Touraivane
Tél: (33) 4 91 8 85 38
Fax: (33) 4 91 8 85 18 - (33) 4 91 8 85 10
Touraivane@gbm.esil.univ-mrs.fr
http://gbm.esil.univ-mrs.fr/~tourai

Les premiers pas avec Qt

Version V.0

Septembre 1997

ESIL
Ecole Supérieure d'Ingénieurs de Luminy
Département Génie Bio Médical

Table des matières

Avant-propos

Qt est une plateforme de développement d'applications graphiques. ***Qt*** est disponible pour les versions de *X Window* sous les systèmes suivants: *Linux*, *Solaris*, *SunOS*, *FreeBSD*, *OSF/1*, *Irix*, *BSD/OS*, *NetBSD*, *SCO*, *HP-UX* and AIX. Il est également disponible sous *Windows 95* et *Windows NT*.

Ces applications écrites en C++ sont indépendantes de du système utilisé. Une application réalisée en avec ***Qt*** devrait touner aussi bien sous X Window que sur *Windows 95* et *Windows NT*; à condition évidemment de disposer de ***Qt*** pour ces divers systèmes et de recompiler les sources.

Qt propose une plateforme de developpment de haut niveau; le programmeur ne se soucie plus des détails de bas niveau de la librairie graphique du système cible.

Un ambitieux projet nommé *kde*, utilise ***Qt*** comme outil de développement pour réaliser un environnment graphique complet sous X (Window Manager et applications graphiques) permettant d'utiliser le « *drag and drop* » comme sous le système *NeXT*.

On trouvear tous les rensignements sur ***Qt*** à l'adresse suivante: `http://www.troll.no`

A qui s'adresse ce document ? Le présent document s'adresse aux étudiants qui connaissent le *langage C++*: c'est le seul pré requis.

Ce document n'est pas une présentation complète de ***Qt***, il s'agit tout au plus d'un support de travaux pratiques pour ceux qui désirent découvrir ***Qt***. Chaque chapitre correspond à environ quatre heures de travaux pratiques (1h30 de présentation). Les exemples qui figurent dans ce document sont généralement de taille réduite.

Pour avoir une meilleur idée de ***Qt*** consulter l'excellente documentation qui est disponible à http://www.troll.no. Une copie de cette document est disponible également à `http://gbm.esil.univ-mrs.fr/~tourai/Qt`

Chapitre 1

Découvrir Qt

Sommaire

Une application ***Qt*** est une application graphique qui va interagir avec l'utilisateur de cette application. Le concept de base d'une application graphique est le `widget`. Un *widget* est l'entitié de base d'une interface graphique qui réagit aux actions d'un utilisateur: manipulation de la souris, du clavier et tout autre évènement du système graphique.

Les *widgets* sont de forme rectangulaires sont organisés hiérarchiquement. Un *widget* particulier appelé *widget principal* *widget racine* ou se trouve au sommet de cette hiérarchie.

Du point de vue du programmeur, une application ***Qt*** est un objet de la classe `QApplication`C'est cet objet qui se charge de la gestion des évènements d'une application donnée: c'est la classe centrale de ***Qt*** qui recoit de l'environnement graphique des évènements et qui les transmet à un objet graphique particulier (*widget*).

Chaque programme ne comporte qu'un objet de type `QApplication` et la variable globale `qApp` fait référence à cet objet (objet défini par `extern QApplication *qApp`).

Les *widgets* de ***Qt*** sont des objets de la classe `QWidget`.

1.1 Une première application graphique

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

```
#include <qapp.h>
```

La classe QPushButton Le fichier `qapp.h` ne contient pas tous 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 `qpushbt.h`.

```
#include <qpushbt.h>
```

Un «bouton» est un objet de la classe `QPushButton` et possède par défaut un aspect (un *look* diront certains) prédéfini. Nous verrons plus loin, comment modifier cet aspect par défaut.

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 crée 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`.

Présenation de argv et argc

Le constructeur `QApplication` supprime de `argv` les éléments qu'il a utilisé (en mettant à jour `argc`): ce sont généralement des arguments nécessaires à l'environnement graphique.

```
#include <iostream.h>
#include <qapp.h>

int main( int argc, char **argv ) {
  for ( int i=0; i<argc; i++ )
    cout «  argv[i] « "  ";

  cout «  "  ***  ";

  QApplication a( argc, argv );

  for ( int i=0; i<argc; i++ )
    cout «  argv[i] « "  ";
}
```

Si le fichier exécutable se nomme `args`, la commande

```
args -display gbm3:0 arg1 -geometry 10x20 arg2
```

produira l'affichage

```
args  -display  gbm3:0  arg1  -geometry  10x20  arg2 *** args  arg1  arg2
```

1.2 Créer un bouton

Un «bouton» est un objet de la classe `QPushButton` que l'on crée comme tout autre objet.

```
QPushButton (QWidget* parent=0, const char* name=0)
QPushButton (const char* text, QWidget* parent=0, const char* name=0)
```

Pour le moment, on se contentera de noter que l'on peut créer un objet de type `QPushButton(chaine)` où `chaine` désigne la chaîne de caractères qui sera affichée sur le bouton.

```
QPushButton bouton( "Coucou !!!" );
```

La classe `QPushButton` dispose d'un certain nombre de méthodes dont, par exemple, celle qui définit la taille d'un *widget*.

```
bouton.resize(100, 50);
```

Les deux entiers constituent la taille en nombre de pixels horizontaux et verticaux.

Pour notre premier exemple, le widget principal est le bouton que venons de définir, ce qui se définit comme suit:

```
a.setMainWidget( &bouton);
```

Si le widget principal est fermé alors l'application s'arrête.

A priori, il n'est pas nécessaire d'avoir un widget principal, mais la plupart des programmes en ont un.

Les *widgets* ne sont jamais visibles lors de leur création. Pour les rendre visibles, il faut le demander explicitement en appliquant la méthode `show` à l'objet *bouton*.

```
bouton.show();
```

Une fois les widgets créés, il faut fair faire appel à ***Qt*** pour qu'il prenne en main la gestion des évènements utilisateurs.

```
a.exec();
```

Et voilà, pour cette première application. Il ne reste plus qu'à compiler ce fichier et exécuter le programme.

```
// Fichier: exemple1.cpp
// Une premiere application qui se contente d'ouvrir une fenêtre
// contenant un bouton étiqueté "Coucou !!!"

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

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

  QPushButton bouton( "Coucou !!!" );
  bouton.resize( 100, 30 );

  a.setMainWidget( &bouton );
  bouton.show();
  return a.exec();
}
```

On compilera le fichier `exemple1.cpp` en n'oubliant pas d'inclure la librairie de ***Qt***.

```
g++ -o exemple1 exemple1.cpp -lqt
```

et vous devriez obtenir le résultat suivant:

1.3 Des widgets personalisés

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 contient deux boutons; pour ce faire, nous allons créer une nouvelle sous classe de la classe `QWidget`.

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

FIG. 1.1 – *Exemple1*

```
#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 founi en argument. Le deuxième argument de ce constructeur est le nom du widget. Ce n'est la chaîne figurant sur le widget mais un nom permettant de rechercher un widget particulier parmis l'ensemble des widget d'une application donnée. L'intérêt de cet argument sera plus évident vers la fin de document.

Ne pas oublier de parler de ca à la fin

```
// ********** 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" );
  }
```

Le programme principal est le suivant:

```
// ********** 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 );
}
```

FIG. 1.2 – *Deux boutons*

Le système de coordonnés de Qt

1.4 Gérer des évènements

Nous allons modifier l'exemple précédant pour y inclure une première gestion des évènements utilisateur i.e. effectuer une action en réaction à une action de l'utilisateur.

1.4.1 Slots et signaux

La méthode `connect` est la méthode la plus importante de ***Qt***. C'est une méthode `static` de la classe `QObject` [1].

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

Cette méthode établit une communication unidirectioinnelle entre deux objets ***Qt***. A chaque objet ***Qt*** (objet de la classe `QObject` ou d'une classe dérivée), on peut associer des *siganux* 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 basée sur cette notion d'émission et reception 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.

1. Attention: il existe une autre méthode `connect` pour les sockets

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 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`.

1.4.2 L'émetteur 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;
}
```

1.4.3 Le 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;
};
```

1.4.4 L'utilitaire moc

Une fois déclarées les classes `objet1` et `objet2`, on utilise l'utilitaire `moc` pour produire un ou plusieurs fichier 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`.

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

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

```
gbm > 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.

Chapitre 2

Gérer le temps

Sommaire

2.1 Gestion d'évènements graphiques

Complétons l'exemple du widget `NouveauWidget` pour que les boutons `Compter` et `stop` réagissent aux actions de l'utilisateur. Il nous faut donc modifier la classe `NouveauWidget` pour y ajouter les slots.

```
// ************  nwidg.h  ************
#include <iostream.h>
#include <qapp.h>
#include <qpushbt.h>

class NouveauWidget : public QWidget {
  Q_OBJECT;
public slots:
  void coucou()
    { cout « "coucou" « endl; }
public:
  NouveauWidget( QWidget *pere=0, const char *nom=0 );
};
```

On se contente d'afficher la chaîne de caractères `coucou` chaque fois que l'on «cliquera» sur le bouton `coucou`.

```
QObject::connect(go, SIGNAL(clicked()), this, SLOT(coucou()));
```

De plus, un bouton `quit` permet de quitter l'application.

```
QObject::connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
```

D'où le programme suivant:

```
// ************  nwidg.cpp  ************

NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name ) {
  QPushButton *go = new QPushButton( "Coucou !", this, "go" );
  go->setGeometry( 10, 10, 60, 30 );
  QObject::connect(go, SIGNAL(clicked()), this, SLOT(coucou()));

  QPushButton *quit = new QPushButton( "Quit", this, "quit" );
  quit->setGeometry( 100, 10, 60, 30 );
  QObject::connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```

Le programme principal reste inchangé:

```
// ************  main.cpp  ************
#include <qapp.h>
#include "nwid.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();
  }
```

2.2 Le timer

La classe `QTimer` propose des siganux permettant de déclencher à intervalles réguliers des signaux `timeout`. Cet intervalle est précisé lors du démarrage d'un objet `QTimer` à l'aide la méthode `start(int t, bool type)` où `t` est l'intervalle de temps en millisecondes et `type` est une valeur booléenne (`TRUE` si l'émission du signal n'est fait qu'une fois et `FALSE` pour une émission régulière).

Cette émission de signaux peut être arretée à tout moment par l'invocation de la méthode `stop()`.

La méthoode `changeInterval(int t)` permet de modifier l'intervalle du temps de déclenchement des signaux `timeout`qtimer.html#a7 et la méthode `isActive()`qtimer.html#a2 retourne la valeur vraie si l'objet `QTimer` est actif et faux sinon.

Pour illustrer cette classe, nous allons créer une application qui affiche, sur le terminal, la chaîne de caractère `coucou` toutes les secondes dès que l'on clique sur le bouton `go`. Cet affichage s'arrête en cliquant sur le bouton `stop`.

Comme dans les exemples précédants, créons un widget contenant les boutons `go`, `stop` et `quit`.

```
// *********  nwind.cpp  **********
#include "nwid.h"

NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name ) {
  // Creer un objet de type QTimer
```

```
    timer = new QTimer( this );
    // Connecter le siganl timeout du timer au slot coucou()
    connect( timer, SIGNAL(timeout()), SLOT(coucou()) );

    // Creer le bouton go
    QPushButton *go = new QPushButton( "Partez", this, "go" );
    pb_go->setGeometry( 10, 10, 60, 30 );
    // Connecter ce bouton au slot go() qui démarre le chornomètre
    connect(pb_go, SIGNAL(clicked()), this, SLOT(go()));


    // Creer le bouton stop
    QPushButton *stop = new QPushButton( "Stop", this, "stop" );
    pb_stop->setGeometry( 80, 10, 60, 30 );
    // Connecter ce bouton au slot stop()
    // qui stop l'affichage la chaîne "coucou".
    connect(pb_stop, SIGNAL(clicked()), this, SLOT(stop()));


    // Creer le bouton quit
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    pb_quit->setGeometry( 45, 50, 60, 30 );
    // Connecter ce bouton au slot quit()
    // qui permet de quitter l'application.
    connect(pb_quit, SIGNAL(clicked()), qApp, SLOT(quit()));
  }
```

La classe `NouveauWidget` doit contenir les slots `go()` et `stop()` permettant de déclencher et interrompre l'affichage de la chaîne `coucou`.

La méthode `void go()` que nous allons définir, elle va tout simplement démarrer le `timer` et fixer la fréquence d'émission du signal. `timeout` à 1000 ms.

```
  void go() {
    timer->start( 1000, FALSE);     // Démarrer le timer
    coucou();
  }
```

Quant à la méthode `void stop()` , elle va stoper le `timer` et donc donc stopper les émission des siganux `timeout`.

```
  void stop() {
    timer->stop();                  // Arrêter le timer
  }
```

Enfin, la méthode `void coucou()`, qui est appelée chaque fois que le `timer` envoie un signal `timeout`, afficher sur le terminal la chaîne de carcatère `" ... coucou"`.

```
  void coucou()   {
    cout « " ... coucou" « endl; // Chaine à afficher
  }
```

D'où le programme suivant:

```
  // *********  nwind.h  **********
```

```
#include <iostream.h>
#include <qapp.h>
#include <qpushbt.h>
#include <qtimer.h>

class NouveauWidget : public QWidget {
  Q_OBJECT;
public slots:
  void go() {
    timer->start( 1000, FALSE);     // Démarrer le timer
    coucou();
  }
  void stop() {
    timer->stop();                  // Arrêter le timer
  }
  void coucou()   {
    cout « " ... coucou" « endl; // Chaine à afficher
  }
public:
  NouveauWidget( QWidget *pere=0, const char *nom=0 );
private:
  QTimer *timer;
};
```

2.3 Le chronomètre

Nous allons à présent utiliser les concepts que l'on vient de voir pour programmer un chronomètre. Pour réaliser cette application, nous utiliserons un nouveau widget appelé `QLCDNumber` capable d'afficher un nombre de manière esthétique: c'est encore un nouveau *widget*.

FIG. 2.1 – *QLCDNumber*

Dans cet exemple qui est très proche de l'exemple précédant, à chaque signal `timeout` émis, nous allons incrémenter un compteur `interrupt`; il contiendra donc le nombre de fois que le siganl `timeout` a été émis. Il suffit donc afficher ce nombre régulièrement pour voir défiler les unités de temps sur notre chonomètre. En disposant d'un timer qui émet un signal toutes les 10 ms, la variable `interrupt` contiendra le temps écoulé, en 1/100 de secondes) depuis le *click* sur la bouton *go*.

