姓名：章崇文　　学号：202202296　　班级：计算机 222

实验三、嵌入式 Linux GUI 编程实验

一、实验目的

了解 Qt 类库的使用方法，熟悉 Qt 的嵌入式开发方法。

二、实验基本要求

1. 掌握 Qt 的信号/槽机制。
2. 编写简单的 Qt 程序，分别采用 qtmake 和 qtemake 编译生成在虚拟机运行和实验箱运行的可执行程序。

三、实验原理

1．QT 简介

QT 是一个跨平台的 C++ GUI 应用构架，它提供了丰富的窗口部件集，具有面向对象、易于扩展、真正的组件编程等特点，更为引人注目的是目前 Linux 上最为流行的 KDE 桌面环境就是建立在 QT 库的基础之上。Qt 在刚出现的时候，对于 Linux 和 Unix 系统，只有构建于 Xlib 之上的 Qt X11 版。但随着 Linux 操作系统在嵌入式领域的应用日渐广泛，Qt 推出了嵌 入式的版本 Qt-embedded。尽管称做 Qt-embedded，其实不仅可以生成在开发板运行的 ARM 版，也可以编译生成 PC 机上运行的 X86 版本。

2．信号和槽机制

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 QT 的核心特性，也是 QT 区别于其它工具包的重要地方。信号和槽是 QT 自行定义的一种通信机制，它独立于标准的 C/C++ 语言，因此要正确的处理信号和槽，必须借助一个称为 moc（Meta Object Compiler）的 QT 工具，该工具是一个 C++ 预处理程序，它为高层次的事件处理自动生成所需要的附加代码。在我们所熟知的很多 GUI 工具包中，窗口小部件 (widget) 都有一个回调函数用于响应它们能触发的每个动作，这个回调函数通常是一个指向某个函数的指针。但是，在 QT 中信号和槽取代了这些凌乱的函数指针，使得我们编写这些通信程序更为简洁明了。 信号和槽能携带任意数量和任意类型的参数，他们是类型完全安全的，不会像回调函数那样产生 core dumps。

所有从 QObject 或其子类 (例如 Qwidget) 派生的类都能够包含信号和槽。当对象改变其状态时，信号就由该对象发射 (emit) 出去，这就是对象所要做的全部事情，它不知道另一端是谁在接收这个信号。这就是真正的信息封装，它确保对象被当作一个真正的软件组件来使用。槽用于接收信号，但它们是普通的对象成员函数。一个槽并不知道是否有任何信号与自己相连接。而且，对象并不了解具体的通信机制。

你可以将很多信号与单个的槽进行连接，也可以将单个的信号与很多的槽进行连接，甚

至于将一个信号与另外一个信号相连接也是可能的，这时无论第一个信号什么时候发射系统都将立刻发射第二个信号。总之，信号与槽构造了一个强大的部件编程机制。

（1）信号

当某个信号对其客户或所有者发生的内部状态发生改变，信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时，与其相关联的槽将被立刻执行，就象一个正常的函数调用一样。信号-槽机制完全独立于任何 GUI 事件循环。只有当所有的槽返回以后发射函数（emit）才返回。 如果存在多个槽与某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地 执行，但是它们执行的顺序将会是随机的、不确定的，我们不能人为地指定哪个先执行、哪个后执行。

信号的声明是在头文件中进行的，QT 的 signals 关键字指出进入了信号声明区，随后即可声明自己的信号。例如，下面定义了三个信号：

```
signals:
    void mySignal();
    void mySignal(int x);
    void mySignal(int x,int y);
```

在上面的定义中，signals 是 QT 的关键字，而非 C/C++ 的。接下来的一行 void mySignal() 定义了信号 mySignal，这个信号没有携带参数；接下来的一行 void mySignal(int x) 定义 了重名信号 mySignal，但是它携带一个整形参数，这有点类似于 C++ 中的虚函数。从形式上 讲信号的声明与普通的 C++ 函数是一样的，但是信号却没有函数体定义，另外，信号的返回 类型都是 void，不要指望能从信号返回什么有用信息。

