# Comparaison des API graphiques :

# GTK, QT, MFC, SWING

Coureux Eric
Maitrehut Loic
2002-2003

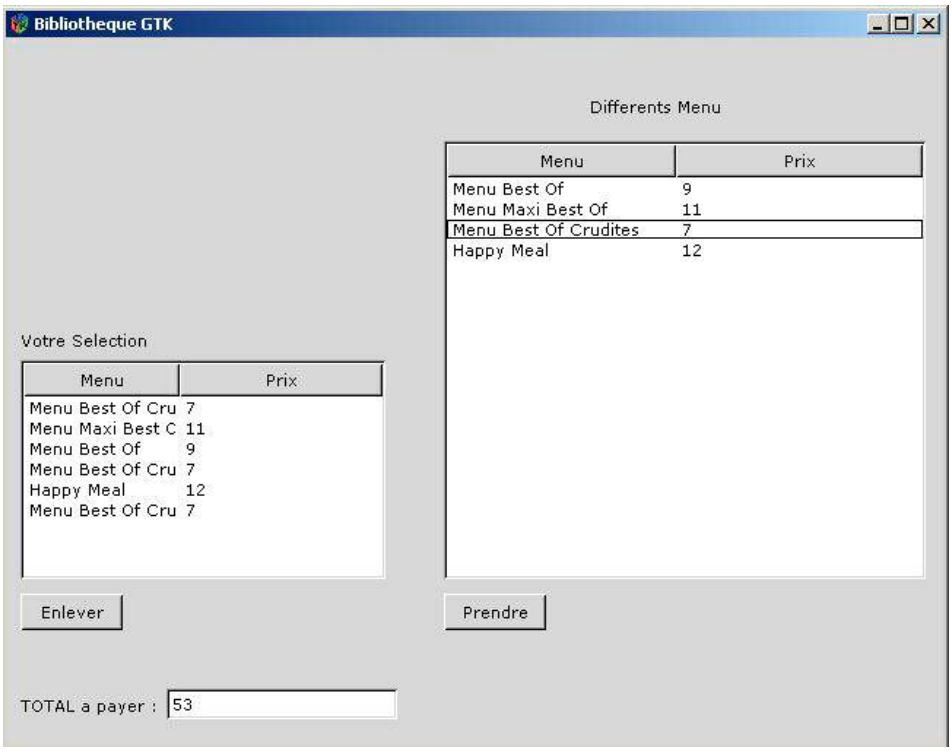
# Table des matières

# 1 Présentation du projet

Nous allons comparer les API SWING, MFC, GTK et QT pour voir ce qui les caractérise, et les comparer entre elles.

## 1.1 Un terminal point de vente ou caisse enregistreuse

Nous les testerons en réalisant 4 fois un même programme en faisant appel à chacune des librairies.




*Version avec l'API GTK*

C'est un programme qui permet de sélectionner son repas et de voir le prix final que l'on doit payer.
Ceci est un terminal point de vente ou caisse enregistreuse.

# 2 GTK+

## *2.1 Présentation*

### 2.1.1 Introduction

GTK = Gimp ToolKit, boite à outil utilisée d'abord par GIMP (General Image Manipulation Program), puis étendue pour pouvoir être utilisée pour n'importe quelle application.

#### 2.1.1.1 Licence

GTK est sous Licence LGPL , ce qui permet à tous de l'utiliser, de la faire évoluer, sans avoir à payer le moindre droit d'auteur et ce, même dans un contexte professionnel.

#### 2.1.1.2 Encapsulation

En fait, parler de LA bibliothèque GTK+ est légèrement incorrect puisqu'il s'agit en réalité de trois bibliothèques l'une sur l'autre.

##### 2.1.1.2.1 Glib

La première, Glib, est destinée à simplifier la vie du programmeur en proposant des fonctions de gestion de mémoire, de listes chaînées, de chaînes de caractères... La bibliothèque Glib est distribuée séparément de GTK+ depuis la version 1.1.x.

La *glib* fournit de nombreuses fonctions et définitions utiles, prêtes à être utilisées lorsqu'on crée des applications GDK et GTK. Beaucoup sont des répliques des fonctions standard de la *libc*.

##### 2.1.1.2.2 GDK

La deuxième, GDK (Gimp Drawing Kit), constitue une surcouche à la bibliothèque Xlib (la bibliothèque de base de X-Window). Elle représente une sorte d'abstraction entre GTK+ et X-Window; ce qui a permis, entre autres, de porter GTK+ sous d'autres environnements graphiques comme les Microsoft Windows.

##### 2.1.1.2.3 GTK+

La troisième est bien sûr GTK+ elle-même. Elle utilise activement les deux précédentes, et fournit les fonctions de gestion des Widgets.

### 2.1.1.3 Style de Programmation

Ce genre de bibliothèque exige une programmation essentiellement déclarative :

- On commence par dire à l'ordinateur de mettre des fenêtres aux endroits voulus, et de les peupler de la manière voulue.
- Puis on associe différentes actions ( par exemple, quitter le programme ,faire apparaître une nouvelle fenêtre,.. ) à certain événement (appuyer sur le bouton " quitter ", bouger la souris, cliquer à un certain endroit, etc).
- enfin, on informe l'ordinateur que cette partie descriptive est terminée et qu'il peut commencer à attendre les événements.

### 2.1.1.4 Multi-plateforme

GTK+ est un toolkit multi-plateforme :

- Win32
- BeOs
- Linux/Unix

### 2.1.1.5 Langage

La Bibliothèque GTK+ est disponible pour plusieurs langages :

- C (GTK+)
- C++ (GTK-)
- Ada
- Perl
- Python (pyGTK)

GTK+ est écrit en C orienté objet :

- implémente les notions de classe,
- d'héritage (simple),
- de méthodes virtuelles,
- de typage fort (à l'exécution)
- et de fonctions de rappel (callbacks).

## 2.1.2 Fonctionnement de GTK

### 2.1.2.1 « Widget »

Chaque élément (bouton, menu, barre …) d'une interface graphique est appelé « Widget », et pour en faire la déclaration rien de plus simple :

```
GtkWidget *NomDuWidget ;
```

En fait, tout ce qui peut constituer un élément d'interface utilisateur est un widget.
On trouve par exemple, les fenêtres, les boutons, les ascenseurs, les labels, les zones de textes, les boîtes invisibles qui permettent de ranger tout cela...

La bibliothèque GTK+ étant organisée de façon 'objet', chaque widget hérite des caractéristiques de ses ancêtres.

Ainsi, par exemple, le widget 'Dialogue' peut utiliser la fonction
`gtk_window_set_title(GTK_WINDOW(Dialogue), "Bonjour");`
car les widgets de type 'dialog' descendent directement des widgets de type 'window'.

### 2.1.2.2 Types de données

Les types en GTK peuvent etre des *gint*, *gchar*, etc., ce sont des redéfinitions respectivement de *int* et *char*.
Leur raison d'être est de s'affranchir des dépendances ennuyeuses concernant la taille des types de données simples lorsqu'on réalise des calculs. Un bon exemple est *gint32* qui désignera un entier codé sur 32 bits pour toutes les plateformes, que ce soit une station Alpha 64 bits ou un PC i386 32 bits.
Elles sont toutes décrites dans le fichier *glib/glib.h* (qui est inclus par *gtk.h*).
On notera aussi la possibilité d'utiliser un *GtkWidget* lorsque la fonction attend un *GtkObject*. GTK possède une architecture orientée objet, et un widget est un objet.

### 2.1.2.3 Les événements

GTK est dirigé par des événements : ce qui signifie que l'objet restera inactif jusqu'à ce qu'un événement Survienne. Il sera alors « transformé » en signal par le widget qui a été pressé.

```
/* retourne un identificateur de la fonction de rappel */
gint gtk_signal_connect( GtkObject *object,     /* widget émetteur */
                         gchar *name,           /* nom du signal */
                         GtkSignalFunc func,    /* fctn de rappel */
                         gpointer func_data );  /* données */
```

connecte le signal '*name*' de l'objet '*object*' (qui sera souvent, mais pas toujours, un widget) à la fonction *'func'* (appelée fonction de rappel ou '*CallBack*' en anglais) qui recevra le paramètre '*func_data*' en troisième paramètre.

En fait, à chaque fois qu'il arrive quelque chose à un widget, celui-ci envoie un signal à GTK+ qui appelle alors la fonction définie par défaut pour ce widget et ce signal.
C'est comme cela que, par exemple, GTK+ sait qu'il doit redessiner un widget qui redevient visible car l'utilisateur a bougé une fenêtre.
On peut bien sûr ajouter des fonctions à celles qui sont définies par défaut. C'est le rôle de la fonction `gtk_signal_connect`

### 2.1.3 Exemple

#### 2.1.3.1 Source

```
/* simple.c */
#include <gtk/gtk.h>
void main(int argc, char *argv[])
{
GtkWidget *Fenetre;
gtk_init(&argc, &argv);
Fenetre = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_widget_show(Fenetre);
gtk_main();
}
```

Il affiche simplement une fenêtre carrée

#### 2.1.3.2 Compilation

Pour compiler notre fichier bonjour.c :

gcc -Wall -g -I/usr/lib/glib/include bonjour.c -o bonjour_monde -L/usr/X11R6/lib -lgtk -lgdk -lglib -lXext -lX11
ou
gcc bonjour.c -o bonjour_monde `gtk-config --cflags --libs`
Le script gtk-config est un programme fourni avec la bibliothèque GTK+ et qui connaît les bons paramètres à passer à gcc pour compiler un programme GTK+.

### 2.1.4 Squelette d'un programme utilisant la bibliothèque GTK+

#### 2.1.4.1 les « Include »

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

Est obligatoire dans tout programme GTK+.
Par cette ligne, on inclut tous les symboles, types, variables, macros et fonctions de GTK+.

#### 2.1.4.2 Le prototype de la fonction main

```
void main(int argc, char *argv[])
```

doit comporter ces deux arguments car GTK+ en a besoin pour connaître d'éventuels arguments d'invocation que l'utilisateur pourrait passer à GTK+.

#### 2.1.4.3 L'appel de la fonction

```
gtk_init(&argc, &argv);
```

est également obligatoire. Elle initialise la bibliothèque GTK+ avec les arguments de la ligne de commande. Elle retire les arguments qu'elle a utilisés. Elle doit bien sûr être la première des fonctions GTK+ à être appelée.

