1 Gtk dans ses grandes lignes

1.1 Gtk, qu'est-ce que c'est ?

Gtk est une bibliothèque développée en C, c'est-à-dire que Gtk met à disposition des structures et des méthodes que l'on peut intégrer à un programme C. Parmi ces structures, on trouve des objets graphiques, tels que la fenêtre, le bouton, le label, etc.

1.2 Gtk, comment ça s'utilise ?

Lorsque votre programme C réalise des opérations d'entrée/sortie, vous écrivez en entête l'instruction :

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

Celle-ci inclut le fichier d'entête "stdio.h" dans votre propre fichier source et vous donne ainsi accès aux fonctions d'entrées/sorties (scanf, printf,...). Ces fonctions sont *déclarées* dans le fichier "stdio.h", dans lequel sont également définies les différentes structures et constantes symboliques associées. En revanche, les fonctions sont *définies* dans un autre fichier qui peut être compilé sous forme d'objet (fichier binaire d'extension usuelle ".o") ou de bibliothèque (sorte d'archive permettant de regrouper différents objets dans un même fichier, d'extension usuelle ".a"). Pour pouvoir compiler puis exécuter un programme qui fait appel à de tels éléments externes, il faut donc, lors de la compilation :

- pour la compilation, indiquer l'accés au fichier de déclaration (fichier ".h") ;
- pour l'édition de liens, indiquer l'usage et l'accès des méthodes compilées (fichier ".o" ou ".a").

Ainsi, pour utiliser Gtk, il faut :

1. dans la zone d'entête, placer la macro :

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

2. dans la commande de compilation ou dans le makefile, préciser que l'on utilise la bibliothèque Gtk (option "-l" de la commande gcc) ;
3. par le biais de variables d'environnement ou des options "-I" et "-L" de la commande gcc, localiser les répertoires d'accès (à "gtk/ght.h", à la bibliothèque).

Avantage : Gtk a tout prévu ! Pour ces deux derniers points, on n'a qu'à taper la commande 'pkg-config --cflags --libs gtk+-2.0' dans la ligne de commande de compilation :

```
gcc fichier.c 'pkg-config --cflags --libs gtk+-2.0'
```

1.3 De l'interface graphique à la programmation événementielle

Gtk permet de développer des interfaces graphiques... mais qu'est-ce qui caractérise une interface graphique ? Nous répondrons de façon très schématique par les deux points suivants :

- l'application affiche une fenêtre (qui contient éventuellement des menus, des boutons etc.) ;
- le déroulement de l'application est guidé par les interactions avec l'utilisateur (pas de logique séquentielle : on ne sait pas à l'avance ce que va faire l'utilisateur !).

D'un point de vue programmation, cela se traduit de la façon suivante :

- **Les conteneurs.** L'objet central (ou la structure C) qu'on manipulera sera l'objet "fenêtre" (type GtkWindow en Gtk). D'une part, on ne peut penser une interface sans fenêtre principale. D'autre part, tout élément graphique dépent nécessairement d'une fenêtre : un bouton, un label ou encore, une zone de saisie appartiennent à une fenêtre ; une fenêtre "pop up" est lancée à partir d'une autre fenêtre, etc. Cela fait apparaître deux relations : la notion de contenu/contenant d'une part (un bouton appartient au contenant qu'est la fenêtre), la notion de filiation d'autre part (une fenêtre est lancée, soit par la fonction "main" s'il s'agit de la fenêtre principale, soit par une autre fenêtre ; dans le premier cas la fenêtre est de premier niveau, dans le scond, elle est "fille" de la fenêtre appelante). Cette notion de filiation permet de gérer le retour à la fenêtre appelante lorsque la fenêtre appelée a terminé son traitement.

