

# Créer son widget GTK+ en Langage C

par [Franck.H](#)

Date de publication : 10/09/2006

Dernière mise à jour :

Ce tutoriel explique comment créer un widget GTK+ en Langage C.

- I - Introduction
- II - Le fichier d'entête
- III - Le fichier source
- IV - Notes
- V - Code source complet
- VI - Remerciements

## I - Introduction

La bibliothèque GTK+ possède beaucoup de widgets, mais il peut cependant arriver que vous ayez besoin d'un widget spécifique, ou d'en modifier un existant. Dans ce tutoriel, je vais vous décrire une méthode de création de widget qui s'appuie simplement sur GtkWidget, comme parent ; Nous ferons donc un widget en partant de rien !

Le widget que nous allons réaliser est un widget représentant une Led avec ses deux états, allumée/éteinte, état qui pourra être changé par le biais d'un bouton, par exemple dans un programme.

Le nom de notre widget: GtkLed !



Si vous avez des questions concernant GTK+ l'équipe de [developpez.com](#) sera ravie de vous aider, rendez-vous sur le [Forum GTK+ de developpez.com](#)

## II - Le fichier d'entête

D'après les conventions de GTK+, chaque widget possède un fichier d'entête, un seul, nous allons voir ensemble celui de notre widget. Ce que nous savons, c'est que notre widget va hériter du type **GtkWidget** (son parent), il nous faut donc inclure, après **gtk.h**, l'entête **gtkwidget.h** :

```
/* gtkled.h */

#ifndef __GTK_LED_H
#define __GTK_LED_H

#include <gtk/gtk.h>
#include <gtk/gtkwidget.h>

G_BEGIN_DECLS
```

La macro `G_BEGIN_DECLS`, qui suit l'inclusion des entêtes, nous rappelle le C++.

On peut remarquer une convention à respecter pour le développement de widgets GTK+, c'est la forme des identifiants des macros de protection contre les inclusions multiples, il faut, en effet, commencer par deux traits de soulignement (*underscore*): `__` , puis le nom du module et enfin le type du fichier.

Voyons maintenant les macros dont nous avons besoin, elles sont au nombre de trois:

```
#define GTK_LED(obj) \
    GTK_CHECK_CAST(obj, gtk_led_get_type (), GtkLed)
#define GTK_LED_CLASS(klass) \
    GTK_CHECK_CLASS_CAST(klass, gtk_led_get_type (), GtkLedClass)
#define GTK_IS_LED(obj) \
    GTK_CHECK_TYPE(obj, gtk_led_get_type ())
```

- **GTK\_LED** permet de fournir un pointeur vers un pointeur de type `GtkLed` : si le pointeur n'a pas le bon type, l'action échoue, et un message d'erreur est envoyé sur la sortie standard.
- **GTK\_LED\_CLASS** permet la même action que `GTK_LED`, mais pour fournir un pointeur vers un pointeur de type `GtkLedClass`.
- **GTK\_IS\_LED** permet de déterminer si l'objet passé en paramètre est du type `GtkLed` : renvoie **TRUE** si oui, **FALSE** sinon.

Passons maintenant à la création des structures de notre widget. On peut voir une autre convention de GTK+ qui s'applique aux noms des structures qui commencent tous avec un trait de soulignement ( `_` ), puis le nom du widget dont chaque mot doit commencer par une majuscule:

```
typedef struct _GtkLed GtkLed;
typedef struct _GtkLedClass GtkLedClass;

struct _GtkLed {
    GtkWidget widget;

    const gchar * on_image;
    const gchar * off_image;

    GdkPixbuf * on_pixbuf;
    GdkPixbuf * off_pixbuf;

    gint width;
    gint height;

    gboolean state;
};
```

```

struct _GtkLedClass {
    GtkWidgetClass parent_class;
};

```

Jusque là, rien de bien nouveau, deux **typedef** et deux structures, mais un point important subsiste dans la structure **\_GtkLed** : *GtkWidget widget*,

L'intégration d'un **GtkWidget**, au début de la structure **\_GtkLed**, permet clairement d'identifier le widget comme parent type de **GtkWidget**. A ce propos, il est une chose essentielle à noter.