```
//                              nwid.h

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

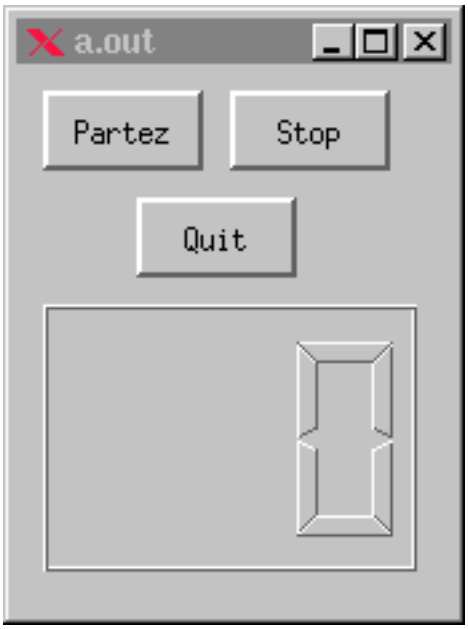

FIG. 2.2 – *Le chonomètre*

```
#include <qtimer.h>
#include <qlcdnum.h>


class NouveauWidget : public QWidget {
  Q_OBJECT;
public slots:
  void go() {
    if (timer->isActive()) return;
    interrupt = 0;
    timer->start( 10, FALSE);
  }

  void stop() {
    timer->stop();
  }

void avancer() {
  interrupt ++;
  emit affiche(interrupt);
}

public:
  NouveauWidget( QWidget *pere=0, const char *nom=0 );
signals:
  void affiche(int);
private:
```

```
  int interrupt;
  QTimer *timer;
};


//                              nwid.cpp
#include "nwid.h"

NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name ) {
  QPushButton *pb_go = new QPushButton( "Partez", this, "go" );
  pb_go->setGeometry( 10, 10, 60, 30 );
  QObject::connect(pb_go, SIGNAL(clicked()), this, SLOT(go()));

  QPushButton *pb_stop = new QPushButton( "Stop", this, "stop" );
  pb_stop->setGeometry( 80, 10, 60, 30 );
  QObject::connect(pb_stop, SIGNAL(clicked()), this, SLOT(stop()));

  QPushButton *pb_quit = new QPushButton( "Quit", this, "quit" );
  pb_quit->setGeometry( 45, 50, 60, 30 );
  QObject::connect(pb_quit, SIGNAL(clicked()), qApp, SLOT(quit()));

  QLCDNumber *lcd = new QLCDNumber(3, this, "lcd");
  lcd->setGeometry(20, 90, 100, 80);

  timer = new QTimer( this );
  connect( timer, SIGNAL(timeout()), SLOT(avancer()) );
  connect( this, SIGNAL(affiche(int)), lcd, SLOT(display(int)) );

  adjustSize();
}
```

2.4 Une animation graphique

Puisque l'on sait afficher à intervalles réguliers, un nombre dans une fenêtre, on pourrait utiliser cette technique pour afficher des images. Si l'on se donne une serie d'images que l'on afficher successivement dans une portion de la fenêtre de l'application, on aura réaliser une animation graphique.

Donc, le dernier exemple d'utilisation d'un timer est la création d'une animation graphique avec les images de la figure 2.3.

Pour créer l'animation, il nous faut pouvoir afficher une image sur un widget. Le premier problème à régler de déterminer la manière d'afficher une image sur un widget. ***Qt*** fournit une classe `QPixmap` qui permet de faire cela.

2.4.1 Afficher un image sur un widget

Avant de commencer l'application qui produit l'animation, nous allons tout d'abord modifier l'application chronomètre en remplaçant les étiquettes `go`, `stop` et `quit` par des icônes (ou images). Pour ce faire, on définira un objet de type `QPixmap` qui charge l'image (en format bitmap). Cet objet sera alors passé en argument de la méthode `setPixmap(QPixmap pm)` comme suit:

```
QPixmap pm;
if (pm.load("nom du fichier bitmap")) // Charger le fichier image
```

FIG. 2.3 – *Une animation*

```
    pb_go->setPixmap( pm );           // affecter l'image au bouton
else                                  // si le fichier image n'existe pas
    pb_go->setText( "No pixmap" );    // afficher le texte "No pixmap"
```

Le programme de notre chronomètre se modifie de la manière suivante:

```
// *********  nwind.cpp  **********

#include <qpixmap.h>
#include "nwid.h"

NouveauWidget::NouveauWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
    {
    ...

    QPixmap pm;
    if (pm.load("go.bmp")) pb_go->setPixmap( pm );
    else  pb_go->setText( "No pixmap" );
    if (pm.load("stop.bmp")) pb_stop->setPixmap( pm );
    else  pb_stop->setText( "No pixmap" );
    if (pm.load("quit.bmp")) pb_quit->setPixmap( pm );
    else  pb_quit->setText( "No pixmap" );

    }
```

ET voilà le résultat:

2.4.2 Et enfin l'animation

L'application que l'on va réaliser est toujours composée d'un widget contenant les trois boutons `go`, `stop` et `quit` et d'un zone pouvant contenir les images qui constituent l'animation. Reprenons l'exemple du chronomètre pour le modifier.

La zone où l'on veut faire afficher l'animation est un objet de la classe `QFrame`. La suite d'images que l'on veut afficher sera stockée dans un tableau de `QPixmap`.

```
class NouveauWidget : public Widget {
    ...
    private:
    ...
    QPixmap images[16];
    QFrame *qf;
    int i;
    }
```

Complétons donc le constructeur de la classe `NouveauWidget` pour rajouter ce nouveau widget dans le widget principal.

```
    qf = new QFrame(this, "animation");
    qf->setGeometry(0, 100, 130, 80);
```

Ensuite, chargeons la suite d'images dans le tableau défini plus haut:

```
    images[0].load("tumble/T1.bmp");
    images[1].load("tumble/T2.bmp");
    ...
    images[15].load("tumble/T16.bmp");
```

Enfin, modifions le slot `avancer()` pour l'adapter à la réalisation de l'animation. On dispose de 16 images, on utilise un compteur qui s'incrémente de 1 à chaque émission du signal `timeout` (et tout cela modulo 16 bien évidemment). Le compteur incrémenté, on affiche l'image contenu dans le tableau de pixmap à l'indice indiqué par le compteur. Et le tour est joué !

```
void avancer()
  i = (i+1) % 16;
  qf->setBackgroundPixmap( images[i] );
```

Chapitre 3

Gérer la souris et le clavier

Sommaire

3.1 La gestion de la souris (QMouseEvent)

La classe `QMouseEvent` permet la description des « évènments souris » tels la pression sur un bouton, le déplacement de la souris , le relachement d'un bouton, etc.

En règle générale, on s'intéresse au déplacement de la souris en ayant un bouton pressé. Pour gérer les déplacements sans action sur un bouton, il faut modifier le comportement des widgets en utilisant la méthode `setMouseTracking (bool)`.

De même, on déactivera la réception des évènements (souris ou clavier) sur un widget en utilisant la méthode `setEnable (bool)`.

Dans l'exemple qui suit, nous allons réaliser une application permettant de dessiner sur une fenêtre à l'aide de la souris. Les méthodes virtuelles `mousePressEvent(QMouseEvent*)`, `mouseReleaseEvent (QMouseEvent*)` `mouseDoubleClickEvent (QMouseEvent*)`, et `mouseMoveEvent (QMouseEvent*)` ont des noms assez explicites et se passent d'explications.

La classe `QMouseEvent` dispose des méthodes

```
const QPoint& pos () const
int x () const
int y () const
int button () const
int state () const
```

permettant de connaître la position sur la fenêtre où l'évènement « souris » se produit, le bouton appuyé et éventuellement de la combinaison d'une action de la souris et d'une action clavier.

Pour réaliser cette très simple feuille de dessin, il faut repérer la postion de la souris quand l'utilisateur «clique» dessus. Cette position est stocké dans les variables x et y, membre privés de la classe `dessin` que l'on va définir.

```
void dessin::mousePressEvent( QMouseEvent *e) {
```

```
  x = e->x(); y = e->y();
}
```

Puis à chaque déplacement de souris (souris appuyée), on va dessiner un petit trait entre la postion repérée précédemment et la position actuelle de la souris; on conserve ensuite, dans les variables `x` et `y`, la position actuelle de la souris.

```
void dessin::mouseMoveEvent( QMouseEvent *e) {
  paint->drawLine(x, y, e->x(), e->y());
  x = e->x(); y = e->y();
}
```

On supposera connu l'existence de la méthode dans la classe `QPainter`

```
void drawLine (int x1, int y1, int x2, int y2)
```

pour dessiner un trait entre les points de coordonnées (x_1, y_1) et (x_2, y_2).

Enfin, lorsque la souris sera relachée, on terminera le dernier trait.

```
void dessin::mouseReleaseEvent( QMouseEvent *e) {
  paint->drawLine(x, y, e->x(), e->y());
}
```

```
#include <qapp.h>
#include <qwidget.h>
#include <qpainter.h>

class dessin : public QWidget {
private:
  int x, y;         // Pour conserver les coordonnées de la
                    // précédante position de la souris
  QPainter *paint;
public:
  dessin( QWidget *parent=0, const char *name=0 ):QWidget(parent, name) {
    setBackgroundColor(white);
    paint = new QPainter();
    paint->begin(this);
    paint->setPen( blue );
  }
  ~dessin() {
    paint->end();
    delete paint;
  }
protected:
  void  mousePressEvent( QMouseEvent *e) {
    x = e->x(); y = e->y();
  }
  void  mouseReleaseEvent( QMouseEvent *e) {
    paint->drawLine(x, y, e->x(), e->y());
    x = e->x();
    y = e->y();
  }
  void  mouseMoveEvent( QMouseEvent *e) {
    paint->drawLine(x, y, e->x(), e->y());
```

```
  }
};
```

On notera que dans cet exemple, le dessin est ephémère: si la fenêtre se ferme ou est masquée par une autre fenêtre, à la réapparition de cette première, le dessin aura disparu. Ce phénomène et la manière d'y remédier seront étudiés en détail dans le chapitre 4.

3.2 La gestion du curseur (QCursor)

La classe `QCursor` permet de gérer la forme et la position du curseur selon le widget sur lequel il se trouve. Il existe un certain nombre de formes prédéfinies pour le curseur: `arrowCursor`, `upArrowCursor`, `crossCursor`, `waitCursor`, `ibeamCursor`, `sizeVerCursor`, `sizeHorCursor`, `sizeBDiagCursor`, `sizeFDiagCursor`, `sizeAllCursor`, `blankCursor`. Mais, on peut également définir un curseur à partir d'une image bitmap.

A chaque widget, on peut associer une forme de curseur différente avec la méthode `setCursor()` de la classe `QWidget`. Par contre, pour la modifier de manière globale, on utilisera la méthode `setOverrideCursor` de la classe `QApplication`.

3.2.1 Les curseurs standards

```
#include <qapp.h>
#include <qwidget.h>
#include <qcursor.h>
#include <qbttngrp.h>
#include <qradiobt.h>

class curseur : public QWidget {
  Q_OBJECT;
public:
  curseur(QWidget *pere=0, const char *nom=0);

public slots:
  void ChoixRadioButton( int i) {
    setCursor(curseurs[i]);
  }
private:
  QButtonGroup *bg;
  QRadioButton *rb;
  static QCursor curseurs[];
  static char * curseur::noms[];
};

#include "exo14.h"

curseur::curseur(QWidget *pere=0, const char *nom=0): QWidget(pere, nom) {
  bg = new QButtonGroup( this, "radioGroup" );
  bg->setTitle( "Les curseurs" );
  for (int i = 0; i<11; i++) {
    rb = new QRadioButton( bg );
    rb->setText( noms[i]);
    rb->setGeometry( 10, 15 +i*20, 150,25 );
  }
```

```
  bg->adjustSize();
  connect( bg, SIGNAL(clicked(int)), SLOT(ChoixRadioButton(int)) );
}

QCursor curseur::curseurs[] = {
  arrowCursor, upArrowCursor, crossCursor,
  waitCursor, ibeamCursor, sizeVerCursor,
  sizeHorCursor, sizeBDiagCursor, sizeFDiagCursor,
  sizeAllCursor, blankCursor};

char * curseur::noms[] = {
  "arrowCursor", "upArrowCursor", "crossCursor",
  "waitCursor", "ibeamCursor", "sizeVerCursor",
  "sizeHorCursor", "sizeBDiagCursor", "sizeFDiagCursor",
  "sizeAllCursor", "blankCursor"};

int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  curseur w;
  a.setMainWidget( &w );
  w.show();
  w.adjustSize();
  return a.exec();
}
```

3.2.2 Les curseurs personalisés

A terminer !!!!!!!!!!!

3.3 La gestion du clavier (QKeyEvent)

La classe **QKeyEvent** permet de gérer les «évènements clavier» grâce aux méthodes:

```
QKeyEvent (int type, int key, int ascii, int state)
int key () const
int ascii () const
int state () const
bool isAccepted () const
void accept ()
void ignore ()
```

Si l'on ne veut pas gérer l'évènement, on devra utiliser la méthode **ignore()** de la classe **QKeyEvent**.

La méthode **setEnable(bool)** de la classe **QWidgetrmet** de gérer ou d'ignorer les évnènements clavier et souris.

