1

Introduction

coucou, ben oui, c'est un peu court, jeune homme.

2

Présentation

1. Qu'est ce que GTK+ ?

GTK+ est une bibliothèque ou plutôt en ensemble de trois bibliothèques de fonctions permettant de construire une interface utilisateur agréable et intuitive. Les auteurs de GTK+ ont voulu qu'elle soit peu gourmand en mémoire et le plus efficace possible, tout en restant suffisamment flexible pour que tout type d'interface puisse être créé.

A l'instar de Xaw ou Motif, GTK+ permet de développer rapidement et facilement une interface graphique composée d'objets appelés *widget*. Un widget est un élément graphique comme un bouton, un label, un menu, une barre de défilement, etc. GTK+ s'occupe de dessiner au mieux ces éléments graphiques et gère la plupart des événements utilisateurs. Ainsi le programmeur peut se concentrer sur la réalisation de son code sans avoir à se soucier de redessiner les widgets. Un développeur utilisant GTK+ crée donc une interface graphique facilement à l'aide d'appels aux bibliothèques de GTK+ et connecte des fonctions à des événements particuliers pouvant intervenir sur tel ou tel élément graphique.

Elle a été développée en C en utilisant une technologie orientée objet avec notamment un mécanisme d'héritage des widgets très puissant.

2. Historique

Dans un premier temps, GTK+ a été développé par Peter Mattis, Spencer Kimball et Josh MacDonald car ils avaient besoin d'une boîte à outils graphique pour l'utilitaire de retouche d'images bien connu : *The Gimp*. The Gimp, à ses débuts, avait été écrit avec l'interface utilisateur Motif, mais comme Motif n'était pas un logiciel libre, la diffusion de The Gimp risquait d'être limitée. C'est pourquoi ils entreprirent la création d'une nouvelle interface graphique qui serait à la fois plus souple, plus puissante, moins complexe à mettre en œuvre et plus adaptée à leur besoin spécifique qui était d'offrir une interface à leur outil de retouche d'images. Gimp a en effet été dévéloppé au départ par Spencer Kimball et Peter Mattis.

Josh MacDonald <jmacd@CS.Berkeley.EDU> s'est joint à eux très tôt pour développer GTK+ du temps où il était encore inclus dans Gimp. Il s'est chargé entre autres de coder la boîte de sélection des fichiers avec le complétement automatique des fichiers. Puis il s'est intéressé à la création d'un widget permettant d'afficher et de saisir du texte.

3. Les 3 couches de GTK+

GTK+ est en fait composé de trois bibliothèques de fonctions que l'on peut envisager comme étant l'une au dessus de l'autre (chacune appelle des fonctions définies dans la précédante) : la glib qui est une bibliothèque générale (general library), le GDK dont les initiales signifient Gimp Drawing Kit, ou kit de dessin de Gimp et le GTK qui signifie Gimp ToolKit, ou boîte à outils pour Gimp.

GTK+ est une dénomination générale qui regroupe ces trois bibliothèques. En fait le + est apparu au moment où Peter Mattis à réécrit l'ensemble des widgets afin d'en faire un ensemble possédant une hiérarchie d'objets. Au même moment, les trois bibliothèques composants GTK+ se sont séparées assez distinctement. On

peut donc dire que le + dénote un changement majeur dans la nature même de cette bibliothèque.

Ce que l'on appelle *application GTK+* est une application qui utilise au moins GTK, et en général, dans une moindre mesure la glib et le GDK.

Le schéma d'utilisation des bibliothèques pour une application GTK+ est représenté sur la figure 1.

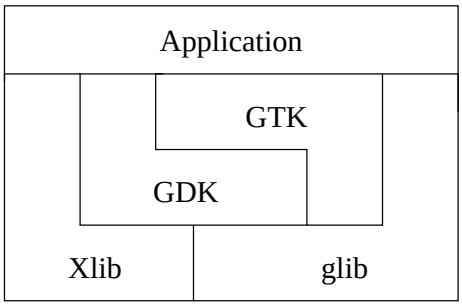


fig. 1

On voit qu'une application GTK+ peut faire appel à chacune des bibliothèques glib, GDK et GTK mais aussi directement à la Xlib[1] pour des appels particuliers. On voit également que GTK n'utilise pas directement la Xlib mais passe par GDK qui est chargé d'encapsuler tous les appels bas niveau au serveur X. Ainsi il suffit donc en théorie de réécrire le GDK pour faire tourner GTK sur une autre plateforme que *X-window system*. Ce n'est malheureusement pas aussi simple dans la pratique.

4. Installation de GTK+

Avant de pouvoir programmer avec GTK+, vous devez l'installer sur votre système. Comme GTK+ est une bibliothèque libre distribuée sous license LGPL, vous devez pouvoir vous la procurer gratuitement, par exemple sur :

```
ftp://ftp.gtk.org/pub/gtk/v1.2/
```

ou, pour utiliser un des miroirs en france :

```
ftp://ftp.minet.net/pub/gimp/gtk/v1.2/
```

Si vous utilisez une distribution récente de Linux, il y a de forte chance pour que votre CD-ROM contiennent déjà les packages correspondant à GTK+.

Si vous utilisez une distribution *Debian* ou compatible, vous devez installez dans l'ordre, les packages suivants :

```
libglib1.2_1.2.x-1_i386.deb
libglib1.2-dev_1.2.x-1_i386.deb
libgtk1.2_1.2.x-1_i386.deb
libgtk1.2-dev_1.2.x-1_i386.deb
```

[1] La Xlib est la bibliothèque standard pour programmer une application qui tourne sous X-Window

où le x correspond en fait au numéro de sous-version. Les deux premiers installent la glib[2] et les fichiers permettant de développer avec la glib, et les deux derniers font la même chose avec GDK et GTK qui sont regroupés dans les mêmes packages. Faites bien attention de bien vérifier qu'il n'existe pas de version de développement de glib ou gtk antérieure à 1.2 sur votre système en lançant la commande :

```
gtk-config --version
```

Cette commande doit vous répondre quelque chose comme `1.2.x`

Si vous utilisez une distribution qui utilise les packages `RPMs`, comme par exemple *RedHat*, ou *Suse*, les packages à installer sont :

```
glib-1.2.x-1.i386.rpm
glib-devel-1.2.x-1.i386.rpm
gtk+-1.2.x-1.i386.rpm
gtk+-devel-1.2.x-1.i386.rpm
```

Si vous préférez[3] installer GTK+ à partir de ses sources, ce que je vous conseille vivement puisque vous y serez amené à un moment où à un autre, voici la marche à suivre.

Commençons par la glib :

- vérifiez d'abord qu'il n'existe pas de version de GTK+ antérieure à la 1.2.0 sur votre machine, avec la commande :
  `ldconfig -p | grep gtk`
- si cette commande renvoie quelque chose, supprimer les packages ou les fichiers correspondants,
- placez vous dans le répertoire `/usr/src/gtk/` (créez-le s'il n'existe pas),
- copiez le fichier `glib-1.2.x.tar.gz` dans ce répertoire,
- *détarrez* ce fichier avec la commande :
  `tar xvfz glib-1.2.x.tar.gz`
- placez vous dans le répertoire glib-1.2.x, et lancez les commandes suivantes :
  `./configure`
  `make`
- devenez *root* si ce n'était pas encore le cas, puis lancez :
  `make install`
- vérifiez ensuite que le fichier `/etc/ld.so.conf` contienne bien la ligne `/usr/local/lib/`,
- lancez la commande
  `ldconfig`
- vérifiez également que la variable d'environnement `PATH` contienne bien le répertoire `/usr/local/bin/` ; si ce n'est pas le cas, tapez les lignes suivantes :
  `export PATH="$PATH:/usr/local/bin/"`
  `echo 'export PATH="$PATH:/usr/local/bin/"' >> /etc/profile`

---

[2] La glib est distribuée dans un package séparé de gtk depuis la version 1.1

[3] ou si vous n'avez pas d'autres choix...

– vérifiez que l'installation de la glib s'est déroulé correctement, le résultat de la commande `glib-config --version` doit être le numéro de version de la glib que vous venez d'installer et la commande :
`glib-config --cflags --libs gmodule gthread` doit répondre :

```
-I/usr/local/lib/glib/include -I/usr/local/include -D_REENTRANT
-L/usr/local/lib -rdynamic -lgmodule -lgthread -lglib -lpthread
-ldl
```

Passons ensuite à l'installation de GDK et GTK :

– retournez dans le répertoire `/usr/src/gtk`,
– copiez le fichier `gtk+-1.2.x.tar.gz` dans ce répertoire,
– lancez les commandes suivantes :

```
tar xvfz gtk+-1.2.x.tar.gz
cd gtk+-1.2.x
./configure
make
make install
ldconfig
```

– testez si tout a bien fonctionné en lançant le programme `./testgtk` situé dans le répertoire `gtk`. Ce programme présente une grande partie des possibilités et des widgets de GTK+.

Si tout s'est bien passé, vous voici donc en mesure de commencer vos propres programmes GTK+.

3

La Glib

1. Les intérêts d'une telle bibliothèque

La glib, ou 'bibliothèque générale', a été créée dans le but de simplifier la vie des programmeurs en C. Elle prend en charge la gestion de beaucoup de petites choses souvent pénibles comme la mémoire, les listes, les chaînes de caractères. Ce sont des choses que tout programmeur en C a déjà fait et doit refaire sans arrêt. Ces fonctions sont donc là pour éviter d'avoir à réinventer l'eau chaude à chaque nouvelle application.

De plus, la glib redéfinit les types les plus courants de manière à rendre une application plus portable, que la plateforme soit 16-bit, 32-bit, big-endian ou little-endian.

La glib n'a rien de graphique et n'est pas vraiment liée à GTK+. Elle peut donc être utilisée dans pratiquement tous les programmes développés en C. Et comme à l'usage, l'ensemble des fonctions qu'elle propose est très pratique, je ne peux que vous recommander d'en user et d'en abuser dans tous vos développements.

2. Les Makefiles et glib-config

Pour compiler un programme utilisant la glib, ses concepteurs ont pensé à inclure un petit utilitaire nommé 'glib-config' qui permet de connaître quelles doivent être les options à passer au compilateur pour compiler une application utilisant la glib.

Ainsi, on peut prendre connaissance rapidement de la version installée de la glib sur une machine en faisant :

```
coruscant:~$ glib-config --version
1.2.3
```

Ce qui signifie que, sur ma machine, la version actuellement installée est la 1.2.3. Il est possible aussi de savoir quelles sont les options qu'il faut passer au compilateur pour compiler un fichier C qui utilise la glib :

```
coruscant:~$ glib-config --cflags
-I/usr/local/lib/glib/include -I/usr/local/include
```

Ce qui signifie que l'on peut compiler notre fichier en faisant :

```
cc -c fichier.c -o fichier.o -I/usr/local/lib/glib/include
                             -I/usr/local/include
```

En langage clair, cela signifie au compilateur qu'il doit chercher les fichiers *include* dans les répertoires /usr/local/lib/glib/include et /usr/local/include en plus de ceux dans lesquels il cherche habituellement.

Concrètement, on utilisera plutôt

```
cc -c fichier.c -o fichier.o ‘glib-config --cflags‘
```

ou

```
cc -c fichier.c -o fichier.o $(glib-config --cflags)
```

De la même façon, on peut connaître quelles sont les options que l'on doit passer à l'éditeur de liens avec :

```
coruscant:~$ glib-config --libs
-L/usr/local/lib -lglib
```

Ce qui signifie que l'éditeur de liens doit rajouter /usr/local/lib à la liste des répertoires dans lesquels il cherche des bibliothèques et qu'il doit lier notre programme à glib.

Là aussi, concrètement, nous utiliserons

```
cc fichier.o -o fichier $(glib-config --libs)
```

pour l'édition des liens de notre programe.

La glib peut optionnellement être compilée avec un support pour les modules chargeables et/ou un support pour les threads. Si notre programme utilise une des ces extensions, il est fort probable que nous devions inclure des bibliothèques supplémentaires. C'est pourquoi il est possible d'appeler le programme `glib-config` différemment :

```
coruscant:~$ glib-config --libs gthread
-L/usr/local/lib -lgthread -lglib -lpthread
coruscant:~$ glib-config --libs gmodule
-L/usr/local/lib -rdynamic -lgmodule -lglib -ldl
coruscant:~$ glib-config --libs gthread gmodule
-L/usr/local/lib -rdynamic -lgmodule -lgthread
-lglib -lpthread -ldl
```

Si vous préférez utiliser un Makefile (ce qui est fortement conseillé), en voici un exemple typique qui fonctionne dans 99% des cas :

```
GLIBS = ‘glib-config --libs gthread gmodule‘
GFLAGS = ‘glib-config --cflags‘
fichier: fichier.o
  cc fichier.o -o fichier $(GLIBS)

fichier.o: fichier.c
  cc fichier.c -c -o fichier.o $(GFLAGS)
```

3. Les types de la glib

Pour une plus grande portabilité, et une souplesse d'utilisation accrue, la glib redéfinit des types de base du langage C. Les types introduits par cette bibliothèque sont reconnaissables par leur 'g' initial. Ces redéfinitions peuvent être globalement réparties en 4 groupes :

1. les types entiers qui ont une taille fixe sur toutes les plateformes : `gint8`, `guint8`, `gint16`, `guint16`, `gint32`, `guint32`, `gint64`, et `guint64`.

2. les types qui sont plus faciles à utiliser que les types de base du langage C : `gpointer`, `gconstpointer`, `guchar`, `guint`, `gushort` et `gulong`.
3. les types qui correspondent exactement aux types de base du langage C et qui ne sont là que pour rendre la glib exhaustive : `gchar`, `gint`, `gshort`, `glong`, `gfloat` et `gdouble`.
4. les types qui n'ont pas de correspondance directe avec les types standards du C : `gboolean`, `gsize`, `gssize`.

Il existe bien sûr d'autres types utilisés par la glib, par exemple pour la gestion des listes chaînées, mais les types présentés ici sont les types simples (par opposition aux structures) définis par la glib et qui ne sont pas liés à l'utilisation de tel ou tel groupe de fonctions de la glib.

3.1. Les types entiers de taille fixe

L'utilisation de ces types peut être très utile lorsque l'on veut être sûr qu'une variable a toujours la même taille (en nombre de bits) quelle que soit la platforme sur laquelle notre application est compilée. Ainsi les types `gint8`, `gint16`, `gint32` et `gint64` permettent de déclarer des entiers signés respectivement de 8, 16, 32 et 64 bits de longs, sans avoir à ce soucier de l'architecture. De la même façon, `guint8`, `guint16`, `guint32` et `guint64` sont les types correspondant à des entiers non signés de 8, 16, 32 ou 64 bits. Notez que les types `gint64` et `guint64` ne sont pas toujours disponibles car ils présupposent que le compilateur est capable de gérer des entiers de 64 bits. Ceci peut être testé à l'aide de la définition `G_HAVE_GINT64`. Par exemple, on peut utiliser ce bout de code :

```
#ifdef G_HAVE_GINT64
  gint64 variable64;
#else
  gint32 variable32[2];
#endif
```

Le compilateur gcc (et sûrement d'autres) définit le type `long long int` pour les entiers sur 64 bits qui est utilisable même sur des architectures 32-bit.

3.2. Les types de confort

Ces types ne sont pas des nouveautés en eux-mêmes, mais ils sont bien pratiques, soit parce qu'ils sont plus courts que leurs équivalents en C standard soit parce que leur nom est plus significatif. Ainsi `gpointer` est un synonyme pour `void *`, c'est à dire qu'il définit un pointeur sur un type quelconque. De la même façon, `gconstpointer` qui est équivalent à `const void *` définit un pointeur sur une constante. Et les types `guchar`, `guint`, `gushort` et `gulong` sont simplement des abréviations de `unsigned char`, `unsigned int`, `unsigned short int` et `unsigned long int` mais plus court à taper, dans la plus pure tradition de la loi du moindre effort des informaticiens.

3.3. Les types qui ne sont là que pour la cohérence de l'ensemble

La glib définit aussi quelques types qui sont rigoureusement équivalents à certains types du langage C. Ils ne différent de leur équivalents que par la lettre 'g' initiale. Ils sont là pour que l'on ait pas à réfléchir pour savoir si l'on doit utiliser `int` ou `gint`. La réponse est toujours que la version avec un 'g' initial existe et qu'il vaut mieux l'utiliser. Ainsi pour un entier non signé, on utilisera `guint` et pour un entier signé, on utilisera `gint`. Bien entendu, rien ne vous oblige à utiliser ces types mais ils sont bien pratiques pour la cohérence des déclarations de variables et ne rajoutent qu'une seule lettre de plus... Ces types sont `gchar`, `gint`, `gshort`, `glong`, `gfloat` et `gdouble`.

3.4. Les types nouveaux de la glib

Ces types n'ont pas de correspondance directe avec les types standards du C et s'emploient dans des contextes assez particuliers. Ainsi `gboolean` est en fait un entier qui normalement ne doit contenir que les valeurs `TRUE` ou `FALSE`. Ce type doit donc être réservé au stockage de valeur de conditions. `gsize` et `gssize` sont respectivement des `guint32` et `gint32` qui doivent être utilisés pour stocker des tailles de structures par exemple :

```
gssize taille = sizeof(struct MaStructure);
```

De plus, la glib propose des constantes qui permetent de connaître quelles sont les valeurs minis et maxis que l'on peut placer dans chaque type. Ansi un `gint` est compris entre `G_MININT` et `G_MAXINT`, un `gshort` est compris entre `G_MINSHORT` et `G_MAXSHORT`, un `long` est compris entre `G_MINLONG` et `G_MAXLONG`, un `gfloat` est compris entre `G_MINFLOAT` et `G_MAXFLOAT`, et un `gdouble` est compris entre `G_MINDOUBLE` et `G_MAXDOUBLE`.

4. Les fonctions de gestion de la mémoire

La gestion des blocs de mémoire est le cauchemar de beaucoup de programmeurs. C'est pourquoi la glib propose une série de fonctions pour faciliter tout cela.

Tout d'abord, la glib propose quatre macros permettant de transformer de façon portable un pointeur en entier et réciproquement. Ces macros fonctionnent de la même manière que le système soit 16-bit, 32-bit ou 64-bit et qu'il soit little-endian ou big-endian.

Ainsi `gpointer GINT_TO_POINTER(entier)` et `gpointer GUINT_TO_POINTER(entier)` transforment respectivement un entier signé et un entier non signé en un pointeur. Réciproquement, `gint GPOINTER_TO_INT(pointeur)` et `guint GPOINTER_TO_UINT(pointeur)` réalisent la transformation inverse. Ces quatre macros sont utiles pour transmettre une valeur entière là où un pointeur générique est attendu, ce qui sera notamment le cas avec les fonctions de rappels de GTK.

La glib propose également des équivalents aux fonctions standards d'allocation de mémoire. On a donc :

– `gpointer g_malloc(gulong taille);` qui alloue un bloc de mémoire de 'taille' octets de long et renvoie un pointeur sur le début de ce bloc.
– `gpointer g_malloc0(gulong size);` qui réalize la même chose, mais qui initialize le bloc a 0.
– `gpointer g_realloc(gpointer mem, gulong size);` qui permet de changer la taille d'un bloc préalablement alloué.
– `void g_free(gpointer mem);` qui libère un espace mémoire préalablement alloué.

Ces quatre fonctions sont généralement identiques à leurs correspondants dans la bibliothèque standard C qui sont `malloc()`, `malloc0()`, `realloc()` et `free()`. Cependant, si la glib a été compilée avec les options de déboggage, de vérification et d'optimisation de la mémoire, chacune de ces fonctions fait également des vérifications de validités et collectent des statistiques.

Ces statistiques peuvent être affichées par un appel à la fonction `void g_mem_profile(void);`. Toujours afin de vérifier les éventuels problèmes de mémoire, la fonction `void g_mem_check(gpointer mem);` permet de vérifier si le bloc mémoire commençant en 'mem' a déjà été libéré ou non.

Les habitués du C++ pourront utiliser les macros suivantes :

– `g_new(type, nombre)` qui alloue un bloc suffisant pour contenir 'nombre' données de type 'type' et qui renvoie un pointeur de type 'type *' – `g_new0(type, nombre)` qui réalise la même chose en initialisant la zone à 0. – `g_renew(type, pointeur, nombre)` qui permet de changer la taille d'un bloc à la manière d'un `g_realloc()`.

La glib propose un autre mécanisme d'allocation de mémoire. Il s'agit de l'allocation par morceau (chunk en anglais). L'idée est qu'il peut être très efficace d'allouer des blocs qui ont tous la même taille. L'avantage de cette méthode est qu'elle permet une allocation/désallocation de la mémoire très rapide. L'inconvénient est que tous les blocs du morceau doivent avoir la même taille.

Un nouveau descripteur de morceau doit être alloué avant de pouvoir allouer des blocs dans le morceau. Cela se fait à l'aide de la fonction :

```
GMemChunk* g_mem_chunk_new(gchar  *Nom,
                           gint    taille_bloc,
                           gulong  nombre_de_blocs,
                           gint    type);
```

où `Nom` est une chaîne de caractères contenant le nom du morceau que l'on crée, `taille_bloc` est la taille d'un bloc de ce morceau, `nombre_de_blocs` est le nombre maximal de blocs que notre morceau devra pouvoir être capable de contenir, enfin `type` est un indicateur qui vaut soit `G_ALLOC_ONLY`, dans quel cas les blocs alloués dans le morceau ne pourront pas être désalloué individuellement, soit `G_ALLOC_AND_FREE` si l'on veut pouvoir désallouer les blocs un par un. La valeur renvoyée est un pointeur sur une structure `GMemChunk`. La définition exacte de cette structure n'est absolument pas importante pour utiliser le mécanisme d'allocation par morceau.

Le morceau tout entier (et les blocs qui le composent) peut être libéré grâce à la fonction :

```
void g_mem_chunk_destroy(GMemChunk *morceau_memoire);
```

où `morceau_memoire` est un pointeur sur le descripteur du morceau que l'on veut désallouer.

À l'intérieur d'un morceau, on peut bien entendu allouer ou libérer des blocs mémoires un par un à l'aide des fonctions :

```
gpointer g_mem_chunk_alloc(GMemChunk *morceau_memoire);
gpointer g_mem_chunk_alloc0(GMemChunk *morceau_memoire);
void g_mem_chunk_free(GMemChunk *morceau_memoire,
                      gpointer pointeur);
g_chunk_new(type, morceau_memoire)
g_chunk_new0(type, morceau_memoire)
```

dont le rôle doit paraître évident si on les compare aux fonctions d'allocation de mémoire classiques. Les deux seules différences sont d'une part la nécessité de rappeler `morceau_memoire` à chaque appel pour signifier dans quel morceau doit être faite l'allocation, et d'autre part l'impossibilité de fixer la taille du blocs puisqu'elle a été fixée une fois pour toute lors de la création du morceau.

On peut libérer de la place mémoire en effaçant les blocs qui ne sont plus utilisés dans le morceau en appelant :

```
void g_mem_chunk_clean(GMemChunk *morceau_memoire);
```

Ceci réarrange les blocs à l'intérieur du morceau et libère réellement la mémoire occupé par des blocs qui avaient été désalloués par un appel à `g_mem_chunk_free()`[1].

De la même façon, `void g_blow_chunks(void);` permet cette optimisation de la mémoire pour tous les morceaux actuellement utilisés par notre programme.

Il est possible de libérer tous les blocs d'un morceau sans pour autant détruire le morceau grâce à :

```
void g_mem_chunk_reset(GMemChunk *morceau_memoire);
```

Lors de l'utilisation des morceaux de mémoire, les deux fonctions

```
void g_mem_chunk_print(GMemChunk *morceau_memoire);
void g_mem_chunk_info(void);
```

permettent d'afficher beaucoup d'informations utiles au déboggage relatives soit à un seul morceau soit à tous ceux actuellement utilisés.

---

[1] `g_mem_chunk_free()` ne libère pas directement la mémoire afin de gagner du temps. Son rôle se limite donc à marquer un bloc comme étant effacé, ainsi, lors d'une nouvelle allocation de bloc, la glib n'aura pas besoin de refaire une réelle allocation mais pourra simplement marquer ce bloc comme étant à nouveau utilisé et renvoyer l'adresse de se bloc. Ce système permet d'accélérer considérablement le processus d'allocation et de désallocation.

5. Les fonctions de gestion des tableaux

Les morceaux de mémoire sont utilisés par la glib pour proposer une autre fonctionalité : les tableaux dynamiques. Il s'agit de tableaux dont la taille augmente ou rétrécit au fur et à mesure que des éléments sont ajoutés ou retirés. Ils combinent la facilité d'utilisation des tableaux et la souplesse des listes chaînées. On crée facilement un tel tableau avec la fonction :

```
GArray* g_array_new(gboolean zero_terminal,
                    gboolean mise_a_zero,
                    guint taille_element);
```

où `zero_terminal` est un drapeau qui vaut TRUE si on désire que le tableau soit terminé par un élément nul. `mise_a_zero` est aussi un drapeau qui, s'il est mis à TRUE, demande à la glib que les éléments de ce tableau soient automatiquement mis à zéro lors de leur création. `taille_element` représente la taille d'un élément du tableau. La valeur de retour est un pointeur sur une structure `GArray`. Ce pointeur nous servira de référence lors des autres appels pour signifier à la glib de quel tableau l'on parle.

Une fois le tableau créé, on peut ajouter un élément à la fin du tableau en appelant la fonction :

```
GArray *g_array_append_val(GArray *tableau, valeur);
```

où `tableau` est le pointeur sur une structure `GArray` renvoyé par un appel à `g_array_new()` et `valeur` est la valeur de l'élément que l'on veut ajouter au tableau. Vous remarquerez que `valeur` n'a pas de type particulier car un tableau dynamique peut contenir des valeurs de n'importe quel type. Il est souvent désirable de pouvoir insérer plusieurs valeurs à la fois dans un tableau dynamique (notamment pour transformer un tableau statique en tableau dynamique). La fonction

```
GArray *g_array_append_vals(GArray *tableau,
                            gconstpointer elements,
                            guint nombre_d_elements);
```

est faite pour cela. `tableau` est toujours un pointeur sur une structure `GArray`, `elements` est un pointeur sur le premier élément que l'on veut insérer et `nombre_d_elements` est ... le nombre d'éléments que l'on veut insérer. Les éléments doivent, bien entendu, être consécutifs en mémoire, ce qui est le cas s'ils proviennent d'un tableau statique du C standard.

De la même façon, on peut ajouter des éléments au début d'un tableau avec les fonctions :

```
GArray *g_array_prepend_val(GArray *tableau, valeur);
GArray *g_array_prepend_vals(GArray *tableau,
                             gconstpointer elements,
                             guint nombre_d_elements);
```

Enfin, on peut insérer des éléments au milieu d'un tableau dynamique grâce à :

```
GArray *g_array_insert_val(GArray *tableau,
                           guint index, valeur);
GArray *g_array_insert_vals(GArray *tableau,
                            guint index,
                            gconstpointer elements,
                            guint nombre_d_elements);
```

où `index` est simplement l'index du premier élément qui doit être inséré.

On peut bien sûr accéder à un élément particulier grâce à la macro

```
g_array_index(tableau, type, index)
```

qui renvoie l'élément numéro `index` du tableau `tableau`. Le paramètre `type` indique le type des éléments de ce tableau.

Les tableaux dynamiques peuvent être désalloués avec :

```
void  g_array_free(GArray *tableau, gboolean libere_element);
```

où `libere_element` est un drapeau qui est à `TRUE` si l'on veut que l'espace mémoire occupé par les éléments soit aussi être libéré lors de la désallocation du tableau.

Les éléments du tableau peuvent également être enlevés un par un avec :

```
GArray* g_array_remove_index(GArray *tableau, guint index);
```

où `index` est l'index dans le tableau de l'élément que l'on souhaite supprimer. Notez que l'index de tous les éléments suivants dans le tableau est diminué de 1 afin qu'il n'y ait pas de 'trou' dans le tableau.

Enfin, on peut fixer la taille d'un tableau avec :

```
GArray* g_array_set_size(GArray *tableau, guint taille);
```

Sachez aussi qu'il existe des fonctions spécialisées dans la gestion des tableaux dynamiques de pointeurs (`GPtrArray *`) et d'entiers sur 8 bits (`GByteArray *`[2]) et d'entiers sur 8 bits. Ces fonctions ne seront pas traitées dans le cadre de ce livre. Leur principal intérêt est d'effectuer des vérifications de type.

6. Les fonctions de gestion des chaînes de caractères

La glib propose une foison de fonctions permettant de traiter des chaînes de caractères, c'est-à-dire des tableaux de caractères. Il n'est pas facile de trouver un classement général pour ces fonctions, mais on peut grossièrement les répartir en quatre groupes : celles qui transforment une chaîne de caractères, celles qui créent une chaîne de caractères, celles qui traitent des tableaux de chaînes de caractères, et celles qui utilisent le type `GString`.

6.1. Les fonctions qui transforment des chaînes

Ces fonctions s'appliquent toutes à des chaînes de caractères préexistantes. Ainsi,

---

[2] Les plus curieux pourront trouver les prototypes des fonctions correspondantes aux GPtrArrays et GByteArrays dans le fichier glib.h

```
gchar *g_strdelimit(gchar *chaine,
                    const gchar *delimiteurs,
                    gchar nouveau_delimiteur);
```

vérifie pour caractère de la chaîne `chaine` s'il fait partie de la chaîne `delimiteurs` et, si c'est le cas, le remplace par le caractère `nouveau_delimiteur`.

Par exemple, à l'issue de ces deux lignes

```
gchar chaine = "Bonjour0à1tous2!3Comment4allez5vous6?";

g_strdelimit(chaine, "0123456", ' ');
```

la chaîne `chaine` contiendra : `"Bonjour à tous ! Comment allez vous ?"`. Cette fonction était à l'origine surtout destinée à uniformiser des délimiteurs de champs pour une utilisation dans une base de données, d'où son nom.

De plus, les fonctions

```
void g_strdown(gchar *chaine);
```

et

```
void g_strup(gchar *chaine);
```

permettent de transformer une chaîne respectivement en minuscule et en majuscule.

```
void g_strreverse(gchar *chaine);
```

quant à elle permet d'inverser l'ordre des caractères dans une chaîne de caractères.

Lorsque l'on veut supprimer des espaces aux extrémités d'une chaîne, les fonctions :

- `gchar *g_strchug(gchar *chaine);` qui supprime les espaces initiaux d'une chaîne,
- `gchar *g_strchomp(gchar *chaine);` qui supprime les espaces à la fin d'une chaîne et
- `gchar *g_strstrip(gchar *chaine);` qui supprime les espaces au début et à la fin d'une chaîne,

peuvent être d'un grand secours.

Lorsque l'on manipule des chaînes de caractères, il arrive souvent que l'on souhaite en comparer deux afin de savoir laquelle est avant l'autre dans l'ordre alphabétique. Malheureusement, les fonctions de la bibliothèque C standard considèrent toujours qu'une lettre en majuscule est toujours avant une lettre en minuscule, quelles qu'elles soient. Ce comportement n'est pas toujours celui souhaité. C'est pourquoi la glib met à notre disposition les deux fonctions :

```
gint g_strcasecmp(const gchar *chaine1,
                  const gchar *chaine2);
gint g_strncasecmp(const gchar *chaine1,
                   const gchar *chaine2,
                   guint n);
```

Elles sont identiques aux fonctions standards `strcmp() et strncmp()` sauf qu'elles ignorent la casse[3] des caractères des chaînes qu'elles comparent.

6.2. Les fonctions qui créent une chaîne de caractères

La plupart de ces fonctions alloue un espace mémoire qu'il faudra penser à libérer lorsqu'on n'en aura plus besoin.

Les deux premières de ces fonctions permettent de transformer une valeur d'erreur ou de signal en une chaîne de caractères explicite.

```
gchar *g_strerror(gint numero_erreur);
gchar *g_strsignal(gint numero_signal);
```

La première renvoie une chaîne décrivant l'erreur numéro `numero_erreur` qui provient en général de la variable `errno` que beaucoup de fonctions de la bibliothèque standard C positionnent. La seconde réalise la même chose avec un numéro de signal. Les chaînes renvoyées par ces deux fonctions *ne doivent pas* être libérées de la mémoire après utilisation.

La duplication d'une chaîne de caractères peut se faire grâce à :

```
gchar *g_strdup(const gchar *chaine);
gchar *g_strndup(const gchar *chaine,
                 guint nombre_max_caracteres);
gpointer g_memdup(gconstpointer chaine,
                  guint nombre_caracteres);
```

qui sont globalement équivalentes aux fonctions standards `strdup()`, `strndup()` et `memdup()`, si ce n'est que ces chaînes doivent être libérée avec `g_free()`. Rappel : `strdup()` crée une copie conforme d'une chaîne, `strndup()` limite la copie à `nombre_max_caracteres` même si la chaîne est plus longue. Et `memdup()` copie exactement `nombre_caracteres` même si la chaîne est plus courte ou plus longue.

```
gchar *g_strnfill(guint n, gchar caractere);
```

crée une chaîne de caractères comportant exactement `n` fois le caractère `caractere`. Elle devra être libérée par un `g_free()` lorsqu'on n'en aura plus besoin.

La fonction suivante de notre énumération est certainement la plus utile.

```
gchar *g_strdup_printf(const gchar *format, ...);
```

calcule la taille nécessaire pour contenir la chaîne de caractères générée par un

```
printf(format, ...)
```

, crée une chaîne de la bonne longueur et copie le résultat du `printf` dedans. Cette fonction peut être vu comme une version plus sûre d'un `sprintf()`, car la taille de la chaîne est toujours calculée de façon à pouvoir contenir le résultat. Par exemple,

---

[3] majuscule ou minuscule

```
chaine = g_strdup_printf("Bonjour %s 50/33 = %g\n",
                         "à tous !", 50.0/33.0);
```

crée une chaîne de 33 caractères contenant : `"Bonjour à tous ! 50/33 = -1.51515\n"`.

De manière réciproque, on peut transformer une chaîne en nombre avec :

```
gdouble g_strtod(const gchar *chaine,
                 gchar **nouvelle_position);
```

où `chaine` est la chaîne qui est sensée contenir un nombre. La valeur renvoyée est un gdouble qui contient le nombre après conversion, ou 0. Si `nouvelle_position` est non nul, elle pointera sur le premier caractère de la chaîne après ceux qui ont été pris en compte pour la conversion. Cela permet de convertir plusieurs nombres qui serait dans la même chaîne.

Vous connaîssez tous `strcat()` qui permet de concaténer deux chaînes. Et bien,

```
gchar *g_strconcat(const gchar *chaine1, ...);
```

en est une version améliorée. En effet, `g_strconcat` permet de concaténer un nombre quelconque de chaînes passées en paramètre. La dernière chaîne doit être `NULL` afin que la fonction sache où s'arrêter. De plus cette fonction alloue elle-même la mémoire nécessaire pour contenir la chaîne résultante. Celle-ci doit donc être libérée par un `g_free()` par la suite.

6.3. Les fonctions qui manipulent des tableaux de chaînes de caractères

Les habitués du langage Perl seront heureux d'apprendre que la glib propose des fonctions permettant d'émuler le comportement de `split` et `join`. La fonction

```
gchar **g_strsplit(const gchar *chaine,
                   const gchar *delimiteur,
                   gint  nombre_de_champ);
```

est très proche dans son comportement à la fonction `split` de Perl. Elle permet de séparer la chaîne `chaine` contenant des champs séparés par le délimiteur `delimiteur` en un tableau de chaînes de caractères contenant chacune un champ. Le nombre maximal de champs peut être fixé avec le paramètre `nombre_de_champ`. Si ce paramètre vaut 0, le nombre de champs n'est pas limité.

Voici un exemple pour éclaircir les choses :

```
#include <glib.h>

void main(void)
{
  gchar **champs;
  gchar *chaine = "pomme:salade:tomate:";
  int i;
```

```
  champs = g_strsplit(chaine, ":", 0);
  for (i=0; champs[i] != NULL ; i++)
    printf("champ n°%d : '%s'\\n", i, champs[i]);
}
```

L'exécution de ce programme affiche :

```
champ n°0 : 'pomme'
champ n°1 : 'salade'
champ n°2 : 'tomate'
champ n°3 : ''
```

La mémoire utilisée par un tableau créé avec `g_strsplit` doit être libérée avec :

```
void g_strfreev(gchar **tableau_chaines);
```

De la même façon, la fonction `join` de Perl peut être émulée par l'une des deux fonctions suivantes :

```
gchar *g_strjoinv(const gchar *delimiteur,
                  gchar **tableau_chaines);
gchar *g_strjoin(const gchar *delimiteur, ...);
```

Le rôle de ces fonctions est de réaliser exactement l'inverse de `g_strsplit`. C'est-à-dire qu'elles concatènent un certain nombre de chaînes en insérant `delimiteur` entre chaque. La version `g_strjoinv` attend un tableau de chaînes de caractère du genre de ceux créés par `g_strsplit` en deuxième paramètre. Il doit donc être terminé par un `NULL`. La version `g_strjoin` attend les champs directement parmi ses paramètres. Là aussi, le dernier paramètre doit être `NULL`.

Exemple :

```
#include <glib.h>

void main(void)
{
  gchar **champs;
  gchar *chaine = "pomme:salade:tomate";
  int i;

  champs = g_strsplit(chaine, ":", 0);
  printf("recette = %s\n", g_strjoinv(" + ", champs));
  g_strfreev(champs);
}
```

affichera :

```
recette = pomme + salade + tomate
```

6.4. Les GStrings

La glib offre également un type `GString` qui peut être utilisé pour simplifier la gestion des chaînes de caractères à l'aide de toutes ces fonctions. Les GStrings peuvent être transtypés sans problème en `gchar *`. Attention, l'inverse n'est pas vrai... L'avantage des GStrings est qu'elles adaptent automatiquement leurs tailles suivant les affectations qu'elles subissent.

Les GStrings peuvent être crées par deux fonctions différentes :

```
GString *g_string_new(const gchar *chaine);
GString *g_string_sized_new(guint taille_par_defaut);
```

La première crée une GString initialisée à la valeur `chaine`, la seconde crée une chaîne vide avec une taille initiale.

Une structure GString doit être libérée par :

```
void g_string_free(GString *chaine, gint libere_donnees);
```

où le paramètre `libere_donnees` doit être à `TRUE` si l'on désire que la chaîne de caractères associée soit libérée en même temps (ce qui est généralement le cas).

On peut bien entendu changer le contenu d'une GString avec :

```
GString* g_string_assign(GString *chaine,
                         const gchar *nouvelle_valeur);
```

ou en changer la taille avec :

```
GString* g_string_truncate(GString *chaine,
                           gint nouvelle_taille);
```

Des caractères ou des chaînes de caractères peuvent être ajoutés à des GStrings par les fonctions :

```
GString* g_string_append(GString *chaine,
                         const gchar *chaine_ajoutee);
GString* g_string_append_c(GString *chaine,
                           gchar caractere);
GString* g_string_prepend(GString *chaine,
                          const gchar *chaine_ajoutee);
GString* g_string_prepend_c(GString *chaine,
                            gchar caractere);
GString* g_string_insert(GString *chaine, gint pos,
                         const gchar *chaine_ajoutee);
GString* g_string_insert_c(GString *chaine,
                           gint pos,
                           gchar caractere);
```

dont le rôle est exactement le même que celui des fonctions `g_array_*` correspondantes.

On trouve de même une fonction pour effacer une partie d'une GString :

```
GString* g_string_erase(GString *chaine,
                        gint debut,
                        gint longueur);
```

où `debut` est le premier caractère que l'on veut supprimer et `longueur` le nombre de caractères que l'on veut supprimer. Par exemple, si une GString `chaine` contient initialement `"0123456789"`, après un appel à `g_string(chaine, 3, 2)`, `chaine` contiendra `"01256789"`.

On retrouve les fonctions de mise en majuscule ou minuscule :

```
GString* g_string_down(GString *chaine);
GString* g_string_up(GString *chaine);
```

qui ne sont que des "wrappers" autour des fonctions correspondantes pour les `gchar *`.

Enfin, les fonctions :

```
void g_string_sprintf(GString *chaine,
                      const gchar *format, ...);
void g_string_sprintfa(GString *chaine,
                       const gchar *format, ...);
```

sont à rapprocher de la fonction `g_strdup_printf()`. La première remplace le contenu de la `chaine` par la chaîne calculée par `g_strdup_printf` alors que la seconde l'ajoute à la fin.

7. Les fonctions de gestion des listes chaînées

7.1. Les listes doublement chaînées

La glib propose des fonctions permettant de manier facilement des listes doublement chaînées. Ces listes sont des structures de données reliées entre elles par deux pointeurs, l'un pointant sur la structure précédente et l'autre pointant sur la structure suivante (voir fig. 1).

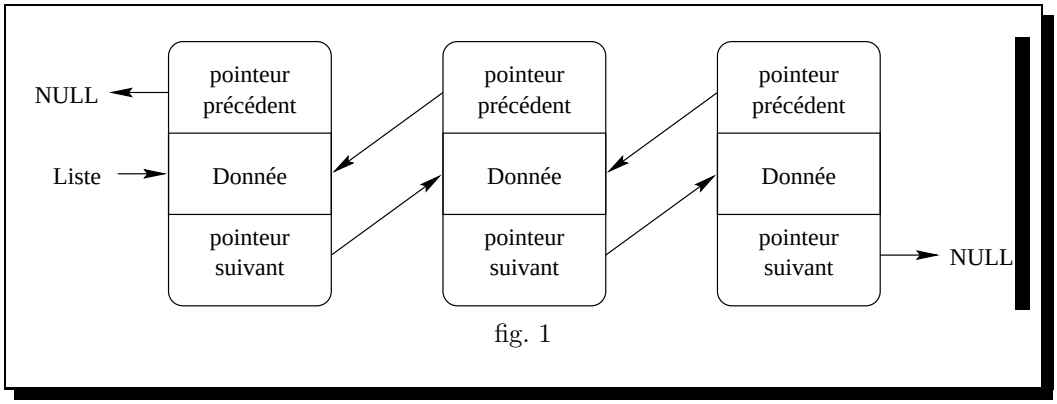


fig. 1

Ainsi, chaque élément de la liste possède trois pointeurs : un sur la donnée, un sur l'élément précédent et un sur l'élément suivant. Aussi, la glib définit-elle le type `GList` de cette façon :

```
typedef struct _GList GList;
struct _GList
{
  gpointer data;
  GList *next;
  GList *prev;
};
```

Les données stockées dans ces listes sont de type `gpointer` c'est-à-dire des pointeurs sur une structure quelconque. Mais grâce aux macros de conversion entre entiers et pointeurs, rien n'empêche de stocker des entiers dans des `GList`.

La plupart des fonctions relatives aux `GList` attendent un pointeur sur le premier élément de la liste comme premier paramètre et le renvoient en valeur de retour. Ce premier élément peut changer (notamment, lors d'une insertion ou d'un tri), c'est pourquoi il faut toujours noter quel est le nouveau premier élément à la suite d'un appel de fonction.

Une telle liste est déclarée et initialisée simplement par :

```
GList *liste = NULL;
```

Par la suite on peut facilement ajouter des éléments à cette liste grâce aux fonctions :

```
GList *g_list_append(GList *liste,
                     gpointer donnee);
GList *g_list_prepend(GList *liste,
                      gpointer donnee);
GList *g_list_insert(GList *liste,
                     gpointer donnee,
                     gint position);
GList *g_list_insert_sorted(GList *liste,
                            gpointer  donnee,
                            GCompareFunc comp);
```

Les trois premières doivent commencer à vous être familières. `g_list_append()` ajoute un élément à la fin de la liste avec comme donnée le pointeur `donnee`. `g_list_prepend()` fait la même chose mais au début de la liste, le premier élément de la liste est donc forcément changé et il convient de le sauvegarder. `g_list_insert` permet d'insérer un élément à la position `position`.

La fonction `g_list_insert_sorted()` mérite que l'on s'y attarde un peu plus. Elle permet d'insérer un élément à la bonne place, de manière à ce que la liste reste triée en permanence. Le paramètre `comp` est un pointeur sur une fonction qui doit renvoyer un entier positif si le premier élément (passé en premier paramètre) doit être placé après le second (passé en second paramètre) dans l'ordre que l'on

souhaite pour notre liste. Voici un exemple permettant de clarifier l'utilisation de cette fonction :

```
#include <glib.h>

typedef struct
{
  int age;
  char *nom;
} personne_type;

int compare(personne_type *P1, personne_type *P2)
{
  if (P1->age > P2->age)
    return 1;
  else
    return -1;
}

void main(void)
{
  GList *liste = NULL;
  personne_type *personne;

  personne = g_malloc(sizeof(personne_type));
  personne->age = 15;
  personne->nom = "Pierre";
  liste = g_list_insert_sorted(liste, personne,
                               (GCompareFunc)compare);
  personne = g_malloc(sizeof(personne_type));
  personne->age = 20;
  personne->nom = "Paul";
  liste = g_list_insert_sorted(liste, personne,
                               (GCompareFunc)compare);
  personne = g_malloc(sizeof(personne_type));
  personne->age = 18;
  personne->nom = "Jacques";
  liste = g_list_insert_sorted(liste, personne,
                               (GCompareFunc)compare);
  printf("1 : %s, 2 : %s, 3 : %s\n",
         ((personne_type *)(liste->data))->nom,
         ((personne_type *)(liste->next->data))->nom,
         ((personne_type *)(liste->next->next->data))->nom);
}
```

Après une compilation de ce programme par :

```
gcc liste.c -o liste ‘glib-config --cflags --libs‘
```

ce programme affiche :

```
1 : Pierre, 2 : Jacques, 3 : Paul
```

car les personnes ont été classées par âge croissant.

Les éléments peuvent être enlever de la liste en utilisant :

```
GList* g_list_remove(GList *liste, gpointer donnee);
```

où `donnee` est la donnée de l'élément que l'on veut supprimer.

Une fois que des éléments sont dans la liste, il existe beaucoup de fonctions permettant de retrouver facilement un élément particulier. Ainsi,

```
GList *g_list_first(GList *element);
GList *g_list_last(GList *element);
```

permettent respectivement de retrouver le premier et le dernier élément d'une liste dont on passe un élément en paramètre.

L'élément suivant ou précédant un élément particulier peut être trouver avec :

```
GList *g_list_next(GList *element);
GList *g_list_previous(GList *element);
```

Le $n^{ième}$ élément d'une liste peut être récupéré par un appel à :

```
GList *g_list_nth(GList *list, guint n);
```

Si l'on préfère retrouver un élément dont on connait la donnée, la fonction

```
GList *g_list_find(GList *liste, gpointer donnee);
```

nous le permet.

De même, on peut retrouver la position dans la liste d'un élément ou même d'un élément dont on ne connait que la donnée avec :

```
gint g_list_position(GList *liste, GList *element);
gint g_list_index(GList *liste, gpointer donnee);
```

On peut à tout moment savoir combien d'éléments comporte notre liste avec :

```
guint g_list_length(GList *liste);
```

Lorsque l'on utilise des structures comme les listes chaînées, on est souvent amené à écrire et réécrire toujours le même type de boucle “`for`”. Les concepteurs de la glib ont une solution particulière pour nous éviter de réinventer l'eau chaude à chaque fois :

```
void g_list_foreach(GList *liste,
                    GFunc fonction,
                    gpointer donnee_sup);
```

Ceci permet d'appeler la fonction `fonction` pour tous les éléments de la liste, avec la donnée de l'élément en premier paramètre et `donnee_sup` comme deuxième

paramètre. Ainsi dans notre exemple précédent, on aurait pu afficher la liste en appelant :

```
g_list_foreach(liste, (GFunc)affiche, NULL);
```

en admettant que la fonction `affiche()` ait été définie comme suit :

```
void affiche(personne_type *element)
{
  printf("nom : %s\n", element->nom);
}
```

Essayez de bien comprendre comment tout cela fonctionne, ce qui vous permettra d'économiser beaucoup de lignes de code dans vos programmes.

Si vous avez deux listes que vous voulez concaténer, la fonction :

```
GList *g_list_concat(GList *liste1, GList *liste2);
```

est faite pour vous.

Une liste peut aussi être triée a posteriori par :

```
GList* g_list_sort(GList *list,
                   GCompareFunc fonction_de_comparaison);
```

où `fonction_de_comparaison` est une fonction de comparaison du même type que celles employées pour la fonction `g_list_insert_sorted()`.