En effet, chaque structure de widget doit commencer par le type du parent qui contiendra la même adresse que notre **GtkLed**. Cette pratique permet simplement de faire pointer chaque objet parent dans la hiérarchie sur la même adresse donc, un pointeur **GtkLed** pourra être considéré comme un **GtkWidget**, tout comme un **GtkWidget** peut être considéré comme un **GtkObject**, et ainsi de suite.

Nous aurons donc la hiérarchie suivante pour notre widget:

```

GObject
+----GInitiallyUnowned
      +----GtkObject
            +----GtkWidget
                  +----GtkLed

```

Les autres variables et/ou pointeurs sont spécifiques à notre widget. Cette structure sert donc à stocker toutes les informations relatives à une instance de l'objet, chaque instance possédera alors ses propres valeurs.

Viens ensuite la structure **\_GtkLedClass** qui elle, sert justement pour toutes les instances de notre widget ; C'est ici que peuvent être stockées les valeurs et fonctions devant agir sur toutes les instances, dans notre cas, juste une classe parente du type **GtkWidget** nous est utile, et d'ailleurs même obligatoire.

Voyons à présent les fonctions publiques dont nous aurons besoin:

```

GtkWidget * gtk_led_get_type (void);
gboolean gtk_led_get_state (GtkLed * led);
void gtk_led_set_state (GtkLed * led, gboolean state);
GtkWidget * gtk_led_new (const gchar * on_image,
                        const gchar * off_image,
                        gboolean state);

G_END_DECLS

#endif /* __GTK_LED_H */

```

- **gtk\_led\_get\_type** permet d'identifier notre widget par le biais d'un numéro unique au sein de la bibliothèque.
- **gtk\_led\_get\_state** permet de déterminer l'état de notre Led, elle renvoie **TRUE** si elle est allumée, **FALSE** sinon.

- **gtk\_led\_set\_state** permet de changer l'état de la Led. Le second argument prend les valeurs suivantes:
  - **TRUE**, Led allumée.
  - **FALSE**, Led éteinte.
- **gtk\_led\_new** est le constructeur de notre widget, c'est lui qu'il nous faudra appeler pour la création d'une nouvelle instance de notre widget, voici ses arguments:
  - **const gchar \* on\_image** : Image de l'état « allumé » de la Led.
  - **const gchar \* off\_image** : Image de l'état « éteint » de la Led.
  - **gboolean state**: L'état par défaut de la Led.

