



# PROJET TYPE FLTK

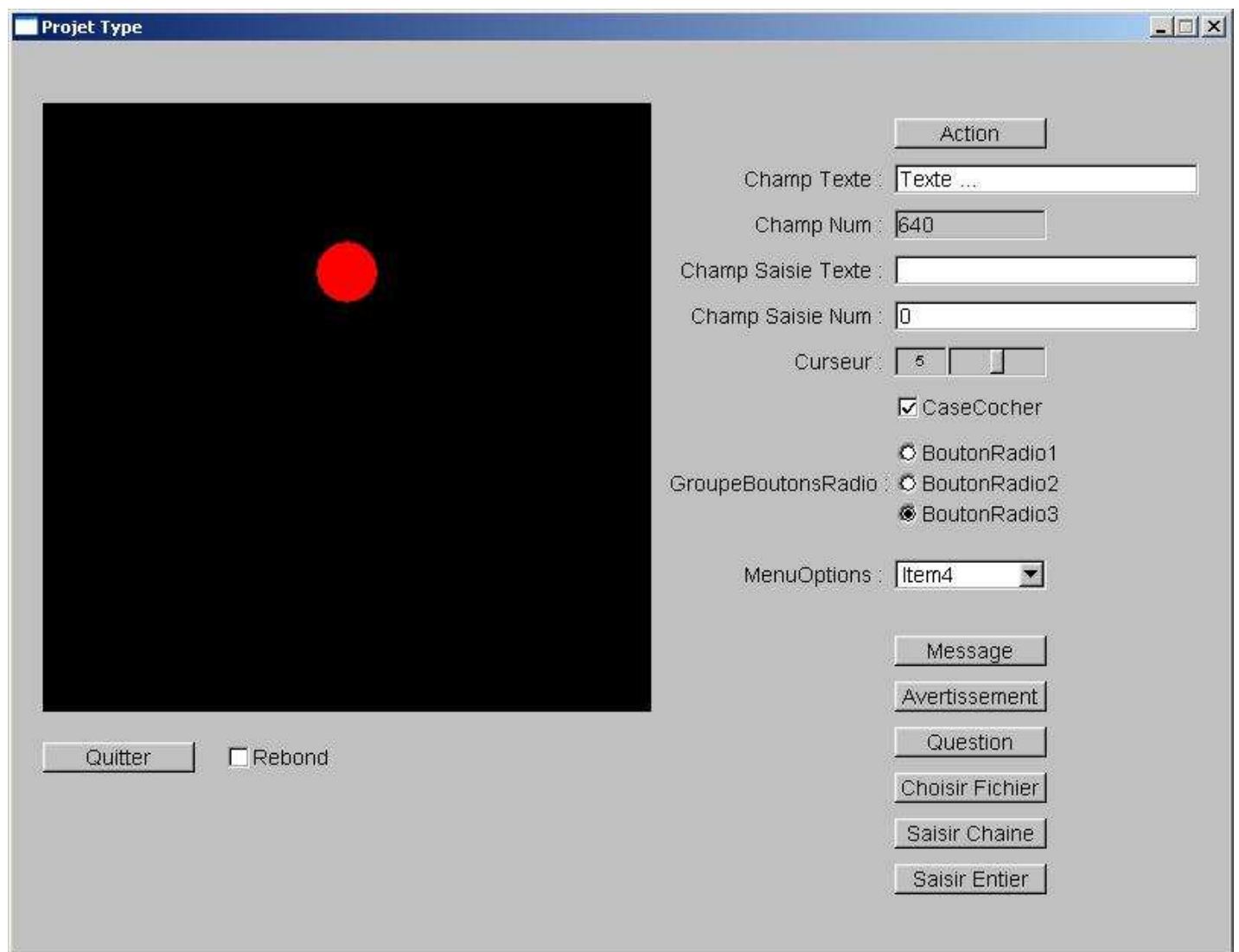
<b>PROJET TYPE FENÊTRE PRINCIPALE</b>	<b>1</b>
<b>C++ RAPPEL SUR LA NOTION DE STRUCTURE</b>	<b>2</b>
<b>C++ CLASSE, MÉTHODE, OBJET</b>	<b>3</b>
<b>C++ CE QU'IL FAUT RETENIR POUR FLTK</b>	<b>4</b>
<b>FLTK REPÈRE, FENÊTRE, BOUTON</b>	<b>5</b>
<b>FLTK BOUTON</b>	<b>6</b>
<b>FLTK CHAMPS D'AFFICHAGE</b>	<b>7</b>
<b>FLTK CHAMP DE SAISIE TEXTE</b>	<b>8</b>
<b>FLTK CHAMP DE SAISIE NUMÉRIQUE</b>	<b>9</b>
<b>FLTK CURSEUR DE SAISIE</b>	<b>10</b>
<b>FLTK CASE À COCHER</b>	<b>11</b>
<b>FLTK BOUTONS RADIO</b>	<b>12</b>
<b>FLTK MENU D'OPTIONS</b>	<b>13</b>
<b>FLTK DIALOGUES MESSAGE, AVERTISSEMENT</b>	<b>14</b>
<b>FLTK DIALOGUE QUESTION</b>	<b>15</b>
<b>FLTK DIALOGUE DE CHOIX D'UN FICHIER</b>	<b>16</b>
<b>FLTK</b>	

