Qt4.0でGUIアプリケーションを作ってみよう

でんさんラボ
Nishio

2007年8月29日

目 次

第 1 章　Hello Qt! **5**
1.1 Hello Qt! 5
1.2 ラベルについて (QLabel) 7
1.3 ボタンを作ってみよう (QPushButton) 8
1.4 フォントを変えてみよう (QFont) 9
1.5 とりあえず日本語を表示してみよう 10

第 2 章　Layout を使ってみよう **13**
2.1 なぜ Layout を使うのか 13
2.2 基本的な Layout の使い方 14
2.3 QGridLayout 15
2.4 Layout に Layout を追加する (addLayout) 17
2.5 メモリ管理について 19

第 3 章　SIGNAL/SLOT を使ってみよう **21**
3.1 SIGNAL/SLOT の基本的な使い方 21
3.2 複数の Slot について 22
3.3 Connection の解除 25
3.4 Signal/Slot のまとめ 25

第 4 章　プログラム作成への準備 **27**
4.1 ダイアログを作ってみよう 27
4.2 モーダルダイアログを作ってみよう 31
4.3 モードレスダイアログを作ってみよう 36

第 5 章　Qt Designer を使ってみよう **41**
5.1 Qt Designer とは 41
5.2 Qt Designer の起動 41
5.3 部品の配置 41
5.4 Signal/Slot を設定してみよう 43

第1章　Hello Qt!

1.1　Hello Qt!

最初は画面に *HelloQt!* と表示するプログラムを作ることにします。まずは何も考えず次のソースコードを入力し、実行できるかどうかのテストをしてみてください。

```
1   #include <QApplication>
2   #include <QLabel>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QLabel* label = new QLabel("Hello Qt!");
8           label->show();
9           return app.exec();
10  }
```

コンパイルを行うには Linux の場合ターミナルを実行しソースコードが置いてあるフォルダまで移動してください。その後、

qmake -project

をタイプしてください。次に

qmake

をタイプし、最後に

make

とします。これでフォルダに実行ファイルが作成されたはずです。

Windows で VisualStudio を使っている場合は、コマンドプロンプトを実行しソースコードが置いてあるフォルダに移動したあと、

qmake -project -t vcapp -o hello.pro

をタイプし、次に

qmake

を実行します。そうすると、hello.vcproj という VisualStudio のプロジェクトファイルが作成されます。このプロジェクトを実行し、コンパイルを行ってください。

プログラムを実行すると Windows 環境の場合、図 1.1 のように表示されると思います。

図 1.1: Windows で実行した Hello

では一行ずつ解説していきます。

1,2 行目の部分は、QApplication と QLabel のクラスの定義をインクルードしています。Qt を使っている GUI アプリケーションには必ず一つの QApplication のオブジェクトが無ければなりません。また、すべての Qt クラスはヘッダーファイルとクラスが同じ名前に設定されています。

6 行目は QApplication オブジェクトを作成しています。QApplication はアプリケーションに関するリソースを管理するものであり、コンストラクタには argc と argv を渡さなければなりません。このため、Qt で作成したプログラムでもコマンドライン引数を使うことが出来ます。

7 行目では Hello Qt!と表示するラベルを作成しています。そして 8 行目でラベルを表示しています。

そして 9 行目でコントロールを Qt アプリケーションに渡します。コントロールを渡した先ではイベントループが起こっています。

これでプログラムは完成です。プログラムをコンパイルして実行してみてください。うまくコンパイルができたでしょうか？

ここで、QLabel は new したのに delete していないのでメモリリークが起こりますが、プログラムが終了すればオペレーティングシステムが勝手に処理してくれるので、細かいことは気にしないでください。

1.2 ラベルについて (QLabel)

1.1 節のプログラムで QLabel を使用しました。ここで QLabel の面白い使い方を見てみましょう。前節の

```
QLabel* label = new QLabel("Hello Qt!");
```

の部分を次のコードに置き換えてコンパイル、実行してみてください。

```
QLabel* label = new QLabel("<i>Hello Qt!</i>");
```

図 1.2 のように、斜体文字となりました。

図 1.2: Windows で実行した Hello2

 という書き方は HTML で主に使用されています。このように HTML スタイルの書き方を QLabel ではサポートしています。他にも

```
QLabel* label = new QLabel("<h2><i>Hello<br>Qt!</i></h2>");
```

とすると、図 1.3 となります。

図 1.3: Linux で実行した Test2

HTML についてはここでは説明しません。興味のある方はインターネットで調べてみてください。とほほの WWW 入門 (http://www.tohoho-web.com/www.htm) なんかがお勧めです。

1.3 ボタンを作ってみよう (QPushButton)

前節ではラベルを作ったので、今度はボタンを作ってみましょう。次のようなコードを書いたとします。

```
1   #include <QApplication>
2   #include <QPushButton>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QPushButton* button = new QPushButton("Hello Qt!");
8           button->resize(200,50);
9           button->move(100,50);
10          button->show();
11          return app.exec();
12  }
```

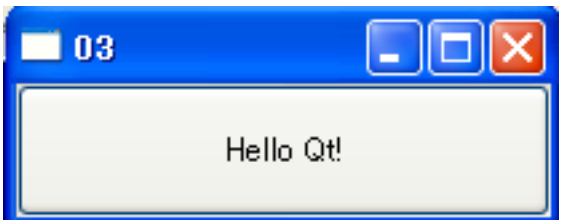

図 1.4: QPushButton の例 (on Windows)

このコードの 2 行目及び 7 行目は、1.1 節の QLabel の部分が QPushButton に変わっただけです。このようにすることでボタンを作成することができます。

8 行目、9 行目は今回始めて出てきた関数です。resize 関数は、ボタンの大きさを変更するものです。この場合は、高さ 200Pixels, 幅 50Pixels となります。

move 関数は、ウィンドウ（この場合ボタン）がデスクトップ上に表示される位置を決めるものです。関数は move(x 座標, y 座標) となっており、デスクトップ画面の一番左端が (0,0) となっています。x 座標はデスクトップの左から右にいくに従い値が増加し、y 座標は上から下に行くに従い値が増加します。

これら resize, move 関数は特に設定しなくても問題ありません。また、これらの関数は QPushButton のみならず、QLabel などの部品でも使うことができます。

QPushButton は QLabel とは違い HTML のタグを使うことはできません。

さて、このプログラムの実行結果は図 1.4 となります。

ここでせっかくボタンを作ったのだから、例えば、ボタンをクリックしたらプログラムが終了するなど何かアクションを起こしたいと思うかもしれません。けど、もう少し待ってください。この事については第 3 章で取り扱います。まずは Widget[1]の基本的な使い方を勉強してしまいましょう。

1.4 フォントを変えてみよう (QFont)

前節でボタン (QPushButton) を作りましたが、もしかしたら文字フォントを変えてみたいと思ったかもしれません。(そんなことは無い？)

Qt ではフォントを変えるのも簡単です。前節のプログラムに次のコードを書き加えるだけです。

```
#include <QFont>

 button->setFont( QFont("Times", 15, QFont::Bold) );
```

QFont を使う場合は、QFont をインクルードしなければなりません。

QFont のコンストラクタは、

```
  QFont(const QString & family, int pointSize = -1,
int weight = -1, bool italic = false );
```

と定義されています (他にもありますが省略します)。

まずは、setFont 関数ですが、この関数に QFont クラスのオブジェクトを渡してあげることでフォントの変更ができます。

QFont のコンストラクタの第 1 引数の引数はフォントの名前を指定してあげます。例では Times というフォントを使うことにしました。他にも System や Windows の場合だと Tahoma なんていうフォントもあります。いろいろ試してみてください。

第 2 引数にはフォントの大きさを指定します。

第 3 引数にはフォントの幅を指定します。例ではフォントの幅を QFont::Bold としました。これは 75 という数字に置き換えられます。ボールド文字 (太い文字) にしたい時には QFont::Bold を使えばよいでしょう。他にも QFont::Normal(50 に置き換えられる) や QFont::Light(25 に置き換えられる) などがあります。0〜99 の値の範囲ならどんな値でもよいです。

[1]Window と Gadget を混ぜた言葉で、日本語に直すとウィンドウの部品ということです。QPushButton や QLabel などが部品に当たります。

第 4 引数は bool 型となっており、イタリック体（斜体）にしたい場合はこの引数に true を与えてあげればよいです。

図 1.5 は `button->setFont( QFont("Times", 15, QFont::Bold , true) );` を追加した場合の実行例です。

図 1.5: QFont の例 (on Windows)

1.5 とりあえず日本語を表示してみよう

もしかしたらもうお気づきの方もいるかもしれません。今までボタンやラベルのキャプションには英語しか使ってきませんでした。なぜ日本語を使わなかったのかというと、実は日本語の扱いは厄介なものだからです。次のようなコードを実行したとしましょう。これは 1.1 節のコードの 7 行目を変えただけです。

```
1   #include <QApplication>
2   #include <QLabel>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QLabel* label = new QLabel("こんにちは Qt");
8           label->show();
9           return app.exec();
10  }
```