信号由 moc 自动产生，它们不应该在 .cpp 文件中实现。

（2）槽

槽是普通的 C++ 成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值。

既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++ 成员函数一样，槽函数也分为三种类型，即 public slots、private slots 和 protected slots。

public slots：在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。

protected slots：在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。

private slots：在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

槽也能够声明为虚函数，这也是非常有用的。

槽的声明也是在头文件中进行的。例如，下面声明了三个槽：

```
public slots:
```

```
    void mySlot();
    void mySlot(int x);
    void mySignalParam(int x,int y);
```

（3）信号与槽的关联

通过调用 QObject 对象的 connect 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：

```
bool QObject::connect ( const QObject * sender, const char * signal,
const QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。当指定信号 signal 时必须使用 QT 的宏 SIGNAL()，当指定槽函数时必须使用宏 SLOT()。如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。

例如，下面定义了两个对象：标签对象 label 和滚动条对象 scroll，并将 valueChanged() 信号与标签对象的 setNum() 相关联，另外信号还携带了一个整形参数，这样标签总是显示滚动条所处位置的值。

```
QLabel      *label   = new QLabel;
QScrollBar *scroll = new QScrollBar;
QObject::connect( scroll, SIGNAL(valueChanged(int)), label,   SLOT(setNum(int)) );
```

一个信号甚至能够与另一个信号相关联，看下面的例子：

```
class MyWidget : public QWidget
{
public:
    MyWidget();
...
signals:
    void aSignal();
...
private:
...
    QPushButton *aButton;
};
MyWidget::MyWidget()
{
    aButton = new QPushButton( this );
    connect( aButton, SIGNAL(clicked()), SIGNAL(aSignal()) );
}
```

在上面的构造函数中，MyWidget 创建了一个私有的按钮 aButton，按钮的单击事件产生的信号 clicked() 与另外一个信号 aSignal() 进行了关联。这样一来，当信号 clicked() 被

发射时，信号 aSignal() 也接着被发射。当然，你也可以直接将单击事件与某个私有的槽函数相关联，然后在槽中发射 aSignal() 信号，这样的话似乎有点多余。

当信号与槽没有必要继续保持关联时，我们可以使用 disconnect 函数来断开连接。其定义如下：

bool QObject::disconnect (const QObject * sender, const char * signal,
const Object * receiver, const char * member) [static]

这个函数断开发射者中的信号与接收者中的槽函数之间的关联。

有三种情况必须使用 disconnect() 函数：

断开与某个对象相关联的任何对象。这似乎有点不可理解，事实上，当我们在某个对象中定义了一个或者多个信号，这些信号与另外若干个对象中的槽相关联，如果我们要切断这些关联的话，就可以利用这个方法，非常之简洁。

disconnect(myObject, 0, 0, 0)

或者

myObject->disconnect()

断开与某个特定信号的任何关联。

disconnect(myObject, SIGNAL(mySignal()), 0, 0)

或者

myObject->disconnect(SIGNAL(mySignal()))

断开两个对象之间的关联。

disconnect(myObject, 0, myReceiver, 0)

或者

myObject->disconnect(myReceiver)

在 disconnect 函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数。但是发射者 sender 不能为 0，其它三个参数的值可以等于 0。

3. Qmake 的使用

qmake 是 Trolltech 公司创建的用来为不同的平台和编译器书写 Makefile 的工具。手写 Makefile 是比较困难并且容易出错的，尤其是需要给不同的平台和编译器组合写几个 Makefile。使用 qmake，开发者创建一个简单的“项目”文件并且运行 qmake 生成适当的 Makefile。qmake 会注意所有的编译器和平台的依赖性，可以把开发者解放出来只关心他们的代码。Trolltech 公司使用 qmake 作为 Qt 库和 Qt 所提供的工具的主要连编工具。qmake 也注意了 Qt 的特殊需求，可以自动的包含 moc 和 uic 的连编规则。

命令格式：

