name is `<name space>_<name>_init`, for example, `t_double_init`.
- value_table: This is usually only useful for fundamental types. If the type is descendant of GObject, assign NULL.

These information is kept by the type system and used when the object is created or destroyed. Class_size and instance_size are used to allocate memory for the class and instance. Class_init and instance_init functions are called when class or instance is initialized.

The C program `example3.c` shows how to use `g_type_register_static`.

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

typedef struct _TDouble
{
  GObject parent;
  double  value;
} TDouble;

typedef struct _TDoubleClass
{
  GObjectClass parent;
} TDoubleClass;

static void
t_double_class_init (TDoubleClass *class)
{
}

static void
t_double_init (TDouble *self)
{
}

GType
t_double_get_type (void)
{
  static GType type;

  if (!type)
    {
      GTypeInfo info = {
        .class_size    = sizeof (TDoubleClass),
33          .class_init    = (GClassInitFunc)t_double_class_init,
34          .instance_size = sizeof (TDouble),
35          .instance_init = (GInstanceInitFunc)t_double_init,
36        };
37        type = g_type_register_static (G_TYPE_OBJECT, "TDouble", &info,
             G_TYPE_FLAG_NONE);
38      }
39    return type;
40  }
41
42  #define T_DOUBLE_TYPE (t_double_get_type ())
43
44  int
45  main (void)
46  {
47    GType dtype = T_DOUBLE_TYPE;
48
49    if (dtype)
50      {
51        g_print ("Type␣registration␣was␣successful!\n");
52        g_print ("The␣type␣ID␣is␣%lx.\n", dtype);
53      }
54    else
55      {
56        g_print ("Type␣registration␣failed.\n");
57        exit (EXIT_FAILURE);
58      }
59
60    TDouble *d = g_object_new (T_DOUBLE_TYPE, NULL);
61    if (d)
62      {
63        g_print ("Instantiation␣was␣successful!\n");
64        g_print ("The␣instance␣address␣is␣%p.\n", d);
65      }
66    else
67      {
68        g_print ("Instantiation␣failed.\n");
69        exit (EXIT_FAILURE);
70      }
71
72    g_object_unref (d);
73  }
```

- 16-22: A class initialization function and an instance initialization function. The argument `class` points the class structure and the argument `self` points the instance structure. They do nothing here but they are necessary for the registration.
- 24-43: `t_double_get_type` function. This function returns the type of the TDouble object. The name of a function is always `<name space>_<name>_get_type`. And a macro `<NAME_SPACE>_TYPE_<NAME>` (all characters are upper case) is replaced by this function. Look at line 3. `T_TYPE_DOUBLE` is a macro replaced by `t_double_get_type ()`. This function has a static variable `type` to keep the type of the object. At the first call of this function, `type` is zero. Then it calls `g_type_register_static` to register the object to the type system. At the second or subsequent call, the function just returns `type`, because the static variable `type` has been assigned non-zero value by `g_type_register_static` and it keeps the value.
- 30-40 : Sets `info` structure and calls `g_type_register_static`.
- 45-64: Main function. Gets the type of TDouble object and displays it. The function `g_object_new` is used to instantiate the object. The GObject API reference says that the function returns a pointer to a GObject instance but it actually returns a gpointer. Gpointer is the same as `void *` and it can be assigned to a pointer that points any type. So, the statement `d = g_object_new (T_TYPE_DOUBLE, NULL);` is correct. If the function `g_object_new` returned `GObject *`, it would be necessary to cast the returned pointer. After the creation, it shows the address of the instance. Finally, the instance is released and destroyed with the function `g_object_unref`.

`example3.c` is in the src/misc directory.

Execute it.

```
$ cd src/misc; _build/example3
Registration was a success. The type is 56414f164880.
Instantiation was a success. The instance address is 0x56414f167010.
```

3.5 G_DEFINE_TYPE macro

The registration above is always done with the same algorithm. Therefore, it can be defined as a macro such as `G_DEFINE_TYPE`.

`G_DEFINE_TYPE` does the following:

- Declares a class initialization function. Its name is `<name space>_<name>_class_init`. For example, if the object name is `TDouble`, it is `t_double_class_init`. This is a declaration, not a definition. You need to define it.
- Declares a instance initialization function. Its name is `<name space>_<name>_init`. For example, if the object name is `TDouble`, it is `t_double_init`. This is a declaration, not a definition. You need to define it.
- Defines a static variable pointing to the parent class. Its name is `<name space>_<name>_parent_class`. For example, if the object name is `TDouble`, it is `t_double_parent_class`.
- Defines a `<name space>_<name>_get_type ()` function. For example, if the object name is `TDouble`, it is `t_double_get_type`. The registration is done in this function like the previous subsection.

Using this macro reduces lines of the program. See the following sample `example4.c` which works the same as `example3.c`.

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

typedef struct _TDouble
{
  GObject parent;
  double  value;
} TDouble;

typedef struct _TDoubleClass
{
  GObjectClass parent_class;
} TDoubleClass;

static void
t_double_class_init (TDoubleClass *class)
{
}

static void
t_double_init (TDouble *self)
{
}

// creates t_double_get_type, …
G_DEFINE_TYPE (TDouble, t_double, G_TYPE_OBJECT)

#define T_DOUBLE_TYPE (t_double_get_type ())

int
main (void)
{
  GType dtype = T_DOUBLE_TYPE;

  if (dtype)
    {
```

```
37       g_print ("Type␣registration␣was␣successful!\n");
38       g_print ("The␣type␣ID␣is␣%lx.\n", dtype);
39     }
40   else
41     {
42       g_print ("Type␣registration␣failed.\n");
43       exit (EXIT_FAILURE);
44     }
45 
46   TDouble *d = g_object_new (T_DOUBLE_TYPE, NULL);
47   if (d)
48     {
49       g_print ("Instantiation␣was␣successful!\n");
50       g_print ("The␣instance␣address␣is␣%p.\n", d);
51     }
52   else
53     {
54       g_print ("Instantiation␣failed.\n");
55       exit (EXIT_FAILURE);
56     }
57 
58   g_object_unref (d);
59 }
```

Thanks to `G_DEFINE_TYPE`, we are freed from writing bothersome code like `GTypeInfo` and `g_type_register_static`. One important thing to be careful is to follow the convention of the naming of init functions.

Execute it.

```
$ cd src/misc; _build/example4
Registration was a success. The type is 564b4ff708a0.
Instantiation was a success. The instance address is 0x564b4ff71400.
```

You can use `G_DEFINE_FINAL_TYPE` instead of `G_DEFINE_TYPE` for final type classes since GLib version 2.70.

3.6 G_DECLARE_FINAL_TYPE macro

Another useful macro is `G_DECLARE_FINAL_TYPE` macro. This macro can be used for a final type. A final type doesn't have any children. If a type has children, it is a derivable type. If you want to define a derivable type object, use `G_DECLARE_DERIVABLE_TYPE` instead. However, you probably want to write final type objects in most cases.

`G_DECLARE_FINAL_TYPE` does the following:

- Declares `<name space>_<name>_get_type ()` function. This is only declaration. You need to define it. But you can use `G_DEFINE_TYPE`, its expansion includes the definition of the function. So, you actually don't need to write the definition by yourself.
- The C type of the object is defined as a typedef of structure. For example, if the object name is `TDouble`, then `typedef struct _TDouble TDouble` is included in the expansion. But you need to define the structure `struct _TDouble` by yourself before `G_DEFINE_TYPE`.
- `<NAME SPACE>_<NAME>` macro is defined. For example, if the object is `TDouble` the macro is `T_DOUBLE`. It will be expanded to a function which casts the argument to the pointer to the object. For example, `T_DOUBLE (obj)` casts the type of `obj` to `TDouble *`.
- `<NAME SPACE>_IS_<NAME>` macro is defined. For example, if the object is `TDouble` the macro is `T_IS_DOUBLE`. It will be expanded to a function which checks if the argument points the instance of `TDouble`. It returns true if the argument points a descendant of `TDouble`.
- The class structure is defined. A final type object doesn't need to have its own member of class structure. The definition is like the line 11 to 14 in the `example4.c`.

You need to write the macro definition of the type of the object before `G_DECLARE_FINAL_TYPE`. For example, if the object is `TDouble`, then

```
#define T_TYPE_DOUBLE  (t_double_get_type ())
```

needs to be defined before `G_DECLARE_FINAL_TYPE`.

The C file `example5.c` uses this macro. It works like `example3.c` or `example4.c`.

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

struct _TDouble
{
  GObject parent;
  double  value;
};

// creates TDouble, TDoubleClass, T_IS_DOUBLE, …
G_DECLARE_FINAL_TYPE (TDouble, t_double, T, DOUBLE, GObject)

// creates t_double_get_type, …
// G_DEFINE_TYPE (TDouble, t_double, G_TYPE_OBJECT)
G_DEFINE_FINAL_TYPE (TDouble, t_double, G_TYPE_OBJECT)

static void
t_double_class_init (TDoubleClass *class)
{
}

static void
t_double_init (TDouble *self)
{
}

#define T_DOUBLE_TYPE (t_double_get_type ())

int
main (void)
{
  GType dtype = T_DOUBLE_TYPE;

  if (dtype)
    {
      g_print ("Type␣registration␣was␣successful.\n");
      g_print ("The␣type␣ID␣is␣%lx.\n", dtype);
    }
  else
    {
      g_print ("Type␣registration␣failed.\n");
      exit (EXIT_FAILURE);
    }

  TDouble *d = g_object_new (T_DOUBLE_TYPE, NULL);
  if (d)
    {
      g_print ("Instantiation␣was␣successful.\n");
      g_print ("The␣instance␣address␣is␣%p.\n", d);
    }
  else
    {
      g_print ("Instantiation␣failed!\n");
      exit (EXIT_FAILURE);
    }

  if (T_IS_DOUBLE (d))
    {
      g_print ("d␣is␣a␣TDouble␣instance.\n");
    }
  else
    {
      g_print ("d␣is␣not␣a␣TDouble␣instance.\n");
    }

  if (G_IS_OBJECT (d))
    {
      g_print ("d␣is␣a␣GObject␣instance.\n");
    }
  else
    {
      g_print ("d␣is␣not␣a␣GObject␣instance.\n");
    }

  g_object_unref (d);
}
```

Execute it.

```
$ cd src/misc; _build/example5
Registration was a success. The type is 5560b4cf58a0.
Instantiation was a success. The instance address is 0x5560b4cf6400.
d is TDouble instance.
d is GObject instance.
```

3.7 Separate the file into main.c, tdouble.h and tdouble.c

Now it's time to separate the contents into three files, `main.c`, `tdouble.h` and `tdouble.c`. An object is defined by two files, a header file and C source file.

tdouble.h

```
1  #pragma once
2
3  #include <glib-object.h>
4
5  G_DECLARE_FINAL_TYPE (TDouble, t_double, T, DOUBLE, GObject)
6
7  gboolean t_double_get_value (TDouble *self, double *value);
8  void     t_double_set_value (TDouble *self, double value);
9  TDouble *t_double_new (double value);
```

- Header files are public, i.e. it is open to any files. Header files include macros, which gives type information, cast and type check, and public functions.
- 1: The directive `#pragma once` prevent the compiler from reading the header file two times or more. It is not officially defined but is supported widely in many compilers.
- 5-6: `T_TYPE_DOUBLE` is public. `G_DECLARE_FINAL_TYPE` is expanded to public definitions.
- 8-12: Public function declarations. They are getter and setter of the value of the object. They are called "instance methods", which are used in object-oriented languages.
- 14-15: Object instantiation function.

tdouble.c

```
#include "tdouble.h"

#define T_DOUBLE_TYPE (t_double_get_type ())

struct _TDouble
{
  GObject parent;
  double  value;
};

G_DEFINE_TYPE (TDouble, t_double, G_TYPE_OBJECT)

static void
t_double_class_init (TDoubleClass *class)
```

```
15 {
16 }
17
18 static void
19 t_double_init (TDouble *self)
20 {
21 }
22
23 gboolean
24 t_double_get_value (TDouble *self, double *value)
25 {
26   g_return_val_if_fail (T_IS_DOUBLE (self), FALSE);
27
28   *value = self->value;
29   return TRUE;
30 }
31
32 void
33 t_double_set_value (TDouble *self, double value)
34 {
35   g_return_if_fail (T_IS_DOUBLE (self));
36
37   self->value = value;
38 }
39
40 TDouble *
41 t_double_new (double value)
42 {
43   TDouble *d = g_object_new (T_DOUBLE_TYPE, NULL);
44   d->value   = value;
45   return d;
46 }
```

- 3-6: Declaration of the instance structure. Since `G_DECLARE_FINAL_TYPE` macro emits `typedef struct _TDouble TDouble`, the tag name of the structure must be `_TDouble`.
- 8: `G_DEFINE_TYPE` macro.
- 10-16: class and instance initialization functions. At present, they don't do anything.
- 18-24: Getter. The argument `value` is the pointer to a double type variable. Assigns the object value (`self->value`) to the variable. If it succeeds, it returns TRUE. The function `g_return_val_if_fail` is used to check the argument type. If the argument `self` is not TDouble type, it outputs error to the log and immediately returns FALSE. This function is used to report a programmer's error. You shouldn't use it for a runtime error. See Glib API Reference – Error Reporting for further information. The function `g_return_val_if_fail` isn't used in static class functions, which are private, because static functions are called only from functions in the same file and the caller knows the type of parameters.
- 26-31: Setter. The function `g_return_if_fail` is used to check the argument type. This function doesn't return any value. Because the type of `t_double_set_value` is `void` so no value will be returned. Therefore, we use `g_return_if_fail` instead of `g_return_val_if_fail`.
- 33-40: Object instantiation function. It has one parameter `value` to set the value of the object.
- 37: This function uses `g_object_new` to instantiate the object. The argument `T_TYPE_DOUBLE` is expanded to a function `t_double_get_type ()`. If this is the first call for `t_double_get_type`, the type registration will be carried out.

main.c

```
1 #include "tdouble.h"
2 #include <stdlib.h>
3
4 int
5 main (void)
6 {
7   double value;
8   double initial_val = 10.0;
9   double updated_val = -20.0;
10
11    g_print ("Initialising␣d␣to␣%.2f.\n", initial_val);
12    TDouble *d = t_double_new (initial_val);
13    if (t_double_get_value (d, &value))
14      {
15        g_print ("Succesfully␣read␣value␣from␣d␣and␣assigned␣its␣value␣%.2f␣to␣another␣
            variable.\n", value);
16      }
17    else
18      {
19        g_print ("t_double_get_value␣call␣failed.\n");
20        exit (EXIT_FAILURE);
21      }
22
23    t_double_set_value (d, updated_val);
24    g_print ("Setting␣d␣to␣%.2f.\n", updated_val);
25    if (t_double_get_value (d, &value))
26      {
27        g_print ("Succesfully␣read␣value␣from␣d␣and␣assigned␣its␣value␣%.2f␣to␣another␣
            variable.\n", value);
28      }
29    else
30      {
31        g_print ("t_double_get_value␣call␣failed.\n");
32        exit (EXIT_FAILURE);
33      }
34
35    g_object_unref (d);
36  }
```

- 2: Includes `tdouble.h`. This is necessary for accessing TDouble object.
- 9: Instantiate TDouble object and set `d` to point the object.
- 10-13: Tests the getter of the object.
- 15-20: Tests the setter of the object.
- 21: Releases the instance `d`.

The source files are located in src/tdouble1. Change your current directory to the directory above and type the following.

```
$ cd src/tdouble1
$ meson setup _build
$ ninja -C _build
```

Then, execute the program.

```
$ cd src/tdouble1; _build/example6
t_double_get_value succesfully assigned 10.000000 to value.
Now, set d (tDouble object) with -20.000000.
t_double_get_value succesfully assigned -20.000000 to value.
```

This example is very simple. But any object has header file and C source file like this. And they follow the convention. You probably aware of the importance of the convention. For the further information refer to GObject API Reference – Conventions.

3.8 Functions

Functions of objects are open to other objects. They are like public methods in object oriented languages. They are actually called "instance method" in the GObject API Reference.

It is natural to add calculation operators to TDouble objects because they represent real numbers. For example, `t_double_add` adds the value of the instance and another instance. Then it creates a new TDouble instance which has a value of the sum of them.