この実行結果は図 1.6 となります。

なんと文字化けしてしまっています。Linux と Windows どちらも使ったことのある方なら、文字コードの問題は厄介なものだと知っていると思います。

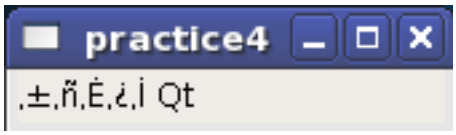

図 1.6: 日本語の表示 (on Linux)

Windows の場合、使われている文字コードは大体が Shift-JIS です。Unix 系の場合は EUC-JP や UTF-8 などでしょうか。この文字コードの違いにより文字化けが起こってしまいます。

この問題を回避するためには、Qt に自分が使っている文字コードを教えてあげる必要があります。その方法は、次のようなコードになります。

```
1   #include <QApplication>
2   #include <QLabel>
3   #include <QTextCodec>
4   #include <QString>
5
6   int main(int argc, char** argv)
7   {
8           QApplication app(argc, argv);
9           QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
10          QLabel* label = new QLabel(QObject::tr("こんにちは Qt"));
11          label->show();
12          return app.exec();
13  }
```

実行結果は、図 1.7 となります。

(a)on Linux (b)on Windows

図 1.7: 日本語の表示

重要なのは、9 行目と 10 行目です。9 行目の

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

という部分で文字コードを指定しています。

setCodecForTr 関数の引数には、QTextCodec 型の値を渡します。

例では、QTextCodec::codecForLocale() を渡していますが、この関数はシステムで使われている標準的な文字コードを返します。よって自分が使っている環境にあった文字コードを指定しているので大体の場合はこの方法で問題ないと思います。

ただ、人によっては文字コードを指定したい場合があるかもしれません。その場合、

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("Shift-JIS")
);
```

とします。codecForName 関数の中に直接使用したい文字コードを書き込みます。

10 行目は QObject::tr 関数を使用しています。この関数の中に文字を書く事により文字コードが正常に変換されます。また、tr 関数は後に自分の作ったプログラムを国際化したい場合にも使われます。

さて、とりあえず日本語は使えるようになりました。しかしながら、かなり説明が雑だと思います。このあたりの説明は後の章でもう一度詳しく説明することになると思いますが、今はこういうものだと思ってください。

第2章　Layoutを使ってみよう

2.1　なぜLayoutを使うのか

第1章ではボタンあるいはラベルがウィンドウに一つしか存在しませんでした。では、次にボタンを二つ並べて表示してみましょう。次のようなコードを書いたとします。

```
1   #include <QApplication>
2   #include <QPushButton>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QPushButton* button = new QPushButton("Hello Qt!");
8           QPushButton* button2 = new QPushButton("Goodbye");
9           button->show();
10          button2->show();
11          return app.exec();
12  }
```

このコードを実行すると、図2.1のようになります。

図 2.1: 実行結果 (on Windows)

本当はボタンが二つ並んで一つのウィンドウに収まってくれることを願っていましたが、これは予想外の結果です。

実は、一つのウィンドウには一つの部品しか表示[1]できないのです。[2]この一つの部品を親としましょう。それではどのようにして複数のボタンを作ったらよいのでしょうか。

その方法は、一つの親を作りその中にどんどん部品を追加していけばよいのです。この追加した部品を子とします。そうすれば、親の部品を表示することにより子も一緒に表示されます。

この親となるのがMainWindowであり、その中にLayoutをセットします。Layoutは部品をどのように配置するかを管理するクラスです。

2.2 基本的なLayoutの使い方

Layoutにはいくつかの種類が存在します。最も良く使われるのが、QHBoxLayoutとQVBoxLayoutでしょう。QHBoxLayoutのHはHorizontal、QVBoxLayoutのVはVerticalであり、水平と垂直を意味しています。

実際使ってみたほうがわかりやすいので、とりあえず使い方を見てみましょう。まずはQHBoxLayoutに3つのボタンを追加し、そのLayoutをメインウィンドウに表示させてみましょう。

```
1   #include <QApplication>
2   #include <QPushButton>
3   #include <QHBoxLayout>
4
5   int main(int argc, char** argv)
6   {
7           QApplication app(argc, argv);
8           QWidget* window = new QWidget;
9           QPushButton* buttonA = new QPushButton("Button A");
10          QPushButton* buttonB = new QPushButton("Button B");
11          QPushButton* buttonC = new QPushButton("Button C");
12          QHBoxLayout* layout = new QHBoxLayout;
13
14          layout->addWidget(buttonA);
```

[1]show 関数を呼び出す事

[2]語弊があるかもしれません。すみません。

```
15            layout->addWidget(buttonB);
16            layout->addWidget(buttonC);
17            window->setLayout(layout);
18            window->show();
19
20            return app.exec();
21    }
```

3 行目では<QHBoxLayout>をインクルードしています。QHBoxLayout を使うには、このヘッダーファイルが必要です。

8 行目では QWidget クラスの window という変数を定義していますが、これがメインウィンドウ（親）となる変数です。この中にレイアウトをセットすることで複数の部品を表示することができます。

14〜16 行目でボタンをレイアウトに追加しています。QHBoxLayout はボタンを追加した順に水平に配置していきます。

17 行目でウィンドウにレイアウトをセットしています。

これで完了です。実行した結果は図 2.2 のようになります。

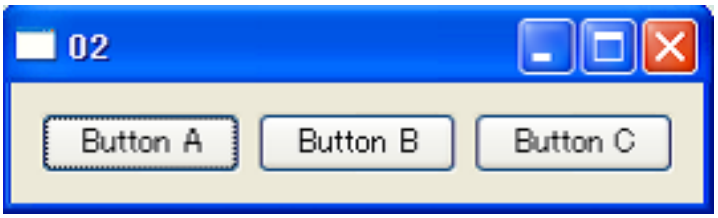

図 2.2: 実行結果 (on Windows)

QHBoxLayout を QVBoxLayout に変えてみてください。QVBoxLayout はボタンを追加した順に垂直に配置していきます。実行結果は図 2.3 となります。

2.3 QGridLayout

QGridLayout は名前の通り格子状に部品を配置していく Layout です。QGridLayout に部品を追加する場合、QHBoxLayout や QVBoxLayout とは少し引数が違います。次のコードを見てください。

```
1    #include <QApplication>
2    #include <QPushButton>
```

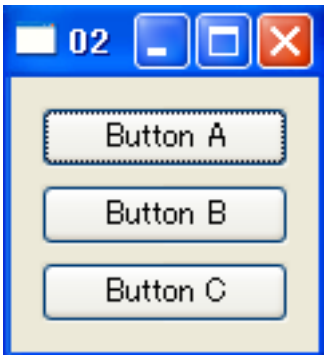

図 2.3: 実行結果 (on Windows)

```
3   #include <QGridLayout>
4
5   int main(int argc, char** argv)
6   {
7           QApplication app(argc, argv);
8           QWidget* window = new QWidget;
9           QPushButton* buttonA = new QPushButton("Button A");
10          QPushButton* buttonB = new QPushButton("Button B");
11          QPushButton* buttonC = new QPushButton("Button C");
12          QGridLayout* layout = new QGridLayout;
13
14          layout->addWidget(buttonA,0,0);
15          layout->addWidget(buttonB,0,1);
16          layout->addWidget(buttonC,1,0,1,2);
17
18          window->setLayout(layout);
19          window->show();
20          return app.exec();
21  }
```

このコードを実行すると、図 2.4 となります。

addWidget 関数が少し変わっています。addWidget の第 1 引数はボタンを配置を開始する縦の位置、第 2 引数は横の位置を表しています。第 3 引数は縦の大きさ（例の場合ボタン 1 個分）、第 4 引数は横の大きさ（例の場合ボタン 2 つ分）を表しています（図 2.5)。

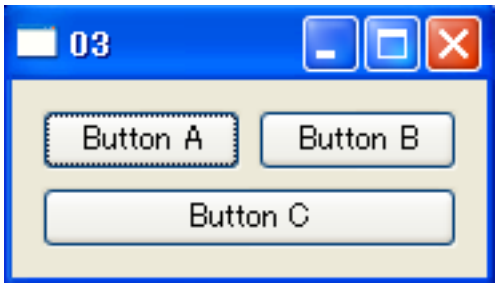

図 2.4: 実行結果 (on Windows)

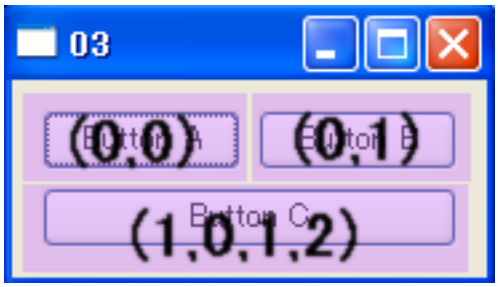

図 2.5: GridLayout の配置

2.4 Layout に Layout を追加する (addLayout)

addLayout 関数を使い Layout に Layout を追加することもできます。次のコードを見てください。