<b>DIALOGUE DE SAISIE TEXTE</b>	<b>17</b>
<b>FLTK DIALOGUE DE SAISIE NUMÉRIQUE</b>	<b>18</b>
<b>FLTK MÉTHODES UTILES</b>	<b>19</b>
<b>FLTK ZONE DE DESSIN</b>	<b>20</b>
<b>FLTK FONCTIONS DE DESSIN</b>	<b>21</b>
<b>FLTK TRAITEMENT CYCLIQUE</b>	<b>22</b>
<b>PROJET TYPE NOTION D'UNITÉ</b>	<b>23</b>
<b>PROJET TYPE UNITÉS</b>	<b>24</b>
<b>PROJET TYPE STRUCTURES DE DONNÉES</b>	<b>25</b>
<b>PROJET TYPE PRINCIPES</b>	<b>26</b>

# Projet Type

## *Fenêtre Principale*

---



# C++

## *Rappel sur la notion de structure*

---

Fltk ( Fast Light Toolkit ) est une librairie graphique écrite en C++ et qui utilise donc les concepts de la programmation orienté objet.

Qu'est ce que la notion d'objet, de classe d'objets ?

En C on peut créer des variables structurées (Cf. cours C05 ), c'est à dire des variables comportant plusieurs informations ( champs ), éventuellement de natures différentes :

```
// Structure Article
struct Article
{
    int Code ;
    float Prix ;
}

// Procédure d'initialisation d'un article
void InitialiserArticle( Article* Art, int UnCode, float UnPrix)
{
    Art->Code = UnCode ;
    Art->Prix = UnPrix ;
}

// Procédure d'affichage d'un article
void AfficherArticle( Article Art )
{
    printf( "Code = %d Prix = %f\n », Art.Code, Art.Prix ) ;
}

// Programme principal
void main()
{
    // Définition d'un article
    Article Art1 ;

    // Initialisation de l'article
    InitialiserArticle( &Art1, 1000, 53.25 ) ;

    // Affichage de l'article
    AfficherArticle( Art1 ) ;
}
```

# C++

## Classe, méthode, objet

---

Une **classe** C++ est une structure C, dans laquelle on a incorporé des sous-programmes ( qu'on appelle **méthodes** ) qui permettent de demander à un **objet** de cette classe ( une variable de ce type) de réaliser un traitement.

```
// Classe Article
class Article
{
    int Code ;
    float Prix ;

    Article( int UnCode, float UnPrix ) ;
    void Afficher() ;
} ;

// Méthode d'initialisation d'un article ( appelée automatiquement par new )
Article::Article( int UnCode, float UnPrix )
{
    Code = UnCode ;
    Prix = UnPrix ;
}

// Méthode d'affichage d'un article
void Article::Afficher()
{
    printf( "Code = %d Prix = %f\n », Code, Prix ) ;
    // Les valeurs de Code et Prix affichées seront celles de l'objet
    // sur laquelle la méthode sera appelée ( * )
}

// Programme principal
void main()
{
    // Définition d'un article
    Article* Art1 ;
    // Création et initialisation de l'article
    Art1 = new Article( 1000, 53.25 ) ;

    // Affichage de l'article
    Art1->Afficher() ; // on demande à Art1 de s'afficher ( * )
}
```

# C++

## *Ce qu'il faut retenir pour Fltk*

---

Pour utiliser les objets graphiques de Fltk, il suffit de retenir ce qui suit :

- Une objet graphique se manipule au travers d'une variable globale pointeur :

```
Classe* gObjet ;
```

- On crée l'objet avec l'opérateur **new** en lui transmettant ses caractéristiques ( fonction du type d'objet ) :

```
gObjet = new Classe( arg1, arg2, ... ) ;
```

- On invoque les méthodes d'un objet avec l'opérateur **->** :

```
gObjet->Methode( arg1, arg2, ... ) ;
```

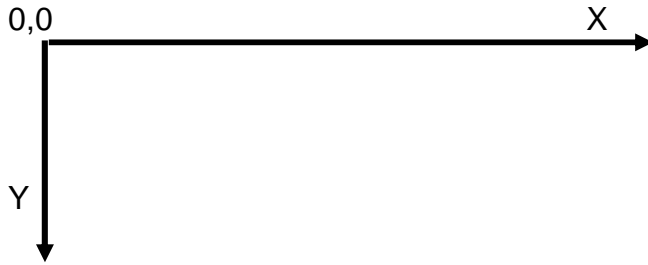
# Fltk

## *Repère, Fenêtre, Bouton*

---

### Repère graphique

Le repère est le suivant dans Fltk :



### Fenêtre

Un logiciel peut être constitué de plusieurs fenêtres, nous nous limitons à des applications mono-fenêtre dans le cadre des projets.

```
Fl_Double_Window* Fenetre ;  
...  
gInterface.Fenetre = new Fl_Double_Window( Largeur, Hauteur ) ;
```