```
TDouble *
t_double_add (TDouble *self, TDouble *other) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);
  g_return_val_if_fail (T_IS_DOUBLE (other), NULL);
  double value;

  if (! t_double_get_value (other, &value))
    return NULL;
  return t_double_new (self->value + value);
}
```

The first argument `self` is the instance the function belongs to. The second argument `other` is another TDouble instance.

The value of `self` can be accessed by `self->value`, but don't use `other->value` to get the value of `other`. Use a function `t_double_get_value` instead. Because `self` is an instance out of `other`. Generally, the structure of an object isn't open to other objects. When an object A access to another object B, A must use a public function provided by B.

3.9 Exercise

Write functions of TDouble object for subtraction, multiplication, division and sign changing (unary minus). Compare your program to `tdouble.c` in src/tdouble2 directory.

4 Signals

4.1 Signals

Signals provide a means of communication between objects. Signals are emitted when something happens or completes.

The steps to program a signal is shown below.

1. Register a signal. A signal belongs to an object, so the registration is done in the class initialization function of the object.
2. Write a signal handler. A signal handler is a function that is invoked when the signal is emitted.
3. Connect the signal and handler. Signals are connected to handlers with `g_connect_signal` or its family functions.
4. Emit the signal.

Step one and Four are done on the object to which the signal belongs. Step three is usually done outside the object.

The process of signals is complicated and it takes long to explain all the features. The contents of this section is limited to the minimum things to write a simple signal and not necessarily accurate. If you need an accurate information, refer to GObject API reference. There are four parts which describe signals.

- Type System Concepts – signals
- Funcions (g_signal_XXX series)
- Funcions Macros (g_signal_XXX series)
- GObject Tutorial – How to create and use signals

4.2 Signal registration

An example in this section is a signal emitted when division-by-zero happens. First, we need to determine the name of the signal. Signal name consists of letters, digits, dash (`-`) and underscore (`_`). The first character of the name must be a letter. So, a string "div-by-zero" is appropriate for the signal name.

There are four functions to register a signal. We will use `g_signal_new` for "div-by-zero" signal.

```
guint
g_signal_new (const gchar *signal_name,
              GType itype,
              GSignalFlags signal_flags,
              guint class_offset,
              GSignalAccumulator accumulator,
              gpointer accu_data,
              GSignalCMarshaller c_marshaller,
              GType return_type,
              guint n_params,
              ...);
```

It needs a lot to explain each parameter. At present I just show you `g_signal_new` function call extracted from `tdouble.c`.

```
t_double_signal =
g_signal_new ("div-by-zero",
              G_TYPE_FROM_CLASS (class),
              G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
              0 /* class offset.Subclass cannot override the class handler (default
                 handler). */,
              NULL /* accumulator */,
              NULL /* accumulator data */,
              NULL /* C marshaller. g_cclosure_marshal_generic() will be used */,
              G_TYPE_NONE /* return_type */,
              0     /* n_params */
              );
```

- `t_double_signal` is a static guint variable. The type guint is the same as unsigned int. It is set to the signal id returned by the function `g_signal_new`.
- The second parameter is the type (GType) of the object the signal belongs to. `G_TYPE_FROM_CLASS (class)` returns the type corresponds to the class (`class` is a pointer to the class of the object).
- The third parameter is a signal flag. Lots of pages are necessary to explain this flag. So, I want leave them out now. The argument above can be used in many cases. The definition is described in the GObject API Reference – SignalFlags.
- The return type is G_TYPE_NONE which means no value is returned by the signal handler.
- `n_params` is a number of parameters. This signal doesn't have parameters, so it is zero.

This function is located in the class initialization function (`t_double_class_init`).

You can use other functions such as `g_signal_newv`. See GObject API Reference for details.

4.3 Signal handler

Signal handler is a function that is called when the signal is emitted. The handler has two parameters.

- The instance to which the signal belongs
- A pointer to a user data which is given in the signal connection.

The "div-by-zero" signal doesn't need user data.

```
void div_by_zero_cb (TDouble *self, gpointer user_data) { ... ... ...}
```

The first argument `self` is the instance on which the signal is emitted. You can leave out the second parameter.

```
void div_by_zero_cb (TDouble *self) { ... ... ...}
```

If a signal has parameters, the parameters are between the instance and the user data. For example, the handler of "window-added" signal on GtkApplication is:

```
void window_added (GtkApplication* self, GtkWindow* window, gpointer user_data);
```

The second argument `window` is the parameter of the signal. The "window-added" signal is emitted when a new window is added to the application. The parameter `window` points a newly added window. See GTK API reference for further information.

The handler of "div-by-zero" signal just shows an error message.

```
static void
div_by_zero_cb (TDouble *self, gpointer user_data) {
  g_print ("\nError:␣division␣by␣zero.\n\n");
}
```

4.4 Signal connection

A signal and a handler are connected with the function `g_signal_connect`.

```
g_signal_connect (self, "div-by-zero", G_CALLBACK (div_by_zero_cb), NULL);
```

- `self` is an instance the signal belongs to.
- The second argument is the signal name.
- The third argument is the signal handler. It must be casted by `G_CALLBACK`.
- The last argument is an user data. The signal doesn't need a user data, so NULL is assigned.

4.5 Signal emission

Signals are emitted on the object. The following is a part of `tdouble.c`.

```
TDouble *
t_double_div (TDouble *self, TDouble *other) {
... ... ...
  if ((! t_double_get_value (other, &value)))
    return NULL;
  else if (value == 0) {
    g_signal_emit (self, t_double_signal, 0);
    return NULL;
  }
  return t_double_new (self->value / value);
}
```

If the divisor is zero, the signal is emitted. `g_signal_emit` has three parameters.

- The first parameter is the instance that emits the signal.
- The second parameter is the signal id. Signal id is the value returned by the function `g_signal_new`.
- The third parameter is a detail. "div-by-zero" signal doesn't have a detail, so the argument is zero. Detail isn't explained in this section but usually you can put zero as a third argument. If you want to know the details, refer to GObject API Reference – Signal Detail.

If a signal has parameters, they are fourth and subsequent arguments.

4.6 Example

A sample program is in src/tdouble3.

tdouble.h

```
#pragma once

#include <glib-object.h>

G_DECLARE_FINAL_TYPE (TDouble, t_double, T, DOUBLE, GObject)

/* getter and setter */
gboolean t_double_get_value (TDouble *self, double *value);
void     t_double_set_value (TDouble *self, double value);
TDouble *t_double_add (TDouble *self, TDouble *other);
TDouble *t_double_sub (TDouble *self, TDouble *other);
TDouble *t_double_mul (TDouble *self, TDouble *other);
TDouble *t_double_div (TDouble *self, TDouble *other);
TDouble *t_double_uminus (TDouble *self);
TDouble *t_double_new (double value);
```

tdouble.c

```
#include "tdouble.h"

#define T_TYPE_DOUBLE (t_double_get_type ())

static guint t_double_signal;
```

```c

struct _TDouble
{
  GObject parent;
  double  value;
};

G_DEFINE_TYPE (TDouble, t_double, G_TYPE_OBJECT)

static void
t_double_class_init (TDoubleClass *class)
{
  GSignalFlags signal_flags = G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE |
      G_SIGNAL_NO_HOOKS;
  t_double_signal
      = g_signal_new ("div-by-zero", G_TYPE_FROM_CLASS (class), signal_flags, 0,
          NULL, NULL, NULL, G_TYPE_NONE, 0);
}

static void
t_double_init (TDouble *self)
{
}

/* getter and setter */
gboolean
t_double_get_value (TDouble *self, double *value)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), FALSE);

  *value = self->value;
  return TRUE;
}

void
t_double_set_value (TDouble *self, double value)
{
  g_return_if_fail (T_IS_DOUBLE (self));

  self->value = value;
}

/* arithmetic operator */
/* These operators create a new instance and return a pointer to it. */
#define t_double_binary_op(op) \
  if (!t_double_get_value (other, &value)) \
    return NULL; \
  return t_double_new (self->value op value);

TDouble *
t_double_add (TDouble *self, TDouble *other)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);
  g_return_val_if_fail (T_IS_DOUBLE (other), NULL);
  double value;

  t_double_binary_op (+)
}

```

```c
TDouble *
t_double_sub (TDouble *self, TDouble *other)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);
  g_return_val_if_fail (T_IS_DOUBLE (other), NULL);
  double value;

  t_double_binary_op (-)
}

TDouble *
t_double_mul (TDouble *self, TDouble *other)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);
  g_return_val_if_fail (T_IS_DOUBLE (other), NULL);
  double value;

  t_double_binary_op (*)
}

TDouble *
t_double_div (TDouble *self, TDouble *other)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);
  g_return_val_if_fail (T_IS_DOUBLE (other), NULL);
  double value;

  if ((!t_double_get_value (other, &value)))
    {
      return NULL;
    }
  else if (value == 0)
    {
      g_signal_emit (self, t_double_signal, 0);
      return NULL;
    }
  return t_double_new (self->value / value);
}

TDouble *
t_double_uminus (TDouble *self)
{
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  return t_double_new (-self->value);
}

TDouble *
t_double_new (double value)
{
  TDouble *d;

  d        = g_object_new (T_TYPE_DOUBLE, NULL);
  d->value = value;
  return d;
}
```

main.c

```c
#include "tdouble.h"
#include <glib-object.h>

static void
div_by_zero_cb (TDouble *self, gpointer user_data)
{
  g_printerr ("\nError:␣division␣by␣zero.\n\n");
}

static void
t_print (char *op, TDouble *d1, TDouble *d2, TDouble *d3)
{
  double v1, v2, v3;

  if (!t_double_get_value (d1, &v1))
    {
      return;
    }
  if (!t_double_get_value (d2, &v2))
    {
      return;
    }
  if (!t_double_get_value (d3, &v3))
    {
      return;
    }

  g_print ("%lf␣%s␣%lf␣=␣%lf\n", v1, op, v2, v3);
}

int
main (void)
{
  TDouble *d1 = t_double_new (10.0);
  TDouble *d2 = t_double_new (20.0);
  TDouble *d3;
  if ((d3 = t_double_add (d1, d2)) != NULL)
    {
      t_print ("+", d1, d2, d3);
      g_object_unref (d3);
    }

  if ((d3 = t_double_sub (d1, d2)) != NULL)
    {
      t_print ("-", d1, d2, d3);
      g_object_unref (d3);
    }

  if ((d3 = t_double_mul (d1, d2)) != NULL)
    {
      t_print ("*", d1, d2, d3);
      g_object_unref (d3);
    }

  if ((d3 = t_double_div (d1, d2)) != NULL)
    {
      t_print ("/", d1, d2, d3);
      g_object_unref (d3);
    }

  g_signal_connect (d1, "div-by-zero", G_CALLBACK (div_by_zero_cb), NULL);
  t_double_set_value (d2, 0.0);
  if ((d3 = t_double_div (d1, d2)) != NULL)
    {
      t_print ("/", d1, d2, d3);
      g_object_unref (d3);
    }

  double v1, v3;

  if ((d3 = t_double_uminus (d1)) != NULL && (t_double_get_value (d1, &v1)) &&
      (t_double_get_value (d3, &v3)))
    {
      g_print ("-%lf␣=␣%lf\n", v1, v3);
      g_object_unref (d3);
    }

  g_object_unref (d1);
  g_object_unref (d2);
}
```

Change your current directory to src/tdouble3 and type as follows.

```
$ meson setup _build
$ ninja -C _build
```

Then, Executable file `tdouble` is created in the `_build` directory. Execute it.

```
$ _build/tdouble
10.000000 + 20.000000 = 30.000000
10.000000 - 20.000000 = -10.000000
10.000000 * 20.000000 = 200.000000
10.000000 / 20.000000 = 0.500000

Error: division by zero.

-10.000000 = -10.000000
```

4.7 Default signal handler

You may have thought that it was strange that the error message was set in `main.c`. Indeed, the error happens in `tdouble.c` so the message should been managed by `tdouble.c` itself. GObject system has a default signal handler that is set in the object itself. A default signal handler is also called “default handler” or “object method handler”.

You can set a default handler with `g_signal_new_class_handler`.

```
guint
g_signal_new_class_handler (const gchar *signal_name,
                            GType itype,
                            GSignalFlags signal_flags,
                            GCallback class_handler, /*default signal handler */
                            GSignalAccumulator accumulator,
                            gpointer accu_data,
                            GSignalCMarshaller c_marshaller,
                            GType return_type,
                            guint n_params,
                            ...);
```

The difference from `g_signal_new` is the fourth parameter. `g_signal_new` sets a default handler with the offset of the function pointer in the class structure. If an object is derivable, it has its own class area, so you can set a default handler with `g_signal_new`. But a final type object doesn’t have its own class area, so it’s impossible to set a default handler with `g_signal_new`. That’s the reason why we use `g_signal_new_class_handler`.

The C file `tdouble.c` is changed like this. The function `div_by_zero_default_cb` is added and `g_signal_new_class_handler` replaces `g_signal_new`. Default signal handler doesn’t have `user_data` parameter. A `user_data` parameter is set in the `g_signal_connect` family functions when a user connects their own signal handler to the signal. Default signal handler is managed by the instance, not a user. So no user data is given as an argument.

```
static void
div_by_zero_default_cb (TDouble *self) {
  g_printerr ("\nError:␣division␣by␣zero.\n\n");
}
```

```

static void
t_double_class_init (TDoubleClass *class) {
  t_double_signal =
  g_signal_new_class_handler ("div-by-zero",
                              G_TYPE_FROM_CLASS (class),
                              G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE |
                                  G_SIGNAL_NO_HOOKS,
                              G_CALLBACK (div_by_zero_default_cb),
                              NULL /* accumulator */,
                              NULL /* accumulator data */,
                              NULL /* C marshaller */,
                              G_TYPE_NONE /* return_type */,
                              0     /* n_params */
                              );
}
```

`g_signal_connect` and `div_by_zero_cb` are removed from `main.c`.

Compile and execute it.

```
$ cd src/tdouble4; _build/tdouble
10.000000 + 20.000000 = 30.000000
10.000000 - 20.000000 = -10.000000
10.000000 * 20.000000 = 200.000000
10.000000 / 20.000000 = 0.500000

Error: division by zero.

-10.000000 = -10.000000
```

The source file is in the directory src/tdouble4.

If you want to connect your handler (user-provided handler) to the signal, you can still use `g_signal_connect`. Add the following in `main.c`.

```
static void
div_by_zero_cb (TDouble *self, gpointer user_data) {
  g_print ("\nError␣happens␣in␣main.c.\n");
}

int
main (int argc, char **argv) {
... ... ...
  g_signal_connect (d1, "div-by-zero", G_CALLBACK (div_by_zero_cb), NULL);
... ... ...
}
```

Then, both the user-provided handler and default handler are called when the signal is emitted. Compile and execute it, then the following is shown on your display.

```
10.000000 + 20.000000 = 30.000000
10.000000 - 20.000000 = -10.000000
10.000000 * 20.000000 = 200.000000
10.000000 / 20.000000 = 0.500000

Error happens in main.c.

Error: division by zero.

-10.000000 = -10.000000
```

This tells us that the user-provided handler is called first, then the default handler is called. If you want your handler called after the default handler, then you can use `g_signal_connect_after`. Add the lines below to `main.c` again.

```
static void
div_by_zero_after_cb (TDouble *self, gpointer user_data) {
  g_print ("\nError␣has␣happened␣in␣main.c␣and␣an␣error␣message␣has␣been␣
      displayed.\n");
}

int
main (int argc, char **argv) {
... ... ...
  g_signal_connect_after (d1, "div-by-zero", G_CALLBACK (div_by_zero_after_cb),
      NULL);
... ... ...
}
```

Compile and execute it, then:

```
10.000000 + 20.000000 = 30.000000
10.000000 - 20.000000 = -10.000000
10.000000 * 20.000000 = 200.000000
10.000000 / 20.000000 = 0.500000

Error happens in main.c.

Error: division by zero.

Error has happened in main.c and an error message has been displayed.

-10.000000 = -10.000000
```

The source files are in src/tdouble5.

4.8 Signal flag

The order that handlers are called is described in GObject API Reference – Sigmal emission.

The order depends on the signal flag which is set in `g_signal_new` or `g_signal_new_class_handler`. There are three flags which relate to the order of handlers' invocation.

- `G_SIGNAL_RUN_FIRST`: the default handler is called before any user provided handler.
- `G_SIGNAL_RUN_LAST`: the default handler is called after the normal user provided handler (not connected with `g_signal_connect_after`).
- `G_SIGNAL_RUN_CLEANUP`: the default handler is called after any user provided handler.

`G_SIGNAL_RUN_LAST` is the most appropriate in many cases.

Other signal flags are described in GObject API Reference.

5 Properties

GObject system provides properties. Properties are values kept by instances, which is a descendant of GObject, and they are open to other instances. They can be accessed with their names.

For example, GtkWindow has "title", "default-width", "default-height" and other properties. The string "title" is the name of the property. The name of a property is a string that begins with a letter followed by letters, digits, dash ('-') or underscore ('_'). Dash and underscore is used as separators but they cannot be mixed. Using dash is more efficient than underscore. For example, "value", "double" and "double-value" are correct property names. "_value" or "-value" are incorrect.

Properties have various types of values. The type of "title" property is string. The type of "default-width" and "default-height" is integer.

Properties are set and got with functions defined in GObject.

- Properties can be set with several GObject functions. `g_object_new` and `g_object_set` are often used.
- Properties can be get with several GObject functions. `g_object_get` is often used.

The functions above belongs to GObject, but they can be used for any descendant object of GObject. The following is an example of GtkWindow, which is a descendant object of GObject.

An instance is created and its properties are set with `g_object_new`.