```
1   #include <QApplication>
2   #include <QPushButton>
3   #include <QHBoxLayout>
4   #include <QVBoxLayout>
5
6   int main(int argc, char** argv)
7   {
8           QApplication app(argc, argv);
9           QWidget* window = new QWidget;
10          QPushButton* buttonA = new QPushButton("Button A");
11          QPushButton* buttonB = new QPushButton("Button B");
12          QPushButton* buttonC = new QPushButton("Button C");
```

```
13          QPushButton* buttonD = new QPushButton("Button D");
14
15          QVBoxLayout* mainLayout = new QVBoxLayout;
16          QHBoxLayout* layoutA = new QHBoxLayout;
17          QVBoxLayout* layoutB = new QVBoxLayout;
18
19          layoutA->addWidget(buttonA);
20          layoutA->addWidget(buttonB);
21          layoutB->addWidget(buttonC);
22          layoutB->addWidget(buttonD);
23
24          mainLayout->addLayout(layoutA);
25          mainLayout->addLayout(layoutB);
26
27          window->setLayout(mainLayout);
28          window->show();
29          return app.exec();
30  }
```

この例では、mainLayout を作りその中に二つのレイアウトを縦に配置しています。実行結果は図 2.6 となります。

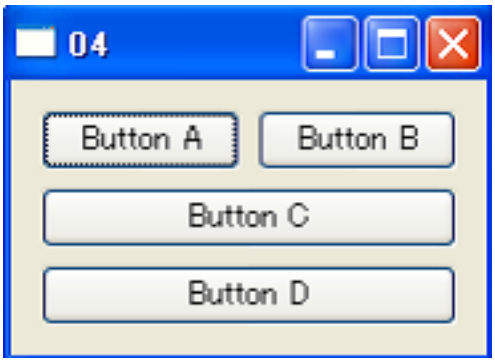

図 2.6: 実行結果 (on Windows)

2.5 メモリ管理について

さて、ここまで部品を new した場合 delete をしていませんでした。小さなプログラムでは少しくらい delete しなくても問題ありませんが、大規模なプログラムとなるとメモリリーク[3]が心配です。

でも、今まで buttonA や buttonB 等作ってきましたが、これを全部 delete するのはとても面倒なことです。

実はこの事を心配する必要はありません。Qt では親が delete される時に一緒に子の部品も自動的に delete してくれる仕組みとなっています。

よって必要なくなったら親のみ delete すればよいでしょう。

[3]new したものを delete しないで置く事

第3章　SIGNAL/SLOTを使ってみよう

3.1　SIGNAL/SLOTの基本的な使い方

Qt で最も良く使い、Qt の特徴でもある機能がシグナル/スロット (Signal and Slot) です。

シグナルは、例えばボタン (QPushButton) が押された時には clicked という関数が呼ばれます。この clicked 関数がシグナルとなります。そしてシグナルが発生した時には SLOT で指定した関数が呼び出されます。例えば、SLOT に quit 関数 (この関数はプログラムを終了する時に呼び出す) を指定しておけば、ボタンを押した後、プログラムが終了します。わかりにくいと思うので、例を挙げて説明します。

次のコードを見てください。

```
1   #include <QApplication>
2   #include <QPushButton>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QPushButton* button = new QPushButton("Quit");
8           QObject::connect(button, SIGNAL( clicked() ),
9                           &app, SLOT(quit()) );
10          button->show();
11          return app.exec();
12  }
```

このコードを実行すると、図 3.1 のようになります。

connect 関数の第一引数はシグナル (信号) が発生する部品のアドレスを渡します。例の場合、Quit と表示されている button をセットしています。そして、第 2 引数でシグナルとなる関数を指定します。関数をシグナルに設定する場合、SIGNAL(

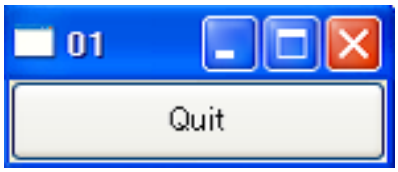

図 3.1: 実行結果 (on Windows)

) のカッコで囲った中に関数を書きます。QPushButton の場合、ボタンがクリックされると clicked 関数が呼び出されます。この clicked 関数がシグナルとなります

そして第 3,4 引数で発生したシグナルを app に結び付けています。

第 3 引数にはスロット側の部品のアドレスを渡します。そして第 4 引数として SLOT( ) のカッコで囲った中に書いた関数をスロットとなる関数と設定します。QApplication クラスの関数である quit 関数はプログラムを終了する時に使います。

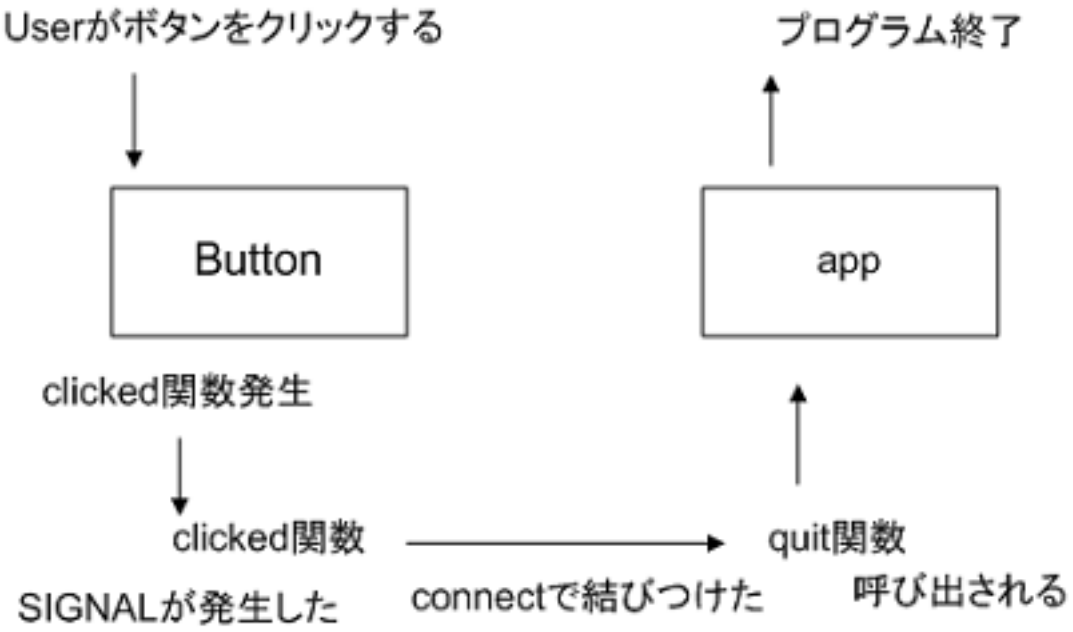

図 3.2: Signal/Slot のイメージ図

3.2 複数の Slot について

次は図 3.3 のようなウィンドウを作ります。

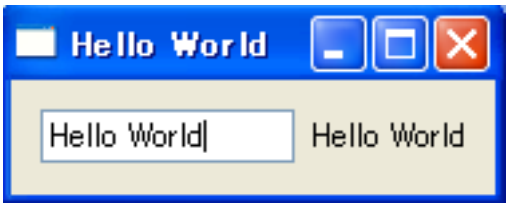

図 3.3: 実行結果 (on Windows)

QLineEdit と QLabel という部品を使い、Signal を textChanged 関数 (QLineEdit が編集されたら発生) とします。そして今回は Slot を複数設定してみましょう。

Signal が発生したら、QLabel にも QLineEdit と同じ内容が表示されるようにします。QLabel に文字をセットする場合、setText 関数を使います。

また同時に Window のタイトルも QLineEdit と同じ内容を表示させてみます。Window のタイトルをセットするには setWindowTitle 関数を使います。

では次のようなコードを書いてみましょう。

```
1   #include <QApplication>
2   #include <QLabel>
3   #include <QHBoxLayout>
4   #include <QLineEdit>
5
6   int main(int argc, char** argv)
7   {
8           QApplication app(argc, argv);
9           QLabel* label = new QLabel("Hello");
10          QLineEdit* edit = new QLineEdit;
11          QWidget* window = new QWidget;
12          QHBoxLayout* layout = new QHBoxLayout;
13
14          QObject::connect(edit,SIGNAL(textChanged(QString)),
15                           label,SLOT(setText(QString)) );
16          QObject::connect(edit,SIGNAL(textChanged(QString)),
17                           window,SLOT(setWindowTitle(QString)) );
18
19          layout->addWidget(edit);
20          layout->addWidget(label);
21          window->setLayout(layout);
```

