Group 3 Project

Generated by Doxygen 1.7.6.1

Thu Apr 13 2017 22:23:58

Contents

Chapter 1

EECE 435L Games Project

Author

Rita Aoun
Rawan Moukalled

Date

13-04-2017

Runs the application.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 QCP Namespace Reference

Enumerations

- enum MarginSide { msLeft = 0x01, msRight = 0x02, msTop = 0x04, msBottom = 0x08, msAll = 0xFF, msNone = 0x00 }
- enum AntialiasedElement { aeAxes = 0x0001, aeGrid = 0x0002, aeSubGrid = 0x0004, aeLegend = 0x0008, aeLegendItems = 0x0010, aePlottables = 0x0020, aeItems = 0x0040, aeScatters = 0x0080, aeErrorBars = 0x0100, aeFills = 0x0200, aeZeroLine = 0x0400, aeAll = 0xFFFF, aeNone = 0x0000 }
- enum PlottingHint { phNone = 0x000, phFastPolylines = 0x001, phForceRepaint = 0x002, phCacheLabels = 0x004 }
- enum Interaction { iRangeDrag = 0x001, iRangeZoom = 0x002, iMultiSelect = 0x004, iSelectPlottables = 0x008, iSelectAxes = 0x010, iSelectLegend = 0x020, iSelectItems = 0x040, iSelectOther = 0x080 }

Functions

- bool **isInvalidData** (double value)
- bool **isInvalidData** (double value1, double value2)
- void **setMarginValue** (QMargins &margins, QCP::MarginSide side, int value)
- int **getMarginValue** (const QMargins &margins, QCP::MarginSide side)

6.1.1 Detailed Description

The QCP Namespace contains general enums and QFlags used throughout the Q-CustomPlot library

6.1.2 Enumeration Type Documentation

6.1.2.1 enum QCP::AntialiasedElement

Defines what objects of a plot can be forcibly drawn antialiased/not antialiased. If an object is neither forcibly drawn antialiased nor forcibly drawn not antialiased, it is up to the respective element how it is drawn. Typically it provides a *setAntialiased* function for this.

`AntialiasedElements` is a flag of or-combined elements of this enum type.

See also

QCustomPlot::setAntialiasedElements, QCustomPlot::setNotAntialiasedElements

Enumerator:

aeAxes `0x0001` Axis base line and tick marks

aeGrid `0x0002` Grid lines

aeSubGrid `0x0004` Sub grid lines

aeLegend `0x0008` Legend box

aeLegendItems `0x0010` Legend items

aePlottables `0x0020` Main lines of plottables (excluding error bars, see element aeErrorBars)

aeItems `0x0040` Main lines of items

aeScatters `0x0080` Scatter symbols of plottables (excluding scatter symbols of type ssPixmap)

aeErrorBars `0x0100` Error bars

aeFills `0x0200` Borders of fills (e.g. under or between graphs)

aeZeroLine `0x0400` Zero-lines, see QCPGrid::setZeroLinePen

aeAll `0xFFFF` All elements

aeNone `0x0000` No elements

6.1.2.2 enum QCP::Interaction

Defines the mouse interactions possible with QCustomPlot.

`Interactions` is a flag of or-combined elements of this enum type.

See also

QCustomPlot::setInteractions

Enumerator:

iRangeDrag `0x001` Axis ranges are draggable (see QCPAxisRect::setRangeDrag, QCPAxisRect::setRangeDragAxes)

iRangeZoom `0x002` Axis ranges are zoomable with the mouse wheel (see QCPAxisRect::setRangeZoom, QCPAxisRect::setRangeZoomAxes)

iMultiSelect `0x004` The user can select multiple objects by holding the modifier set by QCustomPlot::setMultiSelectModifier while clicking

iSelectPlottables `0x008` Plottables are selectable (e.g. graphs, curves, bars,... see QCPAbstractPlottable)

iSelectAxes `0x010` Axes are selectable (or parts of them, see QCPAxis::setSelectableParts)

iSelectLegend `0x020` Legends are selectable (or their child items, see QCPLegend::setSelectableParts)

iSelectItems `0x040` Items are selectable (Rectangles, Arrows, Textitems, etc. see QCPAbstractItem)

iSelectOther `0x080` All other objects are selectable (e.g. your own derived layerables, the plot title,...)