```
GtkWindow *win;
win = g_object_new (GTK_TYPE_WINDOW, "title", "Hello", "default-width", 800,
    "default-height", 600, NULL);
```

The example above creates an instance of GtkWindow and sets the properties.

- The "title" property is set to "Hello".
- The "default-width" property is set to 800.
- The "default-height" property is set to 600.

The last parameter of `g_object_new` is `NULL` which is the end of the list of properties.

If you have already created a GtkWindow instance and you want to set its title, you can use `g_object_set`.

```
GtkWindow *win;
win = g_object_new (GTK_TYPE_WINDOW, NULL);
g_object_set (win, "title", "Good␣bye", NULL);
```

You can get the value of a property with `g_object_get`.

```
GtkWindow *win;
char *title;
int width, height;

win = g_object_new (GTK_TYPE_WINDOW, "title", "Hello", "default-width", 800,
    "default-height", 600, NULL);
g_object_get (win, "title", &title, "default-width", &width, "default-height",
    &height, NULL);
g_print ("%s,␣%d,␣%d\n", title, width, height);
g_free (title);
```

The rest of this section is about implementing properties in a descendant of GObject. It is divided into two things.

- Register a property
- Define `set_property` and `get_property` class method to complement `g_object_set` and `g_object_get`.

5.1 GParamSpec

GParamSpec is a fundamental object. GParamSpec and GObject don't have parent-child relationship. GParamSpec has information of parameters. "ParamSpec" is short for "Parameter specification".

For example,

```
double_property = g_param_spec_double ("value", "val", "Double␣value",
                                       -G_MAXDOUBLE, G_MAXDOUBLE, 0.0,
                                        G_PARAM_READWRITE);
```

This function creates a GParamSpec instance, more precisely a GParamSpecDouble instance. GParamSpecDouble is a child of GParamSpec.

The instance has information:

- The value type is double. The suffix of the function name, `double` in `g_param_spec_double`, implies the type.
- The name is "value".
- The nick name is "val".
- The description is "Double value".
- The minimum value is -G_MAXDOUBLE. G_MAXDOUBLE is the maximum value which can be held in a double. It is described in GLib(2.68.1) Reference Manual – G_MAXDOUBLE and G_MINDOUBLE. You might think the lowest value of double is G_MINDOUBLE, but it's not. G_MINDOUBLE is the minimum positive value which can be held in a double.

- The maximum value is G_MAXDOUBLE.
- The default value is 0.0.
- `G_PARAM_READWRITE` is a flag. `G_PARAM_READWRITE` means that the parameter is readable and writable.

For further information, refer to the GObject API reference.

- GParamSpec and its subclasses
- g_param_spec_double and similar functions
- GValue

GParamSpec is used for the registration for GObject properties. This is extracted from tdouble.c in src/tdouble6.

```
#define PROP_DOUBLE 1
static GParamSpec *double_property = NULL;

static void
t_double_class_init (TDoubleClass *class) {
  GObjectClass *gobject_class = G_OBJECT_CLASS (class);

  gobject_class->set_property = t_double_set_property;
  gobject_class->get_property = t_double_get_property;
  double_property = g_param_spec_double ("value", "val", "Double␣value",
      -G_MAXDOUBLE, G_MAXDOUBLE, 0.0, G_PARAM_READWRITE);
  g_object_class_install_property (gobject_class, PROP_DOUBLE, double_property);
}
```

The variable `double_property` is static. GParamSpec instance is assigned to `double_property`.

The function `g_object_class_install_property` installs a property. It must be called after `set_property` and `get_property` methods are overridden. These methods will be explained later. The arguments are TDoubleClass class, PROP_DOUBLE (property id) and GParamSpec instance. Property id is used to identify the property in `tdouble.c`. It is a positive integer.

5.2 Overriding set_property and get_property class methods

Property values vary from instance to instance. Therefore, the value is stored to each instance of the object.

The function `g_object_set` is given a value as an argument and stores the value. But how does `g_object_set` know the instance to store? It is compiled before the object is made. So, it doesn't know where to store the value at all. That part needs to be programmed by the writer of the object with overriding.

The function `g_object_set` first checks the property and value, then it creates GValue (generic value) from the value. And it calls a function pointed by `set_property` in the class. Look at the diagram below.

The member `set_property` in GObjectClass class points `g_object_do_set_property` in GObject program, which is made by compiling `gobject.c`. The GObjectClass part of the TDoubleClass structure (it is the same as TDoubleClass because TDoubleClass doesn't have its own area) is initialized by copying from the contents of GObjectClass. Therefore, `set_property` in TDoubleClass class points `g_object_do_set_property` in GObject program. But `g_object_do_set_property` doesn't store the value to the TDouble instance. The writer of TDouble object makes `t_double_set_property` function in `tdouble.c`. And assigns the address of `t_double_set_property` to `set_property` in TDoubleClass class. It is shown with a red curve in the diagram. As a result, `g_object_set` calls `t_double_set_property` instead of `g_object_do_set_property` (red dotted curve) and the value will be stored in the TDouble instance. See the function `t_double_class_init` above. It changes the member `gobject_class->set_property` to point the function `t_double_set_property`. The function `g_object_set` sees the TDoubleClass and call the function pointed by the member `set_property`.

The program of `t_double_set_property` and `t_double_get_property` will shown later.

5.3 GValue

GValue is generic value. GValue consists of type and value.

The type is any Gtype. The table below shows some GType, but not all.

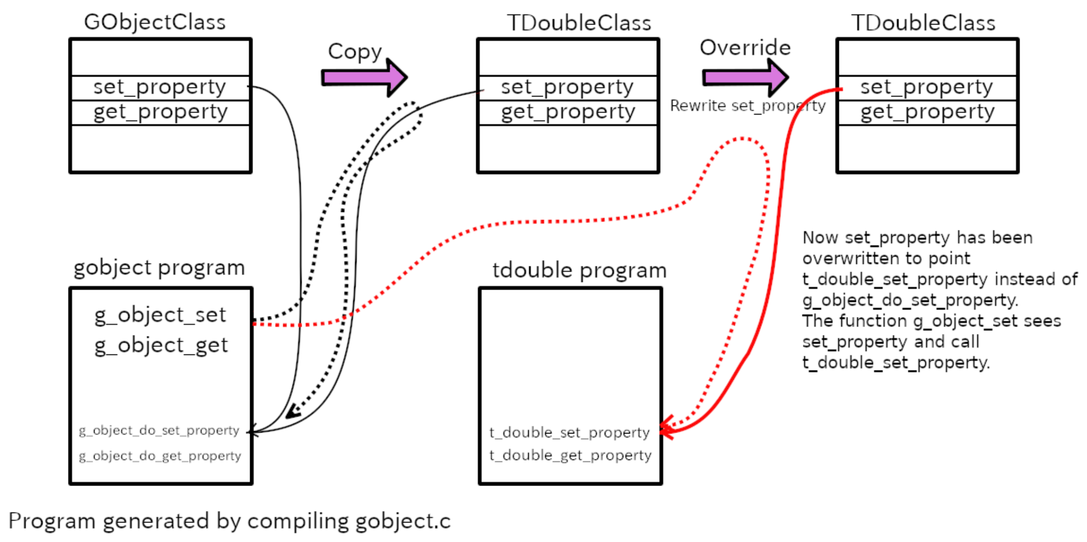

Figure 2: Overriding `set_property` class method

GType	C type	type name	notes
G_TYPE_CHAR	char	gchar	
G_TYPE_BOOLEAN	int	gboolean	
G_TYPE_INT	int	gint	
G_TYPE_FLOAT	float	gfloat	
G_TYPE_DOUBLE	double	gdouble	
G_TYPE_STRING		gchararray	null-terminated Cstring
G_TYPE_PARAM		GParam	GParamSpec
G_TYPE_OBJECT		GObject	
G_TYPE_VARIANT		GVariant	

If the type of a GValue `value` is `G_TYPE_DOUBLE`, `value` can be get with `g_value_get_double` function.

```
GValue value;
value = ... ... ... (a GValue object is assigned. Its type is double.)
double v;
v = g_value_get_double (&value);
```

Conversely, you can set GValue `value` with `g_value_set_double`.

```
g_value_set_double (value, 123.45);
```

Refer to GObject API Reference – GValue for further information.

5.4 t_double_set_property and t_double_get_property

`g_object_set` makes GValue from the value of the property given as an argument. And calls a function pointed by `set_property` in the class. The function is declared in GObjectClass structure.

```
struct  _GObjectClass
{
  ... ... ...
  ... ... ...
  /* overridable methods */
  void       (*set_property)    (GObject        *object,
                                  guint           property_id,
                                  const GValue   *value,
                                  GParamSpec     *pspec);
  void       (*get_property)    (GObject        *object,
                                             guint            property_id,
                                             GValue          *value,
                                             GParamSpec      *pspec);
  ... ... ...
  ... ... ...
};
```

`t_double_set_property` just get the value from GValue `value` and store it to the TDouble instance.

```
1  static void
2  t_double_set_property (GObject *object, guint property_id, const GValue *value,
       GParamSpec *pspec) {
3    TDouble *self = T_DOUBLE (object);
4
5    if (property_id == PROP_DOUBLE)
6      self->value = g_value_get_double (value);
7    else
8      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
9  }
```

- 3: Casts `object` to TDouble object `self`.
- 6: Set `self->value`. The assigned value is got with `g_value_get_double` function.

In the same way, `t_double_get_property` stores `self->value` to GValue.

```
static void
t_double_get_property (GObject *object, guint property_id, GValue *value, GParamSpec
    *pspec) {
  TDouble *self = T_DOUBLE (object);

  if (property_id == PROP_DOUBLE)
    g_value_set_double (value, self->value);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);

}
```

5.5 Notify signal

GObject emits "notify" signal when a property is set. When you connect "notify" signal to your handler, you can specify a detail which is the name of the property. The detail is added to the signal name with the delimiter "::".

```
g_signal_connect (G_OBJECT (d1), "notify::value", G_CALLBACK (notify_cb), NULL);
```

If you don't specify details, the handler is called whenever any properties are set. So, usually the detail is set.

Notify signal doesn't mean that the value of the property is changed. It is emitted even if the same value is set. You might want to emit the notify signal only when the property is actually changed. In that case, you define the GPramSpec with `G_PARAM_EXPLICIT_NOTIFY` flag. Then, the notify signal isn't emitted automatically. Instead you call `g_object_notify_by_pspec` function to emit "notify" signal explicitly when the value of the property is actually changed.

It is possible to make setter and getter for the property. But if you just set the instance member in your setter, notify signal isn't emitted.

```
void
t_double_set_value (TDouble *self, double value) {
  g_return_if_fail (T_IS_DOUBLE (self));

  self->value = value; /* Just set d->value. No "notify" signal is emitted. */
}
```

Users must be confused if they want to catch the "notify" signal. One solution is use `g_object_set` in your setter. Then, notify signal will be emitted even if a user uses the setter function.

```c
void
t_double_set_value (TDouble *d, double value) {
  g_return_if_fail (T_IS_DOUBLE (d));

  g_object_set (d, "value", value, NULL); /* Use g_object_set. "notify" signal will
      be emitted. */
}
```

The other solution is use `g_object_notify_by_pspec` to emit the signal explicitly. Anyway, if you make a setter for your property, be careful about notify signal.

5.6 Define more than one property

If you define more than one property, use an array of property id. It is good for you to see Gtk source files such as `gtklabel.c`. GtkLabel has 18 properties.

There's an example in src/tdouble6 directory.

5.7 Exercise

Make TInt object. It is like TDouble but the value type is int. Define "div-by-zero" signal and "value" property.

Compare your answer to the files in src/tint directory.

6 Derivable type and abstract type

6.1 Derivable type

There are two kinds of types, final type and derivable type. Final type doesn't have any child object. Derivable type has child objects.

The main difference between the two objects is their classes. Final type objects doesn't have its own class area. The only member of the class is its parent class.

Derivable object has its own area in the class. The class is open to its descendants.

`G_DECLARE_DERIVABLE_TYPE` is used to declare derivable type. It is written in a header file like this:

```c
#define T_TYPE_NUMBER             (t_number_get_type ())
G_DECLARE_DERIVABLE_TYPE (TNumber, t_number, T, NUMBER, GObject)
```

6.2 Abstract type

Abstract type doesn't have any instance. This type of object is derivable and its children can use functions and signals of the abstract object.

The examples of this section are TNumber, TInt and TDouble object. TInt and TDouble have already been made in the previous section. They represent integer and floating point respectively. Numbers are more abstract than integer and floating point.

TNumber is an abstract object which represents numbers. TNumber is a parent object of TInt and TDouble. TNumber isn't instantiated because its type is abstract. When an instance has TInt or TDouble type, it is an instance of TNumber as well.

TInt and TDouble have five operations: addition, subtraction, multiplication, division and unary minus operation. Those operations can be defined on TNumber object.

In this section we will define TNumber object and five functions above. In addition, `to_s` function will be added. It converts the value of TNumber into a string. It is like sprintf function. And we will rewrite TInt and TDouble to implement the functions.

6.3 TNumber class

`tnumber.h` is a header file for the TNumber class.

```
 1 #pragma once
 2 
 3 #include <glib-object.h>
 4 
 5 #define T_TYPE_NUMBER             (t_number_get_type ())
 6 G_DECLARE_DERIVABLE_TYPE (TNumber, t_number, T, NUMBER, GObject)
 7 
 8 struct _TNumberClass {
 9   GObjectClass parent_class;
10   TNumber* (*add) (TNumber *self, TNumber *other);
11   TNumber* (*sub) (TNumber *self, TNumber *other);
12   TNumber* (*mul) (TNumber *self, TNumber *other);
13   TNumber* (*div) (TNumber *self, TNumber *other);
14   TNumber* (*uminus) (TNumber *self);
15   char * (*to_s) (TNumber *self);
16   /* signal */
17   void (*div_by_zero) (TNumber *self);
18 };
19 
20 /* arithmetic operator */
21 /* These operators create a new instance and return a pointer to it. */
22 TNumber *
23 t_number_add (TNumber *self, TNumber *other);
24 
25 TNumber *
26 t_number_sub (TNumber *self, TNumber *other);
27 
28 TNumber *
29 t_number_mul (TNumber *self, TNumber *other);
30 
31 TNumber *
32 t_number_div (TNumber *self, TNumber *other);
33 
34 TNumber *
35 t_number_uminus (TNumber *self);
36 
37 char *
38 t_number_to_s (TNumber *self);
```

- 6: `G_DECLARE_DERIVABLE_TYPE` macro. This is similar to `G_DECLARE_FINAL_TYPE` macro. The difference is derivable or final. `G_DECLARE_DERIVABLE_TYPE` is expanded to:
  - Declaration of `t_number_get_type ()` function. This function must be defined in `tnumber.c` file. The definition is usually done with `G_DEFINE_TYPE` or its family macros.
  - Definition of TNumber instance, whose member is its parent only.
  - Declaration of TNumberClass. It should be defined later in the header file.
  - Convenience macros `T_NUMBER` (cast to instance), `T_NUMBER_CLASS` (cast to class), `T_IS_NUMBER` (instance check), `T_IS_NUMBER_CLASS` (class check) and `T_NUMBER_GET_CLASS` are defined.
  - `g_autoptr()` support.
- 8-18: Definition of the structure of TNumberClass.
- 10-15: These are pointers to functions. They are called class methods or virtual functions. They are expected to be overridden in the descendant object. The methods are five arithmetic operators and `to_s` function. `to_s` function is similar to sprintf function.
- 17: A pointer to the default signal handler of "div-by-zero" signal. The offset of this pointer is given to `g_signal_new` as an argument.
- 22-38: Functions. They are public.

`tnumber.c` is as follows.