```
qmake [mode] [options] files
```

qmake 支持两种不同模式的操作,默认情况下,qmake 将会使用 project 中的配置来生成 Makefile 文件,但是也可以用 qmake 生成 pro 文件.如果你想明确的设置选项,你必须在所有其他的选项前指定,模式可以是下面的其中之一的值：

-makefile

qmake 将输出一个 Makefile 文件.

-project

qmake 将输出一个 pro 文件.

4. Ubuntu 系统 Qt 与嵌入式 Qt 开发方法

Ubuntu 系统提供了本机的 Qt 库和嵌入式 Qt 库，分别位于/opt/FriendlyARM/qt 和/opt/FriendlyARM/qte 下。Ubuntu 系统已经将 Qt 和 Qt/E 开发工具 qmake 通过修改 PATH 变量，将位于/opt/FriendlyARM/qt/bin 和/opt/FriendlyARM/qte/bin 两个文件夹加入到 PAHT 变量中，并将/opt/FriendlyARM/qte/bin 中的 qmake 通过软连接命令 ln 修改为 qtemake，将/opt/FriendlyARM/qt/bin 中的 qmake 通过软连接命令 ln 修改为 qtmake，以区分系统 qmake 和嵌入式开发 qmake。

因此，Ubuntu 系统给出两个 qmake 版本:

(1) qtmake

(2) qtemake

这是为了区分系统中的 qmake 而对 qmake 程序的重命名。通过 qtmake 和 qtemake 两个程序可以编译生成基于本机的和 ARM 的 Qt 程序。

5. 实验箱系统嵌入式 Qt 运行方法

实验箱 Linux 系统在启动时默认运行了一个 Qt demo，如在实验箱的使用过程中运行基于界面的 Qt 程序时，首先通过触摸将默认启动的程序结束，再运行如下命令。

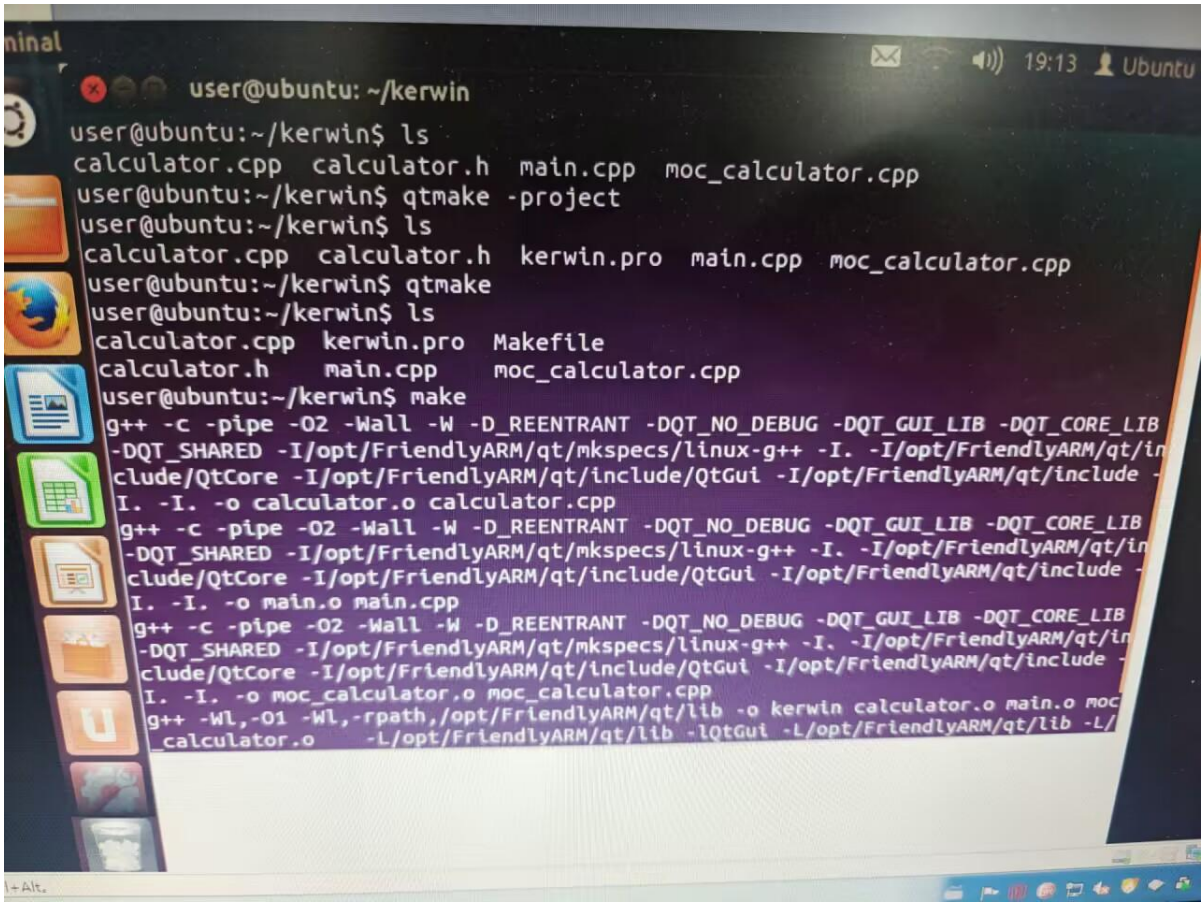
[root@FriendlyARM /]# **qt4** 可执行程序名

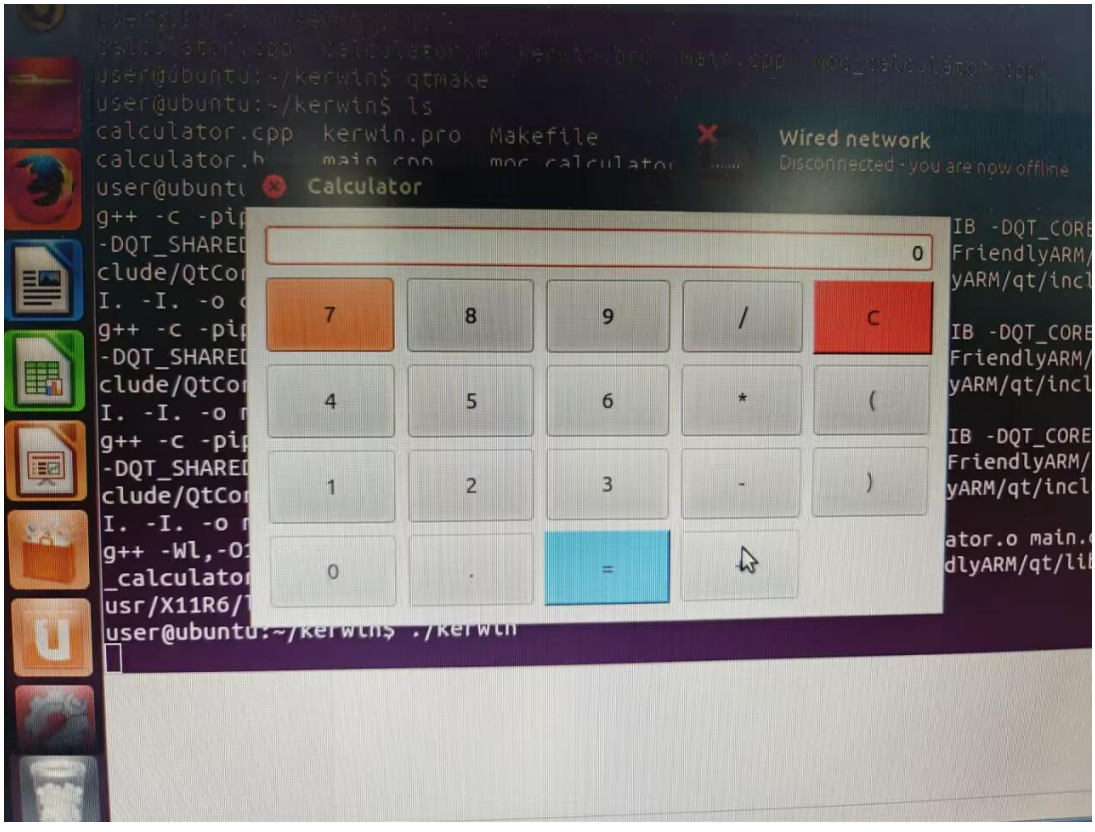
qt4 命令是一个 shell 脚本，可设置 Qt 的触摸屏等信息，然后引导程序，qt4 的路径为/bin/qt4。

四、实验内容

1. 弄清 Qt 的发展历史和版本信息。
2. 熟悉 Qt 信号与槽机制。
3. 按照给出的示例，新建一个目录，在其中编写 firstDalog.h、firstDalog.cpp、main.cpp 三个文件，即先设计一个对话框，结合信号与槽，在其中实现点击一个按钮计算 2 个数的和，点击另一个按钮计算 2 个数的差，如下图所示。