- **La programmation événementielle.** Comment coder une application dont on ne sait pas à l'avance ce qu'elle va faire ? Puisque c'est l'utilisateur qui guide le déroulement de l'application par interaction avec les objets graphiques, il s'agit alors de capter ces interactions. Le principe de base est le suivant : chaque objet, lorsqu'il subit une action de l'utilisateur (clic souris, saisie clavier, perte de focus,...), émet un signal. Il n'y a dès lors plus qu'à capter ces signaux et à agir en conséquence. C'est ce que fait Gtk : on lance une boucle infinie, appelée *boucle événementielle*, qui capte les signaux émis par les objets. Si, à un signal donné d'un objet graphique donné, une action est associée, alors cette action sera réalisée. Ainsi, ce qu'on programme en Gtk d'un point de vue applicatif revient à :

  – définir isolemment des fonctions pour un événement survenu à un objet ;

  – associer une fonction à un couple (signal, objet).

  En gros, en événementiel, si l'on ne maîtrise pas le déroulement séquentiel de l'application, on connaît toutefois le traitement à effectuer en cas d'événement survenu à un objet.

1.4 Implémentation de Gtk

1.4.1 Penser objet

Gtk est programmée en C, on manipule donc des structures et des fonctions C. Néanmoins, la bibliothèque Gtk a la particularité d'être *pensée objet*. Cela se traduit par le fait que les structures sont définies et organisées de façon hiérarchique, par *héritage*, qui traduit une relation de *spécialisation/généralisation*. Cela permet (notamment !) de ne pas avoir à définir plusieurs fois la même chose : une fenêtre de dialogue est une fenêtre particulière, une fenêtre est un objet graphique particulier. Ces trois objets partagent donc des caractéristiques communes (*i.e.*, attributs communs et méthodes communes, *ex.*, paramètre couleur, méthode d'affichage), qui sont définies au niveau de *l'objet graphique*, qui sont spécialisées et/ou enrichies d'abord au niveau *fenêtre* (*ex.*, une fenêtre peut contenir différents objets, ce qui n'est pas le cas de tout objet graphique), puis au niveau *fenêtre dialogue* (*ex.*, qui ne peut pas contenir de menu). L'objet graphique Gtk "de base" s'appelle GtkWidget. Tous les autres types Gtk sont donc des "*cas particuliers*" (*i.e.*, héritent) de GtkWidget.

1.4.2 Nommage, typage et pointage

Nommage Ne soyez pas effrayés par la syntaxe, considérez-la avec pragmatisme. D'une part, les objets (ou types) s'appellent tous "Gtk*QuelqueChose*" où *QuelqueChose* est le mot anglais permettant de désigner la structure implémentée (*ex.*, GtkLabel, GtkButton, GtkWindow,...). De même, les noms de fonction sont de la forme gtk_*QuelqueChose*_action : ils sont préfixés par "gtk" ; ils comportent ensuite le type d'objet que la fonction qu'ils désignent concerne ("window", "label", "button",...) ; ils comportent enfin le mot anglais permettant de décrire l'action de la fonction (*ex.*, "gtk_label_set_label" pour affecter le libellé d'un label, "gtk_button_new" pour créer un nouveau

bouton, “gtk_window_set_title” pour donner un titre à une fenêtre). Il s'agit là de conventions de nommage classiques (à toujours adopter d'ailleurs !) : par exemple, les fonctions de manipulation de chaînes en C sont bien préfixées par “str” pour “string”. Enfin, les noms de constante obéissent aux mêmes conventions que les noms de fonction, mais sont écrits en caractères majuscules (convention également classique), de sorte à bien les distinguer. Par exemple, “GTK_WINDOW_TOPLEVEL” est une constante concernant le niveau des objets GtkWindow (cette valeur signifie que la fenêtre est une fenêtre principale).

Quelques précisions : Gtk est construite à partir de la librairie Glib, qu'on utilise donc également. Tout objet Gtk est un objet Glib particulier (Gtk hérite de Glib) et certains objets qu'on sera amené à manipuler sont définis au niveau de Glib. Ces objets et les fonctions qui leur sont associées sont alors préfixés par “g” (*et non plus par “gtk”*). Par exemple, GObject et G_OBJECT désignent respectivement le type “objet générique Glib” et la macro de *conversion de type* associée (les macros de conversion de type sont abordées plus loin dans ce document).