```
1 #include "tnumber.h"
2 
```

```
static guint t_number_signal;

G_DEFINE_ABSTRACT_TYPE (TNumber, t_number, G_TYPE_OBJECT)

static void
div_by_zero_default_cb (TNumber *self) {
  g_printerr ("\nError:␣division␣by␣zero.\n\n");
}

static void
t_number_class_init (TNumberClass *class) {

  /* virtual functions */
  class->add = NULL;
  class->sub = NULL;
  class->mul = NULL;
  class->div = NULL;
  class->uminus = NULL;
  class->to_s = NULL;
  /* default signal handler */
  class->div_by_zero = div_by_zero_default_cb;
  /* signal */
  t_number_signal =
  g_signal_new ("div-by-zero",
                G_TYPE_FROM_CLASS (class),
                G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
                G_STRUCT_OFFSET (TNumberClass, div_by_zero),
                NULL /* accumulator */,
                NULL /* accumulator data */,
                NULL /* C marshaller */,
                G_TYPE_NONE /* return_type */,
                0     /* n_params */
                );
}

static void
t_number_init (TNumber *self) {
}

TNumber *
t_number_add (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_NUMBER (self), NULL);
  g_return_val_if_fail (T_IS_NUMBER (other), NULL);

  TNumberClass *class = T_NUMBER_GET_CLASS(self);

  return class->add ? class->add (self, other) : NULL;
}

TNumber *
t_number_sub (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_NUMBER (self), NULL);
  g_return_val_if_fail (T_IS_NUMBER (other), NULL);

  TNumberClass *class = T_NUMBER_GET_CLASS(self);

  return class->sub ? class->sub (self, other) : NULL;
}

TNumber *
t_number_mul (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_NUMBER (self), NULL);
  g_return_val_if_fail (T_IS_NUMBER (other), NULL);

67    TNumberClass *class = T_NUMBER_GET_CLASS(self);
68
69    return class->mul ? class->mul (self, other) : NULL;
70  }
71
72  TNumber *
73  t_number_div (TNumber *self, TNumber *other) {
74    g_return_val_if_fail (T_IS_NUMBER (self), NULL);
75    g_return_val_if_fail (T_IS_NUMBER (other), NULL);
76
77    TNumberClass *class = T_NUMBER_GET_CLASS(self);
78
79    return class->div ? class->div (self, other) : NULL;
80  }
81
82  TNumber *
83  t_number_uminus (TNumber *self) {
84    g_return_val_if_fail (T_IS_NUMBER (self), NULL);
85
86    TNumberClass *class = T_NUMBER_GET_CLASS(self);
87
88    return class->uminus ? class->uminus (self) : NULL;
89  }
90
91  char *
92  t_number_to_s (TNumber *self) {
93    g_return_val_if_fail (T_IS_NUMBER (self), NULL);
94
95    TNumberClass *class = T_NUMBER_GET_CLASS(self);
96
97    return class->to_s ? class->to_s (self) : NULL;
98  }
```

- 5: `G_DEFINE_ABSTRACT_TYPE` macro. This macro is used to define an abstract type object. Abstract type isn't instantiated. This macro is expanded to:
  - Declaration of `t_number_init ()` function.
  - Declaration of `t_number_class_init ()` function.
  - Definition of `t_number_get_type ()` function.
  - Definition of `t_number_parent_class` static variable that points the parent class.
- 3, 7-10, 26-35: Defines division-by-zero signal. The function `div_by_zero_default_cb` is a default handler of "div-by-zero" signal. Default handler doesn't have user data parameter. The function `g_signal_new` is used instead of `g_signal_new_class_handler`. It specifies a handler as the offset from the top of the class to the pointer to the handler.
- 12-36: The class initialization function `t_number_class_init`.
- 16-21: These class methods are virtual functions. They are expected to be overridden in the descendant object of TNumber. NULL is assigned here so that nothing happens when the methods are called.
- 23: Assigns the address of the function `dev_by_zero_default_cb` to `class->div_by_zero`. This is the default handler of "div-by-zero" signal.
- 38-40: `t_number_init` is a initialization function for an instance. But abstract object isn't instantiated. So, nothing is done in this function. But you can't leave out the definition of this function.
- 42-98: Public functions. These functions just call the corresponding class methods if the pointer to the class method is not NULL.

6.4 TInt object.

`tint.h` is a header file of the TInt class. TInt is a child class of TNumber.

```
#pragma once

#include <glib-object.h>

#define T_TYPE_INT  (t_int_get_type ())
G_DECLARE_FINAL_TYPE (TInt, t_int, T, INT, TNumber)
```

```c
 7
 8 /* create a new TInt instance */
 9 TInt *
10 t_int_new_with_value (int value);
11
12 TInt *
13 t_int_new (void);
```

- 9-13:Declares public functions. Arithmetic functions and `to_s` are declared in TNumber, so TInt doesn't declare those functions. Only instance creation functions are declared.

The C file `tint.c` implements virtual functions (class methods). And the pointers of the methods in TNumberClass are rewritten here.

```c
 1 #include "tnumber.h"
 2 #include "tint.h"
 3 #include "tdouble.h"
 4
 5 #define PROP_INT 1
 6 static GParamSpec *int_property = NULL;
 7
 8 struct _TInt {
 9   TNumber parent;
10   int value;
11 };
12
13 G_DEFINE_TYPE (TInt, t_int, T_TYPE_NUMBER)
14
15 static void
16 t_int_set_property (GObject *object, guint property_id, const GValue *value,
       GParamSpec *pspec) {
17   TInt *self = T_INT (object);
18
19   if (property_id == PROP_INT)
20     self->value = g_value_get_int (value);
21   else
22     G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
23 }
24
25 static void
26 t_int_get_property (GObject *object, guint property_id, GValue *value, GParamSpec
       *pspec) {
27   TInt *self = T_INT (object);
28
29   if (property_id == PROP_INT)
30     g_value_set_int (value, self->value);
31   else
32     G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
33 }
34
35 static void
36 t_int_init (TInt *self) {
37 }
38
39 /* arithmetic operator */
40 /* These operators create a new instance and return a pointer to it. */
41 #define t_int_binary_op(op) \
42   int i; \
43   double d; \
44   if (T_IS_INT (other)) { \
45     g_object_get (T_INT (other), "value", &i, NULL); \
46     return  T_NUMBER (t_int_new_with_value (T_INT(self)->value op i)); \
47   } else { \
48     g_object_get (T_DOUBLE (other), "value", &d, NULL); \
49     return  T_NUMBER (t_int_new_with_value (T_INT(self)->value op (int) d)); \
  }

static TNumber *
t_int_add (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (+)
}

static TNumber *
t_int_sub (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (-)
}

static TNumber *
t_int_mul (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (*)
}

static TNumber *
t_int_div (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  int i;
  double d;

  if (T_IS_INT (other)) {
    g_object_get (T_INT (other), "value", &i, NULL);
    if (i == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_int_new_with_value (T_INT(self)->value / i));
  } else {
    g_object_get (T_DOUBLE (other), "value", &d, NULL);
    if (d == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_int_new_with_value (T_INT(self)->value / (int)  d));
  }
}

static TNumber *
t_int_uminus (TNumber *self) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  return T_NUMBER (t_int_new_with_value (- T_INT(self)->value));
}

static char *
t_int_to_s (TNumber *self) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  int i;

  g_object_get (T_INT (self), "value", &i, NULL);
  return g_strdup_printf ("%d", i);
}

```

```
114 static void
115 t_int_class_init (TIntClass *class) {
116   TNumberClass *tnumber_class = T_NUMBER_CLASS (class);
117   GObjectClass *gobject_class = G_OBJECT_CLASS (class);
118 
119   /* override virtual functions */
120   tnumber_class->add = t_int_add;
121   tnumber_class->sub = t_int_sub;
122   tnumber_class->mul = t_int_mul;
123   tnumber_class->div = t_int_div;
124   tnumber_class->uminus = t_int_uminus;
125   tnumber_class->to_s = t_int_to_s;
126 
127   gobject_class->set_property = t_int_set_property;
128   gobject_class->get_property = t_int_get_property;
129   int_property = g_param_spec_int ("value", "val", "Integer␣value", G_MININT,
        G_MAXINT, 0, G_PARAM_READWRITE);
130   g_object_class_install_property (gobject_class, PROP_INT, int_property);
131 }
132 
133 TInt *
134 t_int_new_with_value (int value) {
135   TInt *i;
136 
137   i = g_object_new (T_TYPE_INT, "value", value, NULL);
138   return i;
139 }
140 
141 TInt *
142 t_int_new (void) {
143   TInt *i;
144 
145   i = g_object_new (T_TYPE_INT, NULL);
146   return i;
147 }
```

- 5-6, 15-33, 127-130: Definition of the property “value”. This is the same as before.
- 8-11: Definition of the structure of TInt. This must be defined before `G_DEFINE_TYPE`.
- 13: `G_DEFINE_TYPE` macro. This macro expands to:
  - Declaration of `t_int_init ()` function.
  - Definition of `t_int_get_type ()` function.
  - Definition of `t_int_parent_class` static variable which points the parent class.
- 35-37: `t_int_init`.
- 41-112: These functions are connected to the class method pointers in TIntClass. They are the implementation of the virtual functions defined in `tnumber.c`.
- 41-50: Defines a macro used in `t_int_add`, `t_int_sub` and `t_int_mul`. This macro is similar to `t_int_div` function. Refer to the explanation below for `t_int_div`.
- 52-71: The functions `t_int_add`, `t_int_sub` and `t_int_mul`. The macro `t_int_binary_op` is used.
- 73-95: The function `t_int_div`. The first argument `self` is the object on which the function is called. The second argument `other` is another TNumber object. It can be TInt or TDouble. If it is TDouble, its value is casted to int before the division is performed. If the divisor (`other`) is zero, “div-by-zero” signal is emitted. The signal is defined in TNumber, so TInt doesn’t know the signal id. Therefore, the emission is done with `g_signal_emit_by_name` instead of `g_signal_emit`. The return value of `t_int_div` is TNumber type object However, because TNumber is abstract, the actual type of the object is TInt.
- 97-102: A function for unary minus operator.
- 104-112: The function `to_s`. This function converts int to string. For example, if the value of the object is 123, then the result is a string “123”. The caller should free the string if it becomes useless.
- 114- 131: The class initialization function `t_int_class_init`.
- 120-125: The class methods are overridden. For example, if `t_number_add` is called on a TInt object, then the function calls the class method `*tnumber_class->add`. The pointer points `t_int_add` function. Therefore, `t_int_add` is finally called.
- 133-147: Instance creation functions are the same as before.

6.5 TDouble object.

TDouble object is defined with `tdouble.h` and `tdouble.c`. The definition is very similar to TInt. So, this subsection just shows the contents of the files.

tdouble.h

```c
#pragma once

#include <glib-object.h>

#define T_TYPE_DOUBLE  (t_double_get_type ())
G_DECLARE_FINAL_TYPE (TDouble, t_double, T, DOUBLE, TNumber)

/* create a new TDouble instance */
TDouble *
t_double_new_with_value (double value);

TDouble *
t_double_new (void);
```

tdouble.c

```c
#include "tnumber.h"
#include "tdouble.h"
#include "tint.h"

#define PROP_DOUBLE 1
static GParamSpec *double_property = NULL;

struct _TDouble {
  TNumber parent;
  double value;
};

G_DEFINE_TYPE (TDouble, t_double, T_TYPE_NUMBER)

static void
t_double_set_property (GObject *object, guint property_id, const GValue *value,
    GParamSpec *pspec) {
  TDouble *self = T_DOUBLE (object);
  if (property_id == PROP_DOUBLE) {
    self->value = g_value_get_double (value);
  } else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_double_get_property (GObject *object, guint property_id, GValue *value, GParamSpec
    *pspec) {
  TDouble *self = T_DOUBLE (object);

  if (property_id == PROP_DOUBLE)
    g_value_set_double (value, self->value);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_double_init (TDouble *self) {
}

/* arithmetic operator */
/* These operators create a new instance and return a pointer to it. */
#define t_double_binary_op(op) \
  int i; \
  double d; \
  if (T_IS_INT (other)) { \
    g_object_get (T_INT (other), "value", &i, NULL); \
    return  T_NUMBER (t_double_new_with_value (T_DOUBLE(self)->value op (double)
        i)); \
  } else { \
    g_object_get (T_DOUBLE (other), "value", &d, NULL); \
    return  T_NUMBER (t_double_new_with_value (T_DOUBLE(self)->value op d)); \
  }

static TNumber *
t_double_add (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  t_double_binary_op (+)
}

static TNumber *
t_double_sub (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  t_double_binary_op (-)
}

static TNumber *
t_double_mul (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  t_double_binary_op (*)
}

static TNumber *
t_double_div (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  int i;
  double d;

  if (T_IS_INT (other)) {
    g_object_get (T_INT (other), "value", &i, NULL);
    if (i == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_double_new_with_value (T_DOUBLE(self)->value / (double)
          i));
  } else {
    g_object_get (T_DOUBLE (other), "value", &d, NULL);
    if (d == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_double_new_with_value (T_DOUBLE(self)->value / d));
  }
}

static TNumber *
t_double_uminus (TNumber *self) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  return T_NUMBER (t_double_new_with_value (- T_DOUBLE(self)->value));
}

static char *
```

```c
t_double_to_s (TNumber *self) {
  g_return_val_if_fail (T_IS_DOUBLE (self), NULL);

  double d;

  g_object_get (T_DOUBLE (self), "value", &d, NULL);
  return g_strdup_printf ("%lf", d);
}

static void
t_double_class_init (TDoubleClass *class) {
  TNumberClass *tnumber_class = T_NUMBER_CLASS (class);
  GObjectClass *gobject_class = G_OBJECT_CLASS (class);

  /* override virtual functions */
  tnumber_class->add = t_double_add;
  tnumber_class->sub = t_double_sub;
  tnumber_class->mul = t_double_mul;
  tnumber_class->div = t_double_div;
  tnumber_class->uminus = t_double_uminus;
  tnumber_class->to_s = t_double_to_s;

  gobject_class->set_property = t_double_set_property;
  gobject_class->get_property = t_double_get_property;
  double_property = g_param_spec_double ("value", "val", "Double␣value",
      -G_MAXDOUBLE, G_MAXDOUBLE, 0, G_PARAM_READWRITE);
  g_object_class_install_property (gobject_class, PROP_DOUBLE, double_property);
}

TDouble *
t_double_new_with_value (double value) {
  TDouble *d;

  d = g_object_new (T_TYPE_DOUBLE, "value", value, NULL);
  return d;
}

TDouble *
t_double_new (void) {
  TDouble *d;

  d = g_object_new (T_TYPE_DOUBLE, NULL);
  return d;
}
```

6.6 main.c

main.c is a simple program to test the objects.

```c
#include <glib-object.h>
#include "tnumber.h"
#include "tint.h"
#include "tdouble.h"

static void
notify_cb (GObject *gobject, GParamSpec *pspec, gpointer user_data) {
  const char *name;
  int i;
  double d;

  name = g_param_spec_get_name (pspec);
  if (T_IS_INT (gobject) && strcmp (name, "value") == 0) {
    g_object_get (T_INT (gobject), "value", &i, NULL);
    g_print ("Property␣\"%s\"␣is␣set␣to␣%d.\n", name, i);
16    } else if (T_IS_DOUBLE (gobject) && strcmp (name, "value") == 0) {
17      g_object_get (T_DOUBLE (gobject), "value", &d, NULL);
18      g_print ("Property␣\"%s\"␣is␣set␣to␣%lf.\n", name, d);
19    }
20  }
21
22  int
23  main (int argc, char **argv) {
24    TNumber *i, *d, *num;
25    char *si, *sd, *snum;
26
27    i = T_NUMBER (t_int_new ());
28    d = T_NUMBER (t_double_new ());
29
30    g_signal_connect (G_OBJECT (i), "notify::value", G_CALLBACK (notify_cb), NULL);
31    g_signal_connect (G_OBJECT (d), "notify::value", G_CALLBACK (notify_cb), NULL);
32
33    g_object_set (T_INT (i), "value", 100, NULL);
34    g_object_set (T_DOUBLE (d), "value", 12.345, NULL);
35
36    num = t_number_add (i, d);
37
38    si = t_number_to_s (i);
39    sd = t_number_to_s (d);
40    snum = t_number_to_s (num);
41
42    g_print ("%s␣+␣%s␣is␣%s.\n", si, sd, snum);
43
44    g_object_unref (num);
45    g_free (snum);
46
47    num = t_number_add (d, i);
48    snum = t_number_to_s (num);
49
50    g_print ("%s␣+␣%s␣is␣%s.\n", sd, si, snum);
51
52    g_object_unref (num);
53    g_free (sd);
54    g_free (snum);
55
56    g_object_set (T_DOUBLE (d), "value", 0.0, NULL);
57    sd = t_number_to_s (d);
58    if ((num = t_number_div(i, d)) != NULL) {
59      snum = t_number_to_s (num);
60      g_print ("%s␣/␣%s␣is␣%s.\n", si, sd, snum);
61      g_object_unref (num);
62      g_free (snum);
63    }
64
65    g_object_unref (i);
66    g_object_unref (d);
67    g_free (si);
68    g_free (sd);
69
70    return 0;
71  }
```

- 6-20: "notify" handler. This handler is upgraded to support both TInt and TDouble.
- 22-71: The function `main`.
- 30-31: Connects the notify signals on `i` (TInt) and `d` (TDouble).
- 33-34: Set "value" properties on `i` and `d`.
- 36: Add `d` to `i`. The answer is TInt object.
- 47: Add `i` to `d`. The answer is TDouble object. The addition of two TNumber objects isn't

commutative because the type of the result will be different if the two objects are exchanged.
- 56-63: Tests division by zero signal.

6.7 Compilation and execution

The source files are located under src/tnumber. The file `meson.buld`, which controls the compilation process, is as follows.

```
project('tnumber', 'c')

gobjdep = dependency('gobject-2.0')

sourcefiles = files('main.c', 'tdouble.c' 'tint.c', 'tnumber.c',)

executable('tnumber', sourcefiles, dependencies: gobjdep, install: false)
```

Compilation and execution is done by the usual way.