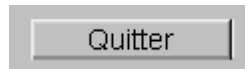
# Fltk

## Bouton

---

### Bouton

Un bouton est un objet permettant à l'utilisateur de déclencher une action :



```
Fl_Button* BoutonQuitter ;  
...  
gInterface.BoutonQuitter = new Fl_Button( X, Y, L, H, Libelle ) ;  
gInterface.BoutonQuitter->callback( BoutonQuitterCB, NULL ) ;
```

La méthode 'callback' d'un bouton permet de spécifier la fonction qui sera appelée automatiquement quand on clique dessus ( c'est ce qu'on appelle une fonction callback, ou tout simplement une callback, on les suffixe par CB pour les différencier des autres sous-programmes) :

```
void BoutonQuitterCB( Fl_Widget* w, void* data )  
{  
    // Fin du programme  
    exit(0) ;  
}
```

Remarque : Ignorez les arguments w et data.



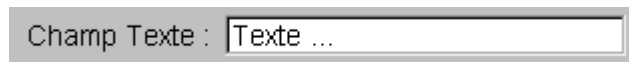
# Fltk

## Champs d'affichage

---

### Champ d'affichage texte

Un champ d'affichage texte permet d'afficher une chaîne de caractères :



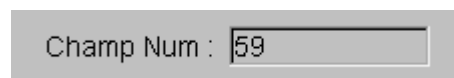
```
Fl_Output* ChampTxt ;  
...  
gInterface.ChampTxt = new Fl_Output( X, Y, L, H, Libelle ) ;
```

La méthode 'value' permet de modifier la valeur affichée :

```
gInterface.ChampTxt->value( "Texte ..." ) ;
```

### Champ d'affichage numérique

Un champ d'affichage numérique permet d'afficher une valeur numérique :



```
Fl_Value_Output* ChampNum ;  
...  
gInterface.ChampNum = new Fl_Value_Output( X, Y, L, H, Libelle ) ;
```

La méthode 'value' permet de modifier la valeur affichée :

```
gInterface.ChampNum->value( 59 ) ;
```

Cet objet gère indifféremment des valeurs entières ou flottantes.

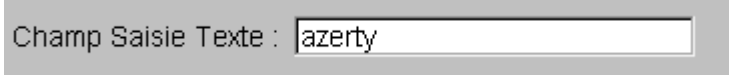
# Fltk

## *Champ de saisie texte*

---

### Champ de saisie texte

Un champ de saisie texte est une zone dans laquelle on peut entrer une chaîne de caractères :



```
Fl_Input* ChampSaisieTxt ;  
...  
gInterface.ChampSaisieTxt = new Fl_Input( X, Y, L, H, Libelle ) ;  
gInterface.ChampSaisieTxt->when( FL_WHEN_ENTER_KEY | FL_WHEN_NOT_CHANGED ) ;  
gInterface.ChampSaisieTxt->callback( ChampSaisieTxtCB, NULL ) ;
```

On lui associe généralement une callback qui est appelée chaque fois que l'utilisateur entre une valeur. La méthode 'value' permet de récupérer la valeur saisie :

```
void ChampSaisieTxtCB( Fl_Widget* w, void* data )  
{  
    char Saisie[80] ;  
    strcpy( Saisie, gInterface.ChampSaisieTxt->value() ) ;  
    ...  
}
```

La méthode 'value' permet également de changer la valeur affichée. Par exemple pour effacer la valeur précédemment saisie :

```
gInterface.ChampSaisieTxt->value( "" ) ;
```

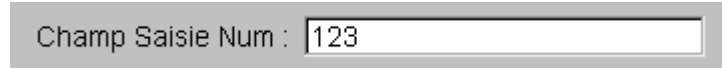
# Fltk

## *Champ de saisie numérique*

---

### Champ de saisie numérique

Un champ de saisie numérique est une zone dans laquelle on peut entrer une valeur numérique :



```
Fl_Value_Input* ChampSaisieNum ;  
...  
gInterface.ChampSaisieNum = new Fl_Value_Input( X, Y, L, H, Libelle ) ;  
gInterface.ChampSaisieNum->when( FL_WHEN_ENTER_KEY | FL_WHEN_NOT_CHANGED ) ;  
gInterface.ChampSaisieNum->callback( ChampSaisieTxtCB, NULL ) ;
```

On lui associe généralement une callback qui est appelée chaque fois que l'utilisateur entre une valeur, la méthode 'value' permet de récupérer la valeur saisie :

```
void ChampSaisieNumCB( Fl_Widget* w, void* data )  
{  
    int Valeur ;  
    Valeur = gInterface.ChampSaisieNum->value() ;  
    ...  
}
```

La méthode 'value' permet également de changer la valeur affichée, pour par exemple la réinitialiser :

```
gInterface.ChampSaisieNum->value( 0 ) ;
```

Cette objet gère indifféremment des valeurs entières ou flottantes ( tout dépend du type de variable, de valeur utilisé ).

