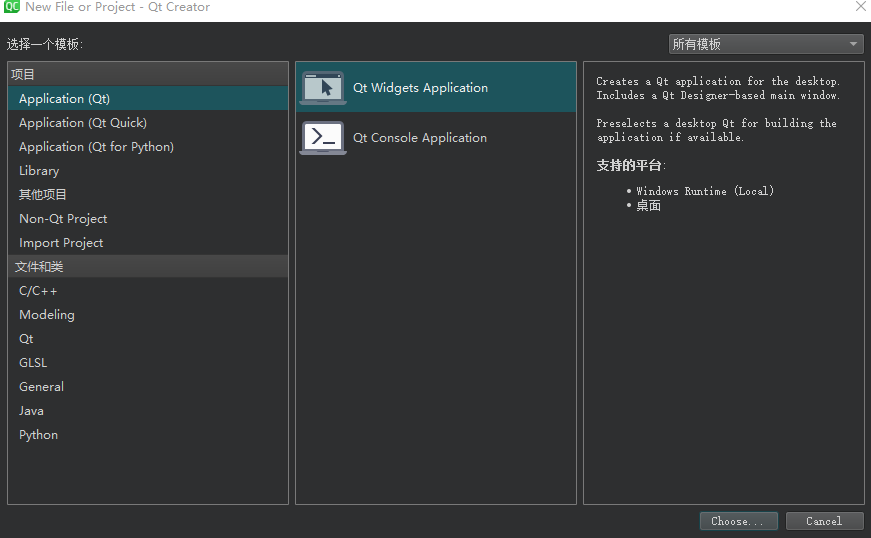
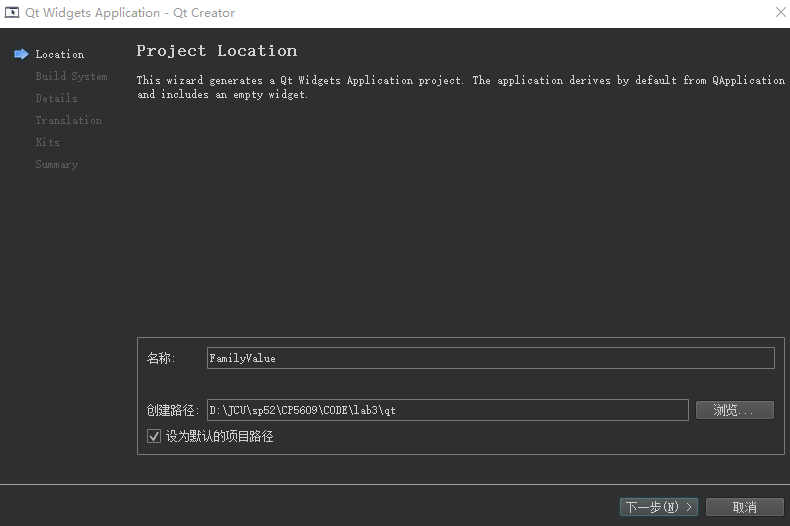
**QT tutorial**

First of all, I used qt creator to create the new project, for each task, I created a new project.





**Task 1 hello world**.

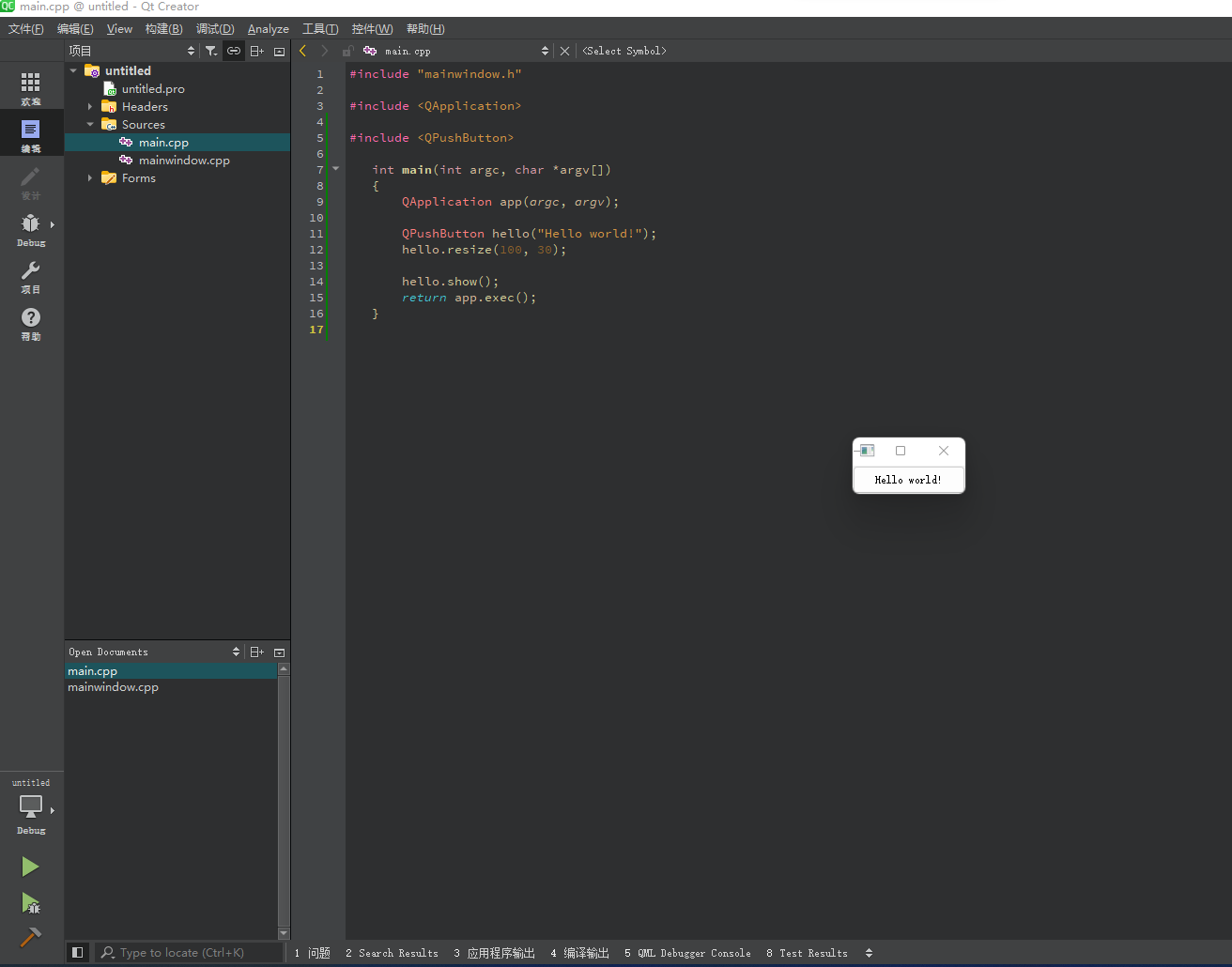
The following file has two class.

QApplication is in charge of managing application-wide resources like the default font and cursor.

QPushButton is a graphical user interface push button that the user may press and release.

I used the resize method to resize the window and the show function to demonstrate.

When the program ends, QCoreApplication::exec() returns.