```
$ cd src/tnumber
$ meson setup _build
$ ninja -C _build
$ _build/tnumber
```

Then the following is shown on the display.

```
Property "value" is set to 100.
Property "value" is set to 12.345000.
100 + 12.345000 is 112.
12.345000 + 100 is 112.345000.
Property "value" is set to 0.000000.

Error: division by zero.
```

The two answers are different because of the different types.

This section has shown a simple example of derivable and abstract class. You can define your derivable object like this. If your object isn't abstract, use `G_DEFINE_TYPE` instead of `G_DEFINE_ABSTRACT_TYPE`. And you need one more thing, how to manage private data in your derivable object. There is a tutorial in GObject API Reference. See the tutorial for learning derivable object.

It is also good to see source files in GTK.

6.8 Class initialization process

6.8.1 Initialization process of TNumberClass

Because TNumber is an abstract object, you cannot instantiate it directly. And you cannot create the TNumber class as well. But when you create its descendant instance, TNumber class is made and initialized. First call for `g_object_new (T_TYPE_INT, ...)` or `g_object_new (T_TYPE_DOUBLE, ...)` creates and initializes TNumberClass if the class doesn't exist. After that, TIntClass or TDoubleClass is created and followed by the creation for TInt or TDouble instance respectively.

And the initialization process for the TNumber class is as follows.

1. GObjectClass has been initialized before the function `main` starts.
2. Memory is allocated for TNumberClass.
3. The parent (GObjectClass) part of the class is copied from GObjectClass.
4. The class initialization function `t_number_class_init` is called. It initializes class methods (Pointers to the class methods) and a default signal handler.

The diagram below shows the process.

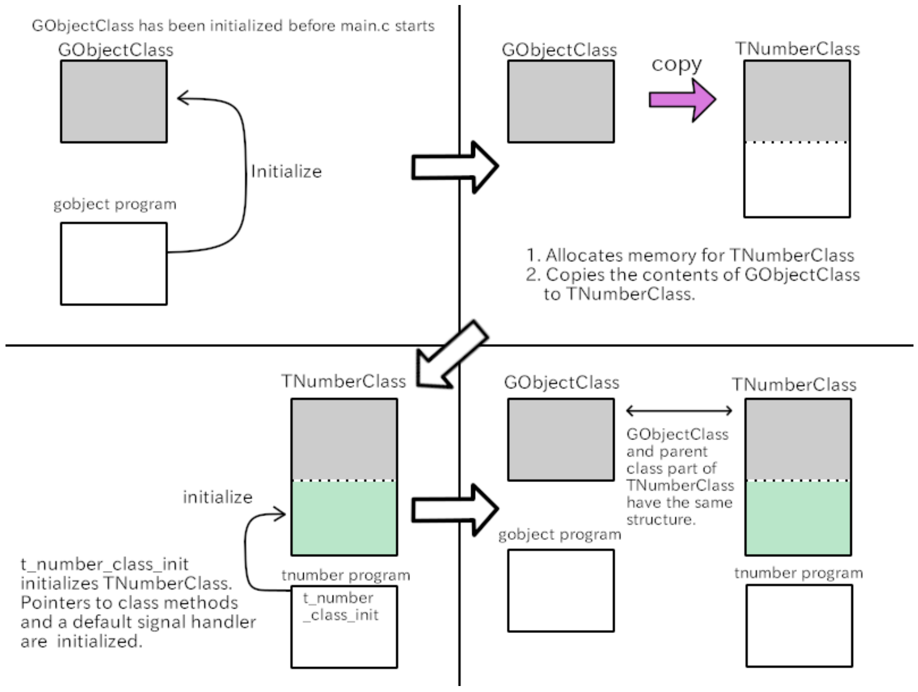
GObjectClass has been initialized before main.c starts
GObjectClass
Initialize
gobject program
GObjectClass
copy
TNumberClass
1. Allocates memory for TNumberClass
2. Copies the contents of GObjectClass to TNumberClass.
TNumberClass
initialize
t_number_class_init initializes TNumberClass. Pointers to class methods and a default signal handler are initialized.
tnumber program
t_number _class_init
GObjectClass
TNumberClass
GObjectClass and parent class part of TNumberClass have the same structure.
gobject program
tnumber program

Figure 3: TNumberClass initialization

6.8.2 Initialization process of TIntClass

1. TNumberClass has been initialized before the initialization of TIntClass starts.
2. First call for `g_object_new (T_TYPE_INT, ...)` initializes TIntClass. And the initialization process is as follows.
3. Memory is allocated for TIntClass. TIntClass doesn't have its own area. Therefore its structure is the same as its parent class (TNumberClass).
4. The parent (TNumberClass) part of the class (This is the same as whole TIntClass) is copied from TNumberClass.
5. The class initialization function `t_int_class_init` is called. It overrides class methods from TNumber, `set_property` and `get_property`.

The diagram below shows the process.

7 Derivable and non-abstract type

It is more common to make a non-abstract derivable type than abstract type. This section covers how to make non-abstract derivable type objects. A derivable type example is an object for string. It is TStr. And its child is an object for numeric string. A numeric string is a string that expresses a number. For example, "0", "-100" and "123.45". The child object (numeric string) will be explained in the next section.

This section describes memory management for strings before derivable objects.

7.1 String and memory management

TStr has a string type value. It is similar to TInt or TDouble but string is more complex than int and double. When you make TStr program, you need to be careful about memory management, which is not necessary to TInt and TDouble.

7.1.1 String and memory

String is an array of characters that is terminated with '\0'. String is not a C type such as char, int, float or double. But the pointer to a character array behaves like a string type of other languages. So, we often call the pointer string.

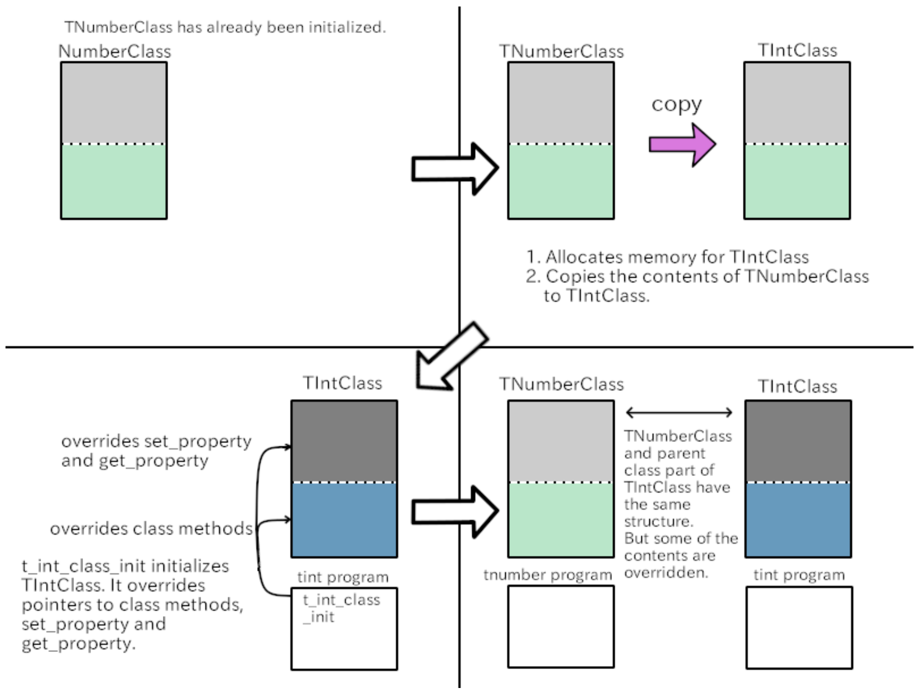
TNumberClass has already been initialized.
NumberClass
TNumberClass
copy
TIntClass
1. Allocates memory for TIntClass
2. Copies the contents of TNumberClass to TIntClass.
TIntClass
overrides set_property and get_property
overrides class methods
t_int_class_init initializes TIntClass. It overrides pointers to class methods, set_property and get_property.
tint program
t_int_class _init
TNumberClass
tnumber program
TNumberClass and parent class part of TIntClass have the same structure. But some of the contents are overridden.
TIntClass
tint program

Figure 4: TIntClass initialization

If the pointer is NULL, it points nothing. So, the pointer is not a string. Programs with string will include bugs if you aren't careful about NULL pointer.

Another annoying problem is memory allocation. Because string is an array of characters, memory allocation is necessary to create a new string. We don't forget to allocate memory, but often forget to free the memory. It causes memory leak.

```
char *s;
s = g_strdup ("Hello.");
... ... ... do something with s
g_free (s);
```

`g_strdup` duplicates a string. It does:

- Calculates the size of the string. The size of "Hello." is 7 because strings are zero-terminated. The string is an array 'H', 'e', 'l', 'l', 'o', '.' and zero ('\0').
- Requests the system to allocate 7 bytes memories.
- Copies the string to the memory.
- Returns the pointer to the newly-allocated memory.
- If the argument is NULL, then no memory is allocated and it returns NULL.

If the string `s` is no longer in use, `s` must be freed, which means the allocated 7 bytes must be returned to the system. `g_free` frees the memory.

Strings bounded by double quotes like "Hello." are string literals. They are an array of characters, but the contents of the array are not allowed to change. And they mustn't be freed. If you write a character in a string literal or free a string literal, the result is undefined. Maybe bad things will happen, for example, a segmentation fault error.

There's a difference between arrays and pointers when you initialize them with a string literal. If an array is initialized with a string literal, the array can be changed.

```
char a[]="Hello!";
a[1]='a';
g_print ("%s\n", a); /* Hallo will appear on your display. */
```

The first line initializes an array `a`. The initialization is not simple. First, the compiler calculates the length of "Hello!". It is seven because the string literal has '\0' at the end of it. Then seven bytes memory is allocated in static memory or stack memory. It depends on the class of the array, whether `static` or `auto`. The memory is initialized with “Hello!”. So, the string in the array can be changed. This program successfully displays ‘Hallo!.

The first line of the program above is the same as follows.

```
char a[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

If you define a pointer with string literal, you can’t change the string pointed by the pointer.

```
char *a = "Hello";
*(a+1) = 'a'; /* This is illegal. */
g_print ("%s\n", a);
```

The first line just assigns the address of the string literal to the variable `a`. String literal is an array of characters but it’s read-only. It might be in the program code area or some other non-writable area. It depends on the implementation of your compiler. Therefore, the second line tries to write a char ‘a’ to the read-only memory and the result is undefined, for example, a segmentation error happens. Anyway, don’t write a program like this.

In conclusion, a string is an array of characters and it is placed in one of the following.

- Read-only memory. A string literal is read-only.
- Stack. If the class of an array is `auto`, then the array is placed in the stack. stack is writable. If the array is defined in a function, its default class is `auto`. The stack area will disappear when the function returns to the caller.
- Static area. If the class of an array is `static`, then the array is placed in the static area. It keeps its value and remains for the life of the program. This area is writable.
- Heap. You can allocate and free memory for a string. For allocation, `g_new` or `g_new0` is used. For freeing, `g_free` is used.

7.1.2 Copying string

There are two ways to copy a string. First way is just copying the pointer.

```
char *s = "Hello";
char *t = s;
```

Two pointers `s` and `t` points the same address. Therefore, you can’t modify `t` because `t` points a string literal, which is read-only.

Second way is creating memory and copying the string to the memory.

```
char *s = "Hello";
char *t = g_strdup (s);
```

The function `g_strdup` allocates memory and initializes it with “Hello”, then returns the pointer to the memory. The function `g_strdup` is almost same as the function `string_dup` below.

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

char *
string_dup (char *s) {
  int length;
  char *t;

  if (s == NULL)
    return NULL;
  length = strlen (s) + 1;
  t = g_new (char, length);
  strcpy (t, s);
  return t;
}
```

If `g_strdup` is used, the two pointers `s` and `t` point different memories. You can modify `t` because it is placed in the memory allocated from the heap area.

It is important to know the difference between assigning pointers and duplicating strings.

7.1.3 const qualifier

The qualifier `const` makes a variable won't change its value. It can also be applied to an array. Then, the elements of the array won't be changed.

```
const double pi = 3.1415926536;
const char a[] = "read␣only␣string";
```

An array parameter in a function can be qualified with `const` to indicate that the function does not change the array. In the same way, the return value (a pointer to an array or string) of a function can be qualified with `const`. The caller mustn't modify or free the returned array or string.

```
char *
string_dup (const char *s) {
  ... ...
}

const char *
g_value_get_string (const GValue *value);
```

The qualifier `const` indicates who is the owner of the string when it is used in the function of objects. "Owner" is an object or a caller of the function that has the right to modify or free the string.

For example, `g_value_get_string` is given `const GValue *value` as an argument. The GValue pointed by `value` is owned by the caller and the function doesn't change or free it. The function returns a string qualified with `const`. The returned string is owned by the object and the caller mustn't change or free the string. It is possible that the string will be changed or freed by the object later.

7.2 Header file

The rest of this section is about TStr. TStr is a child of GObject and it holds a string. The string is a pointer and an array of characters. The pointer points the array. The pointer can be NULL. If it is NULL, TStr has no array. The memory of the array comes from the heap area. TStr owns the memory and is responsible to free it when it becomes useless. TStr has a string type property.

The header file `tstr.h` is as follows.

```
#pragma once

#include <glib-object.h>

#define T_TYPE_STR  (t_str_get_type ())
G_DECLARE_DERIVABLE_TYPE (TStr, t_str, T, STR, GObject)

struct _TStrClass {
  GObjectClass parent_class;
  /* expect that descendants will override the setter */
  void (*set_string)  (TStr *self, const char *s);
};

TStr *
t_str_concat (TStr *self, TStr *other);

/* setter and getter */
void
t_str_set_string (TStr *self, const char *s);

char *
t_str_get_string (TStr *self);

```

```
24 /* create a new TStr instance */
25 TStr *
26 t_str_new_with_string (const char *s);
27
28 TStr *
29 t_str_new (void);
```

- 6: Uses `G_DECLARE_DERIVABLE_TYPE`. The TStr class is derivable and its child class will be defined later.
- 8-12: TStrClass has one class method. It is `set_string` member of the TStrClass structure. This will be overridden by the child class `TNumStr`. Therefore, Both TStr and TNumStr has `set_string` member in their classes but they point different functions.
- 14-15: The public function `t_str_concat` connects two strings of TStr instances and returns a new TStr instance.
- 18-22: Setter and getter.
- 25-29: Functions to create a TStr object.

7.3 C file

The C file `tstr.c` for TStr is as follows.