6.1.2.3 enum QCP::MarginSide

Defines the sides of a rectangular entity to which margins can be applied.

See also

QCPLayoutElement::setAutoMargins, QCPAxisRect::setAutoMargins

Enumerator:

msLeft `0x01` left margin

msRight `0x02` right margin

msTop `0x04` top margin

msBottom `0x08` bottom margin

msAll `0xFF` all margins

msNone `0x00` no margin

6.1.2.4 enum QCP::PlottingHint

Defines plotting hints that control various aspects of the quality and speed of plotting.

See also

QCustomPlot::setPlottingHints

Enumerator:

phNone `0x000` No hints are set

phFastPolylines `0x001` Graph/Curve lines are drawn with a faster method. This reduces the quality especially of the line segment joins. (Only relevant for solid line pens.)

phForceRepaint `0x002` causes an immediate repaint() instead of a soft update() when QCustomPlot::replot() is called with parameter QCustomPlot::rpHint. - This is set by default to prevent the plot from freezing on fast consecutive replots (e.g. user drags ranges with mouse).

phCacheLabels `0x004` axis (tick) labels will be cached as pixmaps, increasing replot performance.

Chapter 7

Class Documentation

7.1 Barn Class Reference

Public Slots

- void sheepIn ()

  Triggers the end of the game once a sheep collides with the barn.

Public Member Functions

- Barn (QObject ∗parent=0)

  Default constructor.
- virtual ∼Barn ()

  Destructor.

7.1.1 Constructor & Destructor Documentation

7.1.1.1 Barn::Barn (QObject ∗ *parent* = 0) `[explicit]`

Default constructor.

Sets the barn image and timer to check for collisions

7.1.1.2 Barn::∼Barn ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.1.2 Member Function Documentation

7.1.2.1 void Barn::sheepIn ( ) [slot]

Triggers the end of the game once a sheep collides with the barn.

Called by the timer, checks if there are colliding items with the barn If the sheep is part of the moving line, stop the game Otherwise, the sheep was shot and the game proceeds normally

The documentation for this class was generated from the following files:

- game1/barn.h
- game1/barn.cpp

7.2 Box Class Reference

Public Member Functions

- Box (QObject ∗parent=0)

  Default constructor.
- virtual ∼Box ()

  Destructor.
- void drawShaun ()

  Sets pixmap to Shaun.
- void drawBitzer ()

  Sets pixmap to Bitzer.
- void setAbove ()

  Marks that the top of the box has been drawn.
- void setLeft ()

  Marks that the left of the box has been drawn.
- void setUnder ()

  Marks that the bottom of the box has been drawn.
- void setRight ()

  Marks that the right of the box has been drawn.
- bool isClosed () const

  Checks if the box has been closed.
- int numberOfLinesDrawn () const

  Checks how many lines are drawn in the box.
- bool wasClosedByUser () const

  Returns whether the user has closed this box.

7.2.1 Constructor & Destructor Documentation

7.2.1.1 Box::Box (QObject $*$ *parent* = 0) [explicit]

Default constructor.

Sets Box properties.

7.2.1.2 Box::∼Box ( ) [virtual]

Destructor.

Frees allocated memory.

7.2.2 Member Function Documentation

7.2.2.1 void Box::drawBitzer ( )

Sets pixmap to Bitzer.

Draws Bitzer on the box.

7.2.2.2 void Box::drawShaun ( )

Sets pixmap to Shaun.

Draws Shaun on the box.