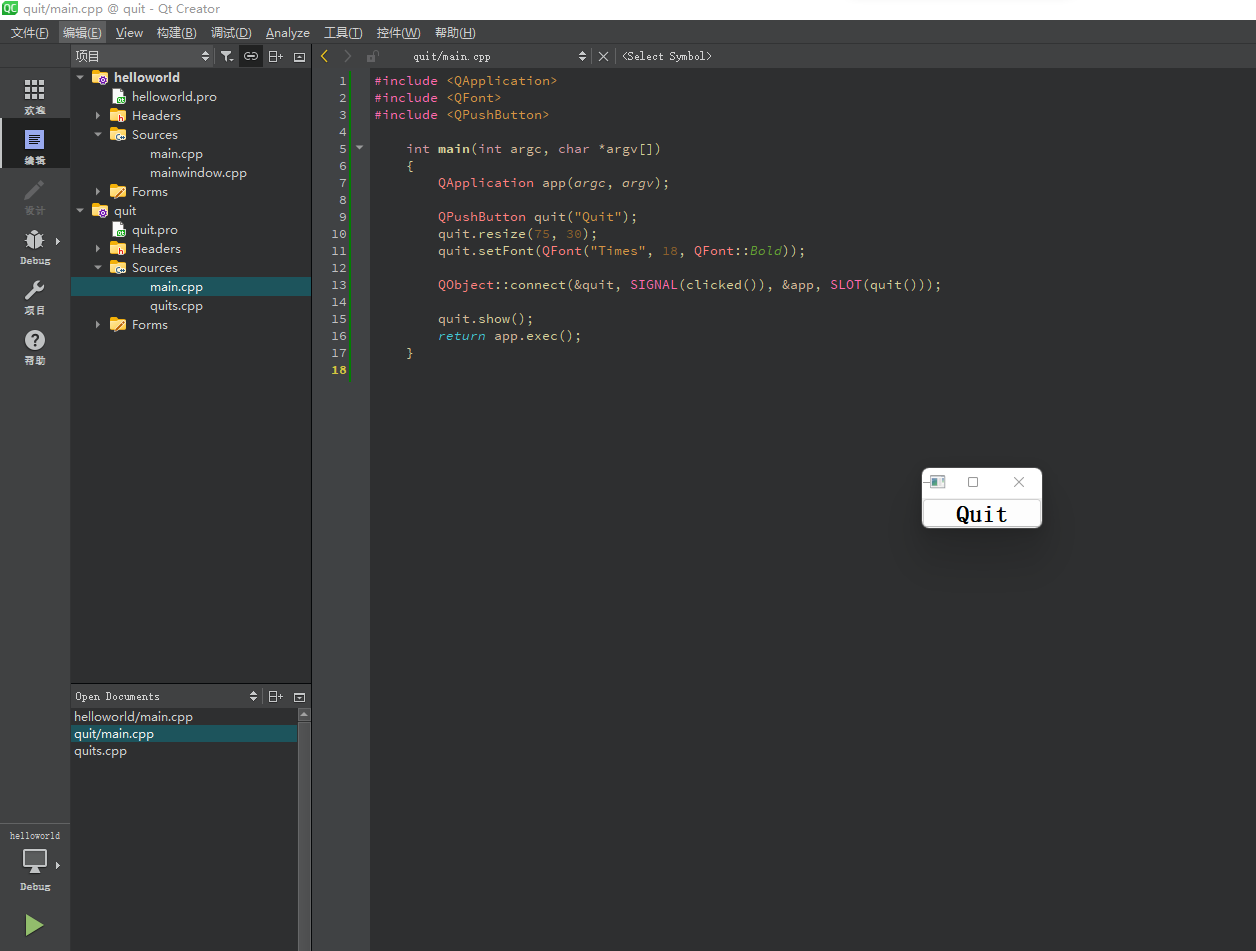
**Task 2 Quits**

Since the software relied on QFront, it must be included.

This time, the button reads Quit, and when the user hits it, the software follows.

The font size has been increased to 18 points bold from the Times family.

Furthermore, this connect() method creates a one-way link between two Qt objects (objects that directly or indirectly inherit QObject). Every Qt object can have both signals (for message transmission) and slots (to receive messages). All widgets are Qt objects since they derive from QWidget, which derives from QObject.



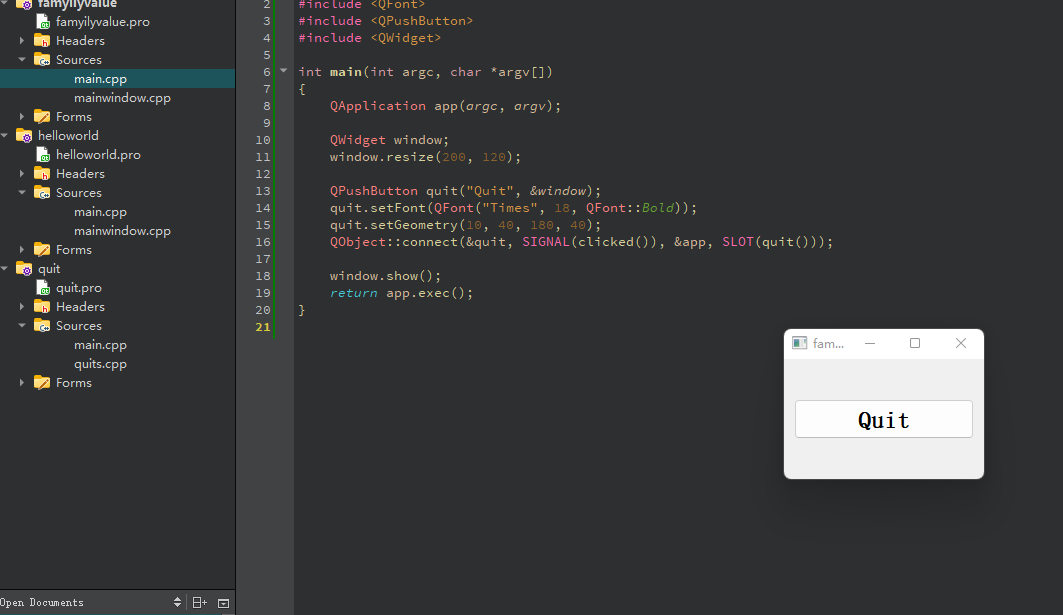
**Task 3 FamilyValue**

We include QWidget to acquire the underlying widget class we'll be using. We just make a simple widget object. The QWidget class serves as the foundation for all user interface objects. As a result, the Qwidget class can interact with other classes.

Window's child is quit in the project.

The QWidget::setGeometry() method produces a button that extends from (10, 40) to (10, 40). (190, 80).

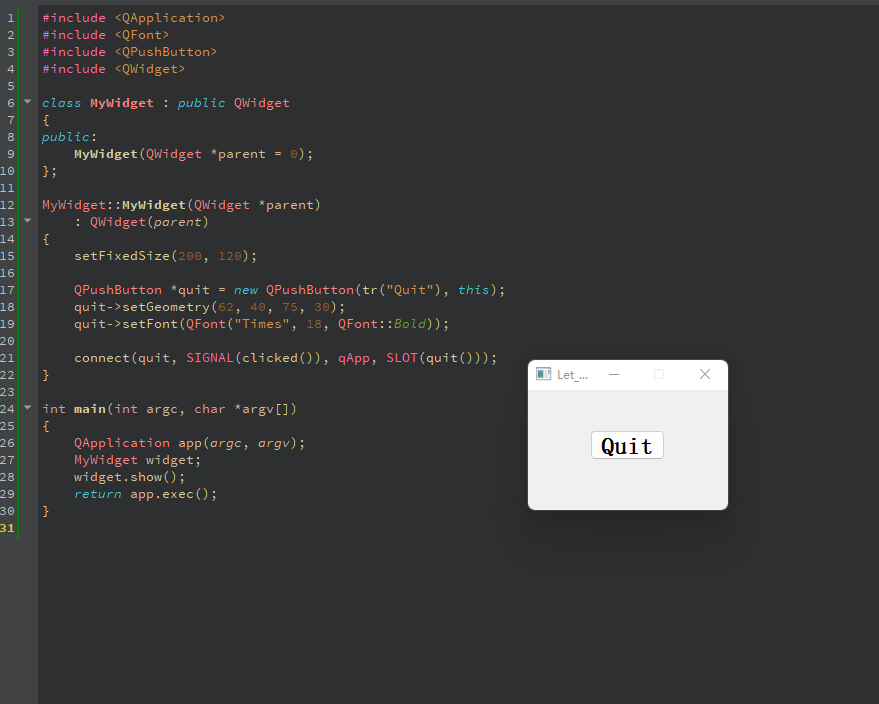
To alter the parent variable, we used window.show(). At the same time, it will schedule a performance for all of its youngsters.