Enfin, une liste entière peut être libérée[4] grâce à :

```
void g_list_free(GList *liste);
```

7.2. Les listes simplement chaînées

Les listes simplement chaînées sont des structures de données qui ressemblent beaucoup aux listes doublement chaînées. Leur principale différence est que les éléments des listes simplement chaînées ne possèdent pas de pointeur sur l'élément précédent. (voir fig. 2)

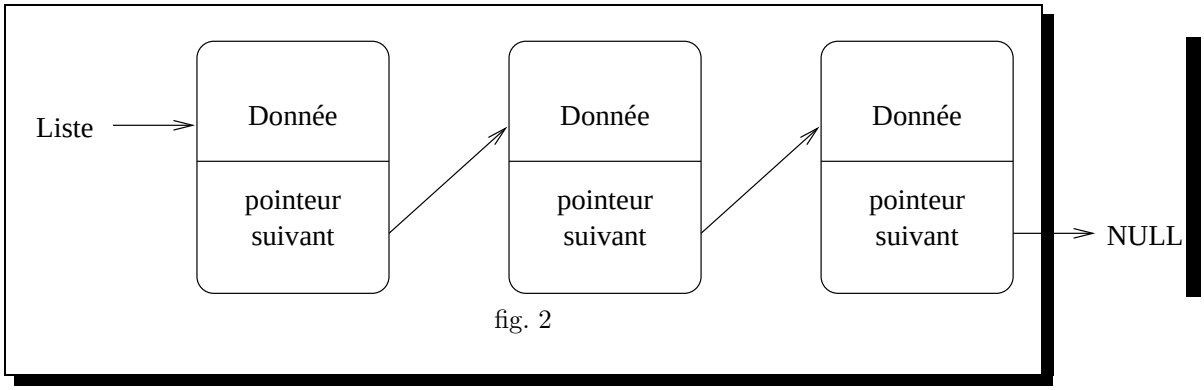


fig. 2

[4] Attention ! Cela ne libère pas la mémoire éventuellement allouée pour stockée les données des éléments.

Ainsi, la structure `GSList` est déclarée :

```
typedef struct _GSList GSList;
struct _GSList
{
  gpointer data;
  GSList *next;
};
```

Une liste simplement chaînée se déclare et s'initialise ainsi :

```
GSList *liste = NULL;
```

Les fonctions s'appliquant aux `GSLists` sont les mêmes que celles s'appliquant aux `GLists`. Il suffit de remplacer le `g_list` initial par un `g_slist`. Seule la fonction `g_list_previous()` n'a (évidemment) pas d'équivalent pour les GSList. Je ne redévelopperais donc pas l'ensemble de ces fonctions.

Sachez aussi que la glib propose aussi des fonctions de gestion des arbres, qu'ils soient binaires ou pas. Cependant leur étude dépasse le cadre de ce livre car ils ne sont pratiquement pas utilisés dans le cadre de GTK+.

8. Les fonctions utilitaires

La glib propose quelques définitions et macros d'usage fréquent, telles que `FALSE` qui vaut 0, `TRUE` qui vaut 1, `MAX(a, b)` qui renvoie la plus grande des deux valeurs `a` et `b`, `MIN(a, b)` qui renvoie la plus petite des deux valeurs `a` et `b`, `ABS(a)` qui renvoie la valeur absolue de `a`.

Et `CLAMP(a, min, max)` qui renvoie
`a` si `a` est compris entre `min` et `max`,
`min` si `a` est inférieur à `min`,
`max` si `a` est supérieur à `max`.

9. Les assertions

Les assertions sont un moyen de débogguer une application.

La glib propose quatre variantes d'assertions. La première, `g_assert(expr)` écrit un message contenant le nom du fichier source et le numéro de la ligne sur laquelle est cette assertion et termine le programme si l'expression est fausse. Ceci peut être particulièrement utile pour tester un cas pathologique qui ne devrait jamais arriver dans le fonctionnement normal de notre programme.

La deuxième, `g_assert_not_reached()` écrit un message avec nom de fichier et numéro de ligne et sort du programme. En fait cette assertion permet de savoir si notre programme passe par une ligne par laquelle il ne devrait pas. On l'utilise par exemple dans le "`default:`" d'un `switch` dans lequel on pense avoir traité tous les cas.

La troisième, `g_return_if_fail(expr)` sort de la fonction immédiatement si l'expression `expr` est fausse. Ceci sert surtout pour tester la validité des paramètres

passés à une fonction. Ainsi, si les paramètres ne sont pas adaptés, la fonction peut choisir de finir son exécution.

La quatrième, `g_return_val_if_fail(expr, val)` est l'équivalente de la précédente pour les fonctions qui doivent retourner une valeur. Cette assertion sort de la fonction et renvoie la valeur `val` si l'expression `expr` est fausse.

Voici un exemple d'utilisation de tout cela :

```
typedef enum
{
  Masculin,
  Feminin,
  Autre
} sexe;

void Affiche_Entete(gchar *nom, sexe genre, gint age)
{
  g_assert(age>=0);
  g_return_if_fail(nom != NULL);
  switch (genre)
  {
    case Masculin:
      printf("M. ");
      break;
    case Feminin:
      printf("Mme. ");
      break;
    default:
      g_assert_not_reached();
  }
  printf("%s\n", nom);
}
```

Ici, la fonction `Affiche_Entete()` terminera le programme si l'âge passé en paramètre est négatif, ce qui nous évitera d'écrire un courrier à quelqu'un qui n'est pas encore né. Le programme s'arrêtera aussi si le sexe d'une personne (déjà née) n'est ni `Masculin`, ni `Feminin`, ce qui est aussi un cas assez pathologique. En revanche, la fonction affichera un message d'erreur et se terminera, sans finir le programme si le nom passé en paramètre est le pointeur `NULL`.

10. Les timers

Les timers permettent de savoir précisément combien de temps s'est écoulé entre deux instants. Pour ce faire, on initialise d'abord une structure `GTimer` à l'aide de la fonction :

```
GTimer *g_timer_new();
```

Puis le timer peut être lancé par

```
g_timer_start(GTimer *Timer);
```

et stoppé par

```
g_timer_stop(GTimer *Timer);
```

Par la suite on peut consulter le temps écoulé entre ces deux appels grâce à la fonction

```
gdouble g_timer_elapsed(GTimer *timer,
                        gulong *microsecondes);
```

où `timer` est notre timer, `microsecondes`, s'il est non nul, est une adresse où sera stockée la partie "microsecondes" de la durée écoulée et la valeur renvoyée est le nombre de secondes écoulées entre les deux appels.

Exemple :

```
GTimer *timer;

timer = g_timer_new();
g_timer_start(timer);
appel_d_une_fonction_tres_longue_a_executer();
g_timer_stop(timer);
printf("L'appel de la fonction a duré : %g\n",
       g_timer_elapsed(timer, NULL));
```

11. Les modules chargeables

Les modules sont des fichiers objets (`.so` généralement sous Unix ou `.DLL` sous Windows). Ces fichers contiennent des fonctions ou symboles qui pourront être appelés dans nos programmes. Et comme ils ne sont pas liés statiquement, ils peuvent être chargés et déchargés dynamiquement pendant l'exécution de nos applications. On les appelle parfois "*plug-ins*". Les fonctions présentées ici implémentent une façon portable de charger ces fichiers dynamiquement. L'utilisation de ces fonctions n'est possible que si le système supporte la fonction `dlopen()` ou la fonction `shl_load()`.

Avant de tenter d'essayer d'utiliser les modules chargeables dans une application, il faut d'abord vérifier que la plateforme sur laquelle elle sera exécutée supporte bien cette fonctionnalité. La fonction

```
gboolean g_module_supported(void);
```

permet justement de savoir cela. La valeur renvoyée sera `TRUE` si la plateforme supporte les modules chargeables et `FALSE` dans le cas contraire.

Une fois que l'on est sûr que les modules chargeables sont supportés, on peut en ouvrir un en appelant la fonction

```
GModule *g_module_open(const gchar *nom_fichier,
                       GModuleFlags drapeau);
```

où `nom_fichier` est le nom complet du fichier contenant le module, et `drapeau` indique si tous les symboles doivent être lus immédiatement (il vaut alors zéro), ou si au contraire ils doivent n'être lu que sur commande (`drapeau` vaut alors `G_MODULE_BIND_LAZY`). Ceci peut être utile si le module comporte de nombreux symboles. La valeur de retour de `g_module_open()` est un pointeur sur une structure `GModule` qui définit entièrement le module et qui ne doit être utilisée qu'au travers des fonctions qui suivent. Si le module ne peut être ouvert, la valeur renvoyée est `NULL`.

Une fois le module chargé en mémoire, on peut en extraire les symboles du module avec :

```
gboolean g_module_symbol(GModule *module,
                         const gchar *nom_du_symbole,
                         gpointer *symbole);
```

où `module` est un pointeur vers la structure qui décrit notre module, `nom_du_symbole` est le nom du symbole que l'on souhaite retrouver, et `symbole` est l'adresse mémoire où sera rangé un pointeur sur ce symbole. La valeur de retour est `TRUE` si le symbole a pu être retrouvé, `FALSE` sinon.

Par exemple, imaginons que le module `monmodule.so` définisse une fonction nommée "`Affiche`", qui prenne en paramètre une chaîne de caractères et qui l'affiche à l'écran. Voici un bout de programme qui charge le module, retrouve la fonction `Affiche`, et l'appelle :

```
[ ... ]
  GModule *module;
  void (*fonction_affiche)(gchar *);

  module = g_module_open("monmodule.so", 0);
  g_module_symbol(module, "Affiche",
                  (gpointer *)(&fonction_affiche));
  fonction_affiche("coucou");
  [ ... ]
```

À tout moment, on peut connaître le nom d'un module grâce à :

```
gchar *g_module_name(GModule *module);
```

Par la suite, on peut décharger le module de la mémoire en appelant :

```
gboolean g_module_close(GModule *module);
```

où `module` est évidemment un pointeur sur la structure décrivant le module.

Il peut cependant être très utile d'interdire le déchargement d'un module. Par exemple, dans le bout de code précédent, on peut vouloir interdire que le module "`monmodule.so`" ne soit déchargé, afin que la variable `fonction_affiche` contienne toujours une adresse de fonction valide. La fonction :

```
void g_module_make_resident(GModule *module);
```

empêche que les appels ultérieurs à `g_module_close()` ne déchargent véritablement le module.

Toutes les fonctions qui renvoient une valeur booléenne pour signifier si tout s'est bien passé positionnent en fait un message d'erreur interne qui peut être affiché par un appel à :

```
gchar *g_module_error(void);
```

Bien sûr, seule la dernière erreur peut être affichée par ce biais.

De plus, la glib propose deux mécanismes afin d'initialiser un module automatiquement lors de son chargement et par la suite de "faire le ménage" au déchargement du module.

Si un module contient une fonction nommée `g_module_check_init()`, elle est automatiquement appelée au chargement du module. Elle doit avoir un prototype compatible avec le suivant :

```
const gchar* g_module_check_init(GModule *module);
```

et doit renvoyer la valeur `NULL` si tout s'est bien passé, ou une chaîne de caractères décrivant le problème, s'il y a eu une erreur.

De la même façon, si le module possède une fonction `g_module_unload`, celle-ci sera automatiquement appelée avant que le module ne soit déchargé de la mémoire. Cette fonction doit avoir un prototype compatible avec le suivant :

```
void g_module_unload(GModule *module);
```

Voici un exemple qui résume l'utilisation des modules :

testmodule.c

```
#include <stdio.h>
#include <gmodule.h>

void main(void)
{
  GModule *module;
  void (*fonction_affiche)(gchar *);

  module = g_module_open("./monmodule.so", 0);
  g_module_symbol(module, "Affiche",
                  (gpointer *)(&fonction_affiche));
  printf("Le nom du module est : %s\n",
         g_module_name(module));
  fonction_affiche("coucou");
  g_module_close(module);
}
```

monmodule.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
char* g_module_check_init(void)
{
  printf("Initialisation du module\n");
  return NULL;
}

void Affiche(char *Message)
{
  printf("%s\n", Message);
}

void g_module_unload(void)
{
  printf("Déchargement du module\n");
}
```

Makefile

```
all: testmodule monmodule.so

GLIBS = ‘glib-config --libs gmodule‘
GFLAGS = ‘glib-config --cflags gmodule‘

testmodule: testmodule.o
cc testmodule.o -o testmodule $(GLIBS) -Wall

testmodule.o: testmodule.c
cc -c testmodule.c -o testmodule.o $(GFLAGS) -Wall


monmodule.so: monmodule.o
cc -o monmodule.so.0.0.1 monmodule.o \
     -shared -Wl,-soname,monmodule.so
ln -s monmodule.so.0.0.1 monmodule.so

monmodule.o: monmodule.c
cc -c monmodule.c -o monmodule.o -Wall
```

Saisissez ces trois fichiers, compilez le tout en lançant `make`. Ceci créera le module `monmodule.so` et le programme `testmodule`. Lancez ce programme en essayant de comprendre tout ce qui se passe. Les utilisations des modules chargeables sont extrêmement nombreuses puisqu'ils permettent de construire des applications totalement modulaires.

12. Les gthreads[5]

Les threads sont un peu comme des processus, à la différences près que des threads issus d'un même processus partagent la même portion de mémoire. Ceci présente quelques avantages, car la communication entre threads est très facile et se fait directement en mémoire. La contre-partie est qu'une application utilisant des threads peut devenir très difficile à débogguer. En effet, l'accès à des données communes par deux threads simultanés peut engendrer des bogues très pernicieux (les fameux "heisenbogues") si l'on ne prend pas toutes les précautions nécessaires. Pour empêcher cela, on peut avoir recours à des mécanismes de synchronisation. Le rôle de la glib est ici de proposer ce genre de mécanismes, mais n'attendez pas de la glib des fonctions permettant de créer facilement des threads (du moins pour l'instant ; la glib est un projet vivant, il évolue sans arrêt).

Ainsi, la glib propose, entre autres, des mécanismes d'exclusion mutuelle pour les accès à la mémoire, et des fonctions utilitaires pour protéger l'accès aux données d'un thread.

Avant même d'essayer d'appeler une des fonctions de gestion des threads, il faut vérifier que la glib a été compilée avec le support pour les thread. Si c'est le cas, le symbole `G_THREADS_ENABLED` est défini, on peut donc inclure un test au début de notre programme du genre :

```
#ifndef G_THREADS_ENABLED
#error "Ce programme demande une version de la \
        glib supportant les thread"
#endif
```

Une fois ce test passé, on peut initialiser le système de gestion des threads de la glib en appelant :

```
void g_thread_init(GThreadFunctions *vtable);
```

où `vtable` est un pointeur sur une table interne de la glib. En règle générale, on passe simplement la valeur `NULL` et la glib se débrouille pour remplir la bonne table.

`g_thread_init()`[6] ne peut être utiliser qu'une seule fois. Le second appel à cette fonction arrêtera l'exécution du programme en affichant une erreur.

Heureusement, il est possible de savoir si on l'a déjà appelée grâce à la fonction :

```
gboolean g_thread_supported(void);
```

Cette fonction renvoie `TRUE` si le système de gestion des threads a déjà été initialisé, et `FALSE` sinon.

---

[5] cette partie est quelque peu inspirée de la documentation de référence de la glib disponible en anglais sur le site `http ://www.gtk.org`

[6] pour utiliser `g_thread_init()` dans vos programmes, vous devez le lier avec certaines bibliothèques. Le plus simple est d'appeler : `glib-config --libs gthread` pour savoir exactement ce qu'il faut passer à l'éditeur de liens

12.1. Les mutex

Les mutex (ou exclusion mutuelle) sont une méthode possible pour gérer les conflits lorsque deux threads veulent d'accéder à la même donnée.

Voyons un exemple afin de mieux comprendre l'utilité des mutex :

```
int nombre_suivant()
{
  static int nombre = 0;
   /* imaginons que calcule_prochain_nombre
    * soit une fonction compliquée qui nous
    * donne le prochain nombre d'une suite
    * tarabiscotée
    */
  nombre = calcule_prochain_nombre(nombre);
  return nombre;
}
```

On voit bien que cette fonction de fonctionne pas correctement dans un environnement multithread. En effet si deux (ou plus) threads l'appellent en même temps ou presque, le nombre calculé risque bien d'être indéfini puisqu'il peut être changé entre le moment où on appelle `calcule_prochain_nombre()` et le moment où l'on renvoie le nouveau nombre par un autre thread concurrent qui appelerait `calcule_prochain_nombre()` pendant ce temps. L'accès à la variable `nombre` doit donc être protégé.

Une possibilité (boguée) pour gérer cela pourrait être :

```
int nombre_suivant()
{
  static int nombre = 0;
  int valeur_de_retour;
  static GMutex * mutex = NULL;

  if ( !mutex)
    mutex = g_mutex_new();
  g_mutex_lock(mutex);
  valeur_de_retour = nombre
                   = calcule_prochain_nombre(nombre);
  g_mutex_unlock(mutex);
  return valeur_de_retour;
}
```

Cela ressemble tout à fait à un code correct, mais si on y regarde de plus près, il peut arriver que deux threads concurrents passent en même temps sur le `if ( !mutex)` et appellent donc tous les deux la fonction `g_mutex_new()` en même temps ce qui conduit à une valeur de `mutex` encore une fois indéterminée. Il ne faut donc pas utiliser ce code simpliste dans vos programmes. Une (vraie) solution pourrait être :

```
static GMutex *nombre_mutex = NULL;


/* Cette fonction doit être appelée avant le premier
 * appel à la fonction nombre_suivant(). Elle doit
 * être appelée une seule fois par un seul thread */
void initialise_nombre_suivant()
{
  g_assert(nombre_mutex == NULL);
  nombre_mutex = g_mutex_new();
}

int nombre_suivant()
{
  static int nombre = 0;
  int valeur_de_retour;

  g_mutex_lock(nombre_mutex);
  valeur_de_retour = nombre = calcule_nombre_suivant(nombre);
  g_mutex_unlock(nombre_mutex);
  return valeur_de_retour;
}
```

Ainsi, une fois que la fonction `initialise_nombre_suivant()` a été appelée (une fois et une seule), les threads peuvent appeler la fonction `nombre_suivant()` sans avoir de problème d'accès. Bien sûr, la variable `nombre` est partagée entre tous les threads et rien n'empêche un thread de changer sa valeur plusieurs fois entre chaque appel d'un autre thread, mais ce problème aussi peut être résolue avec des fonctions de la glib.

Voyons maintenant en détails le rôle de chacune des fonctions de gestion des mutex. Tout d'abord, un mutex doit être créé en appelant la fonction :

```
GMutex *g_mutex_new(void);
```

`GMutex` est une structure opaque qui définit complétement un mutex.

Une fois le mutex créé, on peut le verrouiller avec :

```
void g_mutex_lock(GMutex *mutex);
```

Si le mutex est déjà verrouillé par un autre thread, le thread courant suspend son exécution et attend son tour jusqu'à ce que le mutex soit déverrouillé par l'autre thread. Chaque thread doit donc penser à déverrouiller tous les mutex qu'il verrouille sinon, les autres threads risquent d'attendre indéfiniment.

Il n'est pas toujours souhaitable d'attendre longtemps si on a autre chose à faire, ainsi :

```
void g_mutex_trylock(GMutex *mutex);
```

essaie de verrouiller un mutex. Si le mutex est déjà verrouillé par un autre thread, cette fonction retourne immédiatement `FALSE`, sinon, elle verrouille le mutex et renvoie `TRUE`.

Une fois que l'on a fini d'accéder à la variable sensible, il faut déverrouiller le mutex avec :

```
void *g_mutex_unlock(GMutex *mutex);
```

Cette fonction déverrouille le mutex, et si un autre thread attendait ce moment, il se réveille automatiquement, verrouille lui-même le mutex et continue son exécution.

Enfin, une fois que l'on a plus besoin d'un mutex, on peut le libérer de la mémoire (commune à tous les threads) par un appel à :

```
void g_mutex_free(GMutex *mutex);
```

12.2. Les mutex statiques

Les mutex statiques fonctionnent comme les mutex tout court, mais ils possèdent un réel avantage sur ces derniers. On n'a pas besoin de les créer par un `g_mutex_new()`, mais ils sont créés comme des variables statiques classiques. On a donc plus le problème de savoir quel est *le* thread qui doit créer le mutex, puisqu'il est créé au moment de la compilation. Voici une nouvelle version de l'exemple précédent qui est plus courte, plus sûre et plus facile à concevoir :

```
int nombre_suivant()
{
  static int nombre = 0;
  int valeur_de_retour;
  static GStaticMutex mutex = G_STATIC_MUTEX_INIT;

  g_static_mutex_lock(&mutex);
  valeur_de_retour = nombre
                   = calcule_prochain_nombre(nombre);
  g_static_mutex_unlock(&mutex);
  return valeur_de_retour;
}
```

Comme vous le voyez, les `GStaticMutex` sont nettement plus faciles d'utilisation et plus naturels que les `GMutex`. En revanche, il est indispensable de bien initialiser tous les mutex statiques en utilisant la valeur `G_STATIC_MUTEX_INIT`. Notez aussi que maintenant, `mutex` désigne une *structure* `GStaticMutex` et non plus un *pointeur* sur une structure `GMutex`.

Les fonctions de gestion des `GStaticMutex` sont globalement identiques à celles utilisées pour les `GMutex`, ainsi on retrouve les trois fonctions :

```
void g_static_mutex_lock(GStaticMutex *mutex);
void g_static_mutex_trylock(GStaticMutex *mutex)
void g_static_mutex_unlock(GStaticMutex *mutex)
```

Cependant certaines opérations sont impossibles à réaliser avec des mutex statiques[7]. De temps en temps, il est nécessaire de trouver quel est le `GMutex` associé à un `GStaticMutex`. C'est le rôle de la fonction :

```
GMutex *g_static_mutex_get_mutex(GStaticMutex *mutex);
```

Comme il peut être difficile d'arriver à se souvenir de la syntaxe de ces fonctions, les concepteurs de la glib ont créé des macros permettant d'utiliser facilement les possibilités de verrouillage offertes par les mutex. Ces macros dont le nom commence généralement par `G_LOCK`, présentent l'avantage de pouvoir être utilisée même si la glib n'a pas été compilée avec le support pour les threads (ces macros ne réalisent alors aucune opérations...). Ainsi,

```
G_LOCK_DEFINE(verrou);
```

définit un verrou dont le nom est `verrou`. On l'utilise comme une déclaration de variable[8]. On utilise en général le nom de la variable que l'on veut protéger comme nom de verrou, comme dans l'exemple suivant :

```
int nombre_suivant()
{
  static int nombre = 0;
  int valeur_de_retour;
  G_LOCK_DEFINE_STATIC(nombre);

  G_LOCK(nombre);
  valeur_de_retour = nombre
                   = calcule_prochain_nombre(nombre);
  G_UNLOCK(nombre);
  return valeur_de_retour;
}
```

Vous l'aurez compris, la macro :

```
G_LOCK_DEFINE_STATIC(verrou);
```

fonctionne de la même façon que `G_LOCK_DEFINE` mais il crée une variable statique.[9]

De la même façon, on peut faire référence à un verrou défini dans un autre module, en utilisant :

```
G_LOCK_EXTERN(nom);
```

qui réalise la même chose que la macro `G_LOCK_DEFINE` mais en utilisant le mot-clef `extern`.

---

[7] vous ne croyiez tout de même pas qu'ils n'avaient que des avantages ?

[8] en fait la macro `G_LOCK_DEFINE` définit un mutex statique nommé `g__name_lock`

[9] Attention à ne pas confondre. `G_LOCK_DEFINE` et `G_LOCK_DEFINE_STATIC` déclarent tous les deux un mutex statique, simplement ce dernier déclare la variable correspondante en utilisant le mot clef `static` ce qui est souvent souhaitable pour déclarer le verrou dans une fonction.

Une fois le verrou défini, on peut l'utiliser facilement un peu comme les fonctions de gestion des mutex ; ainsi `G_LOCK()` fonctionne de la même façon que `g_mutex_lock()`, `G_TRYLOCK()` correspond à `g_mutex_trylock()` et `G_UNLOCK()` à `g_mutex_unlock()`.

En règle générale, utilisez plutôt les macros `G_LOCK*` que les fonctions `g_mu\-tex*` ou `g_sta\-tic_mu\-tex*` dans vos programmes. C'est à la fois plus simple et plus sûr.

12.3. Les données privées

Comme je l'ai déjà dit, les threads issus d'un même processus partagent le même espace mémoire et partagent donc toutes leur données. Ce n'est pas toujours ce que l'on souhaite. Par exemple, dans notre fonction `nombre_suivant()` nous pouvons légitimement souhaiter que la variable `nombre` ne soit pas partagée. Ainsi, chaque thread pourrait obtenir la liste complète des nombres calculés par `calcule_pro\-chain_nom\-bre()`. Ceci peut être implémenté en utilisant ce que la glib appelle des `GPrivate` comme suit :

```
GPrivate* nombreID = NULL; /* devra être initialisé quelque
                            * part en utilisant
                            * g_private_new();
                            */
int nombre_suivant()
{
  gint *nombre = g_private_get(nombID);

  if (!nombre)
  {
    nombre = g_new(gint,1);
    *nombre = 0;
    g_private_set(nombreID, nombre);
  }
  *nombre = calcule_prochain_nombre(*nombre);
  return *nombre;
}
```

Ici, le pointeur correspondant à l'identifiant `nombreID` est lu. S'il est `NULL`, c'est qu'il n'a pas encore été créé. On alloue alors la mémoire nécessaire pour contenir un `gint`, initialise le `nombre` à zéro et réassigne le pointeur à l'identifiant `nombreID`. Nous avons maintenant une variable de type `gint`, qui est résérvée au thread courant[10]. C'est cette variable qui est appelée *donnée privée*.

Voyons à présent un peu plus en détail les fonctions de gestion des données privées. Tout d'abord, un identifiant de variable privé doit être obtenu en appelant :

[10] Les fonctions de verrouillage ont été supprimées de cet exemple pour des raison de clarté, bien entendu, dans un exemple réel, l'utilisation d'un mutex est nécessaire pour protéger l'accès à la variable `nombre`.

```
GPrivate *g_private_new(destructeur);
```

où `destructeur` est un pointeur sur une fonction. Cette fonction sera chargée de détruire la donnée privée[11] lorsqu'on n'en aura plus besoin. Lorsque le thread se terminera, et si le pointeur correspondant à l'identifiant privé n'est pas `NULL`, la fonction `destructeur` sera appelée avec ce pointeur comme seul paramètre. Dans notre exemple, comme dans la plupart des cas, cette fonction sera simplement :

```
void destructeur(int *pointeur)
{
  g_free(pointeur);
}
```

Si destructeur vaut `NULL`, aucune fonction ne sera appelée.

Notez que les `GPrivate` ne peuvent pas (pour des raisons techniques) être désalloués, essayez donc de les réutiliser dans la mesure du possible.

Une fois que le `GPrivate` (c'est-à-dire l'identifiant privé) a été créé, il peut être associé à un pointeur avec :

```
void g_private_set(GPrivate *ID, gpointer pointeur);
```

où `ID` est un identifiant privé (`GPrivate *`) préalablement alloué avec `g_private_new()`, et `pointeur` est un pointeur sur une structure que l'on veut garder privée.

On pourra par la suite accéder à cette structure grâce à :

```
gpointer g_private_get(GPrivate *ID);
```

qui retourne le pointeur associé à l'identifiant privé `ID` pour le *thread courant*. Ce pointeur est `NULL` tant qu'il n'a pas été affecté par un appel à `g_private_set()`.

Le problème avec les `GPrivate` est qu'il est nécessaire de les allouer de façon dynamique pendant l'exécution du programme, et qu'ils ne peuvent même pas être désalloués. C'est pourquoi il existe un équivalent statique aux `GPrivate`. Il s'agit des `GStaticPrivate`. On peut dire qu'il existe les mêmes différences entre les `GPrivate` et les `GStaticPrivate` qu'entre les `GMutex` et les `GStaticMutex`. Voyons maintenant ce que devient notre exemple en utilisant des `GStaticPrivate`[12] :

```
int nombre_suivant()
{
  static GStaticPrivate nombreID = G_STATIC_PRIVATE_INIT;
  int *nombre = g_static_private_get(&nombreID);

  if (!nombre)
  {
    nombre = g_new(int,1);
```

[11] la mémoire réservée pour stoquer le nombre dans notre exemple

[12] Si l'on voulait être totalement exhaustif, il faudrait ajouter les fonctions de verrouillage dans cet exemple et dans le précédent. Elles ont été volontairement omises pour ne pas surcharger l'ensemble.

```
    *nombre = 0;
    g_static_private_set(&nombreID, nombre, g_free);
  }
  *nombre = calcule_prochain_nombre(*nombre);
  return *nombre;
}
```

Comme pour les `GStaticMutex`, vous constaterez que les `GStaticPrivate` sont plus faciles à utiliser que les `GPrivate`, la seule contrepartie est qu'il faut absoluement initialiser les `GStaticPrivate` avec la valeur `G_STATIC_PRIVATE_INIT`.

L'association d'un pointeur avec un identifiant privé statique est cependant un peu plus complexe que pour un identifiant privé dynamique :

```
void g_static_private_set(GStaticPrivate *ID,
                          gpointer pointeur,
                          GDestroyNotify destructeur);
```

Ici, l'identifiant privé statique est `ID` et on l'associe au pointeur `pointeur`. Le paramètre `destructeur` est un pointeur sur une fonction qui sera appelée lorsque le pointeur sera réassigné ou que le thread prendra fin. Ici aussi, c'est en général simplement une fonction de libération de la mémoire.

Et de la même façon que pour les `GPrivate`, on récupère le pointeur associé à un identifiant privé statique en appelant la fonction :

```
gpointer g_static_private_get(GStaticPrivate *ID);
```

13. Bien d'autres choses encore...

Sachez aussi que la glib recèle encore de bien d'autres groupes de fonction, comme, par exemple, les `g_scanner*` qui permettent la réalisation d'un analyseur lexical à peu de frais, ou les `g_completion*` qui réalisent le complètement de chaînes de caractères, ou encore des fonctions de gestion des dates qui sont immunisées contre le bogue de l'an 2000... Mais celles-ci ne sont pratiquement pas utilisées par GTK, elles sortent donc du propos du présent ouvrage.

4

Le GDK

Le GDK ou Gimp Drawing Kit est la bibliothèque de GTK+ chargée de l'affichage à l'écran et de la gestion des interactions avec le clavier et la souris. En fait, le GDK est surtout une encapsulation de la plupart des fonctions de la Xlib. Ainsi un programmeur X-Window devrait pouvoir comprendre très facilement la grande majorité des fonctions présentées ici. De plus le GDK apporte un certain nombre de simplifications bien venues notamment au niveau de la gestion des couleurs.

1. Le modèle X-Window

X-Window est l'interface graphique la plus utilisée sous Unix. Elle propose un environnement graphique fenêtré avec une philosophie client/serveur. En effet X-window est séparé en deux parties :

- le serveur X, proche du hardware, se charge de l'affichage à l'écran et de la gestion du clavier et conserve la plupart des informations concernant l'état du serveur ;
- la Xlib est une bibliothèque de fonctions C qui fait la jonction entre le serveur X et les applications.

La Xlib et le serveur X discutent à travers un réseau du genre TCP/IP, ce qui permet, entre autres, de faire tourner le serveur X et nos applications sur des machines totalement différentes reliées par n'importe quel réseau de ce type.

Ainsi une application tournant sous X-Window utilise la Xlib pour emettre des requètes vers le serveur X du genre : "déplace cette fenêtre", "dessine un cercle" et le serveur X envoie des informations sur des événements à notre application via le réseau et la Xlib. Ces informations peuvent être du genre : "la souris a été déplacée" ou "cette fenêtre doit être redessinée".

La Xlib propose donc pléthore de fonctions permettant de demander au serveur X de dessiner telle ou telle chose. En revanche les événements provenant du serveur et destinés à notre application sont stockés par la Xlib en attendant que nous demandions à les connaître.

En résumé, une application X-Window passe son temps à lire les événements en attente et réagit en fonction de ceux-ci en demandant au serveur X (via la Xlib) d'accomplir telle ou telle action. C'est pourquoi on entend souvent dire qu'une telle application utilise une programmation *événementielle*.

Comme on le voit, la Xlib est la couche incontournable reliant une application et un serveur X. Cependant, il s'agit d'une couche très bas niveau et écrire un programme de grande envergure en n'utilisant que des appels à la Xlib se révèle très vite pénible et complexe. C'est pourquoi, il existe des bibliothèques venant se greffer au dessus de la Xlib afin d'encapsuler et de rendre tout ça plus simple. Le GDK est l'une de ces bibliothèques.

Ainsi, la fonction permettant de changer le titre d'une fenêtre avec GDK est :

```
void gdk_window_set_title(GdkWindow *fenetre,
                          const gchar *titre);
```

Pour faire la même chose en utilisant directement la Xlib, on devra utiliser :

```
void XmbSetWMProperties(Display *display,
```

```
                                    Window w,
                                    char *window_name,
                                    char *icon_name,
                                    char *argv[],
                                    int argc,
                                    XSizeHints *normal_hints,
                                    XWMHints *wm_hints,
                                    XClassHint *class_hints);
```

ce qui est tout de même moins évident à utiliser, même si la fonction de la Xlib est beaucoup plus puissante. En général, une fonction de GDK est uniquement une encapsulation d'une fonction de la Xlib avec les bons paramètres. Donc, si vous désirez en savoir plus sur une fonction particulière de GDK, il suffit d'aller fouiner dans les sources à la recherche de cette fonction et de se reporter à la documentation Xlib de la fonction encapsulée.

Les fonctions proposées par le GDK sont reconnaissables par le fait qu'elles commencent toutes par `gdk_`.

2. Les fenêtres

Vous l'aurez sans doute compris, l'une des tâches les plus importantes de X-Window est la gestion des fenêtres. Une fenêtre au sens X-Window du terme est simplement une zone rectangulaire[1] de l'écran. Cette zone ne constitue pas forcément (et même pas souvent) une fenêtre complète avec ses décorations, mais est un rectangle remarquable soit par son aspect, soit parce qu'il doit réagir à une action particulière. Par exemple, un bouton dans une interface graphique est une fenêtre X-Window, de même qu'une boîte de dialogue ou une zone d'affichage d'un texte.

Ainsi, chaque fenêtre constitue l'élément de base tant du point de vue de notre application que de celui du serveur X. Par exemple, les événements générés par le serveur ne s'adressent qu'à une seule fenêtre en général ce qui implique que seul le programme ayant créé cette fenêtre puisse connaître cet événement.

Les fenêtres X-Window peuvent être à l'intérieur d'autres fenêtres, dans quel cas seule la partie commune pourra être affichée. En effet, une fenêtre ne peut pas dépasser d'une fenêtre dans laquelle elle est incluse. Une fenêtre à l'intérieur d'une autre est dite *fille*, et celle contenant une autre est dite *parente*. Il existe donc une généalogie ou *hiérarchie* des fenêtres. La plus "vieille" est la fenêtre *root* qui constitue le fond de l'écran et qui est l'ancêtre de toutes les autres.

C'est la responsabilité du serveur de maintenir la liste de toutes les fenêtres de toutes les applications, de savoir laquelle est dans une autre, dans quel ordre elles sont affichées, leur taille, etc. Toutes ces informations sont stockées dans le serveur lui-même. Les applications ne connaissent qu'un identifiant de chaque fenêtre. Cet identifiant est de type `Window` qui est en fait une variable entière. Comme il est très fréquent qu'une `Window` soit passée en paramètre aux fonctions de la Xlib, et donc

[1] Il existe des moyens de créer des fenêtres non-rectangulaires, à l'aide de l'entension *XShape* de X-Window mais ce n'est pas le propos ici.

transite par le réseau jusqu'au serveur, il vaut mieux que le type `Window` soit le moins gourmand en mémoire possible. C'est pourquoi toutes les informations concernant une fenêtre restent du côté du serveur. Bien entendu, il existe des fonctions dans la Xlib pour récupérer telle ou telle particularité d'une fenêtre comme sa taille, sa position, etc. Beaucoup d'objets de la Xlib ont le même comportement, c'est-à-dire que l'essentiel de leur structure est stockée par le serveur et que l'application n'en connait qu'un identifiant.

Le GDK nous permet de gérer les fenêtres via le type `GdkWindow` qui est essentiellement une encapsulation du type `Window` avec quelques informations supplémentaires telles que la taille, la fenêtre parente ou la liste de ses filles en interne de façon à ne pas avoir à demander cela au serveur lorsqu'on en a besoin, afin de limiter le traffic sur le réseau.

2.1. La création des `GdkWindows`

Une telle fenêtre peut être créée avec la fonction :

```
GdkWindow* gdk_window_new(GdkWindow     *parent,
                          GdkWindowAttr *attributs,
                          gint           masque_attributs);
```

où `parent` est la fenêtre parente de celle que l'on veut créer. On passera `NULL` si la fenêtre est directement la fille de la fenêtre *root* et une fenêtre préalablement créée sinon. Le paramètre `attributs` est un pointeur sur une structure `GdkWindowAttr` qui permet de choisir certains des aspects de la nouvelle fenêtre. Cette structure est définie ainsi :

```
typedef struct  __GdkWindowAttr GdkWindowAttr;

struct __GdkWindowAttr
{
  gchar         *title;
  gint           event_mask;
  gint16         x, y;
  gint16         width;
  gint16         height;
  GdkWindowClass wclass;
  GdkVisual     *visual;
  GdkColormap   *colormap;
  GdkWindowType  window_type;
  GdkCursor     *cursor;
  gchar         *wmclass_name;
  gchar         *wmclass_class;
  gboolean       override_redirect;
};
```

Cependant, on a pas toujours envie de remplir une telle structure à chaque création de fenêtre, sachez donc que seuls les champs `event_mask`, `width`, `height`,

`wclass` et `window_type` sont obligatoires. Si l'on positionne l'un des autres champs, il faut l'indiquer en positionnant le drapeau correspondant dans le paramètre `masque_attributs`. Ce paramètre est un champ de bits où chaque bit correspond à un champ de `GdkWindowAttr`. Ces bits sont nommés en fonction du champ qu'ils valident, le tableau suivant résume les noms des bits, des champs ainsi que les valeurs par défaut des champs.

champ	bit	défaut
`title`	`GDK_WA_TITLE`	nom du programme
`x`	`GDK_WA_X`	0
`y`	`GDK_WA_Y`	0
`visual`	`GDK_WA_VISUAL`	visual par défaut
`colormap`	`GDK_WA_COLORMAP`	colormap par défaut
`cursor`	`GDK_WA_CURSOR`	curseur de la fenêtre mère
`wmclass_name`	`GDK_WA_WMCLASS`	aucun
`wmclass_class`	`GDK_WA_WMCLASS`	aucun
`override_redirect`	`GDK_WA_NOREDIR`	`FALSE`

Voyons maintenant un peu plus en détail le rôle de chacun des champs de `GdkWindowAttr` :

- `title` spécifie le titre de la fenêtre que le gestionnaire de fenêtres pourra afficher dans la barre de titre. Ce champ n'est évidemment utile que pour les fenêtres qui sont directement des filles de la fenêtre *root*. Sa valeur par défaut est le nom du programme.
- `event_mask` : il s'agit d'un champ de bits qui indique quels sont les événements que la fenêtre accepte de recevoir. Nous reviendrons sur ce point dans le paragraphe **FIXME PLEASE**. Ce champ est obligatoire.
- `x` et `y` sont les coordonnées du coin supérieur gauche de la fenêtre par rapport à sa fenêtre mère. Notez que ces coordonnées peuvent être négatives, dans quel cas uniquement la partie de notre fenêtre qui est effectivement à l'intérieur de sa fenêtre mère sera affichée. La valeur par défaut de `x` et `y` est 0.
- `width` et `height` sont respectivement la largeur et la hauteur de la fenêtre. Ces champs sont obligatoires.
- `wclass` ne peut prendre que deux valeurs : `GDK_INPUT_OUTPUT` qui est utilisé pour les fenêtres normales et `GDK_INPUT_ONLY` qui définit une fenêtre qui ne peut pas avoir de représentation physique mais qui peut recevoir des événements. Ce type de fenêtre est très peu fréquent.
- `visual` est le type de *visual* qui doit être utilisé pour cette fenêtre. Les *visuals* seront expliqués dans le paragraphe **FIXME PLEASE**.
- `colormap` est la *colormap* qui doit être utilisée pour cette fenêtre. Les *colormaps* seront étudiées dans le paragraphe **FIXME PLEASE**.
- `window_type` spécifie le type de la fenêtre au sens GDK du terme. Ce champ peut être :
  - `GDK_WINDOW_ROOT` pour la fenêtre *root*. On ne peut pas créer une telle fenêtre dans nos programme. Cette valeur est réservée par le GDK.

  - `GDK_WINDOW_TOPLEVEL` pour une fenêtre qui est directement une fille de la fenêtre *root*. Ces fenêtres sont gérées par le *gestionnaire de fenêtres* qui les décore en général en plaçant un cadre autour d'elles. Les fenêtres gérées par le gestionnaire de fenêtres sont appelées *top-level*. Pour ces fenêtres, le paramètre `parent` de la fonction `gdk_window_new()` doit être `NULL` car ces fenêtres ont toutes la fenêtre *root* comme mère.
  - `GDK_WINDOW_CHILD` pour une fenêtre "normale" qui est une fenêtre incluse dans une autre. C'est le cas le plus général.
  - `GDK_WINDOW_DIALOG` pour une boîte de dialogue. Cette valeur est à rapprocher de `GDK_WINDOW_TOPLEVEL`. La seule différence réside dans le fait que GDK précise au *gestionnaire de fenêtres* que cette fenêtre est en réalité une boîte de dialogue afin qu'elle soit décorée différemment.
  - `GDK_WINDOW_TEMP` pour les fenêtres qui ne sont affichées qu'un court instant, comme les menus par exemple. Ces fenêtres sont des fenêtres *top-level* et l'on doit donc passer `NULL` comme premier paramètre à `gdk_window_new()`. Ces fenêtres ont toujours le paramètre `override_redirect` positionné à `TRUE` ce qui empêche le gestionnaire de fenêtres de les déplacer, les redimensionner ou les décorer.
  - `GDK_WINDOW_PIXMAP` ne doit pas être utilisé. Cette valeur est utilisée en interne par GDK pour spécifier un `GdkPixmap`. Elle est interdite pour `gdk_window_new()`. Les GdkPixmaps doivent être créés à l'aide de `gdk_pixmap_new()`.
  - `GDK_WINDOW_FOREIGN` ne peut pas non plus être utilisé. Cette valeur permet à GDK d'identifier une fenêtre qui n'a pas été créée par lui, mais dont il a pris le contrôle à l'aide de la fonction `gdk_window_foreign_new()`.
- `cursor` spécifie quel est le curseur de la souris qui doit être affiché lorsque le pointeur passe sur cette fenêtre. Les curseurs seront étudiés au paragraphe **FIXME PLEASE**.
- `wmclass_name` et `wmclass_class` n'ont de sens que lorsqu'ils sont utilisés avec des fenêtres top-level. Ils indiquent au gestionnaire de fenêtres que la fenêtre appartient à une classe de fenêtres dont le nom est `wmclass_name` et la classe est `wmclass_name`. Certains gestionnaires de fenêtres utilisent ces informations pour décorer différemment certaines fenêtres, choisir une icône particulier, etc.
- `override_redirect` permet de spécifier si une fenêtre de type top-level doit être placée, dimensionnée et décorée par le gestionnaire de fenêtres. La valeur normale est `FALSE`, ce qui signifie que le gestionnaire de fenêtres a le contrôle. Une valeur de `TRUE` indique que la fenêtre préfère se débrouiller toute seule et se placer là où elle le souhaite.

Comme nous l'avons vu, il est possible de transformer une fenêtre créée directement par la Xlib en une fenêtre GDK. Ce n'est pas une technique recommandée, mais cela peut être utile pour porter un programme initialement prévu pour la Xlib en un programme GTK+. Pour cela, il faut d'abord créer la fenêtre Xlib, puis l'encapsuler dans une `GdkWindow` à l'aide de la fonction suivante :

```
GdkWindow *gdk_window_foreign_new(guint32 fenetre);
```

Faites attention cependant. Il faut absolument inclure le fichier `gdk/gdkx.h` pour pouvoir utiliser des fonctions de la Xlib dans un programme utilisant GDK.

Lorsque l'on n'a plus besoin d'une fenêtre, on doit la détruire avec :

```
void gdk_window_destroy(GdkWindow *fenetre);
```

2.2. Les manipulations de fenêtres

Créer une fenêtre ne suffit pas à la rendre visible à l'écran. En effet, la création d'une fenêtre ne fait que réserver un espace mémoire et mettre en place quelques paramètres. Une fois créée, une fenêtre peut être affichée et cachée autant de fois qu'on le désire à l'aide des deux fonctions :

```
void gdk_window_show(GdkWindow *fenetre);
void gdk_window_hide(GdkWindow *fenetre);
```

Ceci permet de ne pas avoir à recréer sans cesse une fenêtre qui doit souvent être affichée.

La bibliothèque GDK propose deux fonctions permettant de savoir si une fenêtre donnée est visible ou non. Ces fonctions sont :

```
gboolean gdk_window_is_visible(GdkWindow *fenetre);
gboolean gdk_window_is_viewable(GdkWindow *fenetre);
```

Vous vous demandez sûrement pourquoi on a besoin de deux fonctions pour cela. En fait, la première teste si la fenêtre a été montrée avec la fonction `gdk_window_show()`. Mais cela ne suffit pas pour que cette fenêtre soit réellement affichée à l'écran. En effet, il faut aussi que toutes les fenêtres parentes jusqu'à la fenêtre top-level aient été rendues visibles par un appel à `gdk_window_show()`. En effet une fenêtre n'est visible que si toutes ses ancêtres le sont. La fonction `gdk_window_is_viewable()` teste justement qu'une fenêtre est réellement visible à l'écran en testant successivement si toutes les fenêtres parentes sont effectivement visibles.

Voici un exemple simple de création de fenêtre à l'aide du GDK :

```
/* fenêtre gdk */
#include <gdk/gdk.h>

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkWindowAttr attr;

  /* Initialisation de la bibliothèque */
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
  attr.event_mask = 0;
```

```
  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
      gdk_event_get();
}
```

Saisissez ce petit programme et compilez-le avec :

```
gcc fenetregdk.c -o fenetregdk 'gtk-config --cflags --libs'
```

Ce programme commence par initialiser la bibliothèque GDK, ce qui est une étape nécessaire. En effet, tous les programmes utilisant le GDK doivent débuter par un appel à :

```
void gdk_init(int *argc, char ***argv);
```

Cela initialise les variables internes du GDK. Les paramètres passés sont les adresses des paramètres de la fonction `main()`. GDK comprends quelques arguments, et les ôtent automatiquement de la liste des arguments `argv`. Les arguments compris par le GDK sont :

- `--gdk-debug=all` qui demande au GDK d'afficher un maximum d'informations de débogage. Si l'on désire être plus spécifique, on peut remplacer `all` par l'une des valeurs suivantes :
  - `events` pour les événements ;
  - `dnd` pour le glisser/déplacer ;
  - `color-context` pour la gestion des couleurs ;
  - `xim` pour les extensions du serveur X ;
  - `misc` pour les autres aspects.
- `--gdk-no-debug` pour supprimer l'affichage des informations de débogage.
- `--display` pour indiquer sur quel *display* le programme doit s'exécuter.
- `--sync` pour que tous les appels au serveurs soient effectués de manière synchrone. Cela ralentit notablement l'exécution du programme, mais permet souvent de déboguer plus facilement.
- `--no-xshm` pour ne pas utiliser le système de mémoire partagé du serveur X, qui est normalement utilisé pour accélérer grandement certaines opérations si le programme et le serveur X s'exécutent sur la même machine.
- `--name="nom du programme"` qui permet de changer le nom du programme, sinon ce nom est celui que l'on peut récupérer dans `argv[0]`.
- `--class="classe du programme"` qui permet de changer le nom de la classe du programme. Cette information est utilisée par certains gestionnaire

de fenêtres pour assigner une icône particulière aux programmes de certaines classes par exemple. Par défaut le nom de la classe du programme est celui présent dans `argv[0]`.

Après l'initialisation, cet exemple crée une fenêtre en initialisant le minimum d'attributs. La fenêtre créée est incapable de recevoir des événements, a une taille de 150 pixels par 150 et est une fenêtre visible top-level normale. Cette fenêtre est ensuite affichée et le programme se met en attente des éventuels événements. Nous reviendrons sur les événements dans le paragraphe **FIXME PLEASE**. Le résultat de ce programme est visible sur la figure 1. Si vous trouvez cet exemple un peu triste, c'est tout à fait normal. Nous verrons dans les paragraphes suivants qu'il est possible de changer la couleur du fond de la fenêtre et même d'y placer une image.

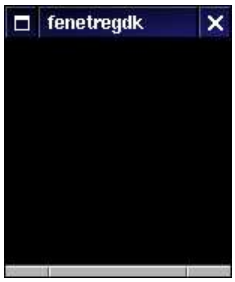


fig. 1

Il est possible de changer et de connaître la plupart des attributs d'une fenêtre après sa création à l'aide de fonctions beaucoup plus simples à utiliser que la fonction `gdk_window_new()`.

Par exemple, on peut facilement changer la taille et/ou la position d'une fenêtre à l'aide de ces trois fonctions :

```
void gdk_window_move(GdkWindow *fenetre,
                     gint x, gint y);
void gdk_window_resize(GdkWindow *fenetre,
                       gint largeur, gint hauteur);
void gdk_window_move_resize(GdkWindow *fenetre,
                            gint x, gint y,
                            gint largeur, gint hauteur);
```

dont le rôle paraît assez évident.

De la même façon, ses informations peuvent être récupérées à l'aide des fonctions suivantes :