Les méthodes virtuelles **keyPressEvent()** et **keyReleaseEvent()** de la classe **QWidget** sont celles qui reçoivent les évènements clavier.

```
#include <qapp.h>
#include <qwidget.h>
#include <qevent.h>
#include <qframe.h>
```

```
#include <qlabel.h>
#include <qstring.h>

class clavier : public QWidget {
public:
  clavier(QWidget *pere=0, const char *nom=0) : QWidget(pere, nom) {
    msg = new QLabel( this, "message" );
    msg->setFrameStyle( QFrame::Panel | QFrame::Sunken );
    msg->setAlignment( AlignCenter );
    msg->setGeometry( 0,0, 300,30 );
    msg->setFont( QFont("times",12,QFont::Bold) );
  }
protected:
  void keyPressEvent(QKeyEvent *e) {
    QString str;
    QString s;
    str =  "Touche appuyee: ";
    switch(e->state()) {
    case ShiftButton:
      str += " ShiftButton + "; break;
    case ControlButton:
      str += " ControlButton + "; break;
    case AltButton:
      str += " AltButton + "; break;
    default:
      str += " Normal + "; break;
    }
    s.sprintf( "[%c] (%d)",  e->ascii(), e->key());
    str = str + s;
    msg->setText( str );
  }
private:
  QLabel     *msg;
};

int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  clavier w;
  a.setMainWidget( &w );
  w.show();
  w.adjustSize();
  return a.exec();
}
```

3.4 Les autres évènements

Voici, à présent, une liste des évènements que l'on peut gérer:

- `mousePressEvent`:
- `mouseReleaseEvent`:

- `mouseDoubleClickEvent`:
- `mouseMoveEvent`:
- `keyPressEvent`:
- `keyReleaseEvent`:
- `focusInEvent`:
- `focusOutEvent`:
- `enterEvent`:
- `leaveEvent`:
- `paintEvent`:
- `moveEvent`:
- `resizeEvent`:
- `closeEvent`:

A TERMINER !!!!!!!!!!!!!!

Voir documentation officielle de ***Qt***.

Chapitre 4

Dessiner sur une fenêtre graphique

Sommaire

La classe `QPainter` founit une moyen efficce et simple de dessiner des formes géométriques, du texte ainsi que des images pixmap. Une utilisation typique consiste à

- démarrer les dessins avec la méthode `begin`
- affecter les caractéristiques du contour (`setPen`) et du fond `setBrush`
- dessiner les formes voulues
- finir avec l'appel de la méthode `end`

```
    void  paintEvent( QPaintEvent * ) {
      QPainter paint;
      paint.begin( this );
      paint.setPen( blue );
      // ...
      // Dessiner
      // ...
      paint.end();
      }
};
```

Les objets `QPainter` ne sont quasiment jamais utilisés en dehors de la méthode `QPaintEvent()`. A tout moment, les widgets doivent être capable de se dessiner (ou se redessiner). En effet, une fenêtre contenant un widget peut être momentanément invisble (par superposition d'une autre fenêtre, par fermeture de la fenêter, etc.); il est indispensable, lorsque celle-ci redevient visible de pouvoir redessiner tout son contenu i.e. tous les widgets, les sous widgets etc. Tout ce travail doit être localisé dans la méthode `QPaintEvent()` et celle-ci sera utilisée chaque fois que les méthodes `update` ou `repaint` seront appelées.

4.1 Rafraîssement d'une fenêtre

La méthode `QPaintEvent()` est une méthode virtuelle; son comprtement par défaut (défini dans la classe `QWidget`) n'est généralement pas très utile. Si notre widget contient des éléments que la méthode `QPaintEvent()` de la classe `QWidget` ne peut gérer toute seule, il faut absoluement définir une méthode `QPaintEvent()` adapté à notre widget.

Reprenons l'exemple de la feuille de dessin 3.1. Si l'on exécute ce programme, on voit apparaître une fenêtre dans un fond blanc (`setBackgroundColor(white)`) et à l'aide la souris on arrive à dessiner des formes.

A présent, si l'on cache cette fenêtre par une autre fenêtre ou qu'on « l'iconifie » et qu'on la fait fait réapparaître, tous nos dessins ont disparus. Par contre, le fond de la fenêtre est toujours blanc.

Que s'est-il passé? N'ayant pas implanté la méthode `QPaintEvent()`, ***Qt*** ne sait pas ce qu'il faut redessiner. Tout ce qu'il est capable de faire (comprtement par défaut) consiste à redonner à la fenêtre l'apparence qu'il avait avant que l'on commnce à dessiner: il a bien mis un fond blanc (et non pas gris comme tous les widgets par défaut). Autrementdit, la méthode par défaut est capable de retrouver les propriétés d'un widget mais pas son contenu dynamique.

Pour corriger ceci, il faut absoluement définir la méthode `QPaintEvent()` pour que, à tout moment, on puisse être capable de refaire le dessin réaliser par l'utilisateur. Pour ce faire, il faudra mémoriser tous les points par lesquels la souris est passée et redessiner entièrement le contenu de la fenêtre.

On prendra une implantation simpliste : Deux tableaux d'entier pouvant contenir les coordonées des points; tableaux qu'on limitera à ne contenir qu'une centaine de points.

```
const int maxpts = 100;
int x[maxpts], y[maxpts];  // Pour conserver toutes les coordonnées
                           // des positions successives de la soris
int sommet;                // nombre de points mémorisés
```

Lorsqu'on «clique» sur la souris, on marque le début d'une suite de points en empilant la valeur -1; puis la position de la souris est empilée.

```
void  mousePressEvent( QMouseEvent *e) {
  empiler(-1,-1);
  empiler(e->x(), e->y());
}
```

Puis à chaque déplacement de souris (souris appuyée), on va dessiner un petit trait entre la postion repérée précédemment et la position actuelle de la souris; et tout ceci, en ayant pris le soin d'empiler la position de la souris.

```
void  mouseMoveEvent( QMouseEvent *e) {
  empiler(e->x(), e->y());
  paint->drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
}
```

Enfin, lorsque la souris sera relachée, on terminera le dernier trait, après avoir empilé la position de souris.

```
void  mouseReleaseEvent( QMouseEvent *e) {
  empiler(e->x(), e->y());
  paint->drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
}
```

Il ne reste plus qu'à définir la méthode `paintEvent` pour notre widget i.e. demander à refaire le dessin complètement.

```
void  paintEvent( QPaintEvent * ) {
  for (int i=0; i<sommet; i++)
```

```
    if (x[i]==-1) i++;
    else  paint->drawLine(x[i-1], y[i-1], x[i], y[i]);
}
```

Voilà donc le programme complet:

```
#include <qapp.h>
#include <qwidget.h>
#include <qpainter.h>

class dessin : public QWidget {
private:
  const int maxpts = 100;
  int x[maxpts], y[maxpts];  // Pour conserver toutes les coordonnées
                             // des positions successives de la soris
  int sommet;                // nombre de points mémorisés
  void empiler(int i, int j) {
    if (sommet < maxpts)
       x[sommet] = i; y[sommet++] = j;
  }
public:
  dessin( QWidget *parent=0, const char *name=0 ):QWidget(parent, name) {
    setBackgroundColor(white);
    sommet = 0;
  }
protected:
  void  mousePressEvent( QMouseEvent *e) {
    empiler(-1,-1);
    empiler(e->x(), e->y());
  }
  void  mouseReleaseEvent( QMouseEvent *e) {
    empiler(e->x(), e->y());
    paint->drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
  }
  void  mouseMoveEvent( QMouseEvent *e) {
    empiler(e->x(), e->y());
    paint->drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
  }
  void  paintEvent( QPaintEvent * ) {
    QPainter paint;
    paint.begin(this);
    paint.setPen( blue );
    for (int i=0; i<sommet; i++)
      if (x[i]==-1) i++;
      else  paint.drawLine(x[i-1], y[i-1], x[i], y[i]);
    paint.end();
  }
};

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

```
    a.setMainWidget( &w );
    w.show();
    w.adjustSize();
    return a.exec();
}
```

4.2 Redessiner tout ou pas ?

Comme nous l'avons dit plus haut, la méthode `QPaintEvent()` est utilisée chaque fois que les méthodes `update` ou `repaint` seront appelées; et elles le sont soit automatiquement par ***Qt*** soit explicitement par le programmeur.

Pourquoi deux méthodes pour faire la même chose? Eh bien, parce qu'elles ne font pas la même chose. En effet, la méthode `update` efface tout le contenu du widget et le redessine entièrement (en utilisant la méthode `QPaintEvent()`). Alors que la méthode `repaint` ne redessine qu'une portion précise du widget. Le prototype de ces deux fonctions sont :

```
void update ()
void repaint ( int x, int y, int w, int h, bool erase=TRUE)
```

Les arguments `x`, `y`, `w` (largeur) et `h` (hauteur) définissent la zone à redessiner. Si `w` (resp. `h`) est négatif, elle est remplacer par `width()-x` (resp. `height()-y`).

Dans beaucoup de cas, la méthode `update` est suffisante mais cette manière de faire peut conduire un effet de clignotement désagréable. Pour éviter ces clignotement, lorsque c'est possible, on pourra utiliser la méthode `repaint`.

Comme on s'en doute, la méthode `repaint` est généralement plus rapide que la méthode `update`, mais plusieurs appels de la méthode `update` peut, si les circonstances le permettent, générer qu'un seul appel à la méthode `QPaintEvent`.

> Si la méthode `repaint` est utilisée dans une autre méthode qui elle-même peut être appellée par la méthode `paintEvent`, cela peu conduire à une récursion infinie.
>
> Ceci ne se produit pas avec la méthode `update`

Voici un exemple qui permet de bien visualiser cet effet de clignotement dû à la méthode `update` et de l'amélioration obtenue par l'utilisation de la méthode `repaint`. Il s'agit d'un petit widget dont le fond est diversement coloré et qui contient la chaîne de caractère à son centre, chaîne qui change de couleur toutes les 500 ms.

On dispose d'un timer, de la chaîne à afficher, et d'une variable booléenne permettant de changer de couleur. Toutes les 500 ms, la méthode `changer` est appelée. Cette méthode appelle la méthode `update` pour rafraîchir le widget et en profiter pour changer la couleur de la chaîne de caractère.

```
// Fichier clignotement.h
#include <qapp.h>
#include <qwidget.h>
#include <qtimer.h>
#include <qpainter.h>

class cligno : public QWidget {
    Q_OBJECT
public:
    cligno(const char *text, QWidget *parent=0, const char *name=0 );
    ~cligno();
```

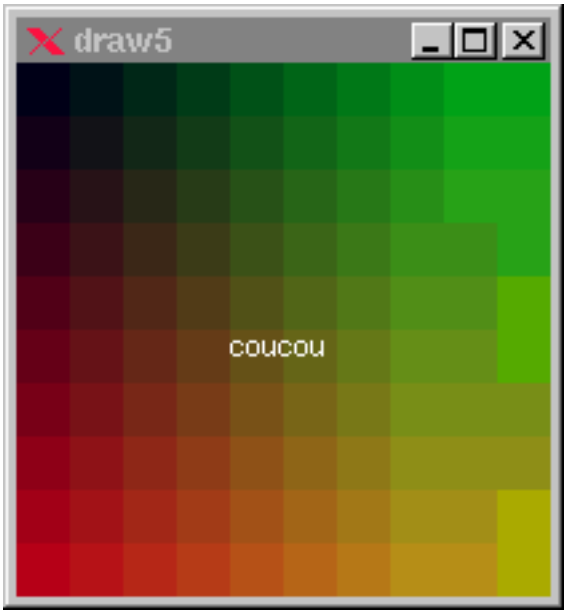

FIG. 4.1 – *Effet de clignotement*

```
signals:
    void clicked(void);
protected:
    void paintEvent( QPaintEvent * );
private slots:
    void changer(void);
private:
    const char *t;
    QTimer *timer;
    bool b;
};

// Fichier clignotement.cpp
#include "draw5.h"
#include "qcolor.h"

cligno::cligno( const char *text, QWidget *pere, const char *nom )
    : QWidget(pere, nom), t(text) {
  timer = new QTimer(this);
  connect( timer, SIGNAL(timeout()), SLOT(animate()) );
  timer->start( 500 );
  resize( 200, 100 );
  b = TRUE;
  setGeometry(100, 100, 200, 200);
}

cligno::~cligno() {
  delete timer;
```

```
}

void cligno::changer(void) {
  b = !b; update();
}

void cligno::paintEvent( QPaintEvent * e) {
  QPainter paint;
  paint.begin(this);
  for (int j=0; j<200; j+=20)
    for (int i=0; i<200; i+=20) {
      paint.setPen(QColor(i, j, 25));
      paint.setBrush(QColor(i, j, 25));
      paint.drawRect(j,i, 20, 20);
   }
  paint.setPen(b ? white : magenta);
  paint.drawText(80,110, "coucou");
  paint.end();
}


int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  cligno w("coucou");
  a.setMainWidget( &w );
  w.show();
  return a.exec();
}
```

A l'exécution de ce programme vous devriez avoir un désagéable effet de clignotement. Comme la seule chose qui change régulièrement, c'est la couleur de la chaîne de caractères, on va limiter le rafraîchissement à la seule zone où cette chaîne s'affiche. Pour cela, il suffit de remplacer la méthode `cligno::changer(void)`

```
void cligno::changer(void) {
  b = !b; repaint(80, 100, 40,10, FALSE);
}
```

Avec cette modification, le clignotement devrait cesser.

4.3 Tout redessiner sans clignotement

Il est évident qu'il existe des applications où il n'est pas possible de déterminer une petite zone du widget à mettre à jour. Est-on pour autant condamné à supporter ce clignotement dans de tels cas. Certes non ! Une technique connue sous le nom de *double buffering* est utilisée dans de telles situations. L'idée est la suivante: au lieu de dessiner directement à l'écran, on dispose d'un objet qui serait un équivalent de l'écran et sur lequel on dessinera. Une fois tout le dessin achevé, en bloc, on affichera tout le contenu de cet objet à l'écran. Cette approche a un inconvénient: la place mémoire supplémentaire requise; l'avantage que l'on y trouve est la disparition du clignotement dans certains types d'applications.

Ce fameur buffer dont on parle ici, ce n'est rien d'autre qu'un objet de la classe `QPixmap`. La techinque du *double buffering* consiste à

1. Créer un objet de la classe ayant la taille de widget.

   ```
   QPixmap  pm( size() );
   ```

2. Colorier cet objet avec la couleur du fond du widget.

```
pm.fill( backgroundColor() );
```

3. Colorier ou dessiner tout ce qui doit l'être sur cet objet. Si on reprend l'exemple précédant, on aura:

```
QPainter p;
paint.begin( &pm );
for (int j=0; j<200; j+=20)
  for (int i=0; i<200; i+=20) {
    paint.setPen(QColor(i, j, 25));
    paint.setBrush(QColor(i, j, 25));
    paint.drawRect(j,i, 20, 20);
    }
paint.setPen(b ? white : magenta);
paint.drawText(80,110, "coucou");
paint.end();
```

4. Recopier en bloc cet objet sur le widget avec la fonction `bitBlt`.