**Task 4 Let There Be Widgets**

We're going to make a new class here (MyWidget). Because it derives from QWidget, the new class is a widget that may be either a top-level window or a child widget. There is only one constructor. Furthermore, this is a child widget (we used "this" for the mywidget class, and the tr() function call surrounding the string literal "Quit" marks the text for translation, allowing it to be changed at runtime based on the contents of a translation file.

In the main function, we construct and display the Mywidget.



**Task 5 Building blocks**

Similarly, we used the constructor. However, there are three more variables.

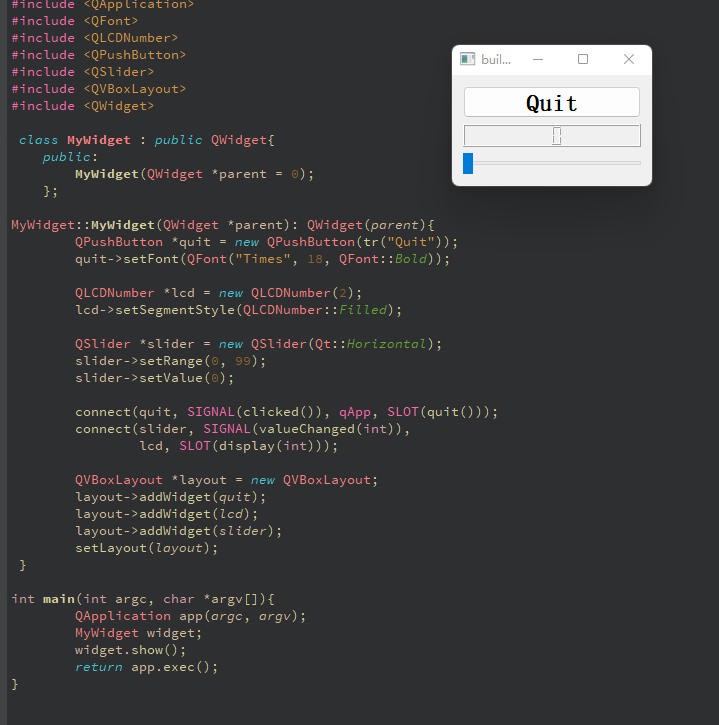
lcd is a [QLCDNumber](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qlcdnumber.html), a widget that displays numbers in an LCD-like fashion. This instance is set up to display two digits. We set the [QLCDNumber::segmentStyle](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qlcdnumber.html" \l "segmentStyle-prop) property to [QLCDNumber::Filled](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qlcdnumber.html" \l "SegmentStyle-enum) to make the LCDs more readable.

We can use the [QSlider](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qslider.html) widget to adjust an integer value in a range. Here we create a horizontal one, set its minimum value to 0, its maximum value to 99, and its initial value to 0.

Therefore, we used connect() to connect these two variables.

In addition,we used a [QVBoxLayout](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qvboxlayout.html) to manage the geometry of its child widgets. In addition, using a layout ensures that the child widgets are resized when the window is resized.

The [QWidget::setLayout](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qwidget.html" \l "setLayout)() function installs the layout on MyWidget. This makes the layout a child widget of MyWidget so we don't have to worry about deleting it; it will be deleted together with MyWidget.



**Task 6 Build block galore**

LCDRange::LCDRange(QWidget \*parent)

: QWidget(parent)

{

QLCDNumber \*lcd = new QLCDNumber(2);

lcd->setSegmentStyle(QLCDNumber::Filled);

QSlider \*slider = new QSlider(Qt::Horizontal);

slider->setRange(0, 99);

slider->setValue(0);

connect(slider, SIGNAL(valueChanged(int)),

lcd, SLOT(display(int)));

QVBoxLayout \*layout = new QVBoxLayout;

layout->addWidget(lcd);

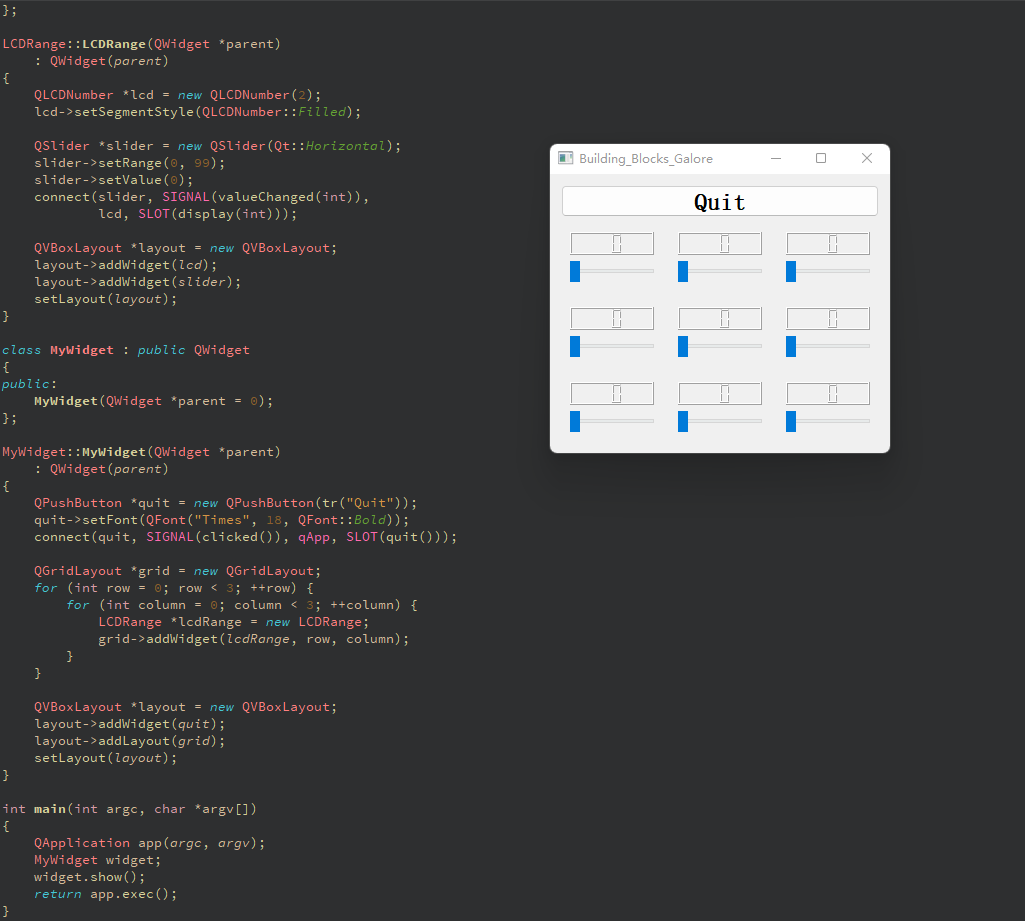
layout->addWidget(slider);

setLayout(layout);

}

This is comparable to the previous project's constructor.