4. 分别采用 qtmake 和 qtemake 编译生成在虚拟机运行和实验箱运行的可执行程序。


user@ubuntu: ~/kerwin
user@ubuntu:~/kerwin$ ls
calculator.cpp calculator.h main.cpp moc_calculator.cpp
user@ubuntu:~/kerwin$ qtmake -project
user@ubuntu:~/kerwin$ ls
calculator.cpp calculator.h kerwin.pro main.cpp moc_calculator.cpp
user@ubuntu:~/kerwin$ qtmake
user@ubuntu:~/kerwin$ ls
calculator.cpp kerwin.pro Makefile
calculator.h main.cpp moc_calculator.cpp
user@ubuntu:~/kerwin$ make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB
-DQT_SHARED -I/opt/FriendlyARM/qt/mkspecs/linux-g++ -I. -I/opt/FriendlyARM/qt/in
clude/QtCore -I/opt/FriendlyARM/qt/include/QtGui -I/opt/FriendlyARM/qt/include -
I. -I. -o calculator.o calculator.cpp
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB
-DQT_SHARED -I/opt/FriendlyARM/qt/mkspecs/linux-g++ -I. -I/opt/FriendlyARM/qt/in
clude/QtCore -I/opt/FriendlyARM/qt/include/QtGui -I/opt/FriendlyARM/qt/include -
I. -I. -o main.o main.cpp
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB
-DQT_SHARED -I/opt/FriendlyARM/qt/mkspecs/linux-g++ -I. -I/opt/FriendlyARM/qt/in
clude/QtCore -I/opt/FriendlyARM/qt/include/QtGui -I/opt/FriendlyARM/qt/include -
I. -I. -o moc_calculator.o moc_calculator.cpp