# Fltk

## Curseur de saisie

---

### Curseur

Un curseur est un objet permettant de choisir une valeur numérique bornée :



```
Fl_Value_Slider* Curseur ;  
...  
gInterface.Curseur = new Fl_Value_Slider( X, Y, L, H, Libellé ) ;  
gInterface.Curseur->type( FL_HORIZONTAL ) ;  
gInterface.Curseur->align( FL_ALIGN_LEFT ) ;  
gInterface.Curseur->callback( CurseurCB, NULL ) ;
```

On définit les bornes avec la méthode 'bounds' :

```
gInterface.Curseur->bounds( Minimum, Maximum ) ;
```

On lui associe généralement une callback pour mémoriser dans une variable globale la valeur choisie, la méthode 'value' permet de récupérer cette valeur :

```
void CurseurCB ( Fl_Widget* w, void* data )  
{  
    gDonnees.Parametre = gInterface.Curseur->value() ;  
    ...  
}
```

La méthode 'value' permet également de changer la valeur affichée, pour par exemple la réinitialiser à 0 :

```
gInterface.Curseur->value( 0 ) ;
```

Par défaut cet objet gère une valeur flottante à 2 décimales, on peut choisir une autre précision, ou choisir de gérer une valeur entière ( precision == 0 ) en le précisant après avoir créé l'objet :

```
gInterface.Curseur->precision( 0 ) ;
```

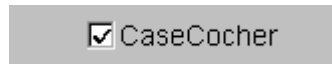
# Fltk

## Case à cocher

---

### Case à cocher

Une case à cocher est un objet permettant à l'utilisateur d'activer / de désactiver une option :



```
Fl_Check_Button* CaseCocher ;  
...  
gInterface.CaseCocher = new Fl_Check_Button( X, Y, L, H, Libellé ) ;  
gInterface.CaseCocher->callback( CaseCocherCB, NULL ) ;
```

On lui associe généralement une callback pour mémoriser dans une variable globale l'état de l'option, la méthode 'value' renvoie 1 si l'option est cochée, 0 sinon :

```
void CaseCocherCB( Fl_Widget* w, void* data )  
{  
    gDonnees.Option1 = gInterface.CaseCocher->value() ;  
    ...  
}
```

La méthode 'value' permet également d'initialiser l'état de la case à cocher :

```
gInterface.CaseCocher->value( 0 ) ;
```

# Fltk

## Boutons radio

---

### Groupe de boutons radio

Un groupe de boutons radio permet de faire un choix exclusif parmi plusieurs options :



```
gInterface.GroupeBoutonsRadios = new Fl_Group( X, Y, L, H, Libellé ) ;
gInterface.GroupeBoutonsRadios->begin() ;

gInterface.BoutonRadio1 = new Fl_Round_Button( 580, 260, 100, 20, "BoutonRadio1" ) ;
gInterface.BoutonRadio1->type(FL_RADIO_BUTTON) ;
gInterface.BoutonRadio1->callback(BoutonRadioCB, NULL ) ;
gInterface.BoutonRadio2 = new Fl_Round_Button( 580, 280, 100, 20, "BoutonRadio2" ) ;
gInterface.BoutonRadio2->type(FL_RADIO_BUTTON) ;
gInterface.BoutonRadio2 ->callback(BoutonRadioCB, NULL ) ;
gInterface.BoutonRadio3 = new Fl_Round_Button( 580, 300, 100, 20, "BoutonRadio3" ) ;
gInterface.BoutonRadio3->type(FL_RADIO_BUTTON) ;
gInterface.BoutonRadio3 ->callback(BoutonRadioCB, NULL ) ;

gInterface.GroupeBoutonsRadios->end() ;
```

On associe généralement à chaque bouton radio d'un groupe la même callback pour mémoriser dans une variable globale le choix qui a été fait : la méthode 'value' renvoie 1 quand le bouton est enfoncé, 0 sinon :

```
void BoutonRadioCB(Fl_Widget* w, void* data)
{
    if (gInterface.BoutonRadio1->value() == 1 )
        gDonnees.Option2 = 1 ;
    else if (gInterface.BoutonRadio2->value() == 1 )
        gDonnees.Option2 = 2 ;
    else
        gDonnees.Option2 = 3 ;
}
```

La méthode 'value' permet également d'initialiser l'état des boutons radio :

```
gInterface.BoutonRadio1->value( 1 ) ;
```

# Fltk

## *Menu d'options*

---

### Menu d'options

Un menu d'options permet de faire un choix exclusif parmi plusieurs options, la fonctionnalité est la même qu'avec un groupe de boutons radio, mais elle se présente sous la forme d'un menu :



```
Fl_Choice* gInterface.MenuOptions ;  
...  
gInterface.MenuOptions = new Fl_Choice( X, Y, L, H, Libellé ) ;  
gInterface.MenuOptions->add( "Item1", "", MenuOptionsCB ) ;  
gInterface.MenuOptions->add( "Item2", "", MenuOptionsCB ) ;  
gInterface.MenuOptions->add( "Item3", "", MenuOptionsCB ) ;  
gInterface.MenuOptions->add( "Item4", "", MenuOptionsCB ) ;
```

On associe généralement une callback aux items pour mémoriser dans une variable globale le choix qui a été fait, la méthode 'value' renvoie l'indice de l'item choisi :

```
void MenuOptionsCB( Fl_Widget* w, void* data )  
{  
    gDonnees.Option3 = gInterface.MenuOptions->value() ;  
}
```