```
void gdk_window_get_position(GdkWindow *fenetre,
                             gint *x, gint *y);
void gdk_window_get_size(GdkWindow *fenetre,
                         gint *largeur,
                         gint *hauteur);
void gdk_window_get_geometry(GdkWindow *fenetre,
```

```
                              gint *x, gint *y,
                              gint *largeur,
                              gint *hauteur,
                              gint *profondeur);
```

Une fenêtre peut même changer de parent après sa création avec :

```
void gdk_window_reparent(GdkWindow *fenetre,
                         GdkWindow *mere_adoptive,
                         gint x, gint y);
```

où `mere_adoptive` est le nouveau parent de la fenêtre qui sera placée aux coordonnées `(x, y)` dans sa nouvelle fenêtre mère. La fenêtre est automatiquement retirée de son ancien parent et placée dans le nouveau. Puisque l'on parle des fenêtres et de leur parent, sachez qu'il est possible de connaître quelle est la fenêtre mère d'une autre avec :

```
GdkWindow *gdk_window_get_parent(GdkWindow *fenetre);
```

Cette fonction retourne `NULL` si la fenêtre passée en paramètre est une fenêtre top-level.

Mais l'arbre des fenêtres n'est pas limité à une seule génération et une fenêtre peut ainsi être la fille de la fille de la fille de ... de la fille d'une fenêtre top-level. Et il est quelques fois désirable de savoir dans quelle fenêtre top-level est une fenêtre particulière perdue dans la hiérarchie. La fonction :

```
GdkWindow* gdk_window_get_toplevel(GdkWindow *fenetre);
```

permet de retrouver cette information. Ne confondez pas cette fonction avec la suivante :

```
GList *gdk_window_get_toplevels(void);
```

qui renvoie la liste de toutes les fenêtres top-level gérées par notre application.

Dans un autre ordre d'idée, une fenêtre donnée peut avoir besoin de connaître quelles sont toutes les fenêtres qu'elle possède. La fonction suivante :

```
GList* gdk_window_get_children(GdkWindow *fenetre);
```

renvoie une liste doublement chaînée de type `GSList` des enfants de la fenêtre passée en paramètre. Ceci peut être très important pour une fenêtre de type *Container*[2] qui est chargée de ranger ses enfants de manière harmonieuse.

Comme nous l'avons vu dans l'introduction de ce chapitre, il est facile de changer le titre d'une fenêtre à l'aide de la fonction :

```
void gdk_window_set_title(GdkWindow *fenetre,
                          const gchar *titre);
```

Il est facile de vérifier cet effet en plaçant la ligne :

---

[2] voir chapitre **FIXME PLEASE**

```
gdk_window_set_title(Fenetre, "coucou");
```

juste après la création de la fenêtre dans l'exemple précédent.

Comme il peut y avoir plusieurs fenêtres top-level sur l'écran, certaines sont au dessous d'autres, et donc au moins partiellement cachées. L'ordre des fenêtres top-level est géré par le gestionnaire de fenêtres, mais on peut lui demander de placer une fenêtre au dessus ou au dessous des autres à l'aides des fonctions suivantes :

```
void gdk_window_raise(GdkWindow *fenetre);
void gdk_window_lower(GdkWindow *fenetre);
```

On souhaite parfois imposer qu'une fenêtre soit toujours au dessus d'une autre, par exemple on peut vouloir qu'une boîte de dialogue reste toujours au dessus de la fenêtre principale de notre application. La fonction :

```
void gdk_window_set_transient_for(GdkWindow *dessus,
                                  GdkWindow *dessous);
```

permet cela. Faites attention à l'ordre des paramètres qui n'est pas forcément très naturel. Cette fonction n'a bien sûr d'intérêt que pour les fenêtres de type top-level. De plus si la fenêtre que l'on veut placer au dessus de la fenêtre principale est un menu[3], il vaut mieux lui enlever les décorations que le gestionnaire de fenêtres pourrait lui mettre. Ceci peut être fait au moment de la création de la fenêtre menu ou en changeant *a posteriori* l'attribut `override_redirect` à l'aide de la fonction :

```
void
gdk_window_set_override_redirect(GdkWindow *fenetre,
                                 gboolean override_redirect);
```

Les autres attributs que l'on peut changer après la création d'une fenêtre sont tout d'abord le curseur de la souris, que l'on change à l'aide de la fonction :

```
void gdk_window_set_cursor(GdkWindow *fenetre,
                           GdkCursor *curseur);
```

la colormap de la fenêtre que l'on positionne avec :

```
void gdk_window_set_colormap(GdkWindow *fenetre,
                             GdkColormap *colormap);
```

et que l'on peut récupérer avec :

```
GdkColormap *gdk_window_get_colormap(GdkWindow *fenetre);
```

et les événements que la fenêtre accepte de recevoir avec :

```
void gdk_window_set_events(GdkWindow *fenetre,
                           GdkEventMask event_mask);
```

---

[3] Les menus sont en général des fenêtres top-level et pas des fenêtres filles car il est souvent désirable qu'un menu puisse dépasser de la fenêtre s'il est trop long.

Cette liste d'événements peut être relue par la suite avec :

```
GdkEventMask gdk_window_get_events(GdkWindow *fenetre);
```

afin de rajouter un événement à cette liste sans forcément la connaître. Les événements de GDK seront étudiés au paragraphe **PLEASE FIXME**.

Il y a également deux attributs qui ne peuvent pas être changés après la création mais que l'on peut tout de même connaître. Ces attibuts sont le *visual* et le type de la fenêtre. Ces informations sont récupérables avec les fonctions suivantes :

```
GdkVisual *gdk_window_get_visual(GdkWindow *fenetre);
GdkWindowType gdk_window_get_type(GdkWindow *fenetre);
```

2.3. Interactions avec le gestionnaire de fenêtres

En règle générale, le gestionnaire de fenêtres est libre de faire ce qu'il veut avec les fenêtres top-level. Ou plus exactement, l'utilisateur est libre de choisir le gestionnaire qui lui convient le mieux et de configurer son environnement comme il le souhaite, en déplaçant telle ou telle fenêtre, en la redimensionnant, en lui affectant une icône particulière, etc.

Cependant les programmes peuvent donner des indications au gestionnaire de fenêtres sur la façon dont ils souhaitent que leurs fenêtres soient gérées. Rappelez-vous bien que ce ne sont que des indications et que seuls les gestionnaires de fenêtres les plus coopératifs honoreront toutes ces indications. Heureusement, ce sont les plus nombreux.

Tout d'abord, notre application peut demander qu'une fenêtre ait des contraintes au niveau de sa taille minimale et maximale et de sa position à l'aide de la fonction :

```
gdk_window_set_hints(GdkWindow *fenetre,
                     gint x, gint y,
                     gint largeur_min,
                     gint hauteur_min,
                     gint largeur_max,
                     gint hauteur_max,
                     gint drapeaux);
```

où `drapeaux` est un champs de bits qui indique quelles sont les informations que l'on souhaite indiquer. Ainsi, le bit `GDK_HINT_POS` doit être positionné si l'on indique la position souhaitée dans `x` et `y`, le bit `GDK_HINT_MIN_SIZE` signifie que l'on désire fixer la taille minimale de notre fenêtre à `largeur_min` et `hauteur_min`, et le bit `GDK_HINT_MAX_SIZE` est positionné si l'on veut donner une taille maximale à notre fenêtre dans `largeur_max` et `hauteur_max`. Attention, cependant. La taille par défaut de notre fenêtre demeure inchangée. Simplement l'utilisateur ne pourra pas dimensionner la fenêtre à une taille qui ne respecterait pas les limites que l'on a imposées.

La fonction `gdk_window_set_hints()` est en fait une version simplifiée d'une fonction bien plus générale qui permet de contrôler comment une fenêtre doit être redimensionnée :

```
void gdk_window_set_geometry_hints(GdkWindow *fenetre,
                                   GdkGeometry *geometrie,
                                   GdkWindowHints masque);
```

Cette fonction fonctionne un peu comme `gdk_window_new()`. C'est-à-dire qu'elle prend comme deuxième et troisième paramètres un pointeur sur une structure (`geometrie`) et un champ de bits (`masque`) définissant quels sont les champs de la structure qui doivent être pris en compte. La structure `GdkGeometry` est définie ainsi :

```
typedef struct _GdkGeometry  GdkGeometry;

struct _GdkGeometry
{
  gint min_width;
  gint min_height;
  gint max_width;
  gint max_height;
  gint base_width;
  gint base_height;
  gint width_inc;
  gint height_inc;
  gdouble min_aspect;
  gdouble max_aspect;
};
```

où les champs sont regroupés deux par deux :

- `min_width` et `min_height` spécifient la largeur et la hauteur minimale que doit avoir notre fenêtre. Ces champs ne sont pris en compte que si le bit `GDK_HINT_MIN_SIZE` est inclus dans le champ de bit `geom_masque`.
- `max_width` et `max_height` spécifient la largeur et la hauteur maximale que doit avoir notre fenêtre. Ces champs ne sont pris en compte que si le bit `GDK_HINT_MAX_SIZE` est inclus dans le champ de bit `geom_masque`.
- `base_width` et `base_height` spécifient la largeur et la hauteur par défaut que l'on souhaite pour notre fenêtre. Ces champs ne sont pris en compte que si le bit `GDK_HINT_BASE_SIZE` est inclus dans `geom_masque`.
- `width_inc` et `height_inc` spécifient les granularités horizontale et verticale souhaitées lors de l'agrandissement ou la réduction d'une fenêtre. Cela signifie que la fenêtre ne pourra être réduite ou agrandie que d'un multiple entier de `width_inc` en largeur et de `height_inc` en hauteur. Par exemple, si une fenêtre a une largeur originelle de 100 pixels et que `width_inc` vaut 10, alors la fenêtre ne pourra avoir que des largeurs de 90, 100, 110, 120, etc. Cela peut être utile par exemple si notre fenêtre contient du texte ; la valeur donnée à ces deux

champs est alors la largeur et la hauteur d'un caractère. On a ainsi toujours un nombre entier de caractères affichés. Le programme `xterm` fonctionne de cette façon. Essayez de le redimensionner afin de comprendre l'effet de ces champs. Ces champs ne sont pris en compte que si le bit `GDK_HINT_RESIZE_INC` est présent dans `geom_masque`.

- `min_aspect` et `max_aspect` représentent le rapport largeur/hauteur minimal et maximal de la fenêtre. Par exemple, si vous désirez qu'une fenêtre soit toujours parfaitement carrée, il suffit de placer la valeur 1.0 dans ces deux champs. Ces champs ne sont pris en compte que si le bit `GDK_HINT_ASPECT` est positionné.

Une autre opération fréquente avec les gestionnaires de fenêtres est la mise en icône d'une fenêtre. Le GDK permet de donner un nom particulier à l'icône associé à une fenêtre avec :

```
void gdk_window_set_icon_name(GdkWindow *fenetre,
                              gchar *nom);
```

dont les paramètres doivent être évident. Nous verrons bientôt qu'il est aussi possible de spécifier une image à utiliser pour l'icône.

Le rôle du gestionnaire de fenêtres ne se limite pas à permettre de déplacer, redimensionner et icônifier des fenêtres, il les décore aussi en plaçant un cadre et différents boutons autour des fenêtres top-level. Le nombre et le rôle de ces boutons varient beaucoup d'un gestionnaire à l'autre, mais la plupart placent des décorations classiques. On peut choisir quelles sont les décorations qui doivent entourer une fenêtre en particulier avec la fonction :

```
void gdk_window_set_decorations(GdkWindow *fenetre,
                                GdkWMDecoration decorations);
```

où `decorations` est un champ de bits pouvant contenir les bits suivants :

- `GDK_DECOR_ALL` qui indique que la fenêtre doit être décoré avec le maximum de décorations.
- `GDK_DECOR_BORDER` indique que l'on désire qu'un cadre soit déssiné autour de la fenêtre.
- `GDK_DECOR_RESIZEH` indique que l'on souhaite que le bouton permettant le redimensionnement de la fenêtre, s'il existe, soit affiché.
- `GDK_DECOR_TITLE` indique que l'on souhaite que notre fenêtre soit munie d'une barre de titre.
- `GDK_DECOR_MENU` indique qu'un menu doit être attaché à la fenêtre, permettant la plupart des opérations sur la fenêtre.
- `GDK_DECOR_MINIMIZE` indique que le bouton permettant l'icônification de la fenêtre doit être présent.
- `GDK_DECOR_MAXIMIZE` indique que le bouton permettant de rendre la fenêtre plein écran doit être présent.

On n'utilise cette fonction qu'assez rarement car le type de la fenêtre indiqué dans le champ `window_type` lors de sa création suffit généralement à indiquer quelles sont les décorations souhaitables pour notre fenêtre.

La dernière fonction concernant l'interaction entre les `GdkWindows` et le gestionnaire de fenêtres est la suivante :

```
void gdk_window_set_group(GdkWindow *fenetre,
                          GdkWindow *leader);
```

Elle permet de désigner une fenêtre particulière comme étant la principale (`leader`) de notre application et les autres fenêtres peuvent ainsi être attachées au groupe de fenêtres dirigées par la fenêtre `leader`. Ainsi, lorsque l'utilisateur iconifie la fenêtre principale, toutes les autres fenêtres du groupe le sont également.

2.4. Les autres fonctions concernant les GdkWindows

Il existe quelques autres fonctions qui s'appliquent aux `GdkWindows` mais qui n'ont aucun rapport avec le gestionnaire de fenêtres. Ce paragraphe présente les plus intéressantes.

On a souvent besoin de stocker une donnée qui a un rapport avec une fenêtre mais on ne veut pas utiliser une variable globale pour cela. Le GDK nous permet d'associer cette donnée à une fenêtre grâce à la fonction :

```
void gdk_window_set_user_data(GdkWindow *fenetre,
                              gpointer donnee);
```

où `donnee` est de type `gpointer`, c'est-à-dire un pointeur générique. Notre donnée peut donc être n'importe quoi.

Ce pointeur pourra être récupéré plus tard avec :

```
void gdk_window_get_user_data(GdkWindow *fenetre,
                              gpointer *donnee);
```

Ce genre d'astuce est bien pratique par exemple pour le traitement des boîtes de dialogue mais nous verrons que le GTK possède des mécanismes encore plus puissants et plus souples pour cela.

Indépendamment de la gestion des événements, on peut vouloir connaître la position de la souris par rapport à une fenêtre donnée. La fonction :

```
GdkWindow *gdk_window_get_pointer(GdkWindow *fenetre,
                                  gint *x, gint *y,
                                  GdkModifierType *masque);
```

renvoie ce renseignement et plus encore. La position de la souris par rapport à la fenêtre passée en paramètre est retournée dans les paramètres `x` et `y`. La fonction retourne la fenêtre qui contient le pointeur de la souris si la souris est bien dans une de nos fenêtres. Le paramètre `masque` contient, au retour de la fonction, un champ de bits décrivant quels sont, parmis l'ensemble des touches modifiantes du clavier ainsi que des boutons de la souris, ceux qui sont enfoncés.

Les noms symboliques des bits sont les suivants :

- `GDK_SHIFT_MASK` correspond à l'appui sur l'une des touches *Shift* ;
- `GDK_LOCK_MASK` est positionné si la touche *Caps Lock* est active ;

- `GDK_CONTROL_MASK` correspond à l'appui sur l'une des touches *control* ;
- `GDK_MOD1_MASK` correspond à la touche *Alt* sur les claviers français ;
- `GDK_MOD2_MASK` correspond à la touche *AltGr* sur les claviers français ;
- `GDK_MOD3_MASK` correspond au premier modifieur supplémentaire du clavier, il peut s'agir de la touche *Windows* de certains claviers ;
- `GDK_MOD4_MASK` correspond au deuxième modifieur supplémentaire du clavier, il peut s'agir de la touche *Menu* de certains claviers ;
- `GDK_MOD5_MASK` correspond au troisième modifieur supplémentaire du clavier ;
- `GDK_BUTTON1_MASK` correspond au bouton gauche de la souris ;
- `GDK_BUTTON2_MASK` correspond au bouton du milieu de la souris ;
- `GDK_BUTTON3_MASK` correspond au bouton droit de la souris ;
- `GDK_BUTTON4_MASK` correspond au premier bouton supplémentaire de la souris ;
- `GDK_BUTTON5_MASK` correspond au second bouton supplémentaire de la souris.

Si, par exemple, au moment de l'appel à cette fonction, l'utilateur pressait le bouton gauche de la souris et une touche shift, alors la variable `*masque` contiendrait la valeur `GDK_SHIFT_MASK | GDK_BUTTON1_MASK` à l'issu de l'appel à cette fonction.

Si vous n'avez pas besoin d'autant d'informations, vous pouvez utiliser une version allégée de cette fonction :

```
GdkWindow *gdk_window_at_pointer(gint *win_x, gint *win_y);
```

qui renvoie la fenêtre qui contient la souris et les coordonnées `x` et `y` de la souris à l'intérieur de cette fenêtre. Notez que si la souris n'est à l'intérieur d'aucune de nos fenêtres, la valeur renvoyée est `NULL` et `x` et `y` contiennent la valeur -1.

3. Les Pixmaps et les Bitmaps

Les pixmaps ressemblent beaucoup aux fenêtres si ce n'est qu'ils ne peuvent pas être affichés et en réalité le GDK traite les deux un peu de la même façon. Ce sont en fait des images qui ont une taille et un certain nombre de couleurs. Ces couleurs définissent un certain nombre de *bit-plan*[4] qui correspond à la profondeur de l'image. Ainsi, une image en noir et blanc sera composée d'un seul bit-plan et aura donc une profondeur de 1 ; de la même façon, une image en 256 couleurs sera composée de 8 bit-plan (car $2^8 = 256$) et aura une profondeur de 8. Les pixmaps ayant une profondeur de 1 sont appelées *Bitmap* suivant la terminologie de X-Window. Attention, seul un petit nombre de profondeurs sont autorisées. Cela dépend du *visual* utilisé.

De plus, les pixmaps sont souvent utilisés pour être copiés vers une fenêtre, mais cette opération n'est possible que si les deux ont la même profondeur.

---

[4] voir le paragraphe **FIXME PLEASE** à ce sujet

3.1. La créations des pixmaps

Il existe plusieurs façons de créer un `GdkPixmap`. La plus spartiate est la suivante :

```
GdkPixmap *gdk_pixmap_new(GdkWindow *fenetre,
                          gint largeur,
                          gint hauteur,
                          gint profondeur);
```

Cette fontion crée un pixmap de la taille et de la profondeur indiquée. Le paramètre `fenetre` est une fenêtre préalablement créée et dont le pixmap héritera quelques paramètres internes. Pour être sûr que la profondeur sera acceptée, on peut récupérer la profondeur d'une fenêtre préexistante avec :

```
profondeur = gdk_window_get_visual(fenetre)->depth;
```

car la profondeur est un des aspect des visuals. On est ainsi sûr que l'on pourra copier le pixmap (ou une partie de celui-ci) dans la fenêtre et réciproquement par la suite.

Cependant, la fonction `gdk_pixmap_new()` crée un pixmap non initialisé. Cela peut être ce que l'on souhaite, par exemple si l'on veut dessiner dedans par la suite, mais les pixmaps sont plutôt créés pour contenir une image dès le départ.

Ainsi, on peut directement créer un bitmap à partir d'un fichier au format `.xbm` à l'aide de la fonction :

```
GdkPixmap *gdk_bitmap_create_from_data(GdkWindow *window,
                                       const gchar *donnees,
                                       gint largeur,
                                       gint hauteur);
```

Ceci crée un bitmap (c'est-à-dire un pixmap en noir et blanc). De la hauteur et de la largeur souhaitée. Le paramètre `donnees` est un pointeur sur un tableau de données de types `char`, comme défini dans un fichier `.xbm`. Voici par exemple le contenu du fichier icone.xbm :

```
#define icone_width 32
#define icone_height 32
static char icone_bits[] =
{
  0xf0,0xff,0x3f,0x00,0x10,0x00,0x60,0x00,
  0x10,0x38,0xa0,0x00,0x10,0xee,0x20,0x01,
  0x10,0x92,0x20,0x02,0x10,0xbb,0x21,0x04,
  0x10,0x09,0x21,0x08,0x10,0x93,0xe1,0x1f,
  0x10,0x82,0x07,0x18,0x10,0xee,0x25,0x18,
  0x10,0xb8,0x58,0x18,0x10,0x08,0xc0,0x18,
  0x10,0x10,0x47,0x18,0x10,0x90,0x4a,0x18,
  0x10,0x4c,0x91,0x19,0x10,0xc4,0x10,0x19,
  0x10,0x5c,0xd1,0x19,0x10,0x90,0x4a,0x18,
  0x10,0x10,0x47,0x18,0x10,0x08,0x80,0x18,
```

```
  0x10,0xd8,0x7c,0x18,0x10,0xee,0x24,0x18,
  0x10,0x92,0x07,0x18,0x10,0xbb,0x01,0x18,
  0x10,0x09,0x01,0x18,0x10,0x93,0x01,0x18,
  0x10,0x82,0x00,0x18,0x10,0xee,0x00,0x18,
  0x10,0x38,0x00,0x18,0x10,0x00,0x00,0x18,
  0xf0,0xff,0xff,0x1f,0xe0,0xff,0xff,0x1f
};
```

On peut donc créer un bitmap contenant ces données avec le bout de programme suivant :

```
#include "icone.xbm"
[ ... ]
void CreationDePixmap(void)
{
  pixmap = gdk_bitmap_create_from_data(fenetre,
                                       icone_bits,
                                       icone_width,
                                       icone_height);
}
```

Il est aussi possible de créer un pixmap ayant une profondeur différente de 1, à partir du même fichier `.xbm`. Dans ce cas, il faut préciser quelles doivent être les couleurs à employer pour l'avant plan et l'arrière plan. La fonction à utiliser est la suivante :

```
GdkPixmap *
gdk_pixmap_create_from_data(GdkWindow *window,
                            const gchar *donnees,
                            gint largeur,
                            gint hauteur,
                            gint profondeur,
                            GdkColor *avantplan,
                            GdkColor *arriereplan);
```

Par exemple, on peut créer un pixmap en bleu et rouge de la profondeur courante à l'aide du fichier icone.xbm précédant avec l'extrait de programme suivant :

```
#include "icone.xbm"
[ ... ]
pixmap = gdk_pixmap_create_from_data(fenetre,
                 icone_bits,
                 icone_width, icone_height,
                 gdk_window_get_visual(fenetre)->depth,
                 bleu,
                 rouge);
```

En supposant que les couleurs `bleu` et `rouge` aient été créées comme on le verra au paragraphe **PLEASE FIXME**.

Cependant le format `.xbm` tend à disparaître car les images en noir et blanc se font de plus en plus rares.

Le format `.xpm` est beaucoup plus puissant puisqu'il permet un nombre de couleur quasi illimité et permet même de définir une couleur comme transparente. Aussi, on préférera souvent créer des pixmaps à l'aide de la fonction suivante :

```
GdkPixmap *
gdk_pixmap_create_from_xpm(GdkWindow *fenetre,
                           GdkBitmap **masque,
                           GdkColor *transparent,
                           const gchar *NomFichier);
```

qui crée en fait deux pixmaps. Le premier est celui qui est renvoyé par la fonction et le second est un masque. Le masque est un bitmap de la même taille que le pixmap renvoyé où les pixels blancs sont ceux qui sont transparents dans le fichier `.xpm` dont le nom est passé dans le paramètre `NomFichier`. Au retour de cette fonction le paramètre `transparent` contient la couleur qui a été choisie pour représenter les pixels transparents du fichier de départ.

Pour allouer des couleurs, la fonction `gdk_pixmap_create_from_xpm()` utilise la colormap[5] de la fenêtre passée en paramètre. Si l'on désire utiliser une colormap différente, on peut utiliser la fonction :

```
GdkPixmap *
gdk_pixmap_colormap_create_from_xpm(GdkWindow *fenetre,
                                    GdkColormap *colormap,
                                    GdkBitmap **masque,
                                    GdkColor *transparent,
                                    const gchar *NomFichier);
```

qui est strictement équivalente à la précédente si ce n'est que la colormap désirée est passée en deuxième paramètre.

C'est effectivement pratique de pouvoir charger un fichier image directement dans un pixmap, mais cela suppose que l'on sache exactement comment se nomme le fichier et où il se trouve. C'est pourquoi il existe deux fonctions complémentaires des précédentes permettant d'inclure le fichier directement dans le code source. Ces fonctions sont les suivantes :

```
GdkPixmap *
gdk_pixmap_create_from_xpm_d(GdkWindow *fenetre,
                             GdkBitmap **masque,
                             GdkColor *transparent,
                             gchar **donnees);
GdkPixmap *
gdk_pixmap_colormap_create_from_xpm_d(GdkWindow *fenetre,
```

[5] voir le paragraphe sur la gestion des couleurs

```
                                        GdkColormap *colormap,
                                        GdkBitmap **masque,
                                        GdkColor *transparent,
                                        gchar **donnees);
```

Voyons un exemple d'utilisation de ces fonctions. Imaginons que vous possédiez un fichier icone.xpm contenant ceci :

```
/* XPM */
static char * icone_xpm[]={
/* width height num_colors chars_per_pixel */
"32 32 9 1",
/* colors */
"  s None c None",
". c black",
"X c #008080",
"o c cyan",
"O c white",
"+ c yellow",
"@ c SlateGray",
"# c yellow4",
"$ c #c0c0c0",
/* pixels */
"                      .....       ",
"                     .XXXXX.      ",
"                    .XXXXXXX.     ",
"                    .oXXXXXX.     ",
"            ..      .XoooooX.     ",
"          .O+..      .XXXXX.      ",
"          .+O+O..     .....       ",
"         .+O+O+O+....XXX..        ",
"         .O+O+O+O+.X.oXX.X.       ",
"        .O+O+O+O+.oX.oXX.XX.      ",
"        .+O+O+O+O.oXX...XXX.      ",
"       .+O+O+O+O+.XoXXXXXoX.      ",
"       .O+O+O+O+O.XXoooooXX..     ",
"      .O+O+O+O+O+O.XXXXXXX.@#..   ",
"      .+O+O+O+O+O+O.......@###.   ",
"     .+O+O+O+O+O+O+O+O.@#@#@#.    ",
"     .O+O+O+O+O+O+O+O+.#@#@##.    ",
"    .O+O+O+O+O+O+O+O+O#@#@##.     ",
"    .+O+O+O+O+O+O+O+O+O#@###.     ",
"   .+O+O+O+O+O+O+O+O+O+O+O#.      ",
"  .+O+O+O+O+O+O+O+O+O+O+O##.      ",
"...+O+O+O+O+O+O+O+O+O+O+O+#.      ",
".O+O+O+O+O+O+O+O+O+O+O+O+##.      ",
```

```
" .O+O+O+O+O+O+O+O+O+O+O+O#.     ",
"  .$+O+O+O+O+O+O+O+O+O+O##.     ",
"   ..+O+O+O+O+O+O+O+O+O+#.      ",
"     ..O+O+O+O+O+O+O+O+#..      ",
"       ..+O+O+O+O+O+O@$..       ",
"         ..O+O+O+.......        ",
"           ..+O+@.OO+$.         ",
"             ....$+$..          ",
"                .....           "};
```

L'extrait de programme suivant permet de créer un `GdkPixmap` à partir de ce fichier :

```
#include "icone.xpm"
[ ... ]
pixmap = gdk_pixmap_create_from_xpm_d(fenetre,
                                      &masque,
                                      &transparent,
                                      icone_xpm);
```

3.2. Les utilisations des pixmaps

Les pixmaps (et les bitmaps) ont beaucoup de différentes applications. Parmi celles-ci, l'une est de pouvoir servir de fond à une fenêtre créée par GDK. Ceci se réalise grâce à la fonction suivante :

```
void gdk_window_set_back_pixmap(GdkWindow *fenetre,
                                GdkPixmap *pixmap,
                                gint copie);
```

où `fenetre` est la fenêtre dont on veut changer le fond, `pixmap` est le pixmap devant servir de fond à la fenêtre et `copie` s'il est non nul, demande de ne pas prendre le paramètre `pixmap` en compte et de copier le fond de la fenêtre parente. Ce qui est pratique pour simuler des fenêtres partiellement transparentes.

Une autre utilisation fréquente des pixmaps est la création d'une icône pour les fenêtres. Ceci est réalisé à l'aide de la fonction suivante :

```
void gdk_window_set_icon(GdkWindow *fenetre,
                         GdkWindow *icone,
                         GdkPixmap *pixmap,
                         GdkBitmap *masque);
```

où `fenetre` est bien évidemment la fenêtre à laquelle on veut donner une icône particulière, `icone` est une fenêtre où l'on désire que l'icône soit placée. En géneral, il vaut mieux passer la valeur `NULL` à la place, afin que X-Window crée lui-même la fenêtre dont il a besoin. `pixmap` et `masque` sont les deux pixmaps que peut renvoyer une fonction comme `gdk_pixmap_create_from_xpm_d()`. `pixmap` représente l'icône proprement dite et `masque` est un bitmap servant à créer la forme de l'icône.

Voici un programme dont le résultat est visible sur la figure 2 qui reprend plusieurs des utilisation des pixmaps :

```
/* pixmap gdk */
#include <gdk/gdk.h>
#include "icone.xpm"

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkPixmap *Pixmap;
  GdkBitmap *Masque;
  GdkColor   Transparent;
  GdkWindowAttr attr;

  attr.event_mask = 0;
  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Création du pixmap */
  Pixmap = gdk_pixmap_create_from_xpm_d(Fenetre,
                                        &Masque,
                                        &Transparent,
                                        icone_xpm);
  /* on place le pixmap dans le fond de la fenêtre */
  gdk_window_set_back_pixmap(Fenetre, Pixmap, FALSE);
  /* Création de l'icône de la fenêtre */
  gdk_window_set_icon(Fenetre, NULL, Pixmap, Masque);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
      gdk_event_get();
}
```

Essayez d'icônifier la fenêtre afin de vérifier que l'icône est bien celle choisie.

fig. 2

Je vous ai dit au début du paragraphe sur les fenêtres qu'elles étaient des zones rectangulaires de l'écran. En fait ce n'est pas tout à fait vrai. Plus exactement, elles sont toujours rectangulaires, mais n'en ont pas toujours l'air. En effet il est possible de rendre certaines parties d'une fenêtre transparente[6]. L'une des applications les plus connues utilisant cette fonctionnalité, est sans doute le très célèbre `xeyes`. Pour ce faire, on doit d'abord construire un masque qui est un `GtkBitmap` où les points laissés en blanc seront transparents dans la fenêtre et les points mis en noir seront opaques. Une fois ce masque créé, par exemple à l'aide d'une des fonctions présentées dans le paragraphe précédent, on appelle la fonction :

```
void gdk_window_shape_combine_mask(GdkWindow *fenetre,
                                   GdkBitmap *masque,
                                   gint x, gint y);
```

où `x` et `y` sont un offset qui sera appliqué au masque avant de l'appliquer. Pour voir l'effet de cette fonction, ajoutez la ligne suivante :

```
gdk_window_shape_combine_mask(Fenetre, Masque, 0, 0);
```

dans l'exemple précédent juste avant l'affichage de la fenêtre. La grande majorité de la fenêtre devient transparente et le cadre autour de la fenêtre semble flotter.

Quand on utilise des fenêtres transparentes ou non-rectangulaires, il faut aussi penser que si des fenêtres filles sont dans la fenêtre à un endroit où celle-ci est transparente, elles seront elles aussi transparentes à cet endroit. Rappelez vous en effet que la partie visible d'une fenêtre fille est la partie commune entre la fenêtre mère et la fenêtre fille. Pour résoudre ce problème, il suffit de le prendre dans l'autre sens. C'est-à-dire que l'on crée d'abord les fenêtres filles, qui peuvent être rectangulaires ou non, puis on calcule la forme de la fenêtre mère à l'aide de la fonction :

```
void gdk_window_set_child_shapes(GdkWindow *Fenetre);
```

qui s'occupe de tout.

---

[6] si votre serveur X supporte cette extension, ce qui est très souvent le cas

Voici un exemple d'utilisation de cette fonction très pratique :

```
/* shape gdk */
#include <gdk/gdk.h>

int main(int argc, char *argv[])
{
  GdkWindow *FenetreMere, *FenetreFille;
  GdkWindowAttr attr;

  attr.event_mask = 0;
  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre mère et affichage */
  FenetreMere = gdk_window_new(NULL, &attr, 0);
  gdk_window_show(FenetreMere);
  /* Création des fenêtres filles */
  attr.width = 50;
  attr.height = 50;
  attr.window_type = GDK_WINDOW_CHILD;
  FenetreFille = gdk_window_new(FenetreMere,
                                &attr, 0);
  gdk_window_show(FenetreFille);
  attr.x = 100;
  attr.y = 100;
  FenetreFille = gdk_window_new(FenetreMere, &attr,
                                GDK_WA_X | GDK_WA_Y);
  gdk_window_show(FenetreFille);
  gdk_window_set_child_shapes(FenetreMere);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
      gdk_event_get();
}
```

De plus, si la fenêtre mère possède déjà une forme propre non rectangulaire, et que l'on désire que les formes des fenêtres filles soient ajoutées à celle de la fenêtre mère, on peut utiliser de la même façon la fonction suvante :

```
void gdk_window_merge_child_shapes(GdkWindow *fenetre);
```

Ces fenêtres non rectangulaires sont un tout petit peu difficiles à mettre en œuvre, mais elle peut rendre une interface très attractive. Mais faites aussi attention

à ne pas en abuser, car une interface où toutes les fenêtres ont des formes bizarres peut facilement dérouter l'utilisateur non averti.

4. La gestion des couleurs, les visuals, les colormaps

Une des grandes puissances du X Window System est qu'il est disponible sur un grand nombre de plateformes et qu'il supporte beaucoup de cartes graphiques différentes. Le nombre de couleurs que ces cartes peuvent afficher varie de 2 (cartes monochromes) à près de 17 millions[7]. Ce nombre défini la *profondeur* maximale que peut avoir une fenêtre ou un pixmap sur un serveur donné. Par exemple un serveur capable d'afficher 65.536 couleurs sera capable de supporter des profondeurs de au plus 16 car 16 bits sont suffisants pour représenter toutes les valeurs de 0 à 65.535. De plus la plupart des serveurs 8 bits affichent 256 couleurs choisies parmi beaucoup plus. Il y a donc un nombre de configurations assez impressionnant. Ceci est sans doute très agréable pour l'utilisateur mais peut facilement devenir un enfer pour le programmeur.

X-Window propose une solution originale pour gérer tout ça. Et le GDK encapsule bien sûr tout cela. Pour X-Window, un pixel est un nombre entier dont la taille dépend du nombre de couleurs que le serveur peut afficher. Ce nombre sert en fait d'index dans un tableau où l'on peut trouver les composantes rouges, vertes et bleues du pixel en question. Un tel tableau est appelé *colormap*. Les colormaps peuvent avoir un format différent suivant les capacités du serveur. Ainsi, sur un serveur monochrome, les colormaps sont des tableaux de deux éléments. Suivant les cas, les colormaps peuvent être statiques, c'est-à-dire que le tableau est constitué une fois pour toute par le serveur. Ou dynamique, c'est-à-dire que le tableau est rempli petit à petit à chaque fois que l'on a besoin d'une nouvelle couleur. La forme, la taille et le comportement des colormaps sont régis par une structure que l'on appelle un *visual*. Les fenêtres et les pixmaps de GDK (et donc de X-Window) ne peuvent avoir qu'un visual et une colormap à la fois, mais rien n'empêche le serveur d'en avoir plus.

4.1. Les GdkVisuals

Lors de son initialisation, GDK récupère la liste des visuals supportés par le serveur. Et il en choisi un qui lui parait le plus puissant. Ce visual particulier peut être récupéré par la fonction :

```
GdkVisual *gdk_visual_get_system(void);
```

Ce visual pourra être utilisé en général partout où on a besoin d'un visual. Pour savoir si le visual est adapté à nos besoins, seuls deux des champs de la structure `GdkVisual` sont intéressants. Le champ `depth` définit la profondeur gérée par ce visual (et donc le nombre de couleurs disponibles dans les colormaps correspondantes).

L'autre champ intéressant est le champ `type` qui définit le type du visual. Il existe 6 types de visuals différents :

– `GDK_VISUAL_STATIC_GRAY`, cela signifie que le serveur est monochrome ou en dégradé de gris. Dans ce cas la colormap est unique et ne peut être changée.

[7] 16.777.216 pour être précis...

Un pixel de X-Window représente directement l'intensité lumineuse du pixel à l'écran.

- `GDK_VISUAL_GRAYSCALE`, le serveur ne peut afficher que des niveaux de gris, mais la colormap est dynamique et peut donc être changée. Chaque fenêtre peut donc déclarer sa propre colormap et n'utiliser que les niveaux de gris dont elle a besoin.
- `GDK_VISUAL_STATIC_COLOR`, le serveur supporte la couleur mais ne peut avoir qu'une seule colormap qui contient toutes les couleurs affichable. Ce visual n'existe que sur des serveurs comportants très peu de couleurs (moins de 10.000) sinon la colormap occuperait trop de mémoire.
- `GDK_VISUAL_PSEUDO_COLOR` est le type de visual utilisé par les serveurs qui ne peuvent afficher que peu de couleurs parmi une palette beaucoup plus grande. Par exemple, il y a quelques années, la plupart des cartes graphiques affichaient 256 couleurs choisies parmi 17 millions. Les colormaps associées à ce type de visual ont donc 256 entrées, chacune représentant une couleur.
- `GDK_VISUAL_TRUE_COLOR`, le serveur est en couleur et possède en fait trois colormaps – une pour le rouge, une pour le vert et une pour le bleu – qui sont fixées une fois pour toutes. Dans ce cas un pixel X-Window contient en fait trois index, un pour chaque colormap. Pour ce type de visual, chaque colormap a en général 256[8] entrées ce qui permet d'afficher $256^3 = 16.777.216$ couleurs à la fois. C'est le type de visual le plus souple et le plus répandu de nos jours.
- `GDK_VISUAL_DIRECT_COLOR` ressemble beaucoup au précédent si ce n'est que les colormaps sont vides au départ et peuvent être différentes d'une fenêtre à l'autre. Du point de vue de GDK, ces deux types sont complètement équivalents.

Si le type du visual par défaut ne vous convient pas, vous pouvez essayer d'en savoir plus sur les visuals disponibles sur le serveur.

Tout d'abord, vous pouvez connaître la profondeur maximale des colormaps que le serveur sait gérer avec :

```
gint gdk_visual_get_best_depth(void);
```

Vous pouvez en déduire le nombre de couleurs affichables simultanément avec la formule suivante :

$$\text{nombre de couleurs} = 2^{\text{profondeur}}$$

De même, vous pouvez connaître quel est le meilleur type de visual supporté par le serveur avec :

```
GdkVisualType gdk_visual_get_best_type(void);
```

où la valeur renvoyée est l'un des 6 types de visual.

Ensuite, vous pouvez trouver quel est le meilleur visual ayant une profondeur donnée, un type donné, ou les deux avec les fonctions :

```
GdkVisual *
```

---

[8] mais cela peut être beaucoup plus comme 65.536 par exemple.

```
gdk_visual_get_best_with_depth(gint profondeur);
GdkVisual *
gdk_visual_get_best_with_type(GdkVisualType visual_type);
GdkVisual *
gdk_visual_get_best_with_both(gint profondeur,
                              GdkVisualType visual_type);
```

Enfin, si vous avez toujours du mal à faire votre choix, vous pouvez obtenir la liste des profondeurs disponibles avec :

```
void gdk_query_depths(gint **profondeurs, gint  *nombre);
```

qui renvoie le nombre de profondeurs différentes supportées par le serveur dans le paramètre `nombre` et un tableau d'entier contenant les profondeurs disponibles.

De la même façon, la liste des types de visual supportés peut être récupérée avec :

```
void gdk_query_visual_types(GdkVisualType **visual_types,
                            gint *nombre);
```

Enfin il est possible d'obtenir une liste doublement chaînée de tous les visuals supportés par le serveur avec :

```
GList *gdk_list_visuals(void);
```

Voici un exemple qui utilise les trois dernières fonctions présentées afin de connaître un peu plus les capacités du serveur :

```
/* visual gdk */
#include <stdio.h>
#include <gdk/gdk.h>

static const gchar* nom_des_visuals[] =
{
  "static gray",
  "grayscale",
  "static color",
  "pseudo color",
  "true color",
  "direct color"
};

void AfficheVisual(GdkVisual *visual)
{
  printf("  Type : %s, profondeur : %d\n",
         nom_des_visuals[visual->type],
         visual->depth);
}
```

```
int main(int argc, char *argv[])
{
  int *profondeurs;
  GdkVisualType *types;
  int n, i;
  GList *visuals;

  gdk_init(&argc, &argv);
  /* Les profondeurs */
  gdk_query_depths(&profondeurs, &n);
  printf("Profondeurs disponibles :");
  for (i=0 ; i<n-1 ; i++)
    printf(" %d,", profondeurs[i]);
  printf(" %d\n", profondeurs[i]);
  /* Les types de visuals */
  gdk_query_visual_types(&types, &n);
  printf("Types de visuals disponibles :");
  for (i=0 ; i<n-1 ; i++)
    printf(" %s,", nom_des_visuals[types[i]]);
  printf(" %s\n", nom_des_visuals[types[i]]);
  /* Les visuals */
  printf("Visuals disponibles :\n");
  visuals = gdk_list_visuals();
  g_list_foreach(visuals, (GFunc)AfficheVisual, NULL);
  return 0;
}
```

4.2. Les colormaps et l'allocation des couleurs

Bien, nous avons à présent un visual avec lequel travailler. D'après la profondeur de celui-ci, nous pouvons même savoir combien de couleurs le serveur est capable d'afficher. Cependant, comme nous l'avons vu, ces couleurs doivent appartenir à une colormap avant de pouvoir être utilisées. La colormap sera un tableau (ou 3 tableaux) contenant la relation entre un numéro et une couleur réelle.

Comme pour les visuals, le plus facile est de récupérer la colormap du système avec :

```
GdkColormap *gdk_colormap_get_system(void);
```

Mais il peut arriver que l'on ait besoin d'en déclarer une nouvelle, par exemple parce que le serveur ne peut afficher qu'un nombre restreint de couleurs et que l'on désire que chacune de nos fenêtres affiche beaucoup de couleurs. Pour se faire, on peut appeler la fonction :

```
GdkColormap *gdk_colormap_new(GdkVisual *visual,
                              gint privee);
```

où `visual` est un visual qui a pu être obtenu avec l'un des fonctions présentées dans le paragraphe précédent. Le paramètre `privee` indique si l'on accepte que les couleurs présentes dans la colormap créée soit réutilisées par la suite (`privee` vaut alors `FALSE`) on non (`privee` vaut alors `TRUE`).

Notez qu'une colormap est toujours attachée à un visual et un seul. Ce visual peut être obtenu grâce à :

```
GdkVisual *gdk_colormap_get_visual(GdkColormap *colormap);
```

Une fois la colormap créée (ou récupérée), on peut commencer à la remplir avec des couleurs. Mais attention, X-Window ne nous laisse pas faire vraiment ce que l'on veut avec les couleurs et il n'est pas possible de remplir la colormap sans son consentement. Normalement le processus d'allocation est extrêmement compliqué car il dépend du visual utilisé. Le GDK cache une bonne partie de cette complexité.

Du point de vue de GDK, une couleur est simplement une structure `GdkColor`, identique quels que soit le visual et la colormap utilisés. Cette structure est définie comme suit :

```
typedef struct __GdkColor GdkColor;

struct __GdkColor
{
  gulong  pixel;
  gushort red;
  gushort green;
  gushort blue;
};
```

`red`, `green` et `blue` sont respectivement les quantités de rouge, de vert et de bleu que contient notre couleur. Ces quantités peuvent varier de 0 à 65.535. Ainsi une couleur blanche s'obtient en mettant la valeur 65.535 dans chacun de ces champs. Le champ `pixel` représente l'index de la couleur dans la colormap. Ce champ ne peut pas être rempli à la légère. Au contraire, ce champ ne peut être rempli que par le GDK lui-même en appelant la fonction :

```
gboolean gdk_color_alloc(GdkColormap *colormap,
                         GdkColor *couleur);
```

où `colormap` est la colormap dans laquelle on souhaite ajouter une couleur et `couleur` est un pointeur sur une structure `GtkColor` dans laquelle on a rempli les champs `red`, `green` et `blue`. Cette fonction réalise en fait trois actions distinctes :

- elle demande à X-Window de réserver une case dans la colormap pour la couleur passée en paramètre, si cela est possible ;
- elle remplit le champ `pixel` de la structure `GdkColor` avec la valeur indiquée par X Window ;
- elle renvoie `TRUE` si la couleur a pu être enregistrée dans la colormap, et `FALSE` sinon.

Par la suite, on se référera a une couleur surtout grâce au champ `pixel` de la structure GdkColor. C'est en effet le seul paramètre qui sera passé au serveur X pour affecter une couleur à un autre élément.

Cependant, cette méthode de réservation de couleur n'est pas sans risque. En effet, une colormap peut n'avoir qu'un nombre de couleur très restreint, surtout si la carte vidéo est 8-bits. Heureusement, il existe une méthode d'allocation des couleurs qui elle fonctionne pratiquement toujours. Elle consiste à appeler la fonction :

```
gboolean gdk_colormap_alloc_color(GdkColormap *colormap,
                                  GdkColor *couleur,
                                  gboolean changeante,
                                  gboolean compromis);
```

qui peut paraître un peu plus complexe que la précédante, mais est beaucoup plus puissante. Les deux premiers paramètres et la valeur renvoyée ont exactement le même sens que pour `gdk_color_alloc()` mais les deux paramètres supplémentaires permettent une souplesse beaucoup plus grande. En effet, si le paramètre `compromis` à la valeur `TRUE`, alors aucune nouvelle couleur ne sera allouée, mais GDK cherchera la couleur la plus proche dans la colormap et adaptera les champs de `couleur` pour qu'il reflète la couleur choisie. Avec cette méthode la couleur est toujours trouvée car la palette du système contient toujours au moins deux couleurs, et l'une d'elles sera donc forcément le meilleur compromis.

Le paramètre `changeante` est totalement différent et incompatible avec le paramètre `compromis`. Si `changeante` est à `TRUE`, alors la couleur est réservé comme une couleur qui pourra changer par la suite. L'idée est de pouvoir dessiner dans une couleur, puis de changer cette couleur par la suite sans avoir à tout redessiner. Cette fonctionalité n'est pas supportée par tous les visuals et n'est pas recommendée puisqu'elle va à l'encontre de l'idée de compromis.

Il peut arriver que l'on ait besoin d'une couleur qui ressemble à une autre. On veut alors récupérer ses quantités de rouge, de vert et de bleu. Pour cela il existe une fonction qui permet justement d'obtenir une copie d'une couleur.