Typage La gestion des types par héritage implique quelques précautions de codage. Puisque tout est GtkWidget, on déclarera tous nos objets comme GtkWidget (même si l'on sait que l'on veut définir un label ou une fenêtre) :

```
GtkWidget* pFenetre; //déclaration d'un pointeur sur GtkWidget
```

Mais à la création de l'objet (son instanciation), puisque c'est bien un objet “fenêtre” que l'on souhaite (*i.e.*, disposant des attributs de la structure fenêtre), il faudra faire appel à la fonction de création (constructeur) du type GtkWindow :

```
pFenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

De plus, si l'on utilise la variable pFenetre dans une fonction spécifique aux fenêtres, on précisera le type de pFenetre à l'appel de la fonction :

```
gtk_window_set_title(GTK_WINDOW(pFenetre));
```

Pour tout objet pObjet et pour tout type, la macro “GTK_QUELQUECHOSE(pObjet)” indique au compilateur qu'il doit considérer pour l'instruction courante la variable pObjet comme un pointeur sur le type QUELQUECHOSE. Vous utilisez déjà ce type de conversions, par exemple : pour obtenir le code ASCII d'une valeur de type “char” ; pour obtenir la partie entière d'une valeur flottante “float x=1.65;” ; pour convertir une variable “int” en une variable “unsigned int” (le premier bit est alors interprété comme une puissance de 2, et non plus comme le signe), *etc.* :

```
printf("Partie entière de %f = %d\n", x, (int)x);" // Affiche "Partie entière de 1.65 = 1"
```

Pointage Qu'est-ce qu'un pointeur ? Il s'agit tout simplement d'une variable qui encode une adresse. Alors, pourquoi est-ce que l'on doit préciser le type d'objet pointé lorsque l'on déclare un pointeur, autrement dit :

qu'est-ce qui différencie int* p; *de* float* p *?*

En fait, du point de vue du contenu de la variable p, cela n'a pas d'incidence : p prend la place nécessaire au stockage d'une adresse mémoire (soit, d'un entier) et contiendra donc toujours un entier. En revanche, le compilateur doit savoir, lorsque l'on fait référence, non plus à p, mais à l'objet *pointé par* p, comment lire/traiter cet objet. L'objet pointé par p, désigné par *p, est ce qui est stocké en mémoire à l'adresse p (à partir de la case numéro p, si vous préférez). Selon l'objet codé (un entier, un réel, un tableau, une fenêtre Gtk,...), on n'aura pas, ni le même nombre d'octets à lire, ni la même façon de les interpréter ! Voilà pourquoi on précise le type de la donnée pointée : c'est une façon de déclarer une variable par l'intermédiaire de son adresse.

En Gtk, on va manipuler des pointeurs sur des types de la librairie Gtk, essentiellement des pointeurs sur le type GtkWidget (en déclaration) qu'il faudra penser ensuite à "convertir" conformément à l'objet qu'ils pointent réellement, et ce en fonction du contexte d'utilisation. Pour plus d'information sur les pointeurs, vous pouvez (notamment !) vous référer aux chapitres 4&5 du document :

http://www-lipn.univ-paris13.fr/∼toulouse/doc/cexpress/cexpress.pdf

2 Les indispensables de la programmation d'une application Gtk

2.1 Les 3+1 lignes à connaître

Les 3 premières... Considérons le programme C suivant :

```
#include <gtk/gtk.h>              // (1) Inclure les fichiers d'entête
int main(int argc, char** argv)
{
    gtk_init(&argc, &argv);       // (2) Initialiser l'environnement Gtk
    gtk_main();                   // (3) Lancer la boucle événementielle
}
```

Ce code produit une application Gtk, qui compile et qui tourne.

(1) **#include <gtk/gtk.h>**
Toujours indiquer qu'on utilise les structures Gtk... *sinon : les objets Gtk ne sont pas reconnus, ça ne compile pas !*
(2) **gtk_init(&argc, &argv);**
La fonction gtk_init() initalise l'environnement Gtk (définition de valeurs par défaut et choses du genre) ; toujours placer cette fonction avant toute instruction impliquant un objet Gtk ! On peut passer des paramètres à cette fonction au moment du lancement de l'exécutable : ces paramètres sont transmis par la fonction "main".
gtk_main() est la fonction qui lance la fameuse boucle événementielle. C'est une boucle infinie qui se met en attente de réception des signaux. Tant qu'un objet n'aura pas invoqué l'arrêt de cette boucle, rien ne nous en fera sortir ! Si ce code compile et tourne, il le fait donc... à l'infini !

... la quatrième :