g++ -Wl,-O1 -Wl,-rpath,/opt/FriendlyARM/qt/lib -o kerwin calculator.o main.o moc
_calculator.o -L/opt/FriendlyARM/qt/lib -lQtGui -L/opt/FriendlyARM/qt/lib -L/


Wired network
Disconnected - you are now offline
Calculator
0
7
8
9
/
C
4
5
6
*
(
1
2
3
-
)
0
.
=

五、实验结果及分析：

1．叙述 Qt 的信号与槽机制

Qt 的信号与槽机制是其核心特性之一，是一种用于对象间通信的强大机制，它使得不同组件之间的协作变得非常灵活和高效。在我本次实验编写的简单 Qt 程序中，按钮（QPushButton）的点击动作触发了一个信号（clicked()），这个信号被连接到了我们自定义的一个槽函数（onButtonClicked()），从而实现了点击按钮后标签（QLabel）文本的更新。

- **核心概念：** 当一个特定事件发生时（例如按钮被点击、窗口标题改变、数据接收完毕等），一个对象可以**发射 (emit)** 一个**信号 (signal)**。信号本身不包含任何处理逻辑，它只负责通知外界某个事件发生了。其他对象中定义的**槽 (slot)** 函数可以**连接 (connect)** 到这个信号。当信号被发射时，所有连接到它的槽函数会被自动依次执行。
- **解耦性：** 信号的发射者不需要知道是哪个对象的哪个槽会接收这个信号，同样，槽的接收者也不需要知道是哪个对象的哪个信号激活了它。这种机制极大地降低了对象之间的耦合度，使得模块化设计和代码复用更加容易。一个信号可以连接到多个槽，一个槽也可以响应多个信号，信号甚至可以直接连接到另一个信号。
- **类型安全：** 信号和槽的连接在编译时或运行时会进行参数类型的检查。只有参数类型匹配的信号和槽才能成功连接，这保证了通信的类型安全，避免了传统回调函数中可能出现的类型不匹配导致的运行时错误。
- **MOC (Meta-Object Compiler)：** 由于信号和槽机制是 Qt 对标准 C++的扩展，Qt 引入了一个元对象编译器（MOC）。MOC 在编译前处理源代码中包含 Q_OBJECT 宏的类定义，为这些类生成实现信号和槽机制所必需的附加 C++代码。因此，所有需要使用信号和槽的类都必须继承自 QObject (或其子类) 并在类声明中包含 Q_OBJECT 宏。
- **信号的声明与发射：** 信号在类的 signals: 区域声明，形式上类似函数声明但没有函数体，返回类型为 void。通过 emit 关键字发射信号。