```
22          window->show();
23
24          return app.exec();
25  }
```

Signal/Slot のイメージ図は図 3.4 となります。

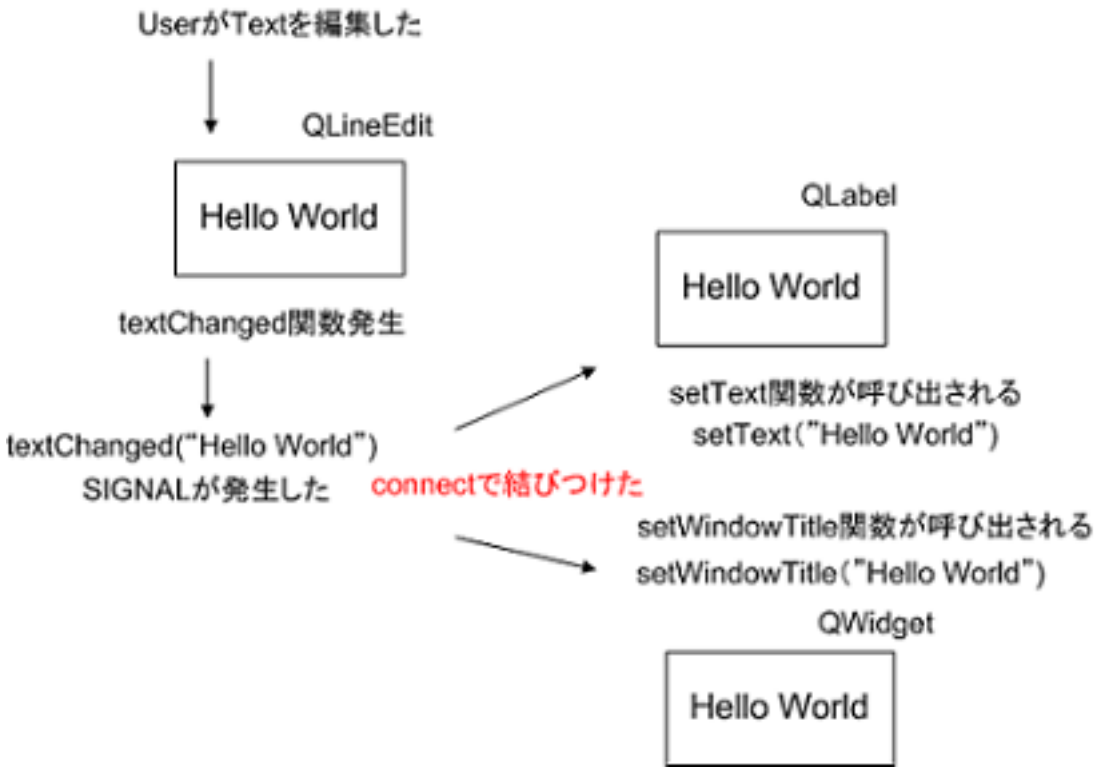

図 3.4: Signal/Slot のイメージ図

14,15 行目では edit の textChanged 関数と label の setText 関数を結び付けています。QString というのは、C++でいう string と同じです。textChanged() だけではダメです。ちゃんと引数が付いている textChanged(QString) を呼び出さなければなりません。そして、setText 関数で label に edit で編集された文字列をセットしています。

16,17 行目も同様に setWindowTitle 関数で window の Title に edit で編集された文字列をセットしています。

3.3 Connectionの解除

いままで connect 関数を使い、Signal と Slot を設定していました。しかし、場合によってはセットした connect を解除したい場合があるかもしれません。そういった場合、disconnect 関数を使います。使い方は connect とほとんど同じです。

3.1 節のコードに disconnect 関数を付け加えたものが次のコードです。

```
1   #include <QApplication>
2   #include <QPushButton>
3
4   int main(int argc, char** argv)
5   {
6           QApplication app(argc, argv);
7           QPushButton* button = new QPushButton("Quit");
8           QObject::connect(button, SIGNAL( clicked() ),
9                            &app, SLOT(quit()) );
10          QObject::disconnect(button, SIGNAL( clicked() ),
11                           &app, SLOT(quit()) );
12          button->show();
13          return app.exec();
14  }
```

connect の部分が disconnect になっただけです。これでコネクションを解除することができます。もちろん、このプログラムは実行してボタンを押しても何も起こりません。

3.4 Signal/Slotのまとめ

connect 関数

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot) );
```

sender：信号が発生する部品のアドレスを渡す
SIGNAL(signal)：signal に信号とする関数を渡す
receiver：信号を受け取る部品のアドレスを渡す
SLOT(slot)：信号を受け取った際に呼び出す関数を渡す

第4章　プログラム作成への準備

4.1　ダイアログを作ってみよう

今までコードはすべて main 関数の中に書いてきましたが、このままコードを拡張していったら main 関数が 1000 行くらいになってしまうかもしれません。

ということで、今回からはクラスを使って関数を分割していくことにします。また、ソースファイルも 01.cpp と main.h ファイルに分割します。

まず、今回作成するプログラムの実行結果は、図 4.1, 4.2 のようになります。

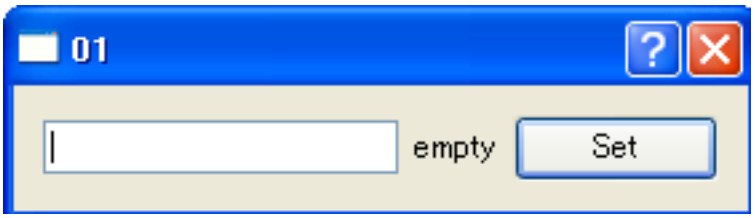

図 4.1: 初期状態 (on Windows)

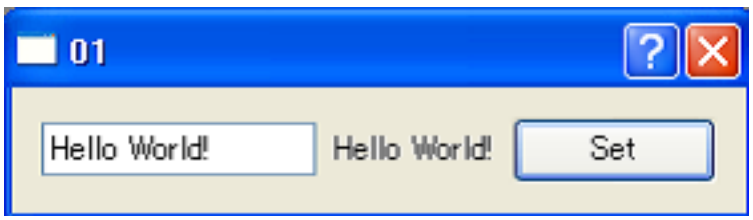

図 4.2: ボタンを押した後 (on Windows)

起動した直後は図 4.1 の状態です。そして、テキストに何か文章を書き set ボタンを押すと、ラベルにテキストの内容がコピーされます（図 4.2)。

このプログラムは次のようなコードとなります。

```
1  //main.h
2
3  #ifndef MAIN_H_
4  #define MAIN_H_
5
6  #include <QDialog>
7
8  class QLabel;
9  class QPushButton;
10 class QLineEdit;
11
12 class MainDialog : public QDialog
13 {
14         Q_OBJECT
15 public:
16         MainDialog(QWidget* parent = 0);
17 private slots:
18         void setLabelText();
19 private:
20         QLabel* label;
21         QPushButton* setButton;
22         QLineEdit* lineEdit;
23 };
24
25 #endif
```

```
1  //01.cpp
2
3  #include <QtGui>
4  #include "main.h"
5
6  MainDialog::MainDialog(QWidget* parent)
7          : QDialog(parent)
8  {
9          label = new QLabel(tr("empty") );
10         setButton = new QPushButton(tr("Set") );
11         lineEdit = new QLineEdit;
```

```
12
13          connect(setButton,SIGNAL(clicked() ),this,SLOT(setLabelText() ) );
14
15          QHBoxLayout* layout = new QHBoxLayout;
16          layout->addWidget(lineEdit);
17          layout->addWidget(label);
18          layout->addWidget(setButton);
19          setLayout(layout);
20  }
21
22  void MainDialog::setLabelText()
23  {
24          QString text = lineEdit->text();
25          label->setText(text);
26  }
27
28  int main(int argc, char** argv)
29  {
30          QApplication app(argc,argv);
31          MainDialog* dialog = new MainDialog;
32          dialog->show();
33          return app.exec();
34  }
```

今回のコードはかなり長いですね。では、main.h ファイルのコードから見ていきましょう。

まず 3,4 行目のコードは C 言語や C++を使ってやや規模の大きなプログラムを作ったことのある人なら知っていると思います。この方法はインクルードガードと呼ばれています。知らない方はインターネットや書籍で勉強してみてください。Visual C++などの`#pragma once` と全く同じです。

6 行目では QDialog ヘッダをインクルードしています。今回はダイアログを作りたいので、このヘッダファイルを使います。

8,9,10 行目では QLabel,QPushButton,QLineEdit のプロトタイプ宣言です。この宣言では QLabel などのクラスの詳細はここでは定義していないけれど、他のヘッダファイルなどにはこのクラスが存在しているよ、ということをコンパイラ側に伝えています。

普通に`#include <QLabel>`などでヘッダファイルをインクルードしてもよいの

ですが、8〜10 行目のように宣言することでいちいちヘッダファイルへクラスを探しにいく手間が省け、コンパイル時間を短縮することができます。

12 行目で MainWindow クラスを作成しています。このクラスがメインのダイアログとなります。ダイアログを作成する場合は QDialog を継承する事により、基本的なダイアログの機能を引き継ぐことができます。よって拡張したい機能だけを自分で付け足せばよいわけです。（図 4.3）