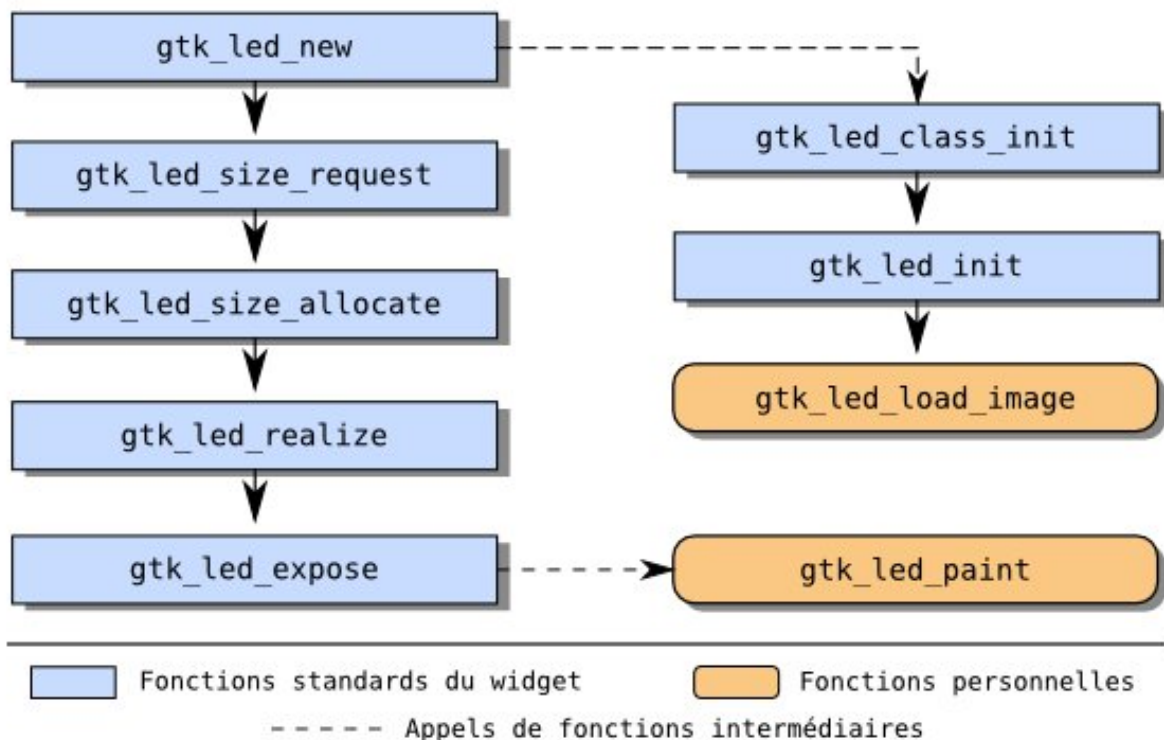
### III - Le fichier source

Voyons à présent l'implémentation de nos fonctions, publiques et privées, par la liste des fonctions privées:

```
/* gtkled.c */
#include "gtkled.h"

static void gtk_led_class_init (GtkLedClass * klass);
static void gtk_led_init (GtkLed * led);
static void gtk_led_load_image (GtkLed * led);
static void gtk_led_size_request (GtkWidget * widget,
                                 GtkRequisition * requisition);
static void gtk_led_size_allocate (GtkWidget * widget,
                                   GtkAllocation * allocation);
static void gtk_led_realize (GtkWidget * widget);
static gboolean gtk_led_expose (GtkWidget * widget,
                                GdkEventExpose * event);
static void gtk_led_paint (GtkWidget * widget);
static void gtk_led_destroy (GtkObject * object);
```

Je vais vous les décrire une par une, mais voyons tout d'abord l'ordre des appels de ces fonctions:



Ce qu'on peut remarquer de particulier dans ce schéma, c'est l'appel de la fonction **gtk\_led\_new** qui, elle-même, appelle automatiquement **gtk\_led\_class\_init** et **gtk\_led\_init**, puis une fois revenue appelle la fonction personnelle **gtk\_led\_load\_image**, dont la fonction est le chargement des images passées en argument à notre fonction **gtk\_led\_new**.

**GtkType gtk\_led\_get\_type (void)**

```

GtkType
gtk_led_get_type (void)
{
    static GtkType gtk_led_type = 0;

    if (! gtk_led_type) {
        static const GtkTypeInfo gtk_led_info = {
            "GtkLed",
            sizeof (GtkLed),
            sizeof (GtkLedClass),
            (GtkClassInitFunc) gtk_led_class_init,
            (GtkObjectInitFunc) gtk_led_init,
            NULL,
            NULL,
            (GtkClassInitFunc) NULL
        };
        gtk_led_type = gtk_type_unique (GTK_TYPE_WIDGET, &gtk_led_info);
    }

    return gtk_led_type;
}

```

Cette fonction est une fonction publique obligatoire, celle-ci sert à retrouver le type de notre widget par le numéro d'identification unique de l'objet que GTK+ lui à attribué, lors de son enregistrement !

Vous n'avez ici qu'à recopier simplement cette fonction, et en changer les noms relatifs à votre widget.