- **槽的声明与实现：** 槽是普通的 C++成员函数，在类的 public slots:、protected slots: 或 private slots: 区域声明，并像普通成员函数一样在 .cpp 文件中实现。它们可以有任意参数，也可以是虚函数。
- **连接：** 使用 QObject::connect() 静态函数来建立信号和槽之间的连接，需要使用 SIGNAL() 和 SLOT() 宏来指定信号和槽的签名。例如，在本次实验中，我们使用了 connect(m_button, SIGNAL(clicked()), this, SLOT(onButtonClicked())); 将按钮的 clicked() 信号连接到 MyWidget 类的 onButtonClicked() 槽。

通过本次实验，我直观地体验到了信号与槽机制的便捷性。它使得处理用户界面事件（如按钮点击）的逻辑非常清晰，代码也更易于理解和维护。例如，当按钮 m_button 被点击时，它会自动发射 clicked() 信号，而程序无需关心这个信号如何传递，只需确保相应的槽函数 onButtonClicked() 被正确连接并实现预期的功能即可。这种机制是 Qt 强大功能和良好设计的重要体现。

2．将点击一个按钮实现计算 2 个数的差相关代码写在下面。

实现了一个完整的计算器，使用逆波兰表达式来完成计算。

calculator.h

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
```

```
#include <QDialog>
#include <QLineEdit>
#include <QPushButton>
#include <QGridLayout>

class Calculator : public QDialog {
    Q_OBJECT

public:
    Calculator(QWidget *parent = 0);

private slots:
    void digitClicked();
    void operatorClicked();
    void equalClicked();
    void clearClicked();

private:
    QLineEdit *display;
    QString expression;   // 存储完整表达式
    bool newNumber;       // 标记新数字输入

    QPushButton* createButton(const QString &text, const char *member);
    bool lastCharIsOperator();   // 检查最后一个字符是否是运算符
};

#endif
```