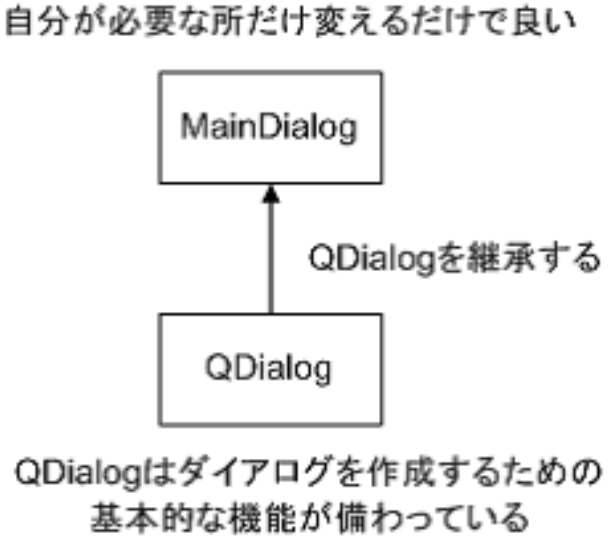

図 4.3: 継承のイメージ図

14 行目の `Q_OBJECT` は 17 行目で slots を使うためのマクロです。最後にセミコロンが付かない事に注意してください。

17 行目の `private slots:` はその先の setLabelText 関数が Slot として呼ばれる事があるという意味です。このようにする事で connect 関数を使って setLabelText 関数をどこかの Signal と結びつけることができます。他にも Signal として呼ばれる可能性のある関数には `signals:` としておきます。`signals:` は `private signals:` や `public signals:` にはならない事に注意してください。

次に 01.cpp ファイルの説明に入ります。

まず 3 行目では`<QtGui>`をインクルードしています。これは QLabel や QPushButton などといった Qt の GUI に関わる部分のヘッダファイルをすべてインクルードしているのと同じです。[1]クラスの規模が大きくなると、どのヘッダファイルをインクルードしたらよいのか分からなくなってしまいます。そういった事をいちいち考えなくてもよくなるので QtGui はたいへん便利です。

[1]正確には QtCore と QtGui の部分をインクルードしている

7 行目では QDialog のコンストラクタに親へのアドレスを渡しています。parent を 0 にした場合は、その部品が新しいウィンドウとして表示されます。つまり親となるわけです。

13 行目では connect 関数を使ってボタンが押されたら setLabelText 関数が呼び出されるようにしています。QDialog を継承すると connect 関数の QObject::というプリフィクスが不要となります。[2]

22 行目の setLabelText 関数では lineEdit の内容を label にセットしています。以上で説明は終了です。他に部品を追加したりしていろいろ試してみてください。

4.2 モーダルダイアログを作ってみよう

複数のウィンドウを表示したい場合があると思います。そういった場合に用いられるのが、モーダルダイアログとモードレスダイアログです。

モーダルダイアログとして新しいダイアログが呼び出された場合、呼び出した側のダイアログはモーダルダイアログが表示されている間は操作をすることができません。モードレスダイアログではそういった制限は特にありません。

今回はモーダルダイアログを作っていきたいと思います。完成したプログラムは図 4.4〜4.6 となります。

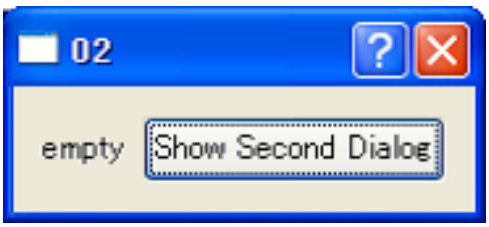

図 4.4: 初期状態 (on Windows)

まず始めに図 4.4 のウィンドウが表示されます。Show Second Dialog ボタンを押すと図 4.5 のモーダルダイアログが表示されます。そして、モーダルダイアログの中にあるテキストに何か文字を書き込み OK ボタンを押すと、始めにあったダイアログのラベルにテキストの文字がセットされます。(図 4.6)

このプログラムは次のようなソースコードとなります。今回はファイルを 5 分割してあり、コードも長くなっています。しかしながら決して難しくは無いと思います。

[2] これは QDialog が QObject から継承されている為です。更にいうと、QDialog は QWidget から継承されており、QWidget が QObject から継承されています。

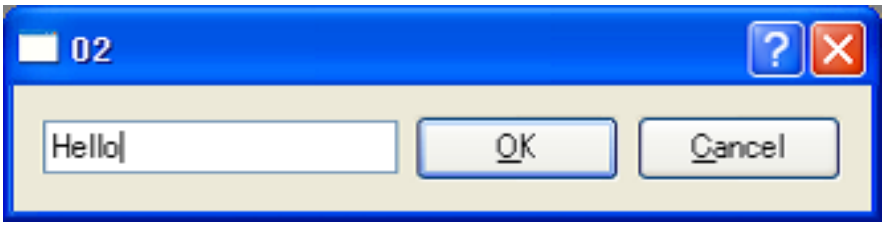

図 4.5: ボタンを押した後に表示されたモーダルダイアログ (on Windows)

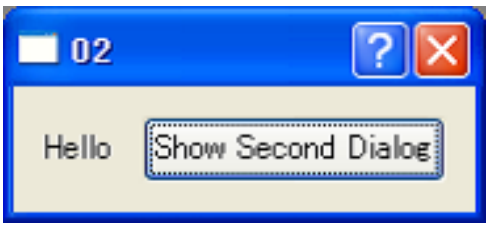

図 4.6: モーダルダイアログの ok ボタンを押した後 (on Windows)

```
1   //main.cpp
2
3   #include <QtGui>
4   #include "MainDialog.h"
5   #include "SecondDialog.h"
6
7   int main(int argc, char** argv)
8   {
9           QApplication app(argc, argv);
10          MainDialog* dialog = new MainDialog;
11          dialog->show();
12          return app.exec();
13  }
```

```
1   //MainDialog.h
2
3   #ifndef MAINDIALOG_H_
4   #define MAINDIALOG_H_
5
6   #include <QDialog>
```

```
 7
 8  class QPushButton;
 9  class QLabel;
10
11  class MainDialog : public QDialog
12  {
13          Q_OBJECT
14  public:
15          MainDialog(QWidget* parent = 0);
16  private slots:
17          void showSecondDialog();
18  private:
19          QPushButton* showDialogButton;
20          QLabel* textLabel;
21  };
22
23  #endif
```

```
 1  //MainDialog.cpp
 2
 3  #include <QtGui>
 4  #include "MainDialog.h"
 5  #include "SecondDialog.h"
 6
 7  MainDialog::MainDialog(QWidget* parent) : QDialog(parent)
 8  {
 9          showDialogButton = new QPushButton("Show Second Dialog");
10          textLabel = new QLabel("empty");
11          connect(showDialogButton,SIGNAL(clicked()),
12                          this,SLOT(showSecondDialog()) );
13
14          QHBoxLayout* layout = new QHBoxLayout;
15          layout->addWidget(textLabel);
16          layout->addWidget(showDialogButton);
17          setLayout(layout);
18  }
19
```

```
20  void MainDialog::showSecondDialog()
21  {
22          SecondDialog secondDialog(this);
23          if(secondDialog.exec()) {
24                  QString str = secondDialog.getLineEditText();
25                  textLabel->setText(str);
26          }
27  }
```

```
1   //SecondDialog.h
2
3   #ifndef SECONDDIALOG_H_
4   #define SECONDDIALOG_H_
5
6   #include <QDialog>
7
8   class QPushButton;
9   class QLineEdit;
10
11  class SecondDialog : public QDialog
12  {
13          Q_OBJECT
14  public:
15          SecondDialog(QWidget* parent = 0);
16          QString getLineEditText();
17  private:
18          QPushButton* okButton;
19          QPushButton* cancelButton;
20          QLineEdit* editor;
21  };
22
23  #endif
```

```
1   //SecondDialog.cpp
2
3   #include <QtGui>
4   #include "SecondDialog.h"
```

```
5
6   SecondDialog::SecondDialog(QWidget* parent) : QDialog(parent)
7   {
8           okButton = new QPushButton(tr("&OK") );
9           cancelButton = new QPushButton(tr("&Cancel") );
10          editor = new QLineEdit;
11
12          QHBoxLayout* layout = new QHBoxLayout;
13          layout->addWidget(editor);
14          layout->addWidget(okButton);
15          layout->addWidget(cancelButton);
16          setLayout(layout);
17
18          connect(okButton, SIGNAL(clicked()), this, SLOT(accept()) );
19          connect(cancelButton,SIGNAL(clicked()), this, SLOT(reject()) );
20  }
21
22  QString SecondDialog::getLineEditText()
23  {
24          return editor->text();
25  }
```

まず、main.cpp ですが、これはもう見てわかると思うので説明は省略します。

MainDialog.h ではメインのダイアログクラスの定義を行っています。

MainDialog.cpp のコンストラクタの中にある connect は、Show Second Dialog ボタンが押された時に showSecondDialog 関数が呼び出されるように設定しています。