L'appel de cette fonction ne se fait qu'une seule fois pendant la durée de vie d'un widget, et c'est au moment de l'appel de **gtk\_led\_new**. La variable **gtk\_led\_type** permet de stocker le numéro unique que GTK+, créé lors de l'appel à **gtk\_type\_unique**, puis nous renvoyons cette valeur à la fin de la fonction.

**gboolean gtk\_led\_get\_state (GtkLed \* led)**

```

gboolean
gtk_led_get_state (GtkLed * led)
{
    return led->state;
}

```

**void gtk\_led\_set\_state (GtkLed \* led, gboolean state)**

```

void
gtk_led_set_state (GtkLed * led, gboolean state)
{
    led->state = state;
    gtk_led_paint (GTK_WIDGET (led));
}

```

Voyons, rapidement, nos fonctions publiques de récupération et changement d'état de notre widget. Ici, rien de bien compliqué, **gtk\_led\_get\_state** renvoi **TRUE** si la Led est allumée et **FALSE** sinon. Puis, **gtk\_led\_set\_state** permet de changer l'état de la Led du widget, grâce au second argument qui prend les mêmes valeurs que le retour de la fonction précédente, et appelle notre fonction privée **gtk\_led\_paint**, pour mettre à jour l'affichage du widget !

```
GtkWidget * gtk_led_new (const gchar * on_image, const gchar * off_image, gboolean state)
```

```
GtkWidget *
gtk_led_new (const gchar * on_image,
             const gchar * off_image,
             gboolean state)
{
    GtkLed * led;

    if (on_image != NULL && off_image != NULL) {
        led = gtk_type_new (gtk_led_get_type ());

        if (led != NULL) {
            led->on_image = on_image;
            led->off_image = off_image;
            led->state = state;

            gtk_led_load_image (led);
        }
    }

    return GTK_WIDGET (led);
}
```

Notre widget possède qu'une seule fonction de création, alors que certains widgets en possèdent plusieurs, suivant si vous voulez initialiser le widget pendant sa création, ou simplement le créer comme un **GtkLabel** ; Par exemple, on peut en créer un vide, ou appeler une autre fonction de création, dont on va devoir fournir le texte à insérer dans le widget.

C'est pendant l'appel à **gtk\_type\_new** que les fonctions **gtk\_led\_class\_init**, et **gtk\_led\_init**, sont appelées, comme le montre le schéma plus haut.

Une fois le nouveau type créé avec succès, on copie les arguments dans les éléments de la structure du widget, c'est le seul moment où l'on peut le faire. Puis nous chargeons les images en appelant la fonction **gtk\_led\_load\_image**, c'est également le seul moment, et surtout le plus adéquat pour le changement des images car, avant, les chemins ne sont pas copiés vers les pointeurs de la structure du widget ; Enfin, **gtk\_led\_new** est également appelé une seule fois, donc c'est le moment le plus propice !

Pour finir, on retourne notre nouveau pointeur, initialisé, en le transtypant vers un type GtkWidget, grâce à la macro **GTK\_WIDGET**.

```
static void gtk_led_class_init (GtkLedClass * klass)
```

```
static void
gtk_led_class_init (GtkLedClass * klass)
{
    GtkWidgetClass * widget_class;
    GObjectClass * object_class;

    widget_class = (GtkWidgetClass *) klass;
    object_class = (GObjectClass *) klass;

    widget_class->realize = gtk_led_realize;
    widget_class->size_request = gtk_led_size_request;
    widget_class->size_allocate = gtk_led_size_allocate;
    widget_class->expose_event = gtk_led_expose;

    object_class->destroy = gtk_led_destroy;
}
```

Le rôle principal de cette fonction est d'établir la liaison avec les fonctions de rappel (callback) que nous voulons redéfinir pour notre widget, en transmettant leur adresse au pointeur de fonction adéquat.

On peut retrouver la liste complète des fonctions de rappel d'un GtkWidget à l'adresse suivante dans la partie **Signals** : <http://developer.gnome.org/doc/API/2.0/gtk/GtkWidget.html>

Pour la classe d'un GObject, il n'y a que la fonction de rappel destroy qui est prise en charge !