```
Dans une fonction :
    gtk_main_quit();    // (4) Quitter la boucle infinie
```

(4) **gtk_main_quit();**
gtk_main_quit() est justement la fonction qui permet d'arrêter la boucle événementielle. On ne peut l'invoquer de la fonction "main", puisque celle-ci reste bloquée sur la boucle infinie lancée par l'appel gtk_main(). En revanche, un objet pourra l'invoquer (par exemple : lorsque l'on clique sur le bouton "Fermer", sur le coin Nord-Est, ou encore sur le menu idoine de la fenêtre principale). Nous verrons plus tard comment associer une action à un événement survenu à un objet (cette association crée une fonction dite de *callback*). Mais tout d'abord... essayons d'afficher quelque chose.

2.2 Un premier objet graphique : la fenêtre

Considérons le programme C suivant :

```
#include <stdlib.h>
#include <gtk/gtk.h>

int main(int argc, char** argv)
{
    // (1) Declaration des variables
    GtkWidget* pFenetre;
    // Initialisation de l'environnement gtk
    gtk_init(&argc, &argv);
    // (2) Creation d'une fenêtre
    pFenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    // (3) Titre de la fenêtre
    gtk_window_set_title(GTK_WINDOW(pFenetre), "Ze hello world window...");
    // (4) Affichage de la fenêtre
    gtk_widget_show(pFenetre);
    // Lancement de la boucle événementielle
    gtk_main();
    return EXIT_SUCCESS;
}
```

Ce code produit une application Gtk, qui compile et affiche une fenêtre.

(1) **GtkWidget* pFenetre;**
Déclaration d'une variable qui est un *pointeur* sur un objet *GtkWidget*.
(2) **pFenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);**
La fonction gtk_window_new crée un objet de type GtkWindow (bien que le type renvoyé soit un pointeur sur GtkWidget). La variable pFenetre prend alors pour valeur l'adresse mémoire où est stocké l'objet qui vient d'être créé. La fonction gtk_window_new() prend un paramètre qui décrit le type de fenêtre ; pour la fenêtre principale, il faut mettre la valeur "GTK_WINDOW_TOPLEVEL".
(3) **gtk_window_set_title(GTK_WINDOW(pFenetre), "Ze hello world window...");**
La fonction gtk_window_set_title permet de donner un titre (second argument, ici, "Ze hello world window...") à une fenêtre (premier argument, ici, GTK_WINDOW(pFenetre)). Comme gtk_window_set_title() est définie au niveau de la structure GtkWindow, il faut préciser au compilateur qu'il doit interpréter la variable pointée par pFenetre comme un objet GtkWindow : c'est ce qui explique (et nécessite !) l'utilisation de la macro GTK_WINDOW.
(4) **gtk_widget_show(pFenetre);**
Ce n'est pas le tout de créer une fenêtre, encore faut-il l'afficher ! C'est ce que permet la fonction gtk_widget_show(). Comme cette fonction est définie au niveau de la classe GtkWidget et que l'objet pointé par pFenetre a été déclaré comme tel, on n'a pas besoin de spécifier autrement l'argument qui lui est donné. Ceci dit, on peut malgré tout écrire :

gtk_widget_show(GTK_WIDGET(pFenetre));

Cela ne change rien, mais utiliser systématiquement les macro de *conversion de type* permet d'éviter les erreurs.

Exercice 1

Une fois la fenêtre affichée... que se passe-t-il ? Et si l'utilisateur ferme la fenêtre ?

2.3 À la capture de l'événement

Il faut 3 éléments pour qu'un événement survenu à un objet génère une action :

1. La boucle événementielle doit être lancée (ben oui, sinon les signaux correspondant aux événements ne sont pas captés).

2. Une action (par l'intermédaire d'une fonction, plus précisément, d'un nom de fonction) doit être rattachée à l'objet pour le signal émis au cas où l'événement considéré se produirait.

3. La fonction en question (dite "de callback") doit être définie quelque part.

Connecter une fonction, un objet, un signal Poursuivons l'exemple précédent et considérons le programme C suivant :