今回新しく出てくるところは、showSecondDialog 関数内のコードです。

まず、22 行目では 2 つ目のダイアログである secondDialog 変数を作成しています。引数に this を渡していますが、これは secondDialog.cpp のコンストラクタの定義を見てわかると思います。MainDialog を secondDialog の親にするということです。

次に出てくる secondDialog.exec() ですが、これは secondDialog をモーダルダイアログとして呼び出すということです。モーダルダイアログとして呼び出しているので、secondDialog が表示されている間は MainDialog 側を操作することができません。

また、secondDialog.exec() 関数は QDialog::Accepted または QDialog::Rejected

を返します。この QDialog::Accepted は 1、QDialog::Rejected は 0 と定義されています。つまり、Accepted が返された場合は 24,25 行目が実行され、Rejected が返された場合は 24,25 行目は実行されません。

SecondDialog.h は 2 つ目のダイアログクラスを定義しています。

SecondDialog.cpp の 18,19 行目を見てください。ここで okButton を押した場合、accept 関数を呼び出し、cancelButton を押した場合、reject 関数を呼び出すよう設定しています。accept 関数も reject 関数もウィンドウを隠すという動作を行います。ただ、MainDialog.cpp の 23 行目で呼び出した exec 関数の返却値を QDialog::Accepted にするか QDialog::Rejected にするかが違います。

4.3 モードレスダイアログを作ってみよう

今度はモードレスダイアログを作ってみましょう。モードレスダイアログはモーダルダイアログのように他のウィンドウが操作できないなどの制約は特にありません。

モードレスダイアログのコードは次のようになります。ボタンの配置などは 4.2 節のコードは同じです。

```
1   //main.cpp
2
3   #include <QtGui>
4   #include "MainDialog.h"
5   #include "SecondDialog.h"
6
7   int main(int argc, char** argv)
8   {
9           QApplication app(argc, argv);
10          MainDialog* dialog = new MainDialog;
11          dialog->show();
12          return app.exec();
13  }
```

```
1   //MainDialog.h
2
3   #ifndef MAINDIALOG_H_
4   #define MAINDIALOG_H_
5
```

```
6   #include <QDialog>
7
8   class QPushButton;
9   class QLabel;
10  class SecondDialog;
11
12  class MainDialog : public QDialog
13  {
14          Q_OBJECT
15  public:
16          MainDialog(QWidget* parent = 0);
17  public slots:
18          void showSecondDialog();
19          void setTextLabel();
20  private:
21          QPushButton* showDialogButton;
22          QLabel* textLabel;
23          SecondDialog* secondDialog;
24  };
25
26  #endif
```

```
1   //MainDialog.cpp
2
3   #include <QtGui>
4   #include "MainDialog.h"
5   #include "SecondDialog.h"
6
7   MainDialog::MainDialog(QWidget* parent) : QDialog(parent), secondDialog(NULL)
8   {
9           showDialogButton = new QPushButton("Show Second Dialog");
10          textLabel = new QLabel("empty");
11          connect(showDialogButton, SIGNAL(clicked()),
12                                    this, SLOT(showSecondDialog()) );
13
14          QHBoxLayout* layout = new QHBoxLayout;
15          layout->addWidget(textLabel);
```

```
16          layout->addWidget(showDialogButton);
17          setLayout(layout);
18  }
19
20  void MainDialog::showSecondDialog()
21  {
22          if(!secondDialog){
23                  secondDialog = new SecondDialog;
24                  connect(secondDialog, SIGNAL(okButtonClicked() ),
25                                          this, SLOT(setTextLabel()) );
26          }
27          if(secondDialog->isHidden() ) {
28                  secondDialog->show();
29          }else{
30                  secondDialog->activateWindow();
31          }
32  }
33
34  void MainDialog::setTextLabel()
35  {
36          QString str = secondDialog->getLineEditText();
37          textLabel->setText(str);
38  }
```

```
1   //SecondDialog.h
2
3   #ifndef SECONDDIALOG_H_
4   #define SECONDDIALOG_H_
5
6   #include <QDialog>
7
8   class QPushButton;
9   class QLineEdit;
10  class QString;
11
12  class SecondDialog : public QDialog
13  {
```

```
14          Q_OBJECT
15  public:
16          SecondDialog(QWidget* parent = 0);
17          QString getLineEditText();
18  signals:
19          void okButtonClicked();
20  private:
21          QPushButton* okButton;
22          QPushButton* cancelButton;
23          QLineEdit* editor;
24  };
25
26  #endif
```

```
1   //SecondDialog.cpp
2
3   #include <QtGui>
4   #include "SecondDialog.h"
5
6   SecondDialog::SecondDialog(QWidget* parent) : QDialog(parent)
7   {
8           okButton = new QPushButton(tr("&OK") );
9           cancelButton = new QPushButton(tr("&Cancel") );
10          editor = new QLineEdit;
11
12          QHBoxLayout* layout = new QHBoxLayout;
13          layout->addWidget(editor);
14          layout->addWidget(okButton);
15          layout->addWidget(cancelButton);
16          setLayout(layout);
17
18          connect(okButton,SIGNAL(clicked()),this,SIGNAL(okButtonClicked()) );
19          connect(okButton,SIGNAL(clicked()), this, SLOT(close()) );
20          connect(cancelButton,SIGNAL(clicked()), this, SLOT(close()) );
21  }
22
23  QString SecondDialog::getLineEditText()
```

```
24  {
25          return editor->text();
26  }
```

まず、MainDialog.cpp の showSecondDialog 関数を見てください。この関数は Show Second Dialog ボタンを押すことによって呼び出されます。

22 行目の if 文の中身は secondDialog が NULL であるときに呼び出されます。つまりはじめて secondDialog を呼び出すときに実行されます。

24 行目では secondDialog の okButtonClicked 関数が呼び出されたときに setTextLabel 関数が呼び出されます。okButtonClicked 関数は SecondDialog.h に定義されています。この関数の説明はもう少し先で説明します。

setTextLabel 関数では、secondDialog にあるテキストの中身を MainDialog のラベルにセットしています。getLineEditText 関数も SecondDialog.h および SecondDialog.cpp に定義されています。

SecondDialog.h の 18,19 行目及び SecondDialog.cpp の 18,19 行目を見てください。SecondDialog.h の 18 行目の `signals:` は 19 行目の okButtonClicked 関数をシグナル関数とするためのマクロです。この okButtonClicked 関数の中身は Qt が勝手に作成します。よって定義のみ書けばよいです。

そして SecondDialog.cpp の 18 行目で okButton が押されたときに okButtonClicked 関数が呼び出されるようにセットしています。今までは connect 関数の第 4 引数は SLOT としていましたが、18 行目のように SIGNAL を呼び出すこともできます。

そして 19,20 行目では okButton 及び cancelButton が押されたときにダイアログが閉じるよう close 関数を connect しています。

最後の getLineEditText 関数では secondDialog 内のテキストの中身を取得するための関数です。

第5章　Qt Designerを使ってみよう

5.1　Qt Designerとは

この章ではQt Designerと呼ばれるGUIアプリケーションを簡単に作成するためのツールの基本的な使い方を説明していきたいと思います。今まではWidget（部品）をどのように配置するのかはコードの上で書いていました。しかし以外とこの作業は面倒なうえ、どのような仕上がりになっているかはコンパイルしてプログラムを実行するまでわかりませんでした。

この問題を解決するためのツールがQt Designerです。まずはQt Designerを使ってどのようなことができるかを見ていきましょう。

5.2　Qt Designerの起動

ではDesignerを起動してみましょう。

Windowsを使っている方は大体は、スタート → Qt by Trolltech v4.3.1 (Open-Source) → Designer の順で起動できると思います。Linuxを使っている方はコマンドラインにて designer と入力することで起動できます。

起動したらDialog without Buttonsを選択してOKを押しましょう。（図5.1）真ん中に表示されているDialog -untitled という部分がダイアログです。ここに左側にあるウィジェットボックスから部品をドラッグアンドドロップして貼り付けます。

5.3　部品の配置

では、ウィジェットボックスからLine Edit,Label,Push Buttonを使って図5.2のように配置してみましょう。Push Buttonをダブルクリックするとボタンに表示される文字を変えることができます。また、&Ok とするとOのしたに下線を引くことができます。