calculator.cpp

```
#include "calculator.h"
#include <QGridLayout>
#include <QDebug>

Calculator::Calculator(QWidget *parent) : QDialog(parent) {
    display = new QLineEdit("0");
    display->setReadOnly(true);
    display->setAlignment(Qt::AlignRight);
    display->setMaxLength(30);
    expression = "";
    newNumber = true;

    QGridLayout *mainLayout = new QGridLayout;
    mainLayout->setSizeConstraint(QLayout::SetFixedSize);
```

```
    mainLayout->addWidget(display, 0, 0, 1, 5);

    QString buttons[5][5] = {
        {"7", "8", "9", "/", "C"},
        {"4", "5", "6", "*", "("},
        {"1", "2", "3", "-", ")"},
        {"0", ".", "=", "+", ""},
        {" ", " ", " ", " ", " "}
    };

    for (int row = 0; row < 4; ++row) {
        for (int col = 0; col < 5; ++col) {
            if (buttons[row][col].isEmpty()) continue;

            QPushButton *button = createButton(buttons[row][col], SLOT(digitClicked()));

            if (buttons[row][col] == "+" || buttons[row][col] == "-" ||
                buttons[row][col] == "*" || buttons[row][col] == "/") {
                connect(button, SIGNAL(clicked()), this, SLOT(operatorClicked()));
            } else if (buttons[row][col] == "=") {
                connect(button, SIGNAL(clicked()), this, SLOT(equalClicked()));
            } else if (buttons[row][col] == "C") {
                connect(button, SIGNAL(clicked()), this, SLOT(clearClicked()));
            }

            mainLayout->addWidget(button, row + 1, col);
        }
    }

    setLayout(mainLayout);
    setWindowTitle("Calculator");
}

QPushButton* Calculator::createButton(const QString &text, const char *member) {
    QPushButton *button = new QPushButton(text);
    button->setMinimumSize(50, 50);
    if (text == "C") button->setStyleSheet("background-color: #ff6666;");
    if (text == "=") button->setStyleSheet("background-color: #66ccff;");
    if ((text >= "0" && text <= "9") || text == ".")
        connect(button, SIGNAL(clicked()), this, member);
    return button;
}

bool Calculator::lastCharIsOperator() {
```

```
    if (expression.isEmpty()) return false;
    QChar last = expression.at(expression.size()-1);
    return (last == '+' || last == '-' || last == '*' || last == '/');
}

void Calculator::digitClicked() {
    QPushButton *clickedButton = qobject_cast<QPushButton *>(sender());
    QString value = clickedButton->text();

    if (newNumber) {
        expression.clear();
        newNumber = false;
        if (value == ".") expression = "0";
    }

    if (value == ".") {
        if (expression.contains('.') && !lastCharIsOperator()) return;
    }

    expression += value;
    display->setText(expression);
}

void Calculator::operatorClicked() {
    QPushButton *clickedButton = qobject_cast<QPushButton *>(sender());
    QString op = clickedButton->text();

    if (expression.isEmpty()) {
        if (op == "-") expression = "0";   // 允许负数
        else return;
    }

    if (lastCharIsOperator()) {
        expression.chop(1);   // 替换前一个运算符
    }

    expression += op;
    newNumber = false;
    display->setText(expression);
}

void Calculator::equalClicked() {
    if (expression.isEmpty()) return;
```

```
    // 处理最后字符是运算符的情况
    if (lastCharIsOperator()) expression.chop(1);

    // 简单表达式解析（按输入顺序计算）
    QStringList numbers = expression.split(QRegExp("[+\\-*/]"), QString::SkipEmptyParts);
    QStringList ops = expression.split(QRegExp("[0-9.]+"), QString::SkipEmptyParts);

    if (numbers.size() < 1 || ops.size() != numbers.size()-1) {
        display->setText("Error");
        return;
    }

    double result = numbers[0].toDouble();
    for (int i = 1; i < numbers.size(); ++i) {
        QString op = ops[i-1];
        double num = numbers[i].toDouble();

        if (op == "+") result += num;
        else if (op == "-") result -= num;
        else if (op == "*") result *= num;
        else if (op == "/") {
            if (num == 0) {
                display->setText("Error");
                 return;
            }
            result /= num;
        }
    }

    QString resultStr = QString::number(result, 'g', 12);
    QString displayText = expression + "=" + resultStr;
    display->setText(displayText);
    expression = resultStr;
    newNumber = true;
}

void Calculator::clearClicked() {
    expression.clear();
    display->setText("0");
    newNumber = true;
}
```

Main.cpp

```
#include <QApplication>
```

```
#include "calculator.h"

int main(int argc, char *argv[]) {
	QApplication app(argc, argv);
	Calculator calculator;
	calculator.show();
	return app.exec();
}
```