```
#include <stdlib.h>
#include <gtk/gtk.h>

int main(int argc, char** argv)
{
    // Declaration des variables
    GtkWidget* pFenetre;
    // Initialisation de l'environnement gtk
    gtk_init(&argc, &argv);
    // Creation d'une fenêtre
    pFenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    // Titre de la fenêtre
    gtk_window_set_title(GTK_WINDOW(pFenetre), "Ze hello world window...");
    // Affichage de la fenêtre
    gtk_widget_show(pFenetre);
    // (1) Association de la fonction gtk_ main_ quit au signal destroy de la fenêtre
    gtk_signal_connect(GTK_OBJECT(pFenetre), "destroy", G_CALLBACK(gtk_main_quit), NULL);
    // Lancement de la boucle événementielle
    gtk_main();
    return EXIT_SUCCESS;
}
```

Il a le même effet que le code précédent... à la différence près qu'ici, lorsque l'on ferme la fenêtre, on quitte l'application.

(1) **gtk_signal_connect(GTK_OBJECT(pFenetre), "destroy", G_CALLBACK(gtk_main_quit), NULL);**
C'est la fonction gtk_signal_connect() qui permet d'associer une fonction de callback (troisième argument, ici, la fonction gtk_main_quit) à un objet graphique (premier argument, ici, pFenetre), qui est appelée lorsqu'un certain signal donné (deuxième argument, ici, "destroy") est émis par cet objet. Ainsi, dès que la fenêtre pointée par pFenetre émet le signal "destroy" (si l'on clique sur le coin Nord-Est de la fenêtre par exemple), la boucle événementielle capte celui-ci. Elle regarde alors si une action est attachée à l'objet émetteur pour ce signal. Ici, c'est le cas : la fonction gtk_main_quit est attachée à pFenetre pour le signal de destruction. On applique alors cette fonction... ce qui a pour effet, en l'occurrence, de quitter la boucle événementielle et donc, l'application.

La fonction gtk_signal_connect() est définie au niveau du type GtkObject. Par respect de la signature de la fonction, il faut penser à préciser :

- que l'objet passé en paramètre est un GtkObject (utilisation de la macro GTK_OBJECT) ;
- que la fonction passée en paramètre est un GCallback de la bibliothèque Glib (utilisation de la macro G_CALLBACK).

Nous verrons plus tard l'usage du dernier paramètre de la fonction gtk_signal_connect() (mis à NULL dans cet exemple). Une chose à retenir dès maintenant : chaque type Gtk émet un ensemble de signaux prédéfinis. En effet, à tous ne surviennent pas les mêmes événements : on ne clique pas sur une fenêtre tandis que l'on clique sur un bouton, on ne détruit pas un bouton tandis qu'on détruit une fenêtre. Pour chaque signal défini pour un type d'objet et pour chaque objet de ce type, on peut associer une fonction de callback.

Utilisation d'une fonction spécifique Si dans l'exemple précédent, on a associé au couple (objet pFenetre, signal de destruction) une fonction existante, on peut bien entendu aussi définir de nouvelles fonctions. De nouveau, nous poursuivons l'exemple précédent :

```
#include <stdlib.h>
#include <gtk/gtk.h>

// Déclarations
void OnDestroy(GtkWidget*, gpointer);

// Définitions
int main(int argc, char** argv)
{
    // Declaration des variables
    GtkWidget* pFenetre;
    // Initialisation de l'environnement gtk
    gtk_init(&argc, &argv);
    // Creation d'une fenêtre
    pFenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    // Titre de la fenêtre
    gtk_window_set_title(GTK_WINDOW(pFenetre), "Ze hello world window...");
    // Affichage de la fenêtre
    gtk_widget_show(pFenetre);
    // (1) Association de la fonction OnDestroy au signal destroy de la fenêtre
    gtk_signal_connect(GTK_OBJECT(pFenetre), "destroy", G_CALLBACK(OnDestroy), NULL);
    // Lancement de la boucle événementielle
    gtk_main();
    return EXIT_SUCCESS;
}