```
bitBlt( this, 0,0, &pm, 0, 0, width(), height(), CopyROP, FALSE );
```

Les prototypes de la fonction `bitBlt` sont les suivants

```
void bitBlt ( QPaintDevice *dst, int dx, int dy,
              const QPaintDevice *src, int sx, int sy, int sw, int sh,
              RasterOp rop, bool ignoreMask)
void bitBlt ( QPaintDevice *dst, const QPoint &dp,
              const QPaintDevice *src, const QRect &sr,
              RasterOp rop)
```

où

- `dst` est la chose sur laquelle on veut copier (ici notre widget qui est une sous classe de `QPaintDevice`)
- `dx` et `dy` constitue la position où l'on veut recopier
- `src` est la chose à copier (ici notre `QPixmap`qui est également une sous classe de `QPaintDevice`). Ceci suggère que l'on peut utiliser le double buffering que sur partie du widget si on le désire.
- `sx` et `sy` consitue une position dans `src`. Ceci suggère que l'on peut ne recopier qu'une partie de notre pixmap si nécessaire.
- `sw` et `sh` sont la largeur et la hauteur de ce qu'on veut copier
- `rop` est la transformation (*raster operation*) que l'on veut effecter au moment de la recopie. Il y a, en effet, la possibilité de d'utiliser les constantes suivantes:
  - CopyROP: dst = src.
  - OrROP: dst = dst OR src.
  - XorROP: dst = dst XOR src.
  - EraseROP: dst = (NOT src) AND dst
  - NotCopyROP: dst = NOT src
  - NotOrROP: dst = (NOT src) OR dst
  - NotXorROP: dst = (NOT src) XOR dst
  - NotEraseROP: dst = src AND dst
  - NotROP: dst = NOT dst

A terminer !!!!!!!!!

4.4 La classe `QPainter`

Chapitre 5

Créer des widgets

Sommaire

Ce chapitre manque beaucoup d'explication Détailler les différents widgets

5.1 Introduction

Qt possède plusieurs types de widgets:

`QPushButton`
: Nous avons déjà rencontré ce type de widget; il s'agit d'un bouton etiquetté. Un bouton émet le signal `clicked` lorsque l'utilisateur presse sur le bouton.

`QLCDNumber`
: Nous avons également déjà rencontré ce type de widget; il s'agit d'une zone permettant d'afficher de manière esthétique un nombre.

`QFrame`
: Ce type de widget a été utilisé dans l'application «animation». Il s'agit d'une zone vierge sur laquelle on peut inclure d'autres widgets.

`QRadioButton`
: Il s'agit d'une bouton à cocher etiquetté.

`QComboBox`
: Choix déroulant

`QListBox`
: Choix défilant

`QFileDialog`
: Dialogue pour l'ouverture de fichiers.

`QMessageBox`
: Un «message box» est une fenêtre de dialogue qui affiche un texte et un bouton.

`QLineEdit`
: Une zone de texte d'une ligne où l'utilisateur peut écrire.

`QMultiLineEdit` Un éditeur simple.

`QMenuBar`
: Barre de menu horizontale

`QPopupMenu`
: Sous menu.

`QScrollBar`
: Ascenceur vertival ou horizontal

`QSlider`
: Glisière horizontal ou vertical

`QButtonGroup`
: Regroupe des `QButton` en groupes.

`QLabel`
: Permet d'afficher un texte static ou une image.

Il existe encore bien d'autres widgets (voir documentation officielle de Qt).

5.2 La classe QPushButton

5.3 La classe QRadioButton

5.4 La classe QCheckBox

5.5 La classe QComboBox

5.6 La classe QGroupButton

5.7 La classe QFrame

5.8 La classe QListBox

5.9 La classe QSlider

5.10 La classe QScrollBar

5.11 La classe QLabel

5.12 La classe Qmenu

5.13 La classe QPopupMenu

5.14 La classe QTabDialog

5.15 La classe QMessageBox

5.16 QFileDialog

5.17 Quelques widgets

Nous allons à présent créer une application possédant les widgets suivants:

- QFrame
- QLabel
- QPushButton, QCheckButton et QRadioButton
- QLabel
- QListBox et QComboBox
- QSlider et QScrollBar
- Qmenu et QPopupMenu
- QFileDialog

Cette application se contente d'afficher l'action effectuée par l'utilisateur dans un `QLabel`. On va donc associer des slots aux siganux émis par les widgets. Il faudra donc utiliser l'utilitaire `mocr` la classe `PleinDeWidgets` que nous allons définir.

```
// ***************************
// ********  PleinDeWidgets.h
// ***************************
#include <qdialog.h>
#include <qmsgbox.h>
#include <qpixmap.h>
#include <qapp.h>
#include <qbttngrp.h>
#include <qchkbox.h>
#include <qcombo.h>
#include <qframe.h>
#include <qgrpbox.h>
#include <qlabel.h>
#include <qlcdnum.h>
#include <qlined.h>
#include <qlistbox.h>
#include <qpushbt.h>
#include <qradiobt.h>
#include <qslider.h>
#include <qscrbar.h>
#include <qtooltip.h>
#include <qdialog.h>
#include <qfiledlg.h>
#include <qtabdlg.h>
#include <qpopmenu.h>
#include <qkeycode.h>
#include <qmenubar.h>

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

Voici à présent les slots qui se contente d'afficher l'action réalisée

```
private slots:
    void clicBouton1();
    void clicBouton2();
    void ChoixCheckBox( int );
    void ChoixRadioButton( int );
    void ValeurModifieeSlider( int );
    void ValeurModifieeScroll( int );
    void ChoixItemListBox( int );
    void ChoixItemComboBox( int );
    void ChoixItemEditableComboBox( const char * );
    void TextModifieeLineEdit( const char * );
    void PleinDeWidgets::printer();
    void PleinDeWidgets::file();
```

```
    void PleinDeWidgets::fax();
    void PleinDeWidgets::printerSetup();
    void PleinDeWidgets::open();
    void PleinDeWidgets::save();
    void PleinDeWidgets::closeDoc();
    void PleinDeWidgets::undo();
    void PleinDeWidgets::redo();
    void PleinDeWidgets::normal();
    void PleinDeWidgets::bold();
    void PleinDeWidgets::underline();
    void PleinDeWidgets::about();
```

Et enfin les champs privés qui sont constitués des différents widgets de l'application

```
private:
    QPixmap pm;
    QLabel     *msg;
    QCheckBox  *cb[3];
    QPushButton *pb1, *pb2, *quitButton;
    QButtonGroup *bg ;
    QRadioButton *rb1, *rb2, *rb3;
    QListBox *lb ;
    QSlider *sb;
    QScrollBar *scb;
    QComboBox *combo ;
    QComboBox *edCombo;
    QLineEdit *le ;
    QLabel *msgLabel ;
    QFrame *separator;
    QMenuBar *menu;
    QPopupMenu *m_print, *m_file, *m_edit, *m_options,  *m_help;

    bool isBold;
    bool isUnderline;
    int boldID, underlineID;
    };

// ***************************
// ********  PleinDeWidgets.cpp
// ***************************

PleinDeWidgets::PleinDeWidgets(QWidget *pere=0, const char *nom=0)
                               : QWidget(pere, nom)
    {
    «création bouton1, button2 et bouton quit (QPushButton)»
    «création groupe de radio boutons (QButtonGroup et QCheckBox)»
    «création groupe de radio boutons (QButtonGroup et QRadioBox)»
    «création choix déroulant (QComboBox)»
    «création choix défilant  (QListBox)»
    «création glisière (QSlider)»
    «création ascenseur (QScrollBar)»
    «Création de l'éditeur ligne (QLineEdit)»
    «création d'un separateur»
```

```
«création de la zone de texte static (QLabel)»
«création de la barre de menu et des menus associés (QMenuBar et QPopupMenu)»
}
```

Creation bouton1, button2 et bouton quit (`QPushButton`)

```
pb1 = new QPushButton( this, "button1" );
pb1->setText( "Push button 1" );
pb1->setGeometry( 10,50, 120,30 );
connect(pb1, SIGNAL(clicked()), SLOT(clicBouton1()) );
QToolTip::add(pb1, "push button 1" );

pb2 = new QPushButton( this, "button2" );
bool pix = TRUE;
if ( pm.load("qt.bmp") )
    pb2->setPixmap( pm );
else

    QMessageBox::message( "Error", "Could not load qt.bmp pixmap" );
    pb2->setText( "No pixmap" );
    pix = FALSE;

pb2->setGeometry( 150,30, 60,60 );
connect( pb2, SIGNAL(clicked()), SLOT(clicBouton2()) );
QToolTip::add(pb2, "push button 2" );

quitButton = new QPushButton( this, "quitButton" );
quitButton->setText( "Quit" );
quitButton->setGeometry( 300,50, 80,30 );
connect( quitButton, SIGNAL(clicked()), qApp, SLOT(quit()) );
QToolTip::add( quitButton, "quit the application" );
```

Creation groupe de radio boutons (`QButtonGroup` et `QCheckBox`)

```
bg = new QButtonGroup( this, "checkGroup" );
bg->setTitle( "Check Boxes" );
cb[0] = new QCheckBox( bg );
cb[0]->setText( "Read" );
cb[0]->setGeometry( 10,15, 100,25 );
cb[1] = new QCheckBox( bg );
cb[1]->setText( "Write" );
cb[1]->setGeometry( 10,45, 100,25 );
cb[2] = new QCheckBox( bg );
cb[2]->setText( "Execute" );
cb[2]->setGeometry( 10,75, 100,30 );
bg->setGeometry( 10,100, 120,110 );
connect( bg, SIGNAL(clicked(int)), SLOT(ChoixCheckBox(int)) );
QToolTip::add( cb[0], "check box 1" );
QToolTip::add( cb[1], "check box 2" );
QToolTip::add( cb[2], "check box 3" );
```

Creation groupe de radio boutons (`QButtonGroup` et `QRadioButton`)

```
bg = new QButtonGroup( this, "radioGroup" );
bg->setTitle( "Radio buttons" );
rb1 = new QRadioButton( bg );
rb1->setText( "AM" );
rb1->setGeometry( 10, 15, 100,25 );
QToolTip::add( rb1, "radio button 1" );
rb2 = new QRadioButton( bg );
rb2->setText( "FM" );
rb2->setGeometry( 10, 45, 100,25 );
QToolTip::add( rb2, "radio button 2" );
rb3 = new QRadioButton( bg );
rb3->setText( "Short Wave" );
rb3->setGeometry( 10, 75, 100,25 );
bg->setGeometry( 140, 100, 120, 110 );
connect( bg, SIGNAL(clicked(int)), SLOT(ChoixRadioButton(int)) );
QToolTip::add( rb3, "radio button 3" );
```

Creation choix déroulant (`QComboBox`)

```
combo = new QComboBox( FALSE, this, "comboBox" );
combo->insertItem( "True" );
combo->insertItem( "False" );
combo->insertItem( "Maybe" );
combo->insertItem( "Certainly" );
combo->insertItem( "Nope" );
combo->setGeometry( 140, 270, 120, 30 );
connect( combo, SIGNAL(activated(int)), SLOT(ChoixItemComboBox(int)) );
QToolTip::add( combo, "read-only combo box" );

edCombo = new QComboBox( TRUE, this, "edComboBox" );
edCombo->insertItem( "Permutable" );
edCombo->insertItem( "Malleable" );
edCombo->insertItem( "Adaptable" );
edCombo->insertItem( "Alterable" );
edCombo->insertItem( "Inconstant" );
edCombo->setGeometry( 140, 310, 120, 30 );
connect( edCombo, SIGNAL(activated(const char *)),
         SLOT(ChoixItemEditableComboBox(const char *)) );
QToolTip::add( edCombo, "editable combo box" );
```

Creation choix défilant (`QListBox`)

```
lb = new QListBox( this, "listBox" );
for ( int i=0; i<100; i++ )                // remplir la list box
    QString str;
    str.sprintf( "line %d", i );
    if ( i == 42 && pix )
        lb->insertItem( pm );
    else
```

```
        lb->insertItem( str );

    lb->setGeometry( 10, 220, 120, 160 );
    int lbIH = lb->itemHeight();
    lb->resize( lb->width(), ((lb->height()+lbIH)/lbIH)*lbIH +
                lb->frameWidth()*2 );
    connect( lb, SIGNAL(selected(int)), SLOT(ChoixItemListBox(int)) );
    QToolTip::add( lb, "list box" );
```

Creation glisière (QSlider)

```
    sb = new QSlider( QSlider::Horizontal,this,"Slider" );
    sb->setTickmarks( QSlider::Below );
    sb->setGeometry( 140, 220, 120, sb->sizeHint().height() );
    connect( sb, SIGNAL(valueChanged(int)), SLOT(ValeurModifieeSlider(int)) );
    QToolTip::add( sb, "slider" );
```

Creation ascenseur (QScrollBar)

```
    scb = new QScrollBar( 0, 99,                    // plage de valeurs
                          1, 10,                    // saut ligne/page
                          0,                        // valuer initiale
                          QScrollBar::Vertical,     // orientation
                          this, "scrollbar" );
    scb->setGeometry( 320, 120, 15, 150 );
    connect( scb, SIGNAL(valueChanged(int)), SLOT(ValeurModifieeScroll(int)) );
    QToolTip::add( sb, "slider" );
```

Creation de l'éditeur ligne (QLineEdit)

```
    le = new QLineEdit( this, "lineEdit" );
    le->setGeometry( 140, 360, 230, 25 );
    connect( le, SIGNAL(textChanged(const char *)),
                 SLOT(TextModifieeLineEdit(const char *)) );
    QToolTip::add( le, "single line editor" );
```

Creation de la zone de texte static (QLabel)

```
    msgLabel = new QLabel( this, "msgLabel" );
    msgLabel->setText( "Message:" );
    msgLabel->setAlignment( AlignRight|AlignVCenter );
    msgLabel->setGeometry( 10,420, 85, 30 );
    QToolTip::add( msgLabel, "label 1" );

    msg = new QLabel( this, "message" );
    msg->setFrameStyle( QFrame::Panel | QFrame::Sunken );
    msg->setAlignment( AlignCenter );
    msg->setGeometry( 100,420, 300,30 );
    msg->setFont( QFont("times",12,QFont::Bold) );
    QToolTip::add( msg, "label 2" );
```

Creation d'un separateur (`QFrame`)

```
separator = new QFrame( this, "separatorLine" );
separator->setFrameStyle( QFrame::HLine | QFrame::Sunken );
separator->setGeometry( 5, 410, 440, 4 );
QToolTip::add( separator, "tool tips on a separator! wow!" );
```