```
#include "tstr.h"

enum {
  PROP_0,
  PROP_STRING,
  N_PROPERTIES
};

static GParamSpec *str_properties[N_PROPERTIES] = {NULL, };

typedef struct {
  char *string;
} TStrPrivate;

G_DEFINE_TYPE_WITH_PRIVATE(TStr, t_str, G_TYPE_OBJECT)

static void
t_str_set_property (GObject *object, guint property_id, const GValue *value, GParamSpec *pspec) {
  TStr *self = T_STR (object);

/* The returned value of the function g_value_get_string can be NULL. */
/* The function t_str_set_string calls a class method, */
/* which is expected to rewrite in the descendant object. */
  if (property_id == PROP_STRING)
    t_str_set_string (self, g_value_get_string (value));
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_str_get_property (GObject *object, guint property_id, GValue *value, GParamSpec *pspec) {
  TStr *self = T_STR (object);
  TStrPrivate *priv = t_str_get_instance_private (self);

/* The second argument of the function g_value_set_string can be NULL. */
  if (property_id == PROP_STRING)
    g_value_set_string (value, priv->string);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

/* This function just set the string. */
/* So, no notify signal is emitted. */
static void
t_str_real_set_string (TStr *self, const char *s) {
  TStrPrivate *priv = t_str_get_instance_private (self);

  if (priv->string)
    g_free (priv->string);
  priv->string = g_strdup (s);
}

static void
t_str_finalize (GObject *object) {
  TStr *self = T_STR (object);
  TStrPrivate *priv = t_str_get_instance_private (self);

  if (priv->string)
    g_free (priv->string);
  G_OBJECT_CLASS (t_str_parent_class)->finalize (object);
}

static void
t_str_init (TStr *self) {
  TStrPrivate *priv = t_str_get_instance_private (self);

  priv->string = NULL;
}

static void
t_str_class_init (TStrClass *class) {
  GObjectClass *gobject_class = G_OBJECT_CLASS (class);

  gobject_class->finalize = t_str_finalize;
  gobject_class->set_property = t_str_set_property;
  gobject_class->get_property = t_str_get_property;
  str_properties[PROP_STRING] = g_param_spec_string ("string", "str", "string", "",
      G_PARAM_READWRITE);
  g_object_class_install_properties (gobject_class, N_PROPERTIES, str_properties);

  class->set_string = t_str_real_set_string;
}

/* setter and getter */
void
t_str_set_string (TStr *self, const char *s) {
  g_return_if_fail (T_IS_STR (self));
  TStrClass *class = T_STR_GET_CLASS (self);

/* The setter calls the class method 'set_string', */
/* which is expected to be overridden by the descendant TNumStr. */
/* Therefore, the behavior of the setter is different between TStr and TNumStr. */
  class->set_string (self, s);
}

char *
t_str_get_string (TStr *self) {
  g_return_val_if_fail (T_IS_STR (self), NULL);
  TStrPrivate *priv = t_str_get_instance_private (self);

  return g_strdup (priv->string);
}

TStr *
t_str_concat (TStr *self, TStr *other) {
  g_return_val_if_fail (T_IS_STR (self), NULL);
106   g_return_val_if_fail (T_IS_STR (other), NULL);
107
108   char *s1, *s2, *s3;
109   TStr *str;
110
111   s1 = t_str_get_string (self);
112   s2 = t_str_get_string (other);
113   if (s1 && s2)
114     s3 = g_strconcat (s1, s2, NULL);
115   else if (s1)
116     s3 = g_strdup (s1);
117   else if (s2)
118     s3 = g_strdup (s2);
119   else
120     s3 = NULL;
121   str = t_str_new_with_string (s3);
122   if (s1) g_free (s1);
123   if (s2) g_free (s2);
124   if (s3) g_free (s3);
125   return str;
126 }
127
128 /* create a new TStr instance */
129 TStr *
130 t_str_new_with_string (const char *s) {
131   return T_STR (g_object_new (T_TYPE_STR, "string", s, NULL));
132 }
133
134 TStr *
135 t_str_new (void) {
136   return T_STR (g_object_new (T_TYPE_STR, NULL));
137 }
```

- 3-9: `enum` defines a property id. The member `PROP_STRING` is the id of the "string" property. Only one property is defined here, so it is possible to define it without `enum`. However, `enum` can be applied to define two or more properties and it is more common. The last member `N_PROPERTIES` is two because `enum` is zero-based. An array `str_properties` has two elements since `N_PROPERTIES` is two. The first element isn't used and it is assigned with NULL. The second element will be assigned a pointer to a GParamSpec instance in the class initialization function.
- 11-13: TStrPrivate is a C structure. It is a private data area for TStr. If TStr were a final type, then no descendant would exist and TStr instance could be a private data area. But TStr is derivable so you can't store such private data in TStr instance that is open to the descendants. The name of this structure is "<object name>Private" like `TStrPrivate`. The structure must be defined before `G_DEFINE_TYPE_WITH_PRIVATE`.
- 15: `G_DEFINE_TYPE_WITH_PRIVATE` macro. It is similar to `G_DEFINE_TYPE` macro but it adds the private data area for the derivable instance. This macro expands to:
  - Declaration of `t_str_class_init` which is a class initialization function.
  - Declaration of `t_str_init` which is an instance initialization function.
  - Definition of `t_str_parent_class` static variable. It points to the parent class of TStr.
  - The function call that adds private instance data to the type. It is a C structure and its name is `TStrPrivate`.
  - Definition of `t_str_get_type ()` function. This function registers the type in its first call.
  - Definition of the private instance getter `t_str_get_instance_private ()`.
- 17-28: The function `t_str_set_property` sets the "string" property and it is used by `g_object_set` family functions. It uses `t_str_set_string` function to set the private data with the copy of the string from the GValue. It is important because `t_str_set_string` calls the class method `set_string`, which will be overridden by the child class. Therefore, the behavior of `t_str_set_property` function is different between TStr and TNumStr, which is a child of TStr. The function `g_value_get_string` returns the pointer to the string that GValue owns. So you need to duplicate the string and it is done in the function `t_str_set_string`.
- 30-40: The function `t_str_get_property` gets the "string" property and it is used by `g_object_get`

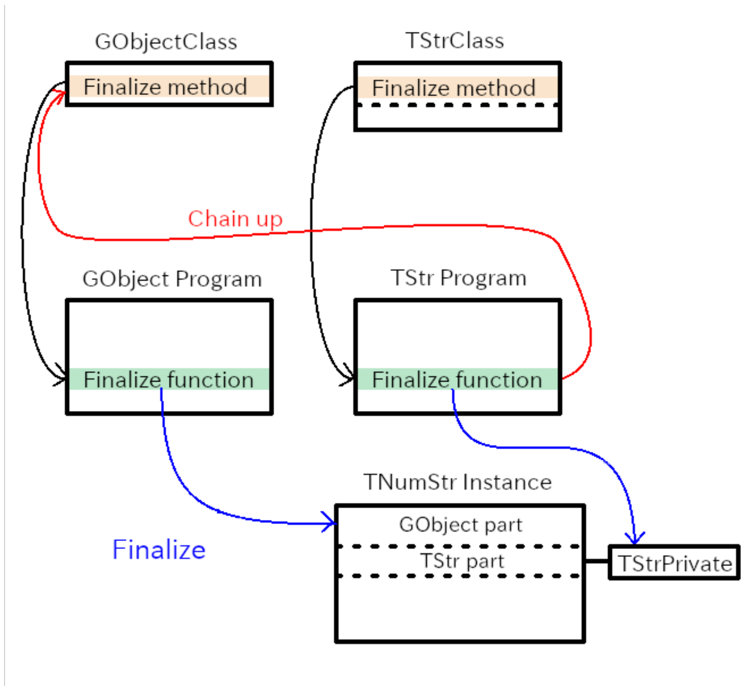

Figure 5: Chaining up process in GObject and TStr

family functions. It just gives `priv->string` to the function `g_value_set_string`. The variable `priv` points the private data area. The second argument `priv->string` is owned by the TStr instance and the function `g_value_set_string` duplicates it to store in the GValue structure.

- 44-51 The function `t_str_real_set_string` is the body of the class method and pointed by `set_string` member in the class. First, it gets the pointer to the private area with `t_str_get_instance_private` function. If the instance holds a string, free it before setting it to a new string. It copies the given string and assigns it to `priv->string`. The duplication is important. Thanks to that, the address of the string is hidden from the out of the instance.
- 53-61: The finalize function `t_str_finalize` is called when TStr instance is destroyed. The destruction process has two phases, "dispose" and "finalize". In the disposal phase, the instance releases instances. In the finalization phase, the instance does the rest of all the finalization like freeing memories. This function frees the string `priv->string` if necessary. After that, it calls the parent's finalize method. This is called "chain up to its parent" and it will be explained later in this section.
- 63-68: The instance initialization function `t_str_init` assigns NULL to `priv->string`.
- 70-81: The class initialization function `t_str_class_init` overrides `finalize`, `set_property` and `get_property` members. It creates the GParamSpec instance with `g_param_spec_string` and installs the property with `g_object_class_install_properties`. It assigns `t_str_real_set_string` to the member `set_string`. It is a class method and is expected to be replaced in the child class.
- 84-101: Setter and getter. The setter method `t_str_set_string` just calls the class method. So, it is expected to be replaced in the child class. It is used by property set function `t_str_set_property`. So the behavior of property setting will change in the child class. The getter method `t_str_get_string` just duplicates the string `priv->string` and return the copy.
- 103-126: The public function `t_str_concat` concatenates the string of `self` and `other`, and creates a new TStr.
- 129-137: Two functions `t_str_new_with_string` and `t_str_new` create a new TStr instances.

7.4 Chaining up to its parent

The "chain up to its parent" process is illustrated with the diagram below.

There are two classes, GObjectClass and TStrClass. Each class has their finalize methods (functions) pointed by the pointers in the class structures. The finalize method of TStrClass finalizes its own part of the TStr instance. At the end of the function, it calls its parent's finalize method. It is the finalize method of GObjectClass. It calls its own finalize function and finalizes the GObject private data.

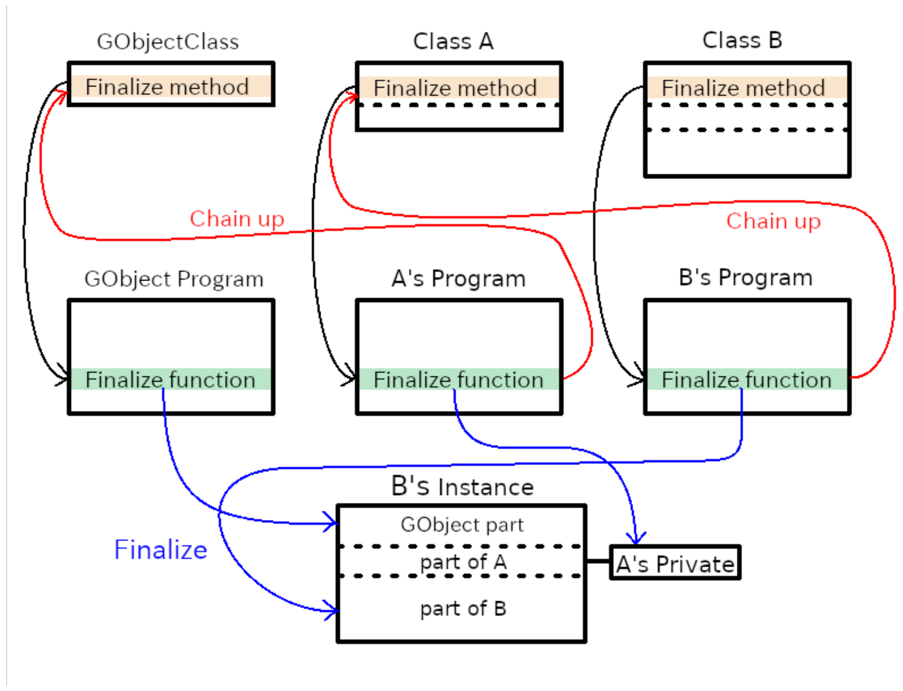

Figure 6: Chaining up process

If the GObjectClass has two or more descendant classes, the number of the finalize functions may be the same as the number of the descendants. And they are connected by "chain up to its parent" way.

7.5 How to write a derivable type

- Use `G_DECLARE_DERIVABLE_TYPE` macro in the header file. You need to write a macro like `#define T_TYPE_STR (t_str_get_type ())` before `G_DECLARE_DERIVABLE_TYPE`.
- Declare your class structure in the header file. The contents of the class are open to the descendants. Most of the members are class methods.
- Use `G_DEFINE_TYPE_WITH_PRIVATE` in the C file. You need to define the private area before `G_DEFINE_TYPE_WITH_PRIVATE`. It is a C structure and the name is "<object name>Private" like "TStrPrivate".
- Define class and instance initialization functions.

8 Child class extends parent's function

The example in this section is TNumStr object. TNumStr is a child of TStr object. TNumStr holds a string and the string type, which is one of `t_int`, `t_double` or `t_none`.

- t_int: the string expresses an integer
- t_double: the string expresses a double (floating point)
- t_none: the string doesn't express the two numbers above.

A t_int or t_double type string is called a numeric string. It is not a common terminology and it is used only in this tutorial. In short, a numeric string is a string that expresses a number. For example, "0", "-100" and "123.456" are numeric strings. They are strings and express numbers.

- "0" is a string. It is a character array and its elements are '0' and '\0'. It expresses 0, which is an integer zero.
- "-100" is a string that consists of '-', '1', '0', '0' and '\0'. It expresses an integer -100.
- "123.456" consists of '1', '2', '3', '.', '4', '5', '6' and '\0'. It expresses a real number (double type) 123.456.

A numeric string is such a specific string.

8.1 Verification of a numeric string

Before defining TNumStr, we need a way to verify a numeric string.

Numeric string includes:

- Integer. For example, “0”, “100”, “-10” and “+20”.
- Double. For example, “0.1”, “-10.3”, “+3.14”, “.05” “1.” and “0.0”.

We need to be careful that “0” and “0.0” are different. Because their type are different. The type of “0” is integer and the type of “0.0” is double. In the same way, “1” is an integer and “1.” is a double.

“.5” and “0.5” are the same. Both are double and their values are 0.5.

Verification of a numeric string is a kind of lexical analysis. A state diagram and state matrix is often used for lexical analysis.

A numeric string is a sequence of characters that satisfies:

1. ‘+’ or ‘-’ can be the first character. It can be left out.
2. followed by a sequence of digits.
3. followed by a period.
4. followed by a sequence of digits.

The second part can be left out. For example, “.56” or “-.56” are correct.

The third and fourth parts can be left out. For example, “12” or “-23” are correct.

The fourth part can be left out. For example, “100.” is correct.

There are six states.

- 0 is the start point.
- 1 is the state after ‘+’ or ‘-’.
- 2 is at the middle of the first sequence of digits (integer part).
- 3 is the state after the decimal point.
- 4 is the end of the string and the string is int.
- 5 is the end of the string and the string is double.
- 6 is error. The string doesn’t express a number.

The input characters are:

- 0: ‘+’ or ‘-’
- 1: digit (‘0’ - ‘9’)
- 2: period ‘.’
- 3: end of string ‘\0’
- 4: other characters

The state diagram is as follows.

The state matrix is:

state input	0	1	2	3	4
0	1	2	3	6	6
1	6	2	3	6	6
2	6	2	3	4	6
3	6	3	6	5	6

This state diagram doesn’t support “1.23e5” style double (decimal floating point). If it is supported, the state diagram will be more complicated. (However, it will be a good practice for your programming skill.)

8.2 Header file

The header file of TNumStr is `tnumstr.h`. It is in the `src/tstr` directory.

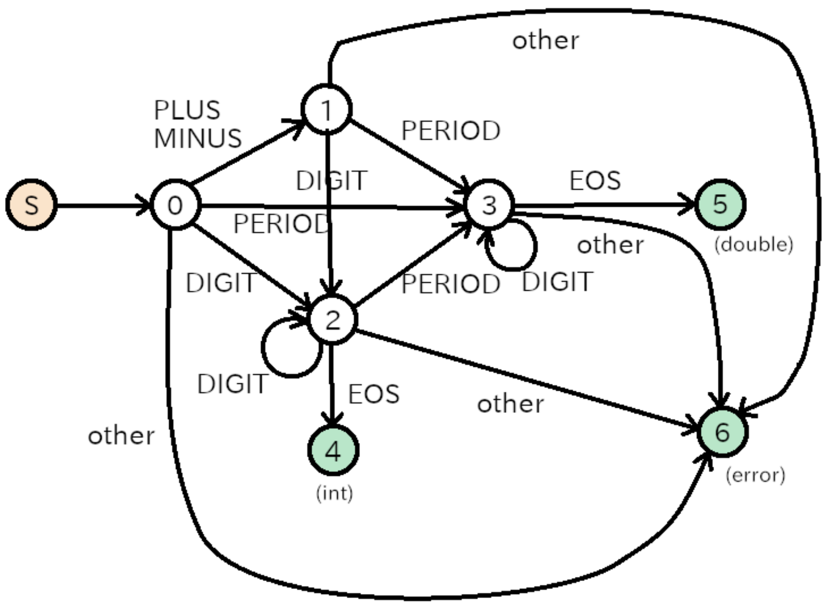

Figure 7: state diagram of a numeric string

```c
#pragma once

#include <glib-object.h>
#include "tstr.h"
#include "../tnumber/tnumber.h"

#define T_TYPE_NUM_STR  (t_num_str_get_type ())
G_DECLARE_FINAL_TYPE (TNumStr, t_num_str, T, NUM_STR, TStr)

/* type of the numeric string */
enum {
  t_none,
  t_int,
  t_double
};

/* get the type of the TNumStr object */
int
t_num_str_get_string_type (TNumStr *self);

/* setter and getter */
void
t_num_str_set_from_t_number (TNumStr *self, TNumber *num);

// TNumStr can have any string, which is t_none, t_int or t_double type.
// If the type is t_none, t_num_str_get_t_number returns NULL.
// It is good idea to call t_num_str_get_string_type and check the type in advance.

TNumber *
t_num_str_get_t_number (TNumStr *self);

/* create a new TNumStr instance */
TNumStr *
t_num_str_new_with_tnumber (TNumber *num);

TNumStr *
t_num_str_new (void);
```

- 8: The macro `G_DECLARE_FINAL_TYPE` for TNumStr class. It is a child class of TStr and a final type class.
- 11-15: These three enum data define the type of TNumStr string.
  - `t_none`: No string is stored or the string isn't a numeric string.
  - `t_int`: The string expresses an integer
  - `t_double`: The string expresses an real number, which is double type in C language.
- 18-19: The public function `t_num_str_get_string_type` returns the type of the string TStrNum object has. The returned value is `t_none`, `t_int` or `t_double`.
- 22-30: Setter and getter from/to a TNumber object.
- 33-37: Functions to create new TNumStr objects.

8.3 C file

The C file of TNumStr is `tnumstr.c`. It is in the `src/tstr` directory.

