

# Gestion des erreurs en C avec la GLib

par [Julian Ibarz \(home\)](#)

Date de publication : 16 avril 2008

Dernière mise à jour : 13 mai 2008

Après une courte introduction sur les *GQuark*, nous verrons comment gérer efficacement les erreurs grâce à la GLib. Ce cours est largement inspiré de la **documentation officielle**.

---

I - Introduction au quark.....	3
I-A - Comment créer un quark ?.....	3
I-B - Fonctions avancées.....	3
II - Gestion des exceptions.....	4
II-A - Comment créer un GError ?.....	5
II-B - Quand utiliser GError ?.....	5
II-C - Comment utiliser GError ?.....	5
II-D - Comment renvoyer l'erreur ?.....	6
II-E - Comment renvoyer les erreurs à l'intérieur d'une fonction ?.....	6
II-F - Fonctions avancées.....	7
II-G - Faire des tests sur un GError.....	9
II-H - Comment créer des domaines facilement ?.....	9
III - Remerciements.....	11
IV - Références.....	12

## I - Introduction au quark

Les *GQuark* étant utilisés pour la gestion des erreurs, voici une courte introduction afin que vous ne soyez pas perdu par la suite.

### I-A - Comment créer un quark ?

Héhé, si c'était aussi simple en vrai, les physiciens seraient contents... Hum, très facile :

```
GQuark g_quark_from_string (const gchar* string);  
GQuark g_quark_from_static_string (const gchar* string);
```

Les deux fonctions font (presque) la même chose. Vous donnez une chaîne de caractères, la GLib regarde si elle ne connaît pas déjà cette chaîne et si oui, elle vous renvoie l'entier correspondant ou alors elle crée une nouvelle association à cette chaîne un entier dont elle veillera bien à ce qu'il ne soit pas déjà utilisé (pour une chaîne un seul entier associé).

*g\_quark\_from\_string* copie la chaîne que vous lui fournissez. Vous pouvez donc la supprimer de la mémoire, la GLib en aura fait une copie et *g\_quark\_from\_static\_string* prend juste l'adresse du pointeur. Veuillez donc à ce que votre chaîne soit statique ou au moins à ce qu'elle ne soit pas supprimée jusqu'à qu'on n'utilise plus cette association.

### I-B - Fonctions avancées

Si vous voulez savoir à quelle chaîne est associé un *GQuark*, utilisez la fonction suivante :

```
G_CONST_RETURN gchar* g_quark_to_string (GQuark quark);
```

Vous donnez le quark et vous recevez une chaîne, c'est tout simple.

Pour savoir si une chaîne de caractère est associée à un *GQuark*, utilisez cette fonction :

```
GQuark g_quark_try_string (const gchar *string);
```

Vous donnez la chaîne est on vous renvoie le *GQuark* si il y a une relation sinon la fonction renvoie 0.

## II - Gestion des exceptions

Le langage C n'ayant pas une gestion des exceptions intégrée dans le langage, la GLib nous en fournit une (et très intéressante...). Ceci implique que l'on ne doit pas utiliser ce système à sa manière mais à la manière de la GLib pour rendre les programmes homogènes entre eux et surtout qu'ils puissent s'envoyer des erreurs sans planter. La GLib utilise ce système. Vous êtes donc venu ici pour comprendre comment sa fonctionne pour recevoir les erreurs que donne la GLib ou pour apprendre à l'utiliser. Désolé mais vous apprendrez tout en même temps ;-). Mais venons-en directement au coeur du problème, la structure *GError* :

```
struct GError
{
    GQuark domain;
    gint code;
    gchar* message;
};
```

*domain* est le domaine de l'erreur (on l'avait pas deviné). Celui-ci obéit à une règle stricte dont vous devez vous plier si vous faites de nouveaux types d'erreurs. La voici :

Le domaine d'erreur est appelé <NAMESPACE>\_<MODULE>\_ERROR. NAMESPACE désigne le préfixe que vous mettez à toutes vos fonctions et structures pour éviter les conflits de nom comme le fait la GLib avec son G\_. Le module est par exemple le module THREAD de la GLib. C'est en quelque sorte une sous-partie logique de votre programme. Si votre programme n'a pas de module, la partie <MODULE> est bien évidemment à supprimer... Voici un exemple : G\_THREAD\_ERROR, FOO\_ERROR.