```
GdkColor *gdk_color_copy(GdkColor *couleur);
```

Attention, cette fonction crée une structure et réserve de la mémoire pour la couleur copiée. Cette partie de la mémoire devra être libérée par la suite avec :

```
void gdk_color_free(GdkColor *couleur);
```

Comme je l'ai signalé un peu plus tôt, une colormap créée par le système contient toujours au moins deux couleurs, ce sont le noir et le blanc. Ces couleurs peuvent être obtenues en appelant respectivement les fonctions suivantes :

```
gboolean gdk_color_black(GdkColormap *colormap,
                         GdkColor *couleur);
gboolean gdk_color_white(GdkColormap *colormap,
                         GdkColor *couleur);
```

Contrairement à la fonction `gdk_color_alloc()`, la structure `GdkColor` n'a pas besoin d'être remplie avant l'appel, puisque ces fonctions remplissent elle-même les quatre champs.

Peut être n'êtes vous pas un expert en colorimétrie, et il vous est pénible de trouver les triplets `(red, green, blue)` correspondant à la couleur que vous avez en tête. La fonction suivante :

```
gboolean gdk_color_parse(const gchar *nom,
                         GdkColor *couleur);
```

devrait faire votre bonheur. Elle prend en premier paramètre le *nom* d'une couleur, et remplie la structure `couleur` pour vous et renvoie la valeur `TRUE` si la traduction a réussi. Rassurez-vous, il n'y a rien de magique là-dessous. Cette fonction utilise simplement le fichier */usr/lib/X11/rgb.txt* pour trouver la correspondance. Consultez ce fichier pour voir quelles sont les couleurs disponibles.

5. Les polices de caractères

Une autre caractéristique des serveurs X est leur gestion des polices de caractères ou *fontes*.

Avant de pouvoir utiliser une police, par exemple pour dessiner un texte, il faut d'abord la charger avec :

```
GdkFont *gdk_font_load(const gchar *NomPolice);
```

où `NomPolice` est un nom de police X Window respectant le standard XLFD où *X Logical Font Description*. Ce qui est à la fois une bonne et une mauvaise nouvelle. Une bonne car cela permet de décrire très précisément la police que l'on souhaite. Une mauvaise car le format de ce nom est un ensemble de 14 champs séparés par des '-'. Ansi, un nom de police ressemble à :

```
-freefont-agate-normal-r-normal--15-140-75-75-p-65-iso8859-1
```

ou

```
-adobe-courier-demibold-o-normal--15-140-75-75-p-82-iso8859-1
```

Ces quatorze champs représente dans l'ordre :

- Le fondeur : l'auteur de la police. Le nom *fondeur* vient du temps reculé où les polices de caractères était faites en plomb pour des imprimeries à la Gutenberg. Ce nom peut être `adobe` pour les polices créées par l'éditeur de logiciel du même nom, `freefont` pour les polices gratuites, etc.
- La famille : la forme de la police, comme `courier`, `agate`, `times`, etc.
- La graisse : la police est elle en caractères normaux, en caractères gras, ou autre, les valeurs classiques sont : `bold`, `demibold`, `normal`, etc.
- L'inclinaison : romane, italique ou oblique, les valeurs autorisées sont `r`, `i` ou `o`.
- La largeur proportionnelle : spécifie si la fonte est condensée, élargie, ou normale, les valeurs classiques dont `normal`, `condensed`, `narrow`, etc.

- Le style : permet de spécifier un style additionnel. Ce champ est très peu utilisé et presque toujours vide.
- La taille en pixels de la police : combien de pixels occupe chaque caractère de la police.
- La taille en dixième de point de la police : les deux exemples si dessus ont une taille de 14 points. Il y a 72 points dans un pouce. Cela donne donc la taille physique de la police.
- La résolution horizontale : en nombre de points par pouce.
- La résolution verticale : en nombre de points par pouce.
- L'espacement : indique si la police est à espacement régulier comme la police `courier` par exemple — on a alors un `m` — ou si l'espacement dépand de la lettre — on a alors un `p`.
- La largeur moyenne : la largeur moyenne de tous les caractères de la police, exprimée en dixième de points.
- Le standard du jeu de caractères : le nom de l'organisation qui a crée ce standard. Il s'agit souvent de `iso8859` pour les jeux de caractères européens.
- Le code du jeu de caractères : il s'agit d'un nombre qu désigne un jeu particulier parmis ceux définis par le standard du champ précédent. Par exemple, une police dont le nom fini par `iso8859-1` est une police qui contient les principaux accents européens.

À première vue, cela semble être une tâche insurmontable de choisir une police sur un système X-Window. Rassurez-vous, aucun de ces champs n'est obligatoire. Si vous désirez omettre un champ, il suffit de le remplacer par une `*`. Ansi, si vous voulez simplement une police courier en 14 points, il suffit d'appeler la fonction `gdk_font_load()` de cette manière avec :

```
"-*-courier-*-*-*-*-*-140-*-*-*-*-*-*"
```

comme paramètre. Notez que les paramètres omis doivent tout de même être présent sous forme d'astérisque. Dans ce cas, le serveur X choisira la première police correspondant à ce format. Si aucune police adaptée n'est trouvée, la fonction `gdk_font_load()` renvoie la valeur `NULL`. Il faut donc toujours tester cette valeur afin de vérifier l'existence d'une police.

Vous pouvez connaître l'ensemble des polices installées sur votre système et faire votre choix grâce à l'utilitaire `xfontsel`.

Les principales opérations que l'on fait avec une police de caractères (mis à part l'affichage de texte que l'on abordera dans le paragraphe **FIXME PLEASE**) est la recherche de la taille en pixels utilisée par telle ou telle chaîne de caractères lorsqu'elle est affichée dans une police donnée.

Ainsi, il existe trois fonctions permettant de connaître une largeur :

```
gint gdk_string_width(GdkFont *police,
                      const gchar *chaine);
gint gdk_text_width(GdkFont *police,
                    const gchar *texte,
                    gint longueur);
```

```
gint gdk_char_width(GdkFont *police,
                    gchar caractere);
```

Dans ces trois fonctions, le paramètre `police` est une police de caractères telle que la fonction `gdk_font_load()` renvoie. La première, `gdk_string_width()`, renvoie la largeur en pixels de la chaîne passée en second paramètre. La deuxième réalise la même chose mais ne prend en compte que le nombre de caractères passé dans le paramètre `longueur`. La troisième calcule la largeur occupée par un seul caractère de la police.

Il existe bien évidemment des fonctions équivalentes permettant de connaître la hauteur d'une chaîne :

```
gint gdk_string_height(GdkFont *police,
                       const gchar *chaine);
gint gdk_text_height(GdkFont *police,
                     const gchar *texte,
                     gint longueur);
gint gdk_char_height(GdkFont *police,
                     gchar caractere);
```

Cependant ces six dernières fonctions ne donnent pas toujours toutes les informations souhaitées. Aussi existe-t-il deux fonctions renvoyant un maximum d'indications sur les tailles d'une chaîne de caractère passées en paramètre :

```
void gdk_text_extents(GdkFont *police,
                      const gchar *texte,
                      gint longueur,
                      gint *bordgauche,
                      gint *borddroit,
                      gint *largeur,
                      gint *elevation,
                      gint *profondeur);
void gdk_string_extents(GdkFont *police,
                        const gchar *chaine,
                        gint *bordgauche,
                        gint *borddroit,
                        gint *largeur,
                        gint *elevation,
                        gint *profondeur);
```

Vous l'aurez compris `gdk_string_extents()` concerne les chaînes de caractères complètes alors que `gdk_text_extents` est utilisé pour une partie d'une chaîne dont on passe la longueur comme troisième argument. Les cinq derniers arguments de ces deux fonctions ont le même rôle dans les deux cas. Ils représentent les tailles que l'on recherche. À l'issu de l'appel à l'une de ces fonctions, `largeur` contiendra la largeur naturelle de la chaîne en pixels, `elevation` le nombre de pixels utilisés au dessus de la ligne de base (celle sur laquelle sont posés les caractères sans

empattement) et `profondeur` le nombre de pixels utilisés par les différents empattement des lettres comme `p` ou `g` éventuellement présentes dans la chaîne. La hauteur de la chaîne est donc la somme de `elevation` et de `profondeur`.

Lorsqu'une police italique est utilisée, les caractères sont penchés et peuvent donc dépasser à droite ou à gauche. Ainsi le paramètre `borddroit` contiendra le nombre de pixels dont la dernière lettre dépasse vers la droite, et `bordgauche` le nombre de pixels débordant de la gauche de la première lettre.

6. Les contextes graphiques

Les contextes graphiques sont des ensembles de paramètres qui sont utilisés lorsque l'on veut dessiner quelque chose dans une fenêtre ou un pixmap. Ils regroupent des paramètres comme la couleur de dessin, la police de caractère à utiliser, le motif de remplissage, etc.

Un contexte graphique du point de vu de GDK est appelé `GdkGC`, et est accédé comme pour la plupart des structure de GDK via un pointeur. La façon la plus simple de créer un contexte graphique est de récupéré celui attaché par défaut à la fenêtre dans laquelle on désire dessiner quelque chose à l'aide de la fonction :

```
GdkGC *gdk_gc_new(GdkWindow *fenetre);
```

Les contextes graphiques doivent être détruits lorsque l'on n'en a plus besoin avec un appel à :

```
void gdk_gc_destroy(GdkGC *gc);
```

Le paramètre d'un contexte graphique le plus utilisé est très certainement la couleur de dessin, dite couleur d'avant-plan. Elle peut être fixée à l'aide de la fonction suivante :

```
void gdk_gc_set_foreground(GdkGC *gc,
                           GdkColor *couleur);
```

où `couleur` est une couleur qui doit avoir été allouée dans la colormap de la fenêtre auparavant.

De la même façon, la couleur d'arrière plan sera indiquée grâce à :

```
void gdk_gc_set_background(GdkGC  *gc,
                           GdkColor *couleur);
```

Si l'on désire dessiner un texte dans une fenêtre ou un pixmap avec une police de caractères particulière, celle-ci doit tout d'abord être chargée avec `gtk_font_load()` puis on la fixe dans le contexte graphique avec la fonction :

```
void gdk_gc_set_font(GdkGC *gc,
                     GdkFont *police);
```

Par défaut, lorsque l'on affiche un dessin dans une fenêtre, celui-ci se superpose à ce qui est déjà affiché dans la fenêtre. C'est ce que l'on appelle le mode `GDK_COPY`. C'est-à-dire que le contenu du dessin (que celui-ci soit une ligne, une ellipse ou autre chose) est copié, pixel par pixel par dessus l'ancien contenu de la fenêtre. Cependant si

ce comportement est souvent souhaité, ce n'est pas le seul possible. En effet, lors d'une opération de dessin dans une fenêtre, le serveur X calcule une fonction logique entre le dessin et le contenu antérieur de la fenêtre. Le résultat de cette fonction devient le nouveau contenu de la fenêtre. Cette fonction logique peut être changée grâce à :

```
void gdk_gc_set_function(GdkGC *gc,
                         GdkFunction fonction);
```

où fonction est l'une des valeur suivantes :

- `GDK_COPY` le dessin est simplement copié dans la fenêtre,
- `GDK_INVERT` chaque pixel du dessin est inversé avant d'être copié dans la fenêtre.
- `GDK_XOR` les pixels de la fenêtre et du dessin sont combinés avec la fonction *ou exclusif* qui a l'énorme avantage d'être réversible. C'est-à-dire que si l'on dessine deux fois le même dessin avec cette fonction, la fenêtre se retrouve exactement dans son état d'origine.
- `GDK_CLEAR` Les pixels de la fenêtre correspondant au dessin sont effacés quelques soient les couleurs du dessin.
- `GDK_SET` Les pixels de la fenêtre correspondant au dessin sont mis en blancs, quelques soient les couleurs du dessin.

Pour les valeurs `GDK_AND`, `GDK_AND_REVERSE`, `GDK_AND_INVERT` `GDK_NOOP`, `GDK_OR`, `GDK_EQUIV`, `GDK_OR_REVERSE`, `GDK_COPY_INVERT`, `GDK_OR_INVERT`, et `GDK_NAND` les pixels du dessin et de la fenêtre sont combinés suvant la fonction logique correspondante.

Dans les faits seuls `GDK_COPY` et `GDK_XOR` sont réellement utilisés les autres ne sont là que pour être exhaustifs vis-à-vis de la Xlib.

Beaucoup des dessins que propose la Xlib (et donc le GDK) sont linéaires, comme les lignes, les rectangles, les arcs d'ellipses. La fonction suivante permet de positionner les principaux aspects de ce genre de dessins dans le contexte graphique :

```
void gdk_gc_set_line_attributes(GdkGC *gc,
                                gint largeur_ligne,
                                GdkLineStyle style_ligne,
                                GdkCapStyle style_bout,
                                GdkJoinStyle style_jointure);
```

où `largeur_ligne` est la largeur en nombre de pixels du trait. Notez que, curieusement, la valeur par défaut est 0. En fait une ligne de largeur 0 est simplement dessinée plus rapidement qu'une ligne de largeur 1, car le serveur utilise un algorithme légèrement différent lorsqu'il rencontre une telle largeur. `style_ligne` peut prendre l'une des trois valeurs :

- `GDK_LINE_SOLID` qui est la valeur par défaut, la ligne est alors une ligne continue, normale ;
- `GDK_LINE_ON_OFF_DASH` la ligne est alors une ligne pointillée, et seulement les traits du pointillé sont tracés. Les interstices entre les traits du pointillés sont laissés transparent.

– `GDK_LINE_DOUBLE_DASH` la ligne est là aussi pointillée, mais les interstices entre les traits du pointillés sont dessinés avec la couleur de fond du contexte graphique.

`style_bout` définit comment les extrémités des lignes doivent être dessinées. Ce paramètre peut prendre les valeurs suivantes :

– `GDK_CAP_BUTT` la ligne est un rectangle qui s'arrête exactement au points de l'extrémité de la ligne ;
– `GDK_CAP_NOT_LAST`, identique au précédent, mais la dernière rangée de pixels n'est pas dessinée ;
– `GDK_CAP_ROUND`, chaque extrémité de la ligne est terminée par un demi cercle dont le centre est l'extrémité indiquée de la ligne et le rayon la moitié de la largeur de la ligne ;
– `GDK_CAP_PROJECTING`, le bord de la ligne est bien rectagulaire, mais la ligne dépasse de son extrémité de la moitié de sa largeur.

`style_jointure` définit comment l'intersection de deux lignes succéssives doit être dessinée. Ce paramètre peut prendre l'une des trois valeurs suivantes :

– `GDK_JOIN_MITER`, les bords extérieurs des lignes sont prolongés et forment un triangle ;
– `GDK_JOIN_ROUND`, les bords extérieurs sont réunis par un cercle dont le diamètre est égale à la largeur des lignes ;
– `GDK_JOIN_BEVEL`, les bors extérieurs des lignes sont simplement joints en ligne droite et l'interstice entre les bords et la ligne est paint avec le même style que les lignes. (voir figure 3)

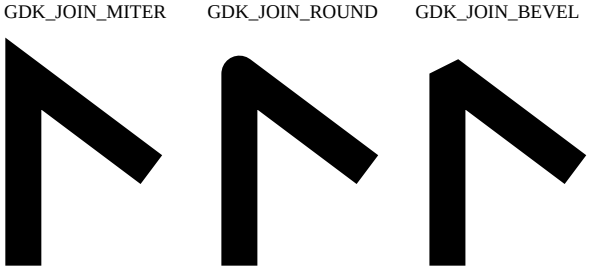


fig. 3

Comme vous venez de le voir, le GDK permet de dessiner des lignes en pointillée. Le motif utilisé pour dessiné ces pointillés peut être changé en utilisant la fonction suivante :

```
void gdk_gc_set_dashes(GdkGC *gc,
                       gint decalage,
                       gchar list_longueur[],
                       gint n);
```

où `list_longueur` est un tableau contenant les différentes longueurs du motif. Ainsi, si ce tableau contient les valeurs 2, 3, 4 et 3, les lignes seront dessinées pendant

2 pixels, puis laissées en blanc pendant 3 pixels, puis dessinées pendant 4 pixels, puis laissées en blanc pendant 3 pixels, et ainsi de suite. `n` indique simplement la longueur du tableau, et `decalage` indique le décalage en pixels de la ligne par rapport au début du tableau.

Certaines primitives de GDK permettent de dessiner des formes pleines. Elles sont généralement remplies d'une couleur unie. Mais ce n'est pas toujours le cas. En effet, il est possible de définir un motif de remplissage. Ce motif est appelé *stipple* lorsqu'il est monochrome et *tile* s'il est en couleur. Les tiles et les stipples d'un contexte graphique peuvent être définis avec les deux fonctions suivantes :

```
void gdk_gc_set_tile(GdkGC *gc,
                     GdkPixmap *tile);
void gdk_gc_set_stipple(GdkGC *gc,
                        GdkPixmap *stipple);
```

Lorsqu'un tile ou un stipple sont utilisés comme motif de remplissage, ils sont copiés autant de fois que nécessaire pour couvrir la surface à remplir. Le point `(0, 0)` du motif coïncide normalement avec celui de la fenêtre ou du pixmap où le dessin est tracé. Ceci peut être changé en appliquant un offset au tile ou au stipple à l'aide de la fonction :

```
void gdk_gc_set_ts_origin(GdkGC *gc,
                          gint x, gint y);
```

Les tiles et les stipples semblent complètement interchangeables mais il sont effectivement deux choses très différentes du point de vue de X Window. Aussi un contexte graphique peut tout-à-fait comporter un tile *et* un stipple. On peut donc choisir si le remplissage d'une forme doit être effectué avec une couleur unie, un tile ou un stipple avec :

```
void gdk_gc_set_fill(GdkGC *gc,
                     GdkFill style);
```

où le paramètre `style` définit la façon dont les prochains remplissages seront effectués. Il peut prendre les valeurs suivantes :

- `GDK_SOLID`, dans ce cas les remplissages se feront de manière unie, avec la couleur d'avant plan.
- `GDK_TILED` indique que les remplissages utiliseront le *tile* comme motif.
- `GDK_STIPPLED`, alors les remplissages se feront en utilisant le stipple comme un masque. Les pixels positionnés dans le dessin et dans le stipple seront dessinés avec la couleur d'avant plan, les autres resteront inchangés.
- `GDK_OPAQUE_STIPPLED` indique que les remplissages doivent se faire avec *stipple* comme motif. Les pixels mis à 1 dans le stipple seront dessinés avec la couleur d'avant plan, les autres seront dessinés avec la couleur d'arrière plan.

En général, les dessins peuvent être dessinés sur toute la suface de la fenêtre ou du pixmap. Cependant, une méthode, nommée *coupure* ou *clipping* permet de limiter l'effet d'un dessin à une zone précise. Tous les pixels situés en dehors de cette zone ne seront pas modifiés par une opération de dessin dans la fenêtre ou le pixmap.

Par exemple, on peut limiter toute tentative de dessin à un rectangle en utilisant la fonction :

```
void gdk_gc_set_clip_rectangle(GdkGC *gc,
                               GdkRectangle *rectangle);
```

où `rectangle` est un pointeur sur une structure `GdkRectangle` définie ainsi :

```
typedef struct _GdkRectangle  GdkRectangle;

struct _GdkRectangle
{
  gint16 x;
  gint16 y;
  guint16 width;
  guint16 height;
};
```

Ainsi, si l'on veut limiter les prochains dessins au rectangle dont les coins opposés sont `(50, 50)` et `(100, 150)`, on pourra utiliser le bout de code suivant :

```
GdkRectangle rect;

rect.x = 50;
rect.y = 50;
rect.width = 50;
rect.height = 100;
gdk_gc_set_clip_rectangle(gc, &rect);
```

Si l'on veut une forme plus complexe qu'un rectangle comme zone de clipping, il faut tout d'abord créer un `GdkBitmap`, c'est-à-dire un pixmap de profondeur égale à 1, qui aura exactement la forme voulue, et qui servira de masque de clipping. Puis on affectera ce masque au contexte graphique grâce à la fonction suivante :

```
void gdk_gc_set_clip_mask(GdkGC *gc,
                          GdkBitmap *masque);
```

Par défaut, l'origine du masque de clipping coïncide exactement avec l'origine de la fenêtre ou du pixmap. Cela peut être pénible si la zone de clipping que l'on veut définir est en bas à droite, c'est pourquoi il est possible de définir quelle doit être la position de l'origine du masque de clipping dans la fenêtre avec la fonction suivante :

```
void gdk_gc_set_clip_origin(GdkGC *gc,
                            gint x, gint y);
```

Nous veno6ns de voir toutes[9] les caractéristiques des contextes graphiques. Il est possible de créer un contexte graphique possédant directement toutes les caractéristiques voulues. Pour cela, il faut dans un premier temps remplir les champs désirés dans une structure `GdkGCValues` définie comme suit :

[9] presque toutes pour être honnête, en fait toutes celles qui peuvent avoir une utilité dans un programme GTK+, les autres ne sont d'ailleurs que très peu utilisées

```
typedef struct _GdkGCValues  GdkGCValues;

struct _GdkGCValues
{
  GdkColor          foreground;
  GdkColor          background;
  GdkFont          *font;
  GdkFunction       function;
  GdkFill           fill;
  GdkPixmap        *tile;
  GdkPixmap        *stipple;
  GdkPixmap        *clip_mask;
  GdkSubwindowMode  subwindow_mode;
  gint              ts_x_origin;
  gint              ts_y_origin;
  gint              clip_x_origin;
  gint              clip_y_origin;
  gint              graphics_exposures;
  gint              line_width;
  GdkLineStyle      line_style;
  GdkCapStyle       cap_style;
  GdkJoinStyle      join_style;
};
```

La plupart des champs de cette structure doivent vous être familiers si vous avez lus les paragraphes précédents. Bien entendu, tous les champs ne doivent pas forcément être renseignés. Une fois les champs que l'on souhaite fixer remplis, il faut appeler la fonction suivante :

```
GdkGC *gdk_gc_new_with_values(GdkWindow *fenetre,
                              GdkGCValues    *carac,
                              GdkGCValuesMask masque_carac);
```

où `fenetre` est la fenêtre de rattachement du contexte graphique, `carac` est un pointeur sur la structure `GdkGCValues` que l'on vient de remplir, et `masque_carac` est un champ de bits indiquant quelles sont les caractéristiques que l'on a définies dans la structure pointée par `carac`. Ces bits sont nommés suivant la caractéristique qu'ils définissent comme indiqué dans le tableau suivant :

nom du bit	caractéristique
GDK_GC_FOREGROUND	foreground
GDK_GC_BACKGROUND	background
GDK_GC_FONT	font
GDK_GC_FUNCTION	function
GDK_GC_FILL	fill
GDK_GC_TILE	tile
GDK_GC_STIPPLE	stipple
GDK_GC_CLIP_MASK	clip_mask
GDK_GC_SUBWINDOW	subwindow_mode
GDK_GC_TS_X_ORIGIN	ts_x_origin
GDK_GC_TS_Y_ORIGIN	ts_y_origin
GDK_GC_CLIP_X_ORIGIN	clip_x_origin
GDK_GC_CLIP_Y_ORIGIN	clip_y_origin
GDK_GC_EXPOSURES	graphics_exposures
GDK_GC_LINE_WIDTH	line_width
GDK_GC_LINE_STYLE	line_style
GDK_GC_CAP_STYLE	cap_style
GDK_GC_JOIN_STYLE	join_style

Ainsi, si l'on positionne uniquement les couleurs d'avant et d'arrière plan, et la police de caractères dans la structure pointée par `carac`, alors `masque_carac` aura la valeur `GDK_GC_FOREGROUND | GDK_GC_BACKGROUND | GDK_GC_FONT`.

De manière réciproque, on peut connaître l'ensemble des caractéristiques d'un contexte graphique à l'aide de la fonction suivante :

```
void gdk_gc_get_values(GdkGC *gc,
                       GdkGCValues *values);
```

Cela peut être utile si l'on veut créer un contexte graphique ressemblant beaucoup à un autre. En effet il suffit d'appeler cette fonction sur le contexte d'origine, puis de changer quelques champs et d'appeler `gdk_gc_new_with_values()` pour créer le nouveau contexte graphique.

Il existe cependant un autre moyen de réaliser cela, en effet la fonction suivante :

```
void gdk_gc_copy(GdkGC *gc_dest,
                 GdkGC *gc_source);
```

permet de réaliser une copie du contexte `gc_source`, et de la stoquer dans `gc_dest`. Il est ensuite facile de modifier telle ou telle caractéristique à l'aide des fonctions présentées précédemment.

7. Les primitives de dessin

Nous en savons maintenant assez pour commencer à réellement dessiner des figures dans nos fenêtres. En effet, avant de pouvoir faire cela, nous devions d'abord

savoir ouvrir une fenêtre, choisir un visual, une colormap, une police de caractères, régler les différents paramètres des contextes graphiques liés à notre fenêtre, etc. Nous allons pouvoir utiliser nos acquis afin de faire de jolis dessins à l'écran. Ces dessins se font à l'aide de primitives simples, comme des points, des rectangles, des ellipses, qui peuvent être combinés.

La primitive la plus simple consiste à dessiner un simple point à l'écran. Pour cela, on appelle la fonction suivante :

```
void gdk_draw_point(GdkDrawable *drawable,
                    GdkGC *gc,
                    gint x, gint y);
```

où `x` et `y` sont les coordonnées de notre point dans la fenêtre. Le point sera tracé avec la couleur d'avant plan définie dans le contexte graphique. Bien entendu, les caractérisques de ce contexte influencent beaucoup sur l'aspect de ce point, ainsi il peut avoir une certaine grosseur, être clippé, etc. Vous remarquerez que toutes les fonctions de dessins de primitives utilisent comme premier paramètre un `GdkDrawable`, il s'agit en fait d'un type générique regroupant aussi bien les fenêtres (`GdkWindow`) que les pixmaps (`GdkPixmap` et `GdkBitmap`). En effet, les primitives peuvent, en général, se dessiner aussi bien dans des fenêtres que dans des pixmaps.

On utilise rarement un point isolé. Il est en effet nettement plus fréquents de dessiner un nuage de points, ou les points représentant une fonction mathématique. Dans ce cas, le plus simple est de placer les coordonnées des points dans un tableau de type `GdkPoint`, puis d'appeler la fonction suivante :

```
void gdk_draw_points(GdkDrawable *drawable,
                     GdkGC *gc,
                     GdkPoint *points,
                     gint npoints);
```

où `points` est le tableau de points et `npoints` le nombre de points dans le tableau. La structure `GdkPoint` est définie ainsi :

```
typedef struct _GdkPoint  GdkPoint;

struct _GdkPoint
{
  gint16 x;
  gint16 y;
};
```

Pour éclaircir l'utilisation de cette fonction, voici un bout de code qui trace cinq points disposés en croix :

```
GdkPoint points[5];

points[0].x =  50; points[0].y =  50;
points[1].x = 100; points[1].y =  50;
```

```
points[2].x = 100; points[2].y = 100;
points[3].x =  50; points[3].y = 100;
points[4].x =  75; points[4].y =  75;

gdk_draw_points(fenetre, gc, points, 5);
```

La primitive suivante, qui est aussi l'une des plus employées est la ligne. Pour tracer une ligne du point de coordonnées `(x1, y1)` au point `(x2, y2)`, il suffit d'appeler la fonction suivante :

```
void gdk_draw_line(GdkDrawable *drawable,
                   GdkGC *gc,
                   gint x1, gint y1,
                   gint x2, gint y2);
```

Comme pour les points, il est possible de dessiner d'un seul coup plusieurs lignes. Pour cela, il faut remplir un tableau de type `GdkSegment` et appeler la fonction suivante :

```
void gdk_draw_segments(GdkDrawable *drawable,
                       GdkGC *gc,
                       GdkSegment *segs,
                       gint nsegs);
```

où `segs` est bien évidemment le tableau de lignes et `nsegs` est le nombre de lignes contenues dans le tableau `segs`. La structure GdkSegment est définie comme suit :

```
typedef struct _GdkSegment  GdkSegment;

struct _GdkSegment
{
  gint16 x1;
  gint16 y1;
  gint16 x2;
  gint16 y2;
};
```

et elle définit la ligne allant du point `(x1, y1)` au point `(x2, y2)`. Cependant cette fonction ne permet de tracer que des lignes indépendantes. Ce n'est pas toujours ce que l'on souhaite. On désire en effet souvent simplement joindre des points par une ligne brisée. Pour cela, on remplit simplement un tableau de points comme pour la fonction `gdk_draw_points()` et l'on appelle la fonction :

```
void gdk_draw_lines(GdkDrawable *drawable,
                    GdkGC *gc,
                    GdkPoint *points,
                    gint npoints);
```

Notez bien que `npoints` représente le nombre de points du tableau, le nombre de lignes tracées est donc `npoints`−1. La forme de la jonction entre deux lignes consécutives dépend de la caractéristique `cap_style` du contexte graphique utilisé.

Lorsque l'on dessine une ligne brisée, cela peut être pour dessiner un polygône. Il suffit alors que le dernier point du tableau ait les mêmes coordonnées que le premier point. Cependant, si l'on agit de la sorte, le point de raccord peut être mal dessiné par rapport aux autres, notamment si l'épaisseur de la ligne est importante. C'est pourquoi il vaut mieux utiliser la fonction :

```
void gdk_draw_polygon(GdkDrawable *drawable,
                      GdkGC *gc,
                      gint rempli,
                      GdkPoint *points,
                      gint npoints);
```

pour dessiner des polygones. Ici, `points` est un tableau contenant les coordonnées des points du polygônes, `npoints` est le nombre de points du tableau ou, ce qui revient au même, le nombre de lignes que comporte le polygône. Le paramètre `rempli` est un indicateur qui spécifie si le polygône doit être rempli (`TRUE`) ou s'il doit rester vide (`FALSE`), c'est-à-dire transparent.

Une autre forme couramment utilisée est le rectangle, vide ou plein qui est tracé avec :

```
void gdk_draw_rectangle(GdkDrawable *drawable,
                        GdkGC *gc,
                        gint rempli,
                        gint x, gint y,
                        gint largeur, gint hauteur);
```

dont les arguments semblent assez parlants. Faites seulement attention à ce que `largeur` et `hauteur` soient positifs.

Le GDK ne possède pas de primitives permettant de tracé des cercles, des disques ou des ellipses[10] mais il propose en revanche une fonction bien plus puissante, qui permet dans le cas le plus général de tracer un arc d'ellipse, rempli ou non :

```
void gdk_draw_arc(GdkDrawable *drawable,
                  GdkGC *gc,
                  gint rempli,
                  gint x, gint y,
                  gint largeur, gint hauteur,
                  gint angle1, gint angle2);
```

L'ellipse servant de support à l'arc tracé est dans le rectangle dont le coin supérieur gauche est aux coordonnées `(x, y)`, de largeur `largeur` et de hauteur `hauteur`. Le centre de l'ellipse est donc aux coordonnées `(x+largeur/2, y+hauteur/2)`. Les paramètres `angle1` et `angle2` indiquent les angles de début

[10] Xlib non plus d'ailleurs...

et de fin de l'arc en 64$^{\text{ième}}$ de degré. Par exemple, pour dessiner un quart de cercle centré en `(50, 50)` de rayon 30 et vide, on utilisera :

```
gdk_draw_arc(fenetre, gc,
             FALSE,
             20, 20,
             30, 30,
             0, 90*64);
```

Le GDK propose évidemment des fonctions permettant de dessiner des textes. Comme habituellement, celles-ci ne diffèrent que par le fait que l'une traite des chaînes de caractères entières et l'autre un bout de chaîne dont on indique la longueur :

```
void gdk_draw_string(GdkDrawable *drawable,
                     GdkFont *police,
                     GdkGC *gc,
                     gint x, gint y,
                     const gchar *chaine);
void gdk_draw_text(GdkDrawable *drawable,
                   GdkFont *police,
                   GdkGC *gc,
                   gint x, gint y,
                   const gchar *texte,
                   gint longueur_texte);
```

Notez que `(x, y)` sont les coordonnées de la gauche de la ligne de base du texte.

Enfin, il est possible de copier un pixmap ou une partie d'un pixmap dans un autre pixmap ou dans un fenêtre grâce à la fonction suivante :

```
void gdk_draw_pixmap(GdkDrawable *drawable,
                     GdkGC *gc,
                     GdkPixmap *src,
                     gint xsrc, gint ysrc,
                     gint xdest, gint ydest,
                     gint largeur, gint hauteur);
```

où `drawable` est le pixmap ou la fenêtre de destination, `src` est le pixmap[11] source — celui à partir duquel on effectue la copie — `(xsrc, ysrc)` sont les coordonnées du coin supérieur gauche de la zone à copier dans le pixmap source et `(xdest, ydest)` sont les coordonnées où la partie de pixmap doit être copiée dans le pixmap ou la fenêtre de destination. Enfin, `largeur` et `hauteur` indiquent la taille de la zone à copier.

---

[11] En fait la fonction `gdk_draw_pixmap()` est tout à fait capable de copier à partir d'une fenêtre, mais pour des raisons de lisibilité du code, on utilisera plutôt `gdk_window_copy_area()` dans ce cas.

Cela permet, entre autres, de préparer un dessin complet en dehors de l'écran, dans un pixmap, et de le copier une fois fini, dans la fenêtre affichée. Ainsi on ne voit pas le dessin en train de se dessiner.

Pour copier une zone rectangulaire d'une fenêtre à une autre, on utilisera plutôt la fonction suivante :

```
void gdk_window_copy_area(GdkWindow *fenetre,
                          GdkGC *gc,
                          gint xdest, gint ydest,
                          GdkWindow *fenetre_source,
                          gint xsrc, gint xsrc,
                          gint largeur, gint hauteur);
```

qui fonctionne exactement comme `gdk_draw_pixmap()`, mais est prévue pour copier de fenêtre à fenêtre. Faites cependant attention, car les paramètres de ces deux fonctions n'apparaissent pas dans le même ordre.

Nous arrivons au terme de l'énumération des primitives de dessin du GDK. Il nous reste maintenant à savoir comment effacer un dessin, ou une partie de celui-ci.

Il est facile d'effacer complètement le contenu d'une fenêtre à l'aide de :

```
void gdk_window_clear(GdkWindow *fenetre);
```

La fenêtre est alors remplie avec sa couleur d'arrière plan, ou son motif d'arrière plan si elle en possède un.

Si on préfère n'effacer qu'une zone rectangulaire de la fenêtre, la fonction

```
void gdk_window_clear_area(GdkWindow *fenetre,
                           gint x, gint y,
                           gint largeur, gint hauteur);
```

est faite pour cela. Il suffit d'indiquer les coordonnées du point supérieur gauche de la zone dans `(x, y)` et la taille de la zone dans `largeur` et `hauteur`.

8. Les images RVB

Comme vous le savez, le GDK a été spécialement développé pour Gimp qui est un logiciel de retouche d'images. Gimp passe donc la majeure partie de son temps à dessiner des points individuellement. Une méthode possible pour réaliser cela serait de choisir une couleur, demander au serveur de l'allouer dans la colormap courante, puis de mettre cette couleur comme couleur d'avant plan dans le contexte graphique puis appeler la fonction `gdk_draw_point()`, et recommencer le processus pour chaque point. Cependant cette méthode est très mauvaise, en effet chaque appel à ces fonctions prend du temps, car à chaque fois l'information circule le long du réseau établi entre notre application et le serveur X. C'est pourquoi la méthode choisie par les concepteurs de Gimp (et de cette partie du GDK) consiste à faire le maximum d'opérations en local, à l'aide d'un simple tableau d'octets puis d'envoyer au serveur X uniquement l'image finie. Ces tableaux sont appelés des *images RVB*.

Nous pouvons tout à fait utiliser cette possibilité dans nos programmes GTK+. Pour cela il faut absolument inclure le fichier `gdk/gdkrgb.h` afin de définir les constantes et les prototypes des fonctions, et appeler la fonction

```
void gdk_rgb_init(void);
```

avant tout autre appel aux fonctions `gdk_rgb_*`.

Comme je l'ai indiqué, les images RVB sont de simples tableau unidimensionnels d'octets, non signés (des `guchar`). Ils représentent tout de même une image en deux dimensions et en couleurs. Le format de ce tableau dépend du type d'image que l'on veut générer. En effet, Gimp connait surtout 4 types d'images : les images en 24 bits (17 millions de couleurs), les images 32 bits (toujours 17 millions de couleurs car seul 24 bits sont utilisés mais les architectures modernes gèrent plus facilement des données de 32 bits que des données de 24 bits), les images en niveaux de gris (8 bits pour 256 niveaux de gris) et les images indexées (8 bits pour 256 couleurs choisies parmis une palette plus grande).

Nous allons donc étudier les images RVB suivant ces quatre types d'images.

8.1. Les images RVB 24 bits

Les images RVB 24 bits sont certainement les plus faciles à comprendre. Chaque pixel d'une telle image est codé sur 24 bits, c'est-à-dire 3 octets, un pour le rouge, un pour le vert et un pour le bleu. Une ligne d'une image RVB 24 bits est donc composée d'une suite de triplets d'octets, plus un certain nombre d'octets de remplissage qui serviront à faire de la taille de chaque ligne un multiple simple d'une puissance de 2. Les octets de remplissage ne sont pas du tout obligatoire et ne serve qu'à améliorer les performances du serveurs dans certains cas. Nous supposerons que nos images ne comporte pas d'octets de remplissage. La taille d'une ligne de n pixels est donc de $3 * n$ octets. Et une image de m lignes de n pixels est donc un tableau de $m * 3 * n$ octets non signés. On déclare alors une image RVB 24 bits de cette façon :

```
guchar image[3*LARGEUR*HAUTEUR];
```

Ensuite, on "dessine" dans ce tableau en donnant des valeurs directement aux quantités de rouge, de vert et de bleu de chaque pixels. Les valeurs de chaque composante peuvent aller de 0 à 255. Ainsi, si l'on veut que le premier pixel de l'image (en haut à gauche) soit blancs, les trois premiers octets du tableau seront mis à 255. Si on veut que le pixel aux coordonnées `(x, y)` soit en bleu cyan, on pourra utiliser un bout de code comme celui-ci :

```
/* Quantité de Rouge */
image[3*(LARGEUR*y+x)+0] =   0;
/* Quantité de Vert */
image[3*(LARGEUR*y+x)+1] = 255;
/* Quantité de Bleu */
image[3*(LARGEUR*y+x)+2] = 255;
```

Cela est très contraignant car il faut réinventer plein de choses car le GDK ne propose pas (pour l'instant) de fonctions de dessin de primitives dans de tels tableaux

et on ne peut évidemment pas utiliser les fonctions présentées dans le paragraphe précédent car elles ne s'appliquent qu'à des fenêtres ou à des pixmaps. Mais les images RVB sont plutôt destinées à faire du dessin point par point, cela n'est donc pas trop grave.

Une fois que l'on a rempli le tableau avec les différentes valeurs des composantes rouges, vertes et bleues de chaque pixel de l'image RVB, on peut la copier dans une fenêtre ou un pixmap à l'aide de la fonction suivante :

```
void gdk_draw_rgb_image(GdkDrawable *drawable,
                        GdkGC *gc,
                        gint x, gint y,
                        gint largeur, gint hauteur,
                        GdkRgbDither dith,
                        guchar *image,
                        gint longueur_ligne);
```

où `drawable` est la fenêtre ou le pixmap de destination, `gc` est un contexte graphique compatible avec la destination, `(x ,y)` sont les coordonnées du coin supérieur gauche de la zone de l'image RVB que l'on veut copier, `largeur` et `hauteur` indiquent la taille de cette zone, `image` est le tableau de `guchar` représentant l'image RVB et `longueur_ligne` est le nombre d'octets utilisés pour une ligne, c'est-à-dire trois fois la largeur de l'image plus les éventuels octets de remplissage.

Le paramètre `dith` mérite que l'on s'attarde un peu sur lui. Tout d'abord, il faut bien comprendre que les images RVB, même 24 bits, peuvent être utilisées sur tous les types de visuals et de colormaps. Et si la copie d'une image RVB 24 bits vers une colormap 24 bits ne pose pas de problème, le GDK doit effectuer quelques opérations pour arriver à copier une image RVB dans une fenêtre qui ne peut contenir que 256 couleurs ou moins. Le GDK résoud se problème en effectuant une opération de réduction de couleurs, c'est-à-dire, que pour chaque pixel de l'image source, il lui faut trouver la couleur convenant le mieux dans la colormap de la fenêtre de destination. Cependant, si l'on applique cet algorithme point par point, de vilaines bandes de couleurs (appelées bandes de Mach) apparaissent. Pour que ces bandes disparaisent, il faut que chaque pixel soit perçu dans son contexte.

Imaginons que l'image RVB créée comporte un dégradé vertical du rouge au blanc, et que la fenêtre de destination comporte bien le rouge et le blanc dans sa colormap mais aucune couleur intermédiaire. Si, pour chaque pixel de l'image RVB, on cherche la couleur la plus proche, on trouvera uniquement du rouge ou du blanc, et l'image résultante sera simplement formée d'une grosse bande rouge au dessus d'une grosse bande blanche. On est loin de notre dégradé du départ. En fait pour mieux simuler le dégradé, il faudrait graduellement introduire des points blanc dans la zone rouge et réciproquement. Plus exactement, si, sur une ligne la couleur est un rose comportant 40% de rouge et 60% de blanc, alors, sur cette ligne, il faut dessiner 40% des pixels en rouge et 60% en blanc. Si ces pixels sont suffisamment bien répartis, la ligne parait rose. Notez que cet exemple est simpliste car en général, les images comportent beaucoup de dégradés entremélés. Les algorithmes réalisant la réparition des couleurs et le calcul de la répartitions des pixels sont appelés *ditherings*.

Le paramètre `dith` indique dans quelles conditions le dithering doit être utilisé. Sa valeur peut être :

- `GDK_RGB_DITHER_NONE` : aucun dithering ne sera utilisé, chaque pixel prend simplement la couleur la plus proche présente dans la colormap. Ceci est parfaitement adapté si la colormap est 24 bits car c'est extrémement rapide mais produit des résultats catastrophiques si le nombre de couleur est limité.
- `GDK_RGB_DITHER_NORMAL` : le dithering ne sera utilisé que si la colormap possède 256 couleurs ou moins. Ceci représente en général le meilleur compromis puisque la nécessité du dithering diminue avec le nombre de couleurs.
- `GDK_RGB_DITHER_MAX` : le dithering est toujours utilisé. Il est inutile pour les colormaps de 24 bits. Cette option est la plus lente, mais aussi celle qui donne les meilleurs résultats graphiques.

Voici un exemple d'utilisation des images RVB 24 bits :

```
/* gdk rgb */
#include <gdk/gdk.h>
#include <gdk/gdkrgb.h>

#define LARGEUR (256)
#define HAUTEUR (256)

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkPixmap *Fond;
  GdkWindowAttr attr;
  guchar image[3*LARGEUR*HAUTEUR];
  int x,y;
  gint depth;

  attr.event_mask = 0;
  attr.width = LARGEUR;
  attr.height = HAUTEUR;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  gdk_init(&argc, &argv);
  gdk_rgb_init();
  /* Création de la fenêtre */
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Création du pixmap de fond de la fenêtre */
  depth = gdk_window_get_visual(Fenetre)->depth;
  Fond = gdk_pixmap_new(Fenetre,
                        LARGEUR, HAUTEUR,
                        depth);
  /* Création du dégradé du jaune au bleu */
```

```
  for (y=0 ; y<256 ; y++)
    for (x=0 ; x<256 ; x++)
    {
      /* Le rouge */
      image[3*(y*LARGEUR+x)+0] = 255-((x+y)%256);
      /* Le Vert */
      image[3*(y*LARGEUR+x)+1] = 255-((x+y)%256);
      /* Le bleu */
      image[3*(y*LARGEUR+x)+2] = (x+y)%256;
    }
  /* L'image RVB 24 bits est copiée dans le pixmap fond. */
  gdk_draw_rgb_image(Fond,
                     gdk_gc_new(Fenetre),
                     0,0, LARGEUR, HAUTEUR,
                     GDK_RGB_DITHER_NORMAL,
                     image,
                     3*LARGEUR);
  /* Le pixmap Fond devient le fond de la fenêtre */
  gdk_window_set_back_pixmap(Fenetre, Fond, FALSE);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
      gdk_event_get();
}
```

Vous remarquerez que, dans cet exemple, j'ai préféré copier l'image RVB dans un pixmap servant de fond à la fenêtre plutôt que directement dans la fenêtre. En effet, pour véritablement affiché quelque chose dans une fenêtre, il faut s'assurer que le serveur X ne va pas tenter de l'effacer. En dessinant directement dans le fond de la fenêtre, on est sûr que cela restera dans la fenêtre, même si celle-ci est effacée. En fait, le serveur X efface le contenu des fenêtres dès qu'il leur arrive la moindre chose, comme être afffichée par exemple. Nous verrons comment remédier à cela dans le paragraphe **FIXME PLEASE** .

8.2. Les images RVB 32 bits

Même si les images RVB 24 bits sont les plus souples et les plus courantes, ce ne sont pas les seules employées, ni les plus rapides à traiter. En effet, il est possible de stoquer les informations sur chaque pixel un peu différemment.

Les images RVB 32 bits permettent elles-aussi de manipuler des images ayant pratiquement jusqu'à 17 millions de couleurs. C'est-à-dire que l'information servant à décrire la couleur de chaque pixel n'utilise que 24 bits, les huit bits restant ne sont là que pour faire du remplissage. En effet le GDK traite beaucoup plus rapidement les données sur 32 bits que celle sur 24 bits. Les images RVB 32 bits se manipulent

donc exactement de la même façon que les images RVB 24 bits. Simplement les pixels occupents quatre octets au lieu de trois et seuls les trois premiers sont utilisés.

Les images RVB 32 bits sont copiées dans un pixmap ou une fenêtre en appelant la fonction suivante :

```
void gdk_draw_rgb_32_image(GdkDrawable *drawable,
                           GdkGC *gc,
                           gint x, gint y,
                           gint largeur, gint hauteur,
                           GdkRgbDither dith,
                           guchar *image,
                           gint longueur_ligne);
```

8.3. Les images RVB en niveau de gris

Les images en 17 millions de couleurs ne sont pas les seules que l'on puisse utilisées avec le module `gdkrvb` en effet, il peut aussi traiter des images en 256 niveaux de gris, même si dans ce cas l'appelation RVB[12] n'est pas forcément adaptée.

Une image RVB en niveau de gris est un tableau de `guchar` où chaque pixel occupe exactement un octet. La valeur de cet octet dans le tableau indique la luminosité du pixel. Ainsi, une valeur de 0 indique un pixel noir, une valeur de 255 un pixel blanc et une valeur de 127 un pixel gris moyen.

Le GDK est tout à fait capable d'utiliser un *dithering* même sur une image en niveau de gris, et donc l'appel de la fonction permettant de copier une image RGV niveau de gris dans un pixmal ou une fenêtre est semblable à celui utiliser pour les images RVB en couleur :

```
void gdk_draw_gray_image(GdkDrawable *drawable,
                         GdkGC *gc,
                         gint x, gint y,
                         gint largeur, gint hauteur,
                         GdkRgbDither dith,
                         guchar *image,
                         gint longueur_ligne);
```

8.4. Les images RVB indexées

Si vous utilisez the logiciel Gimp, vous devez savoir qu'il est capable de manipuler des images en 17 millions de couleurs comme les `.jpeg` ou les `.png`, des images en niveaux de gris, mais aussi des images en 256 couleurs parmis 17 millions, comme c'est le cas pour les `.gif`.

Ces images sont dites RVB indexées, car elles utilisent une palette de couleur indexée pour être représentée. Ces palettes ressembles beaucoup aux colormaps que l'on trouve avec les visuals `GDK_VISUAL_PSEUDO_COLOR`, mais attention à ne pas

[12] rouge, vert, bleu

les confondres. On peut tout à fait utiliser les images RVB indexées sur tous les types de visuals. Les palettes de couleurs sont de type `GdkRgbCmap` qui est définie ainsi :

```
typedef struct _GdkRgbCmap GdkRgbCmap;

struct _GdkRgbCmap {
  guint32 colors[256];
  guchar lut[256]; /* for 8-bit modes */
};
```

Comme on peut le voir, il s'agit simplement d'un tableau de 256 entrées qui contiennent chacun un quadulet d'octets définissant une couleur. Pour les visuals 8 bits, un tableau d'octet est utilisé à la place. Ce qui est important pour notre propos est qu'il s'agit d'un tableau de 256 définition de couleur.

Une image RVB indexée en donc constitutée de deux parties distinctes. La palette de couleur et un tableau de `guchar` où chaque entrée contient un index sur la palette de couleur. Par exemple, si une entrée du tableau de l'image contient 12, alors le pixel associée aura pour couleur la couleur définie dans l'entrée 12 de la palette de couleur.

Ainsi, pour afficher une telle image, il faut donc d'abord définir la palette et le tableau de l'image, puis appeler la fonction suivante :

```
void gdk_draw_indexed_image(GdkDrawable *drawable,
                            GdkGC *gc,
                            gint x, gint y,
                            gint largeur, gint hauteur,
                            GdkRgbDither dith,
                            guchar *image,
                            gint longueur_ligne,
                            GdkRgbCmap *palette);
```

Cependant, s'il paraît simple de créer le tableau de l'image, il faut aussi savoir comment créer la palette associée. En fait, il suffit pour cela de remplir un tableau de 256 `guint32` (des entiers non signées de 32 bits) avec les différentes couleurs. La transformation d'une couleur définie par un triplet (rouge, vert, bleu) en un `guint32` se fait grâce à la formule suivante :

```
couleur = rouge << 16 | vert << 8 | bleu
```

où `rouge`, `vert` et `bleu` peuvent prendre des valeurs de 0 à 255.

Une fois le tableau rempli avec les couleurs de notre choix, on peut créer la palette de couleur à l'aide de la fonction suivante :

```
GdkRgbCmap *gdk_rgb_cmap_new(guint32 *couleurs,
                             gint n_couleurs);
```

où `couleurs` est notre tableau de couleurs et `n_couleurs` est le nombre de couleurs que contient ce tableau, c'est-à-dire 256 la plupart du temps.

Lorsque l'on a plus beson d'une palette, on peut la détruite à l'aide de la fonction suivante :

```
void gdk_rgb_cmap_free(GdkRgbCmap *palette);
```

8.5. Les fonctions utilitaires du module gdkrgb

Le module gdbrgb n'est pas limité à l'utilisation des images RVB. Il peut aussi être très utile pour allouer facilement des couleurs.

Tout d'abord, il faut savoir que ce module déclare en interne un visual et une colormap, qu'il utilise pour ses réservation de couleurs. Donc si l'on veut utiliser gdkrgb pour allouer nos propres couleurs, il faut que nos fenêtres et nos contextes graphiques possèdent un visual et une colormap compatible avec ceux utilisés par gdkrgb. Pour cela, on peut savoir quel est le visual utilisé par gdkrgb en appelant :

```
GdkVisual *gdk_rgb_get_visual(void);
```

De la même façon, la colormap peut être obtenue avec :

```
GdkColormap *gdk_rgb_get_cmap(void);
```

À partir de là, on peut allouer directement des couleurs via gdkrgb et sa colormap avec :

```
gulong gdk_rgb_xpixel_from_rgb(guint32 rvb);
```

où `rvb` est une couleur codée sur 32 bits. La valeur renvoyée est la valeur du champ `pixel` de la couleur. Par exemple, si l'on veut obtenir une couleur jaune-orangée, on pourra utiliser :