```c
#include <stdlib.h>
#include <ctype.h>
#include "tnumstr.h"
#include "tstr.h"
#include "../tnumber/tnumber.h"
#include "../tnumber/tint.h"
#include "../tnumber/tdouble.h"

struct _TNumStr {
  TStr parent;
  int type;
};

G_DEFINE_TYPE(TNumStr, t_num_str, T_TYPE_STR)

static int
t_num_str_string_type (const char *string) {
  const char *t;
  int stat, input;
  /* state matrix */
  int m[4][5] = {
    {1, 2, 3, 6, 6},
    {6, 2, 3, 6, 6},
    {6, 2, 3, 4, 6},
    {6, 3, 6, 5, 6}
  };

  if (string == NULL)
    return t_none;
  stat = 0;
  for (t = string; ; ++t) {
    if (*t == '+' || *t == '-')
      input = 0;
    else if (isdigit (*t))
      input = 1;
    else if (*t == '.')
      input = 2;
    else if (*t == '\0')
      input = 3;
    else
      input = 4;

    stat = m[stat][input];

    if (stat >= 4 || *t == '\0')
      break;
  }
  if (stat == 4)
    return t_int;
  else if (stat == 5)
    return t_double;
  else
    return t_none;
}

/* This function overrides t_str_set_string. */
/* And it also changes the behavior of setting the "string" property. */
/* On TStr => just set the "string" property */
/* On TNumStr => set the "string" property and set the type of the string. */
static void
t_num_str_real_set_string (TStr *self, const char *s) {
  T_STR_CLASS (t_num_str_parent_class)->set_string (self, s);
  T_NUM_STR (self)->type = t_num_str_string_type(s);
}

static void
t_num_str_init (TNumStr *self) {
  self->type = t_none;
}

static void
t_num_str_class_init (TNumStrClass *class) {
  TStrClass *t_str_class = T_STR_CLASS (class);

  t_str_class->set_string = t_num_str_real_set_string;
}

int
t_num_str_get_string_type (TNumStr *self) {
  g_return_val_if_fail (T_IS_NUM_STR (self), -1);

  return self->type;
}

/* setter and getter */
void
t_num_str_set_from_t_number (TNumStr *self, TNumber *num) {
  g_return_if_fail (T_IS_NUM_STR (self));
  g_return_if_fail (T_IS_NUMBER (num));

  char *s;

  s = t_number_to_s (T_NUMBER (num));
  t_str_set_string (T_STR (self), s);
  g_free (s);
}

TNumber *
t_num_str_get_t_number (TNumStr *self) {
  g_return_val_if_fail (T_IS_NUM_STR (self), NULL);

  char *s = t_str_get_string(T_STR (self));
  TNumber *tnum;

  if (self->type == t_int)
    tnum = T_NUMBER (t_int_new_with_value (atoi (s)));
  else if (self->type == t_double)
    tnum = T_NUMBER (t_double_new_with_value (atof (s)));
  else
    tnum = NULL;
  g_free (s);
  return tnum;
}
```

```
114
115 /* create a new TNumStr instance */
116
117 TNumStr *
118 t_num_str_new_with_tnumber (TNumber *num) {
119   g_return_val_if_fail (T_IS_NUMBER (num), NULL);
120
121   TNumStr *numstr;
122
123   numstr = t_num_str_new ();
124   t_num_str_set_from_t_number (numstr, num);
125   return numstr;
126 }
127
128 TNumStr *
129 t_num_str_new (void) {
130   return T_NUM_STR (g_object_new (T_TYPE_NUM_STR, NULL));
131 }
```

- 9-12: TNumStr structure has its parent "TStr" and int type "type" members. So, TNumStr instance holds a string, which is placed in the parent's private area, and a type.
- 14: `G_DEFINE_TYPE` macro.
- 16- 54: The function `t_num_str_string_type` checks the given string and returns `t_int`, `t_double` or `t_none`. If the string is NULL or an non-numeric string, `t_none` will be returned. The check algorithm is explained in the first subsection "Verification of a numeric string".
- 60-64: The function `t_num_str_real_set_string` sets TNumStr's string and its type. This is a body of the class method pointed by `set_string` member of the class structure. The class method is initialized in the class initialization function `t_num_str_class_init`.
- 66-69: The instance initialization function `t_num_str_init` sets the type to `t_none` because its parent initialization function set the pointer `priv->string` to NULL.
- 71-76: The class initialization function `t_num_str_class_init` assigns `t_num_str_real_set_string` to the member `set_string`. Therefore, the function `t_str_set_string` calls `t_num_str_real_set_string`, which sets not only the string but also the type. The function `g_object_set` also calls it and sets both the string and type.
- 78-83: The public function `t_num_str_get_string_type` returns the type of the string.
- 86-113: Setter and getter. The setter sets the numeric string from a TNumber object. And the getter returns a TNumber object.
- 117-131: These two functions create TNumStr instances.

8.4 Child class extends parent's function.

TNumStr is a child class of TStr, so it has all the TStr's public funftions.

- `TStr *t_str_concat (TStr *self, TStr *other)`
- `void t_str_set_string (TStr *self, const char *s)`
- `char *t_str_get_string (TStr *self)`

When you want to set a string to a TNumStr instance, you can use `t_str_set_string` function.

```
TNumStr *ns = t_num_str_new ();
t_str_set_string (T_STR (ns), "123.456");
```

TNumStr extends the function `t_str_set_string` and it sets not only a string but also the type for a TNumStr instance.

```
int t;
t = t_num_str_get_string_type (ns);
if (t == t_none) g_print ("t_none\n");
if (t == t_int) g_print ("t_int\n");
if (t == t_double) g_print ("t_double\n");
// => t_double appears on your display
```

TNumStr adds some public functions.

- `int t_num_str_get_string_type (TNumStr *self)`
- `void t_num_str_set_from_t_number (TNumStr *self, TNumber *num)`
- `TNumber *t_num_str_get_t_number (TNumStr *self)`

A child class extends the parent class. So, a child is more specific and richer than the parent.

8.5 Compilation and execution

There are `main.c`, `test1.c` and `test2.c` in `src/tstr` directory. Two programs `test1.c` and `test2.c` generates `_build/test1` and `_build/test2` respectively. They test `tstr.c` and `tnumstr.c`. If there are errors, messages will appear. Otherwise nothing appears.

The program `main.c` generates `_build/tnumstr`. It shows how TStr and TNumStr work.

Compilation is done by usual way. First, change your current directory to `src/tstr`.

```
$ cd src/tstr
$ meson setup _build
$ ninja -C _build
$ _build/test1
$ _build/test2
$ _build/tnumstr
String property is set to one.
"one" and "two" is "onetwo".
123 + 456 + 789 = 1368
TNumStr => TNumber => TNumStr
123 => 123 => 123
-45 => -45 => -45
+0 => 0 => 0
123.456 => 123.456000 => 123.456000
+123.456 => 123.456000 => 123.456000
-123.456 => -123.456000 => -123.456000
.456 => 0.456000 => 0.456000
123. => 123.000000 => 123.000000
0.0 => 0.000000 => 0.000000
123.4567890123456789 => 123.456789 => 123.456789
abc => (null) => abc
(null) => (null) => (null)
```

The last part of `main.c` is conversion between TNumStr and TNumber. There are some difference between them because of the following two reasons.

- floating point is rounded. It is not an exact value when the value has long figures. TNumStr "123.4567890123456789" is converted to TNumber 123.456789.
- There are two or more string expression in floating points. TNumStr ".456" is converted to TNumber x, and x is converted to TNumStr "0.456000".

It is difficult to compare two TNumStr instances. The best way is to compare TNumber instances converted from TNumStr.

9 Interface

Interface is similar to abstract class. Interface defines virtual functions which are expected to be overridden by a function in another instantiable object.

This section provides a simple example, TComparable. TComparable is an interface. It defines functions to compare. They are:

- `t_comparable_cmp (self, other)`: It compares `self` and `other`. The first argument `self` is an instance on which `t_comparable_cmp` runs. The second argument `other` is another instance. This function needs to be overridden in an object which implements the interface.
  - If `self` is equal to `other`, `t_comparable_cmp` returns 0.
  - If `self` is greater than `other`, `t_comparable_cmp` returns 1.
  - If `self` is less than `other`, `t_comparable_cmp` returns -1.

    - If an error happens, `t_comparable_cmp` returns -2.
- `t_comparable_eq (self, other)`: It returns TRUE if `self` is equal to `other`. Otherwise it returns FALSE. You need to be careful that FALSE is returned even if an error occurs.
- `t_comparable_gt (self, other)`: It returns TRUE if `self` is greater than `other`. Otherwise it returns FALSE.
- `t_comparable_lt (self, other)`: It returns TRUE if `self` is less than `other`. Otherwise it returns FALSE.
- `t_comparable_ge (self, other)`: It returns TRUE if `self` is greater than or equal to `other`. Otherwise it returns FALSE.
- `t_comparable_le (self, other)`: It returns TRUE if `self` is less than or equal to `other`. Otherwise it returns FALSE.

Numbers and strings are comparable. TInt, TDouble and TStr implement TComparable interface so that they can use the functions above. In addition, TNumStr can use the functions because it is a child class of TStr.

For example,

```
TInt *i1 = t_int_new_with_value (10);
TInt *i2 = t_int_new_with_value (20);
t_comparable_eq (T_COMPARABLE (i1), T_COMPARABLE (i2)); /* => FALSE */
t_comparable_lt (T_COMPARABLE (i1), T_COMPARABLE (i2)); /* => TRUE */
```

What's the difference between interface and abstract class? Virtual functions in an abstract class are overridden by a function in its descendant classes. Virtual functions in an interface are overridden by a function in any classes. Compare TNumber and TComparable.

- A function `t_number_add` is overridden in TIntClass and TDoubleClass. It can't be overridden in TStrClass because TStr isn't a descendant of TNumber.
- A function `t_comparable_cmp` is overridden in TIntClass, TDoubleClass and TStrClass.

9.1 TComparable interface

Defining interfaces is similar to defining objects.

- Use `G_DECLARE_INTERFACE` instead of `G_DECLARE_FINAL_TYPE`.
- Use `G_DEFINE_INTERFACE` instead of `G_DEFINE_TYPE`.

Now let's see the header file.

```
#pragma once

#include <glib-object.h>

#define T_TYPE_COMPARABLE  (t_comparable_get_type ())
G_DECLARE_INTERFACE (TComparable, t_comparable, T, COMPARABLE, GObject)

struct _TComparableInterface {
  GTypeInterface parent;
  /* signal */
  void (*arg_error) (TComparable *self);
  /* virtual function */
  int (*cmp) (TComparable *self, TComparable *other);
};

/* t_comparable_cmp */
/* if self > other, then returns 1 */
/* if self = other, then returns 0 */
/* if self < other, then returns -1 */
/* if error happens, then returns -2 */

int
t_comparable_cmp (TComparable *self, TComparable *other);

gboolean
```

```
26 t_comparable_eq (TComparable *self, TComparable *other);
27
28 gboolean
29 t_comparable_gt (TComparable *self, TComparable *other);
30
31 gboolean
32 t_comparable_lt (TComparable *self, TComparable *other);
33
34 gboolean
35 t_comparable_ge (TComparable *self, TComparable *other);
36
37 gboolean
38 t_comparable_le (TComparable *self, TComparable *other);
```

- 6: `G_DECLARE_INTERFACE` macro. The last parameter is a prerequisite of the interface. The prerequisite of TComparable is GObject. So, any other object than the descendants of GObject, for example GVariant, can't implement TComparable. A prerequisite is the GType of either an interface or a class. This macro expands to:
  - Declaration of `t_comparable_get_type()`.
  - `Typedef struct _TComparableInterface TComparableInterface`
  - `T_COMPARABLE ()` macro. It casts an instance to TComparable type.
  - `T_IS_COMPARABLE ()` macro. It checks if the type of an instance is `T_TYPE_COMPARABLE`.
  - `T_COMPARABLE_GET_IFACE ()` macro. It gets the interface of the instance which is given as an argument.
- 8-14: `TComparableInterface` structure. This is similar to a class structure. The first member is the parent interface. The parent of `TComparableInterface` is `GTypeInterface`. `GTypeInterface` is a base of all interface types. It is like a `GTypeClass` which is a base of all class types. `GTypeClass` is the first member of the structure `GObjectClass`. (See `gobject.h`. Note that `GObjectClass` is the same as `struct _GObjectClass`.) The next member is a pointer `arg_error` to the default signal handler of "arg-error" signal. This signal is emitted when the second argument of the comparison function is inappropriate. For example, if `self` is TInt and `other` is TStr, both of them are Comparable instance. But they are *not* able to compare. This is because `other` isn't TNumber. The last member `cmp` is a pointer to a comparison method. It is a virtual function and is expected to be overridden by a function in the object which implements the interface.
- 22-38: Public functions.

C file `tcomparable.c` is as follows.

```
1  #include "tcomparable.h"
2
3  static guint t_comparable_signal;
4
5  G_DEFINE_INTERFACE (TComparable, t_comparable, G_TYPE_OBJECT)
6
7  static void
8  arg_error_default_cb (TComparable *self) {
9    g_printerr ("\nTComparable:␣argument␣error.\n");
10 }
11
12 static void
13 t_comparable_default_init (TComparableInterface *iface) {
14   /* virtual function */
15   iface->cmp = NULL;
16   /* argument error signal */
17   iface->arg_error = arg_error_default_cb;
18   t_comparable_signal =
19   g_signal_new ("arg-error",
20                 T_TYPE_COMPARABLE,
21                 G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE | G_SIGNAL_NO_HOOKS,
22                 G_STRUCT_OFFSET (TComparableInterface, arg_error),
23                 NULL /* accumulator */,
24                 NULL /* accumulator data */,
25                 NULL /* C marshaller */,
26                  G_TYPE_NONE /* return_type */,
27                  0      /* n_params */
28                  );
29 }
30
31 int
32 t_comparable_cmp (TComparable *self , TComparable *other) {
33   g_return_val_if_fail (T_IS_COMPARABLE (self) , -2);
34
35   TComparableInterface *iface = T_COMPARABLE_GET_IFACE (self);
36
37   return (iface->cmp == NULL ? -2 : iface->cmp (self , other));
38 }
39
40 gboolean
41 t_comparable_eq (TComparable *self , TComparable *other) {
42   return (t_comparable_cmp (self , other) == 0);
43 }
44
45 gboolean
46 t_comparable_gt (TComparable *self , TComparable *other) {
47   return (t_comparable_cmp (self , other) == 1);
48 }
49
50 gboolean
51 t_comparable_lt (TComparable *self , TComparable *other) {
52   return (t_comparable_cmp (self , other) == -1);
53 }
54
55 gboolean
56 t_comparable_ge (TComparable *self , TComparable *other) {
57   int result = t_comparable_cmp (self , other);
58   return (result == 1 || result == 0);
59 }
60
61 gboolean
62 t_comparable_le (TComparable *self , TComparable *other) {
63   int result = t_comparable_cmp (self , other);
64   return (result == -1 || result == 0);
65 }
```

- 5: `G_DEFINE_INTERFACE` macro. The third parameter is the type of the prerequisite. This macro expands to:
  - Declaration of `t_comparable_default_init ()`.
  - Definition of `t_comparable_get_type ()`.
- 7-10: `arg_error_default_cb` is a default signal handler of "arg-error" signal.
- 12- 29: `t_comparable_default_init` function. This function is similar to class initialization function. It initializes `TComparableInterface` structure.
- 15: Assigns NULL to `cmp`. So, the comparison method doesn't work before an implementation class overrides it.
- 17: Set the default signal handler of the signal "arg-error".
- 18-28: Creates a signal "arg-error".
- 31-38: The function `t_comparable_cmp`. It checks the type of `self` on the first line. If it isn't comparable, it logs the error message and returns -2 (error). If `iface->cmp` is NULL (it means the class method hasn't been overridden), then it returns -2. Otherwise it calls the class method and returns the value returned by the class method.
- 40-65: Public functions. These five functions are based on `t_comparable_cmp`. Therefore, no overriding is necessary for them. For example, `t_comparable_eq` just calls `t_comparable_cmp`. And it returns TRUE if `t_comparable_cmp` returns zero. Otherwise it returns FALSE.

This program uses a signal to give the argument type error information to a user. This error is usually a program error rather than a run-time error. Using a signal to report a program error is not a good way. The best way is using `g_return_if_fail`. The reason why I made this signal is just I wanted to show how to implement a signal in interfaces.

9.2 Implementing interface

TInt, TDouble and TStr implement TComparable. First, look at TInt. The header file is the same as before. The implementation is written in C file.

`tint.c` is as follows.