Le code est un code d'erreur. Lui aussi à des "conventions de nommage" :

Les codes d'erreurs sont dans une énumération appelée <Namespace><Module>Error. Les membres de l'énumération sont nommés comme ceci : <NAMESPACE>\_<MODULE>\_ERROR\_<CODE>

Voici un exemple provenant de la GLib :

```
typedef enum
{
    G_FILE_ERROR_EXIST,
    G_FILE_ERROR_ISDIR,
    G_FILE_ERROR_ACCES,
    G_FILE_ERROR_NAMETOOLONG,
    G_FILE_ERROR_NOENT,
    G_FILE_ERROR_NOTDIR,
    G_FILE_ERROR_NXIO,
    G_FILE_ERROR_NODEV,
    G_FILE_ERROR_ROFS,
    G_FILE_ERROR_TXTBSY,
    G_FILE_ERROR_FAULT,
    G_FILE_ERROR_LOOP,
    G_FILE_ERROR_NOSPC,
    G_FILE_ERROR_NOMEM,
    G_FILE_ERROR_MFILE,
    G_FILE_ERROR_NFILE,
    G_FILE_ERROR_BADF,
    G_FILE_ERROR_INVALID,
    G_FILE_ERROR_PIPE,
    G_FILE_ERROR_AGAIN,
    G_FILE_ERROR_INTR,
    G_FILE_ERROR_IO,
    G_FILE_ERROR_PERM,
    G_FILE_ERROR_FAILED
} GFileError;
```

Si il existe une erreur générique (cela veut dire qu'elle s'applique à toutes les erreurs inconnues ou non encore répertoriées dans l'énumération), vous devez la nommer : <NAMESPACE>\_<MODULE>\_ERROR\_FAILED. Vous ne devez donc pas utiliser le code FAILED pour un autre type d'erreur qu'un type inconnu ou générique...

message est tout simplement la description de l'erreur dans une chaîne de caractère. Attention, ce n'est pas la chaîne associée au *GQuark domain*, pour connaître le domaine en chaîne de caractères, utilisez les fonctions appropriées.

## II-A - Comment créer un GError ?

Deux fonctions :

```
GError* g_error_new (GQuark domain, gint code, const gchar *format, ...);
```

format est en style printf ce qui veut dire que vous pouvez très bien écrire :

```
const gchar* description = "Quelque chose est arrivé";  
GError* error = g_error_new (g_quark_from_string ("PP_MONMODULE_ERROR"),  
PP_MONMODULE_ERROR_FAILED, "Erreur : %s", description);
```

Sinon si c'est une chaîne de caractère classique, vous pouvez utiliser la fonction :

```
GError* g_error_new_literal (GQuark domain, gint code, const gchar *message);
```

Pour détruire :

```
void g_error_free (GError *error);
```

Pour copier :

```
GError* g_error_copy (const GError *error);
```

N'oubliez pas après de supprimer les deux instances...

## II-B - Quand utiliser GError ?

Vous devez utiliser *GError* seulement pour des erreurs lors de l'exécution. Ces erreurs doivent être produites par l'environnement ou l'utilisateur : espace mémoire insuffisant, fichier introuvable, mauvaise saisie, etc. Ces erreurs ne sont pas donc des bugs. Les bugs doivent être corrigés dans le code. Par exemple : essaie de désallocation sur un pointeur de valeur NULL, etc. Pour ces derniers types d'erreur dits erreurs de programmation, vous avez d'autres outils que fournit la GLib : *g\_warning()*, *g\_return\_if\_fail()*, *g\_assert()* et *g\_error()*. L'inverse est aussi vrai : ne pas utiliser ces derniers outils pour des erreurs lors de l'exécution.