```
GdkColor couleur;

couleur.red   = 255 << 8;
couleur.green = 127 << 8;
couleur.blue  =   0 << 8;
couleur.pixel = gdk_rgb_xpixel_from_rgb(255 << 16 |
                                        180 << 8 |
                                        0);
```

Plus pratique encore, il est possible de spécifier directement la couleur d'avant plan et d'arrière plan d'un contexte graphique directement en donnant une couleur codée sur 32 bits avec :

```
void gdk_rgb_gc_set_foreground(GdkGC *gc, guint32 rvb);
void gdk_rgb_gc_set_background(GdkGC *gc, guint32 rvb);
```

5

La gestion des événements avec le GDK

La gestion des événements est certainement la tâche la plus importante de tout programme tournant sur X-Window. En effet, ces programmes sont en général tous construits suivant le même schéma. Ils débutent par une phase d'initialisation et de construction de l'interface. Puis ils entrent dans une boucle sans fin, en attendant des événements et en réagissant suivant les types des événements.

Vous avez sans doute remarqué que, jusqu'à présent, les exemples concernants le GDK finissent tous par ces lignes :

```
for(;;)
  if (gdk_events_pending())
    gdk_event_get();
```

Celles-ci signifie en langage clair, répéter sans fin : "si des événements sont en attente, récupérons-le".

Pour bien comprendre le sens réel de ces lignes, il faut se rappeler que les événements rapportent tout ce qui peut arriver à notre application, comme par exemple : "l'utilisateur a bougé la souris" ou "cette fenêtre devrait se redessiner". Le serveur X stoque tous ces événements dans une pile de type FIFO[1], en attendant que les applications viennent les chercher. Une application ne peut récuperer dans ce tampon que les événements qui lui sont destinés. Ainsi une application peut-elle savoir si des événements sont en attente pour elle avec la fonction :

```
gboolean gdk_events_pending(void);
```

qui renvoieTRUE si des événements destinés à l'application sont en attente dans la FIFO du serveur, et `FALSE` sinon.

Si il y a effectivement un événement en attente, on peut le récupérer, et donc le retirer de la FIFO, grâce à la fonction suivante :

```
GdkEvent *gdk_event_get(void);
```

La valeur renvoyée est une structure décrivant totalement l'événement ou `NULL` s'il n'y a pas d'événements disponibles. Attention, car cette fonction crée véritablement une copie de l'événement, il faudra ensuite libérer la mémoire utilisée par cette copie en appelant la fonction suivante :

```
void gdk_event_free(GdkEvent *ev);
```

Il existe des événements qui sont entièrement traités par le GDK et qui ne peuvent donc pas être récupérés par cette fonction mais ils sont signalés par la fonction `gdk_events_pending()`. C'est pourquoi il faut toujours tester la valeur de retour de `gdk_event_get()` avant d'accéder aux champs de la structure renvoyée, même si `gdk_event_get()` renvoie `TRUE`. Le type `GdkEvent` est une union définie comme suit :

```
typedef union _GdkEvent  GdkEvent;
```

---

[1] *First In, First Out*, premier entré, premier sorti, on nomme parfois une telle structure un tampon, ou une queue

```
union _GdkEvent
{
  GdkEventType        type;
  GdkEventAny         any;
  GdkEventExpose      expose;
  GdkEventNoExpose    no_expose;
  GdkEventVisibility  visibility;
  GdkEventMotion      motion;
  GdkEventButton      button;
  GdkEventKey         key;
  GdkEventCrossing    crossing;
  GdkEventFocus       focus_change;
  GdkEventConfigure   configure;
  GdkEventProperty    property;
  GdkEventSelection   selection;
  GdkEventProximity   proximity;
  GdkEventClient      client;
  GdkEventDND         dnd;
};
```

Le premier membre de cette union définit quel est le type de l'événement qui est survenu. En examinant ce type, on peut savoir quel est véritablement le membre qu'il convient d'utiliser. Par exemple, si l'événement correspond à un mouvement de la souris, `type` contiendra la valeur `GDK_MOTION_NOTIFY` et les informations relatives à l'événement seront disponible dans le membre `motion` de la structure `GdkEvent`. Notez bien que, à cause de la structure d'union, tous les types `GdkEvent*` possède le type de l'événement comme premier champ. Ainsi, si la variable `ev` pointe sur une structure `GdkEvent`, alors

```
ev->type
ev->any.type
ev->expose.type
((GdkEventAny*)ev)->type
((GdkEventExpose*)ev)->type
```

représentent tous la même entité, c'est-à-dire le type de l'événement. Voici un tableau qui résume tous les types d'événements supportés par le GDK et membre de `GdkEvent` associés.

type	membre	origine de l'événement
GDK_NOTHING	aucun	Événement invalide
GDK_DELETE	GdkEventAny	Une fenêtre a été éffacée
GDK_DESTROY	GdkEventAny	Une fenêtre a été détruite
GDK_EXPOSE	GdkEventExpose	Une fenêtre doit être redessinée
GDK_MOTION_NOTIFY	GdkEventMotion	Le pointeur de la souris a bougé
GDK_BUTTON_PRESS	GdkEventButton	Un bouton de la souris a été pressé
GDK_2BUTTON_PRESS	GdkEventButton	Double clique
GDK_3BUTTON_PRESS	GdkEventButton	Triple clique
GDK_BUTTON_RELEASE	GdkEventButton	Un bouton de la souris a été relâché
GDK_KEY_PRESS	GdkEventKey	Une touche du clavier a été préssée
GDK_KEY_RELEASE	GdkEventKey	Une touche du clavier a été relâchée
GDK_ENTER_NOTIFY	GdkEventCrossing	La souris est entrée dans une fenêtre
GDK_LEAVE_NOTIFY	GdkEventCrossing	La souris est sortie d'une fenêtre
GDK_FOCUS_CHANGE	GdkEventFocus	Le focus clavier a changé
GDK_CONFIGURE	GdkEventConfigure	La taille d'une fenêtre a changé
GDK_MAP	GdkEventAny	Une fenêtre a été affichée
GDK_UNMAP	GdkEventAny	Une fenêtre a été cachée
GDK_PROPERTY_NOTIFY	GdkEventProperty	Une propriété d'une fenêtre a changé
GDK_SELECTION_CLEAR	GdkEventSelection	Le contenu du presse-papiers a été vidé
GDK_SELECTION_REQUEST	GdkEventSelection	Le presse-papiers demande une sélection
GDK_SELECTION_NOTIFY	GdkEventSelection	Le contenu du presse papier a changé
GDK_PROXIMITY_IN	GdkEventProximity	Le stylet d'une tablette graphique est sur la tablette
GDK_PROXIMITY_OUT	GdkEventProximity	Le stylet a quitté la tablette
GDK_DRAG_ENTER	GdkEventDND	Ces six événements sont utilisés pour
GDK_DRAG_LEAVE	GdkEventDND	gérer les glisser/déplacer par GTK
GDK_DRAG_MOTION	GdkEventDND	
GDK_DRAG_STATUS	GdkEventDND	
GDK_DROP_START	GdkEventDND	
GDK_DROP_FINISHED	GdkEventDND	
GDK_CLIENT_EVENT	GdkEventClient	Une application nous envoie cet événement
GDK_VISIBILITY_NOTIFY	GdkEventVisibility	Une fenêtre a été obscurcie ou est devenu visible
GDK_NO_EXPOSE	GdkEventNoExpose	Une opération de copie d'une zone de fenêtre a réussie

Tous les événements à partir de GDK_MAP sont en réalité très peu utilisés directement. GTK s'en sert de manière interne. Aussi, nous n'étudierons que les premiers.

Si vous regardez attentivement cette liste, vous remarquerez que les événements sont vraiment produits à tout bout de champ. Heureusement, pour limiter cette surproduction, les applications peuvent indiquer quels sont les événements qui doivent être produit pour chacune de ses fenêtres. Les autres événements sont tout simplement oublié par le serveur X.

La fonction permettant cela est :

```
void gtk_window_set_events(GdkWindow *fenetre,
                           GdkEventMask masque);
```

où `masque` est un champ de bits indiquant quels sont les événements qui doivent être envoyés à la `fenetre`.

Le tableau suivant décrit quel bit doit être employé pour sélectionner chacun des événements.

nom du bit	événement selectionné
GDK_EXPOSURE_MASK	GDK_EXPOSE
GDK_POINTER_MOTION_MASK	GDK_MOTION_NOTIFY
GDK_POINTER_MOTION_HINT_MASK	voir **FIXME PLEASE**
GDK_BUTTON_MOTION_MASK	GDK_MOTION_NOTIFY uniquement si un des boutons de la souris est pressé
GDK_BUTTON1_MOTION_MASK	GDK_MOTION_NOTIFY uniquement si le bouton 1 de la souris est pressé
GDK_BUTTON2_MOTION_MASK	GDK_MOTION_NOTIFY uniquement si le bouton 2 de la souris est pressé
GDK_BUTTON3_MOTION_MASK	GDK_MOTION_NOTIFY uniquement si le bouton 3 de la souris est pressé
GDK_BUTTON_PRESS_MASK	GDK_BUTTON_PRESS, GDK_2BUTTON_PRESS et GDK_3BUTTON_PRESS
GDK_BUTTON_RELEASE_MASK	GDK_BUTTON_RELEASE
GDK_KEY_PRESS_MASK	GDK_KEY_PRESS
GDK_KEY_RELEASE_MASK	GDK_KEY_RELEASE
GDK_ENTER_NOTIFY_MASK	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY_MASK	GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE_MASK	GDK_FOCUS_IN et GDK_FOCUS_OUT
GDK_STRUCTURE_MASK	GDK_CONFIGURE, GDK_DESTROY, GDK_MAP et GDK_UNMAP
GDK_PROPERTY_CHANGE_MASK	GDK_PROPERTY_NOTIFY
GDK_VISIBILITY_NOTIFY_MASK	GDK_VISIBILITY_NOTIFY
GDK_PROXIMITY_IN_MASK	GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT_MASK	GDK_PROXIMITY_OUT
GDK_SUBSTRUCTURE_MASK	GDK_STRUCTURE_MASK mais des fenêtres filles
GDK_ALL_EVENTS_MASK	tous les événements

1. Les événements Delete et Destroy

L'événement *Delete* est émis par le serveur X lorsqu'une fenêtre est sur le point d'être fermée. Cet événement peut être généré lorsque l'utisateur clique sur un bouton mis en place par le gestionnaire de fenêtres. Souvent, on utilise cet événement pour pouvoir éventuellement sauvegarder des données attachées à la fenêtre. C'est la responsabilité du programme de réellement détruire la fenêtre si besoin est.

Voici un exemple qui ouvre une fenêtre et filtre les événements qui arrivent. Il n'y a pas besoin de selectionner l'événement `GDK_DELETE` par un appel à `gtk_window_set_events()` car il est toujours émis par le serveur X.

```
/* gdk delete */
#include <stdio.h>
#include <gdk/gdk.h>
```

```
int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkWindowAttr attr;
  GdkEvent *ev;

  attr.event_mask = 0;
  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
    {
      ev = gdk_event_get();
      if (ev)
      {
        if (ev->type == GDK_DELETE)
          printf("Evénement delete recu\n");
        gdk_event_free(ev);
      }
    }
}
```

Si vous compilez et exécutez ce programme comme d'habitude, vous verrez qu'il fait apparaître une fenêtre. Si vous essayez de la fermer, le message :

```
"Evénement delete recu"
```

s'affichera, et la fenêtre ne sera pas fermée. Attention cependant, différents gestionnaires de fenêtres peuvent envoyer soit l'événement delete, soit l'événement *destroy*. Celui-ci est envoyé juste avant la destruction d'une fenêtre, soit par le gestionnaire de fenêtres, soit par un appel à `gdk_window_destroy()`. Cela permet aussi de sauvegarder des données en urgence, mais la fenêtre sera détruite quoi qu'il arrive.

Ces deux événements utilisent la structure `GdkEventAny` qui est définie ainsi :

```
typedef struct _GdkEventAny  GdkEventAny;

struct _GdkEventAny
```

```
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
};
```

Le premier champ, `type`, obligatoire dans tous les événements contient, pour ces événements, `GDK_DELETE` ou `GDK_DESTROY`. Le champ `window` est un pointeur vers la fenêtre qui doit être fermée ou détruite. Le champ `send_event` est un indicateur qui est à `TRUE` si l'événement a été émis par un programme et `FALSE` s'il a été émis par le serveur. Ces trois champs sont également présents dans toutes les structures d'événements.

2. Les événements Expose et Configure

Un événement *expose* est émis à chaque fois qu'une fenêtre ou une partie de fenêtre doit être redessinée. Cela arrive notamment lorsque la fenêtre est affichée par un `gdk_window_show()` ou lorsque la fenêtre était cachée par d'autres et qu'elle repasse au premier plan. Les événements exposes sont envoyé à l'aide de la structure suivante :

```
typedef struct _GdkEventExpose  GdkEventExpose;

struct _GdkEventExpose
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  GdkRectangle area;
  gint count;
};
```

où les trois premiers champs sont les mêmes que pour `GdkEventAny` puisque ces champs sont communs à tous les événements. Le champ `area` définit la zone de la fenêtre qui doit être redessinée. En effet, il peut être avantageux de ne dessiner que la partie manquante, surtout si la fenêtre contient des dessins complexes. Et le champ `count` désigne le nombre d'événements exposes qui suivent celui-ci. En effet, imaginez qu'une fenêtre soit recouverte par deux autres et qu'elle replace au premier plan. Il y aura alors deux rectangles à redessiner. Dans des situations plus étranges, on peut avoir à redessiner une multitude de rectangles à redessiner.

Donc pour traiter un événement expose, on dispose de deux stratégies. Soit on redessine chaque rectangle pour chaque événement expose en récupérant les dimensions du rectangle dans le champ `area`. Soit on ignore simplement tous les événements dont le champ `count` n'est pas nul et on attend le dernier de la série pour redessiner la fenêtre au grand complet. Le choix d'une stratégie dépend entièrement de ce qu'il y a à redessiner dans la fenêtre.

Un autre événement lié à une fenêtre dans son ensemble est celui lié à son redimensionnement et à sa position. Cela arrive à chaque fois qu'un utilisateur change la taille ou la position d'une fenêtre à l'aide de la souris ou en utilisant les possibilités du gestionnaire de fenêtres. C'est aussi le cas si la fenêtre change de place ou de taille par un appel à `gdk_window_move_resize()` par exemple. Cet événement s'appelle *configure*. Il est relié à la structure suivante :

```
typedef struct _GdkEventConfigure  GdkEventConfigure;

struct _GdkEventConfigure
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  gint16 x, y;
  gint16 width;
  gint16 height;
};
```

La nouvelle position de la fenêtre à l'intérieur de sa fenêtre mère est donnée dans les champs `x` et `y`. Et les nouvelles largeur et hauteur sont transmises dans les champs `width` et `height`. Cela permet de mettre à jour des paramètres de notre application. Ainsi au prochain événement expose, on connaîtra la taille de la fenêtre. Notez que lors l'affichage d'une fenêtre, à l'aide de la fonction `gdk_window_show()` par exemple, la fenêtre reçoit un événement configure suivit d'un événement expose.

Voici un exemple qui illustre l'utilisation des événements expose et configure :

```
/* gdk expose */
#include <stdio.h>
#include <gdk/gdk.h>

int largeur = 150;
int hauteur = 150;

/*
 * Procédure de traitement des événements
 * de type configure
 */
void TraitementConfigure(GdkEventConfigure *ev)
{
  largeur = ev->width;
  hauteur = ev->height;
  printf("Nouvelle largeur : %d\n", largeur);
  printf("Nouvelle hauteur : %d\n", hauteur);
}
```

```
/*
 * Procédure de traitement des événements
 * de type expose
 */
void TraitementExpose(GdkEventExpose *ev)
{
  GdkGC *gc;
  GdkColormap *colormap;
  GdkColor couleur;

  /* on ne redessine la fenêtre que si c'est le dernier
   * événement de la série. */
  if (ev->count > 0)
    return;
  gc = gdk_gc_new(ev->window);
  colormap = gdk_colormap_get_system();
  gdk_color_white(colormap, &couleur);
  gdk_gc_set_foreground(gc, &couleur);
  gdk_draw_arc(ev->window, gc, TRUE,
               0,0, largeur,hauteur,
               0,64*360);
  gdk_gc_destroy(gc);
}

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkWindowAttr attr;
  GdkEvent *ev;

  attr.width = largeur;
  attr.height = hauteur;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  attr.event_mask = GDK_EXPOSURE_MASK;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  for(;;)
    if (gdk_events_pending())
    {
      ev = gdk_event_get();
```

```
      if (ev)
      {
        if (ev->type == GDK_EXPOSE)
        {
          printf("Evénement expose recu\n");
          TraitementExpose((GdkEventExpose *)ev);
        }
        else if (ev->type == GDK_CONFIGURE)
        {
          printf("Evénement configure recu\n");
          TraitementConfigure((GdkEventConfigure *)ev);
        }
        gdk_event_free(ev);
      }
    }
}
```

Cette exemple affiche une fenêtre dans laquelle une ellipse blanche est dessinée. La largeur et la hauteur de l'ellipse sont déterminées en récupérant ces informations lors de la reception d'événements configure. L'ellipse est dessinée chaque fois que nécessaire, c'est-à-dire à chaque reception de l'événement expose.

3. Les mouvements de la souris

Le périphérique générant le plus d'événement est très certainement la souris. Après tout l'environnement X-Window a été conçu surtout pour profiter des avantages de cet outil très pratique.

En fait, il suffit de déplacer la souris pour déclencher des événements. Il existe deux types d'événements liés aux mouvements de la souris. Les premiers sont envoyés si la souris quitte ou entre dans une fenêtre, les seconds sont relatifs aux mouvements de la souris à l'intérieur d'une fenêtre.

3.1. Les événements enter et leave

Si, au moment de la création de la fenêtre, on a inclu les bits `GDK_ENTER_NOTIFY_MASK` et `GDK_LEAVE_NOTIFY_MASK` alors elle recevra un événement de ce type à chaque fois que le pointeur de la souris entrera dans la fenêtre ou en sortira. Ces deux événements sont associés à la structure suivante :

```
typedef struct _GdkEventCrossing  GdkEventCrossing;

struct _GdkEventCrossing
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  GdkWindow *subwindow;
```

```
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble x_root;
  gdouble y_root;
  GdkCrossingMode mode;
  GdkNotifyType detail;
  gboolean focus;
  guint state;
};
```

Les trois premiers champs de cette structure doivent commencer à vous être familiers. Le champ `subwindow` est plus nouveau. Si l'événement concerne aussi une fenêtre inclue dans celle qui recoit l'événement alors `subwindow` contient un pointeur sur cette fenêtre fille, sinon il contient `NULL`. Cela arrive lorsque le pointeur de la souris était dans une fenêtre inclue dans une autre et qu'il sort rapidement des deux à la fois. Dans ce cas, la fenêtre fille reçoit également l'événement. Le champ `time` qui est aussi présent dans de nombreuses structures d'événements, indique à quel moment à eu lieu l'événement. Il est très peu utilisé. Les champs `x` et `y` sont les coordonnées du pointeur de la souris dans la fenêtre au moment où l'événement a eu lieu, alors que les champs `x_root` et `y_root` sont les coordonnées du pointeur de la souris dans l'écran. Le champ `mode` est un peu plus complexe à comprendre. Il peut prendre les valeurs suivantes :

- `GDK_CROSSING_NORMAL` qui indique que l'événement a été produit dans des conditions habituelles,
- `GDK_CROSSING_GRAB` qui indique que l'événement est la conséquence d'une capture de la souris[2]
- `GDK_CROSSING_UNGRAB` qui indique que l'événement est la conséquence du relâchement d'une capture de la souris. Le champ `detail` indique comment s'est produit l'événement enter ou leave, si la souris est passé d'une fenêtre à sa fille, d'une fenêtre à sa mère, d'une fenêtre à une de ses soeurs ou autre. Il est en général complètement inutile. Le champ `focus` indique si la fenêtre qui reçoit l'événement possède le focus du clavier ou nom. Le champ `state` indique quels sont les boutons de la souris et les modifieurs du clavier qui sont enfoncés ou actifs lors de l'événement. Il s'agit d'un champ de bits dont la signification est la même que celle du dernier paramètre de la fonction `gdk_window_get_pointer()` que nous avons vue dans le paragraphe **FIXME PLEASE**.

Les événements enter et leave ne sont pas souvent utiliser directement par les applications. En revanche, GTK+ s'en sert pour mettre en surbrillance les widgets lorsque le pointeur de la souris passe sur eux.

3.2. Les déplacements de la souris dans une fenêtre

Les déplacement de la souris à l'intérieur d'une fenêtre genère des événements

[2] voir paragraphe **FIXME PLEASE**

dont le type est `GDK_MOTION_NOTIFY`. La manière la plus simple de recevoir cet événement est de sélectionner le bit `GDK_POINTER_MOTION_MASK` dans le masque des événements lors de la création de la fenêtre. Ainsi, nous recevrons autant d'événements de mouvements de souris que le serveur X pour en produire dès que la souris bougera sur notre fenêtre. Si notre application est capable de les traiter très rapidement cela ne devrait pas poser de problème, sinon elle risque d'être très vite submergée. Aussi, s'il n'est pas trop grave de rater quelques événements, il vaut mieux sélectionner également le bit `GDK_POINTER_MOTION_HINT_MASK`. Dans ce cas, le serveur X attendra que notre application fasse un appel à `gdk_window_get_pointer()` avant de générer un nouvel événement de déplacement de la souris. C'est en général ce que l'on veut.

De plus on peut encore plus limiter le nombre d'événements générés si l'on n'est intéressé par les mouvements de la souris que lorsque un bouton est pressé en utilisant `GDK_BUTTON_MOTION_MASK`, `GDK_BUTTON1_MOTION_MASK`, `GDK_BUTTON2_MOTION_MASK`, ou `GDK_BUTTON3_MOTION_MASK` à la place de `GDK_POINTER_MOTION_MASK`.

Les événements de déplacement du pointeur de la souris sont transmis sous la forme d'une structure `GdkEventMotion` définie ainsi :

```
typedef struct _GdkEventMotion  GdkEventMotion;

struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  gint16 is_hint;
  GdkInputSource source;
  guint32 deviceid;
  gdouble x_root, y_root;
};
```

Les trois premiers champs de cette structure sont les champs communs à tous les événements. Le champ `time` est, comme pour les événements enter et leave, une indication du moment où a eu lieu d'événement. Les champs `x`, `y`, `x_root` et `y_root` ont également la même signification, c'est-à-dire qu'ils contiennent les coordonnées du pointeur de la souris respectivement par rapport à la fenêtre et par rapport à l'écran. Cependant si le bit `GDK_POINTER_MOTION_HINT_MASK` a été sélectionner, il vaut mieux récupérer les coordonnées grâce à la fonction `gdk_window_get_pointer()`,

car elle donnera la position actuelle du pointeur et pas celle au moment de l'émission de l'événenement. De plus, le serveur X ne générera pas d'autre événenement de mouvement de souris pour cette fenêtre si vous n'appelez pas cette fonction. Les champs `pressure`, `xtilt` et `ytilt` ne sont utilisés que si le périphérique de pointage est une tablette graphique et non une souris. Ils indiquent quelle est la pression exercée sur le stylet ainsi que son inclinaison pendant l'événement. Pour savoir quel est le type périphérique de pointage utilisé, il faut examiner le champ `source` qui ne peut contenir que les valeurs suivantes :

- `GDK_SOURCE_MOUSE` s'il s'agit d'une souris ;
- `GDK_SOURCE_PEN` s'il s'agit du stylet d'une tablette graphique ;
- `GDK_SOURCE_ERASER` s'il s'agit de la gomme d'une tablette graphique ;
- `GDK_SOURCE_CURSOR` s'il s'agit d'un périphérique de pointage d'un autre type.

S'il y a plusieurs périphériques de pointages connectés au serveur X, celui ayant provoqué l'événement est spécifié dans le champ `deviceid`. Par la suite, je supposerais toujours que le périphérique de pointage est une souris, l'étude des possibilités des tablettes graphiques dépassant de loin le cadre de cet ouvrage. Le champ `is_hint` indique si le bit `GDK_POINTER_MOTION_HINT_MASK` a été positionné ou non, cela permet de savoir s'il faut appeler `gdk_window_get_pointer()` ou non. Enfin le champ `state` contient un champ de bits contenant l'état des boutons de la souris et des modifieurs du claviers au moment de l'événement, comme pour enter et leave.

Il arrive que l'on ait besoin de recevoir les événements relatifs aux déplacements de la souris, même lorsque celle-ci est en dehors de la fenêtre. Ceci peut être utile, par exemple dans un programme de dessin pour dessiner une ligne dont une des extrémités est en dehors de la fenêtre. Pour cela, il faut *capturer* le pointeur à l'aide de la fonction suivante :

```
gint gdk_pointer_grab(GdkWindow    *fenetre,
                      gint          destinataire,
                      GdkEventMask  masque,
                      GdkWindow    *confiner_a,
                      GdkCursor    *curseur,
                      guint32       moment);
```

où `fenetre` est la fenêtre qui capture la souris, c'est-à-dire que c'est elle qui va recevoir tous les prochains événements de la souris. Le paramètre `destinataire` indique si les événements doivent être transmis tels quels ou s'il doivent tous être redirigés vers la fenêtre. `masque` est le masque des événements que l'on accepte de traiter pendant le temps où la souris sera captive. Le paramètre `confiner_a`, s'il est non nul, indique que le pointeur de la souris ne pourra pas dépasser les limites de cette fenêtre pendant la durée de la capture. Cela peut être très génant pour l'utilisateur, il vaut donc mieux réserver l'usage de ce paramètre aux cas où il est vraiment nécessaire de confiner la souris dans une fenêtre particulière. En général, il vaut mieux confiner la souris dans la fenêtre qui capture la souris. Le paramètre `moment` est un temps comme celui que l'on peut trouver dans les champs `time` des

structures d'événements. Il ne sert pas à grand chose mais doit pourtant absolument être présent pour des raisons de compatibilités avec la Xlib.

Le paramètre `curseur`, s'il est non nul, permet de définir un curseur spécial qui sera utilisé pour représenter l'emplacement du pointeur de la souris pendant toute la durée de la capture.

Un tel curseur peut être créé soit avec :

```
GdkCursor *gdk_cursor_new(GdkCursorType type);
```

où `type` est l'un des nombreux types de curseurs disponibles que l'on peut trouver dans `<gdk/gdkcursors.h>`, soit avec :

```
GdkCursor *gdk_cursor_new_from_pixmap(GdkPixmap *source,
                                      GdkPixmap *masque,
                                      GdkColor *fg,
                                      GdkColor *bg,
                                      gint x, gint y);
```

où `source` est le bitmap définissant l'aspect du curseur, il est normalement dessiné avec les deux couleurs définies dans `fg` et `bg` peut avoir n'importe quel profondeur supportée par le serveur X. `masque` est un bitmap qui définit quels sont les pixels de `source` qui doivent être transparents. Les coordonnées `(x, y)` sont l'emplacement du point chaud du curseur.

Les curseurs doivent être détruits avec :

```
void gdk_cursor_destroy(GdkCursor *curseur);
```

après utilisation.

Tous les appels à `gdk_pointer_grab()` doivent avoir un appel à :

```
void gdk_pointer_ungrab(guint32 moment);
```

correspondant. Sinon, le serveur X reste coincé dans un état où tous les événements de déplacement de la souris sont envoyés à une même fenêtre appartenant à une même application. Ce n'est donc vraiment pas une bonne chose. L'unique argument utilisé par cette fonction est un instant comme ceux que l'on trouve dans le champs `time` des structures d'événements.

Notez que le serveur X met automatiquement en place une capture entre l'appuis et le relâchement d'un bouton de la souris, de sorte que c'est toujours la même fenêtre qui reçoit les deux événements. Comme il s'agit de la principale raison de vouloir faire une capture, il vaut mieux bien réfléchir avant d'utiliser une capture directement. C'est très rarement utile.

À chaque instant, on peut savoir si le pointeur de la souris a été capturé ou non à l'aide de la fonction suivante :

```
gint gdk_pointer_is_grabbed(void);
```

qui renvoie `TRUE` si le pointeur est actuellement capturé, et `FALSE` sinon.

4. Les boutons de la souris

Les souris déclenchent des événements par leurs déplacements mais aussi lorque l'on presse ou relâche un de leurs boutons. Les événements `GDK_BUTTON_PRESS`, `GDK_2BUTTON_PRESS`, `GDK_3BUTTON_PRESS` et `GDK_BUTTON_RELEASE` sont associés à la même structure d'événement `GdkEventButton` qui est définie comme suit :

```
typedef struct _GdkEventButton  GdkEventButton;

struct _GdkEventButton
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  guint button;
  GdkInputSource source;
  guint32 deviceid;
  gdouble x_root, y_root;
};
```

Dans cette structure, la plupart des champs sont identiques à ceux que l'on trouve dans `GdkEventMotion`. Le seul champ nouveau est `button`. Ce champ permet de savoir quel est le bouton qui a été pressé ou relâché. Il vaut 1 si c'est le bouton gauche de la souris qui a été pressé, 2 pour le bouton du milieu et 3 pour le bouton de droite. Notez que lors d'un double clique, l'événement `GDK_2BUTTON_PRESS` n'est pas le seul reçu. En effet, voici tous les événements qui sont générés :

- `GDK_BUTTON_PRESS` quand on appuie une première fois sur le bouton ;
- `GDK_BUTTON_RELEASE` quand on relâche une première fois le bouton ;
- `GDK_BUTTON_PRESS` quand on appuie une seconde fois sur le bouton ;
- `GDK_2BUTTON_PRESS` pour signifier le double clique ;
- `GDK_BUTTON_RELEASE` quand on relâche une seconde fois le bouton.

Voici un exemple qui utilise les événements de gestion de la souris :

```
/* gdk souris */
#include <gdk/gdk.h>

#define NOMBRE_LIGNE (50)
typedef struct
{
```

```
  gint x1, y1, x2, y2;
} TypeLigne;

int NbLignes = 0;
int Index = 0;
TypeLigne Lignes[NOMBRE_LIGNE];

/*
 * Procédure de traitement des événements
 * de type expose
 */
void TraitementExpose(GdkEventExpose *ev, gboolean force)
{
  GdkGC *gc;
  GdkColormap *colormap;
  GdkColor couleur;
  int i;

  /* Si ce n'est pas un appel depuis
   * TraitementMotion et que ce n'est pas le
   * dernier événement expose de la série, alors
   * on sort immédiatement
   */
  if (force==FALSE && ev->count > 0)
    return;
  /* On efface la fenêtre */
  gdk_window_clear(ev->window);
  gc = gdk_gc_new(ev->window);
  colormap = gdk_colormap_get_system();
  gdk_color_white(colormap, &couleur);
  gdk_gc_set_foreground(gc, &couleur);
  for (i=0 ; i<NbLignes ; i++)
    gdk_draw_line(ev->window, gc,
                  Lignes[i].x1, Lignes[i].y1,
                  Lignes[i].x2, Lignes[i].y2);
  gdk_gc_destroy(gc);
}

/*
 * Procédure de traitement des événements
 * de type Button Press
 */
void TraitementButtonPress(GdkEventButton *ev)
{
  if (NbLignes < NOMBRE_LIGNE)
```

```
    NbLignes++;
  Lignes[Index].x1 = ev->x;
  Lignes[Index].y1 = ev->y;
}

/*
 * Procédure de traitement des événements
 * de type mouvement de souris
 */
void TraitementMotion(GdkEventMotion *ev)
{
  int x, y;
  GdkModifierType mod;

  gdk_window_get_pointer(ev->window, &x, &y, &mod);
  Lignes[Index].x2 = x;
  Lignes[Index].y2 = y;
  /* on force le réaffichage */
  TraitementExpose((GdkEventExpose *)ev, TRUE);
}

/*
 * Procédure de traitement des événements
 * de type Button Release
 */
void TraitementButtonRelease(GdkEventButton *ev)
{
  Lignes[Index].x2 = ev->x;
  Lignes[Index].y2 = ev->y;
  /* on force le réaffichage */
  TraitementExpose((GdkEventExpose *)ev, TRUE);
  Index++;
  Index %= NOMBRE_LIGNE;
}

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkWindowAttr attr;
  GdkEvent *ev;
  gboolean fini = FALSE;

  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
```

```
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  attr.event_mask = GDK_EXPOSURE_MASK |
                    GDK_BUTTON_MOTION_MASK |
                    GDK_POINTER_MOTION_HINT_MASK |
                    GDK_BUTTON_PRESS_MASK |
                    GDK_BUTTON_RELEASE_MASK;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  while ( !fini)
    if (gdk_events_pending())
    {
      ev = gdk_event_get();
      if (ev)
      {
        switch (ev->type)
        {
          case GDK_DELETE:
            /* Si on recoit l'événement delete,
             * on met fin à l'application
             */
            fini = TRUE;
            break;
          case GDK_EXPOSE:
            TraitementExpose((GdkEventExpose *)ev, FALSE);
            break;
          case GDK_BUTTON_PRESS:
            TraitementButtonPress((GdkEventButton *)ev);
            break;
          case GDK_MOTION_NOTIFY:
            TraitementMotion((GdkEventMotion *)ev);
            break;
          case GDK_BUTTON_RELEASE:
            TraitementButtonRelease((GdkEventButton *)ev);
          default:
            ;
        }
        gdk_event_free(ev);
      }
    }
  return 0;
}
```

Ce programme crée une fenêtre prête à recevoir les événements de gestion de la souris et l'événement expose. Si l'on clique dans la fenêtre et que l'on déplace la souris avec un bouton appuyé, on trace une ligne qui restera affichée par la suite. Le programme gère ainsi 50 lignes. Ces lignes sont rangées dans un tableau afin de pouvoir être redessiner sans problème par la fonction `TraitementExpose()`. Plusieurs choses sont intéressantes à remarquer dans ce programme. Premièrement, ce programme est le premier présenté dans ce livre qui puisse être terminé autrement qu'en le *killant* depuis un shell. En effet, comme l'événement `GDK_DELETE` est surveillé, il suffit de fermer la fenêtre pour mettre fin à l'application. De plus, on commence à entrevoir comment sont traités les différents événements. En fait une fonction différente est appelée suivant le type de l'événement. Si l'application avait eu plusieurs fenêtres, il aurait peut-être aussi fallu créer différentes fonctions suivant les fenêtres.

Ces fonctions que l'on appelle pour traiter un événement particulier sont appelées *fonctions de rappel* ou *callback*.

5. La gestion du clavier

La souris n'est pas le seul périphérique avec lequel l'utilisateur puisse générer des événements. Le clavier le permet aussi. Les deux événements qui peuvent être générés avec le clavier sont l'appui sur une touche (`GDK_KEY_PRESS`) et le relâchement de cette touche (`GDK_KEY_RELEASE`). Ces deux événements peuvent être sélectionnés indépendemment mais ils sont représentés par la même structure d'événement :

```
typedef struct _GdkEventKey  GdkEventKey;

struct _GdkEventKey
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  guint state;
  guint keyval;
  gint length;
  gchar *string;
};
```

Les cinq premiers champs de cette structure ont le même rôle que dans les autres événements. Le premier champ intéressant est `keyval` qui est le numéro de la touche qui a été pressée ou relâchée. Cette valeur est utilisée en interne par le serveur X et peut avoir d'autres utilités comme nous allons le voir. Le champ `string` contient la chaîne de caractères correspondant à la touche. Cette chaîne contient en général un seul caractère, comme par exemple `"g"` si la touche appuyée ou relâchée est la touche *g*. Cependant, cette chaîne peut être plus longue, si l'on a affecté différemment les touches. Cela est fréquemment le cas pour les touches de fonctions. Cette chaîne

peut aussi être vide si la touche est une touche de direction par exemple. La taille de cette chaîne est transmise dans le champ `length`.

Pour les touches qui ne renvoie pas de chaîne, on peut tout de même retrouver une chaîne de caractère donnant le nom symbolique de la touche avec :

```
gchar *gdk_keyval_name(guint keyval);
```

où `keyval` est la valeur renvoyée dans le champ du même nom. Bien entendu, si `keyval` correspond à une touche normale, comme une lettre ou un chiffre, `gdk_keyval_name()` renverra la même chaîne que celle présente dans le champ `string`.

Voici un exemple qui affiche toutes les touches qui sont pressées, soit en affichant directement la chaîne présente dans le champ `string` soit en utilisant `gdk_keyval_name()` pour les touches symboliques :

```
/* gdk clavier */
#include <stdio.h>
#include <gdk/gdk.h>

/*
 * Procédure de traitement des événements
 * de type Key Press
 */
void TraitementKeyPress(GdkEventKey *ev)
{
  if (ev->length > 0)
    printf("Touche normale pressée : %s\n",
           ev->string);
  else
    printf("Touche symbolique pressée : %s\n",
           gdk_keyval_name(ev->keyval));
}

int main(int argc, char *argv[])
{
  GdkWindow *Fenetre;
  GdkWindowAttr attr;
  GdkEvent *ev;
  gboolean fini = FALSE;

  attr.width = 150;
  attr.height = 150;
  attr.wclass = GDK_INPUT_OUTPUT;
  attr.window_type = GDK_WINDOW_TOPLEVEL;
  attr.event_mask = GDK_KEY_PRESS_MASK;
  gdk_init(&argc, &argv);
  /* Création de la fenêtre */
```

```
  Fenetre = gdk_window_new(NULL, &attr, 0);
  /* Affichage */
  gdk_window_show(Fenetre);
  /* Boucle d'attente des événements */
  while ( !fini)
    if (gdk_events_pending())
    {
      ev = gdk_event_get();
      if (ev)
      {
        switch (ev->type)
        {
          case GDK_DELETE:
            /* Si on recoit l'événement delete,
             * on met fin à l'application
             */
            fini = TRUE;
            break;
          case GDK_KEY_PRESS:
            TraitementKeyPress((GdkEventKey *)ev);
            break;
          default:
            ;
        }
        gdk_event_free(ev);
      }
    }
  return 0;
}
```

Si vous utilisez ce programme, vous allez remarquer que si l'on laisse une touche appuyée pendant un certain temps, plusieurs événements `GDK_KEY_PRESS` sont émis à la suite. C'est en général le comportement souhaité, mais on peut supprimer la répétition automatique par un appel à la fonction :

```
void gdk_key_repeat_disable(void);
```

On pourra de la même façon rétablir cette répétition automatique avec :

```
void gdk_key_repeat_restore(void);
```

Vous avez sans doute remarqué qu'il faut absoluement que le pointeur de la souris soit dans la fenêtre pour que notre application puisse recevoir les événements `GDK_KEY_PRESS`. C'est tout à fait normal, puisque tous les événements sont envoyés à la fenêtre sur laquelle le pointeur est. Mais si cela est naturel pour les événements liés à la souris, ça l'est beaucoup moins pour ceux liés au clavier. On veut souvent diriger tous les événements clavier vers une seule et même fenêtre, quelle que soit la

position de la souris. Pour cela, il faut *capturer* le clavier, un peu de la façon dont on capture la souris. Une capture des événements clavier peut être réalisée avec :

```
gint gdk_keyboard_grab(GdkWindow *fenetre,
                       gint owner_events,
                       guint32 moment);
```

où `fenetre` est la fenêtre qui recevra dorénavant tous les événements clavier, `destinataire` et `moment` ont le même rôle que pour la capture du pointeur de la souris. Bien entendu, le gestionnaire de fenêtre reste le seul maître pour donner le focus clavier à la fenêtre top-level choisie par l'utilisateur. Mais à partir du moment où le focus sera donné à la fenêtre dans laquelle `fenetre` est incluse, c'est elle et uniquement elle qui recevra les événements `GDK_KEY_PRESS` et `GDK_KEY_RELEASE`. Ceci n'est donc utile que si plusieurs fenêtres sont incluse dans une fenêtre top-level, comme dans une boîte de dialogue par exemple. Comme pour les captures du pointeur de la souris, il ne faut pas oublier de relâcher la capture du clavier par un appel à :

```
void gdk_keyboard_ungrab(guint32 moment);
```

6. Les autres fonctions relatives aux événements

Vous savez déjà que l'on récupère le prochain événement destiné à notre application avec `gdk_event_get()`. Ceci exécute en fait deux actions distinctes. La première est de copier l'événement et la deuxième de le supprimer de la FIFO que le serveur maintien pour nous. Si l'on veut connaître le contenu du prochain événement sans l'ôter de la FIFO, on peut utiliser la fonction suivante :

```
GdkEvent *gdk_event_peek(void);
```

Cela peut être utile pour traiter deux événements simultanément. GTK s'en sert pour réduire le nombre d'événements de type `GDK_EXPOSE`. Il regarde d'abord s'il y en a plusieurs consécutifs et les remplace par un unique dans certains cas. S'il réussit son coup, il supprime les événements de la FIFO, et en remet un autre à la place à l'aide de la fonction :

```
void gdk_event_put(GdkEvent *ev);
```

qui permet de replacer un événement quelconque dans la FIFO.

Les événements récupérés avec `gdk_event_peek()` doivent eux aussi être libérés par un appel à `gdk_event_free()`.

Lorsque l'on doit travailler longuement avec des événements, on est vite amenés à devoir en faire des copies, pour cela, le mieux est d'utiliser la fonction :

```
GdkEvent *gdk_event_copy(GdkEvent *ev);
```

qui réalise cette copie correctement, quelque soit le type de l'événement et de façon très rapide. Bien entendu, les événements copiés de cette façon doivent eux aussi être libérés par un appel à `gdk_event_free()` après utilisation.

Il peut être assez pénible voire difficile de débogguer des applications utilisant le GDK. Aussi, il est possible de demander au GDK d'afficher tous les événements avec leur principaux champs gràce à la fonction suivante :

```
void gdk_set_show_events(gint montre);
```

Si `montre` est `TRUE`, alors les événements seront affichés, s'il vaut `FALSE` les événements ne seront plus affichés. À tout moment, on peut savoir si GDK est ou non dans le mode où il affiche les événements via la fonction :

```
gint gdk_get_show_events(void);
```

Après le clavier et la souris, il existe une dernière source d'événements, qui est plutôt à part. Il s'agit des descripteurs de fichiers, qui peuvent provenir soit de fichiers ouverts, soit de tubes (*pipes* en anglais), soit de sockets. Les événements qui peuvent être émis depuis un descripteur de fichier sont de trois types. Soit des données sont arrivées et l'on peut les lires, soit le descripteur est à nouveau prêt pour une écriture, soit le descripteur de fichier n'est plus disponible, parce qu'il a été fermé par exemple.

La gestion de ces événements se fait par le biais de la fonction suivante :

```
gint gdk_input_add(gint source,
                   GdkInputCondition condition,
                   GdkInputFunction fonction,
                   gpointer donnee);
```

`source` est un descripteur de fichier, qui peut être obtenu par un appel à `open()` ou à `pipe()` par exemple. C'est ce descripteur qui sera surveillé. `condition` est le type d'événement qui sera scruté. `condition` est un champ de 3 bits :

- le bit `GDK_INPUT_READ` indique que l'on désire être prévenu si l'on peut lire de nouvelles données depuis le descripteur `source` ;
- le bit `GDK_INPUT_WRITE` indique que l'on désire être prévenu si l'on peut écrire de nouvelles données dans le descripteur ;
- le bit `GDK_INPUT_EXCEPTION` indique que l'on désire être prévenu si une execption relative au descripteur de fichier survient.

`fonction` est la fonction qui sera appelée automatiquement à chaque fois que l'une des conditions précisées dans `condition` se réalisera et que l'on appelera `gdk_event_get()` ou `gdk_event_peek()`.

Cette fonction doit avoir un prototype compatible avec le suivant :

```
void fonction(gpointer data, gint source,
              GdkInputCondition condition);
```

`data` est une donnée qui sera envoyée automatiquement. C'est celle qui est précisée dans le paramètre `donnee` de la fonction `gdk_input_add()`. `source` est le descripteur de fichier qui à déclencher l'événement et `condition` est la condition qui a fait surgir l'événement. En fait à chaque fois que l'on appelle `gdk_event_peek()` ou `gdk_event_get()`, GDK regarde quel est l'état de chaque descripteur de fichier qu'on lui a demandé de surveillé, et si l'un d'entre eux vérifie l'une des conditions demandées, il appelle la fonction `fonction` avec les paramètres adéquats. La valeur

renvoyée par `gdk_input_add()` est un identificateur qui devra être utilisé lorsque l'on voudra que le GDK cesse sa surveillance en appelant :

```
void gdk_input_remove(gint id);
```

Il existe une version un peu plus complète de la fonction `gdk_input_add()`. Il s'agit de :

```
gint gdk_input_add_full(gint source,
                        GdkInputCondition condition,
                        GdkInputFunction fonction,
                        gpointer donnee,
                        GdkDestroyNotify  destroy);
```

où le paramètre supplémentaire, `destroy`, s'il n'est pas `NULL`, est un pointeur sur une fonction dont le prototype doit être compatible avec le suivant :

```
void destroy(gpointer data);
```

Cette fonction sera automatiquement appelée (avec le même paramètre `data` que `fonction`) lorsque l'on appelera `gdk_input_remove()`. Cela peut être pratique pour "faire le ménage" en libérant de la mémoire utilisée ou autre.

7. Quelques autres fonctions utiles du GDK

Vous savez déjà que beaucoup d'événements transmettent le moment en millisecondes depuis le lancement du serveur X dans le champ `time`. On peut aussi connaître ce temps directement en appelant :

```
guint32 gdk_time_get(void);
```

Les dimensions de l'écran peuvent être obtenues avec :

- `gint gdk_screen_width(void);` qui renvoie la largeur de l'écran en pixels ;
- `gint gdk_screen_height(void);` qui renvoie la hauteur de l'écran en pixels ;
- `gint gdk_screen_width_mm(void);` qui renvoie la largeur de l'écran en millimètres ;
- `gint gdk_screen_height_mm(void);` qui renvoie la hauteur de l'écran en millimètrs.

Un bip peut être émis à tout moment pour signaler une erreur ou une opération interdite que l'utilisateur essaie d'effectuer grâce à la fonction :

```
void gdk_beep(void);
```

Nous avons vu que le transfert des informations entre le programme et le serveur X se fait en général de manière asynchrone. C'est-à-dire que les appels aux fonctions du serveur X sont placés dans un tampon que le serveur X vient lire dès qu'il en a la possibilité. Ce comportement n'est pas toujours souhaitable. Aussi, lorsque l'on veut être sûr que nos appels soient pris en compte immédiatement, on peut forcer

le serveur X à vider le tampon en exécutant toutes les requètes en attente avec la fonction suivante :

```
void gdk_flush(void);
```

Enfin, pour quitter un programme GDK “proprement”. C’est-à-dire en permettant au GDK de libérer toute la mémoire qu’il utilisait, et en retournant un code d’erreur au shell appelant, on peut utiliser la fonction suivante :

```
void gdk_exit(int code_erreur);
```

Cette fonction n’est pas souvent utilisée car la mémoire utilisée par notre programme sera de toute façon récupérée par le système d’exploitation à la fin de son exécution.

8. Conclusion de la deuxième partie

D’après ce que nous venons de voir, le Gimp Drawing Kit est bien plus qu’un simple wrapper autour de la Xlib.

6

Le modèle objet de GTK

1. Présentation

Le GTK ou Gimp ToolKit (boîte à outils pour Gimp) est la bibliothèque qui est au sommet du système GTK+. D'ailleurs le GTK utilise activement la Glib et le GDK. En fait, lors de l'écriture d'une application GTK+, le programmeur sera beaucoup plus souvent confronté au GTK directement qu'aux autres bibliothèques, ce qui est normal car il s'agit de la partie la plus "haut niveau". Le rôle du GTK est d'implémenter le mécanisme de gestion des widgets.

Pour cela, le GTK utilise une hiérarchie d'objets qui ressemble beaucoup à ce que l'on pourrait rencontrer dans une bibliothèque programmée en C++. L'ancêtre commun à tous les objets que le GTK sait manipuler s'appelle `GtkObject`.

Chaque classe d'objet est décrite en réalité par deux structures. La première à pour nom `Gtk` suivit du nom de l'objet, et définit l'instance de l'objet ; la seconde à pour nom `Gtk` suivit du nom de l'objet suivit de `Class` qui définit la classe de l'objet. Par exemple, pour l'objet *frame*, la structure d'instance s'appelle `GtkFrame` et la structure de classe s'appelle `GtkFrameClass`. Cela peut paraître complexe mais on s'y fait vraiment très vite.

Afin de bénéficier d'un mécanisme d'héritage en C semblable à ce qu'il pourrait être en C++, chaque structure a comme premier champ une structure de l'objet parent. Par exemple, les frames descendent directement des bins, ou si l'on préfère, les `GtkFrames` derivent des `GtkBins`, la structure `GtkFrame` est donc définie ainsi :

```
typedef struct _GtkFrame  GtkFrame;

struct _GtkFrame
{
  GtkBin bin;

  gchar *label;
  ...
};
```

Ainsi, une variable `frame` de type `GtkFrame *` peut être facilement transtypée en `GtkBin *` afin d'accéder aux champs des bins. On voit donc que les `GtkFrames` héritent des champs des `GtkBins`. Par exemple, un des champs de `GtkBin` est `child` ; on peut accéder à ce champ à partir d'un `GtkFrame` avec :