Si le widget dispose de données partagées entre toutes les instances de cette classe, il convient également d'en faire l'initialisation dans cette fonction !

```
static void gtk_led_init (GtkLed * led)
```

```
static void
gtk_led_init (GtkLed * led)
{
    led->on_image = NULL;
    led->off_image = NULL;
    led->on_pixbuf = NULL;
    led->off_pixbuf = NULL;
    led->width = 0;
    led->height = 0;
    led->state = FALSE;
}
```

Tout comme **gtk\_led\_class\_init**, **gtk\_led\_init** a pour rôle l'initialisation aux valeurs par défaut des données de notre widget, données qui sont propres à chaque instances de notre objet, et qui se trouvent donc dans la structure même du widget.

```
static void gtk_led_load_image (GtkLed * led)
```

```
static void
gtk_led_load_image (GtkLed * led)
{
    led->on_pixbuf = gdk_pixbuf_new_from_file (led->on_image, NULL);
    led->off_pixbuf = gdk_pixbuf_new_from_file (led->off_image, NULL);
    led->width = gdk_pixbuf_get_width (led->on_pixbuf);
    led->height = gdk_pixbuf_get_height (led->on_pixbuf);
}
```

Voici notre fonction personnelle pour le chargement des images qui sont passées en paramètre à la fonction **gtk\_led\_new** ; Nous en profitons également pour récupérer et stocker la largeur et la hauteur de ces images, en prenant comme base la première, la seconde devant être obligatoirement de la même taille. Rien de bien compliqué jusqu'ici !

```
static void gtk_led_size_request (GtkWidget * widget, GtkRequisition * requisition)
```

```
static void
gtk_led_size_request (GtkWidget * widget,
                    GtkRequisition * requisition)
{
```

```
static void gtk_led_size_request (GtkWidget * widget, GtkRequisition * requisition)
{
    g_return_if_fail (widget != NULL);
    g_return_if_fail (GTK_IS_LED (widget));
    g_return_if_fail (requisition != NULL);

    requisition->width = GTK_LED (widget)->width;
    requisition->height = GTK_LED (widget)->height;
}
```

Encore une fois, après des tests de validité sur les arguments de la fonction, nous stockons la taille préférée de notre widget, cette fonction ne sert qu'à cela !

La seule chose à retenir, ici, est que si notre widget pointe sur le type GtkWidget et que nous voulons accéder comme nous le faisons, ici, à des variables, ou pointeurs, contenus dans sa structure, il nous faut obligatoirement faire pointer l'adresse de notre widget sur le type même de notre widget à savoir **GtkLed** !

```
static void gtk_led_size_allocate (GtkWidget * widget, GtkAllocation * allocation)
{
    static void
    gtk_led_size_allocate (GtkWidget * widget,
                          GtkAllocation * allocation)
    {
        g_return_if_fail (widget != NULL);
        g_return_if_fail (GTK_IS_LED (widget));
        g_return_if_fail (allocation != NULL);

        widget->allocation = *allocation;

        if (GTK_WIDGET_REALIZED (widget)) {
            gdk_window_move_resize (
                widget->window,
                allocation->x, allocation->y,
                allocation->width, allocation->height
            );
        }
    }
}
```

Cette fonction est appelée par le conteneur dans lequel se trouve notre widget, une fois que les contraintes qui lui sont imposées sont fixées, ce qui lui permet de redimensionner au mieux notre widget, d'après le résultat obtenu dans le second argument de la fonction.