Creation de la barre de menu et des sous menus (`QMenuBar` et `QPopupMenu`)

```
m_print = new QPopupMenu;
CHECK_PTR( m_print );
m_print->insertItem( "Print to printer", this, SLOT(printer()) );
m_print->insertItem( "Print to file", this, SLOT(file()) );
m_print->insertItem( "Print to fax", this, SLOT(fax()) );
m_print->insertSeparator();
m_print->insertItem( "Printer Setup", this, SLOT(printerSetup()) );

m_file = new QPopupMenu();
CHECK_PTR( m_file );
m_file->insertItem( "Open",  this, SLOT(open()), CTRL+Key_O );
m_file->insertItem( "Save", this, SLOT(save()), CTRL+Key_S );
m_file->insertItem( "Close", this, SLOT(closeDoc()), CTRL+Key_W );
m_file->insertSeparator();
m_file->insertItem( "Print", m_print, CTRL+Key_P );
m_file->insertSeparator();
m_file->insertItem( "Exit",  qApp, SLOT(quit()), CTRL+Key_Q );

m_edit = new QPopupMenu();
CHECK_PTR( m_edit );
int undoID = m_edit->insertItem( "Undo", this, SLOT(undo()) );
int redoID = m_edit->insertItem( "Redo", this, SLOT(redo()) );
m_edit->setItemEnabled( undoID, FALSE );
m_edit->setItemEnabled( redoID, FALSE );

m_options = new QPopupMenu();
CHECK_PTR( m_options );
m_options->insertItem( "Normal Font", this, SLOT(normal()) );
m_options->insertSeparator();
boldID = m_options->insertItem( "Bold", this, SLOT(bold()) );
underlineID = m_options->insertItem( "Underline", this, SLOT(underline()) );
isBold = FALSE;
isUnderline = FALSE;
m_options->setCheckable( TRUE );

m_help = new QPopupMenu;
CHECK_PTR( m_help );
m_help->insertItem( "About", this, SLOT(about()), CTRL+Key_H );



menu = new QMenuBar( this );
```

```
    CHECK_PTR( menu );
    menu->insertItem( "&File", m_file );
    menu->insertItem( "&Edit", m_edit );
    menu->insertItem( "&Options", m_options );
    menu->insertSeparator();
    menu->insertItem( "&Help", m_help );
```

Définition des slots

```
void PleinDeWidgets::clicBouton1()
    {
    msg->setText( "Clic sur le boutton 1" );
    }

void PleinDeWidgets::clicBouton2()
    {
    msg->setText( "Clic sur le boutton 2" );
    }

void PleinDeWidgets::ChoixCheckBox( int id)
    {
    QString str;
    str.sprintf( "Clicked sur Check box %d : ", id );
    QString chk = "---";
    if ( cb[0]->isChecked() )
        chk[0] = 'r';
    if ( cb[1]->isChecked() )
        chk[1] = 'w';
    if ( cb[2]->isChecked() )
        chk[2] = 'x';
    str += chk;
    msg->setText( str );
    }

void PleinDeWidgets::ChoixRadioButton( int id)
    {
    QString str;
    str.sprintf( "Clic sur Radio button #%d ", id );
    msg->setText( str );
    }

void PleinDeWidgets::ValeurModifieeSlider( int value)
    {
    QString str;
    str.sprintf( "Nouvelle valeur Slider %d", value );
    msg->setText( str );
    }

void PleinDeWidgets::ValeurModifieeScroll( int value)
    {
    QString str;
    str.sprintf( "Nouvelle valeur Scroll %d", value );
```

```
    msg->setText( str );
    }

void PleinDeWidgets::ChoixItemListBox( int index)
    {
    QString str;
    str.sprintf( "Choix Item List box  %d", index );
    msg->setText( str );
    }

void PleinDeWidgets::ChoixItemComboBox( int index)
    {
    QString str;
    str.sprintf( "Choix Item Combo box %d", index );
    msg->setText( str );
    }

void PleinDeWidgets::ChoixItemEditableComboBox( const char * text )
    {
    QString str;
    str.sprintf( "Choix dans Combo Box editable : %s", text );
    msg->setText( str );
    }

void PleinDeWidgets::TextModifieeLineEdit( const char * newText)
    {
    QString str;
    str.resize( strlen(newText)+100 );
    str.sprintf( "Texte dans Line edit : %s", newText );
    msg->setText( str );
    }

void PleinDeWidgets::printer()
    {
    msg->setText( "Menu File->Print->Printer"  );
    }
void PleinDeWidgets::file()
    {
    msg->setText( "Menu File->Print->File"  );
    }
void PleinDeWidgets::fax()
    {
    msg->setText( "Menu File->Print->Fax"  );
    }
void PleinDeWidgets::printerSetup()
    {
    msg->setText( "Menu File->Print->PrinterSetup"  );
    }
void PleinDeWidgets::open()
    {
    msg->setText( "Menu File->open"  );
    }
void PleinDeWidgets::save()
```

```
    {
    msg->setText( "Menu File->save"  );
    }
void PleinDeWidgets::closeDoc()
    {
    msg->setText( "Menu File->close"  );
    }
void PleinDeWidgets::undo()
    {
    msg->setText( "Menu Edit->undo"  );
    }
void PleinDeWidgets::redo()
    {
    msg->setText( "Menu Edit->redo"  );
    }
void PleinDeWidgets::normal()
    {
    msg->setText( "Menu Options->normal"  );
    }
void PleinDeWidgets::bold()
    {
    msg->setText( "Menu Options->bold"  );
    }
void PleinDeWidgets::underline()
    {
    msg->setText( "Menu Options->underline"  );
    }
void PleinDeWidgets::about()
    {
    msg->setText( "Menu Help->About"  );
    }
```

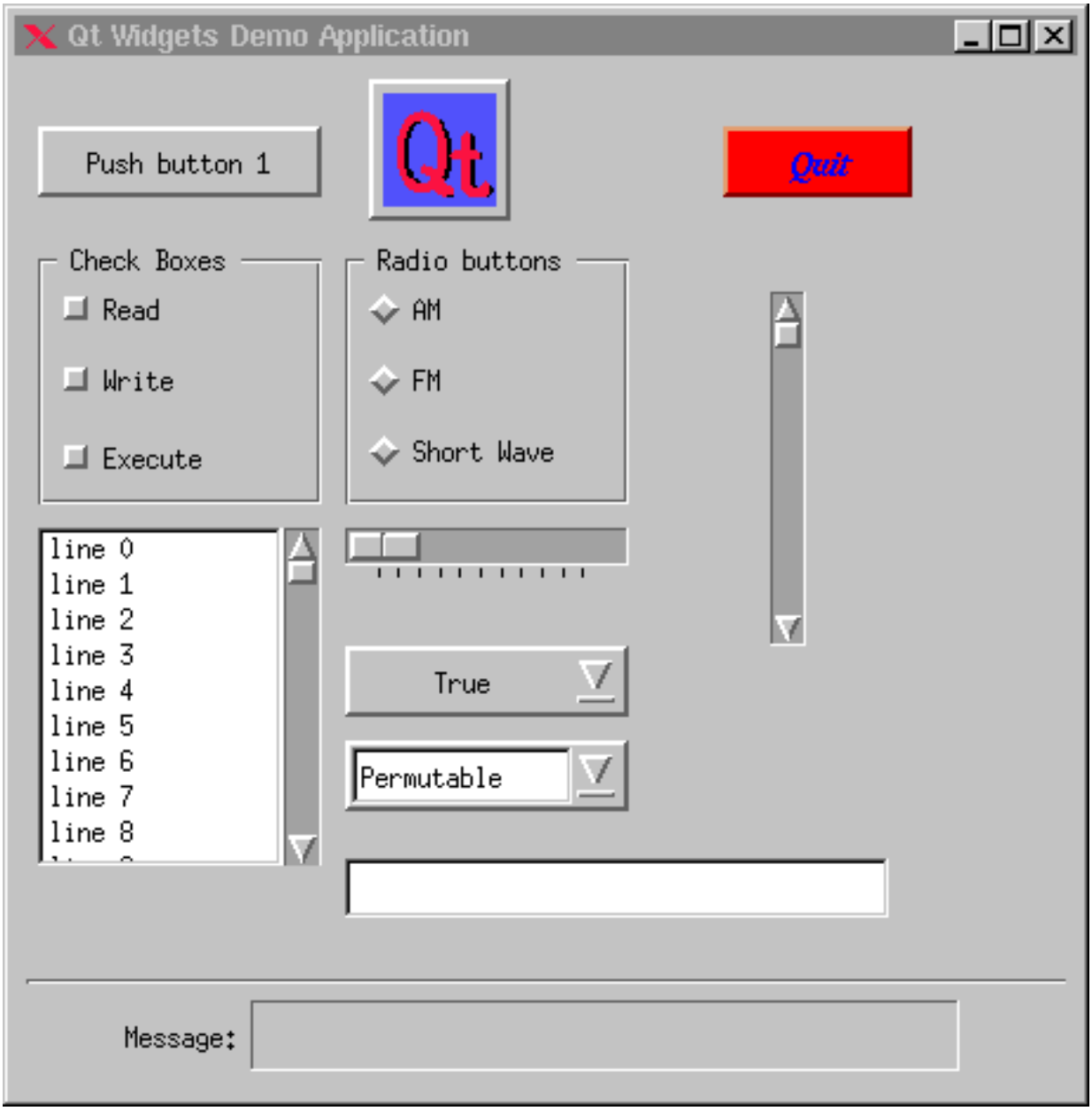

FIG. 5.1 – *Quelques widgets*

Chapitre 6

Ranger les widgets

Sommaire

La disposition géométrique des widgets est parfois fastidieux; il faut calculer « à la main » la taille des widgets, leur position, leur redimensionnement, etc. Pour simplifier cette programmation, ***Qt*** offre la possibilité de spécifier la disposition globale que l'on désire obtenir et laisser le soin à ***Qt*** de gérer leur position et leur redimensionnement. Pour ce faire, ***Qt*** définit une classe abstraite `QLayout` de laquelle se dérive deux sous classes: `QBoxLayout` et `QGridLayout`.

6.1 La classe QBoxLayout

Cette classe permet de ranger, dans un widget, côte à côte, horizontalement ou verticalement, les différents sous widgets. Les sous widgets sont placés et redimensionner de telle sorte que toute la place prise par le widget père soit remplie.

Pour mettre en œuvre ceci, il faut créeer un objet de type `QBoxLayout` qui possède un constructeur:

```
QBoxLayout (QWidget* pere, Direction d, int bordure=0,
            int autoBordure = -1, const char* nom=0);
```

Ce constructeur crée un nouveau `QBoxLayout` de Direction `d` pour le widget `pere`. Le wdiget `pere` ne doit pas être `NULL`. L'argument `boudure` définit le nombre de pixels qui séparent le bord du widget père des widgets fils. L'argument `autoBoudure` définit le nombre de pixels qui séparent les widgets fils entre eux; si sa valeur est -1, c'est la valeur de `bordure` qui est choisie.

Les constantes `LeftToRight`, `RightToLeft`, `TopToBottom` and `BottomToTop` définissent la direction souhaitée: les deux premières étant horizontales et les deux dernières verticales. Par exemple, avec `BottomToTop`, les sous widgets seront disposés de haut en bas, suivant l'ordre dans lequel se font leur ajout par la méthode `addWidget`.

La méthode

```
void QBoxLayout::addWidget ( QWidget *widget, int stretch = 0,
                             int align = AlignCenter)
```

ajoute `widget` dans l'objet `QBoxLayout` sur lequel cette méthode s'applique. La valeur de `stretch` permet de définir une relation relation d'ordre sur l'élasticité des sous widgets, les uns par rapport aux autres jusqu'à attendre la taille maximum (si elle est epécifiée). Le paramètre `align` sert évidement à définir l'alignment du

sous widget. Les valeurs permises sont: `AlignCenter`, `AlignTop` et `AlignBottom` pour une disposition horizontale et `AlignCenter`, centers horizontally in the box. `AlignLeft` et `AlignRight` pour une verticale.

```
#include <qapp.h>
#include <qwidget.h>
#include <qlabel.h>
#include <qlayout.h>

class box : public QWidget {
public:
  box(QWidget *pere=0, const char *nom=0) : QWidget(pere, nom) {

    QBoxLayout *bl = new QBoxLayout(this, QBoxLayout::BottomToTop, 5 , 10);

    l1 = new QLabel(this);
    l1->setText("Je suis le widget 1");
    l1->setAlignment(AlignCenter);
    l1->setBackgroundColor(red);
    bl -> addWidget( l1, 1 );

    l2 = new QLabel(this);
    l2->setText("Je suis le widget 2");
    l2->setAlignment(AlignCenter);
    l2->setBackgroundColor(yellow);
    bl -> addWidget( l2, 2 );

    l3 = new QLabel(this);
    l3->setText("Je suis le widget 3");
    l3->setAlignment(AlignCenter);
    l3->setBackgroundColor(magenta);
    bl -> addWidget( l3, 3 );

  }
private:
  QLabel     *l1;
  QLabel     *l2;
  QLabel     *l3;
};



int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  box w;
  a.setMainWidget( &w );
  w.show();
  w.adjustSize();
  return a.exec();
}
```

FIG. 6.1 – *BoxLayout*

6.2 La classe QGridLayout

Cette classe permet de ranger, dans un widget, les différents sous widgets dans une configuration de gille. Les sous widgets sont placés et redimensionner de telle sorte que toute la place prise par le widget père soit remplie.

Comme plus haut, pour mettre en œuvre ceci, il faut créeer un objet de type `QGridLayout` qui possède un constructeur:

```
QGridLayout (QWidget* parent, int nRows, int nCols, int bordure=0,
             int autoBordure = -1,const char* name=0)
```

Ce constructeur crée un nouveau `QGridLayout` de`nRows` lignes, de `cCols` colonnes pour le widget `pere`. Le wdiget `pere` ne doit pas être `NULL`. L'argument `boudure` définit le nombre de pixels qui séparent le bord du widget père les widgets fils. L'argument `autoBoudure` définit le nombre de pixels qui séparent les widgets fils entre eux; si sa valeur est -1, c'est la valeur de `bordure` qui est choisie.

La méthode

```
void QGridLayout::addWidget ( QWidget *w, int row, int col, int align = 0)
```

ajoute `widget` dans l'objet `QBoxLayout` sur lequel cette méthode s'applique. Les valeurs `row` et `col` définissent l'emplacement du sous widget. Le parmère `align` sert évidement à définir l'alignment du sous widget. Les valeurs permises sont: `AlignLeft`, `AlignRight`, `AlignHCenter`, `AlignTop`, `AlignBottom`, `AlignVCenter`, `AlignCenter`, `ExpandTabs` et `WordBreak` comme pour les alignements de `QLabel`.