## II-C - Comment utiliser GError ?

Si vous faites une bibliothèque, vous devez renvoyer les erreurs de runtime (lors de l'exécution) et leur description grâce à *GError*. Une librairie et en général un ensemble de fonction donc ce sont ces fonctions qui devront renvoyer ces erreurs. En effet, c'est à l'utilisateur de la bibliothèque de décider comment gérer cette erreur : afficher un message d'erreur dans une boîte d'avertissement, arrêter la lecture d'un fichier, etc. Si vous faites une application bien conçue, la partie graphique doit être séparée du reste. Là aussi, vous aurez souvent besoin de renvoyer les erreurs à la partie graphique. C'est en général elle qui traitera l'erreur : affichage d'un message d'avertissement, etc.

## II-D - Comment renvoyer l'erreur ?

Si une fonction est susceptible d'avoir une erreur de runtime, son dernier paramètre doit être `GError** error` :

```
GDir* g_dir_open (const gchar* path, guint flags, GError** error);
```

Pourquoi ce double pointeur ? Mettons-nous à la place de l'utilisateur, cela sera plus facile à comprendre. Voici comment il appellerait cette fonction :

```
GError* err = NULL;
GDir* dir = g_dir_open ("images", 0, &err);
```

Ceci permet à l'utilisateur de ne pas choisir de recevoir d'erreurs :

```
GDir* dir = g_dir_open ("images", 0, NULL);
```

Si l'argument `error` est une valeur égale à `NULL`, aucune erreur ne sera renvoyée. Mais vous pourrez savoir si une erreur c'est produite car `g_dir_open` renvoie `NULL`. C'est à vous de décider si vous voulez juste savoir s'il y a eu erreur ou si vous voulez aussi savoir quel type d'erreur est survenu. Si on renvoie l'adresse de notre pointeur `GError*` :

```
GDir* dir = g_dir_open ("images", 0, &err);
```

le paramètre `error` sera différent de `NULL` (qui je vous le rappelle est une valeur généralement égale à 0) et donc la fonction renverra une erreur :

```
GError* err = NULL;
GDir* dir = g_dir_open ("images", 0, &err);
if (err != NULL)
{
    fprintf (stderr, "Unable to read file: s\n", err->message);
    g_error_free (err);
}
```

Vous voyez que si vous recevez une erreur, c'est à vous de la désallouer avec `g_error_free`.

 **Un pointeur `Gerror*` doit être initialisé à `NULL` ! Sinon on ne sait pas après en faisant le test `if(err!=NULL)` si une erreur s'est réellement produite.**

## II-E - Comment renvoyer les erreurs à l'intérieur d'une fonction ?

```
gint foo_open_file (GError **error)
{
    gint fd;

    fd = open ("file.txt", O_RDONLY);

    if (fd < 0)
    {
        g_set_error (error,
                    FOO_ERROR, /* votre domaine d'erreur */
                    FOO_ERROR_BLAH, /* le code d'erreur */
                    "Impossible d'ouvrir le fichier: %s", /* le message d'erreur en chaîne de caractère */
                    g_strerror (errno));
        return -1;
    }
    else
        return fd;
}
```

Voici une fonction très utile :

```
void g_set_error (GError **err, GQuark domain, gint code, const gchar *format, ...);
```

C'est presque le même prototype que `g_error_new` mais si `err=NULL`, elle ne fait rien et si `error!=NULL` elle alloue un `GError` et met l'adresse dans `*error`. Ceci vous évite de tester si l'utilisateur de votre fonction veut recevoir ou pas une erreur :

```
if (fd < 0)
{
    if (error!=NULL) g_error_new (error,
        FOO_ERROR, /* error domain */
        FOO_ERROR_BLAH, /* error code */
        "Failed to open file: %s", /* error message format string */
        g_strerror (errno));
    return -1;
}
```

## II-F - Fonctions avancées

Imaginez que votre fonction qui renvoie une erreur appelle une fonction qui renvoie une erreur. A première vu on ferait :

```
gboolean ma_fonction_qui_peut_bugger (GError **err)
{
    autre_fonction_qui_peut_bugger (err);

    if (*err != NULL) // Tout se fait implicitement
    {
        return FALSE;
    }
    return TRUE;
}
```

Et voilà, le tour est joué. Mais attention ! Si l'utilisateur de notre fonction ne veut pas recevoir d'erreur et met donc le paramètre `err` à `NULL`, alors taper `*err` est très dangereux ! En effet, `NULL` est égal à 0. Donc le programme va regarder la valeur `x` qui est contenue à l'adresse 0 et va ensuite regarder la valeur contenue à l'adresse `x` (`err` est un double pointeur...). Et la valeur contenue à l'adresse 0 est aléatoire donc on va regarder dans un endroit aléatoire de la mémoire donc on va faire un segfault !