La méthode 'value' permet également d'initialiser l'état du menu :

```
gInterface.MenuOptions->value( 0 ) ; // premier item
```

# Fltk

## *Dialogues Message, Avertissement*

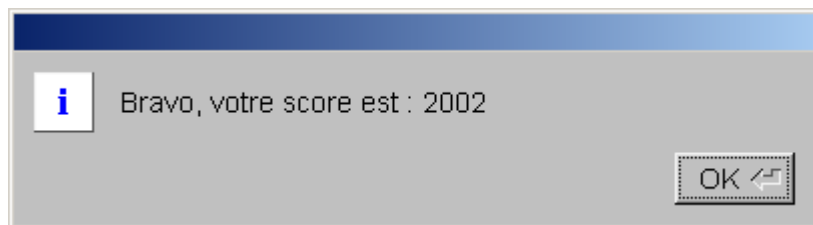
---

Un dialogue est une petite fenêtre qui apparaît au dessus de la fenêtre principale et qui permet d'interagir avec l'utilisateur sur un sujet donné ( autre que ceux directement gérés dans le fenêtre principale de l'application ).

Dans le cadre des projets nous allons utiliser les dialogues standards fournis par Fltk.

### Message

Ce dialogue permet d'afficher un message à l'utilisateur :



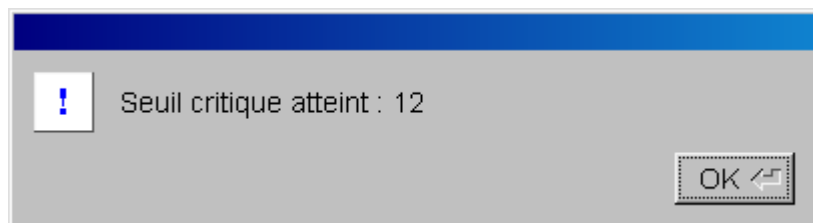
```
fl_message("Bravo, votre score est : %d", Score );
```

fl\_message s'utilise sur le même principe que printf.

L'appel de fonction se termine quand l'utilisateur clique sur OK.

### Avertissement

Ce dialogue permet d'alerter l'utilisateur avec un message :



```
fl_alert("Seuil critique atteint : %d", Seuil );
```

fl\_alert s'utilise sur le même principe que printf.

L'appel de fonction se termine quand l'utilisateur clique sur OK.



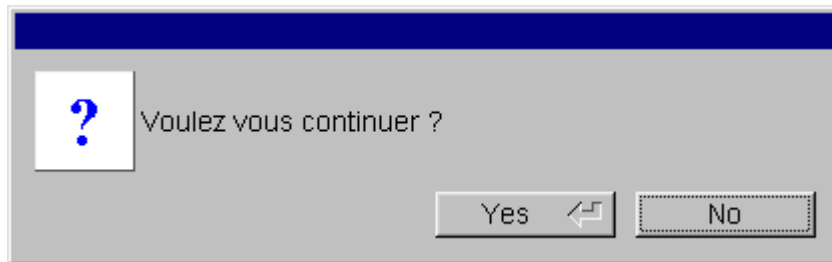
# Fltk

## *Dialogue Question*

---

### Question

Ce dialogue permet de poser une question à l'utilisateur :



```
int Reponse ;  
Reponse = fl_ask("Voulez vous continuer ?");
```

`fl_ask` s'utilise sur le même principe que `printf`.

L'appel de fonction se termine quand l'utilisateur clique sur Yes ou No, la fonction renvoie alors respectivement 1 ou 0.