7.2.2.3 bool Box::isClosed ( ) const

Checks if the box has been closed.

Returns

Whether the box has been closed

7.2.2.4 int Box::numberOfLinesDrawn ( ) const

Checks how many lines are drawn in the box.

Returns

The number of lines drawn

Checks if the box is one line away from being closed.

7.2.2.5 bool Box::wasClosedByUser ( ) const

Returns whether the user has closed this box.

Returns

Whether the user has closed this box

The documentation for this class was generated from the following files:

- game3/box.h
- game3/box.cpp

7.3 QCPAxisPainterPrivate::CachedLabel Struct Reference

Public Attributes

- QPointF **offset**
- QPixmap **pixmap**

The documentation for this struct was generated from the following file:

- account/qcustomplot.h

7.4 Cannon Class Reference

Public Member Functions

- Cannon (QObject ∗parent=0)

  Default constructor.
- virtual ∼Cannon ()

  Destructor.
- void keyPressEvent (QKeyEvent ∗event)

  Entrance point of triggered key events.
- void rotateCannon (bool toTheRight)

  Rotates the cannon.

7.4.1 Constructor & Destructor Documentation

7.4.1.1 Cannon::Cannon (QObject ∗ *parent* = 0) [explicit]

Default constructor.

Sets the cannonimage and initializes variables.

7.4.1.2 Cannon::∼Cannon ( ) [virtual]

Destructor.

Frees allocated memory.

7.4.2 Member Function Documentation

7.4.2.1 void Cannon::keyPressEvent (QKeyEvent ∗ *event*)

Entrance point of triggered key events.

Parameters

event	The event that has been triggered

Checks the key that triggered the event. If the key was a left or right arrow key, the cannon rotates left or right. If the key was a space, a sheep is thrown.

7.4.2.2 void Cannon::rotateCannon (bool *toTheRight*)

Rotates the cannon.

Parameters

toTheRight	Whether the movement should be to the right or to the left

The documentation for this class was generated from the following files:

- game1/cannon.h
- game1/cannon.cpp

7.5 Dot Class Reference

Public Member Functions

- Dot (QObject ∗parent=0)

  Default constructor.
- virtual ∼Dot ()

  Destructor.

7.5.1 Constructor & Destructor Documentation

7.5.1.1 Dot::Dot (QObject ∗ *parent* = 0) [explicit]

Default constructor.

Sets Dot properties.

7.5.1.2 Dot::∼Dot ( ) `[virtual]`

Destructor.

Frees allocated memory.

The documentation for this class was generated from the following files:

- game3/dot.h
- game3/dot.cpp

7.6 Game1 Class Reference

Public Slots

- void goToMainMenu ()

    Slot to go back to the games main menu when pressing Exit.
- void endGame (bool win)

    Slot to handle ending the game once it's over.
- void replay ()

    Reloads the game with the same level.
- void next ()

    Proceed to the next level.
- void save ()

    Saves and exits.
- void closeEvent (QCloseEvent ∗bar)

    Deletes the saved game on close.

Public Member Functions

- Game1 (int level, bool resume=false, QWidget ∗parent=0)

    Constructor.
- virtual ∼Game1 ()

    Destructor.
- void loadNewGame (bool sameLevel)

    Load new game.

7.6.1 Constructor & Destructor Documentation

7.6.1.1 Game1::Game1 (int *level*, bool *resume* = `false`, QWidget ∗ *parent* = `0`) `[explicit]`

Constructor.

Parameters

level	Game level
resume	Whether this is a new game

Sets the size of the window, initializes the graphic items, sets the layouts and connects buttons to their slots.

7.6.1.2 Game1::∼Game1 ( ) [virtual]

Destructor.

Frees allocated memory

7.6.2 Member Function Documentation

7.6.2.1 void Game1::closeEvent (QCloseEvent ∗ *bar*) [slot]

Deletes the saved game on close.