Nous transmettons donc ces données au widget, puis nous lançons la demande de déplacement, et re-dimensionnement, avec l'appel à **gdk\_window\_move\_resize**. Ici encore une fois, vous n'avez rien à inventer !

```
static void gtk_led_realize (GtkWidget * widget)
{
    static void
    gtk_led_realize (GtkWidget * widget)
    {
        GdkWindowAttr attributes;
        guint attributes_mask;

        g_return_if_fail (widget != NULL);
        g_return_if_fail (GTK_IS_LED (widget));

        GTK_WIDGET_SET_FLAGS (widget, GTK_REALIZED);
    }
}
```

```
static void gtk_led_realize (GtkWidget * widget)
```

```

    attributes.window_type = GDK_WINDOW_CHILD;
    attributes.x = widget->allocation.x;
    attributes.y = widget->allocation.y;
    attributes.width = widget->allocation.width;
    attributes.height = widget->allocation.height;
    attributes.wclass = GDK_INPUT_OUTPUT;
    attributes.event_mask = gtk_widget_get_events (widget) | GDK_EXPOSURE_MASK;

    attributes_mask = GDK_WA_X | GDK_WA_Y;

    widget->window = gdk_window_new (
        gtk_widget_get_parent_window (widget),
        &attributes, attributes_mask
    );

    gdk_window_set_user_data (widget->window, widget);

    widget->style = gtk_style_attach (widget->style, widget->window);
    gtk_style_set_background (widget->style, widget->window, GTK_STATE_NORMAL);
}

```

Voici une partie importante de notre widget, car il s'agit de la fonction qui sert à réaliser celui-ci, attribuer des préférences, des événements et enfin, créer la fenêtre de notre widget. J'ai bien précisé le terme « fenêtre », en parlant du widget, car en GTK+ tout widget est une fenêtre !

La macro **GTK\_WIDGET\_SET\_FLAGS** sert à activer le drapeau qui indique que le widget a été réalisé.

On peut maintenant voir la configuration de la fenêtre du widget, étape que nous faisons en remplissant des champs de la structure **GdkWindowAttr**. Dans cette structure, certains champs sont obligatoires (ceux qui ne possèdent pas de drapeaux), et d'autres ne le sont pas, voici tous les champs de cette structure avec leur drapeau respectif :

Champs	Drapeau	Description
gchar * title	GDK_WA_TITLE	Titre de la fenêtre pour la fenêtre parent. Par défaut, c'est le nom de l'application qui est utilisé.
gint event_mask		Événements étant gérés par le widget.
gint x	GDK_WA_X	Coordonnée X relative à la fenêtre parent. Par défaut cette valeur est mise à 0..
gint y	GDK_WA_Y	Coordonnée Y relative à la fenêtre parent. Par défaut cette valeur est mise à 0.
gint width		Largeur du widget.
gint height		Hauteur du widget.
GdkWindowClass * wclass		GDK_INPUT_OUTPUT est une fenêtre normale. GDK_INPUT_ONLY est une fenêtre invisible mais qui peut tout de même intercepter des événements.
GdkVisual * visual	GDK_WA_VISUAL	Aspect visuel de la fenêtre.
GdkColormap * colormap	GDK_WA_COLORMAP	Table de correspondance des couleurs utilisée pour la fenêtre. Par défaut, c'est la table système qui est utilisée.
GdkWindowType window_type		Voir le tableau suivant pour une

Champs	Drapeau	Description
		description de ces constantes.
GdkCursor * cursor	GDK_WA_CURSOR	Curseur de la fenêtre. Par défaut c'est le curseur de la fenêtre parent qui est utilisé.
gchar * wmclass_name gchar * wmclass_class	GDK_WA_CLASS	Ne pas utiliser !
gboolean override_redirect	GDK_WA_NOREDIR	Si la valeur est positionnée sur TRUE, le gestionnaire de fenêtre n'a plus aucun contrôle sur la fenêtre du widget. Par défaut la valeur est sur FALSE sauf si la fenêtre est du type GDK_WINDOW_TEMP

Constante	Description
GDK_WINDOW_ROOT	Cette fenêtre n'a pas de parent, c'est une fenêtre qui couvre l'écran entier et qui est créée par le système de fenêtrage.
GDK_WINDOW_TOPLEVEL	Type utilisé pour implémenter une fenêtre de type GtkWidget
GDK_WINDOW_CHILD	Type utilisé pour implémenter tout autre type de widget sauf les GtkMenu.
GDK_WINDOW_TEMP	Type utilisé pour implémenter les GtkMenu.
GDK_WINDOW_FOREIGN	Fenêtre non créée par GTK+ .