`AlignCenter`, `AlignTop` et `AlignBottom` pour une disposition horizontale et `AlignCenter`, centers horizontally in the box. `AlignLeft` et `AlignRight` pour une verticale.

Les methodes

```
void QGridLayout::setColStretch ( int col, int stretch)
void QGridLayout::setRowStretch ( int row, int stretch)
```

permettent de spécifier l'élasticité relative dans une colonne donnée (resp. dans une ligne donnée) des sous widgets.

La méthode `QGridLayout`permet de fisionner des cases pour n'en faire qu'une.

```
void addMultiCellWidget (QWidget*, int fromRow, int toRow,
                         int fromCol, int toCol, int align = 0)
```

Lorsqu'on ajoute un sous widget avec la méthode

```
gl -> addMultiCellWidget( l12, 2, 4, 5, 6 );
```

celui-ci couvrira les cases 2.5, 2.6, 3.5, 3.6, 4.5, 4.6.

```
#include <qapp.h>
#include <qwidget.h>
#include <qlabel.h>
#include <qlayout.h>
#include <stdio.h>

class evenement : public QWidget {
public:
  evenement(QWidget *pere=0, const char *nom=0) : QWidget(pere, nom) {

    gl = new QGridLayout( this, 2, 3, 20 , 10);

    l11 = new QLabel(this);
    l11->setText("Je suis le widget 1.1");
    l11->setAlignment(AlignCenter);
    l11->setBackgroundColor(red);
    gl -> addWidget( l11, 0, 0);

    l2 = new QLabel(this);
    l2->setText("Je suis le widget 1.2");
    l2->setAlignment(AlignCenter);
    l2->setBackgroundColor(yellow);
    gl -> addMultiCellWidget( l2, 0, 1, 1, 1 );

    l13 = new QLabel(this);
    l13->setText("Je suis le widget 1.3");
    l13->setAlignment(AlignCenter);
    l13->setBackgroundColor(magenta);
    gl -> addWidget( l13, 0, 2 );
    gl -> setRowStretch(0, 2);

    l21 = new QLabel(this);
    l21->setText("Je suis le widget 2.1");
    l21->setAlignment(AlignCenter);
    l21->setBackgroundColor(white);
    gl -> addWidget( l21, 1, 0);

    l23 = new QLabel(this);
    l23->setText("Je suis le widget 2.3");
    l23->setAlignment(AlignCenter);
    l23->setBackgroundColor(gray);
    gl -> addWidget( l23, 1, 2 );
    gl -> setRowStretch(1, 1);
  }
private:
  QLabel     *l11;
  QLabel     *l2;
```

```
  QLabel     *l13;
  QLabel     *l21;
  QLabel     *l23;
  QGridLayout *gl;
};


int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  evenement w;
  a.setMainWidget( &w );
  w.show();
  w.adjustSize();
  return a.exec();
}
```

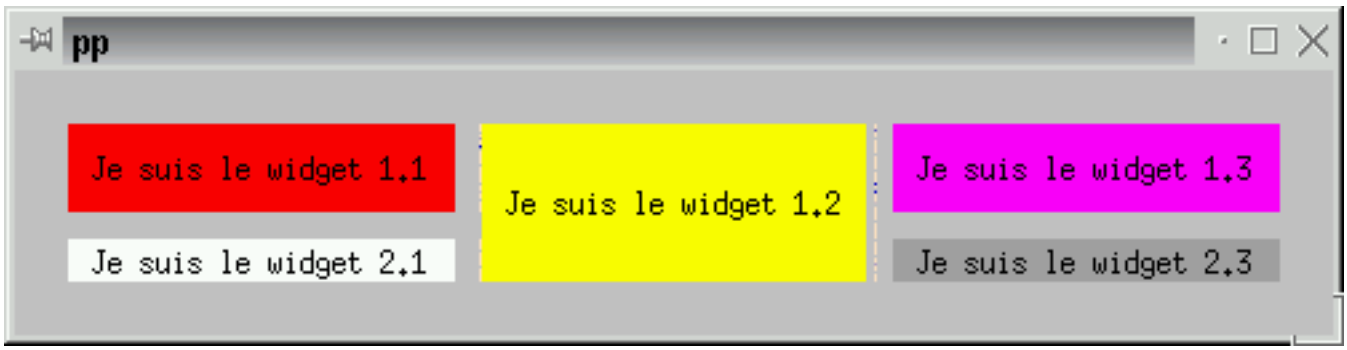

FIG. 6.2 – *GridLayout*

6.3 Des layouts dans des layouts

Il est possible d'inclure des layouts dans des layouts à l'aide de la méthode `addLayout` des classes `QGridLayout` et `QBoxLayout` :

```
void addWidget (QWidget*, int stretch = 0, int alignment = AlignCenter)
void addLayout (QLayout* layout, int row, int col)
```

selon que celui que l'on ajoute dans dans une `QBoxLayout` ou une `QGridLayout`.

```
#include <qwidget.h>
#include <qlabel.h>
#include <qlayout.h>
#include <stdio.h>

class evenement : public QWidget {
public:
  evenement(QWidget *pere=0, const char *nom=0) : QWidget(pere, nom) {
    QGridLayout *gl = new QGridLayout( this, 2, 3, 20 , 10);
```

```
QLabel *l11 = new QLabel(this);
l11->setText("Je suis le widget 1.1");
l11->setAlignment(AlignCenter);
l11->setBackgroundColor(red);
gl -> addWidget( l11, 0, 0);

QLabel *l13 = new QLabel(this);
l13->setText("Je suis le widget 1.3");
l13->setAlignment(AlignCenter);
l13->setBackgroundColor(magenta);
gl -> addWidget( l13, 0, 2 );
gl -> setRowStretch(0, 2);

QLabel *l21 = new QLabel(this);
l21->setText("Je suis le widget 2.1");
l21->setAlignment(AlignCenter);
l21->setBackgroundColor(white);
gl -> addWidget( l21, 1, 0);

QLabel *l23 = new QLabel(this);
l23->setText("Je suis le widget 2.3");
l23->setAlignment(AlignCenter);
l23->setBackgroundColor(gray);
gl -> addWidget( l23, 1, 2 );
gl -> setRowStretch(1, 1);


QBoxLayout *bl = new QBoxLayout(QBoxLayout::LeftToRight, 10);
gl->addLayout(bl, 1,1);

QLabel *l2;
for (int i = 0; i<3; i++)
  l2 = new QLabel(this);
  l2->setText("B/G");
  l2->setAlignment(AlignCenter);
  l2->setBackgroundColor(yellow);
  bl -> addWidget( l2 );



QGridLayout *gll = new QGridLayout(2,2, 10);
gl->addLayout(gll, 0,1);
for (int i = 0; i<2; i++) {
  for (int j = 0; j<2; j++) {
    l2 = new QLabel(this);
    l2->setText("G/G");
    l2->setAlignment(AlignCenter|WordBreak);
    l2->setBackgroundColor(green);
    gll -> addWidget( l2,i,j );
    }
}
```

```
    QGridLayout *glll = new QGridLayout(3,1, 10);
    bl->addLayout(glll);
    for (int i = 0; i<3; i++) {
      l2 = new QLabel(this);
      l2->setText("G/B/G");
      l2->setAlignment(AlignCenter);
      l2->setBackgroundColor(darkGreen);
      glll -> addWidget( l2, i, 0 );
    }
  }
};

int main( int argc, char **argv ) {
  QApplication a( argc, argv );
  evenement w;
  a.setMainWidget( &w );
  w.show();
  w.adjustSize();
  return a.exec();
}
```

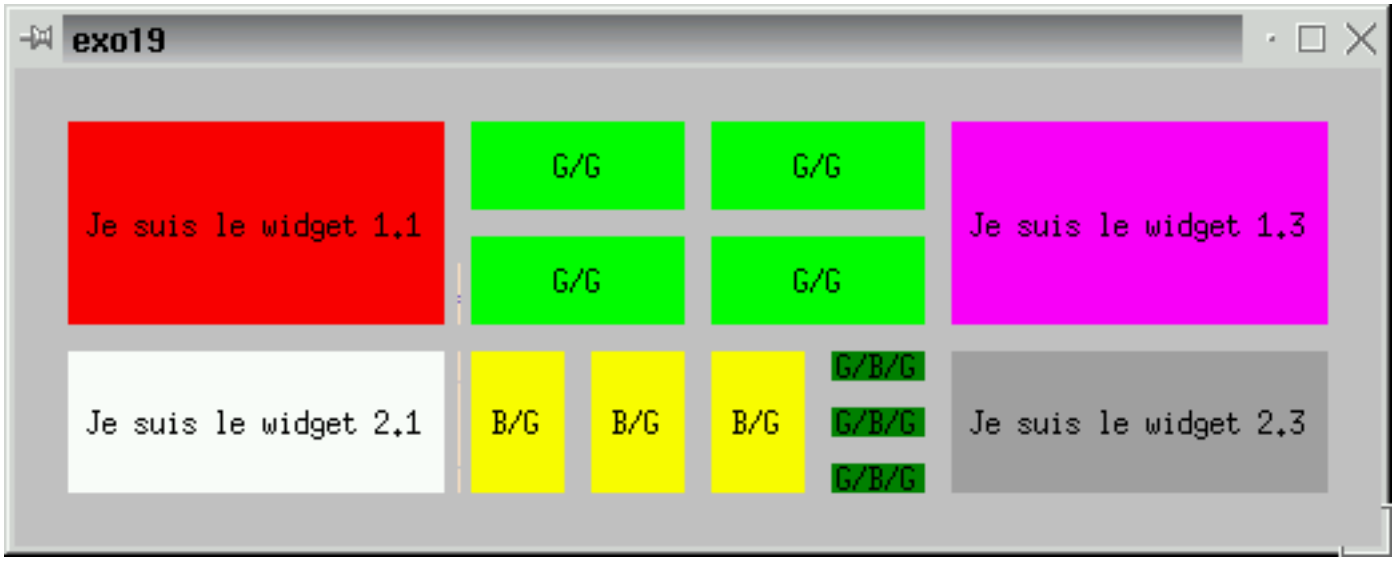

FIG. 6.3 – *Des layout dans des layouts*

Chapitre 7

A propos des fontes

Sommaire

La classe `QFont` permet de la manipulations des fontes disponibles sur le système utilisé. La gestion des fonts est une opération complexe car ***Qt*** ignore les fontes instalées sur le système. En fait, ***Qt*** interroge le système pour savoir si les fontes demandées sont disponibles et dans la négative, ***Qt*** recherche des fontes voisines de celles demandées.

Description de la méthode matching

Lors de la recherche d'une fonte voisine, ***Qt*** examine les attributs des fontes dans l'ordre suivant:

1. le type de fontes
2. le *pitch* (taille fixe ou pas)
3. la taille
4. le poids (léger, normal, semi-gras, gras, noir)
5. l'angle (italic ou pas)

7.1 La classe `QFont`

La classe `QFont` permet de définir les fontes à utilser lorsqu'on dessine du texte. Les attributs des fontes sont: la *famille*, la *taille*, le *poids* et l'*angle*.

Comme pour n'importe quel dessin à réaliser, on créera un objet de la classe `QPainter`et on utlisera les méthodes habitulles:

```
QPainter paint;
paint.begin(this);
// ... dessiner
paint.end();
```

Si l'on veut, à présent, dessiner le texte `coucou` avec la fonte *times*, on écrira:

```
paint.setFont( QFont("times") );
paint.drawText(x, y, "coucou");
```

La taille, le poids et l'angle est ceux définis par défaut par ***Qt***. On pourra interroger ces paramètres grace à la méthode `fontInfo` de la classe `QPainter`.

Si l'on veut, à présent, dessiner le texte `coucou` avec la fonte *lucida*, en 36 pts, en gras et en italic, on écrira:

```
paint.setFont( QFont( "lucida", 36, QFont::Bold, TRUE ) );
paint.drawText( 10, 320, "Text4" );
```

Le protoype des constructeurs de la classe `QFont` sont:

```
QFont ()
QFont (const QFont&)
QFont (const char* family, int pointSize = 12,
       int weight = Normal, bool italic = FALSE)
```

Le premier constructeur crée un objet de type `QFont` ayant les mêmes caractéristiques que la fonte par défaut. Quant au deuxième, il s'agit du constructeur de copie; il crée une fonte par recopie de celle passé en argument. Enfin, le dernier, le plus général, crée un objet de type ayant les caractéristiques données en argument : `family`, `pointSize`, `weight` et `italic`.

Les méthodes `setFamily`, `setPointSize`, `setWeight` and `setItalic` permettent de modifier les attributs des fontes déjà créés. Les méthodes `family`, `pointSize`, `weight` and `italic` permettent de récupérer attributs des fontes.

Chapitre 8

A propos des fontes

Sommaire

Chapitre 9

Découvrir certaines classes utiles

Sommaire

Ce chapitre manque d'exemples d'utilisation et de petites applications Donner un exemple qui utilise un tout ca.

Qt dispose également d'un ensemble de classes pour faciliter la gestion des objets non graphiques. Nous ne ferons pas le tour exhaustif de ces classes; seuls quelques unes sont présentées et de manière incomplète. Reporter vous à la documentation officielle de ***Qt*** pour découvrir le reste.

9.1 Les expressions régulières

9.1.1 Généralités sur les expressions régulières

Les expressions régulières servent à décrire des ensembles de motifs (ou de mots) et sont utilisées par un certain nombre d'utilitaires standard d'UNIX (`ed, sed, csplit, expr, more, pg ...`) et par divers éditeurs de textes *vi*, *emacs*, *xcoral*, etc. Les expressions régulières ne sont propres au système UNIX, on le trouve dans la quasi-totalité des systèmes.

Pour pouvoir décrire des ensembles de motifs, on donnera un sens particulier à un certain nombre de caractères. Avant de décrire la syntaxe complète des expressions régulières, voici un premier exemple:

```
^[^a-z].*[A-Z0-9]\.$
```

Cette expression régulière désigne les lignes commençant par autre chose qu'une lettre minuscule et se terminant par un lettre majuscule ou un chiffre suivi d'un point. Dans cet exemple, ce sont les caractères, `^, [, -, ], ., *` qui permettent de donner un sens particulier à cette notation.

Les caractères spéciaux.

Voici, à présent la liste des caractères qui ont une interprétation précise dans une expression régulière.