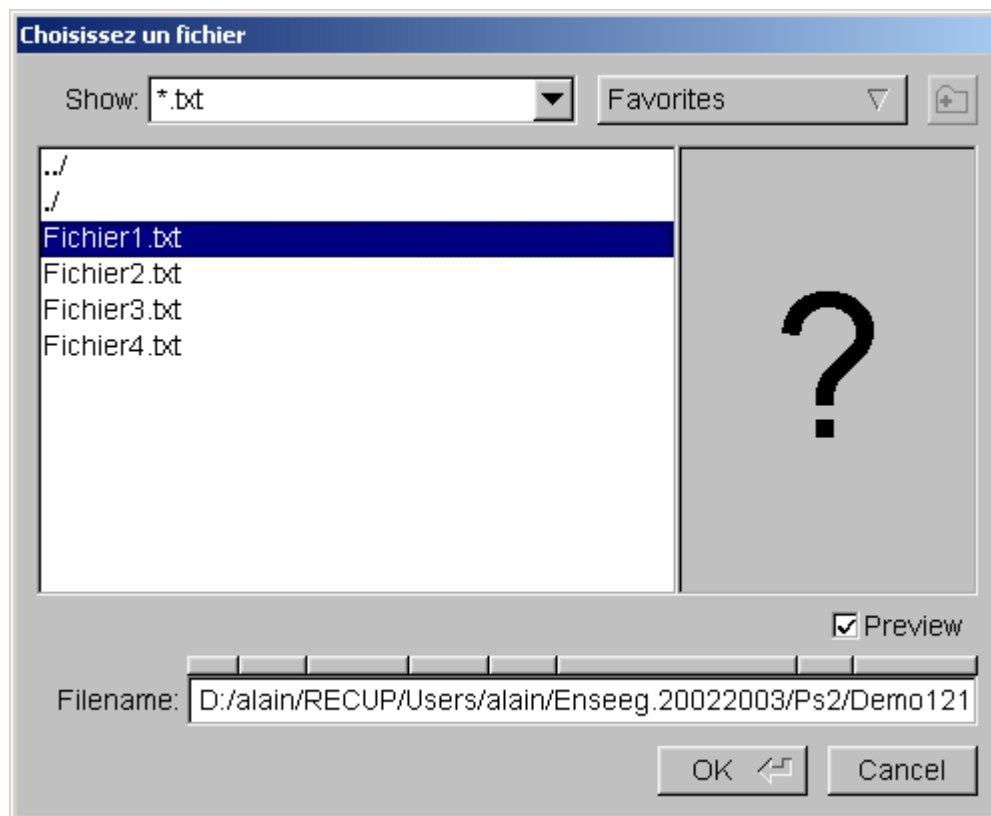
# Fltk

## *Dialogue de choix d'un fichier*

---

### Choix d'un fichier

Ce dialogue permet de demander à l'utilisateur de choisir un fichier ( dans le répertoire où a été lancé le programme ) :



```
const char* NomFichier ; // et pas char NomFichier[128]
```

```
NomFichier = fl_file_chooser("Choisissez un fichier", "*.txt", NULL ) ;
```

Fl\_File\_Chooser renvoie le nom du fichier choisi si l'utilisateur a validé son choix en cliquant sur OK, NULL sinon.

Le deuxième argument sert à filtrer les fichiers présentés : dans l'exemple seuls les fichiers ayant l'extension .txt seront proposés.

On est obligé dans ce contexte d'utiliser une chaîne de caractères NomFichier sous sa forme pointeur ( const char\* ).

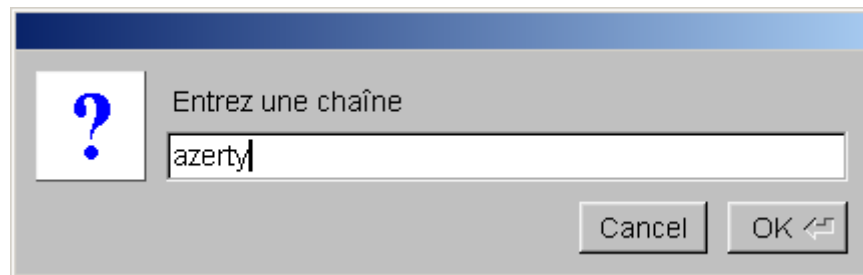
# Fltk

## *Dialogue de saisie texte*

---

### Saisie d'une chaîne de caractère

Ce dialogue permet de demander à l'utilisateur d'entrer une chaîne de caractères :



```
const char* Saisie ; // et pas : char Saisie[80]
char Chaine[80] ;

// Saisie de la chaine
strcpy(Chaine, "" ) ;
do
{
    Saisie = fl_input("Entrez une chaîne", "" ) ;
    if ( Saisie != NULL )
        strcpy (Chaine, Saisie);
} while ( strcmp( Chaine, "" ) == 0 ) ;
```

Fl\_Input renvoie la chaîne de caractères saisie si l'utilisateur la valide en cliquant sur OK, NULL sinon. Elle s'utilise généralement dans une boucle do .. while comme dans l'exemple.

On est obligé dans ce contexte d'utiliser une chaîne de caractères Saisie sous sa forme pointeur ( const char\* ).

# Fltk

## *Dialogue de saisie numérique*

---

### Saisie d'une valeur numérique

On peut utiliser le dialogue précédent pour demander à l'utilisateur d'entrer une valeur entière :



```
const char* Saisie ; // et pas : char Saisie[80]
int Entier ;
int Ok ;

// Saisie de la valeur
Ok = 0 ;
do
{
    Saisie = fl_input("Entrez un entier", "" ) ;
    if ( Saisie != NULL )
        Ok = sscanf( Saisie, "%d", &Entier ) ;
} while ( Ok != 1 ) ;
```

Explications : `sscanf` est en fait une fonction qui renvoie le nombre de valeurs qui ont été extraites conformément à ce qui est demandé (un entier dans l'exemple : `%d`). On se sert de ce nombre pour vérifier la saisie (et éventuellement la relancer si nécessaire : boucle `do ... while`).