```
frame->bin.child
```

ou, mieux encore avec :

```
((GtkBin *)frame)->child
```

Il suffit donc de transtyper un pointeur pour pouvoir accéder aux champs des objets dont un autre objet dérive. Ceci est à rapprocher du mécanisme d'héritage de classes en C++.

Dans la structure d'instance d'un objet, on trouve tout ce qui est particulier à un objet comme certaines quantités qui peuvent varier d'un objet à l'autre d'une même classe, etc. En revanche, dans la structure de classe d'un objet, on trouve uniquement

ce qui est commun à tout les objets de la classe, comme l'ensemble des fonctions de traitement de ce type d'objet car ces fonctions ne varient pas à l'intérieur d'une même classe. On peut comparer cet ensemble de fonctions à la table de fonctions virtuelles des classes en C++.

L'héritage au niveau des structures de classe des objets fonctionne exactement de la même façon que pour les structures d'instances. Ainsi la structure de classes des frames est définie ainsi :

```
typedef struct _GtkFrameClass  GtkFrameClass;

struct _GtkFrameClass
{
  GtkBinClass parent_class;
  ...
};
```

Dans les faits, on est beaucoup plus souvent confronté à l'utilisation des structures d'instance qu'à celle des structures de classes des différents objets du GTK.

2. Les GtkObjects

Les `GtkObjects` sont les ancêtres de tous les objets que le GTK sait manipuler. Cela signifie que tous les objets du GTK peuvent être transtypés en `GtkObject`, et donc que toutes les fonctions que l'on peut utiliser avec des `GtkObjet` peuvent être utilisées avec n'importe quel autre type d'objet au prix d'un simple transtypage. Par exemple, la fonction :

```
void gtk_object_set_data(GtkObject *objet,
                         const gchar *clef,
                         gpointer donnee);
```

peut être utilisée sur des frames de cette manière :

```
  GtkFrame *frame;
  ...
  gtk_object_set_data((GtkObject *)frame, "ma_clef", donnee);
```

Cependant, durant le développement d'un programme, il peut être dangereux de transtyper des pointeurs d'un type à l'autre sans vérification, car cela peut amener à des plantages des difficiles à comprendre si on les utilises mal. Aussi, le GTK propose quatre macros qui permettent de sécuriser le processus de transtypage :

- `GTK_OBJECT(objet)` vérifie si `objet` peut être considéré comme un `GtkObject`, c'est-à-dire que le type de `objet` est bien un descendant de `GtkObject`, puis réalise le transtypage proprement dit. Si le transtypage n'est pas autorisé, GTK affiche un message d'alerte, mais effectue tout de même le transtypage fautif afin de laisser le programme continuer. Si le transtypage est autorise, `GTK_OBJECT(objet)` est globalement équivalent à `(GtkObject *)objet`.

- `GTK_OBJECT_CLASS(classe)` vérifie que `classe` peut être considérée comme une `GtkObjectClass` et réalise le transtypage correspondant.
- `GTK_IS_OBJECT(objet)` vérifie si `objet` est bien d'un type descendant d'un `GtkObject`. Renvoie `TRUE` si c'est le cas et `FALSE` sinon.
- `GTK_IS_OBJECT_CLASS(classe)` vérifie si `classe` est bien d'un type descendant d'un `GtkObjectClass`.

En règle générale, l'utilisation des macros `GTK_OBJECT()` et `GTK_OBJECT_CLASS()` est préférable à un transtypage simple. De plus les vérifications ne sont effectuées que si le GTK a été compilé avec les options de débogages adéquates. Il n'y a donc pas d'incidence sur la rapidité d'exécution finale du programme chez l'utilisateur.

Ces macros existent bien sûr pour tout les types d'objet que le GTK sait manipuler, par exemple pour transtyper un pointeur quelconque en un `GtkFrame`, on peut tout à fait utiliser la macro `GTK_FRAME()`.

Les `GtkObject` sont les objets abstraits que l'on instancie habituellement pas directement. Ils servent plutôt de classe de base à tous les autres types d'objet du GTK. Cependant, il est possible de créer un objet de type `GtkObject` à l'aide de la fonction suivante :

```
GtkObject* gtk_object_new(GtkType type,
                          const gchar *nom_arg,
                          ...);
```

Le premier argument de cette fonction est le type de l'objet que l'on veut créer. En effet, il est possible de créer un objet dont le type est n'importe quel descendant des GtkObjects à l'aide de cette fonction. `type` est simplement un entier qui représente le type en question. Le GTK prédéfinit quelques valeurs pour les types courant du C, mais pour connaître le nombre représentant le type d'un `GtkObject`, il faut utiliser la fonction :

```
GtkType gtk_object_get_type(void);
```

Car cette valeur ne peut pas être connue au moment de la compilation de la bibliothèque GDK.

Les valeurs correspondant aux types les plus courant du C sont données dans le tableau suivant :

valeur	type
GTK_TYPE_NONE	void
GTK_TYPE_CHAR	gchar
GTK_TYPE_UCHAR	guchar
GTK_TYPE_BOOL	gboolean
GTK_TYPE_INT	gint
GTK_TYPE_UINT	guint
GTK_TYPE_LONG	glong
GTK_TYPE_ULONG	gulong
GTK_TYPE_FLOAT	gfloat
GTK_TYPE_DOUBLE	gdouble
GTK_TYPE_STRING	gchar *
GTK_TYPE_ENUM	enum
GTK_TYPE_FLAGS	guint
GTK_TYPE_BOXED	gpointer
GTK_TYPE_POINTER	gpointer

Les arguments suivant de la fonction `gtk_object_new()` vont par deux. Le premier de la paire est le nom d'un argument que supporte les `GtkObjects`, et le second est l'argument proprement dit. La série d'arguement prend fin lorsque le premier d'une paire est `NULL`.

Par exemple, on peut créer un objet qui aura la propriété *user˙data* égale à `donnee` de cette façon :

```
objet = gtk_object_new(gtk_object_get_type(),
                       "GtkObject::user_data", donnee,
                       NULL);
```

Ce mécanisme peut sembler abscon, mais il est en réalité exprèmement puissant. Il permet, entre autres, de créer n'importe quel type d'objet avec la même fonction. par exemple on peut tout à fait créer un objet frame ainsi[1] :

```
frame = gtk_object_new(gtk_frame_get_type(),
                       "GtkFrame::label", "titre",
                       NULL);
```

Le seul problème de cette façon de procéder est qu'il faut connaître le nom et le type de chacun des arguments qu'un type d'objet peut supporter. Pour cela, On peut utiliser la fonction suivante qui donne tout ces renseignements :

```
GtkArg* gtk_object_query_args(GtkType type,
                              guint32 **etat,
                              guint *nb_args);
```

où `type` est le type de l'objet dont on veut connaître les arguments. Le plus simple est de passer le résultat de la fonction `gtk_*_get_type()` correspondante. `etat`,

[1] Rassurrez-vous, il existe heureusement des manières plus simples de créer des objets frames

s'il n'est pas égal à `NULL` contiendra au retour de la fonction un pointeur sur un tableau contenant les états des différents arguments[2]. Ces états ne sont pas très important pour nous, et je vous conseille donc de passer la valeur `NULL` comme deuxième paramètre à cette fonction. Le paramètre `nb_args` contiendra le nombre d'arguments que comprennent les objets de type `type`. Cette fonction renvoie un tableau (qu'il faudra à terme libérer de la mémoire à l'aide d'un `g_free()`) d'argument. Ainsi, si on appelle cette fonction ainsi :

```
GtkArg *args;
guint nb_args;

args = gtk_object_query_args(gtk_frame_get_type(),
                             NULL, &nb_args);
```

On aura dans `args[0].name` le nom du premier argument supporté par les objets de type frame, dans `args[0].type` le type du premier argument supporté par les frames, dans `args[1].name` le nom du second argument, dans `args[1].type` le type du second arguement, etc. Le type est indiqué par une valeur entière de type `GtkType`. Ceci n'est pas très parlant. On peut obtenir ce type sous la forme d'une chaîne de caractères plus explicite grâce à la fonction suivante :

```
gchar *gtk_type_name(GtkType type);
```

Voici un petit programme qui permet d'afficher les arguments supportés par le type d'objet frame :

```
/* Affiche argument */
#include <stdio.h>
#include <gtk/gtk.h>

void main(int argc, char *argv[])
{
  GtkType type;
  GtkArg *args;
  GtkWidget *Frame;
  guint nb_args, i;

  /* Initialisation des bibliothèques de GTK+ */
  gtk_init(&argc, &argv);
  Frame = gtk_frame_new("");
  type = gtk_frame_get_type();
  /* Récupération de la liste des arguments supportés */
  args = gtk_object_query_args(type, NULL, &nb_args);
  /* Affichage */
  printf("Arguments des %s : \n",
```

[2] Certains arguments ne peuvent être que lu, d'autres ne sont accéssible qu'en écriture. Ce tableau permet de savoir ce genre d'informations.

```
        gtk_type_name(type));
  for (i=0 ; i<nb_args ; i++)
  {
    printf("  %s, de type %s\n",
           args[i].name,
           gtk_type_name(args[i].type));
  }
  /* Libération de la mémoire utilisée */
  g_free(args);
}
```

En changeant simplement la ligne :

```
type = gtk_frame_get_type();
```

on peut trouver quels sont les arguments accepter par n'importe quel objet du GTK. La ligne

```
Frame = gtk_frame_new("");
```

est une astuce qui oblige le GTK à créer le type des objets frames, car il ne crée un type que lorsqu'il en a besoin.

Notez que chaque objet du GTK accepte aussi les arguments des objets dont il hérite. Cependant, `gtk_object_query_args()` ne renvoit que les argument acceptés par les objets dont le type est passé en paramètre. Pour vraiment connaître l'ensemble des arguments que l'on peut utiliser lors de la création d'un objet, il faut donc que l'on puisse connaître les types de tous les parents d'un type donné. Cela peut se faire à l'aide de la fonction suivante :

```
GtkType gtk_type_parent(GtkType type);
```

Cette fonction renvoie le type parent du type passé en paramètre s'il existe, sinon, elle renvoie la valeur `GTK_TYPE_INVALID`. Ainsi, en interrogeant `gtk_object_query_args()` pour chaque parent du type de l'objet concerné.

Voici une adaptation du programme précédent qui prend le mécanisme d'héritage en compte, c'est-à-dire que ce programme affiche non seulement les arguments que comprends les frame en particulier, mais aussi tout ceux que comprennent les objets parents du type frame. L'ensemble de ces arguments peu donc être utilisé lors de la création d'un objet frame par la fonction `gtk_object_new()`.

```
/* Affiche argument */
#include <stdio.h>
#include <gtk/gtk.h>

void main(int argc, char *argv[])
{
  GtkType type;
  GtkArg *args;
  GtkWidget *Frame;
```

```
  guint nb_args, i;

  /* Initialisation des bibliothèques de GTK+ */
  gtk_init(&argc, &argv);
  Frame = gtk_frame_new("");
  type = gtk_frame_get_type();
  /* Affichage */
  printf("Arguments des %s : \n",
         gtk_type_name(type));
  while(type != GTK_TYPE_INVALID)
  {
    /* Récupération de la liste des arguments supportés */
    args = gtk_object_query_args(type, NULL, &nb_args);
    for (i=0 ; i<nb_args ; i++)
    {
      printf("  %s, de type %s\n",
             args[i].name,
             gtk_type_name(args[i].type));
    }
    type = gtk_type_parent(type);
    /* Libération de la mémoire utilisée */
    g_free(args);
  }
}
```

Lorsque l'on n'a plus besoin d'un `GtkObject` et que l'on désire rendre la mémoire qu'il occupe à nouveau disponible pour le système, on peut le détruire grâce à la fonction :

```
void gtk_object_destroy(GtkObject *objet);
```

En cours d'utilisation d'un objet (c'est-à-dire entre le `gtk_object_new()` et le `gtk_object_destroy()`), on peut connaître l'état de l'ensemble des arguments d'un objet particulier à l'ai de la fonction suivante :

```
void gtk_object_getv(GtkObject *objet,
                     guint nb_args,
                     GtkArg *args);
```

où `nb_args` est le nombre d'arguments supportés par l'`objet` et `args` est un tableau de `nb_args` éléments de type `GtkArg`. Chaque élément de ce tableau reflète l'état d'un argument. Par exemple, à l'issu de cet appel, `args[0].type` contiendra le numéro du type du premier argument, `args[0].name` contiendra le nom du premier argument et la valeur du premier argument ne pourra être obtenu qu'en utilisant une macro particulière dépendant du type de l'argument. Par exemple si le type de l'argument est `gchar`, alors la valeur actuelle du premier argument pourra être récupérée par `GTK_VALUE_CHAR(args[0])`.

L'ensemble des macros nécessaires pour accéder à la valeur d'un argument selon son type est donné dans le tableau suivant :

type	macro
gchar	`GTK_VALUE_CHAR()`
guchar	`GTK_VALUE_UCHAR()`
gboolean	`GTK_VALUE_BOOL()`
gint	`GTK_VALUE_INT()`
guint	`GTK_VALUE_UINT()`
glong	`GTK_VALUE_LONG()`
gulong	`GTK_VALUE_ULONG()`
gfloat	`GTK_VALUE_FLOAT()`
gdouble	`GTK_VALUE_DOUBLE()`
gchar *	`GTK_VALUE_STRING()` [3]
enum	`GTK_VALUE_ENUM()`
gpointer	`GTK_VALUE_POINTER()`
GtkObject *	`GTK_VALUE_OBJECT()`

Si l'on préfère ne connaître la valeur que de certains arguements d'un `GtkObject`, on utilisera plutôt :

```
void gtk_object_get(GtkObject *objet,
                    const gchar *nom_arg,
                    ...);
```

qui fonctionne comme `gtk_object_new()`. Les paramètres de cette fonction vont deux par deux, le premier d'une paire étant le nom d'un des arguments de l'objet, et le second l'adresse d'une variable de type `GtkArg` où sera stoqué l'état actuel de l'arguement. L'appel de la fonction doit se terminer par `NULL`. Par exemple, pour connaître le label d'une frame, on appellera cette fonction de cette façon :

```
gtk_object_get(GTK_OBJECT(frame),
               "GtkFrame::label", arg,
               NULL);
```

Bien entendu, s'il est possible de connaître l'état d'un argument, il est aussi possible de changer cet état à l'aide de fonctions équivalentes. Ces fonctions sont :

```
void gtk_object_setv(GtkObject *objet,
                     guint nb_args,
                     GtkArg *args);
```

et

```
void gtk_object_set(GtkObject *objet,
                    const gchar *nom_arg,
                    ...);
```

qui fonctionnent comme `gtk_object_getv()` et `gtk_object_get()` respectivement, mais positionnent la valeur des arguments au lieu de la lire.

2.1. Les données attachées aux objets

Les GtkObjects sont des entités abstraites, qui n'existent que pour servir de base à la construction des autres types d'objets du GTK, et à la gestion des signaux comme nous allons bientôt le voir. Les GtkObjects n'ont donc pas de propriétés particulières par défaut. Cepandant, on peut attacher des données à un objet grâce, notamment à la fonction suivante :

```
void gtk_object_set_data(GtkObject *objet,
                         const gchar *clef,
                         gpointer donnee);
```

où `clef` est une chaîne de caractères qui nous permettra d'identifier et de retrouver la donnée que l'on passe dans le paramètre `donnee`. Comme `donnee` est de type `gpointer`, ce qui permet de passer des données de n'importe quel type.

Cette donnée pourra être récupérée par la suite en appelant la fonction suivante :

```
gpointer gtk_object_get_data(GtkObject *objet,
                             const gchar *clef);
```

La donnée reste attachée à l'objet tant que celui-ci existe ou jusqu'à ce que l'on appelle la fonction suivante :

```
void gtk_object_remove_data(GtkObject *objet,
                            const gchar *clef);
```

Attention ! Il arrive souvent que l'on attache une structure entière à un objet en procédant de cette manière :

```
ptr = g_malloc(sizeof(StructureBidule));
gtk_object_set_data(objet, "Bidule", ptr);
```

et on espère que la structure pointée par `ptr` sera libérée lorsque l'on détachera la donnée ou que l'objet sera détruit. En fait, il n'en est rien. En effet, puisque l'on peut utiliser n'importe quel type, rien n'indique au GTK si la donnée peut être détruite, ni comment. On doit donc libérer l'espace mémoire "à la main". Le plus simple pour cela, est d'attacher la donnée non pas avec `gtk_object_set_data()` mais avec la fonction suivante :

```
void gtk_object_set_data_full(GtkObject *objet,
                              const gchar *clef,
                              gpointer donnee,
                              GtkDestroyNotify destroy);
```

où le paramètre `destroy` est une fonction qui sera appelée automatiquement lors de la destruction de l'objet ou de l'appel à `gtk_object_remove_data()`. Cette fonction recoit comme premier et unique paramètre le pointeur `donnee`. Ainsi, pour attacher une donnée qui sera automatiquement détruite, on utilisera un code semblable à celui-ci :

```
void libere_memoire(gpointer donnee)
{
  g_free(donnee);
}

...

  ptr = g_malloc(sizeof(StructureBidule));
  gtk_object_set_data_full(objet, "Bidule",
                           ptr,
                           libere_memoire);
  ...
```

Notez que dans ce cas-là, on aurait tout aussi bien pu utiliser la fonction `g_free()` à la place de `libere_memoire()` dans l'appel de `gtk_object_set_data_full()`.

Si l'on désire détacher une donnée sans que la fonction de destruction soit appelée, il faut utiliser :

```
void gtk_object_remove_no_notify(GtkObject *object,
                                 const gchar *clef);
```

Il peut être pesant d'avoir à se rappeler quel est la clef que l'on a utiliser pour stoquer une donnée dans un objet. C'est pourquoi le GTK propose les deux fonctions :

```
void gtk_object_set_user_data(GtkObject *objet,
                              gpointer donnee);
gpointer gtk_object_get_user_data(GtkObject *objet);
```

qui sont rigoureusement équivalentes aux fonctions `gtk_object_set_data()` et `gtk_object_get_data()` utilisées avec la clef `"user_data"`.

2.2. Les signaux de GTK

Les `GtkObjects` et donc tous les objets du GTK sont capables, dans certaines conditions d'émettre des signaux. Par exemple lorsque l'utilisateur clique sur un `GtkButton`, il emet le signal `"clicked"`. Les signaux sont en fait des messages que les objets envoient au GTK qui décide de ce qu'il doit en faire. Le GTK possède déjà des méthodes pour traiter la plupart des signaux que lui envoie les différents types d'objet. Mais on peut aussi décider que l'on veut être au courant lorsque tel ou tel objet envoie tel ou tel signal. Pour cela, il faut connecter l'objet et le signal à une fonction qui sera appelée automatiquement dès que l'objet enverra le signal en question. Une telle fonction s'appelle "fonction de rappel" ou *callback* en anglais. Pour connecter une fonction de rappel à un object et un signal, on utilise la fonction suivante :

```
guint gtk_signal_connect(GtkObject *objet,
                         const gchar *nom_du_signal,
```

```
                        GtkSignalFunc callback,
                        gpointer donnee);
```

où `objet` est un objet du GTK, c'est-à-dire une variable d'un type dérivé de `GtkObject`. `nom_du_signal` est le nom du signal sous la forme d'une chaîne de caractères, comme par exemple `"clicked"` ou `"destroy"`. `callback` est la fonction de rappel qui sera appelée automatiquement à chaque fois que l'objet emettra le signal. La valeur retournée par `gtk_signal_connect()` est un nombre qui identifie de manière unique la connection effectuée entre l'objet, la fonction et le signal. Cet identifiant nous permettra plus tard de déconnecter le signal et l'objet de la fonction de rappel.

Le prototype de cette fonction dépend du signal. Le premier paramètre transmis à la fonction de rappel est `objet`, le dernier paramètre est `donnee`. Et entre les deux, des paramètres spécifiques au signal peuvent également être transmis. Beaucoup de signaux n'ont pas de paramètres spécifiques et le prototypes des fonctions de rappels se résume souvent à :

```
gboolean callback(GtkObject *objet, gpointer data);
```

Il est tout à fait possible de connecter un signal et un objet à plusieurs fonction en appelant plusieurs fois `gtk_signal_connect()` avec les mêmes deux premiers paramètres. Cependant rien ne garantit l'ordre dans lequels les différentes fonctions de rappels seront appelées lors de l'emission d'un signal. Aussi, si une fonction doit absoluement être appelée *après* une autre, il vaut mieux connecter le signal à l'aide de la fonction suivante :

```
guint gtk_signal_connect_after(GtkObject *objet,
                               const gchar *nom_du_signal,
                               GtkSignalFunc callback,
                               gpointer donnee);
```

Ceci garantit simplement que les fonctions connectées au signal avec `gtk_signal_connect_after()` seront appelées après celles connectées avec `gtk_signal_connect()`. Les fonctions de rappels que le GTK installe par défaut peuvent être appelées avant ou après celles que l'on rajoute.

Souvent, il est pratique d'utiliser directement une fonction du GTK comme fonction de rappel. Par exemple la fonction

```
gtk_widget_destroy(GtkWidget *widget);
```

sert à détruire un widget dont on a plus besoin, et l'on veut souvent détruire un widget lorsque l'utilisateur clique sur un bouton. Si l'on connecte le signal `"clicked"` d'un objet de type `GtkButton` à la fonction `gtk_widget_destroy()` de cette façon :

```
gtk_signal_connect(GTK_OBJECT(bouton), "clicked",
                   (GtkSignalFunc)gtk_widget_destroy,
                   donnee);
```

alors la fonction `gtk_widget_destroy()` sera appelée avec `bouton` comme unique paramètre, le paramètre `donnee` étant tout simplement ignoré. C'est déjà intéressant, mais un peu limité. On souhaite en général détruire un autre widget que celui sur lequel on clique. Pour cela, il vaut mieux connecter le signal et la fonction de rappel à l'aide de :

```
guint gtk_signal_connect_object(GtkObject *objet,
                                const gchar *nom_du_signal,
                                GtkSignalFunc callback,
                                GtkObject *objet_cible);
```

Le rôle de cette fonction est exactement le même que celui de la fonction `gtk_signal_connect()`, mais lors de l'émission du signal, le premier paramètre et le dernier seront inversés. C'est-à-dire que le premier paramètre passé à la fonction de rappel sera `objet_cible` et le dernier sera `objet`. Ainsi, si l'on veut que la boîte de dialogue `dialogue` soit détruite lorsque l'on clique sur le bouton `bouton`, on connectera le signal `"clicked"` de cette façon :

```
gtk_signal_connect_object(GTK_OBJECT(bouton),
                          "clicked",
                          (GtkSignalFunc)gtk_widget_destroy,
                          GTK_OBJECT(dialogue));
```

La fonction :

```
guint gtk_signal_connect_object_after(GtkObject *objet,
                                      const gchar *nom_du_signal,
                                      GtkSignalFunc callback,
                                      GtkObject *objet_cible);
```

cumule les effets de `gtk_signal_connect_object()` et de `gtk_signal_connect_after()`.

Lorsque l'on désire déconnecter une fonction de rappel d'un signal et d'un object, il suffit d'appeler la fonction suvante :

```
void gtk_signal_disconnect(GtkObject *object,
                           guint id);
```

où `id` est l'identifiant de connection retourné par les fonctions `gtk_signal_connect*()`.

2.3. La boucle principale du GTK

Vous le savez déjà, dans un programme GTK+ classique, la fonction `main()` débute par un `gtk_init(&argv, &argv);` et se termine par un appel à `gtk_main()`. C'est en fait dans cette dernière fonction que tout se passe. En résumé, cette fonction installe une boucle qui ne se termine que lorsque l'on appelle la fonction

```
void gtk_main_quit(void);
```

et dans cette boucle, le GTK surveille si des signaux sont en attente et les redistribue aux fonctions de rappels correspondantes. Si, dans vos programmes, vous effectuez une tâche assez longue, le GTK mettra en attente tous les signaux provenants des `GtkObjects` et autres, en particulier, les signaux que les widgets émettent lorsqu'ils doivent être redessinés ne seront pas traités immédiatement. Aussi si l'on veut synchroniser le traitement des signaux du GTK avec une fonction assez longue, il faut régulièrement utiliser le code suivant :

```
while(gtk_events_pending())
  gtk_main_iteration();
```

En effet, la fonction `gtk_events_pending()` vérifie s'il y a des signaux en attente de traitement, et `gtk_main_iteration()` récupère le premier signal en attente et le redistribue. En fait la fonction `gtk_main()` réalise à peu près cela.

C'est un fait un peut plus compliqué que cela. En effet, le GTK offre plusieurs possibilités d'appels automatique de fonctions.

La première est bien évidemment liée au signaux en provenance des `GtkObjets` et des événements.

La deuxième est liée au temps. En effet, il est possible de demander au GTK d'appeler régulièrement l'une de nos fonctions. Bien entendu, il ne peut faire cela que si il est dans la fonction `gtk_main()`. Pour ajouter une fonction parmis celles qui sont appelée régulièrement, il suffit d'appeler la fonction suivante :

```
guint gtk_timeout_add(guint32 interval,
                      GtkFunction fonction,
                      gpointer donnee);
```

où `internal` est l'intervalle de temps, exprimé en millisecondes entre deux appels à la `fonction` passée comme deuxième paramètre. Cette fonction, qui sera donc appelée automatiquement et régulièrement, devra avoir le prototype suivant :

```
gboolean fonction(gpointer donnee);
```

Le paramètre `donnee` sera celui qui a été passé comme troisième paramètre à `gtk_timeout_add()`. La valeur de retour de cette fonction doit être `TRUE` s'il l'on désire que la fonction soit rappeler régulièrement, et `FALSE` si l'on préfère que la fonction ne soit appelée qu'une seule fois. La valeur de retour de `gtk_timeout_add()` est un identifiant qui permet de supprimer l'appel régulier à notre fonction grâce à :

```
void gtk_timeout_remove(guint id);
```

La troisième est lié à l'état *idle*, c'st-à-dire oisif. GTK est dans cet état lorsqu'il n'a rien d'autre à faire. De la même façon qu'avec les timers, il est possible de demander au GTK d'appeler une fonction à chaque fois qu'il est oisif à l'aide de :

```
guint gtk_idle_add(GtkFunction fonction,
                   gpointer donnee);
```

La valeur de retour est là aussi un identifiant permettant de supprimer l'appel automatique à notre fonction en la passant en paramètre à la fonction suivante :

```
void gtk_idle_remove(guint id);
```

Soyez cependant conscients que la fonction appelée automatiquement en utilisant `gtk_idle_add()` sera appelée vraiment très fréquemment si le GTK+ n'a rien d'autre à faire, ce qui arrive souvent. Il faut donc veiller à ce que cette fonction n'utilise pas trop de temps CPU, sous peine de bloquer la machine. Il vaut donc mieux utiliser cette possibilité avec parcimonie. L'utilisation des timers est bien souvent plus adapté que celle de l'état *idle*.

7

Les GtkWidgets

1. Présentation

Les widgets – contraction de *WInDows' GadgET* – sont des objets graphiques qui sont destinés à être affichés à l'écran et qui peuvent avoir une interaction avec l'utilisateur. Cette définition est vague et en effet le rôle des widgets est très entendu, cela va d'un simple bout de texte qui est affiché à l'écran, au notebook, en passant par les boutons et les menus.

2. Les macros des GtkWidgets

Pour GTK, les widgets sont des variables de type `GtkWidget`. Ce type est bien entendu un descendant du type `GtkObject` dont il hérite les principales caractéristiques. Il est de plus l'ancêtre commun à tous les autres types de widgets. Tout comme `GtkObject`, il s'agit d'un type abstrait, qui n'est pas sensé servir directement, mais sert de base à tous les types de widgets utilisés dans le GTK.

Bien entendu, comme tous les types dérivés des `GtkObjects`, les `GtkWidgets` proposent des macros permettant de tester si un pointeur pointe bien sur un object de type `GtkWidget` et de transtyper un tel pointeur. Ces macros sont :

```
GTK_WIDGET(widget)
GTK_WIDGET_CLASS(classe)
GTK_IS_WIDGET(widget)
GTK_IS_WIDGET_CLASS(classe)
```

Mais les `GtkWidgets` sont fournies avec bien d'autres macros très pratiques. Ces macros permettent de connaître l'état actuel d'un widget particulier. Elles s'appliquent évidemment à n'importe quel objet dont le type dérive de `GtkWidget`. Toutes ces macros prennent comme unique paramètre un pointeur sur une structure `GtkWidget` et renvoient soit `TRUE` soit `FALSE`. Les plus importantes[1] de ces macros sont :

- `GTK_WIDGET_TOPLEVEL(widget)` permet de savoir si un widget est une fenêtre gérée par le window manager.
- `GTK_WIDGET_NO_WINDOW(widget)` est vrai si le widget ne possède pas de fenêtre propre. C'est notamment le cas des `GtkLabels` qui utilisent la fenêtre parente pour se dessiner. Les widgets ne possédant pas de fenêtre propre ne peuvent pas non plus recevoir de signaux.
- `GTK_WIDGET_REALIZED(widget)` est vrai si le widget est actuellement entièrement géré par GTK+.
- `GTK_WIDGET_MAPPED(widget)` est vrai si le widget a été affiché par le serveur X et possède donc une fenêtre.
- `GTK_WIDGET_VISIBLE(widget)` est vrai si le widget est actuellement visible.
- `GTK_WIDGET_DRAWABLE(widget)` est une combinaison des deux précédents. Cette macro retourne `TRUE` si l'on peut dessiner dedans.

[1] Les plus curieux d'entre vous pourront trouver les autres dans le fichier `gtkwidget.h`

- `GTK_WIDGET_CAN_FOCUS(widget)` est vrai si le widget peut recevoir le focus du clavier et donc recevoir des événements en provenance du clavier. En effet, certains widgets ne servent que de décorations, comme les label, ou les séparateurs, et il n'ont donc pas besoin de recevoir des événements.
- `GTK_WIDGET_HAS_FOCUS(widget)` est vrai si le widget a actuellement le focus du clavier.
- `GTK_WIDGET_HAS_DEFAULT(widget)` est vrai si le widget peut être activé en pressant la touche 'entrée', même s'il n'a pas le focus du clavier. Cela est utile pour choisir un bouton par défaut dans une boîte de dialogue, comme le bouton OK par exemple.
- `GTK_WIDGET_CAN_DEFAULT(widget)` est vrai si le widget peut obtenir l'état `GTK_WIDGET_HAS_DEFAULT`, par exemple par l'utilisation de la touche 'Tab' dans une boîte de dialogue.
- `GTK_WIDGET_HAS_GRAB(widget)` est vrai si le widget a actuellement fait main basse (to grab) sur le serveur X. C'est-à-dire que le widget récupère actuellement tous les événements souris ou claviers même si le pointeur de la souris n'est pas au dessus de lui.
- `GTK_WIDGET_SENSITIVE(widget)` est vrai si le widget peut recevoir des événements et n'est pas grisé.

  La plupart de ces états peuvent être changé grâce aux macros :

```
GTK_WIDGET_SET_FLAGS(Widget, NomDuFlag);
GTK_WIDGET_UNSET_FLAGS(Widget, NomDuFlag);
```

où `NomDuFlag` est l'état que l'on veut positionner. Voici la table de correspondance entre l'état et la macro permettant de le lire :

état	macro
GTK_TOPLEVEL	GTK_WIDGET_TOPLEVEL()
GTK_NO_WINDOW	GTK_WIDGET_NO_WINDOW()
GTK_REALIZED	GTK_WIDGET_REALIZED()
GTK_MAPPED	GTK_WIDGET_MAPPED()
GTK_VISIBLE	GTK_WIDGET_VISIBLE()
GTK_CAN_FOCUS	GTK_WIDGET_CAN_FOCUS()
GTK_HAS_FOCUS	GTK_WIDGET_HAS_FOCUS()
GTK_HAS_DEFAULT	GTK_WIDGET_HAS_DEFAULT()
GTK_CAN_DEFAULT	GTK_WIDGET_CAN_DEFAULT()
GTK_HAS_GRAB	GTK_WIDGET_HAS_GRAB()
GTK_SENSITIVE	GTK_WIDGET_SENSITIVE()

3. Les champs importants des GtkWidgets

Il est inutile de connaître tous les champs de chaque structure d'objet du GTK. En général, seule une faible partie est intéressante. Voici la structure `GtkWidget` dans laquelle je n'ai laissé que les champs les plus utiles :

```
struct _GtkWidget
{
  GtkObject object;

  ...
  gchar *name;
  ...
  GtkAllocation allocation;
  ...
  GdkWindow *window;
  ...
  GtkWidget *parent;
  ...
};
```

Le premier champ est évidemment un `GtkObject` afin de pouvoir facilement transtyper un `GtkWidget *` en `GtkObject *`. Cela fait partie du mécanisme d'héritage.

Le champ `name` est le nom du widget. Il sert au GTK pour déterminer quel est le style à utiliser pour dessiner ce widget. De plus, il est extrèment pratique pour déboguer une interface récalcitrante. Par défaut, ce champ à la valeur `NULL`, mais si on demande le nom du widget en utilisant la fonction :

```
gchar *gtk_widget_get_name(GtkWidget *widget);
```

alors le nom retourné sera par défaut le nom de la classe du widget, c'est-à-dire `"GtkWidget"` pour les widgets, `"GtkLabel"` pour les label, etc. Ce nom peut être changé grâce à la fonction suivante :

```
void gtk_widget_set_name(GtkWidget *widget,
                         const gchar *name);
```

Le champ `allocation` permet de connaître la taille et la position du widget. Curieusement, il n'existe pas de fonction du GTK permettant de savoir quelle est la taille actuelle d'un widget et où il est placé par rapport au widget qui le contient. Ainsi, si `widget` est un pointeur sur une structure `GtkWidget`, on peut connaître la position du widget avec :

```
x = widget->allocation.x;
y = widget->allocation.y;
```

et sa taille avec :

```
largeur = widget->allocation.width;
hauteur = widget->allocation.height;
```

Cette information n'est disponible que lorsque le widget a été réalisé. C'est-à-dire, si la fonction `gtk_widget_realize()` a été appelée.

Le champ `window` contient un pointeur sur la fenêtre qu'utilise le widget pour se représenter et recevoir les événements. Cette fenêtre n'existe que si le widget a

été réalisé. Si le widget est de type `NO_WINDOW`, alors c'est la fenêtre du widget qui contient notre widget qui est utilisée ici.

Si le widget est à l'intérieur d'un autre, alors celui-ci est dit *parent* du premier et il est présent dans le champ `parent`. Cela arrive très souvent qu'un widget en contienne un ou plusieurs autres. On les appelle des *containers*.

4. Les signaux relatifs aux GtkWidgets

Comme les `GtkWidgets` possèdent en général une fenêtre, ils sont succeptibles de recevoir tous les événements que le GDK génére et envoie à cette fenêtre. Tous les événements reçus sont renvoyés sous forme de signaux par les `GtkWidgets` au GTK qui les traitera comme n'importe quel autre signal. Tous ces signaux utilise un paramètre supplémentaire qui est un pointeur sur une structure de type `GdkEvent*`. Les fonctions de rappels liés à ces signaux ont donc trois paramètres : le premier et le dernier sont un pointeur sur le widget lui-même et la donnée supplémentaire passée lors de l'appel à l'une des fonctions `gtk_signal_connect*()`. Le deuxième paramètre, passé entre les deux autres est un pointeur sur la structure `GdkEvent*`.

Le tableau de la page suivante donne la liste de ces signaux avec la structure d'événement correspondante.

Le signal `"event"` est émis pour tout les événements GDK, puis le signal correspondant réellement à l'événement est émis. Cela permet de créer une procédure de traitement des événements en ne connectant que le signal `"event"` et en effectuant des tests sur le champ `type` de la structure d'événement transmise. Mais le plus simple est généralement de connecter uniquement les signaux dont on a besoin à des fonctions différentes. Par exemple, pour gérér le signal `"expose_event"` on pourra utiliser un code semblable au suivant :

```
gboolean ExposeCallBack(GtkWidget *wid,
                        GdkExposeEvent *ev,
                        gpointer donnee)
{
  ...
  traitement du signal
  ...
}

  ...

  gtk_signal_connect(GTK_OBJECT(widget),
                     "expose_event",
                     (GtkSignalFunc)ExposeCallBack,
                     donnee);
  ...
```

signal	structure d'événement
`"event"`	GdkEvent
`"button_press_event"`	GdkEventButton
`"button_release_event"`	GdkEventButton
`"motion_notify_event"`	GdkEventMotion
`"delete_event"`	GdkEventAny
`"destroy_event"`	GdkEventAny
`"expose_event"`	GdkEventExpose
`"key_press_event"`	GdkEventKey
`"key_release_event"`	GdkEventKey
`"enter_notify_event"`	GdkEventCrossing
`"leave_notify_event"`	GdkEventCrossing
`"configure_event"`	GdkEventConfigure
`"focus_in_event"`	GdkEventFocus
`"focus_out_event"`	GdkEventFocus
`"map_event"`	GdkEventAny
`"unmap_event"`	GdkEventAny
`"property_notify_event"`	GdkEventProperty
`"selection_clear_event"`	GdkEventSelection
`"selection_request_event"`	GdkEventSelection
`"selection_notify_event"`	GdkEventSelection
`"proximity_in_event"`	GdkEventProximity
`"proximity_out_event"`	GdkEventProximity
`"visibility_notify_event"`	GdkEventVisibility
`"client_event"`	GdkEventClient
`"no_expose_event"`	GdkEventAny

Les `GtkWidgets` introduisent également d'autres signaux, qui sont plus ou moins hérités des événements du GDK. Les premiers de ces signaux sont `"selection_get"` et `"selection_received"` et sont liés au fonctionnement du presse papier de X-Window. Les autres, à savoir `"drag_begin"`, `"drag_end"`, `"drag_data_get"`, `"drag_data_delete"`, `"drag_leave"`, `"drag_motion"`, `"drag_drop"` et `"drag_data_received"`, concernent le glisser-déplacer, ou *drag n drop* en anglais. Nous n'étudierons pas ces signaux.

Comme pour les `GdkWindows`, il est possible de choisir quels événements un `CodeWidget` particulier doit recevoir avec la fonction suivante :

```
void gtk_widget_set_events(GtkWidget *widget,
                           gint ev);
```

où ev est un champ de bits identique à celui utilisé par la fonction `gdk_window_set_events()`. Il est également possible d'ajouter des événements à ceux que le widget reçoit déjà avec :

```
void gtk_widget_add_events(GtkWidget *widget,
```

```
                                gint ev);
```

L'ensemble des événements qu'un widget accepte actuellement de recevoir peut être obtenu avec :

```
gint gtk_widget_get_events(GtkWidget *widget);
```

Et comme pour avec les `GdkWindows`, on peut récupérer la position de la souris à l'intériieur d'un widget gràce à :

```
void gtk_widget_get_pointer(GtkWidget *widget,
                            gint *x,
                            gint *y);
```

Comme pour le cas des `GdkWindows`, cette fonction peut être utile si le masque d'événement `GDK_POINTER_MOTION_HINT_MASK`.

5. Création et destruction des GtkWidgets

Les `GtkWidgets` étant des objets abstrait, au même titre que les `GtkObjects`, il ne sont en général pas créés directement. Cependant cela reste possible grâce au même mécanisme que pour la création des `GtkObjects`. Le numéro associé au type `GtkWidget` peut être obtenu avec :

```
GtkType gtk_widget_get_type(void);
```

Rappelons que cet identifiant de type est créé dynamiquement lors du premier appel à cette fonction.

Les `GtkWidgets` peuvent être créer à l'aide de l'un des deux fonctions suivantes :

```
GtkWidget* gtk_widget_new(GtkType type,
                          const gchar *nom_arg,
                          ...);
GtkWidget* gtk_widget_newv(GtkType type,
                           guint nb_args,
                           GtkArg *args);
```

qui ont globalement le même rôle que les fonctions équivalentes des `GtkObjects`. Chacun des arguments des `GtkWidgets` pourra par la suite être consulté ou changé par le biais des fonctions suivantes qui ne sont rien d'autre que des appels aux fonctions `gtk_object*` correspondantes. Elles n'existent en fait que parcequ'il est plus fréquent d'utiliser des widgets que des objects.

```
void gtk_widget_get(GtkWidget *widget,
                    GtkArg *arg);
void gtk_widget_getv(GtkWidget *widget,
                     guint nb_args,
                     GtkArg *args);
void gtk_widget_set(GtkWidget *widget,
                    const gchar *nom_arg,
```

nom de l'argument	type de l'argument
"GtkWidget : :name"	GtkString
"GtkWidget : :parent"	GtkContainer
"GtkWidget : :x"	gint
"GtkWidget : :y"	gint
"GtkWidget : :width"	gint
"GtkWidget : :height"	gint
"GtkWidget : :visible"	gboolean
"GtkWidget : :sensitive"	gboolean
"GtkWidget : :app_paintable"	gboolean
"GtkWidget : :can_focus"	gboolean
"GtkWidget : :has_focus"	gboolean
"GtkWidget : :can_default"	gboolean
"GtkWidget : :has_default"	gboolean
"GtkWidget : :receives_default"	gboolean
"GtkWidget : :composite_child"	gboolean
"GtkWidget : :style"	GtkStyle
"GtkWidget : :events"	GdkEventMask
"GtkWidget : :extension_events"	GdkEventMask

```
                    ...);
void gtk_widget_setv(GtkWidget *widget,
                     guint nb_args,
                     GtkArg *args);
```

Les arguments que les `GtkWidgets` introduisent sont regroupés dans le tableau suivant :

La plupart des arguments sont en fait des états du widgets que l'on peut changer en utilisant des fonctions appropriées.

L'argument `GtkWidget::parent` permet de définir quel est le widget qui va contenir notre widget. Un widget en contenant un ou plusieurs autres est forcément d'un type descendant de `GtkContainer`. Si, par la suite, on ne désire plus que notre widget soit à l'intérieur de son parent, on peut le rendre orphelin avec la fonction suivante :

```
void gtk_widget_unparent(GtkWidget *widget);
```

cela ne détruit pas le widget, ni son parent, simplement ils n'ont plus aucun lien entre eux.

On peut ensuite rattacher notre widget à un autre parent avec :

```
void gtk_widget_set_parent(GtkWidget *widget,
                           GtkWidget *parent);
```

Faites simplement attention à ce qu'un widget ne peut avoir qu'un seul parent à la fois.

Il est possible de réaliser les deux opérations précédentes en une seule fois, c'est-à-dire de changer le widget parent d'un autre à l'aide de :

```
void gtk_widget_reparent(GtkWidget *widget,
                         GtkWidget *nouveau_parent);
```

Il est aussi possible de ne choisir que la fenêtre parente d'un widget. Cela permet notamment à un widget de type `NO_WINDOW` de choisir la fenêtre qui recevra les événements GDK à sa place. Pour ce faire, il suffit d'appeler la fonction suivante :

```
void gtk_widget_set_parent_window(GtkWidget *widget,
                                  GdkWindow *parente);
```

De la même façon, il est possible de connaître la fenêtre parente d'un widget donné à l'aide de la fonction suivante :

```
GdkWindow *gtk_widget_get_parent_window(GtkWidget *widget);
```

Il n'existe pas de fonction permettant de connaître quelle est la fenêtre associée à un widget car cette information est disponible directement en faisant :

```
fenetre = (GTK_WIDGET(widget))->window;
```

Vous avez sans doute compris que les widgets se trouvent fréquemment à l'intérieur d'autres. On peut en effet les emboîtés les uns dans les autres sans réelle limite. Il arrive cependant que l'on ait besoin de savoir quel est le widget de type toplevel qui contient un widget donné. Cela peut être son parent direct mais aussi son grand père, son arrière grand père, etc. Pour obtenir directement cette information, on peut utiliser la fonction suivante :

```
GtkWidget *gtk_widget_get_toplevel(GtkWidget *widget);
```

De la même façon, on peut vérifier si un widget en contient un autre au sens large, c'est-à-dire s'il est un ancêtre plus ou moins proche avec :

```
gboolean gtk_widget_is_ancestor(GtkWidget *widget,
                                GtkWidget *ancetre);
```

Cette fonction renvoie `TRUE` si `widget` fait partie de la descendance de `ancetre`.

Cependant, on ne manipule généralement pas les parents des widgets directement car les `GtkContainers` et leurs descendant proposent des fonctions beaucoup plus pratiques et puissantes.

Un autre état qu'il est important de pouvoir changer est la sensibilité des widgets. Quand un widget est insensible, il est en général grisé et ne reçoit plus les événements du clavier et de la souris. Il est donc momentanément désactivé. Pour changer cet état, il suffit d'utiliser la fonction suivante :

```
void gtk_widget_set_sensitive(GtkWidget *widget,
                              gboolean sensitif);
```

Si `sensitif` vaut `FALSE`, alors le widget est desactivé, sinon, il est réactivé.

On peut aussi demander au GTK de placer un widget en un endroit particulier de son parent. Cette position est souvent ignorée car les `GtkContainers` préfèrent souvent positionner les widgets qu'ils contiennent eux mêmes. Donc, lorsque cela est possible, on peut choisir la position d'un widget en appelant la fonction suivante :

```
void gtk_widget_set_uposition(GtkWidget *widget,
                              gint x, gint y);
```

De la même façon, on peut choisir la taille minimum qu'un widget doit avoir avec la fonction suivante :

```
void gtk_widget_set_usize(GtkWidget *widget,
                          gint width,
                          gint height);
```

Il s'agit encore d'une indication de ce que souhaite le widget. Le widget le contenant peut tout à fait décider d'allouer plus de place que celle demander, par exemple parce que l'utilisteur a agrandi la fenêtre. En général, le widget dispose au moins de la taille qu'il demande.

Pour dessiner dans la fenêtre d'un widget, nous avons besoin de savoir quelle colormap et quel visual elle utilise, ces deux informations sont données par les deux fonctions suivantes :

```
GdkColormap *gtk_widget_get_colormap(GtkWidget *widget);
GdkVisual *gtk_widget_get_visual(GtkWidget *widget);
```

Il vaut mieux ne pas changer le visual ou la colormap d'un widget après sa création car le GTK n'aime pas du tout ça. Mais on peut changer le visual et la colormap que les widgets utilisent lors de leur création avec les fonctions suivantes :

```
void gtk_widget_set_default_colormap(GdkColormap *colormap);
void gtk_widget_set_default_visual(GdkVisual *visual);
```

De la même façon, on peut savoir quel est le visual et la colormap actuellement utilisés pour la création des widgets avec les fonctions suivantes :

```
GdkColormap *gtk_widget_get_default_colormap(void);
GdkVisual *gtk_widget_get_default_visual(void);
```

Lorsque l'on a plus besoin d'un widget, on peut le détruire avec la fonction suivante :

```
void gtk_widget_destroy(GtkWidget *widget);
```

On utilise souvent un une variable globale de type `GtkWidget *` pour stoquer un widget. Cette variable est initialement à `NULL` et un simple test permet de savoir si le widget a déjà été créé ou non. Aussi, lors de sa destruction, il est pratique que cette variable redevienne automatiquement `NULL`. C'est justement le propos de la fonction suivante :

```
void gtk_widget_destroyed(GtkWidget *widget,
                          GtkWidget **pwidget);
```

Le seul rôle de cette fonction est de mettre la variable pointée par le paramètre `pwidget` à `NULL`, mais son prototype en fait une fonction parfaite à utiliser comme fonction de rappel associée au signal `"destroy"`. En effet, pour que le pointeur `MonWidget` revienne automatiquement à `NULL` lors de la destruction du widget pointé par `MonWidget`, il suffit de connecter le signal `"destroy"` ainsi :

```
GtkWidget *MonWidget;

gtk_signal_connect(GTK_OBJECT(MonWidget), "destroy",
                   (GtkSignalFunc)gtk_widget_destroyed,
                   &MonWidget);
```

Une autre façon de procéder est de simplement cacher un widget alors que l'on a demandé sa destruction. Cela permet de le réafficher rapidement au lieu de le recréer si besoin est. Pour cela il suffit de connecter le signal `"delete"` à la fonction suivante :

```
gint gtk_widget_hide_on_delete(GtkWidget *widget);
```

Il est temps de comprendre se qui se passe lors de la destruction d'un widget. En fait, le signal `"delete"` est tout d'abord émis. S'il existe une fonction de rappel associée à ce signal qui renvoie la valeur `TRUE` alors, tout s'arrête et le widget n'est pas détruit. Sinon, le signal `"destroy"` est emis à son tour et quelque soit les valeurs de retour des fonctions de rappels associée à ce signal, le widget sera détruit.

6. Les autres fonctions associées aux GtkWidgets

Lorsque l'on crée un widget avec la fonction `gtk_widget_new()`, la fenêtre `GdkWindow` associée n'est pas crée en même temps. Cela permet notamment au GTK de modifier des paramètres de cette fenêtre et de ne la créer que lorsqu'il connait tous ces paramètres afin de faire le moins d'appel au serveur X que possible. L'opération de création de la fenêtre associée à un widget se nomme la *réalisation du widget*. Pour forcer cette opération, il suffit d'appeler la fonction :

```
void gtk_widget_realize(GtkWidget *widget);
```

en donnant le widget comme paramètre. Cela crée la fenêtre, au sens X-Window du terme, sans l'afficher. On pourra par la suite détruire cette fenêtre (même si dans les faits on utilise que très rarement cette fonction) avec :

```
void gtk_widget_unrealize(GtkWidget *widget);
```

Notez que la destruction d'un widget entraîne automatiquement la destruction de la fenêtre associée.

Une fois que la fenêtre associée à un widget a été créée, on peut l'afficher grâce à la fonction :

```
void gtk_widget_map(GtkWidget *widget);
```

et la cacher avec

```
void gtk_widget_unmap(GtkWidget *widget);
```

Souvent, on crée la fenêtre au moment où on veut l'afficher. Alors, au lieu d'utiliser succéssivement `gtk_widget_realize()` et `gtk_widget_map()`, on utilise plutôt la fonction suivante :

```
void gtk_widget_show(GtkWidget *widget);
```

qui réalise le widget s'il ne l'était pas déjà et affiche le widget en appelant la fonction `gtk_widget_map()`[2].

Même en appelant la fonction `gtk_widget_show()`, le widget n'est pas affiché immédiatement, il faut que le GTK reprenne la main afin de pouvoir demander au serveur X d'afficher la fenêtre. Si l'on souhaite vraiment afficher un widget sans attendre la fin d'une fonction et le retour dans la fonction `gtk_main()`, on peut appeler la fonction :

```
void gtk_widget_show_now(GtkWidget *widget);
```

qui envoie la demande directement au serveur X.

Pour cacher un widget, plutôt de d'utiliser `gtk_widget_unmap()`, on préférera la fonction :

```
void gtk_widget_hide(GtkWidget *widget);
```

qui réalise à peu près la même chose, mais qui possède un nom un peu plus explicite[3]

Souvent, lorsque l'on construit une interface, on crée beaucoup de widgets inclus les uns dans les autres, puis on doit les afficher. Il peut être fastidieux d'avoir à appeler `gtk_widget_show()` pour chacun des widgets de l'interface. Aussi, il suffit d'appeler la fonction :

```
void gtk_widget_show_all(GtkWidget *widget);
```

Cette fonction montre le widget passé en paramètre, puis appelle récursivement `gtk_widget_show()` pour tous les widgets qu'il contient. Ainsi tous les widgets sont affichés d'un seul coup.

De manière symétrique, on peut cacher tous les widgets contenu dans un widget avec la fonction :

```
void gtk_widget_hide_all(GtkWidget *widget);
```

qui est tout de même moins utile puisqu'il suffit de cacher le widget contenant les autres pour que ceux-ci soient également cachés.

En règle général, le GTK sait parfaitement quand il doit redessiner les widgets dont il a la charge. En effet ceux-ci émettent le signal `"expose_event"` lorsqu'ils étaient cachés et qu'il devienne visible. Mais le déroulement d'un programme exige parfois que l'on redessine un widget pour d'autres raison, par exemple parce qu'un état a changé. Pour cela, le plus simple est d'appeler la fonction suivante :

---

[2] `gtk_widget_show()` émet aussi le signal `"show"` qui est très peu utilisé.

[3] En fait, `gtk_widget_hide()` émet également le signal `"hide"` qui n'est pas plus utilisé que le signal `"show"`.