Pour recevoir l'erreur on doit donc faire :

```
gboolean ma_fonction_qui_peut_bugger (GError **error)
{
    GError* tmp_error = NULL;
    autre_fonction_qui_peut_bugger (&tmp_error);
    if (tmp_error != NULL) // Là c'est bon
    {
        return FALSE;
    }
    return TRUE;
}
```

Si celle-ci renvoie une erreur, vous devez la passer à l'utilisateur en utilisant la fonction :

```
void g_propagate_error (GError **dest, GError *src);
```

Cette fonction n'existe que pour vous éviter des tests : Si `dest` est `NULL`, l'adresse `src` est désallouée ; sinon l'adresse `src` est copiée dans `*dest` :

```
gboolean ma_fonction_qui_peut_bugger (GError **err)
{
```

```

GError* tmp_error = NULL;
autre_fonction_qui_peut_bugger (&tmp_error);

if (tmp_error != NULL)
{
    g_propagate_error (err, tmp_error);
    return FALSE;
}
return TRUE;
}
    
```

Si l'utilisateur ne veut pas recevoir d'erreur en mettant *err* à NULL, *propagate* ne fera que désallouer *tmp\_error*. Sinon il y aura copie de *tmp\_error* dans *\*err* et l'utilisateur recevra l'erreur. Utiliser cette fonction revient à écrire :

```

if (err != NULL)
{
    *err = tmp_error;
}
else
{
    g_error_free (tmp_error);
}
return FALSE;
    
```

Maintenant imaginez que votre fonction appelle plusieurs autres fonctions qui peuvent bugger, vous devez tester à chaque fois (c'est important) si une fonction renvoie une erreur. Ce code n'est pas bon :

```

gboolean ma_fonction_qui_peut_bugger (GError **err)
{
    GError* tmp_error = NULL;

    sub_fonction_that_can_fail (&tmp_error);
    other_function_that_can_fail (&tmp_error);

    if (tmp_error != NULL)
    {
        g_propagate_error (err, tmp_error);
        return FALSE;
    }
    return TRUE;
}
    
```

Car si les deux fonctions génèrent des erreurs, elles vont créer deux instances de *Gerror* et on en perdra une instance donc il y aura une fuite de mémoire (si *other\_function\_that\_can\_fail* bugge, elle va changer l'adresse pointée par *tmp\_error* et l'autre ira aux oubliettes : à la fin on ne désallouera qu'une instance sur les deux créés).

Il faut donc coder :

```

gboolean ma_fonction_qui_peut_bugger (GError **error)
{
    GError *tmp_error = NULL;

    sous_fonction_qui_peut_bugger (&tmp_error);
    if (tmp_error != NULL)
    {
        g_propagate_error (error, tmp_error);
        return FALSE;
    }
    autre_fonction_qui_peut_bugger (&tmp_error);
    if (tmp_error != NULL)
    {
        g_propagate_error (error, tmp_error);
        return FALSE;
    }
    return TRUE;
}
    
```

Maintenant si *sous\_fonction\_qui\_peut\_bugger* produit des erreurs mais que celles-ci ne sont pas fatales pour le déroulement de la fonction, celle-ci doit se dérouler normalement :

```
gboolean ma_fonction_qui_peut_bugger (GError **error)
{
    GError *tmp_error = NULL;

    sous_fonction_qui_peut_bugger (&tmp_error);
    if (tmp_error != NULL)
    {
        g_propagate_error (error, tmp_error);
        // On ne quitte pas et on continue
    }
    autre_fonction_qui_peut_bugger (&tmp_error);
    if (tmp_error != NULL)
    {
        g_clear_error (error);
        g_propagate_error (error, tmp_error);
        return FALSE;
    }
    return TRUE;
}
```