Furthermore, for the Mywidget class, the push button that was formerly part of what is now LCDRange has been separated, allowing us to have one Quit button and several LCDRange objects. To establish the grid, we utilized a for loop.



**Task 7 One thing leads to another**

The **lcdrange class** is in a separate file for this assignment.

This file is mostly pulled from main.cpp task 6 in the lcdrange.cpp file.

connect(slider, SIGNAL(valueChanged(int)),

lcd, SLOT(display(int)));

connect(slider, SIGNAL(valueChanged(int)),

this, SIGNAL(valueChanged(int)));

The initial connect() call is identical to job 6. The second is new, it connects the valueChanged() signal of the slider to the valueChanged() signal of this object. When the user moves the slider, the slider detects the change and sends the valueChanged() signal. That signal is linked to the show() slot of the QLCDNumber as well as the valueChanged() signal of the LCDRange. As a result, when the signal is sent, LCDRange sends its own valueChanged() signal. Furthermore, QLCDNumber::display() is called, which displays the new number.

In the main.cpp file.

LCDRange \*previousRange = 0;

for (int row = 0; row < 3; ++row) {

for (int column = 0; column < 3; ++column) {

LCDRange \*lcdRange = new LCDRange;

grid->addWidget(lcdRange, row, column);

if (previousRange)

connect(lcdRange, SIGNAL(valueChanged(int)),

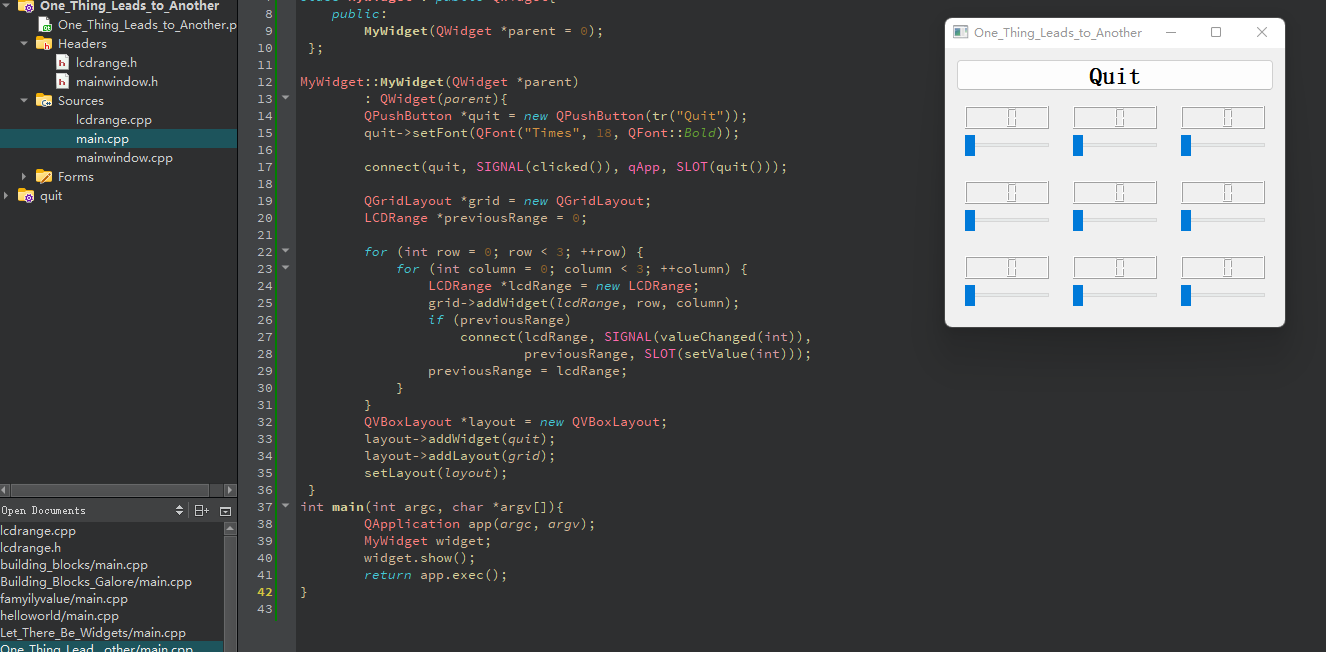
previousRange, SLOT(setValue(int)));

previousRange = lcdRange;

}

}

Except for the constructor for MyWidget, all of main.cpp is duplicated from the previous job. When we build the nine LCDRange objects, we use the signals and slots technique to link them. Each has a valueChanged() signal that is linked to the preceding one's setValue() slot. Because LCDRange generates the valueChanged() signal when its value changes, we're building a signal and slot chain here.



**Task 8 Preparing for Battle**

There are three cpp files.

**Lcdrange** is comparable to the preceding task. Furthermore, it has been enhanced with a new feature that allows the range to be modified using setRange() based on min - max values. The range is required since we configured QLCDNumber to show two digital. If the range requirement is not met, the qWarning() method in Qt can be utilized.

**Cannonfield** is a new custom widget that can show itself and is based on QWidget. We used the same phrase as lcdrange. However, only one public characteristic may be modified, and that is the angle. PaintEvent is one of the Owidget class's events. When modifications occur, Qt manages them. The constructor in the source file sets the angle value to 45 degrees and creates a custom palette for this widget. The RGB value affects Qcolor. The angle attribute is set by three slots below 5, 5-70, and above 70 via the for loop, as shown below:

void CannonField::setAngle(int angle)

{

if (angle < 5)

angle = 5;

if (angle > 70)

angle = 70;

if (currentAngle == angle)

return;

currentAngle = angle;

update();

emit angleChanged(currentAngle);

}

To notify the outside world that the angle has changed, we send the angleChanged() signal.

void CannonField::paintEvent(QPaintEvent \* /\* event \*/)

{

QPainter painter(this);

painter.drawText(200, 200,

tr("Angle = ") + QString::number(currentAngle));

}

The code for the paintEvent shows the angle value in the widget at a fixed point. We start by making a QString with some text and the angle; then we make a QPainter that operates on this widget and uses it to paint the string.

Similarly, mywidget is a class in the main cpp file that controls the two classes mentioned before. We defined LcdRange and set the angle in the constructor. We also developed cannonfield and linked it to lcdrange. As demonstrated by the following code:

  connect(angle, SIGNAL(valueChanged(int)),

cannonField, SLOT(setAngle(int)));

connect(cannonField, SIGNAL(angleChanged(int)),

angle, SLOT(setValue(int)));

We connect the LCDRange's valueChanged() signal to the CannonField's setAngle() slot here. When the user moves the LCDRange, the angle value of CannonField is updated. We also establish the opposite link, such that altering the angle in the CannonField updates the LCDRange value. This exemplifies the need of component programming and correct encapsulation.