```
void gtk_widget_queue_draw(GtkWidget *widget);
```

qui émet le signal `"expose_event"` avec les bons paramètres. Ce signal n'est pas un vrai `"expose_event"` en ce sens qu'il ne provient pas d'un événement GDK, on dit que ce signal a été *synthétisé* par le GTK. Les fonctions de rappel associées à ce signal (le GTK en fourni une par défaut pour pratiquement tous les widgets) se chargeront de redessiner le widget pour nous.

De la même façon, on peut demander de ne redessiner qu'une certaine zone rectangulaire d'un widget avec la fonction :

```
void gtk_widget_queue_draw_area(GtkWidget *widget,
                                gint x, gint y,
                                gint width,
                                gint height);
```

où `x` et `y` sont les coordonnées de l'origine de la zone à redessiner et `width` et `height` sont respectivement la largeur et la hauteur de la zone.

Dans de très rares cas, on peut vouloir effacer tout ou partie d'un widget, cela peut ce faire de la même façon à l'aide des fonctions suivantes :

```
void gtk_widget_queue_clear(GtkWidget *widget);
void gtk_widget_queue_clear_area(GtkWidget *widget,
                                 gint x, gint y,
                                 gint width,
                                 gint height);
```

Dans une boîte de dialogue, il y a souvent plusieurs widgets comme des boutons, des cases à cocher, ou des zones d'entrées de textes, mais un seul de ces widgets peut être manipulé à l'aide du clavier à un moment donné. On dit que ce widget possède le *focus* du clavier. Le focus change à l'intérieur d'une boîte de dialogue en appuyant sur la touche `Tab` du clavier. Pour changer le focus à l'intérieur d'un programme, on peut fixer quel est le widget qui doit recevoir le focus à l'aide de la fonction suivante :

```
void gtk_widget_grab_focus(GtkWidget *widget);
```

De plus, toujours dans une boîte de dialogue, il y a souvent des boutons qui peuvent être activés par un appuis sur la touche `Entrée` du clavier, même s'ils n'ont pas le focus. C'est souvent le cas des boutons `OK` ou `Annuler`. On dit de tels boutons qu'ils ont le *défaut*, ou l'*action par défaut*. On les repères facilement car ils sont dessinés différemment des autres. Du point de vue du programmeur, de tels widgets sont reconnaissables au fait qu'ils ont l'état `GTK_HAS_DEFAULT` positionné. On peut donner le défaut à un widget par le biais de la fonction :

```
void gtk_widget_grab_default(GtkWidget *widget);
```

Bien évidemment, seuls les widgets dont l'état `GTK_CAN_DEFAULT` est positionné peuvent être utilisés pour cette fonction.

Enfin, pour être un peu plus complet, n'oublions pas la fonction :

```
void gtk_widget_shape_combine_mask(GtkWidget *widget,
```

```
                                    GdkBitmap *shape_mask,
                                    gint offset_x,
                                    gint offset_y);
```

qui permet d'avoir des widgets dont la forme n'est pas rectangulaire. Cette fonction s'utilise exactement de la même façon que la fonction du GDK `gdk_window_shape_combine_mask()` qui, elle, s'applique plutôt aux `GdkWindow`.

8

Les GtkMiscs

1. Les GtkMiscs

GtkObject
GtkWidget
GtkMisc

Les `GtkMiscs` sont encore un type abstrait[1] qui sert de classe de base pour d'autres classes de widget, comme les labels, les images, ou les flêches.

Les `GtkMiscs` sont des descendants directs des `GtkWidgets`. Ils héritent donc de tous leurs signaux, de leurs propriétés et de leurs fonctions.

Les GtkMiscs possèdent la faculté de pouvoir être alignés à droite ou à gauche de l'espace qui leur est alloué. Ils peuvent également déclarer qu'ils ont besoin de plus de place, afin de laisser un cadre libre autour d'eux. La structure `GtkMisc` est en effet définie ainsi :

```
typedef struct _GtkMisc GtkMisc;

struct _GtkMisc
{
  GtkWidget widget;

  gfloat xalign;
  gfloat yalign;

  guint16 xpad;
  guint16 ypad;
};
```

Le premier champ est évidemment du type `GtkWidget` afin de respecter le mécanisme d'héritage. Les champs `xpad` et `ypad` représentent le nombre de pixels supplémentaires que le `GtkMisc` désire avoir à droite et à gauche ou au dessus et au dessous de lui. Par exemple, si la taille naturelle d'un descendant d'un `GtkMisc`[2] est de 10 pixels par 10 pixels et que `xpad` vaut 4 et `ypad` vaut 3, alors, la taille réelle du widget sera de 18 pixels par 16 pixels. Les champs `xpad` et `ypad` doivent absolument être positifs, sinon le comportement de GTK+ peut devenir très aléatoire. Aussi, pour changer ces valeurs, il vaut en général mieux utiliser la fonction :

```
void gtk_misc_set_padding(GtkMisc *misc,
                          gint xpad, gint ypad);
```

qui effectue les vérifications nécessaires. La valeur par défaut de `xpad` et `ypad` est 0.

Les champs `xalign` et `yalign` permettent de choisir comment doit être justifié un widget de type `GtkMisc` si on lui alloue une place plus grande que celle dont il a besoin. Par exemple, si `xalign` vaut 0.5, alors le dessin représentnt le widget restera

---

[1] Rassurez-vous, c'est un des derniers !

[2] Les `GtkMiscs` étant des widgets abstraits, ils n'ont pas de "taille naturelle", en revanche, les `GtkLabel`, qui sont des descendant des `GtkMiscs` ont une taille naturelle qui est la taille du texte qu'ils affichent.

centré horizontalement, si la fenêtre associée au widget change de taille. Si `xalign` vaut 0.0, la représentation reste à gauche, et si `xalign` vaut 1.0, elle reste à droite. Le champ `yalign` fonctionne de la même manière mais concerne l'alignement vertical au lieu de l'alignement horizontal. Les champs `xalign` et `yalign` doivent toujours être compris entre 0.0 et 1.0 sinon, les effets les plus étranges peuvent survenir. Aussi est-il plus prudent de n'affecter ces champs que par le biais de la fonction suivante :

```
void gtk_misc_set_alignment(GtkMisc *misc,
                            gfloat xalign,
                            gfloat yalign);
```

La valeur par défaut des champs `xalign` et `yalign` est 0.5, c'est-à-dire que le dessin représentant le widget est normalement centré aussi bien horizontalement que verticalement dans la fenêtre associée au widget.

Comme tous les objets du GTK, les `GtkMiscs` proposent des macros permettant de faire un transtypage d'un pointeur en un pointeur sur un `GtkMisc` ou en un pointeur sur un `GtkMiscClass`. Ces macros sont les suivantes :

```
GTK_MISC(pointeur_objet)
GTK_MISC_CLASS(pointeur_classe)
GTK_IS_MISC(pointeur_objet)
GTK_IS_MISC_CLASS(pointeur_classe)
```

Comme tous les objets du GTK possèdent des macros semblables sont le nom se déduit facilement du type du widget, je ne le rappelerais plus pour les widgets suivants. Il en est de même pour la fonction :

```
GtkType gtk_misc_get_type(void);
```

qui renvoie le numéro associé au type `GtkMisc`. Tous les widgets possèdent une fonction identique.

Les `GtkMisc` ne sont pas destinés à être créés directement, on crée en général un de leur descendant. Il n'y a donc pas de fonction particulière permettant de créer facilement un `GtkMisc`. Mais on peut utiliser la fonction `gtk_widget_new()` pour cela. Les `GtkMiscs` introduisent d'ailleurs quatre nouveaux arguments au mécanisme de création générique de widgets.

Ces quatre arguments sont :

- `"GtkMisc::xalign"` ;
- `"GtkMisc::yalign"` ;
- `"GtkMisc::xpad"` ;
- `"GtkMisc::ypad"`

Ils correspondent exactement aux champs de la structure `GtkMisc`. Ces arguements peuvent naturellement être utilisés lors de la création de widgets descendant des `GtkMiscs`.

2. Les GtkLabels

GtkObject
  GtkWidget

GtkMisc
GtkLabel

Les `GtkLabels`, ou *labels* sont des morceaux de texte qui sont affichés à l'écran dans un but décoratif ou informatif. On les appelles quelques fois *étiquettes* mais le terme "label" est plus facile d'emploi puisqu'il ressemble plus au nom du type. Comme vous le voyez sur l'arbre d'héritage, les `GtkLabels` sont des descendant directs des `GtkMiscs`.

Les labels sont les premiers objets du GTK que nous rencontrons qui ne soit pas des objets abstraits. Ils possèdent donc une fonction permettant de les créer directement. Il s'agit de :

```
GtkWidget* gtk_label_new(const gchar *texte);
```

Cette fonction crée un nouveau widget de type `GtkLabel` avec le texte passé en paramètre. Vous remarquerez que le type renvoyé par cette fonction est `GtkWidget *` et pas `GtkLabel *`. Ceci est dû au fait que le GTK traite tous les widgets de la même façon. Cela permet d'appeler toutes les fonctions du type `gtk_widget*()` sans même avoir à faire de transtypage.

Voici un petit programme utilisant les labels :

```
/* Label.c */
#include <gtk/gtk.h>

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Label;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* Création de la fenêtre top-level */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  /* Création du label */
  Label = gtk_label_new("Bonjour à tous");
  /* Attachement du label dans la fenêtre */
  gtk_container_add(GTK_CONTAINER(Fenetre), Label);
  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
  return 0;
}
```

Ce programme doit être compilé en utilisant cette ligne de commande :

```
gcc Label.c -o Label `gtk-config --cflags --libs`
```

La partie entre `'...'` assure que les bons paramètres seront utilisés pour compiler un programme qui utilise les bibliothèques de GTK+.

Le résultat de ce programme est visible sur la figure 1.

figure 1

Si vous essayez de redimensionner la fenêtre principale de ce programme, vous verrez que le texte reste centré dans la fenêtre. Si vous avez lu le paragraphe concernant les `GtkMiscs`, vous devez comprendre pourquoi.

Les textes des labels peuvent également s'étendre sur plusieurs lignes. Essayez de changer, dans le programme précédant, la ligne de la création du label en :

```
Label = gtk_label_new("Ceci est\n"
                      "un label\n"
                      "réparti sur\n"
                      "plusieurs\n"
                      "lignes");
```

Et vous obtiendrez un label constitué de plusieurs lignes comme vous pouvez le voir sur la figure 2.

figure 2

Comme vous le voyez, par défaut, les lignes de texte qui contituent un label sont centrées les unes par rapport aux autres. Il est possible de changer ce comportement à l'aide de la fonction suiivante :

```
void gtk_label_set_justify(GtkLabel *label,
                           GtkJustification just);
```

où `just` est l'une des valeurs suivantes :

- `GTK_JUSTIFY_LEFT` qui permet de justifier le texte à gauche ;
- `GTK_JUSTIFY_RIGHT` qui permet de justifier le texte à droite ;
- `GTK_JUSTIFY_CENTER` qui permet de centrer les lignes les unes par rapport aux autres, c'est la valeur par défaut ;
- `GTK_JUSTIFY_FILL` qui permet de justifier à droite et à gauche.

La figure 3 représente un label multiligne justifié à droite.

figure 3

Par défaut le texte d'un label ne revient à la ligne que lorsque l'on utilise le caractère `'\n'`. Cela peut cependant vite devenir contraignant si le texte du label est suffisamment long. Aussi, il est possible de laisser le découpage des lignes à la charge du GTK. Le texte du label est alors découpé automatiquement de manière à occuper le moins de place possible. Pour cela il faut utiliser la fonction suivante :

```
void gtk_label_set_line_wrap(GtkLabel *label,
                             gboolean wrap);
```

Si le paramètre `wrap` vaut `TRUE`, alors, GTK essaiera de couper les lignes lui-même, sinon, le découpage en lignes devra être effectué à la main. Il est toujours possible d'insérer des saut de lignes supplémentaires en utilisant le caractère `'\n'`.

La figure 4 montre un label dont le texte est long et qui a été découpé automatiquement par GTK. Ce label a été créé à l'aide des lignes suivantes :

```
Label = gtk_label_new(
  "Ceci est un label très long et coupé automatiquement.\n"
  "Voici un autre paragraphe, remarquez que le retour à "
  "la ligne intervient au mileu de la ligne dans le "
  "paragraphe précédent.");
  gtk_label_set_line_wrap(GTK_LABEL(Label), TRUE);
```

Les labels sont aussi utilisés à l'intérieur d'autres widgets, comme par exemple dans les boutons. Une mode récente veut que, si le bouton peut être activé par l'appui sur une touche, alors la lettre correspondante du label doit être soulignée. Les `GtkLabel` permettent que le texte d'un label soit souligné à certains endroit. Pour cela il faut définir un motif qui est une chaîne de caractères de la même longueur

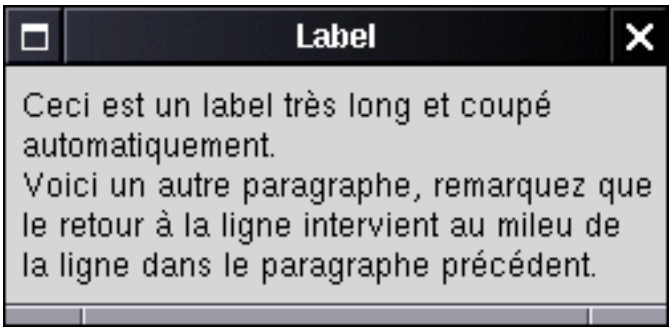


figure 4

que le texte du label. Chaque caractère de ce motif peut être soit `'_'` dans quel cas le caractère correspondant dans le texte du label sera souligné, soit `' '`. Ensuite on associe le motif de soulignement au label à l'aide de la fonction suivante :

```
void gtk_label_set_pattern(GtkLabel *label,
                           const gchar *motif);
```

La figure 5 représente un label dont la première lettre a été souligné à l'aide de cette méthode. Le label de la figure 5 a été créé avec le code suivant :

```
Label = gtk_label_new("Bonjour");
gtk_label_set_pattern(GTK_LABEL(Label), "_      ");
```

figure 5

Après sa création, il demeure possible de changer le texte d'un label à l'aide de la fonction :

```
void gtk_label_set_text(GtkLabel *label,
                        const gchar *texte);
```

De la même façon, il est possible de retrouver quel est le texte d'un label en utilisant :

```
void gtk_label_get(GtkLabel *label,
                   gchar **texte);
```

Le type `GtkLabel` introduit trois nouveaux arguments au mécanisme de création générique des widgets. Il est normalement plus facile d'utiliser les fonctions qui viennent d'être décrites. Ces trois arguments sont :

- `"GtkLabel::label"` qui est de type `gchar *` et qui définit le texte du label.
- `"GtkLabel::pattern"` qui est de type `gchar *` et qui définit le motif de soulignement du label.
- `"GtkLabel::justify"` qui est de type `enum` et qui définit comment le texte du label doit être justifié, il ne peut prendre que les valeurs que l'on a vue lors de l'étude de la fonction `gtk_label_set_justify()`.

3. Les GtkArrows

GtkObject
  GtkWidget
    GtkMisc
      GtkArrow

Les `GtkArrows` dérivent elles aussi directement des `GtkMiscs`. Elles servent à dessiner une flêche dans l'une des quatre directions. On ne les utilise généralement pas directement, mais ce sont elles que l'on retrouve au bout des barres de défilement ou *scrollbar*.

Les flèches sont crées avec la fonction suivante :

```
GtkWidget *gtk_arrow_new(GtkArrowType direction,
                         GtkShadowType ombrage);
```

où `direction` indique dans quelle direction pointe la flêche. Ce paramètre peut prendre l'une de ces quatre valeurs :

- `GTK_ARROW_UP` vers le haut ;
- `GTK_ARROW_DOWN` vers le bas ;
- `GTK_ARROW_LEFT` vers la gauche ;
- `GTK_ARROW_RIGHT` vers la droite ;

Le paramètre `ombrage` définit avec quel type d'effet d'ombrage la flêche doit être dessinée. En fait la flêche n'apparait que gràce à ces effets. Les valeurs possibles pour `ombrage` sont :

- `GTK_SHADOW_NONE` qui est très déconseillé, car la flêche sera dessinée sans aucun effet d'ombrage, c'est à dire que l'on ne la verra pas ;
- `GTK_SHADOW_IN` la flêche semble enfoncée dans l'écran ;
- `GTK_SHADOW_OUT` la flêche semble en relief ;
- `GTK_SHADOW_ETCHED_IN` le pourtour de la flêche semble enfoncé dans l'écran ;
- `GTK_SHADOW_ETCHED_OUT` le pourtour de la flêche semble être en relief.

La figure 6 montre toutes les flêches qu'il est possible de dessiner.

Les effets de reliefs sont des illusions d'optiques et sont donc parfaitement subjectifs. Essayez de retourner ce livre pour vous en convaincre !

Après sa création, il est possible de changer la direction et/ou le type d'ombrage d'une `GtkArrow` à l'aide de la fonction :

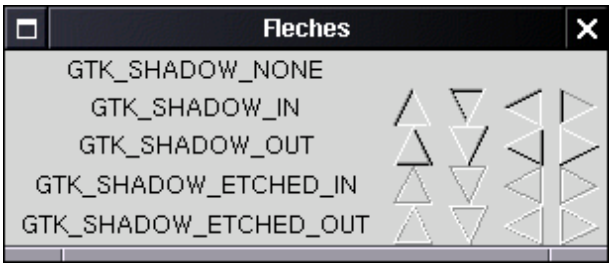


figure 6

```
void gtk_arrow_set(GtkArrow *arrow,
                   GtkArrowType direction,
                   GtkShadowType ombrage);
```

Il est rare de créer directement une `GtkArrow` car ce n'est pas très utile, les barres de défilement les créent toutes seules pour leur besoin. Toutefois, pour être exhaustifs, voici les deux arguments que les `GtkArrows` apportent au mécanisme de création générique des widgets :

- `"GtkArrow::arrow_type"` qui est de type `GtkArrowType` et qui définit la direction de la flêche ;
- `"GtkArrow::shadow_type"` qui est de type `GtkShadowType` et qui définit l'ombrage de la flêche.

4. Les GtkPixmaps

GtkObject
GtkWidget
GtkMisc
GtkPixmap

Les `GtkPixmaps` sont également des descendant des `GtkMiscs`. Ils sont le pendant dans le GTK des `GdkPixmap` du GDK. Leur rôle est d'afficher une image ou une icône. On les retrouve souvent à l'intérieur d'un bouton par exemple. Pour créer un `GtkPixmap`, on utilise la fonction suivante :

```
GtkWidget *gtk_pixmap_new(GdkPixmap *pixmapGDK,
                          GdkBitmap *masque);
```

où `pixmapGDK` est le GdkPixmap qui doit être dessiné sur le widget et `masque` est le masque correspondant qui définit quels pixels doivent être transparents.

Le plus simple pour créer à la fois l'image et le masque est d'utiliser les fonctions du GDK `gdk_pixmap_create_from_xpm()` et `gdk_pixmap_create_from_xpm_d()` permettant cela.

Voici un exemple de création de `GtkPixmap` :

```
/* Pixmap.c */
#include <gtk/gtk.h>
```

```
#include "icone.xpm"

int main(int argc, char *argv[])
{
  GdkPixmap *Pix;
  GdkBitmap *Masque;
  GdkColor   Transparent;
  GtkWidget *Fenetre;
  GtkWidget *Pixmap;

  /* Initialisation des bibliothèques de GTK+ */
  gtk_init(&argc, &argv);
  /* Création de la fenêtre */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  /* Réalisation de la fenêtre */
  gtk_widget_realize(Fenetre);
  /* Création du pixmap GDK */
  Pix = gdk_pixmap_create_from_xpm_d(Fenetre->window,
                                     &Masque,
                                     &Transparent,
                                     icone_xpm);
  /* Création du GtkPixmap */
  Pixmap = gtk_pixmap_new(Pix, Masque);
  /* On a plus besoin du pixmap GDK,
   * ni de son masque
   * donc on les déréférence */
  gdk_pixmap_unref(Pix);
  gdk_pixmap_unref(Masque);
  /* On place le GtkPixmap dans la fenêtre */
  gtk_container_add(GTK_CONTAINER(Fenetre), Pixmap);
  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
  return 0;
}
```

Cet exemple utilise le fichier `"icone.xpm"` que nous avons utilisé dans le chapitre 4. Il y a quelques remarques importante à faire à propos de ce programme. Tout d'abord vous vous souvenez peut-être que pour créer un pixmap, la fonction `gdk_pixmap_create_from_xpm_d()` a besoin d'une fenêtre `GdkWindow`. C'est pour cela que l'on doit réaliser le widget `Fenetre` afin de forcer GTK à créer la fenêtre `GdkWindow` correspondante et à la stoquer dans le champ `window` du widget `Fenetre`. On peut ensuite se servir de ce champ comme fenêtre de base pour la création du pixmap GDK.

Une fois le `GtkPixmap` crée, on peut effacer les deux `GdkPixmap` qui constituent l'image et le masque car le GTK en crée une copie[3] en interne. C'est même une bonne habitude à prendre car cela peut dans certains cas libérer un peu de mémoire.

Le résultat de ce programme est visible sur la figure 7.

figure 7

Après sa création il est possible de changer l'image et/ou le masque d'un `GtkPixmap` à l'aide de la fonction suivante :

```
void gtk_pixmap_set(GtkPixmap *pixmap,
                    GdkPixmap *image,
                    GdkBitmap *masque);
```

L'ancienne image et/ou l'ancien masque sont automatiquement déréférencés.

De la même façon, il est possible de connaître quels sont l'image et le masque actuellement utilisés à l'aide de la fonction suivante :

```
void gtk_pixmap_get(GtkPixmap *pixmap,
                    GdkPixmap **image,
                    GdkBitmap **mask);
```

Cette fonction est surtout utilisée pour copier un `GtkPixmap` dans un autre.

Par défaut, lorsqu'un widget de type `GtkPixmap` est rendu insensible avec un appel à la fonction `gtk_widget_set_sensitive()`, il est dessiné en grisé comme on peut le voir sur la figure 8. Si ce n'est pas ce que l'on souhaite pour un `GtkPixmap` donné, on peut appeler la fonction :

```
void gtk_pixmap_set_build_insensitive(GtkPixmap *pixmap,
                                      guint grise);
```

où `grise` est un indicateur. S'il vaut `TRUE`, alors le `GtkPixmap` sera grisé lorsqu'il est insensible, alors que si `grise` vaut `FALSE`, le `GtkPixmap` sera toujours dessiné de la même façon, qu'il soit sensible ou non.

---

[3] Ce n'est pas tout à fait vrai, en fait le GTK ne fait qu'accroître le nombre de références à ces pixmaps

figure 8

9

Les GtkContainers
Première partie

1. Les GtkContainers

GtkObject
GtkWidget
GtkContainer

Les `GtkContainers` sont une classe particulière de widget. Ils ont la faculté de pouvoir contenir un ou plusieurs widgets. Une des grandes forces du GTK est que justement la plupart des widgets qu'il propose sont en fait des containers. Les `GtkContainers` ne sont pas destinés à être créés directement. En revanche, ils implémentent des fonctions qui seront disponibles pour tous les widgets dérivés. Nous avons déjà rencontré des descendant des `GtkContainers` dans nos exemples. En effet les fenêtres toplevel que nous avons utilisés dans les exemples de programmes présentés jusqu'ici sont des containers. Et d'ailleurs, à chaque fois nous avons utilisé la fonction :

```
void gtk_container_add(GtkContainer *container,
                       GtkWidget *widget);
```

avec ces fenêtres. Vous l'aurez compris, le rôle de cette fonction est d'ajouter le `widget` dans le widget `container`. À l'inverse, on peut retirer un widget qui était dans un container avec la fonction suivante :

```
void gtk_container_remove(GtkContainer *container,
                          GtkWidget *widget);
```

Si vous avez essayé les programmes que j'ai présentés à propos des `GtkLabels`, vous avez du constater que le résultat obtenu ne correspondait pas exactement avec ce qui est présenté sur les figures. En fait, j'ai un peu triché et agrandi les fenêtres pour amélioré l'aspect des captures d'écran. En général, les containers adaptent leur taille pour contenir exactement les widgets que l'on a placés à l'intérieur d'eux[1] Et comme par défaut, les `GtkLabels` ont la taille de leur texte, on se retrouve avec les bords de la fenêtre qui "collent" au texte ce qui est vraiment inesthétique. Dans le cas des `GtkLabels`, on pourrait utiliser la fonction `gtk_misc_set_padding()`, mais dans le cas général, le plus simple est de rajouter un espace dans le container qui entourera tous ses enfants à l'aide de la fonction :

```
void gtk_container_set_border_width(GtkContainer *container,
                                    guint largeur_bord);
```

où `largeur_bord` est le nombre de pixels que l'on désire laisser entre le bord extérieur du container et son bord intérieur, là où résident ses enfants.

Les containers sont regroupés en deux types, suivant le nombre d'enfants qu'ils sont capables de contenir. Les containers qui ne contiennent qu'un seul enfants sont des `GtkBins`.

1.1. Les GtkBins

Les `GtkBins` définissent un type de widget très simple qui dérivent directement des `GtkContainers`. La structure `GtkBin` est définie ainsi :

[1] On dit que ces widgets sont les enfants du container.

```
struct _GtkBin
{
  GtkContainer container;

  GtkWidget *child;
};
```

Le champ `child` est un pointeur sur l'unique enfant que peut contenir le `GtkBin`. C'est-à-dire que pour obtenir ce pointeur, il suffit de faire quelque chose qui ressemble à :

```
enfant = (GTK_BIN(widget)->child);
```

C'est par exemple de cette façon que l'on accède au label qui est présent dans un bouton. Même s'il n'apportent pas de fonctions, les `GtkBins` sont très important dans GTK car beaucoup de widgets en dérivent. Et le fait qu'ils ne puissent contenir qu'un seul enfant n'est pas limitant en soit. Il suffit en effet que cet enfant soit un container capable de contenir à son tour plusieurs enfants pour palier cet inconvénient.

1.2. Les autres types de containers

Les autres types de containers sont tous capables de contenir plusieurs (au moins 2) widgets. C'est pourquoi le GTK propose d'autres fonctions pour manipuler l'ensemble de des enfants.

Tout d'abord, il est possible de récuperer l'ensemble des widgets enfants dans une liste doublement chaînée avec :

```
GList *gtk_container_children(GtkContainer *container);
```

Une fonction encore plus intéressante est la suivante :

```
void gtk_container_foreach(GtkContainer *container,
                           GtkCallback callback,
                           gpointer donnee);
```

Avec cette fonction, il est possible d'appeler une autre fonction avec chacun des widgets enfants du container comme premier paramètre et `donnee` comme second paramètres. Voyons cela dans un exemple. Imaginons que le widget `container` contienne les widget `widget1` et `widget2`. L'appel suivant :

```
gtk_container_foreach(GTK_CONTAINER(container),
                      MaFonction,
                      (gpointer)"donnee");
```

est équivalent aux deux appels suivants :

```
MaFonction(widget1, "donnee");
MaFonction(widget2, "donnee");
```

Il existe plusieurs avantages à utiliser `gtk_container_foreach()`. Le premier est évidemment la concision de l'appel, mais il faut aussi remarquer qu'il est assez rare que l'on garde des pointeurs sur tous les widgets enfants d'un container.

Les `GtkContainers` définissent plusieurs nouveaux signaux, parmis ceux-ci, seulement deux sont vraiment utiles. Le premier est le signal `"add"` est émis par un container dès que l'on ajoute un widget enfant à un container. Le second est le signal `"remove"` qui est émis au contraire lorsqu'un widget est retiré d'un container. Les fonctions de rappel associées à ces deux signaux doivent avoir le prototype suivant :

```
gboolean CallBack(GtkContainer *container,
                  GtkWidget *widget,
                  gpointer donnee);
```

Les `GtkContainers` introduisent deux arguments intéressants au mécanisme de création générique des widgets :

- `"GtkContainer::border_width"` qui est de type `gulong` et qui spécifie quel doit être le nombre de pixels à laisser autour des enfants.
- `"GtkContainer::child"` qui est de type `GtkWidget *` et qui permet de spécifier un enfant lors de la création du container.

1.3. Les GtkEventBox

Il existe un type dérivé des `GtkBins` qui est très pratique. Il s'agit des `GtkEventBox`. Certains widgets ne possèdent pas de fenêtre propre et dessine dans la fenêtre du widget qui le contient. C'est par exemple le cas des `GtkLabels`. Ceci a été fait pour limiter un peu l'utilisation de la mémoire du serveur X. En effet, chaque fenêtre utilise un peu de mémoire. Ceci amène cependant une autre limitation : ces widgets ne peuvent pas revecoir d'événements[2]. Si on veut utiliser ces widgets et tout de même recevoir des événements, il faut les inclure dans un container dont le rôle se limitera à recevoir les événements pour son unique enfant. Ce type de container est justement un `GtkEventBox`. On les crée avec la fonction suivante :

```
GtkWidget *gtk_event_box_new(void);
```

2. Les GtkBox

GtkObject
  GtkWidget
    GtkContainer
      GtkBox

Les `GtkBox` ou *boîtes* sont des dérivés des `GtkContainers`. Contrairement aux `GtkBins`, les `GtkBox` peuvent contenir autant de widgets que l'on veut. Les boîtes disposent leurs enfants alignés soit horizontalement, soit verticalement. Aussi, on ne crée habituellement pas de `GtkBox` directement, mais on utilise l'une de ses deux classes dérivées `GtkHBox` pour les boîtes horizontales ou `GtkVBox` pour les boîtes verticales. Les boîtes sont donc créées avec l'une des fonctions suivantes :

```
GtkWidget *gtk_hbox_new(gboolean homogene,
```

[2] à part ceux que le GTK synthétisent expressément comme l'événement *Expose* par exemple, afin que les widgets `NO_WINDOW` se déssinent tout de même...

```
                          gint espacement);
GtkWidget *gtk_vbox_new(gboolean homogene,
                          gint espacement);
```

Si paramètre `homogene` vaut TRUE, alors tous les widgets enfant auront la même taille dans la boîte, c'est à dire qu'ils auront tous la taille du plus large et du plus haut de la boîte, sinon les widgets enfants gardent leur largeur (ou hauteur) par défaut. Le paramètre `espacement` définit le nombre de pixels qui seront insérés entre deux widgets consécutifs.

Ces deux paramètres pourront être changés après la création des boîtes grâce aux deux fonctions :

```
void gtk_box_set_homogeneous(GtkBox *boite,
                             gboolean homogene);
void gtk_box_set_spacing(GtkBox *boite,
                         gint spacing);
```

Pour placer un widget dans une `GtkHBox` ou une `GtkVBox`, on utilise l'une des deux fonctions suivante :

```
void gtk_box_pack_start(GtkBox *boite,
                        GtkWidget *widget,
                        gboolean expand,
                        gboolean fill,
                        guint padding);
void gtk_box_pack_end(GtkBox *boite,
                      GtkWidget *widget,
                      gboolean expand,
                      gboolean fill,
                      guint padding);
```

La première fonction permet d'insérer un widget en haut dans une boîte verticale ou à gauche dans une boîte horizontale. La seconde insère le widget en bas ou à droite. Le paramètre `expand` contrôle qui de la boîte ou du widget doit adapter sa taille. S'il vaut `TRUE`, le widget est étiré de façon à occuper toute la place encore disponible dans la boîte. S'il vaut `FALSE`, c'est la boîte qui rétrécie. Le paramètre `fill` n'est utilisé que si `expand` vaut `TRUE`. Si c'est le cas et que `fill` vaut `TRUE`, c'est effectivement le widget qui est étiré, sinon `padding` pixels sont rajoutés autour du widget avant que celui-ci ne soit étiré. Souvent, on ne désire pas utiliser autant de paramètres pour simplement placer un widget dans une boîte. Aussi existe-t-il des versions simplifiées de ces versions :

```
void gtk_box_pack_start_defaults(GtkBox *boite,
                                 GtkWidget *widget);
void gtk_box_pack_end_defaults(GtkBox *boite,
                               GtkWidget *widget);
```

qui sont équivalentes aux appels suivants :

```
gtk_box_pack_start(boite, widget, TRUE, TRUE, 0);
gtk_box_pack_end(boite, widget, TRUE, TRUE, 0);
```

Notez aussi qu'il est possible de placer un widget dans une boîte à l'aide de la fonction `gtk_container_add()`, ce qui est équivalents à l'utilisation de `gtk_box_pack_start_defaults()` avec les mêmes paramètres.

Si l'on n'est pas satisfait de l'ordre dans lequel les enfants d'une boîte sont affichés, il est possible de changer la place qu'occupe un enfant particulier avec la fonction :

```
void gtk_box_reorder_child(GtkBox *boite,
                           GtkWidget *enfant,
                           gint position);
```

où `enfant` est le widget enfant de `boite` que l'on veut changer de place, et `position` est la nouvelle position que doit occuper l'enfant. Les positions commencent à 0, et augmentent jusqu'à *nombre d'enfants-1*.

Il est possible de savoir à tout moment comment un enfant a été placé dans une boîte en appelant la fonction :

```
void gtk_box_query_child_packing(GtkBox *boite,
                                 GtkWidget *enfant,
                                 gboolean *expand,
                                 gboolean *fill,
                                 guint *padding,
                                 GtkPackType *pack_type);
```

Au retour de cette fonction, les paramètres `expand`, `fill` et `padding` pointeront sur les valeurs qui ont été utilisée pour placer le widget `enfant` dans la `boite`. Le paramètre `pack_type` est un peu plus particulier. Il pointera sur une variable contenant soit `GTK_PACK_START` si le widget a été mis dans la boîte avec une fonction du type `gtk_box_pack_start*()`, soit `GTK_PACK_END` si le widget a été mis dans la boîte avec une fonction du type `gtk_box_pack_end*()`.

Il est aussi possible de changer ces paramètres, même après que le widget enfant ait été placé dans la boîte avec :

```
void gtk_box_set_child_packing(GtkBox *boite,
                               GtkWidget *enfant,
                               gboolean expand,
                               gboolean fill,
                               guint padding,
                               GtkPackType pack_type);
```

Les `GtkBox` introduisent deux arguments au mécanisme de création générique des widgets :

- `"GtkBox::spacing"` qui est de type `gint` et qui correspond au paramètre `espacement` des fonctions de création des boîtes ;
- `"GtkBox::homogeneous"` qui est de type `gboolean` et qui correspond au paramètre `homogene`.

Voici un exemple d'utilisation des `GtkBox` :

```
/* Boite.c */
#include <stdio.h>
#include <gtk/gtk.h>

void Affiche(GtkLabel *label, gchar *format)
{
  gchar *chaine;

  gtk_label_get(label, &chaine);
  printf(format, chaine);
}

void main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Label;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* La fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement de la boite */
  Boite = gtk_vbox_new(FALSE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);

  /* Les labels à l'intérieur */
  Label = gtk_label_new("Label 1");
  gtk_box_pack_start(GTK_BOX(Boite), Label, FALSE, FALSE, 0);

  Label = gtk_label_new("Label 2");
  gtk_box_pack_start(GTK_BOX(Boite), Label, TRUE, FALSE, 0);

  Label = gtk_label_new("Label 3");
  gtk_box_pack_start(GTK_BOX(Boite), Label, TRUE, TRUE, 10);

  /* On affiche le contenu de la boite */
  gtk_container_foreach(GTK_CONTAINER(Boite),
                        (GtkCallback)Affiche,
                        (gpointer)"%s est dans la boîte\n");
  /* On affiche le tout */
```

```
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
}
```

Si vous exécutez ce programme et que vous agrandi un peu la fenêtre, vous devriez obtenir ce qui est représenté sur la figure 1.

figure 1

De plus ce programme affiche ces trois lignes sur le terminal :

```
Label 1 est dans la boîte
Label 2 est dans la boîte
Label 3 est dans la boîte
```

3. Les GtkTables

GtkObject
  GtkWidget
    GtkContainer
      GtkTable

Les `GtkBox` souffrent d'une limitation : ils ne permettent de ranger les widgets que dans un seul sens. Il y a bien sûr la possibilité de places des boîtes verticales dans des boîtes horizontales, mais ce n'est ni très pratique ni très puissant. Les `GtkTables` permettent un placement dans une grille de manière très souple. Les `GtkTables` ou *tables* sont en fait un tableau comportant un certain nombre de lignes et de colonnes. Et les widgets enfants des containers `GtkTables` sont places dans les cases de ce tableau.

On crée les tables avec la fonction :

```
GtkWidget *gtk_table_new(guint lignes,
```

```
                        guint colonnes,
                        gboolean homogene);
```

où `lignes` et `colonnes` sont respectivement le nombre de lignes et de colonnes du tableau. Ces paramètres des `GtkTables` peuvent être changés par la suite grâce à la fonction suivante :

```
void gtk_table_resize(GtkTable *table,
                      guint lignes,
                      guint colonnes);
```

Le paramêtre `homogene` de `gtk_table_new()`, s'il vaut `TRUE`, indique que toutes les cases du tableau doivent avoir la même hauteur et la même largeur. Sinon, la taille des cases dépend du widget le plus haut sur la même ligne et du widget le plus large sur la même colonne. Ce paramêtre pourra également être changé par la suite en utilisant la fonction :

```
void gtk_table_set_homogeneous(GtkTable *table,
                               gboolean homogene);
```

Une fois que l'on a créé la tables, il est possible de mettre des widgets dedans à l'aide de la fonction suivante :

```
void gtk_table_attach(GtkTable *table,
                      GtkWidget *enfant,
                      guint x1,
                      guint x2,
                      guint y1,
                      guint y2,
                      GtkAttachOptions xoptions,
                      GtkAttachOptions yoptions,
                      guint espacementx,
                      guint espacementy);
```

Les parametres `x1`, `x2`, `y1` et `y2` définissent la ou les cases que va occuper le widget `enfant` dans la `table`. Si la table possède n colonnes, alors `x1` et `x2` doivent être compris entre 0 et n. Par exemple, si `x1` vaut 2 et `x2` vaut 3, alors le widget occupera la troisième colonne de la table. Les paramètres `y1` et `y2` fonctionnent de la même façon pour les lignes. En fait, on peut voir ces quatre paramètres comme les coordonnés des coins du widget à l'intérieur de la table.

La figure 2 montre comment est place un widget dans une table de 3 lignes et de 5 colonnes, si on utilise `x1`= 2, `x2`= 4, `y1`= 1 et `y2`= 3.

Les paramètres `xoptions` et `yoptions` sont des champs de bits qui définissent comment le widget doit réagir à un redimensionnement de la table. Les bits utilisables sont `GTK_FILL`, `GTK_SHRINK` et `GTK_EXPAND`. Le bit `GTK_FILL` a le même sens que le paramètre `fill` utilisé avec les boîtes. De même, le bit `GTK_EXPAND` à le même rôle que le paramètre `expand` utilisé pour les boîtes. Le bit `GTK_SHRINK` indique que le widget doit aussi rétricir si la table rétrici.

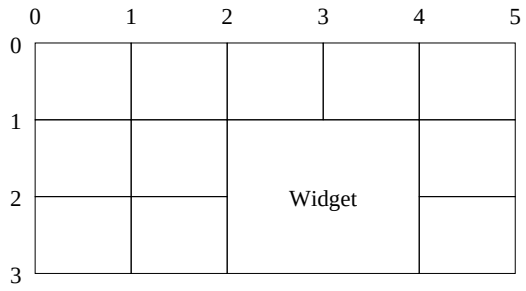


figure 2

Les paramètres `espacementx` et `espacementy` ne sont utilisés que si `xoptions` (resp. `yoptions`) ont au moins les bits `GTK_FILL` et `GTK_EXPAND` sont positionnés. Ils ont alors le même rôle que le paramètre `padding` des boîtes. C'est-à-dire que `espacementx` définit le nombre de pixels qui entoureront le widget à gauche et à droite dans sa case et `espacementy` définit le nombre de pixels qui entoureront le widget en haut et en bas dans sa case.

Cela fait tout de même énormement de paramètres à se souvenir. Il existe donc une version "allégée" pour placer un widget dans une table. Il s'agit de la fonction suivante :

```
void gtk_table_attach_defaults(GtkTable *table,
                               GtkWidget *enfant,
                               guint x1, guint x2,
                               guint y1, guint y2);
```

L'appel à cette fonction est équivalent à l'appel suivant :

```
gtk_table_attach(table, enfant,
                 x1, x2, y1, y2,
                 GTK_FILL | GTK_EXPAND,
                 GTK_FILL | GTK_EXPAND,
                 0, 0);
```

car ces réglages sont souvent ceux que l'on souhaite.

Par défauts, les cases de la table sont collées les unes aux autres. Cela peut parfois être disgracieux.

Aussi, il est possible de rajouté des pixels avant et après une ligne ou une colonne particulière avec les fonctions suivantes :

```
void gtk_table_set_row_spacing(GtkTable *table,
                               guint ligne,
                               guint espacement);
void gtk_table_set_col_spacing(GtkTable *table,
                               guint colonne,
                               guint espacement);
```

Les paramètres `ligne` ou `colonne` désigne le numéro de ligne ou de colonne dont on veut changer l'`espacement`.

Si l'on désire placer le même espacement autour de toutes les lignes de la table, il vaut mieux utiliser la fonction suivante :

```
void gtk_table_set_row_spacings(GtkTable *table,
                                guint espacement);
```

De la même façon, pour placer le même espacement autour de toutes les colonnes de la table, on utilisera la fonction :

```
void gtk_table_set_col_spacings(GtkTable *table,
                                guint espacement);
```

Les `GtkBox` introduisent deux arguments au mécanisme de création générique des widgets :

- `"GtkTable::n_rows"` qui est de type `guint`, définit le nombre de lignes de la table ;
- `"GtkTable::n_columns"` qui est de type `guint`, définit le nombre de colonnes de la table ;
- `"GtkTable::row_spacing"` qui est de type `guint`, définit l'espacement vertical séparant 2 lignes consécutives ;
- `"GtkTable::column_spacing"` qui est de type `guint`, définit l'espacement horizontal séparant 2 colonnes consécutives ;
- `"GtkTable::homogeneous"` qui est de type `gboolean`, correspond au paramètre `homogene` de la fonction `gtk_table_new()`.

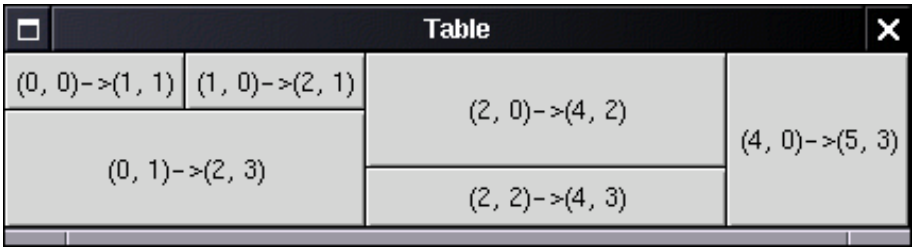


figure 3

Voici un exemple d'utilisation des `GtkTable` dont le résultat est visible sur la figure 3 :

```
/* Table.c */
#include <gtk/gtk.h>

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Table;
```

```
  GtkWidget *Bouton;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* Création de la fenêtre */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement de la table */
  Table = gtk_table_new(3, 5, TRUE);
  gtk_container_add(GTK_CONTAINER(Fenetre), Table);

  /* Les Boutons */
  Bouton = gtk_button_new_with_label("(0, 0)->(1, 1)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 0,1, 0,1,
                   GTK_FILL, GTK_FILL, 0, 0);

  Bouton = gtk_button_new_with_label("(0, 1)->(2, 3)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 0,2, 1,3,
                   GTK_FILL | GTK_EXPAND, GTK_FILL, 0, 0);

  Bouton = gtk_button_new_with_label("(1, 0)->(2, 1)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 1,2, 0,1,
                   GTK_FILL, GTK_FILL | GTK_EXPAND, 0, 0);

  Bouton = gtk_button_new_with_label("(2, 0)->(4, 2)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 2,4, 0,2,
                   GTK_FILL | GTK_EXPAND,
                   GTK_FILL | GTK_EXPAND, 0, 0);

  Bouton = gtk_button_new_with_label("(2, 2)->(4, 3)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 2,4, 2,3,
                   GTK_FILL | GTK_EXPAND | GTK_SHRINK,
                   GTK_FILL | GTK_EXPAND, 0, 0);

  Bouton = gtk_button_new_with_label("(4, 0)->(5, 3)");
  gtk_table_attach(GTK_TABLE(Table), Bouton, 4,5, 0,3,
                   GTK_FILL | GTK_EXPAND | GTK_SHRINK,
                   GTK_FILL | GTK_EXPAND | GTK_SHRINK, 0, 0);
  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
  return 0;
}
```

Cet exemple utilise des widgets que nous ne connaissons pas encore : les `GTk-Boutons`. Ils ont l'avantages sur les labels d'avoir une limite bien dessinée. Essayez de changer des paramètres, et de redimensionner la fenêtre principale de l'application afin de bien comprendre comment tout fonctionne.

10

Les GtkContainers Seconde partie

1. Les GtkFixeds

GtkObject
GtkWidget
GtkContainer
GtkFixed

Les `GtkFixeds` sont un autre descendant des `GtkContainers`. Ils peuvent contenir autant de widgets que l'on veut. Mais contrairement aux boîtes et aux tables qui alignent automatiquement leurs enfants, les `GtkFixeds` permettent à leurs enfants de choisir leur place.

Les `GtkFixeds` se créent simplement à l'aide de la fonction suivante :

```
GtkWidget *gtk_fixed_new(void);
```

Puis des widgets peuvent être placés dedant avec la fonction suivante :