# Fltk

## *Méthodes utiles*

---

Tous les objets graphiques Fltk possèdent les méthodes suivantes :

Pour changer la couleur de fond d'un objet :

```
gInterface.Objet->color( Couleur ) ;
```

Pour changer la couleur du libellé d'un objet :

```
gInterface.Objet ->labelcolor( Couleur ) ;
```

Pour désactiver un objet ( par exemple rendre inactif un bouton ) :

```
gInterface.Objet ->deactivate() ;
```

Pour réactiver un objet ( par exemple rendre de nouveau actif un bouton ) :

```
gInterface.Objet ->activate() ;
```

Pour cacher un objet ( par exemple faire disparaître un bouton ) :

```
gInterface.Objet ->hide() ;
```

Pour (re)afficher un objet ( par exemple rendre un bouton de nouveau visible ) :

```
gInterface.Objet ->show() ;
```

# Fltk

## Zone de dessin

---

Une classe **DrawingArea** vous est fournie dans le projet type :

```
DrawingArea* ZoneDessin ;  
...  
gInterface.ZoneDessin = new DrawingArea( X, Y, L, H ) ;  
gInterface.ZoneDessin->draw_callback( ZoneDessinDessinerCB, NULL ) ;  
gInterface.ZoneDessin->mouse_callback( ZoneDessinSourisCB, NULL ) ;  
gInterface.ZoneDessin->keyboard_callback( ZoneDessinClavierCB, NULL ) ;
```

- **ZoneDessinDessinerCB** : pour dessiner la zone quand nécessaire
- **ZoneDessinSourisCB** : pour traiter les clics souris
- **ZoneDessinClavierCB** : pour traiter les actions claviers

Pour analyser un clic souris, on utilise les fonctions suivantes :

- **Fl::event\_button()** : renvoie le bouton utilisé lors du clic ( 1, 2 ou 3 )
- **Fl::event\_x()** : renvoie l'abscisse du clic
- **Fl::event\_y()** : renvoie l'ordonnée du clic

Pour analyser une action clavier, on utilise la fonction suivante :

- **Fl::event\_key()** : renvoie le code de la touche activée ( FL\_Left, FL\_Right, 'a', 'b', ...)

### ATTENTION :

Les coordonnées X, Y ne sont pas relatives à la zone de dessin mais à la fenêtre principale de l'application.

Il faut donc veiller dans ZoneDessinDessinerCB à bien prendre en compte la position de la zone de dessin pour effectuer les bons traitements.

Ceci est aussi valable pour les fonctions de dessin abordées à la page suivante.

# Fltk

## Fonctions de dessin

---

Pour dessiner dans une zone de dessin, Fltk fournit plusieurs fonctions de dessin, notamment :

**fl\_color**( *Couleur* )

Pour définir la couleur de dessin ( FL\_BLACK, FL\_RED, FL\_BLUE, FL\_CYAN, FL\_YELLOW, FL\_WHITE, FL\_GREEN, ... )

**fl\_line\_style**( *Style, Epaisseur* )

Pour définir le style et la largeur des lignes ( FL\_SOLID, FL\_DASH, FL\_DASH\_DOT, ... )

**fl\_line**( *X1, Y1, X2, Y2* )

Pour dessiner une ligne entre 2 points

**fl\_polygon**( *X, Y, X1, Y1, X2, Y2* )

Pour dessiner une polygon plein

**fl\_rectf**( *X, Y, Largeur, Hauteur* )

Pour dessiner un rectangle plein

**fl\_rect**( *X, Y, Largeur, Hauteur* )

Pour dessiner un rectangle vide

**fl\_pie**( *X, Y, D, D, 0, 360* )

Pour dessiner un cercle plein, d'un diamètre D

**fl\_arc**( *X, Y, D, D, 0, 360* )

Pour dessiner un cercle vide, d'un diamètre D

**fl\_point**( *X, Y* )

Pour dessiner un point

**fl\_font**( *Font, Taille* )

Pour définir la fonte de dessin des chaînes de caractères ( FL\_HELVETICA, FL\_COURIER, ... )

**fl\_draw**( *Chaine, X, Y* )

Pour dessiner une chaîne de caractères.

**fl\_polygon**( *X1, Y1, X2, Y2, X3, Y3* )

Pour dessiner un polygone plein défini par 3 points

**fl\_polygon**( *X1, Y1, X2, Y2, X3, Y3, X4, Y4* )

Pour dessiner un polygone plein défini par 4 points

**fl\_loop**( *X1, Y1, X2, Y2, X3, Y3* )

Pour dessiner le contour d'un polygone défini par 3 points

**fl\_loop**( *X1, Y1, X2, Y2, X3, Y3, X4, Y4* )

Pour dessiner le contour d'un polygone défini par 4 points

# Fltk

## *Traitement cyclique*

---

Fltk ne permet pas directement d'exécuter une fonction cycliquement.

Pour cela il faut utiliser la fonction `Fl::add_timeout(Durée, Fonction, NULL )` qui permet d'exécuter une fonction au bout d'une durée donnée, à charge de la fonction de se réarmer.

Afin de faciliter cet aspect, ce mécanisme est mis en place dans l'unité main du projet type pour appeler de manière cyclique la fonction :

- `void TraiterCycleCB ()`

La durée du cycle est définie par une constante que vous pouvez modifier dans `functions.h` :

```
#define DUREE_CYCLE 0.500 // 500 ms
```



# Projet Type

## *Notion d'unité*

---

Pour des programmes volumineux et/ou réalisés par plusieurs développeurs, il est nécessaire de répartir le source dans plusieurs fichiers qu'on appelle unités de compilation.

La décomposition peut être arbitraire, calquée sur la répartition du travail entre les développeurs, guidée par les sous-ensembles du programme (aspects graphiques, aspects fonctionnels ), etc ...