We used [QGridLayout](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qgridlayout.html) instead of [QVBoxLayout](http://neo.dmcs.p.lodz.pl/po/qt-tutorial/docs/qt/qvboxlayout.html) for this situation, since it control more elements. Therefore, we add the Quit button by

gridLayout->addWidget(quit, 0, 0);

And added angle LCDRange cell (1, 0) by

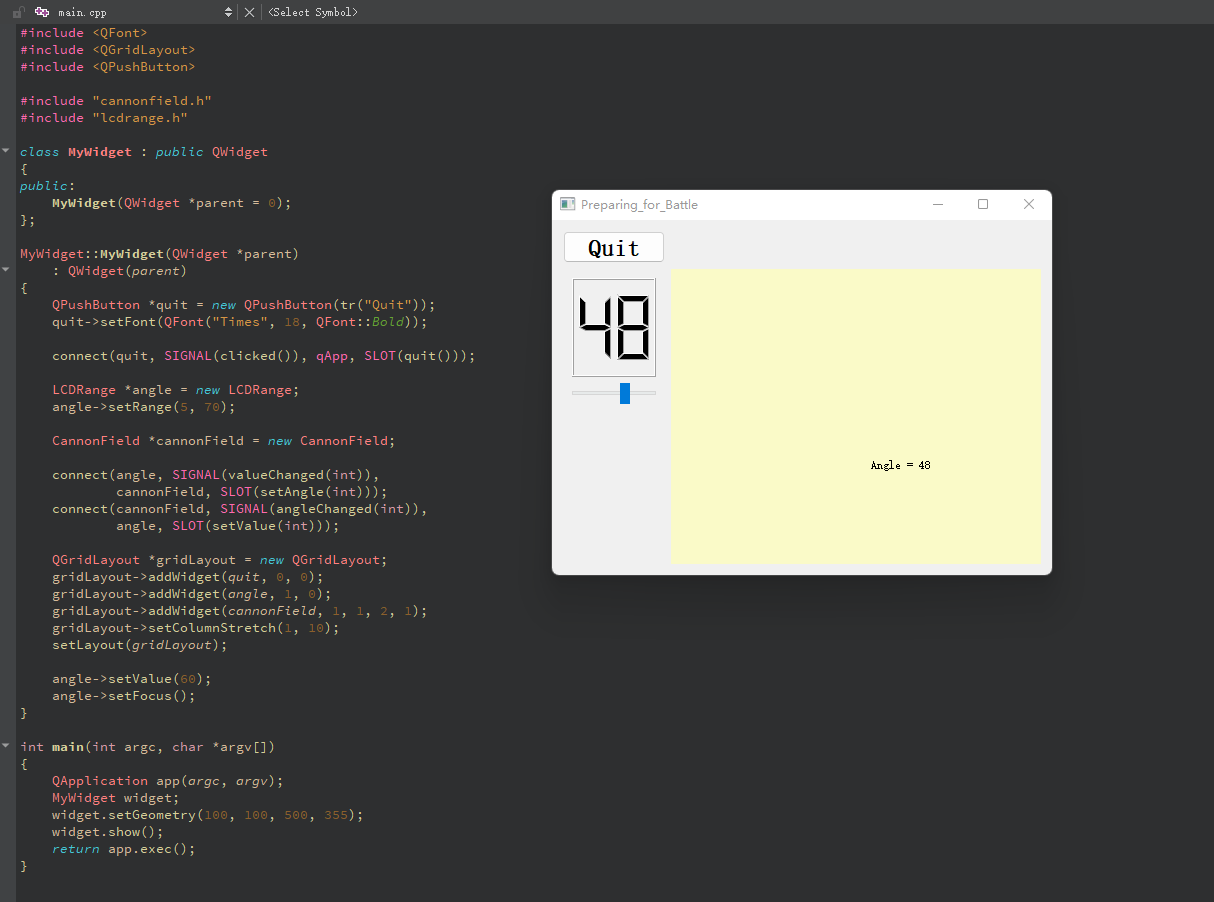
addWidget(angle, 1, 0);

And CannonField object,

addWidget(cannonField, 1, 1, 2, 1);

And we change QGridLayout that the right column (column 1) has a stretch factor of 10.

Finally, we set angle to have keyboard focus and initialized the angle value.



**Task 9 With Cannon You Can**

The code is based on previous tasks. The previous task can change the angle and the corresponding result is showed on the screen. However, in this task, there is cannon on the left bottom side, which can changed its angle dynamically.

For the **cannonfield.cpp**. The new added elements is QPainter, which is the key part of changing the cannon direction.

First and foremost, we designed a painter that works with this widget.

QPainter painter(this);

The pen is used to draw the margins of what

QPainter creates.painter.setPen(Qt::NoPen);

We used a brush to color the items.

painter.setBrush(Qt::blue);

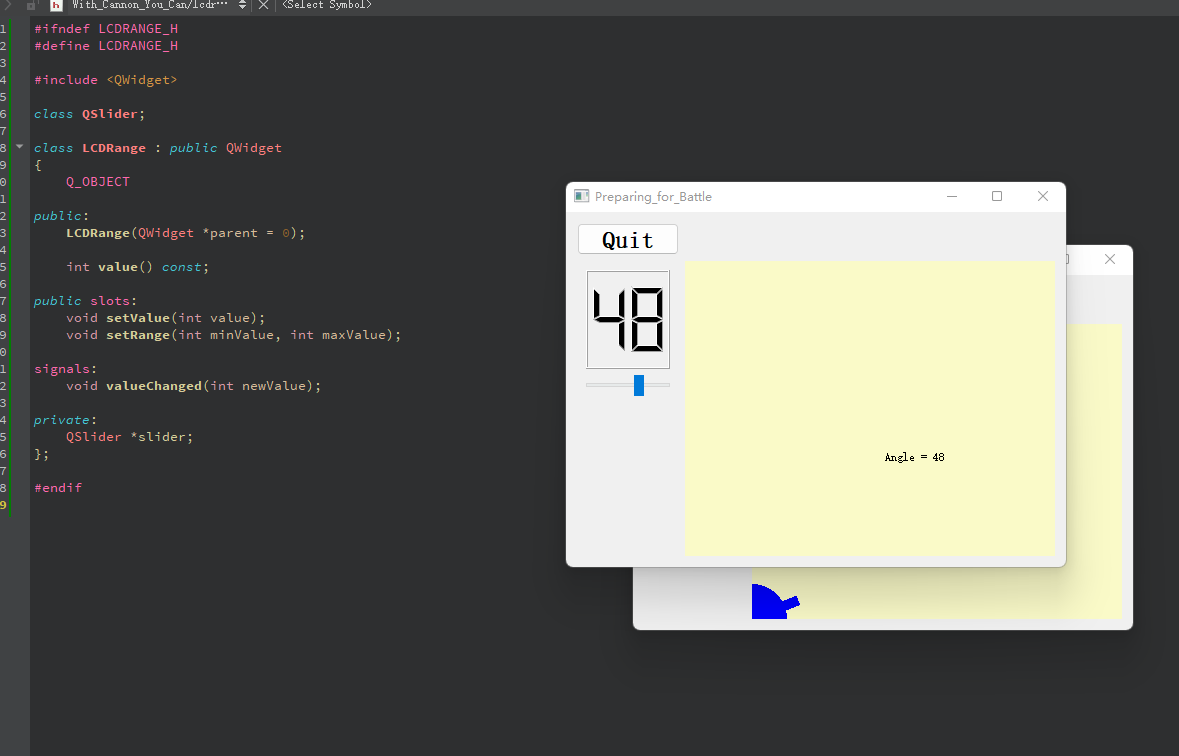
Furthermore, we utilized the translate() method to reposition the coordinate system.

painter.translate(0, rect().height());

Using a start angle and an arc length, we create a pie form inside the supplied rectangle.

painter.drawPie(QRect(-35, -35, 70, 70), 0, 90 \* 16);

Furthermore, the rotate and drawrect methods are relevant to cannon movement.



**Task 10 Smooth as slik**

Similar to challenge 9, however instead of changing the direction of the cannon.

In addition to the angle, the **CannonField** now provides a force value.

  int angle() const { return currentAngle; }

int force() const { return currentForce; }

public slots:

void setAngle(int angle);

void setForce(int force);

signals:

void angleChanged(int newAngle);

void forceChanged(int newForce)

The force is saved in the variable currentForce.