Parameters

bar	The event triggered

Deletes the saved game on close unless it is a newly saved game.

7.6.2.2 void Game1::endGame (bool *win*) [slot]

Slot to handle ending the game once it's over.

Parameters

win	Indicates if the user has won the game

Removes the save and exit button and adds the go back and replay buttons along with their connections

7.6.2.3 void Game1::goToMainMenu ( ) [slot]

Slot to go back to the games main menu when pressing Exit.

Goes to the main menu of Sheep Line

7.6.2.4 void Game1::loadNewGame (bool *sameLevel*)

Load new game.

Parameters

sameLevel	Indicates if the level is the same

Loads a new game of either the same level or the next

7.6.2.5 void Game1::next ( ) [slot]

Proceed to the next level.

Proceed to the next level

7.6.2.6 void Game1::replay ( ) [slot]

Reloads the game with the same level.

Loads a new instance of the Game1 Scene

7.6.2.7 void Game1::save ( ) [slot]

Saves and exits.

Saves and exits

The documentation for this class was generated from the following files:

- game1/game1.h
- game1/game1.cpp

7.7 Game1Options Class Reference

Public Slots

- void gotoGameMainMenu ()

  Takes the user to the game main menu.
- void gotoGame1 (int level)

  Takes the user to game 1.

Public Member Functions

- Game1Options (QWidget ∗parent=0)

  Default constructor.
- virtual ∼Game1Options ()

  Destructor.

Static Public Member Functions

- static int getNumberOfUnlockedLevels ()

  Returns the number of levels in game 1 that the user has unlocked.
- static void unlockExtraLevel (int currLevel)

  Unlocks one new game 1 level if appropriate.

7.7.1 Constructor & Destructor Documentation

7.7.1.1 Game1Options::Game1Options (QWidget ∗ *parent* = 0) [explicit]

Default constructor.

Initializes all buttons and text and shows them on the screen. Also initializes connections.

7.7.1.2 Game1Options::∼Game1Options ( ) [virtual]

Destructor.

Frees allocated memory

7.7.2 Member Function Documentation

7.7.2.1 int Game1Options::getNumberOfUnlockedLevels ( ) [static]

Returns the number of levels in game 1 that the user has unlocked.

Returns

Number of levels

Returns the number of levels that the user has unlocked.

7.7.2.2 void Game1Options::gotoGame1 (int *level*) [slot]

Takes the user to game 1.

Parameters

level	The level of the game

Takes the user to game 1. Called after clicking any level button.

7.7.2.3 void Game1Options::gotoGameMainMenu ( ) [slot]

Takes the user to the game main menu.

Takes the user to the game main menu that corresponds to game 1. Called after clicking the corresponding button.

7.7.2.4 void Game1Options::unlockExtraLevel (int *currLevel*) [static]

Unlocks one new game 1 level if appropriate.

Parameters

currLevel	Level that was just won

Unlocks a new level if the user has just won the last unlocked level. Called after a user wins a certain level.

The documentation for this class was generated from the following files:

- game1/game1options.h
- game1/game1options.cpp

7.8 Game1Scene Class Reference

Public Slots

- void move_line ()

  Move sheep in the line according to a straight line then circle.

Signals

- void Done (bool)

  Signals Game1 that the game is over.

Public Member Functions

- Game1Scene (int level, bool resume=false, QObject ∗parent=0)

  Constructor.
- virtual ∼Game1Scene ()

  Destructor.
- void mousePressEvent (QGraphicsSceneMouseEvent ∗)

  Adjusts focus.
- void moveCurrentSheep (bool toRight)

  Moves sheep with rotating cannon.