```
void gtk_fixed_put(GtkFixed *fixed,
                   GtkWidget *enfant,
                   gint16 x, gint16 y);
```

où `x` et `y` sont simplement les coordonnées qu'occupera le coin supérieur gauche du widget `enfant` dans le `fixed`. Le `GtkFixed` se redimensionne automatiquement de façon à pouvoir effectivement contenir ses enfants. Le widget `enfant` restera ancré à cette position *fixe* même si le `GtkFixed` est redimensionné.

Il reste cependant possible de déplacer un widget enfant dans un `GtkFixed` à l'aide de la fonction suivante :

```
void gtk_fixed_move(GtkFixed *fixed,
                    GtkWidget *enfant,
                    gint16 x, gint16 y);
```

Voici un exemple qui montre l'utilisation des `GtkFixeds` :

```
/* Fixed.c */
#include <gtk/gtk.h>

void main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Fixed;
  GtkWidget *Label;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* La fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement du fixed */
```

```
  Fixed = gtk_fixed_new();
  gtk_container_add(GTK_CONTAINER(Fenetre), Fixed);

  /* Les labels à l'intérieur du fixed */
  Label = gtk_label_new("Label 1");
  gtk_fixed_put(GTK_FIXED(Fixed), Label, 5, 5);

  Label = gtk_label_new("Label 2");
  gtk_fixed_put(GTK_FIXED(Fixed), Label, 50, 50);

  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
}
```

Le résultat de ce programme est visible sur la figure 1

figure 1

2. Les alignements

GtkObject
GtkWidget
GtkContainer
GtkBin
GtkAlignment

Les `GtkAlignments` sont des dérivés des `GtkBins`, c'est-à-dire qu'ils ne contiennent qu'un seul autre widget (qui à son tour peut en contenir d'autres). En fait les alignements sont généralement insérés entre un container et l'un de ses enfants. Ils permettent d'aligner comme on le souhaite un widget enfant dans la place qui lui est normalement allouée.

Les alignements sont créés par la fonction suivante :

```
GtkWidget *gtk_alignment_new(gfloat xalign,
```

```
                                gfloat yalign,
                                gfloat xscale,
                                gfloat yscale);
```

Les paramètres `xalign` et `yalign` sont compris entre 0.0 et 1.0. Ils indiquent ou l'unique enfant sera placé dans l'alignement. Si ils valent 0.5, alors l'enfant sera placé au centre, les valeurs proches de zéro le place plus sur la gauche (resp. le haut) de l'alignement et les valeurs proches de 1.0 le place vers la droite (resp. le bas) de l'alignement.

Les paramètres `xscale` et `yscale` doivent aussi être compris entre 0.0 et 1.0. Ils indiquent quel pourcentage de la taille de l'alignement doit être aloué au widget enfant. Par exemple s'ils valent 1.0, alors le widget sera aussi grand que l'alignement ; s'ils valent 0.5 le widget aura la moitié de la taille de l'alignement.

Ces paramètres du `GtkAlignment` peuvent être changés par la suite grâce à la fonction suivante :

```
void gtk_alignment_set(GtkAlignment *alignment,
                       gfloat xalign,
                       gfloat yalign,
                       gfloat xscale,
                       gfloat yscale);
```

Il n'existe pas de fonction particulière pour placer un widget dans un `GtkAlignment`, on utilise donc la fonction générique à tout les container : `gtk_container_add()`.

Voici un exemple d'utilisation des `GtkAlignments` dont le résultat (après un redimensionnement de la fenêtre principale) est visible sur la figure 2 :

```
/* Alignement.c */
#include <gtk/gtk.h>

void main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Alignement;
  GtkWidget *Bouton;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* La fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement de l'alignement */
  Alignement = gtk_alignment_new(0.0, 0.5, 0.8, 0.3);
  gtk_container_add(GTK_CONTAINER(Fenetre), Alignement);
```

```
  /* Le bouton à l'intérieur de l'alignement */
  Bouton = gtk_button_new_with_label("Coucou");
  gtk_container_add(GTK_CONTAINER(Alignement), Bouton);

  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
}
```

figure 2

Les `GtkAlignments` introduisent quatre nouveaux arguements au mécanisme de création générique des widgets. Ils sont tous de type `gfloat` et correspondent exactement aux paramètres de fonction `gtk_alignment_new()`. Ce sont :

- `"GtkAlignment::xalign"` ;
- `"GtkAlignment::yalign"` ;
- `"GtkAlignment::xscale"` ;
- `"GtkAlignment::yscale"`.

3. Les GtkPaneds

GtkObject
  GtkWidget
    GtkContainer
      GtkPaned

Les `GtkPaneds` sont des containers un peu particuliers. Ils possèdent exactement 2 enfants. Ces enfants sont placés soit l'un au dessus de l'autre dans un *paned* vertical, soit l'un à côté de l'autre dans un *paned* horizontal. Les deux enfants sont séparés par une ligne munie d'une poignée (représentée par un petit carré) qui permet de déplacer cette ligne afin de choisir combien de place on alloue au deux enfants.

On ne crée pas directement de `GtkPaned`, au lieu de cela, on crée un de ces dérivés `GtkHPaned` ou `GtkVPaned` avec l'une des deux fonctions suiivantes :

```
GtkWidget *gtk_hpaned_new(void);
GtkWidget *gtk_vpaned_new(void);
```

La première crée un paned horizontal où les deux enfants seront placés côte à côte. La seconde crée un paned vertical où les deux enfant seront placés l'un au dessus de l'autre.

Une fois que l'on a créé le paned, on peut placer ses enfants dedans. Les paneds distingue le premier enfant (celui qui est en haut ou à gauche) du deuxième (celui qui est à droite ou en bas).

Pour insérer le premier enfant, on utilise :

```
void gtk_paned_pack1(GtkPaned *paned,
                     GtkWidget *enfant,
                     gboolean resize,
                     gboolean shrink);
```

Le paramètre `resize` indique que l'enfant accepte d'être redimensionné. En fait, il suffit que l'un des enfants ait ce paramètre à `TRUE` pour que la taille des deux puisse être changé à l'aide de la poignée.

Le paramètre `shrink`, s'il vaut `TRUE` indique que l'enfant accepte d'être redimensionné à un taille inférieure à sa taille naturelle.

Pour insérer le deuxième enfant, il faut utiliser la fonction suivante :

```
void gtk_paned_pack2(GtkPaned  *paned,
                     GtkWidget *enfant,
                     gboolean resize,
                     gboolean shrink);
```

qui utilise les mêmes paramètres.

Il existe des fonctions plus simple pour insérer un widget dans un `GtkPaned` :

```
void gtk_paned_add1(GtkPaned *paned,
                    GtkWidget *enfant);
void gtk_paned_add2(GtkPaned *paned,
                    GtkWidget *enfant);
```

La première est équivalente à l'appel :

```
gtk_paned_pack1(paned, enfant, FALSE, TRUE);
```

Et la seconde est équivalente à l'appel :

```
gtk_paned_pack2(paned, enfant, TRUE, TRUE);
```

La position de la poignée est normalement choisie par l'utilisateur, mais elle peut aussi être imposée par la fonction :

```
void gtk_paned_set_position(GtkPaned *paned,
                            gint position);
```

où `position` est le nombre de pixels alloués au premier enfant.

La taille par défaut de la poignée est de 10 pixels par 10 pixels, ceci peut être changé grâce à la fonction suivante :

```
void gtk_paned_set_handle_size(GtkPaned *paned,
                               guint16 taille);
```

où `taille` est la nouvelle taille de la poignée, qui reste toujours carrée.

Par défaut, l'espace laissé entre les deux enfants est de 6 pixels, ce qui fait que la poignée n'est pas complètement visible. Il est possible de changer cet espace à l'aide de la fonction suivante pour donner un aspect plus agréable :

```
void gtk_paned_set_gutter_size(GtkPaned *paned,
                               guint16 espace);
```

Voici un exemple d'utilisation des `GtkHPaned` et des `GtkVPaned`. Le résultat de ce programme après un redimensionnement de la fenêtre est visible sur la figure 3.

```
/* Paned.c */
#include <gtk/gtk.h>

void main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *PanedH;
  GtkWidget *PanedV;
  GtkWidget *Bouton;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* La fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement du paned horizontal */
  PanedH = gtk_hpaned_new();
  gtk_container_add(GTK_CONTAINER(Fenetre), PanedH);

  /* Création et attachement du paned vertical */
  PanedV = gtk_vpaned_new();
  gtk_paned_add2(GTK_PANED(PanedH), PanedV);

  /* Le bouton en haut du paned vertical */
  Bouton = gtk_button_new_with_label("Haut");
  gtk_paned_add1(GTK_PANED(PanedV), Bouton);

  /* Le bouton en bas du paned vertical */
  Bouton = gtk_button_new_with_label("Bas");
```

```
  gtk_paned_add2(GTK_PANED(PanedV), Bouton);

  /* Le bouton à gauche du paned horizontal */
  Bouton = gtk_button_new_with_label("Gauche");
  gtk_paned_add1(GTK_PANED(PanedH), Bouton);

  /* Changement de l'epace entre les deux enfants
   * du paned horizontal */
  gtk_paned_set_gutter_size(GTK_PANED(PanedH), 12);

  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
}
```

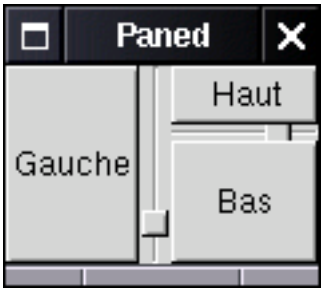


figure 3

4. Les GtkPackers

GtkObject
GtkWidget
GtkContainer
GtkPacker

Les `GtkPackers` certainement les plus versatiles des containers, ils sont aussi très certainement les plus complexes à mettre en œuvre. Ceux d'entres vous qui connaissent bien l'interface graphique `TK` peuvent toutefois le trouver familier.

On les créent facilement à l'aide de la fonction suivante :

```
GtkWidget *gtk_packer_new(void);
```

Il faut imaginer les *packers* comme une grande feuille de papier. Les widgets enfants sont placés dans un coin, sur un bord ou au milieu de la feuille. Puis cette partie

de la feuille devient indisponible pour les prochains widget que l'on voudra placer dans le packer. L'ordre dans lequel les widgets sont insérés est donc très important.

La fonction la plus générale pour placer un widget dans un packer est la suivante :

```
void gtk_packer_add(GtkPacker *packer,
                    GtkWidget *enfant,
                    GtkSideType cote,
                    GtkAnchorType ancrage,
                    GtkPackerOptions options,
                    guint largeur_bord,
                    guint pad_x,
                    guint pad_y,
                    guint i_pad_x,
                    guint i_pad_y);
```

Cela fait vraiment beaucoup d'arguments, mais en les prennant un par un, on arrive à comprendre le rôle de chacun. Tout d'abord, le paramètre `ancrage` indique dans quelle partie de la feuille le widget `enfant` sera placé. Le paramètre `ancrage` ne peut prendre que l'une des valeurs suivantes :

- `GTK_ANCHOR_CENTER` pour placer le widget au centre actuel de la feuille ;
- `GTK_ANCHOR_NORTH` pour placer le widget en haut, au milieu de la feuille ;
- `GTK_ANCHOR_NORTH_WEST` pour placer le widget en haut à gauche de la feuille ;
- `GTK_ANCHOR_NORTH_EAST` pour placer le widget en haut à droite de la feuille ;
- `GTK_ANCHOR_SOUTH` pour placer le widget en bas au milieu de la feuille ;
- `GTK_ANCHOR_SOUTH_WEST` pour placer le widget en bas à gauche de la feuille ;
- `GTK_ANCHOR_SOUTH_EAST` pour placer le widget en bas à droite de la feuille ;
- `GTK_ANCHOR_WEST` pour placer le widget au centre et à gauche de la feuille ;
- `GTK_ANCHOR_EAST` pour placer le widget au centre et à droite de la feuille.

Le paramètre `cote` indique quelle partie de la feuille est supprimée après que widget `enfant` ait été placé. Il est important de remarquer que le widget est placé *avant* que l'on coupe la feuille, et que la feuille est infiniment extensible. Le paramètre `cote` peut prendre les valeurs suivantes :

- `GTK_SIDE_TOP` alors les prochains widgets ne pourront être insérés que en dessous du widget `enfant` ;
- `GTK_SIDE_BOTTOM` alors les prochains widgets ne pourront être insérés que au dessus du widget `enfant` ;
- `GTK_SIDE_LEFT` alors les prochains widgets ne pourront être insérés qu'à droite du widget `enfant` ;
- `GTK_SIDE_RIGHT` alors les prochains widgets ne pourront être insérés qu'à gauche du widget `enfant`.

Le paramètre `options` est un champ de 3 bits :

- `GTK_PACK_EXPAND` a le même rôle que pour les boîtes ;

- `GTK_FILL_X` s'il est positionné, permet au widget `enfant` d'occuper toute la place disponible en largeur ;
- `GTK_FILL_Y` s'il est positionné, permet au widget `enfant` d'occuper toute la place dispônible en hauteur ;

Le paramètre `largeur_bord` définit le nombre de pixels qui seront laissés autour du widget. Les paramètres `pad_x` et `pad_y` ont un peu le même rôle, mais ils n'agissent qu'en hauteur (pour `pad_y`) ou en largeur (pour `pad_x`).

Les paramètres `i_pad_x` et `i_pad_y` définissent le nombre de pixels minimum à laisser à l'intérieur de la zone qu'occupe le widget. Dans les faits ils ont pratiquement le même rôle que `pad_x` et `pad_y`.

Tous les paramètres utilisés lors du placement d'un widget dans un packer peuvent par la suite être changés grâce à la fonction suivante :

```
void gtk_packer_set_child_packing(GtkPacker *packer,
                                  GtkWidget *enfant,
                                  GtkSideType cote,
                                  GtkAnchorType ancrage,
                                  GtkPackerOptions options,
                                  guint largeur_bord,
                                  guint pad_x,
                                  guint pad_y,
                                  guint i_pad_x,
                                  guint i_pad_y);
```

Heureusement, il existe une version simplifiée de la fonction `gtk_packer_add()` où les 5 derniers paramètres prennent une valeur par défaut :

```
void gtk_packer_add_defaults(GtkPacker *packer,
                             GtkWidget *enfant,
                             GtkSideType cote,
                             GtkAnchorType ancrage,
                             GtkPackerOptions options);
```

Les valeurs par défauts qui seront utilisées sont définies grâce aux fonctions suivantes :

```
void gtk_packer_set_default_border_width(GtkPacker *packer,
                                         guint bord);
void gtk_packer_set_default_pad(GtkPacker *packer,
                                guint pad_x,
                                guint pad_y);
void gtk_packer_set_default_ipad(GtkPacker *packer,
                                 guint i_pad_x,
                                 guint i_pad_y);
```

Nous avons vu que l'ordre dans lequel les widgets sont insérés dans le packer a une grande importance. Aussi est-il possible de changer cet ordre grâce à la fonction :

```
void gtk_packer_reorder_child(GtkPacker *packer,
```

```
                                GtkWidget *enfant,
                                gint position);
```

où `position` est la nouvelle position de l'`enfant` dans le packer. Si `position` vaut n, tout ce passe en fait comme s'il avait été véritablement introduit en $n^{\text{ième}}$. C'est-à-dire que tous les découpages de la feuille imaginaire sont recalculés.

Voici un programme qui utilise les `GtkPackers` pour disposer des boutons en spirale. Le résultat de ce programme est visible sur la figure 4.

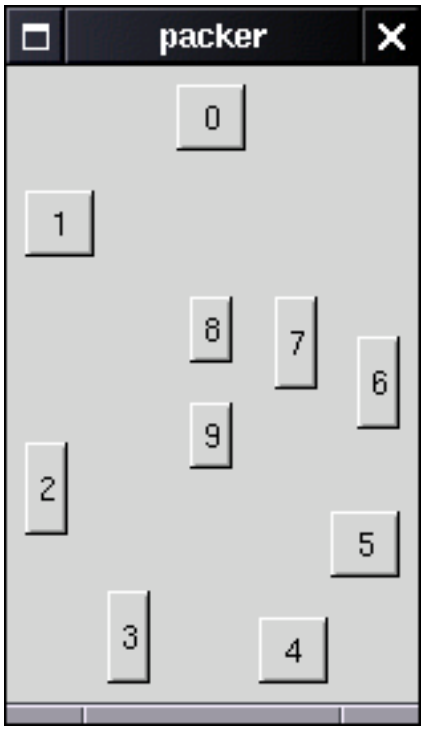


figure 4

```
/* Packer.c */
#include <gtk/gtk.h>

/* Les côtés */
GtkSideType side[] = { GTK_SIDE_TOP,
                       GTK_SIDE_LEFT,
                       GTK_SIDE_BOTTOM,
                       GTK_SIDE_RIGHT
                     };
/* Les points d'ancrage */
```

```
GtkAnchorType ancre[] = { GTK_ANCHOR_NORTH,
                          GTK_ANCHOR_NORTH_WEST,
                          GTK_ANCHOR_WEST,
                          GTK_ANCHOR_SOUTH_WEST,
                          GTK_ANCHOR_SOUTH,
                          GTK_ANCHOR_SOUTH_EAST,
                          GTK_ANCHOR_EAST,
                          GTK_ANCHOR_NORTH_EAST
                        };

void main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Packer;
  GtkWidget *Bouton;
  int i;

  /* Initialisation des bibliothèques */
  gtk_init(&argc, &argv);

  /* La fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* Création et attachement du packer */
  Packer = gtk_packer_new();
  gtk_container_add(GTK_CONTAINER(Fenetre), Packer);

  for (i=0 ; i<10 ; i++)
  {
    gchar *numero;

    numero = g_strdup_printf("%d", i);
    Bouton = gtk_button_new_with_label(numero);
    g_free(numero);
    gtk_packer_add(GTK_PACKER(Packer),
                   Bouton,
                   side[(i/2)%4],
                   ancre[i%8],
                   0,
                   5, 5, 5, 3, 3);
  }
  /* On affiche le tout */
  gtk_widget_show_all(Fenetre);
  /* La boucle principale */
  gtk_main();
```

```
}
```

Les `GtkPackers` introduisent cinq nouveaux arguements au mécanisme de création générique des widgets. Ils sont tous de type `guint` et correspondent aux valeurs par défauts des paramètres de `gtk_packer_add()` que l'on peut définir avec les fonctions `gtk_packer_set_default*()`. Ces arguments sont :

- `"GtkPacker::default_border_width"`
- `"GtkPacker::default_pad_x"`
- `"GtkPacker::default_pad_y"`
- `"GtkPacker::default_ipad_x"`
- `"GtkPacker::default_ipad_y"`

11

Les fenêtres toplevel

1. Les fenêtres toplevel

GtkObject
GtkWidget
GtkContainer
GtkBin
GtkWindow

Les `GtkWindows`, *fenêtres toplevel* ou plus simplement *fenêtres* sont les seules que le gestionnaire de fenêtres peut gérer. Et, par conséquent, tous les autres éléments d'une interface graphique doivent être inclus dans une telle fenêtre. Ce sont donc des containers de type dérivé de `GtkBin`. Les fenêtres toplevel sont normalement entourées d'un certain nombre de décorations comme une barre de titre, des barres de redimensionnement, des boutons permettant d'icônifier, de mettre en plan écran, d'envoyer le signal `"delete"`, etc.

Pour créer une fenêtre, on utilise, comme vous le savez déjà la fonction suivante :

```
GtkWidget *gtk_window_new(GtkWindowType type);
```

Le paramètre `type` peut prendre l'une de ces trois valeurs :

- `GTK_WINDOW_TOPLEVEL` qui est le plus utilisé, il demande que la fenêtre soit dessinée avec toutes les décorations habituelles ;
- `GTK_WINDOW_DIALOG` indique que la fenêtre n'est pas la fenêtre principale de l'application, elle est en général moins décorée ;
- `GTK_WINDOW_POPUP` indique que la fenêtre ne va pas rester longtemps affichée à l'écran, comme pour un menu par exemple. Ces fenêtres ne sont normalement pas décorée du tout.

Pour les fenêtres qui disposent d'une barre de titre, ce titre est généralement le nom de l'application. Si cela est pratique pour la fenêtre principale, çà l'est beaucoup moins pour toutes les autres, il est donc possible de changer le titre d'une `GtkWindow` grâce à la fonction :

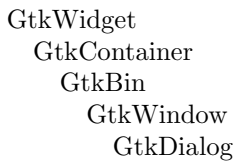
```
void gtk_window_set_title(GtkWindow *fenetre,
                          const gchar *titre);
```

où `titre` est le nouveau titre que l'on veut donner à la fenêtre.

GTK˙WIN˙POS˙NONE, GTK˙WIN˙POS˙CENTER, GTK˙WIN˙POS˙MOUSE
au dessus
grab auto
"GtkWindow : :type", GTK˙TYPE˙WINDOW˙TYPE "GtkWindow : :title", GTK˙TYPE˙STRING "GtkWindow : :auto˙shrink", GTK˙TYPE˙BOOL "GtkWindow : :allow˙shrink", GTK˙TYPE˙BOOL "GtkWindow : :allow˙grow", GTK˙TYPE˙BOOL "GtkWindow : :modal", GTK˙TYPE˙BOOL "GtkWindow : :window˙position", GTK˙TYPE˙WINDOW˙POSITION "GtkWindow : :default˙width", GTK˙TYPE˙INT "GtkWindow : :default˙height", GTK˙TYPE˙INT

2. Les GtkDialogs

GtkObject


GtkWidget
GtkContainer
GtkBin
GtkWindow
GtkDialog

12

Les GtkButtons et leurs dérivés

1. Les GtkButtons

GtkObject
  GtkWidget
    GtkContainer
      GtkBin
        GtkButton

Les boutons sont certainement les widgets les plus utilisés lors de la construction d'une interface graphique. Du point de vue de l'utilisateur, ils sont simplement une zone rectangulaire dont les bords sont bien définis et qui permet de déclencher telle ou telle action en cliquant dessus.

Habituellement les boutons sont simplement représentés par un rectangle plus ou moins mis en valeur par un effet de relief et contenant un label. Cependant GTK+ va plus loin. En effet, comme vous pouvez le voir sur le schéma d'héritage des `GtkButtons`, les boutons dans GTK+ sont des containers de type "bin" qui peuvent donc contenir à peu près n'importe quoi. Bien entendu, ils peuvent contenir un label, et des fonctions sont même prévues pour gérer ce cas particulier très répandu, mais rien n'empêche un bouton de contenir n'importe quelle sorte de widget, comme des GtkPixmaps, des GtkBox ou même des GtkNoteBooks[1]. Cependant, comme tous les containers de type "bin", un bouton ne peut contenir qu'un seul widget à la fois. Cette limitation se lève facilement en utilisant des `GtkBox` ou des `GtkTables` à l'intérieur d'un bouton.

Voyons à présent en détail comment utiliser les `GtkButtons`. La manière la plus simple de créer un bouton directement utilisable, c'est-à-dire non vide, est d'appeler la fonction :

```
GtkWidget *gtk_button_new_with_label(const gchar *texte);
```

qui crée un bouton contenant un label dont le texte est passé en paramètre. Attention ! Cette fonction crée effectivement *deux* widgets : un pour le bouton et un pour le label. Et la seule relation de dépendance entre le bouton et le label est que le label est contenu par le bouton. En particulier, on peut tout à fait cacher le label en laissant le bouton visible. Par exemple, le bouton de la figure 1 a été créé par l'appel suivant :

```
Bouton = gtk_button_new_with_label("Cliquez moi !");
```

fig. 1

Cependant, comme je l'ai dit, la vraie force de GTK+ est qu'il permet de considérer un bouton comme un container, c'est-à-dire qu'il est vide au départ, et peut contenir tout ce que l'on veut par la suite. Pour créer un tel bouton générique, on appelle simplement la fonction :

[1] ceci n'est pas forcément très conseillé pour rendre une interface utilisateur attractive...

```
GtkWidget *gtk_button_new(void);
```

pixmap
GtkPixmap
GtkButton
GdkPixmap
GdkBitmap

qui crée un bouton vide comme on peut le voir sur la figure 2. Libre à nous maintenant de le remplir avec ce que nous voulons.

fig. 2

Le widget que l'on met le plus souvent dans un bouton (mis à part les labels évidemment) est très certainement le `GtkPixmap`. En effet, les icônes cliquables que l'on retrouve dans de plus en plus de programmes ne sont ni plus ni moins que des pixmaps dans des boutons. Et donc en GTK+, se sont des `GtkPixmaps` dans des `GtkButtons`.

Pour créer de tels icônes cliquables, on crée d'abord le `GtkButton` et le `GtkPixmap` séparement. Puis en transtypant le bouton en `GtkContainer`, on place le pixmap dans le bouton[2]en appelant la fonction :

```
void gtk_container_add(GTK_CONTAINER(Bouton), Pixmap);
```

Ceci est possible justement car le type `GtkButton` est un descendant du type `GtkContainer`.

Voici un exemple qui crée justement un bouton contenant un pixmap :

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

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Bouton;
  GdkPixmap *Icone;
  GdkBitmap *Mask;
  GtkWidget *Pixmap;

  gtk_init(&argc, &argv);
  /* Création de la fenêtre principale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  /* On a besoin d'un widget "réalisé".
   * Ainsi, il possède une fenêtre X propre que
   * l'on peut utiliser pour créer le pixmap */
  gtk_widget_realize(Fenetre);
  /* On laisse un peu de place autour du bouton... */
```

[2] On pourra alors récupérer un pointeur sur le `GtkPixmap` dans le champ :

```
(GTK_BIN(Bouton))->Child
```

.

```
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre), 10);

  /* Création du bouton (vide) et
     placement dans la fenêtre */
  Bouton = gtk_button_new();
  gtk_container_add(GTK_CONTAINER(Fenetre), Bouton);

  /* Création du pixmap */
  Icone = gdk_pixmap_create_from_xpm(Fenetre->window,
                                     &Mask, NULL,
                                     "pixmap.xpm");
  Pixmap = gtk_pixmap_new(Icone, Mask);
  gdk_pixmap_unref(Icone); /* Comme on a plus besoin de */
  gdk_pixmap_unref(Mask);  /* ces deux-là, on libère la
                              mémoire qu'ils utilisent */

  /* Installation du pixmap dans le bouton */
  gtk_container_add(GTK_CONTAINER(Bouton), Pixmap);

  /* Affichage des trois widgets */
  gtk_widget_show_all(Fenetre);

  gtk_main();
}
```

Le résultat de ce programme est visible sur la figure 3.

fig. 3

Les boutons sont aussi beaucoup utilisés dans des fenêtres de dialogue où le bouton “OK” permet de confirmer un changement et le bouton “Annuler” permet de revenir sur notre idée. Ces deux boutons sont très souvent associés à un racourci clavier. GTK+ permet d'associer l'appui sur la touche *Entrée* du clavier à un signal pour un widget en particulier. Et pour un bouton, ce signal est évidemment celui qui correspond à un clic sur le bouton. Pour qu'un bouton puisse avoir cette propriété, il faut qu'il positionne son drapeau `GTK_CAN_DEFAULT` ainsi :

```
GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);
```

Cependant, rien ne nous empêche d'affecter cette propriété à plusieurs widgets dans la même fenêtre. De cette manière, le *défaut*[3] passe de widget en widget à l'aide de la touche *Tab*. En revanche, il peut être pratique de choisir quel est le widget qui doit avoir le *défaut* au moment de l'affichage de la boîte de dialogue, afin d'avoir une interface utilisateur cohérente. Un widget peut donc récupérer le *défaut* avec :

```
void gtk_widget_grab_default(GtkWidget *widget);
```

Cette propriété est surtout utilisée pour les boutons. Le bouton qui a actuellement le *défaut* est encadré d'un rectangle avec un effet de relief afin de le différencier facilement. Les boutons qui peuvent recevoir le *défaut* mais qui ne l'ont pas actuellement sont repérables par l'espace plus important qui les entoure. Cet espace est la place nécessaire pour dessiner le rectangle indiquant le *défaut*... Pour illustrer cela, regardez la figure 4 qui est le résultat du programme suivant :

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

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Bouton;

  gtk_init(&argc, &argv);
  /* Création de la fenêtre et de la boîte horizontale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre), 10);
  Boite = gtk_hbox_new(TRUE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);
  /* Création du bouton OK, qui peut avoir le defaut,
   * mais ne le prend pas... */
  Bouton = gtk_button_new_with_label(" OK ");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);
  /* Création du bouton Annuler, qui prend le défaut */
  Bouton = gtk_button_new_with_label(" Cancel ");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);
  gtk_widget_grab_default(Bouton);

  gtk_widget_show_all(Fenetre);
  gtk_main();
  return 0;
}
```

[3] Le *défaut* ou l'*action par défaut* désigne l'état dans lequel se trouve le widget d'une fenêtre qui sera activé par la touche *Entrée*.

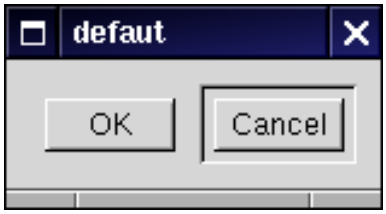


fig. 4

1.1. Les signaux propres aux boutons

Comme beaucoup de widgets, les `GtkButtons` introduisent de nouveaux signaux qui pourront être connectés avec les fonctions habituelles.

Ces signaux sont :

- **`"pressed"`** est émis au moment où le bouton gauche la souris est pressé au dessus du bouton. Ce signal est surtout utilisé par GTK+ lui-même afin de dessiner le bouton d'une manière différente pour signifer qu'il est enfoncé.
- **`"released"`** est émis dès que le bouton gauche de la souris est relaché, que le pointeur soit sur le bouton ou non. Ce signal aussi est surtout utilisé par GTK+, dans le but de redessiner le bouton dans l'état relâché. Bien entendu, le signal `"released"` n'est émis vers un bouton que si le bouton a auparavent reçu le signal `"pressed"`.
- **`"clicked"`** est en quelque sorte une synthèse des deux précédents. Il n'est cependant émis que si l'appui *et* le relâchement du bouton gauche de la souris ont eu lieu au dessus du bouton. C'est véritablement le signal que l'on utilisera le plus dans nos programmes, puisque, en termes simples, il est émis quand l'utilisateur clique sur le bouton.
- **`"enter"`** est le signal qui est émis quand le pointeur de la souris passe au dessus d'un bouton. GTK+ se sert de se signal pour mettre le bouton en surbrillance.
- **`"leave"`** est émis quand le pointeur de la souris quitte le bouton.

Tous ces signaux sont des signaux synthétisés par GTK+ et le prototype des fonctions de rappel associées est le suivant pour ces cinq signaux :

```
void fonction_de_rappel(GtkButton *Bouton, gpointer donnee);
```

Comme souvent, ces signaux peuvent être émis de façon artificielle en appelant directement les fonctions :

```
void gtk_button_pressed(GtkButton *bouton);
void gtk_button_released(GtkButton *bouton);
void gtk_button_clicked(GtkButton *bouton);
void gtk_button_enter(GtkButton *bouton);
void gtk_button_leave(GtkButton *bouton);
```

qui simulent les signaux correspondant pour le `GtkButton` passé en paramètre.

De plus, GTK+ permet une petite coquetterie qui permet de choisir le style du relief qui entoure un bouton à l'aide de la fonction :

```
void gtk_button_set_relief(GtkButton *bouton,
                           GtkReliefStyle type_de_relief);
```

où `type_de_relief` ne peut prendre que l'une des trois valeurs suivantes :

- **`GTK_RELIEF_NORMAL`** qui est la valeur par défaut, le pseudo relief est alors complètement dessiné,
- **`GTK_RELIEF_HALF`** qui demande à GTK+ de dessiner le bouton avec la moitié des effets de relief. En particulier, un bouton qui a le défaut et qui a un relief `GTK_RELIEF_HALF` n'a pas le rectangle indiquant qu'il a le défaut.
- **`GTK_RELIEF_NONE`** qui demande à GTK+ de dessiner le bouton sans aucun relief tant que le pointeur de la souris n'est pas placé dessus.

De la même façon, on peut savoir quel est le niveau de relief actuellement affecté à un bouton avec :

```
GtkReliefStyle gtk_button_get_relief(GtkButton *button);
```

Voici un exemple d'utilisation des différents reliefs, dont le résultat est montré sur la figure 5.

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

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Bouton;

  gtk_init(&argc, &argv);
  /* Création de la fenêtre et de la boîte horizontale */
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre), 10);
  Boite = gtk_hbox_new(TRUE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);
  /* Création du bouton ayant un relief normal */
  Bouton = gtk_button_new_with_label(" Normal ");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  gtk_button_set_relief(GTK_BUTTON(Bouton),
                        GTK_RELIEF_NORMAL);
  GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);
  gtk_widget_grab_default(Bouton);
  /* Création du bouton ayant un demi relief */
  Bouton = gtk_button_new_with_label(" Demi ");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  gtk_button_set_relief(GTK_BUTTON(Bouton),
                        GTK_RELIEF_HALF);
  GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);
```

```
  /* Création du bouton n'ayant aucun relief */
  Bouton = gtk_button_new_with_label(" Aucun ");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  gtk_button_set_relief(GTK_BUTTON(Bouton),
                        GTK_RELIEF_NONE);
  GTK_WIDGET_SET_FLAGS(Bouton, GTK_CAN_DEFAULT);

  gtk_widget_show_all(Fenetre);
  gtk_main();
  return 0;
}
```

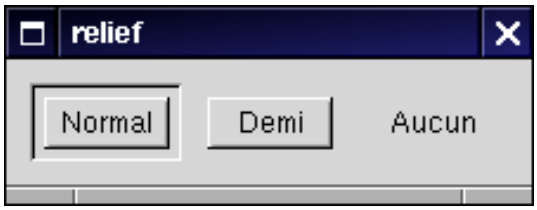


fig. 5

Le type `GtkButton` apporte deux nouveaux arguments au mécanisme de création générique des widgets :

- **`"label"`** qui est de type `gchar *` et que l'on peut lire et écrire.
- **`"relief"`** qui est de type `GtkReliefStyle` et que l'on peut lire et écrire.

Ainsi, le dernier bouton de l'exemple précédent aurait pu être créé en appellant :

```
Bouton = gtk_widget_new(gtk_button_get_type(),
                        "GtkButton::label", " Aucun ",
                        "GtkButton::relief",
                        GTK_RELIEF_NONE,
                        NULL);
```

Mais cette façon de créer les widgets est peu pratique et risque d'induire certains bogues difficiles à trouver. Par exemple, si vous oubliez la virgule après `"aucun"`, le programme compilera normalement, sans le moindre warning alors que l'exécution du programme entraînera une faute de segmentation très peu explicite.

2. Les GtkToggleButtons

GtkObject
  GtkWidget
    GtkContainer
      GtkBin
        GtkButton
          GtkToggleButton

Les `GtkToggleButtons` ou *boutons commutateurs* sont des boutons qui restent enfoncés au premier clic de la souris. Un deuxième clic et le bouton est relâché. Un troisième, il se renfonce, et ainsi de suite. À part ce comportement un peu particulier, les boutons commutateurs ressemblent énormément aux boutons normaux. C'est-à-dire que ce sont des containers *Bin* qui vont donc contenir un autre widget.

Vous l'aurez compris, on crée un bouton commutateur de la même façon que l'on crée un bouton avec les fonctions :

```
GtkWidget *gtk_toggle_button_new(void);
```

et

```
GtkWidget *
gtk_toggle_button_new_with_label(const gchar *label);
```

qui créent respectivement un bouton commutateur vide et un avec un label à l'intérieur.

Là où ils se différencient des boutons, c'est qu'ils possèdent un état persistant entre deux clics. Cet état est soit actif (enfoncé) soit inactif (relâché) et peut être connu à tout moment en appelant la fonction :

```
gboolean
gtk_toggle_button_get_active(GtkToggleButton *bouton);
```

qui renvoie la valeur `TRUE` si le bouton est enfoncé, et `FALSE` sinon.

De la même façon, on peut placer un bouton dans un état particulier en appelant la fonction :

```
void gtk_toggle_button_set_active(GtkToggleButton *bouton,
                                  gboolean actif);
```

Afin de réagir à un changement d'état, les boutons commutateurs introduisent un nouveau signal : **`"toggled"`** qui est émis à chaque fois qu'un bouton change d'état. C'est-à-dire que ce signal est émis quand l'utilisateur clique sur le bouton mais aussi si un appel à la fonction

```
gtk_toggle_button_set_active()
```

change l'état du bouton. Il ne faut donc surtout pas appeler cette fonction à l'intérieur de la fonction de rappel associée au signal `"toggle"` sous peine de rentrer dans une boucle sans fin. Bien entendu, les boutons commutateurs héritent également des signaux des boutons normaux et il est donc possible de récupérer le signal `"clicked"`, mais ce qui est vraiment intéressant est de savoir si un bouton commutateur a changé d'état, et pas s'il a été cliqué.

Comme habituellement, on peut simuler le signal `"toggled"` en appelant directement la fonction :

```
void gtk_toggle_button_toggled(GtkToggleButton *bouton);
```

Le type `GtkToggleButton` apporte deux nouveaux arguments au mécanisme de création générique des widgets :

- **"active"** qui est de type `gboolean` et que l'on peut lire et écrire. Il indique si le bouton doit être enfoncé ou relâché lors de sa création.
- **"draw_indicator"** qui est de type `GtkReliefStyle` et que l'on peut lire et écrire. Celui ci est très particulier et on l'utilise vraiment très peu. Il permet de définir si le bouton doit avoir sa propre fenêtre ou s'il utilise celle du widget parent. Il existe même une fonction permettant de choisir cela après la création du bouton commutateur :

```
void gtk_toggle_button_set_mode(GtkToggleButton *bouton,
                                gboolean draw_indicator);
```

Mais encore une fois, je déconseille l'usage de cette fonction à tous ceux qui ne maîtrisent pas parfaitement les widgets de GTK+.

Voici un exemple qui met en évidence la plupart des fonctionnalités des `GtkToggleButtons`. Le résultat de ce programme est visible sur la figure 6.

```
#include <stdio.h>
#include <gtk/gtk.h>

void FonctionRappel(GtkToggleButton *Bouton, char *Message)
{
  if (gtk_toggle_button_get_active(Bouton))
    printf("Le %s vient d'être enfoncé\n", Message);
  else
    printf("Le %s a été relâché\n", Message);
}

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Bouton;

  gtk_init(&argc, &argv);
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre),
                                 20);
  Boite = gtk_hbox_new(TRUE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);
  /* Le premier bouton, enfoncé au départ */
  Bouton = gtk_toggle_button_new_with_label("Bouton 1");
  gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(Bouton),
                               TRUE);
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
```

```
                    (gpointer)"bouton 1");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  /* Le second bouton, relâché au départ */
  Bouton = gtk_toggle_button_new_with_label("Bouton 2");
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                    (GtkSignalFunc)FonctionRappel,
                    (gpointer)"bouton 2");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);

  gtk_widget_show_all(Fenetre);

  gtk_main();

  return 0;
}
```

fig. 6

3. Les GtkCheckButtons

GtkObject
GtkWidget
GtkContainer
GtkBin
GtkButton
GtkToggleButton
GtkCheckButton

Les `GtkCheckButtons` sont ce que l'on appelle plus généralement des *cases à cocher*. Ils sont en effet représentés à l'écran par un widget (un label le plus souvent), précédé d'une case. Si l'on clique dessus, une coche se dessine sur la case et le bouton passe dans l'état actif. Si l'on reclique, la coche disparait et le bouton revient dans l'état inactif.

Les `GtkCheckButtons` sont donc très proches des boutons commutateurs dans leur fonctionnement et ils ne différent que par leur aspect a l'écran. C'est pourquoi on les manipule surtout via les fonctions destinées aux `GtkToggleButtons`. Seules les fonctions de créations différent :

```
GtkWidget *gtk_check_button_new(void);
```

```
GtkWidget *gtk_check_button_new_with_label(const gchar
                                                  *label);
```

On vérifiera par exemple l'état d'une case à cocher par :

```
if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(
                                  case_a_cocher))
  printf("La case est cochée\n");
else
  printf("La case est décochée\n");
```

en utilisant pleinement le mécanisme d'héritage des widgets de GTK+.

En changeant simplement les fonctions de création dans l'exemple précédent, on obtient la fenêtre représentée sur la figure 7.

fig. 7

On peut donc utiliser indifféremment les boutons commutateurs ou les cases à cocher. C'est surtout le contexte qui nous indiquera s'il est mieux d'utiliser l'un ou l'autre. Par exemple dans une boîte de dialogue de réglage d'options, on utilisera plutôt des cases à cocher, alors que dans une barre d'outils, on utilisera plutôt un bouton commutateur avec une icône.

4. Les GtkRadioButtons

GtkObject
  GtkWidget
    GtkContainer
      GtkBin
        GtkButton
          GtkToggleButton
            GtkCheckButton
              GtkRadioButton

Les `GtkRadioButtons` ou *boutons radios* ressemblent beaucoup aux cases à cocher (dont ils découlent) mais ils fonctionnent par groupes. Chaque bouton radio est associé à un groupe et un seul. Et à l'intérieur d'un groupe, un seul bouton radio peut être actif à un instant donné. En fait, les boutons radios sont une façon d'implémenter un choix d'une option parmis plusieurs. C'est pourquoi on les appelle parfois boutons options.

Il existe plusieurs façons de créer un bouton radio. Les deux premières doivent vous être plus ou moins familières :

```
GtkWidget *gtk_radio_button_new(GSList *groupe);
GtkWidget *gtk_radio_button_new_with_label(GSList *groupe,
                                           const gchar *label);
```

Ces deux fonctions ressemblent beaucoup aux fonctions de créations des autres types de boutons, mais on notera particulièrement la présense d'un paramètre supplémentaire : `groupe` qui indique à quel groupe de boutons radios doit appartenir le `GtkRadioButton` que l'on veut créer. Pour créer le premier bouton radio d'un groupe, on doit passer `NULL` comme premier paramètre à ces fonctions. Cela force GTK+ a créer un nouveau groupe que l'on pourra récupérer grâce à :

```
GSList *gtk_radio_button_group(GtkRadioButton *bouton_radio);
```

où l'unique paramètre est un des boutons radios appartenant au groupe. Et la valeur retournée par cette fonction sera utilisée comme premier paramètre lors de la création d'autres boutons radios appartenant à ce même groupe. Faites cependant attention car chaque création d'un bouton dans un groupe modifie ce groupe[4], il faut donc appeler cette fonction entre chaque création de bouton.

Voici un programme exemple qui crée trois boutons radios dans un seul groupe en utilisant cette méthode de création. Le résultat de ce programme est le même que celui du suivant et est visible sur la figure 8.

```
#include <stdio.h>
#include <gtk/gtk.h>

void FonctionRappel(GtkWidget *Bouton, char *Message)
{
  if (gtk_toggle_button_get_active(
                              GTK_TOGGLE_BUTTON(Bouton)))
    printf("L'%s est maintenant active\n", Message);
}

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Bouton;
  GSList *Groupe;

  gtk_init(&argc, &argv);
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre),
                                 20);
```

[4] Ceci vient du fait que le groupe associé à des boutons radios est en réalité une liste simplement chaînée des différents boutons du groupe. À chaque création d'un nouveau bouton, celui-ci est inséré dans cette liste en tête de liste, ce qui modifie la valeur du premier pointeur du groupe

```
  Boite = gtk_vbox_new(TRUE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);
  /* Le premier bouton, enfoncé au départ */
  Bouton = gtk_radio_button_new_with_label(NULL,
                                           "Option 1");
  gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(Bouton),
                               TRUE);
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 1");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  /* On récupére le groupe */
  Groupe = gtk_radio_button_group(GTK_RADIO_BUTTON(Bouton));
  /* Le deuxième bouton, relâché au départ */
  Bouton = gtk_radio_button_new_with_label(Groupe,
                                           "Option 2");
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 2");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  /* On récupére le groupe */
  Groupe = gtk_radio_button_group(GTK_RADIO_BUTTON(Bouton));
  /* Le troisième bouton, relâché au départ */
  Bouton = gtk_radio_button_new_with_label(Groupe,
                                           "Option 3");
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 3");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);

  gtk_widget_show_all(Fenetre);

  gtk_main();

  return 0;
}
```

Vous remarquerez que l'on positionne l'état d'un bouton en utilisant la fonction :

```
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(Bouton),
                                          TRUE);
```

En effet, il n'existe pas de fonction propre aux boutons radios pour modifier leur état. Notez aussi qu'il est fortement conseillé d'appeler cette fonction au moins une

fois dans le processus de création des boutons radios sinon aucun ne sera actif[5] ! De plus cette fonction a un comportement légèrement différent lorsqu'elle est appelée sur un bouton radio. En effet, non seulement elle rend actif le bouton passé en paramètre, mais elle désactive aussi tous les autres boutons du groupe.

De plus, de même que les cases à cocher, les `GtkRadioButtons` empruntent aux `GtkToggleButtons` la fonction permettant de tester leur état.

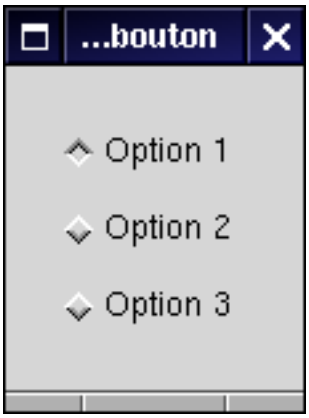


fig. 8

Il existe une autre façon de créer des boutons radios. Celle-ci permet de ne pas avoir à se soucier de cette notion de groupe. Cette création se fait à l'aide des deux fonctions :

```
GtkWidget *gtk_radio_button_new_from_widget(
                                        GtkRadioButton *groupe);
GtkWidget *gtk_radio_button_new_with_label_from_widget(
                                         GtkRadioButton *groupe,
                                         const gchar *label);
```

où le premier paramètre n'est cette fois ci plus un groupe mais un des boutons radios qui composent ce groupe. De la même façon que pour les fonctions utilisant les groupes, le premier bouton radio d'un groupe, on passe simplement la valeur `NULL` comme premier paramètre.

Pour éclaircir un peu tout ça, voici le programme précédent, légérement modifié pour utiliser ces fonctions. Le résultat est évidemment le même (fig. 8) :

```
#include <stdio.h>
#include <gtk/gtk.h>

void FonctionRappel(GtkToggleButton *Bouton, char *Message)
{
```

[5] De plus ce comportement dépend de la version de GTK+ utilisée. Certaines versions séléctionnent automatiquement le premier bouton du groupe

```
  if (gtk_toggle_button_get_active(Bouton))
    printf("L'%s est maintenant active\n", Message);
}

int main(int argc, char *argv[])
{
  GtkWidget *Fenetre;
  GtkWidget *Boite;
  GtkWidget *Bouton;

  gtk_init(&argc, &argv);
  Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_container_set_border_width(GTK_CONTAINER(Fenetre), 20);
  Boite = gtk_vbox_new(TRUE, 5);
  gtk_container_add(GTK_CONTAINER(Fenetre), Boite);
  /* Le premier bouton, enfoncé au départ */
  Bouton = gtk_radio_button_new_with_label_from_widget(
                                          NULL, "Option 1");
  gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(Bouton),
                               TRUE);
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 1");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  /* Le deuxième bouton, relâché au départ */
  Bouton = gtk_radio_button_new_with_label_from_widget(
                                    GTK_RADIO_BUTTON(Bouton),
                                    "Option 2");
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 2");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);
  /* Le troisième bouton, relâché au départ */
  Bouton = gtk_radio_button_new_with_label_from_widget(
                                    GTK_RADIO_BUTTON(Bouton),
                                    "Option 3");
  gtk_signal_connect(GTK_OBJECT(Bouton), "toggled",
                     (GtkSignalFunc)FonctionRappel,
                     (gpointer)"option 3");
  gtk_box_pack_start_defaults(GTK_BOX(Boite), Bouton);

  gtk_widget_show_all(Fenetre);

  gtk_main();
```

```
  return 0;
}
```

Le type `GtkRadioButton` apporte un nouvel argument au mécanisme de création générique des widgets. Cet argument se nomme **`"group"`** et est de type `GtkRadioButton`. C'est-à-dire qu'il s'agit d'un bouton radio appartenant au même groupe. Ainsi le dernier bouton de notre exemple aurait pu être créé en appelant la fonction suivante :

```
Bouton = gtk_widget_new(gtk_radio_button_get_type(),
                        "GtkButton::label", "Option 3",
                        "GtkRadioButton::group", Bouton,
                        NULL);
```

13

Les Menus

1. **Les GtkItems**
2. **Les GtkMenutems**
3. **Les GtkMenuCheckItems**
4. **Les GtkRadioMenuItems**
5. **Les GtkMenuTearOffItems**
6. **Les GtkMenuShells**
7. **Les GtkMenuBars**
8. **Les GtkMenus**
9. **Les GtkOptionMenus**

99

Les décorations

1. Les GtkFrames

GtkObject
GtkWidget
GtkContainer
GtkBin
GtkFrame

2. Les GtkAspectFrames

3. Les Séparateurs

99

Les entrées de texte

1. Les GtkEditables

GtkObject
  GtkWidget
    GtkEditable

2. Les GtkEntries

3. Les GtkDatas

4. Les GtkAdjustments

5. Les GtkSpinButtons

6. Les GtkTexts

7. Les GtkCombos

99

Les Ranges

1. Les GtkRanges

GtkObject
  GtkWidget
    GtkRange

2. Les GtkScrollbars