Dans le champ **event\_mask**, nous récupérons les événements d'un GtkWidget de base, et nous redéfinissons l'événement **expose\_event** de celui-ci, pour pouvoir gérer notre propre affichage du widget.

Étant donné que nous gérons le positionnement du widget, nous devons également déclarer les drapeaux des champs **x** et **y** de la structure, ce que nous faisons, en remplissant la variable **attributes\_mask**. Chaque fois que vous utiliserez un champ qui possède un drapeau, vous devrez obligatoirement en déclarer les drapeaux ainsi !

Nous lançons ensuite la procédure de création de la fenêtre du widget, en appelant la fonction **gdk\_window\_new**.

Dans l'appel à **gdk\_window\_set\_user\_data**, nous spécifions simplement que la donnée à passer dans le premier argument n'est autre que le widget, que nous fournissons en second argument à la fonction.

Nous faisons cela de cette manière simplement, car GTK+ spécifie que chaque widget est une fenêtre de type **GdkWindow**.

L'appel de la fonction **gtk\_style\_attach**, permet de faire analyser le style de la fenêtre (par rapport à toutes les données précédentes), ce qui lui permettra également d'initialiser la fenêtre et si nécessaire, de mettre à jour les modifications que nous avons spécifiées.

Enfin, le dernier appel dans cette fonction sert simplement à initialiser l'affichage du fond de notre widget, toujours d'après les préférences spécifiées plus haut !

```
static gboolean gtk_led_expose (GtkWidget * widget, GdkEventExpose * event)
```

```

static gboolean
gtk_led_expose (GtkWidget * widget,
                GdkEventExpose * event)
{
    g_return_if_fail (widget != NULL);
    g_return_if_fail (GTK_IS_LED (widget));
    g_return_if_fail (event != NULL);

    gtk_led_paint (widget);

    return FALSE;
}

```

Cette fonction est le seul événement que nous redéfinissons pour notre widget. Elle sert à dessiner le widget à l'écran.

Dans notre cas, nous passons par une fonction alternative qui effectuera le travail de dessin, car en fait cet événement n'est appelé que lorsque la fenêtre parent du widget redimensionne, et déplace notre widget.

Où je veux en venir ? Le changement de l'état de la Led se fait par le biais d'un **GtkButton** (dans le cas de notre programme d'exemple) or, lorsque nous cliquerons, la fenêtre parent ne va pas envoyer un événement **expose\_event** à notre widget, et donc nous ne verrons pas le changement de l'état de la Led.

```
static void gtk_led_paint (GtkWidget * widget)
```

```

static void
gtk_led_paint (GtkWidget * widget)
{
    GdkPixbuf * pixbuf = NULL;
    gint center_x = 0;
    gint center_y = 0;

    gdk_window_clear_area (
        widget->window,
        0, 0,
        widget->allocation.width,
        widget->allocation.height
    );

    center_x = (widget->allocation.width / 2) - (GTK_LED (widget)->width / 2);
    center_y = (widget->allocation.height / 2) - (GTK_LED (widget)->height / 2);

    if (GTK_LED (widget)->state)
        pixbuf = GTK_LED (widget)->on_pixbuf;
    else
        pixbuf = GTK_LED (widget)->off_pixbuf;

    gdk_pixbuf_render_to_drawable (
        pixbuf,
        GDK_DRAWABLE (widget->window),
        NULL,
        0, 0, center_x, center_y,
        GTK_LED (widget)->width, GTK_LED (widget)->height,
        GDK_RGB_DITHER_NORMAL, 1, 1
    );
}

```

Dans notre fonction personnelle de dessin du widget, rien de bien compliqué.