- void gameOver (bool win)

  Stops movement of the sheep and triggers finishing the game.
- void fireSheep ()

  Fires the sheep.
- bool collidesWithSheepInLine (QGraphicsItem ∗item)

  Checks the collision list of the item for a sheep in the sheep list.
- int getScore () const

  Returns the current player score.
- void freeze ()

  Freezes gameplay.
- int getCannonAngle () const

  Returns the cannon angle.
- int getCurrentSheepNumber () const

  Returns the number on the current sheep.
- int getNextSheepNumber () const

  Returns the number on the next sheep.
- QString getLastLinePosition () const

  Returns the position of the first in-line sheep.
- QString getInLineSheepNumbers () const

  Returns the numbers of all sheep in the line.

7.8.1 Constructor & Destructor Documentation

7.8.1.1 Game1Scene::Game1Scene (int *level*, bool *resume* = `false`, QObject ∗ *parent* = `0`) `[explicit]`

Constructor.

Parameters

level	Level of the game
resume	Whether this is a new game

Initializes variables and connections.

7.8.1.2 Game1Scene::∼Game1Scene ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.8.2 Member Function Documentation

7.8.2.1 bool **Game1Scene::collidesWithSheepInLine** (QGraphicsItem ∗ *item*)

Checks the collision list of the item for a sheep in the sheep list.

Parameters

item	Pointer to item to check

Returns

Whether the given item collides with a sheep in the list

Returns whether the given item collides with a sheep in the line.

7.8.2.2 void **Game1Scene::fireSheep** ( )

Fires the sheep.

Releases the sheep and makes it move in a straight line.

7.8.2.3 void **Game1Scene::freeze** ( )

Freezes gameplay.

Freezes gameplay. Called when the user stops the game mid-play.

7.8.2.4 void **Game1Scene::gameOver** (bool *win*)

Stops movement of the sheep and triggers finishing the game.

Parameters

win	Indicates if the game has been won

Ends the game

7.8.2.5 int **Game1Scene::getCannonAngle** ( ) const

Returns the cannon angle.

Returns

The cannon angle

Returns the cannon angle Called when saving the game.

7.8.2.6 int **Game1Scene::getCurrentSheepNumber** ( ) const

Returns the number on the current sheep.

Returns

The number on the current sheep

Returns the number on the current sheep. Called when saving the game.

7.8.2.7 QString **Game1Scene::getInLineSheepNumbers** ( ) const

Returns the numbers of all sheep in the line.

Returns

QString containing all numbers of the sheep in the line

Returns the numbers of all sheep in the line in a QString. Called when saving the game.

7.8.2.8 QString **Game1Scene::getLastLinePosition** ( ) const

Returns the position of the first in-line sheep.

Returns

Position of the first in-line sheep

Returns the position of the first in-line sheep. Called when saving the game.

7.8.2.9 int **Game1Scene::getNextSheepNumber** ( ) const

Returns the number on the next sheep.

Returns

The number on the next sheep

Returns the number on the next sheep. Called when saving the game.

7.8.2.10 int **Game1Scene::getScore** ( ) const

Returns the current player score.

Returns

Current player score

Returns the player score for the current game. For each destroyed sheep, the player is awarded 10 points. When the player wins the game, they are awarded m_scoreDisplay->display(m_score); 1 point per remaining sheep in-line move.

7.8.2.11 void Game1Scene::mousePressEvent (QGraphicsSceneMouseEvent *)

Adjusts focus.

Sets focus on the cannon

7.8.2.12 void Game1Scene::move_line ( ) [slot]

Move sheep in the line according to a straight line then circle.

Moves the sheep in the line according to their position on the screen

7.8.2.13 void Game1Scene::moveCurrentSheep (bool *toRight*)

Moves sheep with rotating cannon.

Parameters

toRight	Indicates the direction of the cannon move

Moves current sheep. Called when the cannon rotates.

The documentation for this class was generated from the following files:

- game1/game1scene.h
- bin/moc_game1scene.cpp
- game1/game1scene.cpp

7.9 Game2 Class Reference

Public Slots

- void goToMainMenu ()

  Slot to go back to the games main menu when pressing Exit.
- void endGame ()

  removes save and exit button and adds replay and back buttons
- void replay ()

  removes save and exit button and adds replay and back buttons
- void save ()

Saves the state of the game into the database.