#### 2.1.4.4 Une fenêtre

```
gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

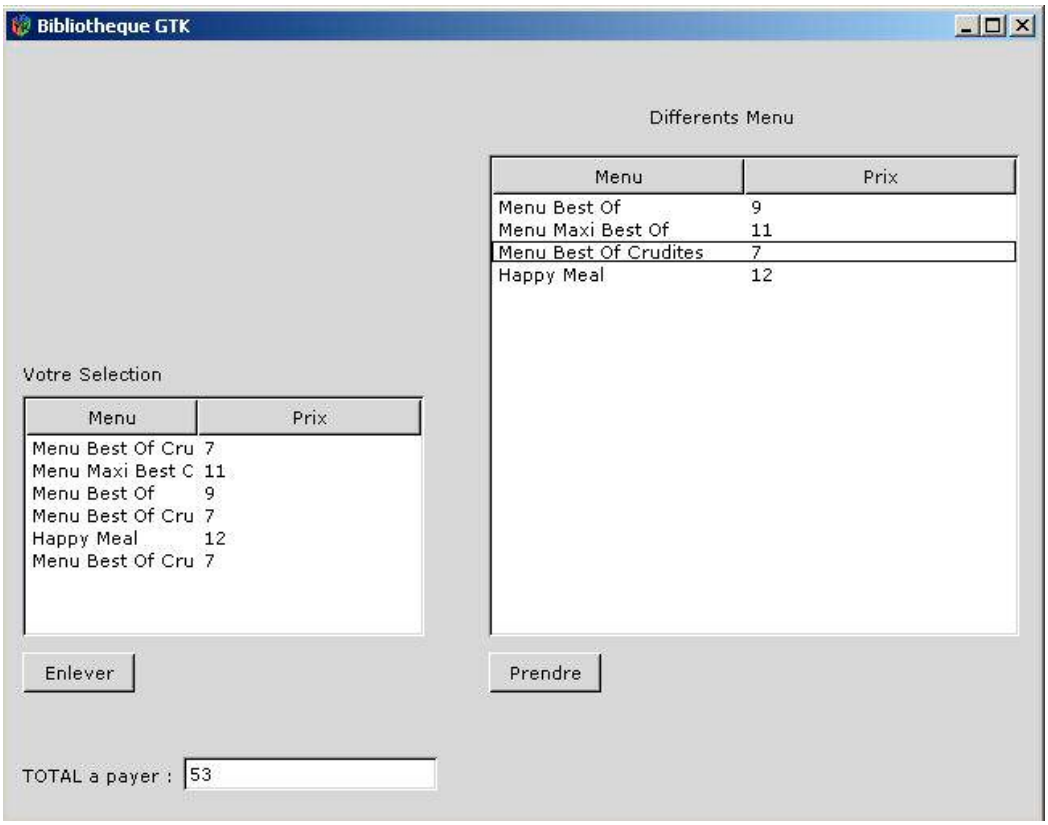
crée une fenêtre.
Le paramètre *GTK_WINDOW_TOPLEVEL* signifie qu'il s'agit d'une fenêtre qui doit être prise en compte par le window manager avec toutes les décorations habituelles.
Les autres valeurs possibles pour ce paramètre sont *GTK_WINDOW_DIALOG* (un peu moins de décoration) pour les boîtes de dialogue et *GTK_WINDOW_POPUP* (aucune décoration) pour les fenêtres destinées à être fermées rapidement.
La fenêtre n'est pas encore affichée, ni même prise en compte par GTK+. En fait, elle n'a même pas encore d'existence au sens X-Window. C'est le rôle de la fonction `gtk_widget_show(Fenetre);`

#### 2.1.4.5 La fonction « gtk_main() »

Lorsque le contrôle atteint ce point, GTK se met en attente d'événements X (click sur un bouton, ou appuie d'une touche, par exemple), de time-outs ou d'entrées-sorties fichier.

## 2.2 Le Programme

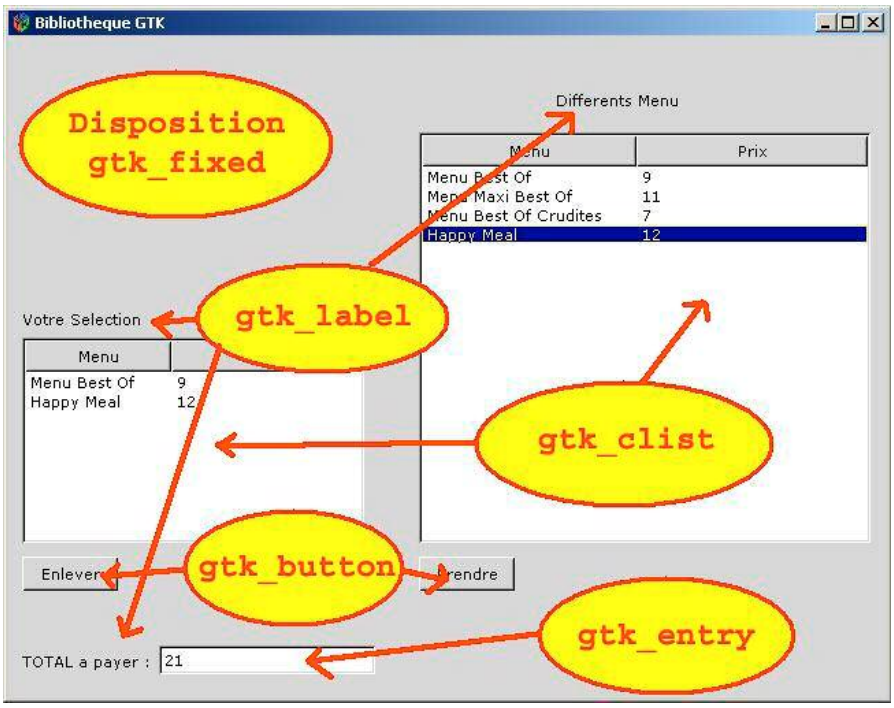
### 2.2.1 Notre Programme utilisant la bibliothèques GTK+




Cette application a été développée sous Windows pour montrer la portabilité de GTK+

### 2.2.1.1 Les « Widget »

On va créer une fenêtre (gtk_window), puis dans celle-ci on y mettra un « fixed » qui nous permettra ainsi de mettre tous nos « widget »

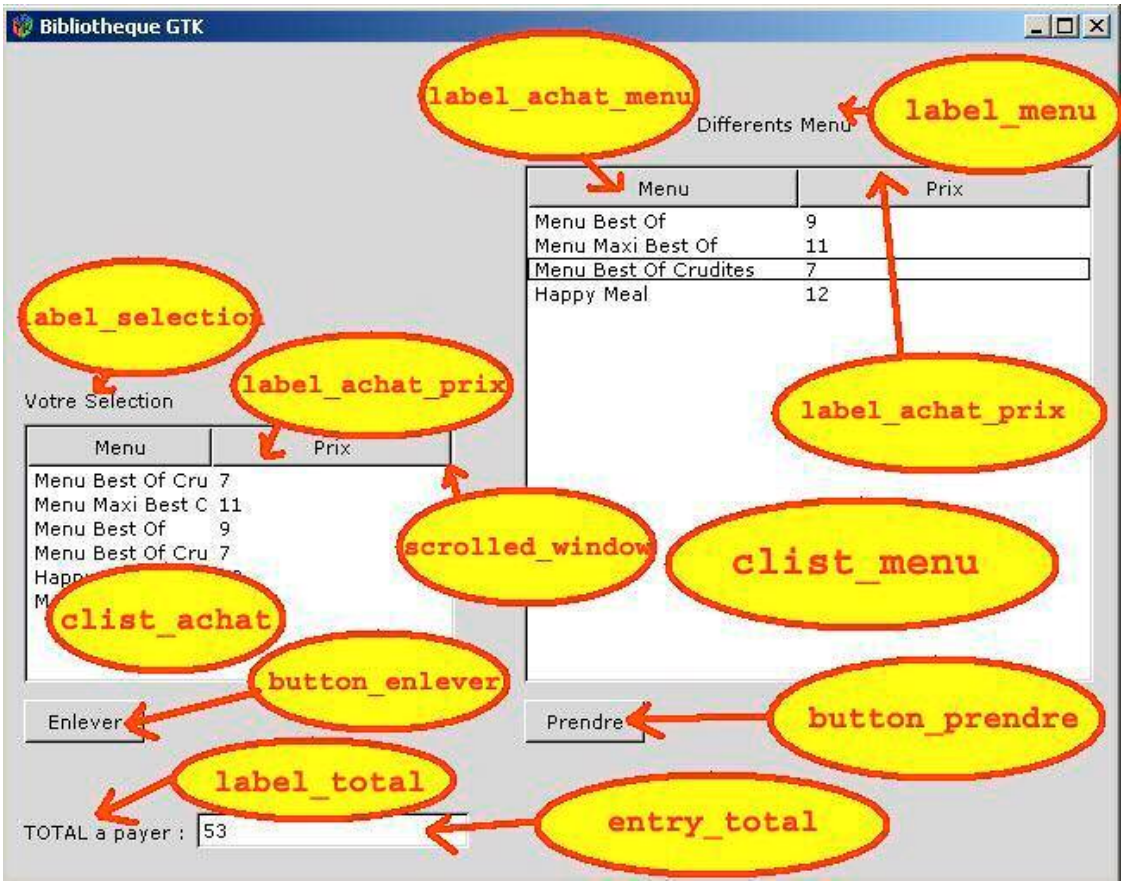


### 2.2.1.2 Déclaration de toutes les variables de type « widget »

Déclaration de pointeurs sur des structures de type GtkWidget :

```
GtkWidget    *window,
             *fixed,
             *button_prendre,
             *button_enlever,
             *scrolled_window,
             *label_selection_prix,
             *label_selection_menu,
       *label_achat_prix,
             *label_achat_menu,
             *label_selection,
             *label_menu,
             *label_total,
             *clist_menu,
             *clist_achat,
             *entry_total;
```

Le fait que toutes les variables sont de type (GtkWidget *) facilite le programmeur



### 2.2.1.3 Initialisation des Widgets

#### 2.2.1.3.1 Creation de la fenêtre

```
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

Juste une ligne nous permet de créer notre fenêtre (sans l'afficher)

#### 2.2.1.3.2 Mise en place du gestionnaire qui nous permettra de mettre tous nos widgets

```
fixed = gtk_fixed_new();
```

```
gtk_container_add(GTK_CONTAINER(window), fixed);
```

On inclut le « fixed » dans notre fenêtre.

« fixed » est un widget qui nous permettra de positionner tous nos futurs widget où l'on voudra sur notre fenetre, ceci juste en indiquant les coordonnées de positionnement.
Par exemple, lorsqu'on voudra mettre un bouton, on aura juste à écrire :

```
gtk_fixed_put (GTK_FIXED (fixed), button, 300, 380);
```

#### 2.2.1.3.3 Scrolled

```
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_show (scrolled_window);
gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_window),
GTK_POLICY_AUTOMATIC , GTK_POLICY_AUTOMATIC );
```

#### 2.2.1.3.4 Les boutons

```
button_enlever = gtk_button_new_with_label("Enlever");
```

```
button_prendre = gtk_button_new_with_label("Prendre");
```

#### 2.2.1.3.5 Les labels

```
label_selection = gtk_label_new("Votre Selection");
```

#### 2.2.1.3.6 Les Tableaux

```
clist_menu = gtk_clist_new(2);
gtk_container_add (GTK_CONTAINER (fixed), clist_menu);
gtk_clist_column_titles_show (GTK_CLIST (clist_menu));
gtk_widget_set_usize (clist_menu, 330, 300);
gtk_signal_connect(GTK_OBJECT(clist_menu), "select_row",
```

### 2.2.1.4 Placement des « widget »

```
gtk_fixed_put (GTK_FIXED (fixed),scrolled_window , 0, 0);

gtk_fixed_put (GTK_FIXED (fixed), button_enlever, 10, 380);

gtk_fixed_put (GTK_FIXED (fixed), button_prendre, 300, 380);

gtk_fixed_put (GTK_FIXED (fixed), label_selection, 10, 200);
...
gtk_fixed_put (GTK_FIXED (fixed), clist_menu, 300, 70);
...
```

Les placements des différents « widget » se font tout simplement en indiquant les coordonnées où l'on veut les placer.
Cependant il existe aussi les objets de type « Box », les gtk_hbox ou gtk_vbox, qui servent à aligner correctement les widget.

### 2.2.1.5 Mise en place des signaux

#### 2.2.1.5.1 Rappel du prototype

La fonction qui nous sert à connecter les signaux aux fonctions callback est :

```
gint g_signal_connect(gpointer *object,
                      const gchar *name,
                      GCallback  func,
                      gpointer func_data );
```

- **object** : pointeur sur l'objet qui émet le signal. Pour cela, il faut convertir le widget concerné avec la macro G_OBJECT(*widget).
- **name** : nom du signal émis par le widget.
- **func** : nom de la fonction callback à appeler. Il faut ici, utiliser la macro G_CALLBACK(func).
- **func_data** : pointeur sur une donnée quelconque qui sera passé à la fonction callback. Si aucune donnée n'est à envoyé à la fonction, il suffit de mettre la valeur NULL.