Callback de destruction associé à la fenêtre
void OnDestroy(GtkWidget* pWidget, gpointer pData)
{
    // quitter la boucle infinie
    gtk_main_quit();
}
```

Dans cet exemple, on définit une nouvelle fonction "OnDestroy". Cette fonction a pour effet d'appeler gtk_main_quit(). En associant cette fonction au couple (pFenetre, "destroy"), on obtient exactement le même résultat que dans l'exemple précédent : lorsque la fenêtre est détruite, elle émet le signal "destroy" ; la boucle infinie capte ce signal et voit qu'elle doit appliquer la fonction OnDestroy ; celle-ci appelle alors gtk_main_quit() et l'on sort de l'application.

La seule contrainte pour les fonctions de callback : leur signature. Le plus souvent, les fonctions que l'on souhaite associer à un couple (objet, signal) doivent avoir la signature suivante :

- *type retour* : void (pas de retour).
- *premier argument* : GtkWidget* pWidget
  pWidget désigne l'objet qui a lancé le signal : lorsque la fonction callback est lancée par la boucle infinie, celle-ci lui passe l'objet émetteur du signal en paramètre.
- *second argument* : gpointer pData
  Il s'agit d'un pointeur générique (gpointer en Glib est l'équivalent de void* en C) qui peut contenir tout et n'importe quoi (tout ce qu'on aura besoin de passer en argument à la fonction de callback).

Par exemple, imaginons une fenêtre qui contient un bouton B et un label L ; si l'action sur le bouton B doit modifier le libellé du label L, pour mettre à jour le libellé de L à la suite de l'événement "cliquer B", il faut que la fonction callback associée à "cliquer B" ait connaissance du label L dont

le libellé doit être modifié. Appelons changerLabel() cette fonction de callback : on passera alors le label L en argument de changerLabel(), et ce passage est indiqué au moment de l'attachement par gtk_signal_connect() de la fonction changerLabel() au signal "clicked" de l'objet B :

```
gtk_signal_connect(GTK_OBJECT(B), "clicked", G_CALLBACK(changerLabel), L);
```

En effet, l'instruction :

```
gtk_signal_connect(GTK_OBJECT(pObjet), strSignal, G_CALLBACK(nomFonction), pData)
```

A exactement l'effet suivant : lorsque l'objet *pObjet* émet le signal *strSignal*, la boucle événementielle appelle la fonction *nomFonction* avec les arguments *pObjet* et *pData* ; autrement dit, elle exécute l'instruction

```
nomFonction(pObjet, pData);        (dans notre exemple : "changerLabel(B,L);")
```

Exercice 2

1. Si l'on commente la ligne "gtk_main_quit();" dans la fonction "OnDestroy", que se passe-t-il ?
2. Si l'on commente la ligne "gtk_signal_connect(...)" dans la fonction "main", que se passe-t-il ?

Gestion mémoire

La fonction "gtk_main_quit()" nettoie l'environnement Gtk en détruisant tout ce qui a été construit. Attention : comme ce nettoyage est fait à l'extérieur de la fonction "main", les objets ne sont pas réinitialisés à NULL ; on n'a donc d'autre choix que de faire confiance à la fonction gtk_main_quit() pour la destruction des objets. Il est également possible de détruire explicitement les objets dans les fonctions de callback par appel à la fonction "void gtk_widget_destroy(GtkWidget*);" ; attention alors à bien prendre soin de détruire *d'abord* l'éventuel contenu de l'objet (c'est-à-dire, les éventuels objets graphiques qu'il contient).

3 Documentation

Une première source de documentation : les fichiers à entête ! Au Sercal, ils se situent dans le répertoire : "/usr/include/gtk-2.0/gtk". Pour les trouver, cherchez de toute façon le répertoire "include" lié à Gtk. Il peut vous être utile de consulter ces fichiers, pour connaître les fonctions définies sur un type d'objet donné, vérifier la signature de ces fonctions, ou encore connaître les différentes constantes qui sont définies (les types énumérés sont regroupés dans le fichier "gtkenums.h").