The constructor in the.cpp file sets the force to zero and the angle to 45.

currentForce = 0;

We made a minor adjustment to the setAngle() method. It just redraws the section of the widget containing the cannon.

if (currentAngle == angle)

return;

setForce() is implemented in a manner similar to setAngle() (). The only difference is that we don't need to repaint the widget because we don't show the force value.

  void CannonField::setForce(int force)

{

if (force < 0)

force = 0;

if (currentForce == force)

return;

currentForce = force;

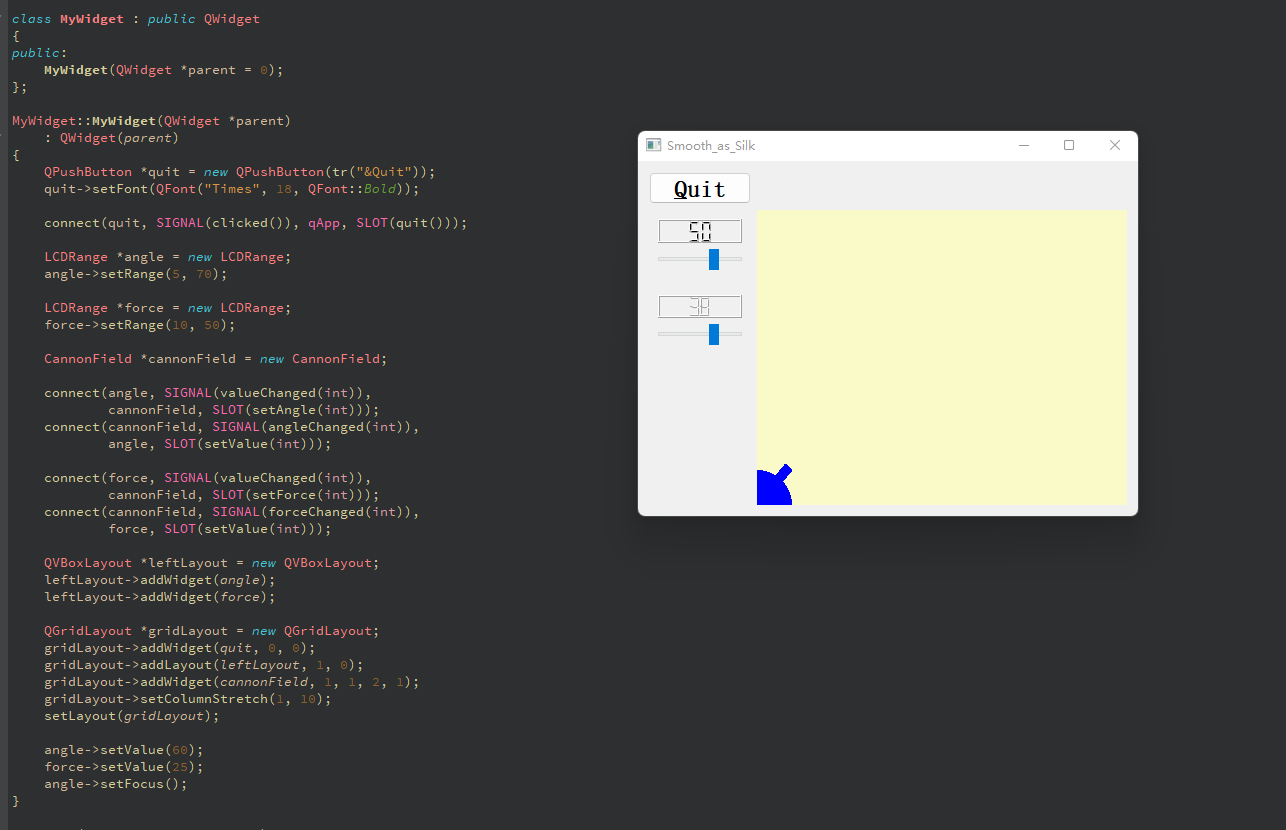
emit forceChanged(currentForce);

}

In **main.cpp**, we want two widgets in that cell, so we create a vertical box, place it in the grid cell, and place each of angle and range in the vertical box, so it can be controlled by the QVBoxLayout and QGridLayout classes.

QVBoxLayout \*leftLayout = new QVBoxLayout;

QGridLayout \*gridLayout = new QGridLayout;



**Task 11. Give it a shot**

**CannonField** now has the ability to shoot.

void shoot();

Invoking this slot forces the cannon to fire while the shell is not in the air.

private slots:

void moveShot();

This function is going to draw the shot.

private:

void paintShot( QPainter \* );

QRect shotRect() const;

This private method returns the rectangle that encapsulates the space occupied by the projectile when it is in the air; otherwise, it returns an undefined rectangle.

int timerCount;

QTimer \* autoShootTimer;

float shoot\_ang;

float shoot\_f;

};

These private variables hold information about the shot. TimerCount records the amount of time that has passed since the firing. Shoot\_ang is the angle at which the gun is shot, and shoot f denotes the power with which the cannon is fired.

We used sin() and cos() in the cannon.cpp code because we needed to draw the bullet's moving route. This private function computes the shell's center point and returns the rectangle that encloses it. It employs the original cannon's strength and angle, as well as the timerCount, to automatically increase the elapsed time.

In a coordinate system with increasing y, we compute the center point. After calculating the center point, we create a QRect of size 6 by 6 and relocate its center to the computed center point. We repeat the process to shift the point to the coordinate system of the widget.

The qRound() function is an embedded function specified in the qglobal.h header file (and included by all other Qt headers). QRound () is a function that transforms a double-precision real value to its closest integer.

QRect CannonField::shotRect() const

{

const double gravity = 4;

double time = timerCount / 20.0;

double velocity = shootForce;

double radians = shootAngle \* 3.14159265 / 180;

double velx = velocity \* cos(radians);

double vely = velocity \* sin(radians);

double x0 = (barrelRect.right() + 5) \* cos(radians);

double y0 = (barrelRect.right() + 5) \* sin(radians);

double x = x0 + velx \* time;

double y = y0 + vely \* time - 0.5 \* gravity \* time \* time;

QRect result(0, 0, 6, 6);

result.moveCenter(QPoint(qRound(x), height() - 1 - qRound(y)));

return result;

}

MoveShot () is a moving shot slot that is called by QTimer.

Its job is to calculate the new position, redraw the screen and put the shell in the new position, and stop the timer if necessary. Furthermore, for the most of online code, the QRegion class uses.unite () to return a rectangle. However, in QT version 5, the method name changed from .unite() to .united(). We spent some time to read the QT 5 document to understand it.

void CannonField::moveShot()

{

QRegion region = shotRect();

++timerCount;

QRect shotR = shotRect();

if (shotR.x() > width() || shotR.y() > height()) {

autoShootTimer->stop();

} else {

region = region.united(shotR);

}

update(region);

}

We set up our new private variable and attach the QTimer::timeout() signal to our moveShot() slot. When the timer runs out, we'll relocate the shot. This function fires a shot as long as the shell is not in the air. The TimerCount variable is set to zero. Shoot ang and shoot f are set to the current cannon's angle and power. Finally, the timer is set.