Nous commençons par vider l'espace utilisé par le widget, puis nous calculons le centre exact de la fenêtre de notre widget, ce qui nous servira à centrer l'image au milieu du widget, puis viens le rendu de l'image courante directement sur le fond de notre widget, à l'emplacement calculé au préalable !

```
static void gtk_led_destroy (GtkObject * object)
{
    static void
    gtk_led_destroy (GtkObject * object)
    {
        GtkLed * led;
        GtkLedClass * klass;

        g_return_if_fail (object != NULL);
        g_return_if_fail (GTK_IS_LED (object));

        led = GTK_LED (object);

        if (led->on_pixbuf != NULL && led->off_pixbuf != NULL) {
            gdk_pixbuf_unref (led->on_pixbuf);
            gdk_pixbuf_unref (led->off_pixbuf);

            led->on_pixbuf = NULL;
            led->off_pixbuf = NULL;
        }

        klass = gtk_type_class (gtk_widget_get_type ());

        if (GTK_OBJECT_CLASS (klass)->destroy) {
            (* GTK_OBJECT_CLASS (klass)->destroy) (object);
        }
    }
}
```

Dernière fonction ultime : la destruction du widget. Ici rien de bien compliqué, nous commençons par supprimer les **GdkPixbuf**, que nous avons créés pendant la création du widget, puis nous passons la main à la fonction de destruction du widget parent, qui détruira le reste du widget, et videra la mémoire système !

## IV - Notes

Nous avons vu que la création d'un widget peut s'avérer être une tâche assez complexe, si nous ne disposons pas de bonnes connaissances sur le fonctionnement interne d'un widget.

Dans ce cours, nous avons créé un widget, en partant de zéro, mais il peut arriver assez souvent que vous ayez besoin de spécialiser un widget déjà existant, dans ce cas-là, le premier membre de la structure du widget devra être celui dont lequel votre widget devra hériter.

Dans cette pratique de création de widget à partir d'un autre (**GtkButton** par exemple), il y a des règles supplémentaires à respecter comme l'utilisation de la fonction de rappel **gtk\_xxx\_send\_configure** (xxx étant le nom de votre widget), et d'en faire un appel explicite à la fin des fonctions **gtk\_xxx\_realize**, et **gtk\_xxx\_size\_request**.

Voici le code de la fonction **gtk\_xxx\_send\_configure**:

```
static void
gtk_xxx_send_configure (GtkXxx * xxx)
{
    GtkWidget * widget;
    GdkEventConfigure event;

    widget = GTK_WIDGET (xxx);

    event.type = GDK_CONFIGURE;
    event.window = widget->window;
    event.x = widget->allocation.x;
    event.y = widget->allocation.y;
    event.width = widget->allocation.width;
    event.height = widget->allocation.height;

    gtk_widget_event (widget, (GdkEvent *) &event);
}
```

Cette fonction permet de stocker les informations de position, et de taille de votre widget, provoquant ensuite l'exécution de la fonction configurée pour traiter l'événement courant.

Le but de cette fonction est de permettre au widget parent (par exemple si nous héritons d'un **GtkButton**) de gérer le redimensionnement, et repositionnement, de son texte et la réactualisation de l'affichage.

## V - Code source complet

Ici, vous pouvez télécharger le code source complet du widget livré avec un programme d'exemple, ainsi que le Makefile Linux et un projet Code::Blocks version Windows: [GtkLed.zip](#)

## VI - Remerciements

Je tiens à remercier gerald et teuf pour leur aide et conseils dans mon étude sur la réalisation des widgets GTK+, ainsi que Khayyam90 (pour m'avoir aidé un peu à entrevoir l'utilisation des outils de la rédaction), gege2061 (pour m'avoir donné l'idée d'écrire des articles sur le Langage C) et Skyrunner (pour m'avoir donné l'idée de ce widget d'introduction) !

Un grand merci également à matrix788 pour la relecture et correction de ce tutoriel !