次にボタンの位置などを整えてみましょう。

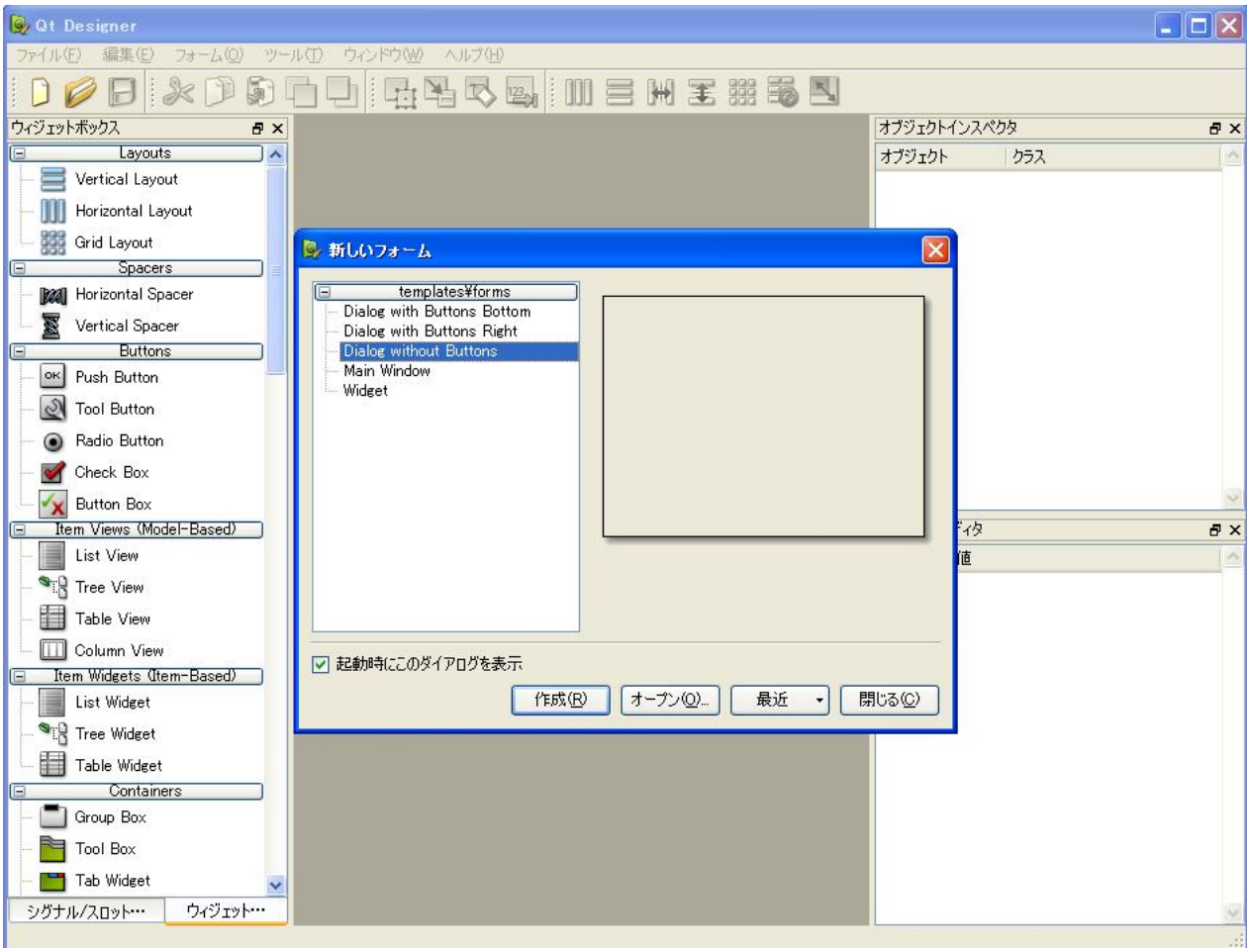

図 5.1: Qt Designer の起動画面 (on Windows)

図 5.3 を見てください。図の左から赤い四角で囲まれている順に、水平に並べる、垂直に並べる、格子状に並べる、サイズ調節となっています。水平に並べるはいままで使ってきた QHBoxLayout にあたります。同様に垂直に並べる、格子状に並べるも QVBoxLayout、QGridLayout と同じです。

Ok ボタンと Cancel ボタン二つを選択して垂直に並べるを押してください。赤い四角で囲まれ、綺麗にボタンが整えられたと思います。

次にダイアログ内部の空いている所をクリックして水平に整えるボタンを押してください。そして最後にサイズ調節ボタンを押してください。

図 5.4 のように配置できたでしょうか。これで部品の配置は終わりです。メニューバーのフォーム → プレビュー を押してみてください。そうすると実際にプログラムが実行された際に表示されるダイアログを見ることができます。

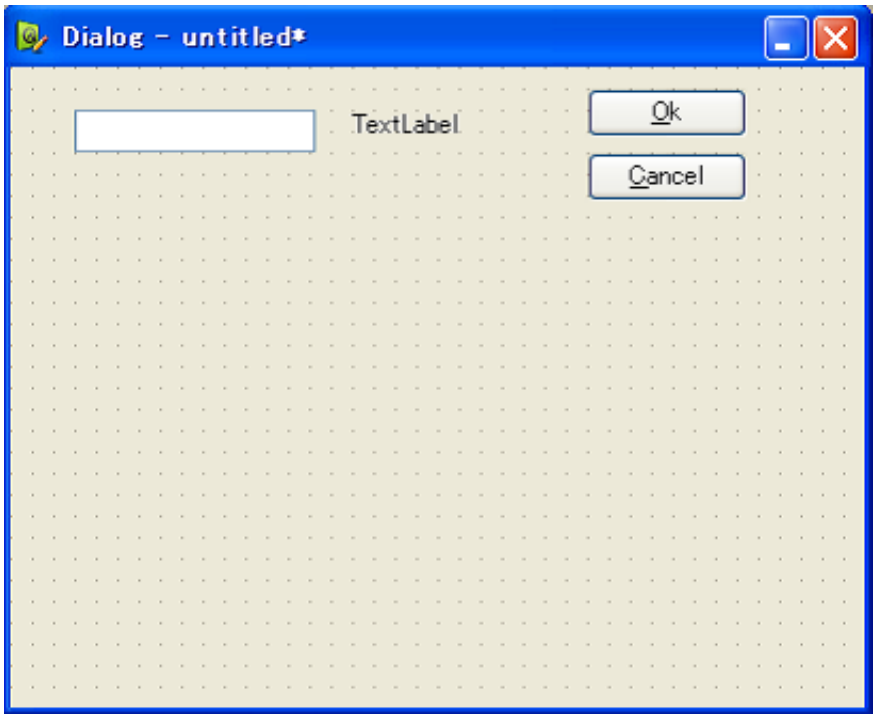

図 5.2: Qt Designer(on Windows)

ここで Label をもう少し大きくしたいと感じた方もいるかも（？）しれません。その場合、Label の幅は最低このくらいで高さがこのくらいといった設定も行うことができます。

まず Label をクリックしてプロパティを見てください。（図 5.5）プロパティの中に minimumSize という設定を行うところがあります。ここで width を 100,height を 0 に設定してみましょう。この設定を行うことで、ウィンドウのサイズが変わっても Label が幅 100 以下になることはありません。

同様にして line Edit の方も minimumSize を幅 100, 高さ 0 に設定してみましょう。

5.4 Signal/Slot を設定してみよう

部品を配置した後は Signal/Slot を設定してみましょう。Qt Designer では基本的なシグナル/スロットは Designer 上で作成することができます。もちろん、自分で作った独自の機能を使いたい場合はソースコードを変更する必要があります。

図 5.6 の赤い四角で囲まれているボタンを押してください。このボタンを押す事によりシグナル/スロット編集モードに入ります。

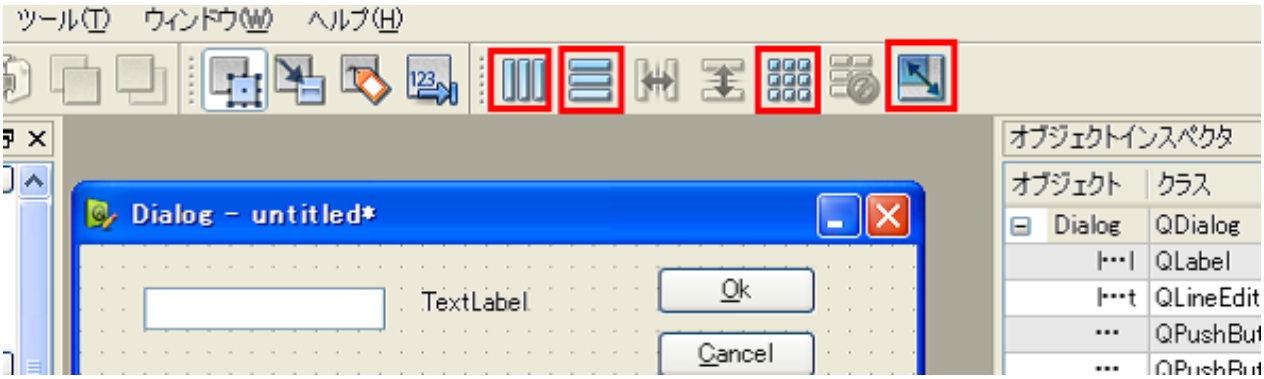

図 5.3: Qt Designer(on Windows)

図 5.4: Qt Designer(on Windows)

また、シグナル/スロットエディタというウィンドウも表示させてください。図 5.7 のようにシグナル/スロットエディタにチェックを入れることで表示することができます。

では、まずは Ok ボタンをクリックしたら (Signal:clicked 関数が発生したら）ダイアログが終了する (Slot:accept 関数が発生する）ように設定してみましょう。ok ボタンをクリックしながら、ダイアログの上でボタンを離してください。