Mais que vient faire cette nouvelle fonction *g\_clear\_error* ? Et bien si *sous\_fonction\_qui\_peut\_bugger* produit une erreur, *\*error* contiendra l'adresse d'une instance d'un *GError* (si *error != NULL*). Et si *autre\_fonction\_qui\_peut\_bugger* renvoie une erreur, on efface l'autre erreur et on met à la place la nouvelle (pour éviter les fuites de mémoire dont je vous parlais plus haut). En effet, *g\_clear\_error* efface *\*error* seulement si *error* est différent de *NULL* (ce qui peut être le cas si *sous\_fonction\_qui\_peut\_bugger* ne produit aucune erreur) et met *\*error* à *NULL*. Appeler *g\_clear\_error* revient à écrire :

```
if (error!=NULL)
{
    if(*error!=NULL) g_free_error (*error);
    *error = NULL;
}
```

Voici encore une autre raison pour initialiser un *GError\** à *NULL* : si l'utilisateur de votre fonction n'avait pas initialisé son *GError\** à *NULL*, il aurait fait un code très dangereux :

```
GError* error;
ma_fonction_qui_peut_bugger (&error);
```

En effet, si *sous\_fonction\_qui\_peut\_bugger* n'avait pas produit d'erreur, l'appel de *g\_clear\_error* aurait fait le test *if(\*error!=NULL)* qui aurait été juste ! Et *g\_free\_error* aurait été appelé causant irrémédiablement un segfault ou la suppression d'une de vos données (étant donné que *\*error* aurait été initialisé aléatoirement et donc qu'il pointerait vers un endroit aléatoire de la mémoire...).

## II-G - Faire des tests sur un GError

Pour savoir si un *GError* appartient à un domaine particulier et à un code précis, on utilise la fonction :

```
gboolean g_error_matches (const GError *error, GQuark domain, gint code);
```

Si *error* est dans le domaine *domain* et a le code *code*, la fonction renvoie *TRUE*, sinon elle renvoie *FALSE*.

## II-H - Comment créer des domaines facilement ?

On va faire comme la GLib. Voyons comment elle a fait pour le domaine *G\_FILE\_ERROR*. Elle crée une fonction :

```
g_file_error_quark ()
```

Celle-ci renvoie le domaine sous sa forme de *guint32* (n'oubliez pas qu'un domaine est sous forme d'entier et de chaîne de caractère : c'est un Quark...). Donc elle doit être codée comme cela :

```
GQuark g_file_error_quark()  
{  
    return g_quark_from_string ("G_FILE_ERROR");  
}
```

Mais bon je pense qu'ils font plutôt comme ça :

```
GQuark g_file_error=0; // ça c'est une variable globale...  
  
GQuark g_file_error_quark ()  
{  
    if (g_file_error == 0)  
    {  
        g_file_error = g_quark_from_string("G_FILE_ERROR");  
    }  
  
    return g_file_error;  
}
```

Car appeler *g\_quark\_from\_string* a un coup que l'on ne peut pas négliger... Je ne parle pas du coup de l'appel mais bien de la complexité de cette fonction (rechercher une chaîne dans une liste n'est jamais rapide...).

Et pour le relier à sa forme chaînée, la GLib fait un *define* astucieux :

```
#define G_FILE_ERROR g_file_error_quark ()
```

Ainsi la GLib ne s'occupe pas de créer tous les domaines, c'est l'utilisateur de celle-ci qui le fait sans s'en rendre compte en tapant un simple *G\_FILE\_ERROR* sans se soucier de tout ce que cela va faire. Cela permet aussi de ne créer que les domaines qui sont utilisés...

### III - Remerciements

J'aimerais remercier **Gege2061** sans qui ce cours serait resté un simple fichier texte perdu dans mon disque dur. Je remercie également Anthony Viillard pour sa relecture attentive.

## IV - Références

- Documentation officielle de la GLib sur les **erreurs**
- **FAQ GLib sur les erreurs**