L'ensemble de caractères. Un ensemble de caractères est délimité par les caractères `[` et `]`.

Exemple: `[abc]` représente l'ensemble des caractères `{a, b, c}`

Sauf lorsqu'ils sont figurent dans la définition d'un ensemble de caractères (délimité par `[` et `]`), les symboles `[, ., *` sont interprétés.

'`[`' C'est le début de la définition d'un ensemble de caractères (comme nous l'avons vu plus haut).

'`.`' Il désigne un caractère quelconque sauf une fin de ligne.

'`*`' C'est un indicateur d'itérations d'un nombre quelconque de fois, sauf s'il figure comme premier caractère d'une expression ou comme caractère suivant la séquence `\(`.

'`]`' C'est la fin de la définition d'un ensemble de caractères (comme nous l'avons vu plus haut).

'`-`' Permet de définir une plage de caractères, `A-Z` désigne l'ensemble des caractères compris entre `A` et `Z` (i.e les caractères majuscules).

'`^`' désigne le début de ligne lorsqu'il figure en début d'expression et le complément ensembliste lorsqu'il suit immédiatement le début d'une définition d'un ensemble de caractères (juste après le caractère `[`).

'`$`' désigne une fin de ligne lorsqu'il est le dernier caractère d'une expression.

'\' précédant un caractère spécial lui enlève son interprétation (sauf dans la définition d'un ensemble).

Les expressions régulières atomiques

Les expression régulières atomiques sont :

1. Un caractère non spécial: il représente ce même caractère.
2. Un caractère spécial précédé du caractère '\': il représente ce même caractère.
3. Le caractère '.': il représente n'importe quel caractère excepté une fin de ligne.
4. une suite de caractères compris entre `[` et `]` et ne commençant pas le caractère '`^`': il représente n'importe quel caractère défini dans cette suite.
5. une suite de caractères compris entre `[^` et `]`: il représente n'importe quel autre caractère que ceux définis dans cette suite.

Les expressions régulières composées

Une expression régulière est de l'une des formes suivantes:

1. une expression régulière atomique
2. une expression régulière atomique suivi du caractère `*`
3. la concaténation de deux expressions régulières.
4. une expression encadrée par `\(` et `\)`). Une expression de cet type est strictement équivalente à cette même expression sans les symboles `\(` et `\)`) mais définit un moyen de repérage et de nommage de sous expressions
5. $\backslash n$ désigne exactement dans une expression le motif de la n^{eme} sous expression repérée par la méthode décrit plus haut. Le nombre n doit être compris entre 1 et 9.
6. une expression régulière atomique suivie de
   - `\{`n`\}` indique n occurrences successives de cette dernière.
   - `\{`n`,\}` indique au moins n occurrences successives de cette dernière.
   - `\{`n`,`m`\}` indique au moins n et au plus m occurrences successives de cette dernière. A vérifier
7. toute expression commençant par `^` (indique que le motif doit être en début de ligne).
8. toute expression se terminant par `$` (indique que le motif doit être en fin de ligne).

Exemple 1: `[a-zA-Z0-9]`
ensemble des caractères alphanumériques.

Exemple2 `[^a-zA-Z0-9]`
ensemble des caractères autres que alphanumériques.

Exemple 3 `[-^[]`
ensemble des trois caractères `-` `[` et `^`

Exemple 4 `^[a-z]*$`
lignes qui ne contiennent que des minuscules.

Exemple 5 `^[a-z]\{3\}[^a-z]\{3\}$`
lignes qui commencent par trois lignes minuscules et finissant par par trois caractères qui ne sont pas minuscules et qui ne contiennent aucun autre caractère.

Exemple 6 `^\(.*\)\1\1`
lignes constitués par des motifs répétés trois fois.

9.1.2 Généralités sur les caractères de substitutions

Les caractères de substitutions (*wildcard* en anglais) sont ceux utilisés généralement pour une liste de noms de fichiers.

`*` désigne n'importe quelle chaîne de caractères (même vide). Par exemple, `a*b` désigne toutes les chaînes commençant par `a` et se terminant par `b`.

`?` désigne n'importe quel caractère). Par exemple, `a?b` désigne toutes les chaînes de trois caractères commençant par `a` et se terminant par `b`.

`[...]` désigne n'importe quel caractère pris dans l'ensemble spécifié. Par exemple, `exo[0-9]` désigne les chaînes `exo0`, `exo1`, `exo2`, `exo3`, `exo4`, `exo5`, `exo6`, `exo7`, `exo8` et `exo9`.

9.1.3 Les expressions régulières et les caractères de substitutions de Qt

Les membres public de la `QRegExp`sont :

- **QRegExp** ()
  Construit une expression régulière vide.

- `QRegExp (const char* pattern, bool caseSensitive=TRUE, bool wildcard=FALSE)`
  Construit une expression régulière où *pattern* est la chaîne représentant l'expression régulière, *caseSensitive* est une valeur booléenne qui précise si l'on veut distinguer les majuscules des minuscules, *wildcard* est une valeur booléenne qui précise si les caractères spéciaux doivent être considérés comme des caractères de subtitution.

- `QRegExp (const QRegExp& r)`
  Construit une expression régulière par copie de l'expression régulière *r*.

- `QRegExp ()`
  Destructeur par défaut.

- `QRegExp& operator= (const QRegExp& r)`
  Affectation avec copie des tous les attributs de *r*.

- `QRegExp& operator= (const char* pattern)`
  Affectation avec copie de la chaîne *pattern*. Les paramètres *caseSensitive* et *wildcard* restent inchangés.

- `bool operator== (const QRegExp& r) const`
  Retourne *vrai* si les deux expressions régulières ainsi que les paramètres *caseSensitive* et *wildcard* sont identiques.

- `bool isEmpty () const`
  Retourne *vrai* si l'expression régulière est vide.

- `bool isValid () const`
  Retourne *vrai* si l'expression régulière est formée (pas d'erreur de syntaxe).

- `bool caseSensitive () const`
  Retourne la valeur du paramètre *caseSensitive*.

- `void setCaseSensitive (bool b)`
  Affecte le paramètre *caseSensitive* à la valeur de *b*.

- `bool wildcard () const`
  Retourne la valeur du paramètre *wildcard*.

- `void setWildcard (bool b)`
  Affecte le paramètre *wildcard* à la valeur de *b*.

- `const char* pattern () const`
  Retourne la chaîne représentant l'expression régulière.

- `int match (const char* str, int index=0, int* len=0) const`
  Vérifie la concordance de la chaîne `str` à l'expression régulière et ce à partir de la position donné par *index*. Retourne -1 s'il n'y pas de concordance et, dans le cas contraire, la position à partir de laquelle il y a une concordance. Si *len* n'est pas un pointeur *NULL*, la longueur de la concordance y est stockée. Example:

```
QRegExp r("[0-9]*.[0-9]+");          // syntaxe des nombres flottants
int len;
r.match("pi=3.1416", 0, &len);        // retourne 3, len == 6
```

9.2 Les objets partagés

9.2.1 Introduction

Certaines classes de ***Qt*** utilise le partage de données entre objets pour optimiser l'espace consommé et le temps d'exécution et ce en évitant les copies. ***Qt*** utilise deux techniques pour cela: le partage *explicite* et le partaghe *implicite*.

Un objet partagé est au moins constitué d'un pointeur vers un bloc contenant

- la donnée à partager
- un conteur de référence

A la création d'un objet, le compteur de référence est initialisé à 1. Celui-ci s'incrémente à chaque fois un nouvel objet partage cette donnée et se décrémente à chaque fois qu'un objet cesse le partage. Lorsque le compteur de référence devient 0, la donnée est effectivement supprimée.

Avec le partage de données, il faut envisager deux manières copier les objets:

- une copie complète (*deep* copy) qui recopie complètement les données de manière à ne pas utiliser le partage de données.Cette manière est évidemment plus coûteuse en temps et en mémoire.
- une copie superficielle (*shallow* copy) qui ne recopie que les pointeurs et incrément le compteur. Cette méthode est évidemment plus économique mais demande une attention particulière lors des destructions des objets.

Pour les données partageant les données, l'affection (avec `operator=`) utilise la copie superficielle. Pour utiliser la copie complète, on pourra utiliser la méthode `copy()`

Qtimplemente deux types de partage:

- le partage implicite. Le prgrammeur n'a pas à soucier des problèmes de partage de données; tout est transparent à l'utilisateur.
- le partage explicite. Le prgrammeur doit se soucier des problèmes de partage de données.

9.2.2 Le partage explicite

La classe `QString`est un exemple de classe dont les objets partagent les structures de manière explicite.

```
QString a = "one";    // 1)     a = "one"
QString b = "two";    // 2)     a = "one",    b = "two"
a = b;                // 3)   a,b = "two"
a[1] = 'a';           // 4)   a,b = "tao"
QString c = a;        // 5) a,b,c = "tao"
a.detach();           // 6)     a = "tao"   b,c = "tao"
a[1] = 'o';           // 7)     a = "too",  b,c = "tao"
```

L'affectation `a = b` détruit le bloc original de `a` puisque son conteur de référence est à 0 et `a` partage les données avec `b`. L'affectation `a[1] = 'a'` modifie la chaîne `a` mais également la chaîne `b`. C'est bien là que réside la distinction entre partage implicite et explicite. Le programmeur doit savoir que `a` et `b` partagent les données et faire très attention

aux diverses modification effectuées.

Si le programmeur avait voulu distinger les deux chaînes de manière à ne pas changer l'objet `b` lorsque l'objet `a` est modifié, il dispose de deux manières de faire:

- Eviter l'opérateur `=` qui permet le partage de données, et utiliser la méthode `copy()` pour disposer des données non communes.
- Spécifier à un instant qu'un objet ne doit plus partager de données par la méthode `detach()`. A la ligne 7, `a` est modifié mais `c` ne l'est pas.

Les classes qui utilisent le partage explicite sont: `QBitArray`, `QByteArray`, `QImage`, `QPointArray`, `QString`et toute instantiation de `QArray`<type>.

9.2.3 Le partage implicite

Ce qui distingue le partage implicite du partage explicite, c'est que dans un partage implicite, un objet en phase de modification est automatiquement détaché si le compteur de référence est supérieure à 1.

Les classes qui utilisent le partage implicite sont: `QBitmap`, `QBrush`, `QCursor`, `QFont`, `QFontInfo`, `QFontMetrics`, `QPen`, `QPixmap`et `QRegion`.

9.3 La classe QString

C'est le fameux tableau de caractères terminé par '`\0`' que l'on connait si bien.

la classe `QString`est une sous classe de `QByteArray`qui est définie comme un `QArray`<char>.

Comme sous classe de la classe `QArray`, la classe `QString`utilise le partage de structure explicite par compteur de référence.

> Les méthodes de `QString`qui prennent en argument des données de la forme `const char *` doivent obligatoirement se terminer par le caractère 0. Dans le cas contraire, les résultats sont non définis.

La manipulation de `QString`est agréable en ce sens que les méthodes fournies permettent une utilisation simplifiée des chaîines de caractères. On ne se souciera pas trop des problèmes de débordement et autres petites contrariétés inhérentes à l'utilisation des chaîines de caractère standard.

Les membres public de la `QString`sont : **QString** ()
QString (int size)
QString (const QString& s)
QString (const char* str)
QString& **operator**= (const QString& s)
QString& **operator**= (const char* str)
bool **isNull** () const
bool **isEmpty** () const
uint **length** () const
bool **resize** (uint newlen)
bool **truncate** (uint pos)
bool **fill** (char c, int len = -1)
QString **copy** () const
QString& **sprintf** (const char* format, ...)
int **find** (char c, int index=0, bool cs=TRUE) const

int **find** (const char* str, int index=0, bool cs=TRUE) const
int **find** (const QRegExp&, int index=0) const
int **findRev** (char c, int index=-1, bool cs=TRUE) const
int **findRev** (const char* str, int index=-1, bool cs=TRUE) const
int **findRev** (const QRegExp&, int index=-1) const
int **contains** (char c, bool cs=TRUE) const
int **contains** (const char* str, bool cs=TRUE) const
int **contains** (const QRegExp&) const
QString **left** (uint len) const
QString **right** (uint len) const
QString **mid** (uint index, uint len) const
QString **leftJustify** (uint width, char fill=' ', bool trunc=FALSE) const
QString **rightJustify** (uint width, char fill=' ', bool trunc=FALSE) const
QString **lower** () const
QString **upper** () const
QString **stripWhiteSpace** () const
QString **simplifyWhiteSpace** () const
QString& **insert** (uint index, const char*)
QString& **insert** (uint index, char)
QString& **append** (const char*)
QString& **prepend** (const char*)
QString& **remove** (uint index, uint len)
QString& **replace** (uint index, uint len, const char*)
QString& **replace** (const QRegExp&, const char*)
short **toShort** (bool* ok=0) const
ushort **toUShort** (bool* ok=0) const
int **toInt** (bool* ok=0) const
uint **toUtoUInt**Int (bool* ok=0) const
long **toLong** (bool* ok=0) const
ulong **toULong** (bool* ok=0) const
float **toFloat** (bool* ok=0) const
double **toDouble** (bool* ok=0) const
QString& **setStr** (const char* s)
QString& **setNum** (short)
QString& **setNum** (ushort)
QString& **setNum** (int)
QString& **setNum** (uint)
QString& **setNum** (long)
QString& **setNum** (ulong)
QString& **setNum** (float, char f='g', int prec=6)
QString& **setNum** (double, char f='g', int prec=6)
bool setExpand (uint index, char c)
operator const char* **()**const
QString& **operator**+= (const char* str)
QString& **operator**+= (char c)

9.4 La classe QFile

la classe `QFile`permet de lire et écrire dans fichiers binaires ou textes. On utilise soit directement les méthodes `readBlock` et `writeBlock` ou bien on utilisera des objets des classes `QDataStream`ou `QTextStream`.

La classe `QFileInfo`permet de manipuler les propriétés des fichiers telles que les droits d'accès, les date de création et modification, le type etc.

La classe `QDir`permet de gérer les répertoire et listes de fichiers.

La classe `QFile`dispose de deux constructeurs

```
QFile ()
QFile (const char* name)
```

La première construit un objet de type `QFile`sans nom alors que la deuxième assigne pour nom la chaîne de caractères passée en argument.