#### 2.2.1.5.2 Arréter la boucle événementielle

```
g_signal_connect(G_OBJECT(window),"destroy", G_CALLBACK(OnDestroy), NULL);
```

Dans notre programme, il serait bon de l'arrêter quand l'utilisateur clique sur la croix de fermeture de la fenêtre. Cependant ce n'est pas la boucle évènementielle qui traite cet événement, mais on peut lui dire que c'est nous qui allons traiter le signal "destroy" avec une fonction qui s'appelle fonction callback.
On va connecter ce signal à la fonction callback. Comme cela, dès que le signal "destroy" de la fenêtre survient, la boucle de traitement des signaux lance la fonction.

Cependant, on peut définir notre fonction callback avec un seul paramètre (GtkWidget *widget) si l'on n'a besoin d'aucune donnée supplémentaire, ou même d'aucun paramètre.
Par exemple, pour la fonction callback suivante, il nous est inutile de connaître le widget qui émet le signal. Nous avons donc une fonction sans paramètre d'entrée.

```
void OnDestroy(void)
{
    gtk_main_quit();
}
```

#### 2.2.1.5.3 Les autres signaux

- **Signal lors d'un click sur les boutons :**

```
gtk_signal_connect (GTK_OBJECT (button_prendre), "clicked",
                    GTK_SIGNAL_FUNC (maj_achat), fixed);
```

- **Signal lors d'un click sur le tableau :**

```
gtk_signal_connect (GTK_OBJECT(clist_menu),
                    "select_row",
                    GTK_SIGNAL_FUNC(selection_made), NULL);
```

- **Voyons le corps de la fonction qui est appelée en callback : « selection_made »**

Cette fonction vérifie si on a bien fait un double-click sur un produit, et appelle une fonction annexe (maj_achat) qui mettra à jour le tableau des achats, et ensuite met à jour le total à payer

```
void selection_made( GtkWidget      *clist, gint row, gint column,
                     GdkEventButton *event, gpointer data)
{
    char buffer [33];

    if (event->type==GDK_2BUTTON_PRESS){

        gtk_clist_get_text(GTK_CLIST(clist), row, 1, &prix);
        gtk_clist_get_text(GTK_CLIST(clist), row, 0, &menu);

        total += atoi(prix);
        maj_achat();

        itoa(total,buffer,10);
        gtk_entry_set_text((GtkEntry *)pEntry, buffer);
    }
}
```

prix et menu sont 2 variables globales pointant sur gchar,

On teste d'abord si l'utilisateur a fait un double-click grâce à l'événement « `GdkEventButton *event` »
`GdkEventButton` est une structure :

```
struct GdkEventButton
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  ...
};
```

Ce qui nous intéresse dans cette structure est le champ « type », c'est grâce à celui-ci qu'on pourra tester si on a bien fait un double-click

```
typedef enum
```

```
{
  GDK_NOTHING           = -1,
  GDK_DELETE            = 0,
  GDK_DESTROY           = 1,
  GDK_EXPOSE            = 2,
  GDK_MOTION_NOTIFY     = 3,
  GDK_BUTTON_PRESS      = 4,
  GDK_2BUTTON_PRESS     = 5,
  GDK_3BUTTON_PRESS     = 6,
  GDK_BUTTON_RELEASE    = 7,
  GDK_KEY_PRESS         = 8,
  ...
} GdkEventType;
```

Maintenant on peut savoir quel genre d'événement a été généré.

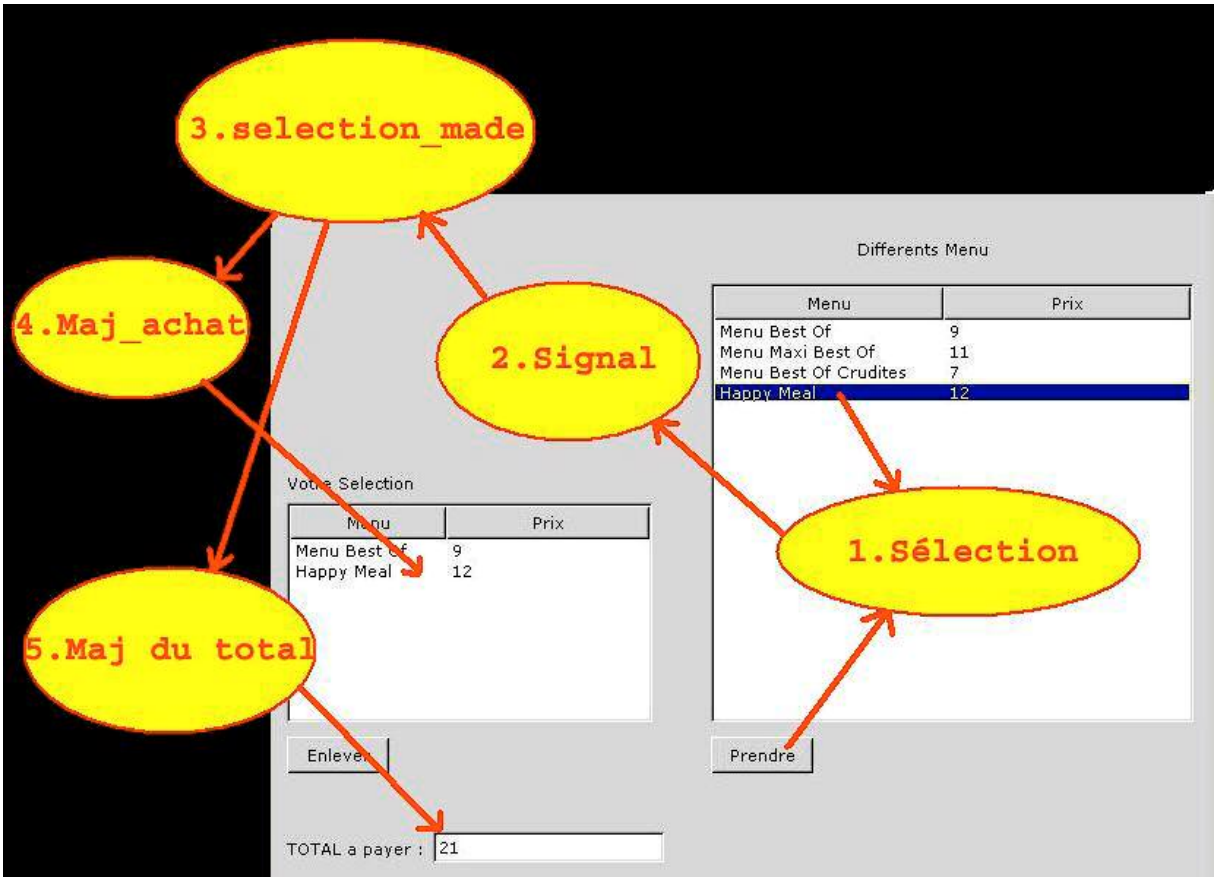
```
gtk_clist_get_text(GTK_CLIST(clist_menu), row, 1, &prix);
```
Ceci nous permet de récupérer la valeur de la case qui à été sélectionnée et de la mettre dans la variable « prix »

```
itoa(total,buffer,10);
```
Conversion de la variable globale « totale » en string pour pouvoir afficher le total à payer

```
gtk_entry_set_text((GtkEntry *) entry_total, buffer);
```
Mise à jour de notre widget « `entry_total` »

Procédure servant à mettre à jour le tableau des achats :

```
void maj_achat(void){

    drink[0][0] = malloc(sizeof(char)*10);
    strcpy(drink[0][0], menu);

    drink[0][1] = malloc(sizeof(char)*10);
    strcpy(drink[0][1], prix);

    gtk_clist_append((GtkCList *) clist_achat, drink[0]);
}
```




Récapitulatif des différentes étapes depuis le click souris jusqu'à la mise à jour du total.

## 2.2.1.6 Lancement

### 2.2.1.6.1 Affichage de tous nos widget

Nous avions jusqu'à présent déclaré, initialisé, positionné nos widget, il ne reste plus qu'à tous les afficher. Ceci se fait simplement en appelant :

```
gtk_widget_show_all(window);
```

### 2.2.1.6.2 Lancement de la boucle évènementielle

C'est seulement après avoir déclaré tous nos widget que l'on peut lancer la boucle évènementielle :

```
gtk_main();
```

## 2.3 Conclusions

### 2.3.1 Points positifs

- Permet de faire des interfaces très simplement et rapidement
- Une bonne API reference est disponible
- Plein de tutoriaux disponibles sur internet
- Licence LGPL
- Multi-plateforme
- Disponible pour plusieurs langages

### 2.3.2 Points négatifs

- Aucun

### 2.3.3 Bilan

GTK est une très bonne bibliothèque, essentielle au monde Unix/Linux mais qui est très peu utilisée dans le monde Windows.

Nous n'avons pas rencontré de gros problèmes pour faire cette application, ceci grâce :

- aux nombreux tutoriaux disponibles sur internet,
- à la facilité de programmation, quelques détails tels qu'en une ligne on peut créer une fenêtre, que toutes les variables sont de même type, tout ceci fait gagner du temps au programmeur.

## 2.4 Ressources

Liens internet qui nous ont servis pour notre recherche sur l'API GTK :

http://gtk.org
http://gtk-fr.org
http://linuxmag.linuxmag-france.org/old/lm6/initgtk.html
http://devernay.free.fr/cours/IHM/GTK1/coursgtk1.pdf
http://ohlabelleprise.sourceforge.net/cdc/node8.html
http://ogmwar.skreel.org/kjus/lesfenetres.php
http://www.linux-france.org/article/devl/gtk/gtk_tut-20.html

# 3 SWING

## *3.1 Présentation*

### 3.1.1 Introduction

SWING est une API (Application Programming Interface) développée par SUN Microsystems pour le langage JAVA dans le but d'apporter des composants graphiques plus riches que ceux déjà présents.

Swing sépare l'affichage et le contrôle d'un composant visuel de son contenu et propose des composants graphiques totalement pris en charge et dessinés par la JVM.

Cela rend l'aspect visuel indépendant du système d'exploitation et contribue à la portabilité du programme, argument principal de java.

#### 3.1.1.1 Encapsulation

Tous les composants de SWING dérivent d'une classe de l'AWT (Abstract Window Toolkit) qui fournie des méthodes d'accès à toutes les ressources graphiques de la machine. Les éléments graphiques utilisés sont liés à la plate-forme locale ce qui provoquait des différences d'apparence. Ceux de SWING sont écrits en Java, ce qui leur confère plus de souplesse et d'adaptabilité.
On peut d'abord distinguer ces nouveaux composants car leur nom commence par J.

Par exemple, la classe bouton est nommée Button en AWT, et JButton en SWING. Les paquetages à importer sont qualifiés par javax.swing au lieu de java.awt. Au point de vue héritage, les classes des composants "swing" étendent les fonctionnalités des classes des composants AWT analogues.

En terme d'importation de package, cela implique qu'un programme SWING doit, malgré tout, importer certains packages de l'AWT.

En conséquence, Swing est bien plus riche que l'AWT

#### 3.1.1.2 Plate-forme

SWING étant lié à JAVA il est "indépendant" de la plate forme et est donc disponible sur tous les OS (Linux, Windows, MacOS etc).