Chaque unité est constituée :

- D'un fichier .h (header) dans lequel on :
  - définit les constantes de l'unité
  - définit les structures de données de l'unité
  - déclare les variables globales de l'unité ( ie avec le mot-clé extern )
  - déclare les sous-programmes de l'unité
- D'un fichier .cpp dans lequel on :
  - définit les variables globales de l'unité
  - définit les sous-programmes de l'unité

Le header d'une unité constitue son interface pour l'extérieur : il décrit ce que fournit l'unité.

Pour pouvoir utiliser dans une autre unité ce que fournit une unité, il suffit d'inclure son header ( de même qu'on inclut iostream pour pouvoir utiliser les fonctions standards d'entrée/sortie comme cin, cout, etc ... )

# Projet Type

## Unités

---

Le projet type est décomposé comme suit :

- main
  - Programme principal **main()**
  - Classe DrawingArea
  - Gestion de l'appel cyclique de **TraiterCycleCB()**

**Nota Bene : Vous ne devez pas toucher à cette unité.**
- u1-interface
  - Objets graphiques de l'interface (gInterface)
  - Procédure **CreerFenetre()** : création de l'ihm
  - Procédure **ActualiserInterface()** : actualisation des éléments de l'interface avec les données du programme
- u2-dessins
  - Procédure **ZoneDessinDessinerCB()** : dessin de la zone
- u3-callbacks
  - Procédure **TraiterCycleCB()** : traitements cycliques
  - Procédure **ZoneDessinSourisCB ()** : gestion des clics souris dans la zone de dessin
  - Procédure **ZoneDessinClavierCB ()** : gestion des actions clavier dans la zone de dessin
  - + les autres callbacks associés aux objets graphiques
- u4-fonctions
  - Données du projet (gDonnees)
  - Procédure **InitialiserDonnees()** : initialisation des données du programme
  - Tous les sous programmes fonctionnels

# Projet Type

## Structures de données

---

Le projet type comporte 2 structures de données déclarées de façon globales (elles sont visibles depuis toutes les parties du programme) :

- Une pour l'interface graphique :
  - o Définition de type struct dans u1-interface.h

```
struct Interface
{
    Fl_Double_Window* Fenetre ;
    ZoneDessin*      Dessin ;
    Fl_Button*        BoutonQuitter ;
    Fl_Check_Button*  CaseRebond ;
    ...
};
```
  - o Déclaration de gInterface dans u1-interface.cpp

```
struct Interface gInterface ;
```
  - o Export de gInterface dans u1-interface.h

```
extern struct Interface gInterface ;
```
- Une pour les données fonctionnelles du programme
  - o Définition de type struct dans u4-fonctions.h

```
struct Donnees
{
    struct Boule  Boule ;

    int      Rebond ;
    char     Texte[80] ;
    unsigned int Valeur ;
    int      Parametre ;
    int      Option1 ;
    int      Option2 ;
    int      Option3 ;
};
```
  - o Déclaration de gDonnees dans u4-fonctions.cpp

```
struct Donnees gDonnees ;
```
  - o Export de gDonnees dans u4-fonctions.h

```
extern struct Donnees gDonnees;
```

Pour rajouter des variables au programme, il suffit de rajouter les nouvelles déclarations à l'intérieur des 2 structures de type struct (Interface et Donnees) définies dans u1-interface.h et u4-fonctions.h

# Projet Type

## *Principes*

---

Le projet type, comme toute application graphique fonctionne ainsi :

**main**

- **CreerFenetre**
- **InitialiserDonnees**
- **ActualiserInterface**
- **Fl::run** ( boucle sans fin de gestion des événements )

Trois types d'événement :

- Clic souris :
  - Sur un objet standard : appel de la **callback** si le clic active l'objet
  - Dans la zone de dessin : appel de **ZoneDessinSourisCB**
- Action clavier :
  - Dans un objet standard : appel de la **callback** si l'action active l'objet
  - Dans la zone de dessin : appel de **ZoneDessinClavierCB**
- Expiration cyclique du timer :
  - Appel de la callback **TraiterCycleCB**

Lorsqu'on traite un événement on est souvent amené à modifier les données de l'application, Il faut alors actualiser l'interface graphique (pour qu'elle reflètent les changements ) :

- En actualisant les valeurs affichées par les objets standards ( champ d'affichage texte, champ d'affichage numérique ...).  
=> pour cela on invoque leur méthode '**value(Valeur)**'.
- En provoquant le rafraîchissement de la zone de dessin.  
=> pour cela on invoque la méthode **redraw()** de la zone de dessin qui provoque automatiquement l'appel de la procédure **ZoneDessinDessinerCB** ().

Fltk est une librairie très performante, on peut se permettre de redessiner complètement la zone de dessin même si ce n'est pas toujours nécessaire, y compris dans le cas d'un traitement cyclique. Ceci est notamment possible grâce à l'utilisation d'une fenêtre de type **FL\_Double\_Window** qui implémente la technique du "double buffering" : les ordres graphiques sont d'abord réalisés sur une image en mémoire, puis c'est l'image résultante qui est affichée en une seule instruction à l'écran, ce qui est performant et résout le problème classique de "flicking" (effet de saccades).