Avant toute utilisation d'un objet de type `QFile`, il faudra ouvrir le fichier par la méthode

```
bool open ( int m) [virtual]
```

Cette méthode retourne `TRUE` si l'opération a réussi ou `FALSE` sinon. Le paramètre `m` The mode parameter m must est une combinaison des drapeaux (*flags* en anglais) suivants:

`IO_RAW` : accès sans buffer

`IO_ReadOnly` : accès en lecture seuleument

`IO_WriteOnly` : accès en écriture seuleument

`IO_ReadWrite` : accès en lecture/écriture.

`IO_Append` : positonnement en fin de fichier

`IO_Truncate` : tronque le fichier.

`IO_Translate` : convertit les retour chariot et les saut de ligne selon le système utilisé (*MS-DOS*, *Window*, *OS/2* et *Macintosh*).

Example:

```
QFile f1( "/tmp/data.bin" );
QFile f2( "readme.txt" );
f1.open( IO_Raw | IO_ReadWrite | IO_Append );
f2.open( IO_ReadOnly | IO_Translate );
```

La méthode

```
int readBlock ( char *p, uint len) [virtual]
```

permet de lire au plus `len` octets sur le fichier et le range dans p et retourne le nombre d'octets effectivement lus (-1 en cas d'aeereur).

La méthode

```
int writeBlock ( const char *p, uint len) [virtual]
```

permet d'écrire le contenu pointé p jusqu'à *len* octets et retourne le nombre d'octets écrits.

A la fin de l'utilisation d'un `QFile`, il faut fermer le fichier avec la méthode

```
void close () [virtual]
```

9.5 La classe QDataStream

Une flot de données (data stream) est un flot de données binaires qui totalement indépendante de l'architecture et du système utilisés. On pourra donc écrire un fichier binaire sous MS-DOS et le lire sous Unix, par exemple.

La classe `QDataStream`implente les primitives de manipulation des structures un peu plus sophistiqués que les octets de la classe `QFile`: on pourra lire des entires, des flottants, des pointeurs de caractères etc.

La classe `QDataStream`s'appuie sur la classe `QIODevice`; et une sous classe de la classe `QIODevice`est la classe `QFile`.

Example (écriture):

```
QFile f( "file.dat" );
f.open( IO_WriteOnly );
QDataStream s( &f );
s « "the answer is";
s « (Q_INT32)42;
f.close();
```

Example (lecture):

```
QFile f( "file.dta" );
f.open( IO_ReadOnly );
QDataStream s( &f );
char   *str;
Q_INT32 a;
s » str » a;
cout « str « a;
f.close();
delete str;
```

9.6 La classe QTextStream

Une flot de texte (text stream) est un flot de données constitué de caractères très similaire à ce qu'on trouve en C++.

Exemple:

```
QString str;
QTextStream ts( str, IO_WriteOnly );
ts « "pi = " « 3.14;                    // str == "pi = 3.14"
```

9.7 La classe QFileInfo

La classe `QFileInfo`permet d'obtenir des information sur les

fichiers et ce de manière totalement indépepenadnte du système utilisé.

La classe `QFileInfo`founit des information sur les noms de fichiers, les droits d'accès, le répertoire où il réside etc.

Pour améliorier les performances, la classe QFileInfo utilise un cache pour founir ces informations. Un méthode `refresh` permet de mettre à jour ce cache. De plus, le système de cache est déconnectable par la méthode `setCaching( FALSE)` .

9.8 La classe QDir

la classe `QDir`permet d'examiner la structure des répertoires
et de son contenu et ce de manière totalement indépendante du système utilisé.
Par exemple, pour savoir si le répertoire `exemple` existe dans le répertoire courant, on pourra écrire:

```
QDir d( "exemple" );
if ( !d.exists() )
  warning( "Répertoire exemple introuvable" );
```

Les méthodes `cd()` et `cdUp()` permettent de se déplacer dans les répertoires.
Example:

```
QDir d = QDir::root();                    // \emph{"/"}
if ( !d.cd("tmp") ) {                     // \emph{"/tmp"}
  warning( "Le repertoire \"/tmp\" n'existe pas" );
}
else {
  QFile f( d.filePath("ex1.txt") );       // \emph{"/tmp/ex1.txt"}
  if ( !f.open(IO_ReadWrite) )
    warning( "Impossible de créer le fichier %s", f.name() );
}
```

Pour lire le contenu d'un répertoire, on utilisera les méthodes `entryList()` et `entryInfoList()`.
Example:

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

int main( int argc, char **argv ) {
  QDir d;
  d.setFilter( QDir::Files | QDir::Hidden | QDir::NoSymLinks );
  d.setSorting( QDir::Size | QDir::Reversed );

  const QFileInfoList *list = d.entryInfoList();
  QFileInfoListIterator it( *list );
  QFileInfo *fi;

  printf( "     BYTES FILENAME\n" );
  while ( (fi=it.current()) ) {
    printf( "%10li %s\n", fi->size(), fi->fileName().data() );
    ++it;
  }
}
```

9.9 La classe QFileDialog

La classe `QFileInfo`fournit un widget pour selectionner des fichiers. Example:

```
QString fileName = QFileDialog::getOpenFileName();
    if ( !fileName.isNull() )
        ...
```

Il existe deux méthodes très utiles qui sont `getOpenFileName()` et `getSaveFileName()` et qui permettent d'ouvrir et de sauver un fichier.

```
QString s( QFileDialog::getOpenFileName() );
if ( s.isNull() )
  return;
open( s );
```

9.10 La classe QImage

9.11 La classe QPixmap

Chapitre 10

Des applications

Sommaire

10.1 Une palette de dessin

10.1.1 La palette graphique

10.1.2 Les formes que l'on veut manipuler

```
#include <qcolor.h>
#include <qwidget.h>
#include <qpainter.h>
#include <qapp.h>
#include <iostream.h>

#include "xpaint.h"

class Forme

public:
  static Forme *liste;
  static Forme *crt;
  static QColor coulContDef;
  static QColor coulFondDef;
  static const int espace;

  Forme();
  QColor getFond() return fond;
  QColor getContour() return contour;
  void setFond(QColor c) fond = c;
  void setContour(QColor c) contour = c;
  void setX(int x)  posx = x;
  void setY(int y)  posy = y;
```

```
  void empiler();
  static void listerTout();
  static void dessinerTout();
  static void selectTout();
  static Forme *quelleForme(int x, int y);
  void supprimer();
  virtual void mousePressEvent( QMouseEvent *e)=0;
  virtual void mouseReleaseEvent( QMouseEvent *e)=0;
  virtual void mouseMoveEvent( QMouseEvent *e)=0;
  void repaint()  xpaintW::feuille->repaint();
protected:
  virtual void lister()=0;
  virtual void dessiner(QPainter *)=0;
  virtual void select(QPainter *)=0;
  virtual int  c_est_moi(int x, int y)=0;
  QColor fond;
  QColor contour;
  int posx, posy;

private:
  Forme *suiv;

;


class rectangle : public Forme
private:
  int dx, dy;
public:
  rectangle(int x, int y, int deltax, int deltay);
  void lister();
  void dessiner(QPainter *);
  void select(QPainter *);
  int c_est_moi(int x, int y);

  void mousePressEvent( QMouseEvent *e) ;
  void mouseReleaseEvent( QMouseEvent *e) ;
  void mouseMoveEvent( QMouseEvent *e) ;
;


class trait : public Forme
private:
  int dx, dy;
public:
  trait(int x, int y, int deltax, int deltay);
protected:
  void lister();
  void dessiner(QPainter *);
  void select(QPainter *);
  int c_est_moi(int x, int y);
```

```
  void mousePressEvent( QMouseEvent *e) ;
  void mouseReleaseEvent( QMouseEvent *e) ;
  void mouseMoveEvent( QMouseEvent *e) ;
;

class rond : public Forme
private:
  int dx, dy;
public:
  rond(int x, int y, int deltax, int deltay);
protected:
  void lister();
  void dessiner(QPainter *);
  void select(QPainter *);
  int c_est_moi(int x, int y);

  void mousePressEvent( QMouseEvent *e) ;
  void mouseReleaseEvent( QMouseEvent *e) ;
  void mouseMoveEvent( QMouseEvent *e) ;
;



#include <qwidget.h>
#include <qpainter.h>
#include <qapp.h>
#include "forme.h"
#include "xpaint.h"
#include <iostream.h>

// ************************************
// Classe Forme
// ************************************

Forme * Forme::liste = NULL;
Forme * Forme::crt = NULL;
QColor Forme::coulContDef(red);
QColor Forme::coulFondDef(magenta);

const int Forme::espace = 3;

Forme::Forme()
fond = coulFondDef;
contour = coulContDef;


void Forme::empiler()
  suiv = liste;
  liste = this;


void Forme::listerTout()
```

```
  Forme *f;
  for (f= liste; f != NULL; f = f->suiv)
    f->lister();

  if (crt != NULL) crt->lister();


void Forme::dessinerTout()
  QPainter * paint = new QPainter(xpaintW::feuille);
  Forme *f;
  //  paint->begin(xpaintW::feuille  );
  for (f= liste; f != NULL; f = f->suiv)
    f->dessiner(paint);
  if (crt != NULL) crt->dessiner(paint);
  paint->end();
  delete paint;


void Forme::selectTout()
  QPainter * paint = new QPainter(xpaintW::feuille);
  Forme *f;
  for (f= liste; f != NULL; f = f->suiv)
    f->select(paint);
  paint->end();
  delete paint;


Forme *Forme::quelleForme(int x, int y)
  Forme *f;

  for (f= liste; f != NULL; f = f->suiv)
    if (f->c_est_moi(x, y))
      break;
  return f;


void Forme::supprimer()
  Forme *f, *pf;

  for (f= liste, pf = NULL; f != NULL && f != this; pf = f, f = f->suiv);
  if (f!=NULL)
    if (pf == NULL) liste = liste->suiv;
    else pf->suiv = f->suiv;
    f->suiv = NULL;



// ************************************
// Classe rectangle
// ************************************

rectangle::rectangle(int x, int y, int deltax, int deltay) : Forme()
```

```
  posx=x; posy=y; dx=deltax; dy=deltay;


void rectangle::lister()
    cout « "fond:" « fond.rgb()  « "  contour:" « contour.rgb()  « "  ";
    cout«"rectangle("«posx«", "« posy«", "«dx«", "«dy«")"«endl;


void rectangle::dessiner(QPainter *paint)
  paint->setBrush( getFond() );
  paint->setPen( getContour() );
  paint->drawRect( posx, posy, dx, dy);


void rectangle::select(QPainter *paint)
  paint->setBrush( black );
  paint->setPen( black );
  paint->drawRect( posx-espace, posy-espace, 2*espace, 2*espace);
  paint->drawRect( posx+dx-espace, posy-espace, 2*espace, 2*espace);
  paint->drawRect( posx-espace, posy+dy-espace, 2*espace, 2*espace);
  paint->drawRect( posx+dx-espace, posy+dy-espace, 2*espace, 2*espace);


int rectangle::c_est_moi(int x, int y)
  return ((x >= posx-espace && x <= posx+espace && y >= posy-espace && y <= posy+espace) ||
          (x >= posx+dx-espace && x <= posx+dx+espace && y >= posy-espace && y <= posy+espace) ||
          (x >= posx-espace && x <= posx+espace && y >= posy+dy-espace && y <= posy+dy+espace) ||
          (x >= posx+dx-espace && x <= posx+dx+espace && y >= posy+dy-espace && y <= posy+dy+espac


void rectangle::mousePressEvent( QMouseEvent *e)
  posx = e->x(); posy = e->y();


void rectangle::mouseReleaseEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();
  empiler();
  Forme:crt = new rectangle(0,0,0,0);


void rectangle::mouseMoveEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();


// ***************************************
// Classe trait
// ***************************************

trait::trait(int x, int y, int deltax, int deltay) : Forme()
  posx=x; posy=y; dx=deltax; dy=deltay;
```

```
void trait::lister()
    cout « "fond:" « fond.rgb()  « "  contour:" « contour.rgb()  « "  ";
  cout«"trait("«posx«", "« posy«", "«dx«", "«dy«")"«endl;


void trait::dessiner(QPainter *paint)
  paint->setPen( getContour() );
  paint->drawLine( posx, posy, posx+dx, posy+dy);


void trait::select(QPainter *paint)
  paint->setBrush( black );
  paint->setPen( black );
  paint->drawRect( posx-espace, posy-espace, 2*espace, 2*espace);
  paint->drawRect( posx+dx-espace, posy+dy-espace, 2*espace, 2*espace);


int trait::c_est_moi(int x, int y)
  return ((x >= posx-espace && x <= posx+espace && y >= posy-espace && y <= posy+espace) ||
          (x >= posx+dx-espace && x <= posx+dx+espace && y >= posy+dy-espace && y <= posy+dy+espac


void trait::mousePressEvent( QMouseEvent *e)
  posx = e->x(); posy = e->y();


void trait::mouseReleaseEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();
  empiler();
  Forme:crt = new trait(0,0,0,0);


void trait::mouseMoveEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();


// ************************************
// Classe rond
// ************************************

rond::rond(int x, int y, int deltax, int deltay) : Forme()
  posx=x; posy=y; dx=deltax; dy=deltay;


void rond::lister()
    cout « "fond:" « fond.rgb() « "  contour:" « contour.rgb()   « "  ";
  cout«"rond("«posx«", "« posy«", "«dx«", "«dy«")"«endl;
```

```
void rond::dessiner(QPainter *paint)
  paint->setBrush( getFond() );
  paint->setPen( getContour() );
  paint->drawEllipse( posx, posy, dx, dy);


void rond::select(QPainter *paint)
  paint->setBrush( black );
  paint->setPen( black );
  paint->drawRect( posx-espace, posy-espace, 2*espace, 2*espace);
  paint->drawRect( posx+dx-espace, posy+dy-espace, 2*espace, 2*espace);


int rond::c_est_moi(int x, int y)
  return ((x >= posx-espace && x <= posx+espace && y >= posy-espace && y <= posy+espace) ||
          (x >= posx+dx-espace && x <= posx+dx+espace && y >= posy+dy-espace && y <= posy+dy+espac


void rond::mousePressEvent( QMouseEvent *e)
  posx = e->x(); posy = e->y();


void rond::mouseReleaseEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();
  empiler();
  Forme:crt = new rond(0,0,0,0);


void rond::mouseMoveEvent( QMouseEvent *e)
  dx = e->x() - posx; dy = e->y() - posy;
  repaint();
```

Chapitre 11

Memo

11.1 QPrinter

11.2 QSignal

11.3 QSocketNotifier

11.4 QWMatrix

Chapitre 12

Dlgedit: Aide à la programmation Qt