#### 3.1.1.3 Environnement de développement

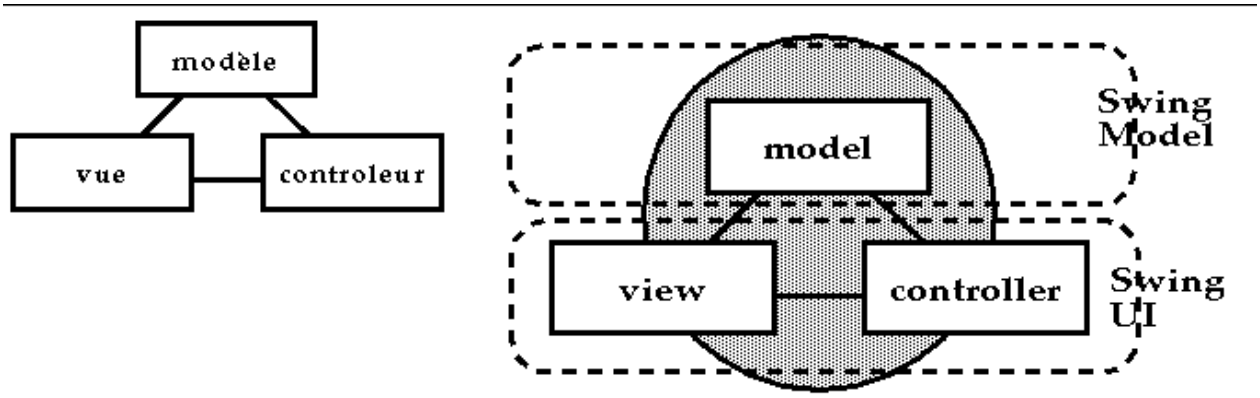
Il existe de nombreux environnements de développement pour Java ; cependant la société SUN propose son propre environnement (gratuit): le JDK (Java Développement Kit).

Swing a été introduit dans le JDK à partir de sa version 1.2

### 3.1.2 Fonctionnement

#### 3.1.2.1 Modèle-Vue-Contrôleur

Swing intègre une architecture basée sur l'approche " Modèle-Vue-Contrôleur " et sur les " Look & Feel .




##### 3.1.2.1.1 Le modèle

Le modèle représente les données de l'application. Il ne connaît rien de ses contrôleurs ou de ses vues.
Il possède quatre types de méthodes:

- Interrogation de son état interne,
- Manipulation de son état interne,
- Ajouter et enlever des écouteurs d'événement,
- Exécuter des événements .

##### 3.1.2.1.2 La vue et le contrôleur

Le contrôleur gère l'interaction de l'utilisateur avec le modèle.
Il intercepte le Focus (dans la vue) et le traduit en changements dans le modèle.

Il possède trois types de méthodes: peinture, retourne des informations géométriques, gestion d'événements.

SWING supporte ce type de modélisation explicitement avec ses widgets JList, JTable, JTree, JEditorPane, etc.

### 3.1.3 Les différents composants

#### 3.1.3.1.1 les composants "top-level"

Il en existe 3 types:

- `JFrame`
  Ce composant sert de base à l'interface graphique d'une application. On en trouve souvent une seule instance.

- `Jdialog`
  Les dialogues sont également des composants "top-level". Ils restent toutefois assujettis à une fenêtre de type frame.

- `JApplet`
  Ce composant sert de base à une interface graphique destinée à être télé-chargée.

#### 3.1.3.1.2 Les composants intermédiaires

Les composants intermédiaires sont destinés principalement à accueillir des composants de base.

- `JPanel`
  C'est le composant de base pour ranger des éléments.

- `JOptionPane`
  C'est le composant de base pour créer un dialogue.

- `JScrollPane`
  Permet l'utilisation d'un ou deux ascenceurs.

- `JSplitPane`
  Permet l'utilisation d'une barre de séparation.

- `JTabbedPane`
  Permet de disposer de plusieurs composants sur un même espace, et de passer de l'un à l'autre grâce à des onglets.

- `JToolBar`
  C'est une barre d'outils.

#### 3.1.3.1.3 Les composants de base

`JButton` `JToggleButton` `JRadioButton` `JComboBox` `JMenuBar` `JMenuItem` `JPopupMenu` `JSeparator` `JTextField` `JTextArea` `JLabel` `JProgressBar` `JToolTip` `JList` `JTable` `JColorChooser` `JFileChooser` `JTree` `JSlider` `JScrollBar` etc ...

### 3.1.3.2 Gestion des évènements

La gestion des événements se fait grâce aux `add***Listener` (*** = nom de l'événement à écouter) qui permettent de relier à un composant un objet "observateur" de ce composant chargé de réagir à certains événements.

### 3.1.3.3 La stratégie de placement

Une stratégie de placement se définit au niveau d'un conteneur grâce au groupe de classes appelé layout manager.

Il en existe 5 classes, plus quelques-unes introduites avec Swing. Les cinq premières sont contenues dans le package java.awt. Les dernières se trouvent dans le package java.swing.

#### 3.1.3.3.1 BorderLayout

Les composants sont en 5 endroits : NORTH, SOUTH, EAST, WEST, CENTER en prenant le maximum d'espace.

#### 3.1.3.3.2 BoxLayout

Positionne les composants avec la possibilité de les aligner.

#### 3.1.3.3.3 CardLayout

Plusieurs composants sont associés à la même zone d'affichage et peuvent être affichés ou non.

#### 3.1.3.3.4 FlowLayout

C'est le manager par défaut de chaque conteneur. Il rajoute simplement le composant les uns à la suite des autres.

#### 3.1.3.3.5 GridLayout

Il place les composants dans un tableau de lignes et colonnes de taille égale.

#### 3.1.3.3.6 GridBadLayout

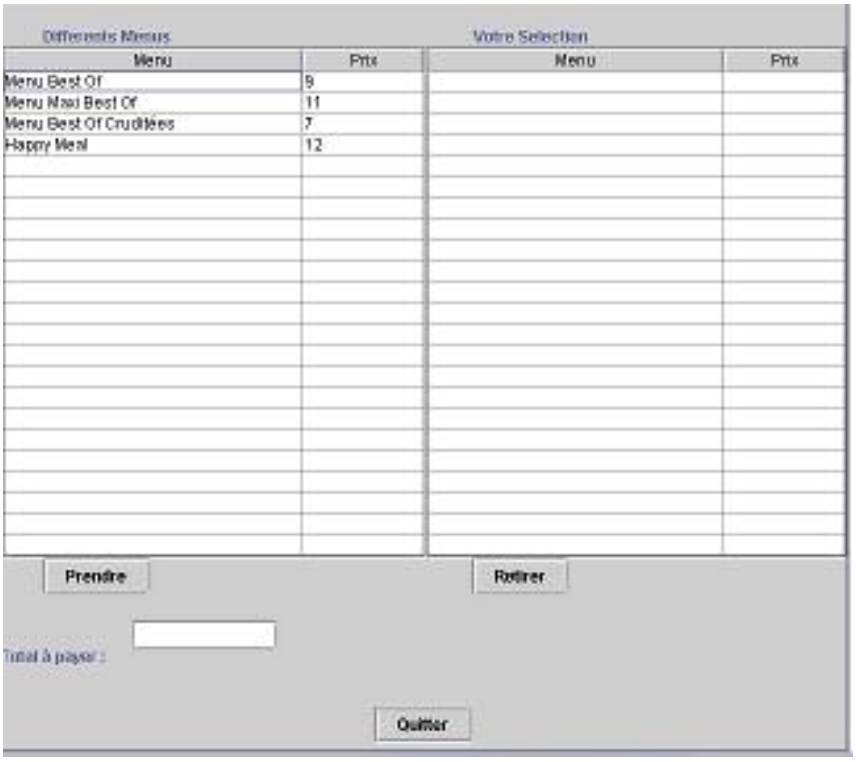
Idem , mais les composants peuvent occuper plus d'une cellule et les cellules ne sont pas forcément de même taille.

### 3.1.4 Exemple simple

Ceci est un exemple tout simple dont voici le résultat :


```
//on importe les classes nécessaires (remarquez que l'on a besoin du
pacquage AWT).
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

//déclaration de la classe
public class Exemple1 {


public static void main(String[] args) {

// initialisation du cadre
JFrame cadre = new JFrame("Test");

// création de quelques composants
// un bouton
JButton bouton = new JButton("Un bouton");
// une zone de texte
JLabel zonetexte = new JLabel("Une zone de texte");

// définition de la stratégie de placement du cadre
cadre.getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

// ajout des composants dans le panneau de contenu
cadre.getContentPane().add(zonetexte);
cadre.getContentPane().add(bouton);

// par principe, nous ajoutons un listener sur le bouton
// pour pouvoir fermer l'application d'un clic
bouton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
System.exit(0);
}
});

// dimensionnement et affichage du cadre
cadre.pack();
cadre.setVisible(true);
}
}
```

## 3.2 Notre programme




### 3.2.1 importation des composants (classes) necessaires

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import javax.swing.table.*;
import java.lang.Integer;
import java.lang.*;
```

### 3.2.2 declaration de la classe

```
public class MacDo extends JFrame {
```

### 3.2.3 declaration des variables privée

qui servent ici de variables globales

```
  private JTextField affPrix;
  private String chaine;
  private int col = -1;
  private int row = -1;
```

```
private int nbChoisie = 0;
private int total = 0;
private String[][] data = {
  {"Menu Best Of", "9"},
  {"Menu Maxi Best Of", "11"},
  {"Menu Best Of Cruditées", "7"},
  {"Happy Meal", "12"},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""}
};

private  String[][] data2 = {
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""},
  {"", ""}
};
```

### 3.2.4 declaration du constructeur

qui se sert du constructeur de la classe JFrame qui crée une fenêtre

```
public MacDo() {
  super("Bibliotheque MFC");

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  });
```

#### 3.2.4.1 Positionnement et dimensionement de la fenêtre

```
Toolkit tk = Toolkit.getDefaultToolkit();

setSize(650, 600);
setLocation(50, 50);
```

#### 3.2.4.2 Contenaire principal

```
Container cp = getContentPane();
```

#### 3.2.4.3 Boite principale

```
Box boiteMenus = Box.createHorizontalBox();
```

#### 3.2.4.4 Table de menus 1

##### 3.2.4.4.1 Noms des champs

```
String[] columnNames = {"Menu",
  "Prix"};
```

##### 3.2.4.4.2 Création et initialisation de la table

```
final JTable table = new JTable(data, columnNames);
```

#### 3.2.4.4.3 Définition de la taille des colonnes

```
TableColumn column = null;
for (int i = 0; i < 2; i++) {
  column = table.getColumnModel().getColumn(i);
  if (i == 0) {
    column.setMaxWidth(300);
  } else {
    column.setMaxWidth(100);
  }
}
```

#### 3.2.4.4.4 Le JScrollPane

une table doit etre contenue dans un JScrollPane sinon elle ne s'affiche pas

```
JScrollPane scrollPane = new JScrollPane(table);
scrollPane.setMaximumSize(new Dimension(600,388));
```

#### 3.2.4.4.5 Boite pour la table 1

```
Box boiteMenus1 = Box.createVerticalBox();
boiteMenus1.add(new JLabel("Differents Menus"),BorderLayout.EAST);
boiteMenus1.add(scrollPane);
```

#### 3.2.4.4.6 bouton prendre

```
JButton prendre = new JButton("Prendre");
boiteMenus1.add(prendre);

on ajoute la boite de table 1 a la boite principale
boiteMenus.add(boiteMenus1);
```

### 3.2.4.5 Table de menus 2

#### 3.2.4.5.1 noms des champs

```
String[] columnNames2 = {"Menu",
  "Prix"};
```

#### 3.2.4.5.2 on crée et initialise la table

```
final JTable table2 = new JTable(data2, columnNames2);
```

#### 3.2.4.5.3 on définie la taille des colonnes

```
TableColumn column2 = null;
for (int i = 0; i < 2; i++) {
  column2 = table2.getColumnModel().getColumn(i);
  if (i == 0) {
    column2.setMaxWidth(300);
  } else {
    column2.setMaxWidth(100);
  }
}
```

#### 3.2.4.5.4 Le JscrollPane

une table doit être contenue dans un JscrollPane, sinon elle ne s'affiche pas.