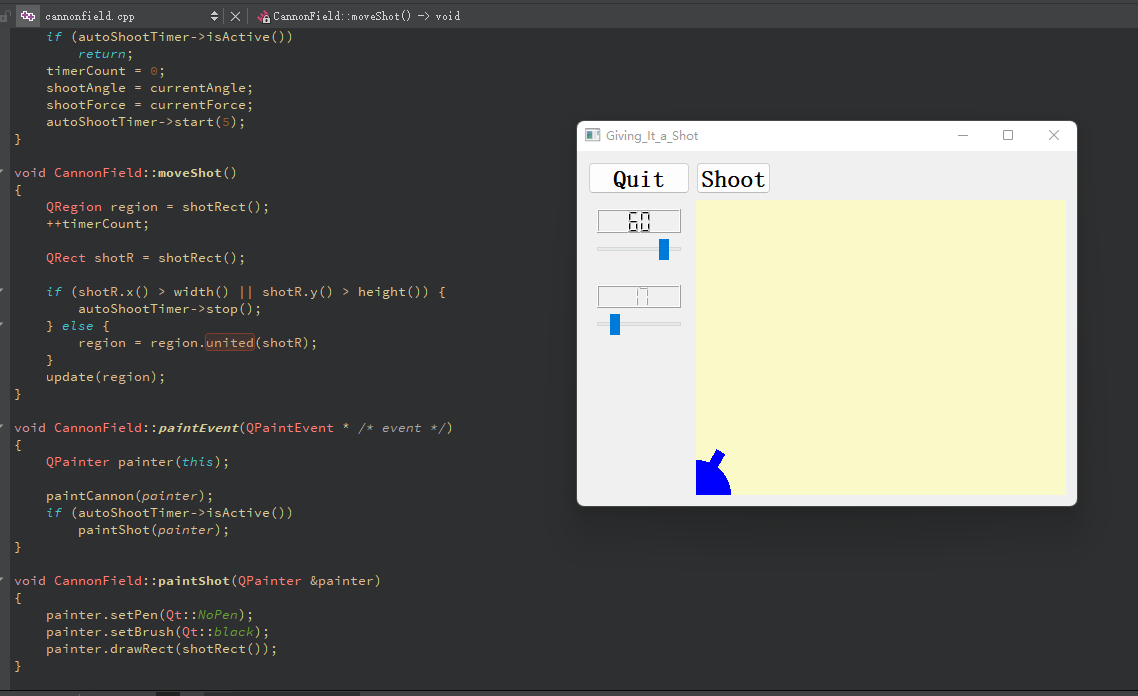
In the **main. cpp**, we added a shoot button, which is done similar with the quit button.

QPushButton \*shoot = new QPushButton(tr("&Shoot"));

shoot->setFont(QFont("Times", 18, QFont::Bold));

we connect the shoot button and shoot function in the cannonfield class like following:

[connect](https://documentation.help/Qt-3.0.5/qobject.html" \l "connect)( shoot, SIGNAL([clicked](https://documentation.help/Qt-3.0.5/qbutton.html" \l "clicked)()), cannonField, SLOT(shoot()) );



**Task 12 Hanging in the Air the Way Bricks Don't**

In the **lcdrange class**, there are two new text label.

  class QLabel;

class QSlider;

A new attribute is added in the constructor to set the label text.

We also have a new variable QLabel, which is a standard Qt widget that may display text or a QPixmap with or without a frame.

The constructor in the lcdrange.cpp code calls the init() method to set the label. We build a QLabel and configure the coordination.

There is also text(), which returns the label text, and setText(), which sets the label text. When a shot reaches the target, the hit() signal is sent. When the shot misses the target, the missed() signal is sent.

void hit();

void missed();

Furthermore, there is a new variable. Qpoint, which is the center of the target.

QPoint target;

In the cannonfield.h, there are two new signals, which are hit() and missed(). In addition, it contains a target.

void newTarget();

We used private function to paints the target.

  void paintTarget(QPainter &painter);

And used targetRect() to return rectangle of the target.

QRect targetRect() const;

<Stdlib.h> is included in the **cannonfield class** because it provides the rand() method, which is utilized in the newTarget() function. Because we need to establish a target at random, we used the newTarget() method. QTime is used to represent the time.

void CannonField::newTarget()

{

static bool firstTime = true;

if (firstTime) {

firstTime = false;

QTime midnight(0, 0, 0);

srand(midnight.secsTo(QTime::currentTime()));

}

target = QPoint(200 + rand() % 190, 10 + rand() % 255);

update();

}

The if condition is used for checking whether the bullet hit the target or not by detection the rectangular area.

if (shotR.intersects(targetRect())) {

autoShootTimer->stop();

emit hit();

} else if (shotR.x() > width() || shotR.y() > height()) {

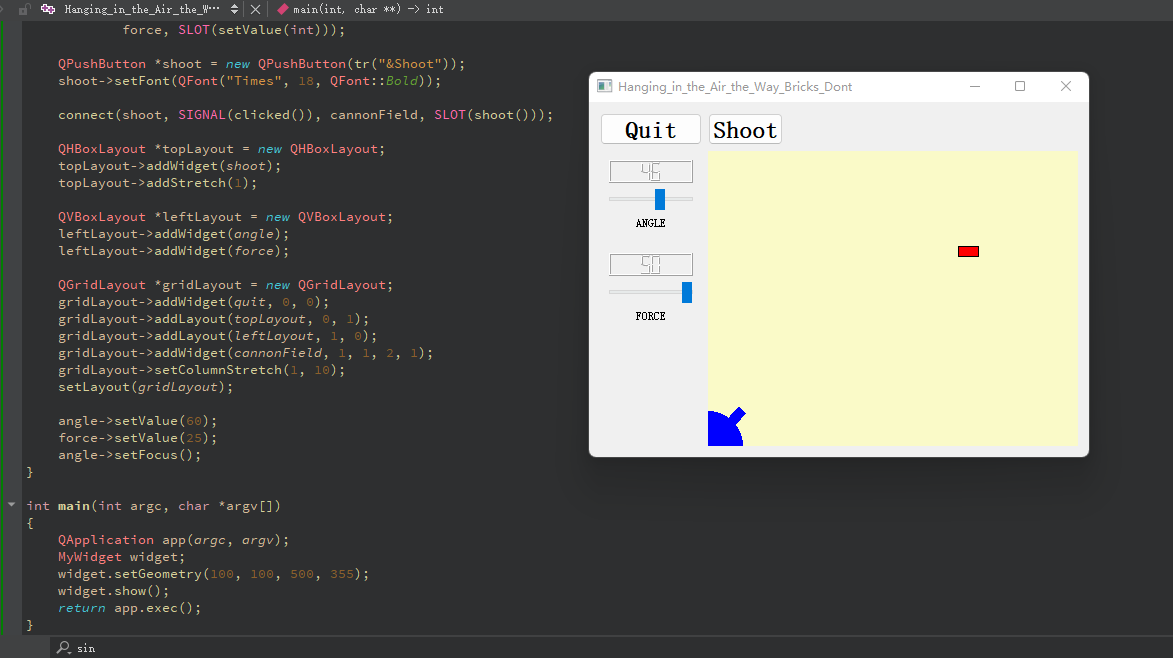
autoShootTimer->stop();

emit missed();

In the main.cpp, these two commends are used to set the label text.

LCDRange \*angle = new LCDRange(tr("ANGLE"));

LCDRange \*force = new LCDRange(tr("FORCE"));



**Task 13 Game Over**

This responsibility is now primarily concerned with the game's length. For example, how can players complete the game?

The layout command was contained in the **GameBoard class**.

We added four slots to the GameBoard class: fire(), hit(), missed(), newGame(), and two OLCDnumbers hits and shotsleft for displaying game status.

The constructor in the GameBoard.cpp code initially connects hit and missed and shoot button and canshoot signal to itself.

  connect(cannonField, SIGNAL(hit()),

this, SLOT(hit()));

connect(cannonField, SIGNAL(missed()),

this, SLOT(missed()));

  connect(shoot, SIGNAL(clicked()),

this, SLOT(fire()));

  connect(cannonField, SIGNAL(canShoot(bool)),

shoot, SLOT(setEnabled(bool)));

After that, like the screenshot shows, there is a new restart game button, which follows the following commend.