そうすると、シグナル/スロット接続を設定するウィンドウが出てくると思います。図 5.8 のように設定してみてください。

同様に Cancel ボタンをクリックしたら reject 関数が発生するように設定してみましょう。

図 5.9 のようになったでしょうか。設定できたら、プレビューを押して動作を確認してみましょう。Ok または Cancel ボタンを押すとダイアログが終了すると思います。

では、他にも lineEdit が編集されたら label に lineEdit の内容が表示されるように

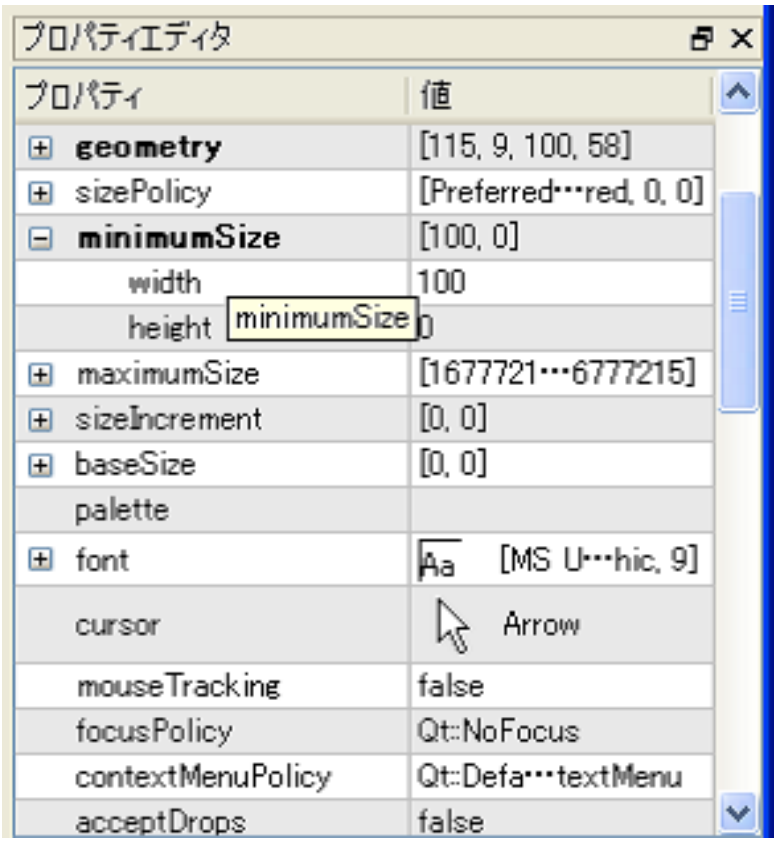

図 5.5: Qt Designer(on Windows)

してみましょう。lineEdit から label に赤い矢印を持っていき、lineEdit の textChanged(QString) 関数をシグナル、label の setText(QString) をスロットとしてみましょう。

また、lineEdit の textChanged(QString) をシグナルとしてダイアログの setWindowTitle(QString) をスロットとしてみましょう。すべての関数を表示したい場合はすべてのシグナルとスロットを表示にチェックを入れてください。

図 5.10 のようになったでしょうか。プレビューで実行し、lineEdit の中身を編集すると label とダイアログのタイトルに lineEdit と同じ内容が表示されれば成功です。

図 5.6: Qt Designer(on Windows)

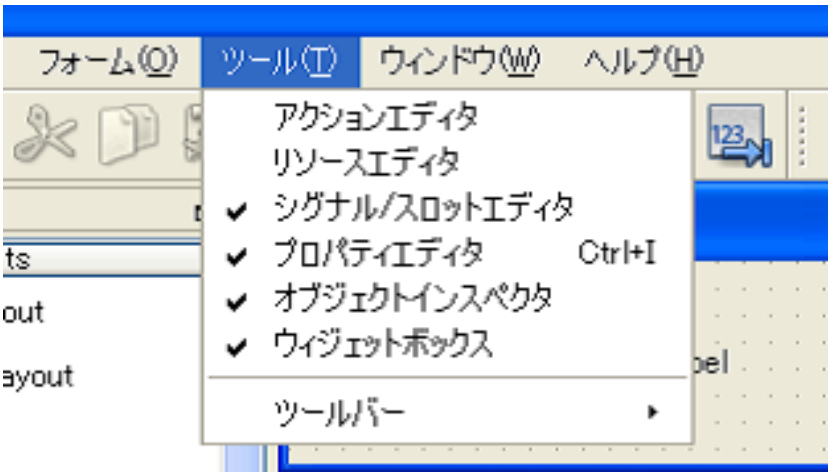

図 5.7: Qt Designer(on Windows)

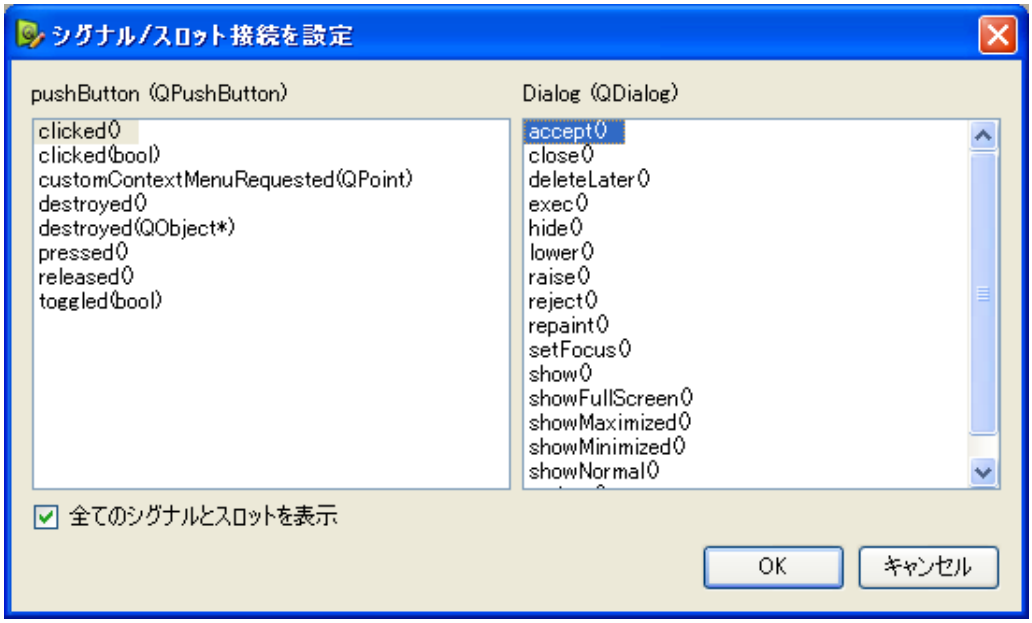

図 5.8: Qt Designer(on Windows)

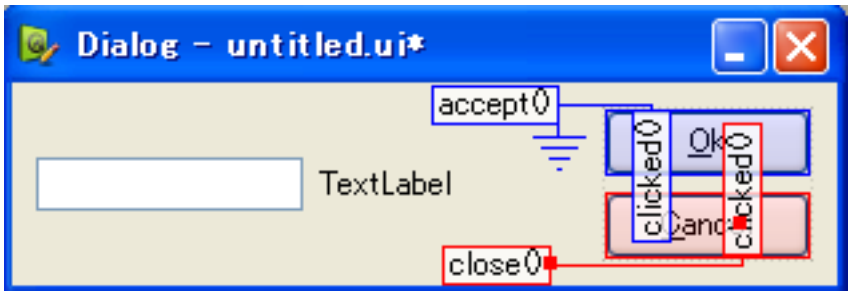

図 5.9: Qt Designer(on Windows)

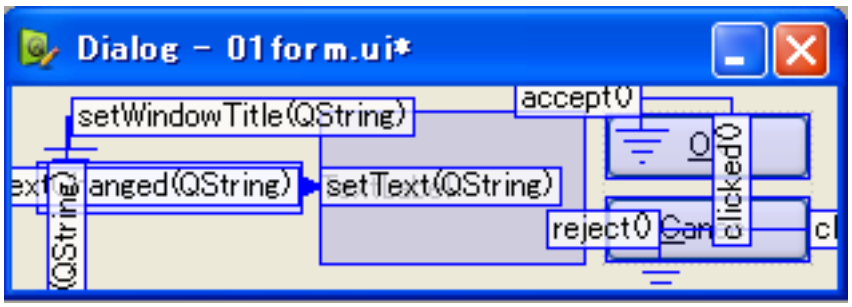

図 5.10: Qt Designer(on Windows)

参考文献

[1] Jasmin Blanchette & Mark Summerfield, C++ GUI Programming with Qt4.

[2] Trolltech, Qt Assistant Tutorial and Examples Qt Tutorial

[3] Trolltech, Qt Assistant All Classes

[4] Trolltech, Qt Assistant Core Features Layout Management