```
JScrollPane scrollPane2 = new JScrollPane(table2);
scrollPane2.setMaximumSize(new Dimension(600,388));
```

#### 3.2.4.5.5 La boite pour la table 2

```
Box boiteMenus2 = Box.createVerticalBox();
boiteMenus2.add(new JLabel("Votre Selection"));
boiteMenus2.add(scrollPane2);
```

#### 3.2.4.5.6 bouton retirer

```
JButton retirer = new JButton("Retirer");
boiteMenus2.add(retirer);
```

on ajoute la boite de table 2 à la boite principale

```
boiteMenus.add(boiteMenus2);
```

### 3.2.4.6 Les écouteurs de bouton

le bouton « retirer » possède maintenant un écouteur qui définit l'action qui lui est associée

```
retirer.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    faireRetirer(table2);
  }
});
```

le bouton « prendre » possède maintenant un écouteur qui définit l'action qui lui est associée

```
prendre.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    fairePrendre(table);
  }
});
```

on ajoute la boite principale à la fenêtre

```
cp.add(boiteMenus,BorderLayout.NORTH);
```

### 3.2.4.7 La zone de texte où l'on affiche les prix

```
JPanel panelP = new JPanel();
affPrix = new JTextField(10);
panelP.add(affPrix);
Box boxP = Box.createHorizontalBox();
boxP.add(new JLabel("Total à payer :"),BorderLayout.NORTH);
panelP.setMaximumSize(new Dimension(150,100));
boxP.add(panelP);
cp.add(boxP,BorderLayout.CENTER);
```

### 3.2.4.8 Le bouton pour quitter en bas

```
JButton quitter = new JButton("Quitter");
quitter.setActionCommand("quit");
```

Le bouton "quitter" permet de quitter l'application

```
quitter.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);}});
```

On met le bouton dans un JPanel pour qu'il garde sa taille préférée

```
JPanel panel = new JPanel();
panel.add(quitter);
```

on le met dans la fenêtre

```
cp.add(panel,BorderLayout.SOUTH);
```

#### 3.2.4.9 L'affichage final

on demande à la fenêtre d'afficher tous ses composants

```
    show();

  }fin du constructeur
```

### 3.2.5 Les procédures supplémentaires

#### 3.2.5.1 La procédure appelée par l'écouteur du bouton « prendre »

```
void fairePrendre(JTable table){

  row = table.getSelectedRow();
  numero de la ligne sélectionnée dans la table

  if ((row >= 0) && (data[row][0] != "")){
  si au moins une ligne est selectionnee

    data2[nbChoisie][0] = data[row][0];
    on copie la ligne sélectionnée de la table 1
    data2[nbChoisie][1] = data[row][1];
    dans la première ligne vide de la table 2

    nbChoisie++; on incremente le nombre de lignes non vides

    total = total + Integer.parseInt((String)data[row][1]);
    on ajoute au prix
    affPrix.setText(""+total);

    repaint(); on met a jour l'affichage
  }
}
```

#### 3.2.5.2 La procédure appelée par l'écouteur du bouton « retirer »

```
void faireRetirer(JTable table){

  row = table.getSelectedRow();
  numéro de la ligne sélectionnée dans la table

  if ((row >= 0) && (data2[row][0] != "")){
  si au moins une ligne est sélectionnée

    total = total - Integer.parseInt((String)data2[row][1]);
    on enleve au prix
    affPrix.setText(""+total);

    if (row >= 0){ on efface la ligne
      data2[row][0] = "";
      data2[row][1] = "";

    for(int i = row; i<=nbChoisie; i++){ on remonte les autres lignes
      data2[i][0] = data2[i+1][0];
      data2[i][1] = data2[i+1][1];
    }

    nbChoisie--; on décrémente le nombre de lignes non vides

    repaint(); on met a jour l'affichage
    }
```

```
    }
  }
```

### 3.2.6 La procédure « Main »

procédure principale qui appelle le constructeur ce qui à pour effet de lancer le programme

```
  public static void main(String[] args) {
    MacDo fenetre = new MacDo();
  }

}
```

## 3.3 Conlusions

### 3.3.1 Points Positifs

- swing est parfaitement portable.
- swing est indépendante du système hote.
- swing est très riche et permet de tout faire ou presque.
- Swing est bien plus riche que l'AWT (même si un composant n'existe pas en natif sur l'OS, rien n'empêche de complètement le redessiner)

### 3.3.2 Points Négatifs

- L'indépendance graphique et la portabilité ont un coût d'exécution important.
- swing nécessite une bonne connaissance des classes graphiques.
- SWING fut donc mis en place pour assurer 100% de portabilité (même un pixel doit avoir la même couleur). Le pari est gagné, mais à un coût non négligeable : pour assurer cette portabilité, un bouton (au tout autre composant graphique) est dessiné non plus par l'OS, mais par Java (ce qui en terme de temps d'exécution a un prix).
- En fait, SWING est présenté comme étant écrit uniquement en Java. Bien entendu, il y a une petite triche : en effet, Java doit au moins être capable d'interagir avec l'OS pour pouvoir tracer un point (il faut donc autre chose que du 100% Java). Or c'est ce que font les classes de base de l'AWT (telles que Component, Container, ...). En conséquence, tous les composants de SWING dérive d'une classe de l'AWT.

### 3.3.3 Conclusion

swing est une très bonne bibliothèque, complète et fonctionnelle.

## 3.4 Ressources

http://java.sun.com
http://www.ac-creteil.fr/util/programmation/java/cours_java/c-swing1.html
http://cermics.enpc.fr/cours/java/notes/notes3-swing.html

# 4 MFC

## 4.1 Présentation

### 4.1.1 Introduction

**MFC** = Microsoft Foundation Classes, sont des ensembles de classes de base permettant de programmer plus aisément sous Windows.
Les MFC constituent une librairie de classes dédiées à la mise en œuvre d'applications graphiques sur plate-forme Windows.

#### 4.1.1.1 Licence

MFC est sous licence **Windows** et elle est **payante**, cependant elle est livrée gratuitement avec Visual Studio.

#### 4.1.1.2 Encapsulation

Les classes MFC encapsulent un maximum de détails directement liés à la programmation Windows.
C'est une **surcouche** plus ou moins objet d'une partie de l'API Win32 développée par Microsoft.

#### 4.1.1.3 Plate-forme

Bibliothèque disponible **uniquement** sur plate-forme Windows

#### 4.1.1.4 Environnement de développement

L'environnement de développement **Visual C++** permet, via de nombreux outils, de simplifier sérieusement la mise en œuvre d'une application MFC.

L'API Windows n'est pas spécialement conçue pour Visual C++, ni pour le C++ de manière générale. Elle doit pouvoir être utilisée dans des programmes écrits dans une variété de langages, dont la plupart ne sont pas orientés objet.

Les MFC constituent un ensemble de classes prédéfinies autour desquelles s'articule la programmation Windows avec Visual C++ : l'écriture d'un programme Windows entraîne la création et l'utilisation d'objets MFC, ou d'objets de classes dérivées des MFC. Les objets créés contiennent des fonctions membres permettant la communication avec Windows, l'échange de messages entre eux, et le traitement des messages Windows.

Les MFC représentent un vaste domaine et font intervenir un grand nombre de classes. Elles composent une structure complète de développement d'applications, où il suffit d'effectuer les opérations de personnalisation dont on a besoin pour adapter les programmes selon les exigences.

Par convention, sachez que toutes :

- les classes des MFC portent des noms qui commencent par *C*, comme *CDocument* ou *CView* ;
- les données membres d'une classe MFC commencent par le préfixe *m_*.

Il va de soi, qu'il est préférable d'adopter cette convention lors de la définition de nos classes, ou de leur dérivation à partir des classes de la bibliothèque MFC.

### 4.1.2 Les ressources

Outre le code exécutable proprement dit, les applications Windows ont une icône qui s'affiche dans la barre des tâches, lorsque la fenêtre est réduite ou bien encore dans le dossier où elle est stockée.

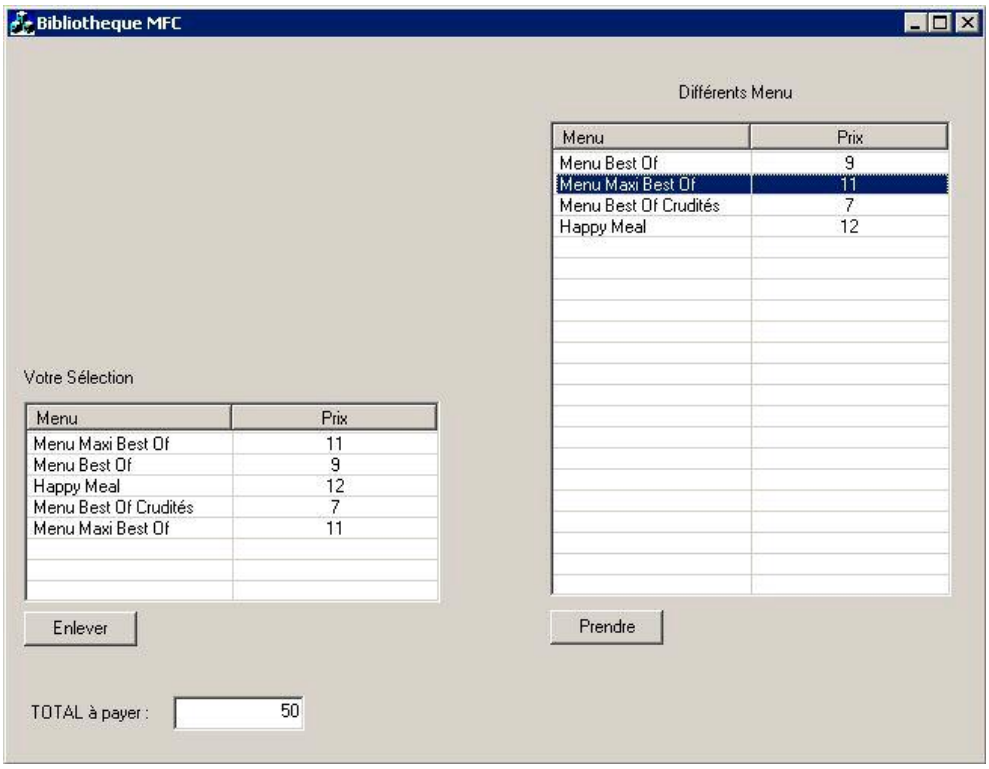
Les icônes, les curseurs, les menus et les boîtes de dialogue sont des "ressources".
Elles sont stockées dans le fichier `.exe` avec le code exécutable, mais elles sont traitées d'une façon différente. En particulier, elles restent sur le disque tant que l'application ne les appelle pas.

Pendant le développement, les ressources sont définies dans un *script de ressources*. C'est un fichier texte ASCII que l'on peut reconnaître grâce à l'extension `.rc`. Ce fichier est compilé par un *compilateur de ressources* qui ajoute à la fin de l'exécutable les ressources et crée une *table des ressources* dans l'en-tête du fichier.

## 4.2 Le Programme

### 4.2.1 Notre programme

Voici notre application écrite avec la bibliothèque MFC de Microsoft Windows




Réalisée à l'aide de Visual C++

Nous obtenons plusieurs fichiers générés par Visual C++ :

- te.h
- te.cpp
- resource.h
- stdafx.h
- stdafx.cpp
- tedlg.cpp
- te.rc

### 4.2.2 Description des fichiers

#### 4.2.2.1 StdAfx.h et StdAfx.cpp

Ces deux fichiers ne contiennent pas vraiment de code. Ils se contentent d'inclure les déclarations de fonctions globales très utiles. La raison pour laquelle on crée ces fichiers pour procéder à une inclusion de *header* plutôt que de les inclure directement dans le fichier source principal, est pour accélérer la vitesse de compilation.

##### 4.2.2.1.1 StdAfx.h

```
// Ce sont des commandes du préprocesseur, pour la
// compilation conditionnelle qui permettent de ne pas
// inclure plus d'une fois ce fichier dans l'exécutable.