```c
#include "../tnumber/tnumber.h"
#include "../tnumber/tint.h"
#include "../tnumber/tdouble.h"
#include "tcomparable.h"

enum {
  PROP_0,
  PROP_INT,
  N_PROPERTIES
};

static GParamSpec *int_properties[N_PROPERTIES] = {NULL, };

struct _TInt {
  TNumber parent;
  int value;
};

static void t_comparable_interface_init (TComparableInterface *iface);

G_DEFINE_TYPE_WITH_CODE (TInt, t_int, T_TYPE_NUMBER,
                         G_IMPLEMENT_INTERFACE (T_TYPE_COMPARABLE,
                            t_comparable_interface_init))

static int
t_int_comparable_cmp (TComparable *self, TComparable *other) {
  if (! T_IS_NUMBER (other)) {
    g_signal_emit_by_name (self, "arg-error");
    return -2;
  }

  int i;
  double s, o;

  s = (double) T_INT (self)->value;
  if (T_IS_INT (other)) {
    g_object_get (other, "value", &i, NULL);
    o = (double) i;
  } else
    g_object_get (other, "value", &o, NULL);
  if (s > o)
    return 1;
  else if (s == o)
    return 0;
  else if (s < o)
    return -1;
  else /* This can't happen. */
    return -2;
}

static void
t_comparable_interface_init (TComparableInterface *iface) {
  iface->cmp = t_int_comparable_cmp;
}

static void
```

```c
t_int_set_property (GObject *object, guint property_id, const GValue *value,
    GParamSpec *pspec) {
  TInt *self = T_INT (object);

  if (property_id == PROP_INT)
    self->value = g_value_get_int (value);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_int_get_property (GObject *object, guint property_id, GValue *value, GParamSpec
    *pspec) {
  TInt *self = T_INT (object);

  if (property_id == PROP_INT)
    g_value_set_int (value, self->value);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_int_init (TInt *self) {
}

/* arithmetic operator */
/* These operators create a new instance and return a pointer to it. */
#define t_int_binary_op(op) \
  int i; \
  double d; \
  if (T_IS_INT (other)) { \
    g_object_get (T_INT (other), "value", &i, NULL); \
    return  T_NUMBER (t_int_new_with_value (T_INT(self)->value op i)); \
  } else { \
    g_object_get (T_DOUBLE (other), "value", &d, NULL); \
    return  T_NUMBER (t_int_new_with_value (T_INT(self)->value op (int) d)); \
  }

static TNumber *
t_int_add (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (+)
}

static TNumber *
t_int_sub (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (-)
}

static TNumber *
t_int_mul (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  t_int_binary_op (*)
}

static TNumber *
t_int_div (TNumber *self, TNumber *other) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  int i;
  double d;

  if (T_IS_INT (other)) {
    g_object_get (T_INT (other), "value", &i, NULL);
    if (i == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_int_new_with_value (T_INT(self)->value / i));
  } else {
    g_object_get (T_DOUBLE (other), "value", &d, NULL);
    if (d == 0) {
      g_signal_emit_by_name (self, "div-by-zero");
      return NULL;
    } else
      return  T_NUMBER (t_int_new_with_value (T_INT(self)->value / (int)  d));
  }
}

static TNumber *
t_int_uminus (TNumber *self) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  return T_NUMBER (t_int_new_with_value (- T_INT(self)->value));
}

static char *
t_int_to_s (TNumber *self) {
  g_return_val_if_fail (T_IS_INT (self), NULL);

  int i;

  g_object_get (T_INT (self), "value", &i, NULL);
  return g_strdup_printf ("%d", i);
}

static void
t_int_class_init (TIntClass *class) {
  TNumberClass *tnumber_class = T_NUMBER_CLASS (class);
  GObjectClass *gobject_class = G_OBJECT_CLASS (class);

  /* override virtual functions */
  tnumber_class->add = t_int_add;
  tnumber_class->sub = t_int_sub;
  tnumber_class->mul = t_int_mul;
  tnumber_class->div = t_int_div;
  tnumber_class->uminus = t_int_uminus;
  tnumber_class->to_s = t_int_to_s;

  gobject_class->set_property = t_int_set_property;
  gobject_class->get_property = t_int_get_property;
  int_properties[PROP_INT] = g_param_spec_int ("value", "val", "Integer␣value",
      G_MININT, G_MAXINT, 0, G_PARAM_READWRITE);
  g_object_class_install_properties (gobject_class, N_PROPERTIES, int_properties);
}

TInt *
t_int_new_with_value (int value) {
  TInt *i;

  i = g_object_new (T_TYPE_INT, "value", value, NULL);
  return i;
}

```

```
181 TInt *
182 t_int_new (void) {
183   TInt *i;
184 
185   i = g_object_new (T_TYPE_INT, NULL);
186   return i;
187 }
```

- 4: It needs to include the header file of TComparable.
- 19: Declaration of `t_comparable_interface_init ()` function. This declaration must be done before `G_DEFINE_TYPE_WITH_CODE` macro.
- 21-22: `G_DEFINE_TYPE_WITH_CODE` macro. The last parameter is `G_IMPLEMENT_INTERFACE` macro. The second parameter of `G_IMPLEMENT_INTERFACE` is `t_comparable_interface_init`. These two macros expands to:
  - Declaration of `t_int_class_init ()`.
  - Declaration of `t_int_init ()`.
  - Definition of `t_int_parent_class` static variable which points to the parent's class.
  - Definition of `t_int_get_type ()`. This function includes `g_type_register_static_simple ()` and `g_type_add_interface_static ()`. The function `g_type_register_static_simple ()` is a convenient version of `g_type_register_static ()`. It registers TInt type to the type system. The function `g_type_add_interface_static ()` adds an interface type to an instance type. There is a good example in GObject Reference Manual, Interfaces. If you want to know how to write codes without the macros, see `tint_without_macro.c`.
- 24-48: `t_int_comparable_cmp` is a function to compare TInt instance to TNumber instance.
- 26-29: Checks the type of `other`. If the argument type is not TNumber, it emits "arg-error" signal with `g_signal_emit_by_name`.
- 34: Converts `self` into double.
- 35-39: Gets the value of `other` and if it is TInt then the value is casted to double.
- 40-47: compares `s` and `o` and returns 1, 0, -1 and -2.
- 50-53: `t_comparable_interface_init`. This function is called in the initialization process of TInt. The function `t_int_comparable_cmp` is assigned to `iface->cmp`.

`tdouble.c` is almost same as `tint.c`. These two objects can be compared because int is casted to double before the comparison.

`tstr.c` is as follows.

```
1  #include "../tstr/tstr.h"
2  #include "tcomparable.h"
3  
4  enum {
5    PROP_0,
6    PROP_STRING,
7    N_PROPERTIES
8  };
9  
10 static GParamSpec *str_properties[N_PROPERTIES] = {NULL, };
11 
12 typedef struct {
13   char *string;
14 } TStrPrivate;
15 
16 static void t_comparable_interface_init (TComparableInterface *iface);
17 
18 G_DEFINE_TYPE_WITH_CODE (TStr, t_str, G_TYPE_OBJECT,
19                             G_ADD_PRIVATE (TStr)
20                             G_IMPLEMENT_INTERFACE (T_TYPE_COMPARABLE,
                                   t_comparable_interface_init))
21 
22 static void
23 t_str_set_property (GObject *object, guint property_id, const GValue *value,
      GParamSpec *pspec) {
24   TStr *self = T_STR (object);
```

```c

/* The returned value of the function g_value_get_string can be NULL. */
/* The function t_str_set_string calls a class method, */
/* which is expected to rewrite in the descendant object. */
  if (property_id == PROP_STRING)
    t_str_set_string (self, g_value_get_string (value));
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

static void
t_str_get_property (GObject *object, guint property_id, GValue *value, GParamSpec
    *pspec) {
  TStr *self = T_STR (object);
  TStrPrivate *priv = t_str_get_instance_private (self);

/* The second argument of the function g_value_set_string can be NULL. */
  if (property_id == PROP_STRING)
    g_value_set_string (value, priv->string);
  else
    G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
}

/* This function just set the string. */
/* So, no notify signal is emitted. */
static void
t_str_real_set_string (TStr *self, const char *s) {
  TStrPrivate *priv = t_str_get_instance_private (self);

  if (priv->string)
    g_free (priv->string);
  priv->string = g_strdup (s);
}

static void
t_str_finalize (GObject *object) {
  TStr *self = T_STR (object);
  TStrPrivate *priv = t_str_get_instance_private (self);

  if (priv->string)
    g_free (priv->string);
  G_OBJECT_CLASS (t_str_parent_class)->finalize (object);
}

static int
t_str_comparable_cmp (TComparable *self, TComparable *other) {
  if (! T_IS_STR (other)) {
    g_signal_emit_by_name (self, "arg-error");
    return -2;
  }

  char *s, *o;
  int result;

  s = t_str_get_string (T_STR (self));
  o = t_str_get_string (T_STR (other));

  if (strcmp (s, o) > 0)
    result = 1;
  else if (strcmp (s, o) == 0)
    result = 0;
  else if (strcmp (s, o) < 0)
    result = -1;
  else /* This can't happen. */
```

```
    result = -2;
  g_free (s);
  g_free (o);
  return result;
}

static void
t_comparable_interface_init (TComparableInterface *iface) {
  iface->cmp = t_str_comparable_cmp;
}

static void
t_str_init (TStr *self) {
  TStrPrivate *priv = t_str_get_instance_private (self);

  priv->string = NULL;
}

static void
t_str_class_init (TStrClass *class) {
  GObjectClass *gobject_class = G_OBJECT_CLASS (class);

  gobject_class->finalize = t_str_finalize;
  gobject_class->set_property = t_str_set_property;
  gobject_class->get_property = t_str_get_property;
  str_properties[PROP_STRING] = g_param_spec_string ("string", "str", "string", "",
      G_PARAM_READWRITE);
  g_object_class_install_properties (gobject_class, N_PROPERTIES, str_properties);

  class->set_string = t_str_real_set_string;
}

/* setter and getter */
void
t_str_set_string (TStr *self, const char *s) {
  g_return_if_fail (T_IS_STR (self));
  TStrClass *class = T_STR_GET_CLASS (self);

/* The setter calls the class method 'set_string', */
/* which is expected to be overridden by the descendant TNumStr. */
/* Therefore, the behavior of the setter is different between TStr and TNumStr. */
  class->set_string (self, s);
}

char *
t_str_get_string (TStr *self) {
  g_return_val_if_fail (T_IS_STR (self), NULL);
  TStrPrivate *priv = t_str_get_instance_private (self);

  return g_strdup (priv->string);
}

TStr *
t_str_concat (TStr *self, TStr *other) {
  g_return_val_if_fail (T_IS_STR (self), NULL);
  g_return_val_if_fail (T_IS_STR (other), NULL);

  char *s1, *s2, *s3;
  TStr *str;

  s1 = t_str_get_string (self);
  s2 = t_str_get_string (other);
  if (s1 && s2)
    s3 = g_strconcat (s1, s2, NULL);
```

```
151   else if (s1)
152     s3 = g_strdup (s1);
153   else if (s2)
154     s3 = g_strdup (s2);
155   else
156     s3 = NULL;
157   str = t_str_new_with_string (s3);
158   if (s1) g_free (s1);
159   if (s2) g_free (s2);
160   if (s3) g_free (s3);
161   return str;
162 }
163 
164 /* create a new TStr instance */
165 TStr *
166 t_str_new_with_string (const char *s) {
167   return T_STR (g_object_new (T_TYPE_STR, "string", s, NULL));
168 }
169 
170 TStr *
171 t_str_new (void) {
172   return T_STR (g_object_new (T_TYPE_STR, NULL));
173 }
```

- 16: Declares `t_comparable_interface_init` function. It needs to be declared before `G_DEFINE_TYPE_WITH_CODE` macro.
- 18-20: `G_DEFINE_TYPE_WITH_CODE` macro. Because TStr is derivable type, its private area (TStrPrivate) is needed. `G_ADD_PRIVATE` macro makes the private area. Be careful that there's no comma after `G_ADD_PRIVATE` macro.
- 68-92: `t_str_comparable_cmp`.
- 70-73: Checks the type of `other`. If it is not TStr, "arg-error" signal is emitted.
- 78-79: Gets strings `s` and `o` from TStr objects `self` and `other`.
- 81-88: Compares `s` and `o` with the C standard function `strcmp`.
- 89-90: Frees `s` and `o`.
- 91: Returns the result.
- 94-97: `t_comparable_interface_init` function. It overrides `iface->comp` with `t_str_comparable_cmp`.

TStr can be compared with TStr, but not with TInt nor TDouble. Generally, comparison is available between two same type instances.

TNumStr itself doesn't implement TComparable. But it is a child of TStr, so it is comparable. The comparison is based on the alphabetical order. So, "a" is bigger than "b" and "three" is bigger than "two".

9.3 Test program.

`main.c` is a test program.

```
#include <glib-object.h>
#include "tcomparable.h"
#include "../tnumber/tnumber.h"
#include "../tnumber/tint.h"
#include "../tnumber/tdouble.h"
#include "../tstr/tstr.h"

static void
t_print (const char *cmp, TComparable *c1, TComparable *c2) {
  char *s1, *s2;
  TStr *ts1, *ts2, *ts3;

  ts1 = t_str_new_with_string("\"");
  if (T_IS_NUMBER (c1))
    s1 = t_number_to_s (T_NUMBER (c1));
  else if (T_IS_STR (c1)) {
    ts2 = t_str_concat (ts1, T_STR (c1));
    ts3 = t_str_concat (ts2, ts1);
    s1 = t_str_get_string (T_STR (ts3));
    g_object_unref (ts2);
    g_object_unref (ts3);
  } else {
    g_print ("c1␣isn't␣TInt,␣TDouble␣nor␣TStr.\n");
    return;
  }
  if (T_IS_NUMBER (c2))
    s2 = t_number_to_s (T_NUMBER (c2));
  else if (T_IS_STR (c2)) {
    ts2 = t_str_concat (ts1, T_STR (c2));
    ts3 = t_str_concat (ts2, ts1);
    s2 = t_str_get_string (T_STR (ts3));
    g_object_unref (ts2);
    g_object_unref (ts3);
  } else {
    g_print ("c2␣isn't␣TInt,␣TDouble␣nor␣TStr.\n");
    return;
  }
  g_print ("%s␣%s␣%s.\n", s1, cmp, s2);
  g_object_unref (ts1);
  g_free (s1);
  g_free (s2);
}

static void
compare (TComparable *c1, TComparable *c2) {
  if (t_comparable_eq (c1, c2))
    t_print ("equals", c1, c2);
  else if (t_comparable_gt (c1, c2))
    t_print ("is␣greater␣than", c1, c2);
  else if (t_comparable_lt (c1, c2))
    t_print ("is␣less␣than", c1, c2);
  else if (t_comparable_ge (c1, c2))
    t_print ("is␣greater␣than␣or␣equal␣to", c1, c2);
  else if (t_comparable_le (c1, c2))
    t_print ("is␣less␣than␣or␣equal␣to", c1, c2);
  else
    t_print ("can't␣compare␣to", c1, c2);
}

int
main (int argc, char **argv) {
  const char *one = "one";
  const char *two = "two";
  const char *three = "three";
  TInt *i;
  TDouble *d;
  TStr *str1, *str2, *str3;

  i = t_int_new_with_value (124);
  d = t_double_new_with_value (123.45);
  str1 = t_str_new_with_string (one);
  str2 = t_str_new_with_string (two);
  str3 = t_str_new_with_string (three);

  compare (T_COMPARABLE (i), T_COMPARABLE (d));
  compare (T_COMPARABLE (str1), T_COMPARABLE (str2));
  compare (T_COMPARABLE (str2), T_COMPARABLE (str3));
  compare (T_COMPARABLE (i), T_COMPARABLE (str1));

  g_object_unref (i);
  g_object_unref (d);
```

```
82    g_object_unref (str1);
83    g_object_unref (str2);
84    g_object_unref (str3);
85
86    return 0;
87  }
```

- 8-42: The function `t_print` has three parameters and builds an output string, then shows it to the display. When it builds the output, strings are surrounded with double quotes.
- 44-58: The function `compare` compares two TComparable objects and calls `t_print` to display the result.
- 60-87: `main` function.
- 69-73: Creates TInt, TDouble and three TStr instances. They are given values.
- 75: Compares TInt and TDouble.
- 76-77: Compares two TStr.
- 78: Compare TInt to TStr. This makes an "arg-error".
- 80-84 Frees objects.

9.4 Compilation and execution

Change your current directory to src/tcomparable.

```
$ cd src/tcomparable
$ meson setup _build
$ ninja -C _build
```

Then execute it.

```
$ _build/tcomparable
124 is greater than 123.450000.
"one" is less than "two".
"two" is greater than "three".

TComparable: argument error.

TComparable: argument error.

TComparable: argument error.

TComparable: argument error.

TComparable: argument error.
124 can't compare to "one".
```

9.5 Build an interface without macros

We used macros such as `G_DECLARE_INTERFACE`, `G_DEFINE_INTERFACE` to build an interface. And `G_DEFINE_TYPE_WITH_CODE` to implement the interface. We can also build it without macros. There are three files in the `tcomparable` directory.

- `tcomparable_without_macro.h`
- `tcomparable_without_macro.c`
- `tint_without_macro.c`

They don't use macros. Instead, they register the interface or implementation of the interface to the type system directly. If you want to know that, see the source files in src/tcomparable.

9.6 GObject system and object oriented languages

If you know any object oriented languages, you probably have thought that GObject and the languages are similar. Learning such languages is very useful to know GObject system.