  QPushButton \*restart = new QPushButton(tr("&New Game"));

restart->setFont(QFont("Times", 18, QFont::Bold));

connect(restart, SIGNAL(clicked()), this, SLOT(newGame()));

In addition, the rest of code is mostly same like other task for initializing the layout and set the fire(), hit(), missed() and newGame() slots.

Furthermore, in the **cannonfield class**, since the new game status are introduced, some of changes are made.

Here are two new slots: setGameOver() and restartGame() and also a new signal canShoot() and also a bool variable to control if the game is over, which is initialized to false in the constructor.

  void setGameOver();

void restartGame();

void canShoot(bool can);

  bool gameEnded;

In the cpp file, there are several new functions. Setgameover() can end the game, which is manipulated outside of the class.

void CannonField::setGameOver()

{

if (gameEnded)

return;

if (isShooting())

autoShootTimer->stop();

gameEnded = true;

update();

}

Shoot() function can fire a bullet.

void CannonField::shoot()

{

if (isShooting())

return;

timerCount = 0;

shootAngle = currentAngle;

shootForce = currentForce;

autoShootTimer->start(5);

emit canShoot(false);

}

And also restartGame function.

  void CannonField::restartGame()

{

if (isShooting())

autoShootTimer->stop();

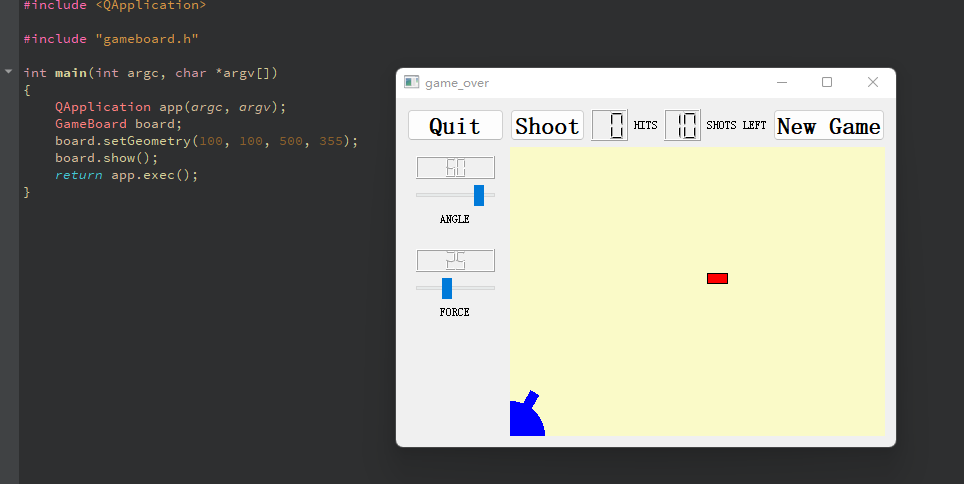
gameEnded = false;

update();

emit canShoot(true);

}

Most of the changes in this class are from paintEvent function. The paint event needs to display “game over”.



**Task 14 Facing the Wall**

Now, we install a wall in the middle.

**In the cannonfield class**, In addition to use mouse click or drag the button to change the angle. The mouse can click the cannon itself and change, since we implement these four methods to listen mouse event.

void paintEvent(QPaintEvent \*event);

void mousePressEvent(QMouseEvent \*event);

void mouseMoveEvent(QMouseEvent \*event);

To paint the barrier, we used these two method.

  void paintBarrier(QPainter &painter);

  QRect barrierRect() const;

In order to track if the bullet hit the barrier, the following method is introduced.

  bool barrelHit(const QPoint &pos) const;

An additional variable is used for monitoring if the bullet pass the barrier.

bool barrelPressed;

In the cpp file, the way of implementing the above methods are mostly done by calculating the coordination.

There are two methods to track the mouse event. MousemoveEvent and mouseReleaseEvent. These two event is called when user move the mouse and release the mouse.

In the paint event, since the wall is needed to set, the following commend is introduced.

  paintBarrier(painter);

Furthermore, in order to track if the bullet hit the barrier, QMarix is used, which defines a coordinate system mapping.

bool CannonField::barrelHit(const QPoint &pos) const

{

QMatrix matrix;

matrix.translate(0, height());

matrix.rotate(-currentAngle);

matrix = matrix.inverted();

return barrelRect.contains(matrix.map(pos));

}

Moreover, in the gameboard class, we used QFrame to set the style

  QFrame \*cannonBox = new QFrame;

cannonBox->setFrameStyle(QFrame::WinPanel | QFrame::Sunken);

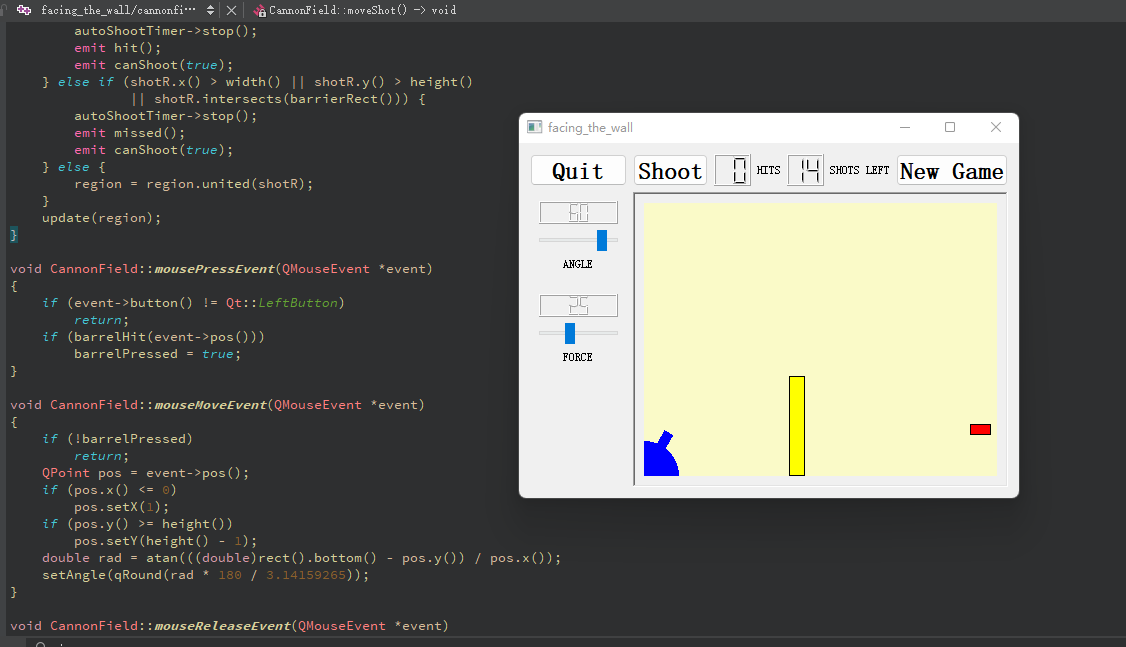
We construct and configure three QShortcut objects. If particular keys are pressed, these objects intercept keyboard events to a widget and call slots.

  (void) new QShortcut(Qt::Key\_Enter, this, SLOT(fire()));

(void) new QShortcut(Qt::Key\_Return, this, SLOT(fire()));

(void) new QShortcut(Qt::CTRL + Qt::Key\_Q, this, SLOT(close()));

The rest of commends in the class are similarly to the previous one.



In summary, the code has been uploaded in the github.