#ifndef __STDAFX_H__
#define __STDAFX_H__

// Inclusions de librairies de fonctions globales MFC
//-------------------------------------------------------
#include <afxwin.h> // composants standards
#include <afxext.h> // MFC extensions

// Fin du bloc pour la compilation conditionnelle.
#endif
```

##### 4.2.2.1.2 StdAfx.cpp

```
// Inclusions du fichier .h
//-------------------------------------------------------
#include "StdAfx.h"
```

##### 4.2.2.1.3 Conclusion

Voilà c'est tout pour ces deux fichiers!
On inclut tout simplement les fichiers *header* **AfxWin.h** et **AfxExt.h** qui contiennent les déclarations de plusieurs fonctions globales des MFC. Par fonctions globales, vous pouvez comprendre que ce ne sont pas des fonctions membres appartenant à un objet, ce sont des fonctions pouvant être appelées à n'importe quel endroit de votre code.
Par exemple, une fonction des Afx se nomme **AfxMessageBox**. Cette fonction peut être appelée pour afficher une boîte de dialogue et n'attend aucun *handle* de fenêtre en paramètre. La fonction API **MessageBox** quant à elle, a absolument besoin qu'on lui transmette un *handle* de fenêtre (**hwnd**).

### 4.2.2.2 Fichiers TE.h et TE.cpp

Maintenant, nous allons créer les deux fichiers **TE**.
Habituellement en C++, un fichier créé correspond à une classe (son fichier d'en-tête et le code source de la classe).
La classe que nous allons créer est dérivée de la classe MFC **CWinApp**.
**CWinApp** est une classe de base de laquelle dérive toute application Windows qui utilise les MFC. Voici donc la déclaration de notre classe **CTEApp**:

#### 4.2.2.2.1 TE.h

```
// Déclaration de la classe CProjet1App
// La classe CProjet1App dérivé de la classe CWinApp
// des MFC //-----------------------------------------------------------


class CTEApp : public CWinApp
{
// Constructeur/Destructeur

public:
	CTEApp();

// Méthodes publics // InitInstance est une fonction membre surchargé.

public:
	virtual BOOL InitInstance();

// Déclaration de la tables des messages

	DECLARE_MESSAGE_MAP()
};
```

#### 4.2.2.2.2 TE.cpp

```
// Inclusions
#include "stdafx.h"
#include "TE.h"
#include "TEDlg.h"

// MFC Debugging Support
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// Liste des messages
BEGIN_MESSAGE_MAP(CTEApp, CWinApp)
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

// Variable Globale
CTEApp theApp;

// Constructeur/Destructeur de CTEApp
CTEApp::CTEApp()
{
}

// CTEApp initialization
// Méthode public
BOOL CTEApp::InitInstance()
{
	AfxEnableControlContainer();

	// Création de la fenêtre principale (fenêtre dialog)
	// Création d'un objet CmainDialog et conservation adresse
	// dans la donnée membre m_pMainWnd.

	CTEDlg dlg;
	m_pMainWnd = &dlg;
	int nResponse = dlg.DoModal();

	// Si la fenêtre « dialog » a été fermée alors return FALSE et le programme se termine
	return FALSE;
}
```

#### 4.2.2.2.3 Conclusion

Le fichier *.cpp* ne contient pas beaucoup de code, la seule section intéressante est la fonction membre **InitInstance**.

Cette fonction est surchargée, ce qui veux dire que c'est une fonction déjà existante dans la classe dérivée (**CWinApp** dans notre cas).

Cette fonction est dite virtuelle, dans la classe **CWinApp** elle a été déclarée à l'aide du mot clé **virtual**.
Une fonction virtuelle peut être surchargée, donc on peut y ajouter notre propre code.
**InitInstance** n'existe que pour donner à l'application l'opportunité de s'initialiser elle-même. C'est dans cette fonction que l'on écrira le code qui doit s'exécuter au démarrage de l'application.
En fait, la fonction **InitInstance**, si on ne la surcharge pas, ne contient par défaut que la commande **return TRUE;**.

La première des choses que l'on peut voir, c'est la création d'un objet **CmainDialog** dont nous allons écrire le code un peu plus loin. On conserve alors l'adresse dans la donnée membre public de **CWnd**.

### 4.2.2.3 Codage

### 4.2.2.4 Initialisation de l'interface (tedlg.cpp)

Maintenant c'est à nous de coder.
Les données qui servent à initialiser notre interface se trouve dans la class
`BOOL CTEDlg::OnInitDialog()`

#### 4.2.2.4.1 Initialisation du tableau des menu

La variable représentant ce tableau est `list1`

```
CListCtrl * list1 = (CListCtrl *) GetDlgItem (IDC_LIST1);
// Associe la variable liste à la resource IDC_LIST1

list1->SetExtendedStyle( LVS_REPORT | LVS_EX_FULLROWSELECT );
// Modifie le style du contrôle de manière à autoriser la sélection sur une ligne entière.

CRect rect;
list1->GetClientRect(&rect);
// Le CRect est utilisé pour la taille des colonnes.
// La classe CRect permet de stocker les coordonnées d'un rectangle. Elle permet aussi quelques fonctions "évoluée" comme calculer la hauteur ou la largeur du rectangle

list1->InsertColumn(0,_T("Menu"),LVCFMT_CENTER, rect.Width()/2 ,0);
list1->InsertColumn(1,_T("Prix"),LVCFMT_CENTER, rect.Width()/2 ,0);
// On initialise les titres des colonnes du tableau
```

Puis on remplit le tableau avec les différents menu :

```
list1->SetItemText(0, 0, "Menu Best Of");
list1->SetItemText(0, 1, "9");

list1->SetItemText(1, 0, "Menu Maxi Best Of");
list1->SetItemText(1, 1, "11");

list1->SetItemText(2, 0, "Menu Best Of Crudités");
list1->SetItemText(2, 1, "7");

list1->SetItemText(3, 0, "Happy Meal");
list1->SetItemText(3, 1, "12");
```

#### 4.2.2.4.2 Initialialisation du tableau de notre sélection (list2)

```
CListCtrl * list2 = (CListCtrl *) GetDlgItem (IDC_LIST4);
// Associe la variable liste à la resource IDC_LIST1

list2->SetExtendedStyle( LVS_REPORT | LVS_EX_FULLROWSELECT );
// Modifie le style du contrôle de manière à autoriser la sélection sur une ligne entière.

CRect rect2;
list2->GetClientRect(&rect2);
// Le CRect est utilisé pour la taille des colonnes.

list2->InsertColumn(0,_T("Menu"),LVCFMT_CENTER, rect2.Width()/2 ,0);
list2->InsertColumn(1,_T("Prix"),LVCFMT_CENTER, rect2.Width()/2 ,0);
```

### 4.2.2.5 Gestion des événements

Les seuls événements que l'on doit gérer sont :

- l'activation des boutons « prendre » et « enlever »
- les double-clicks sur les 2 tableaux (list1, list2)

#### 4.2.2.5.1 Bouton « prendre »

Cette action est gérée dans la class `void CTEDlg::OnButton1()`

Etudions son contenu :

La fonction GetDlgItem permet de récupérer l'adresse d'un objet dialogue. Elle reçoit en paramètre l'identifiant du contrôle. Notons que le casting de l'adresse renvoyée est obligatoire.

```
CListCtrl * edit  = (CListCtrl *) GetDlgItem (IDC_EDIT1);
CListCtrl * list2 = (CListCtrl *) GetDlgItem (IDC_LIST4);
CListCtrl * list  = (CListCtrl *) GetDlgItem (IDC_LIST1);
// Association Contrôle <-> resource.

int ItemFocus = list->GetNextItem(-1,LVNI_SELECTED);
// Parcours de la liste en commençant par son premier élément (-1),
jusqu'à en rencontrer un sélectionné (LVNI_SELECTED).

list2->InsertItem(0,"");
list2->SetItemText(0, 0, list->GetItemText(ItemFocus,0));
list2->SetItemText(0, 1, list->GetItemText(ItemFocus,1));

m_Total += (int)atof(list->GetItemText(ItemFocus,1));
// On met à jour le total de nos achats
UpdateData(FALSE);
```

## 4.3 Conclusions

### 4.3.1 Points Positifs

- MSDN n'est pas très facile à maîtriser, mais c'est certainement la plus grosse base de documentation sur un OS.

- Un avantage de ces classes est qu'il permet de développer une application assez rapidement grâce à un ensemble d'outils aidant à la mise en place du code général.

### 4.3.2 Points Négatifs

- MFC sont une sorte de surcouche objet permettant d'accéder à l'API Windows qui est en C. L'architecture n'est pas très objet. Il est nécessaire dans beaucoup de situations de passer une structure 'C' avec une quinzaine de champs (dont seulement un ou deux nous intéressent).

- L'éditeur de ressources de Visual Studio est fortement limité : il permet de placer des contrôles à l'écran, et c'est à peu près tout. Quelques propriétés sont disponibles via l'onglet 'properties' et le 'class wizard', ce qui permet de créer des variables et méthodes relativement facilement.

- Les MFC ne gèrent pas les réarrangements (layout) dans les fenêtres : cela pose d'énormes problèmes si on veut qu'une fenêtre soit de taille variable. Il faut repositionner soi-même tous les contrôles (cela explique pourquoi tant de dialogues sous Windows ont une taille fixe). Le problème devient plus grave quand on doit traduire une application, car beaucoup de langues ont des expressions plus longues que l'anglais. Vous devez retravailler vos fenêtres pour chaque langue.

- La documentation est un aspect primordial à considérer lorsqu'on veut utiliser une bibliothèque graphique riche en fonctionnalités. Celle de Visual, MSDN semble pléthorique, tient sur 10 CDROM. On y trouve des articles pour faire toutes sorte de choses. Cependant, elle laisse l'impression que ce qui est documenté provient d'une mauvaise architecture. La navigation à l'intérieur est de piètre qualité : impossible d'accéder facilement depuis une classe à ses classes mères ou filles, présentations des méthodes sans la signature, accès difficiles aux méthodes héritées, etc.

- Il est quasiment impossible de développer avec un autre outil que Visual Studio

- Les ressources correspondent à un certain nombre de #define dans le fichier 'Resource.h' (suivi d'une valeur inférieure à 32768). Outre le fait que ce n'est pas propre, cela peut poser des problèmes en cas d'effacement ou de renommage de ressources, ou des conflits si plusieurs DLL utilisent des fichiers 'resource.h' avec les mêmes noms de ressources et des valeurs différentes.

- Pour distribuer votre application MFC, vous dépendez de la présence de la DLL MFC42.dll sur le système d'exploitation cible. Mais pour le même nom, MFC42.dll, il y a en fait trois versions de cette DLL. Il est donc nécessaire de vérifier que l'utilisateur a la version correcte, et sinon, de la mettre à jour. La mise à jour va changer le comportement de nombreuses applications sur son système.

### 4.3.3 Conclusion

La bibliothèques MFC de Microsoft semble être en fin de vie.
Le fait que Microsoft ait décidé de les laisser complètement tomber en faveur de .NET vient corroborer notre point de vue.

## 4.4 Ressources

http://c.developpez.com/
http://www.visionx.com/mfcpro/
http://www.programmationworld.com/site/Cours.asp?Action=cours&Numero=205
http://www.francedev.com/cours/cours1.asp

# 5 QT

## 5.1 Présentation

### 5.1.1 Introduction

La librairie graphique Qt, développée par la société Trolltech est principalement connue dans le monde Linux pour être à la base de l'environnement graphique KDE.
QT est une librairie très puissante pour créer des applications graphiques.
Elle est la base de biens des applications libres sous UNIX.

Qt est une bibliothèque graphique écrite en C++, c'est-à-dire un ensemble de classes permettant de développer l'interface graphique d'un programme C++.

#### 5.1.1.1 Licence

- Sous UNIX, QT est sous licence GPL
- Pour Windows, il existe deux versions de QT: une version "non-commerciale" gratuite qui vous autorise à créer des logiciels libres pour Windows, et une version commerciale payante pour les applications à but lucratif.

#### 5.1.1.2 Multiplatforme

- MS/Windows
- Unix – Linux
- Macintosh - Mac OS X

#### 5.1.1.3 Langage

Langage C++

## 5.1.2 Fonctionnement de QT

### 5.1.2.1 Structure d'un programme QT

#### 5.1.2.1.1 Les « includes »

Toute application ***Qt*** est constituée d'un objet de type `QApplication` . Le fichier `qapp.h` contient les déclarations de la classe `QApplication` . On inclura dans les programmes utilisant ***Qt*** , le fichier de déclaration de la classe `QApplication` .

Mettre l'entête suivante dans votre programme principal:

```
#include <qapplication.h>
```

Le fichier `qapp.h` ne contient pas tout ce dont une application ***Qt*** a besoin. Il est souvent nécessaire d'inclure d'autres fichiers d'interface, selon les besoins de l'application que l'on développe. Par exemple, si l'on veut utiliser d'autres objets graphiques, un bouton par exemple, on devra inclure le fichier `qpushbutton.h` .

```
#include < qpushbutton.h>
```

#### 5.1.2.1.2 Le prototype de la fonction main

```
int main(int argc, char *argv[])
```

doit comporter ces deux arguments car QT en a besoin pour connaître d'éventuels arguments d'invocation que l'utilisateur pourrait passer à QT

#### 5.1.2.1.3 Créer une « Qapplication »

Une application ***Qt*** commence toujours par la création d'un objet de la classe `QApplication` de la manière suivante:

```
QApplication mon_appli(argc, argv );
```

Cet objet devra être créé avant la création des *widgets* .
Cette classe qui possède un constructeur de la forme

```
QApplication( int &argc, char **argv );
```

Les arguments `argc` et `argv` sont ceux de la fonction `main`.

### 5.1.2.2 Des Widgets personnalisés

Dans les applications, on a toujours besoin de définir ses propres widgets avec des caractéristiques particulières. Supposons que l'on veuille créer un widget qui contienne deux boutons; pour ce faire, nous allons créer une nouvelle sous-classe de la classe `QWidget` .

```
// *********** nwidget.h ************

#include < qapp.h>
#include < qpushbt.h>


class NouveauWidget : public QWidget
    {
    public:
        NouveauWidget( QWidget *pere=0, const char *nom=0 );
    };
```

Cette sous-classe ne contient rien d'autre qu'un unique constructeur. Toutes les autres méthodes sont celles dérivées de la classe `QWidget` .

Le premier argument de ce constructeur est le widget père; si c'est un widget racine (ou principal), la valeur 0 (le pointeur `NULL`) sera fournit en argument.

Le deuxième argument de ce constructeur est le nom du widget. Ce n'est pas la chaîne figurant sur le widget mais un nom permettant de rechercher un widget particulier parmi l'ensemble des widget d'une application donnée.

```
// *********** nwidget.cpp ************


NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name ) {
  QPushButton *go = new QPushButton( "Coucou !!!", this, "go" );
  QPushButton *quit = new QPushButton( "Quit", this, "quit" );
  }


// *********** main.c ************

#include < qapp.h>


int main( int argc, char **argv ) {
  QApplication a( argc, argv );


NouveauWidget w;
  w.setGeometry( 100, 100, 200, 120 );
  a.setMainWidget( &w );
  w.show();
  return a.exec();
}
```

Si l'on exécute ce programme, nous aurons la mauvaise surprise de ne voir qu'un seul bouton !!!
Pas de panique, les deux boutons sont bien là, mais ils sont superposés.
Pour éviter ce problème, nous allons disposer les boutons de manière à les visualiser tous les deux. Il faut donc préciser les positions exactes des boutons dans la fenêtre, leurs coordonnées sont relatives au widget qui les contient.

```
NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name ) {
  QPushButton *go = new QPushButton( "Coucou !", this, "go" );
  go->setGeometry( 10, 10, 60, 30 );
  QPushButton *quit = new QPushButton( "Quit", this, "quit" );
  quit->setGeometry( 80, 10, 60, 30 );
}
```

### 5.1.2.3 Exemple

Pour vous donner une idée d'un programme C++ utilisant Qt, nous vous proposons un exemple très simple permettant d'afficher une boîte de dialogue

```
// exemple.cpp


#include <qapplication.h>
#include <qmessagebox.h>
int main(int argc,char* argv[ ])
{

   QApplication a(argc,argv);
   QMessageBox boite;
   boite.information(0,"BOITE A MESSAGE","Bonjour ! ! ! ");

}
```

Pour compiler ce programme :
Vérifiez que vous avez la configuration requise, pour cela, reportez-vous au I.
Pour compiler ce programme, tapez la ligne suivante dans le répertoire le contenant :

```
g++ -I$QTDIR/include/ -L$QTDIR/lib/ exemple exemple.cpp -lqt
```

Vous devez obtenir une fenêtre semblable à celle ci-dessous.

### 5.1.2.4 Gérer les événements

Sous Qt, il existe un système de callback basé sur l'envoi de signaux et leur réception dans des slots qui permet de communiquer entre des classes. Ce système est très souple (on dit explicitement à l'objet d'écouter un signal donné).

#### 5.1.2.4.1 Slots et signaux

La méthode `connect` est la méthode la plus importante de ***Qt*** .
C'est une méthode `static` de la classe `QObject`

```
QObject::connect( &bouton, SIGNAL(clicked()), &a, SLOT(quit()) );
```

Cette méthode établit une communication unidirectionnelle entre deux objets ***Qt*** .
A chaque objet ***Qt*** (objet de la classe `QObject` ou d'une classe dérivée), on peut associer des *signaux* qui permettent d'envoyer des messages et des *slots* pour recevoir des messages.

Notons que la classe `QWidget` est une classe dérivée de la classe `QObject` (comme d'ailleurs toutes les classes ***Qt*** ).

La sémantique de cette instruction est : le signal <<cliquer>> effectué sur le widget `bouton` est relié au slot `quit()` de l'application `a` de telle sorte que l'application s'arrête lorsque on clique sur le bouton.

le système de communication établi en ***Qt*** est basé sur cette notion d'émission et réception de signaux.
L'objet émetteur du signal n'a pas à se soucier de l'existence d'un objet susceptible de recevoir le signal émis.
Ce qui permet une très bonne encapsulation et le développement totalement modulaire.
Un signal peut être connecté à plusieurs slots et plusieurs siganux à un même slot.

Voici un exemple d'utilisation de slots et de siganux qui est totalement indépendant des applications graphiques.

Considérons deux classes `objet1` et `objet2` définies comme suit:

```
class objet1 {
public:
  objet1() { val = -1; };
  int valeur(void) { return val; }
  void mettre_a_jour(int i) { val = i; }
private:
  int val;
};


class objet2 {
public:
  objet2() { val = -1; };
  int valeur(void) { return val; }
  void affecter(int i) { val = i; }
private:
  int val;
};
```

Pour pouvoir établir la communication entre des objets de ces deux classes, il nous faut modifier légèrement la définiton de ces classes. Nous allons faire en sorte qu'une modification du champ `val` d'un objet de la classe `objet2` entraîne automatiquement la modification du champ `val` d'un objet de la classe `objet1`. Le programme principal pourrait être quelque chose du genre:

```
#include "objet1.h"
#include "objet2.h"

main() {
  objet1 a;
  objet2 b;

  QObject::connect(&b,
                   SIGNAL(valeurModifiee(int)),
                   &a,
                   SLOT(mettre_a_jour(int)));

  b.affecter(200);
  cout << a.valeur() << "   " << b.valeur() << endl;
}
```

C'est l'instruction

```
QObject::connect(&b, SIGNAL(valeurModifiee(int)),
                 &a, SLOT(mettre_a_jour(int)));
```

qui permet de modifier le champ `val` à travers la méthode `mettre_a_jour(int)` lorsque l'objet `b` émet le signal `valeurModifiee`.

#### 5.1.2.4.2 Emetteur du signal

Pour que tout cela fonctionne, il faut modifier la classe `objet2`.

- préciser que cette classe est une classe dérivée de la classe `QObject` :

```
// ***********  objet2.h  *************
  class objet2  : public QObject {
```

- toute classe utilisant les signaux ou les slots doit contenir la déclaration `Q_OBJECT`:

```
  Q_OBJECT;
```

- préciser les signaux que les objets de la classe `objet2` est susceptible d'émettre; il s'agit ici du signal `valueModifiee`.

```
  signals:
  void valeurModifiee(int);
```

- émettre le signal `valueModifiee` chaque fois que le champ `val` d'un objet de cette classe est modifiée:

```
public:
  void affecter(int i) {
    if (val != i) {
      val = i;
      emit valeurModifee(i);
    }
  }
```

- compléter la classe avec ses méthodes et champs habiutels:

```
// ***********  objet2.h  *************
#include < qapp.h>


class objet2  : public QObject {
  Q_OBJECT;
signals:
  void valeurModifiee(int);
public:
  void affecter(int i) {
    if (val != i) {
      val = i;
      emit valeurModifiee(i);
    }
  }
  objet2() { val = -1; };
  int valeur(void) { return val; }
private:
  int val;
}
```

#### 5.1.2.4.3 Récepteur du signal

Pour que l'objet `a` de la classe `objet1` réagissent au signal émis par l'objet `b` de la classe `objet2`, il faut également modifé la classe `objet1`

- préciser que cette classe est une classe dérivée de la classe `QObject` :

```
// ***********  objet1.h  *************
class objet1  : public QObject {
```

- Comme pour la classe `objet2`, la classe `objet1` doit contenir la déclaration `Q_OBJECT;`:

```
  Q_OBJECT;
```

- définir le ou les slots (les méthodes qui seront exécutées lors de l'émission d'un signal); ici, il s'agit de la méthode `mettre_a_jour`.

```
public slots:
  void mettre_a_jour(int i) { val = i; }
```

- compléter la classe avec ses méthodes et champs habiutels:

```
// ***********  objet1.h  *************
#include < qapp.h>


class objet1  : public QObject {
  Q_OBJECT;
public slots:
  void mettre_a_jour(int i) { val = i; }
public:
  objet1() { val = -1; };
  int valeur(void) { return val; }
private:
  int val;
};
```

### 5.1.2.5 Utilitaire moc

Une fois déclarées les classes `objet1` et `objet2`, on utilise l'utilitaire `moc` pour produire un ou plusieurs fichiers qui seront inclus pour produire l'exécutable final. Dans notre exemple, si `objet1` et `objet2` sont déclarés dans les fichiers `objet1.h` et `objet2.h`, on génèrera deux fichiers `mocobjet1.cpp` et `mocobjet2.cpp`.

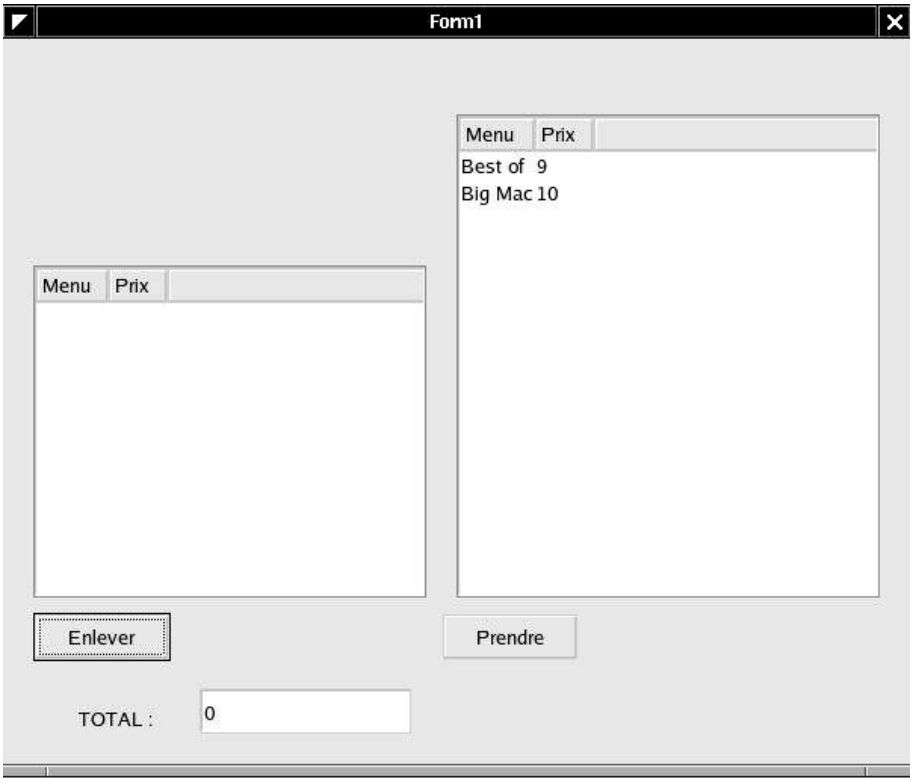
```
moc objet1.h -o mocobjet1.cpp
moc objet2.h -o mocobjet2.cpp
```

L'exécutable final s'obtient en compilant les fichiers `mocobjet1.cpp`, `mocobjet1.cpp` et `main.cpp`

```
g++ -o signal mocobjet1.cpp  mocobjet1.cpp main.cpp -lqt
```

Ce sont les fichiers `mocobjet1.cpp  mocobjet1.cpp` qui mettent en place la communication entre les siganux et les slots.

## 5.2 Notre Programme




### 5.2.1 Déclaration de notre Class Fenêtre

```
class Form1 : public QDialog
{
    Q_OBJECT

public:
    Form1( QWidget* parent = 0, const char* name = 0, bool modal = FALSE,
WFlags fl = 0 );
    ~Form1();

// Les widgets dont on a besoin
    QPushButton* enleve;
    QPushButton* prendre;
    QListView* selection;
    QListView* menu;
    QLabel* TextLabel1;
    QLineEdit* total;

public slots: // Déclaration des slots
    virtual void PushButton1_pressed();
    virtual void PushButton2_pressed();
    virtual void menu_doubleClicked( QListViewItem * );

};
```

### 5.2.2 Création de notre Class Fenêtre

Notre fenêtre principale est une boite de dialogue

```
Form1::Form1( QWidget* parent,  const char* name, bool modal, WFlags fl )
    : QDialog( parent, name, modal, fl )
{
      if ( !name )
      setName( "Form1" );
      resize( 600, 480 );
      setCaption( trUtf8( "Form1" ) );
```

#### 5.2.2.1 Notes sur l'internationalisation `trUtf8()`

L'internationalisation, souvent abrégée i18n, consiste au support des différentes langues ou paramètres locaux (unités, symboles, dates etc.) La traduction représente une grosse partie de l'internationalisation. Comme vous l'avez vu, notre application est en langue anglaise. Cependant les libellés utilisent la macro trUtf8 comme dans cet exemple :

```
menubar->insertItem(trUtf8("&File"),popup);
```

#### 5.2.2.2 Placement de tous les widgets

##### 5.2.2.2.1 Les boutons

```
// placement du bouton « enleve »
enleve = new QPushButton( this, "enleve" );
enleve->setGeometry( QRect( 20, 380, 91, 32 ) );
enleve->setText( trUtf8( "Enlever" ) );

// placement du bouton « prendre »
prendre = new QPushButton( this, "prendre" );
prendre->setGeometry( QRect( 290, 380, 91, 32 ) );
prendre->setText( trUtf8( "Prendre" ) );
```

##### 5.2.2.2.2 Les tableaux

```
// Notre tableau contenant notre choix du repas
selection = new QListView( this, "selection" );
selection->addColumn( trUtf8( "Menu" ) );
selection->addColumn( trUtf8( "Prix" ) );
selection->setGeometry( QRect( 20, 150, 260, 220 ) );
selection->setMinimumSize( QSize( 0, 0 ) );

// Notre tableau contenant tous les menu
menu = new QListView( this, "menu" );
menu->addColumn( trUtf8( "Menu" ) );
menu->addColumn( trUtf8( "Prix" ) );
QListViewItem * item = new QListViewItem( menu, 0 );
item->setText( 0, trUtf8( "Best of" ) );
item->setText( 1, trUtf8( "9" ) );
```

```
// Remplissage du tableau des menu
item = new QListViewItem( menu, item );
item->setText( 0, trUtf8( "Big Mac" ) );
item->setText( 1, trUtf8( "10" ) );

menu->setGeometry( QRect( 300, 50, 280, 320 ) );
```

#### 5.2.2.2.3 Affichage du total

```
// Un label pour préciser le total
TextLabel1 = new QLabel( this, "TextLabel1" );
TextLabel1->setGeometry( QRect( 50, 440, 68, 20 ) );
TextLabel1->setText( trUtf8( "TOTAL :" ) );

// Pour afficher le montant global à payer
total = new QLineEdit( this, "total" );
total->setGeometry( QRect( 130, 430, 140, 30 ) );
total->setText( trUtf8( "0" ) );
```

#### 5.2.2.2.4 Les connections signals et slots

```
// signals and slots connections
connect(prendre, SIGNAL( pressed()), this, SLOT( PushButton1_pressed() ) );
connect(enleve,  SIGNAL( pressed()), this, SLOT( PushButton2_pressed() ) );
connect( menu, SIGNAL( doubleClicked(QListViewItem*) ), this, SLOT(
menu_doubleClicked(QListViewItem*) ) );
}
```

### 5.2.3 Implémentation des actions

```
// Pour prendre un menu
void Form1::PushButton1_pressed()
{

    QListViewItem *tmp = menu->currentItem();
    selection->insertItem(tmp);

    char namebuf[13];
    int ancien_prix = atoi(total->text());
    int nouveau_prix = ancien_prix + atoi(tmp->text(1));

    sprintf( namebuf, "%d", nouveau_prix);
    printf("%s\n",namebuf);
    total->setText(namebuf);
}

// Pour enlever un menu de notre liste
void Form1::PushButton2_pressed()
{
    printf("button enlever\n");
    QListViewItem *tmp = menu->currentItem();
    selection->takeItem(tmp);
}

// Lorsqu'on double-click sur un menu on le prend
```

```
void Form1::menu_doubleClicked( QListViewItem * )
{
    QListViewItem *tmp = menu->currentItem();
    selection->insertItem(tmp);

    char namebuf[13];
    int ancien_prix = atoi(total->text());
    int nouveau_prix = ancien_prix + atoi(tmp->text(1));

    sprintf( namebuf, "%d", nouveau_prix);
    printf("%s\n",namebuf);
    total->setText(namebuf);
}
```

### 5.2.4 Programme principal

```
int main( int argc, char ** argv )
{
    QApplication a( argc, argv );
    Form1 *w = new Form1;
    w->show();
    a.connect( &a, SIGNAL( lastWindowClosed() ), &a, SLOT( quit() ) );
    return a.exec();
}
```

## 5.3 Conclusion

### 5.3.1 Points positifs

- Qt est basé sur une architecture purement objet. Il en résulte une bibliothèque extrêmement cohérente à tous points de vue (nommage, objets, héritage...), complète (classes proposées, méthodes des classes) tout en restant simple d'utilisation.

- Sous Qt, si vous avez un champ d'édition (QlineEdit), vous le créez avec l'opérateur new, comme n'importe quelle classe et vous pouvez appeler toutes ses méthodes n'importe quand à partir du moment où l'objet existe. Manipuler cet objet revient exactement à manipuler l'objet affiché. Il n'y a rien à connaitre, cela fonctionne de la façon la plus simple qu'on peut imaginer.

- Sous Qt, tout peut être codé à la main, parce que le code est simple. Pour créer un bouton, il suffit de faire:
  ```
  button = new QPushButton( "texte de mon bouton", MaFenetre );
  ```
  Pour appeler une méthode 'action()' si le bouton est clické, écrivez :
  ```
  connect( button, SIGNAL( clicked() ), SLOT( action() ) );
  ```

- Il existe de plus un outil, 'Qt Designer', qui permet une programmation proche d'un RAD (vous pouvez associer facilement du code aux actions, l'éditeur de properties est complet, associer des signaux et des slots, etc...). Il gère le réarrangement automatique et génère du code lisible et compréhensible. Le code est généré dans un fichier séparé, de sorte que vous pouvez modifier votre interface alors que vous l'utilisez déjà dans votre code.

- La documentation de Qt est excellente : doc.trolltech.com
  Elle est à la fois copieuse et complète, puisqu'elle couvre tout Qt.
  Toutes les classes et toutes les méthodes sont documentées correctement, avec de nombreux détails et des exemples.

- Qt est gratuit et libre pour sa version Unix (disponible sous licence GPL). Une version gratuite est disponible sous Windows pour utilisation non commerciale. En revanche, pour du développement commercial classique, à sources fermées, vous devez acheter une licence. Cette licence est valable pour une plate-forme, pour un seul développeur, à vie.

### 5.3.2 Points négatifs

- Sous Windows, pour du développement commercial classique, à sources fermées, vous devez acheter une licence.

## 5.4 Ressources

http://www.linuxfrench.net/article.php3?id_article=1083
http://phil.freehackers.org/kde/fr_qt-vs-mfc.html
http://www.linuxfrench.net/article.php3?id_article=1089
http://www.linuxfrench.net/article.php?id_article=535
http://www.esil.univ-mrs.fr/~tourai/Qt/node3.html
http://mapc77.epfl.ch/gnu-generation/CD/dveloppement/qt/
http://www.univ-pau.fr/~artouste/fr/composants/Qt/ManuelutilisateurQt.htm#_Toc478479348
http://www.esil.univ-mrs.fr/~tourai/Qt/Qt.html
http://www.iro.umontreal.ca/~buisteri/info/tips/qtwin.html
http://doc.trolltech.com/3.1/index.html

# 6 Conclusions

## 6.1 Tableau récapitulatif

| Librairie | Langage(s) | Plateforme | Licence |
|---|---|---|---|
| GTK  | C, C++, Ada, Perl, Python, Eiffel | Unix, Windows, BeOS | LGPL |
| MFC  | C++ | Windows | Payant |
| SWING  | JAVA | Unix, Windows, MacOS X | |
| QT  | C++ | Unix, Windows, MacOS X | GPL |

## 6.2 Bilan

Nous avons deux grand vainqueurs :

- GTK est une très bonne API car assez simple dans son codage, mais aussi elle est à la foi multi-plateforme et disponible pour plusieurs langages de programmation.
- QT, par son API purement objet en fait une très bonne bibliothèque