- void closeEvent (QCloseEvent ∗bar)

  Deletes the saved game on close.

Public Member Functions

- Game2 (Difficulty difficulty, bool resume=false, QWidget ∗parent=0)

  Default constructor.
- virtual ∼Game2 ()

  Destructor.

7.9.1 Constructor & Destructor Documentation

7.9.1.1 Game2::Game2 (Difficulty *difficulty*, bool *resume* = `false`, QWidget ∗ *parent* = `0`) `[explicit]`

Default constructor.

Parameters

difficulty	Difficulty of the game
resume	Whether this is a new game

Sets the size of the window, initializes the graphic items, sets the layouts and connects buttons to their slots.

7.9.1.2 Game2::∼Game2 ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.9.2 Member Function Documentation

7.9.2.1 void Game2::closeEvent (QCloseEvent ∗ *bar*) `[slot]`

Deletes the saved game on close.

Parameters

bar	The event triggered

7.9.2.2 void Game2::endGame ( ) `[slot]`

removes save and exit button and adds replay and back buttons

Removes the save and exitbutton and adds the go back and replay buttons along with their connections.

7.9.2.3 void Game2::goToMainMenu ( ) [slot]

Slot to go back to the games main menu when pressing Exit.

Goes to the main menu of Trap the Sheep.

7.9.2.4 void Game2::replay ( ) [slot]

removes save and exit button and adds replay and back buttons

Reloads game.

The documentation for this class was generated from the following files:

- game2/game2.h
- game2/game2.cpp

7.10 Game2Options Class Reference

Public Slots

- void gotoGameMainMenu ()

  Takes the user to the game main menu.
- void gotoGame2 (int difficulty)

  Takes the user to game 2.

Public Member Functions

- Game2Options (QWidget ∗parent=0)

  Constructor.
- virtual ∼Game2Options ()

  Destructor.

7.10.1 Constructor & Destructor Documentation

7.10.1.1 Game2Options::Game2Options (QWidget ∗ *parent* = 0) [explicit]

Constructor.

Initializes all buttons and text and shows them on the screen. Also initializes connections.

7.10.1.2 Game2Options::∼Game2Options ( ) [virtual]

Destructor.

Frees allocated memory.

7.10.2 Member Function Documentation

7.10.2.1 void Game2Options::gotoGame2 (int *difficulty*) [slot]

Takes the user to game 2.

Takes the user to game 2. Called after clicking any level button.

7.10.2.2 void Game2Options::gotoGameMainMenu ( ) [slot]

Takes the user to the game main menu.

Takes the user to the game main menu that corresponds to game 2. Called after clicking the corresponding button.

The documentation for this class was generated from the following files:

- game2/game2options.h
- game2/game2options.cpp

7.11 Game2Scene Class Reference

Public Slots

- void moveSheep ()

  moves the sheep according to the difficulty of the game

Signals

- void Done ()

  Signal sent to game2 to show that the game has ended.

Public Member Functions

- Game2Scene (Difficulty difficulty, bool resume, QObject ∗parent=0)

  Default constructor.
- virtual ∼Game2Scene ()
- QVector< int > tilesToBlock ()

  Sets the layout of the level buttons.

- void placeTiles ()

  Places the tiles to create the full grid.
- void placeTilesResumed (QStringList positions)

  Places tiles with blocks loaded from prevously saved game.
- void placeSheepInitial ()

  places the sheep on a random unblocked tile at the beginning of the game
- void placeSheepInitialResumed (Tile ∗tile)

  places the sheep on a the given unblocked tile at the beginning of the game
- QVector< Tile ∗ > ∗ getNeighbors (Tile ∗center)

  Gets the left, right, upper and lower non-blocked neighbors of the given tile.
- void resetVisited ()

  Sets all the blocks of the grid as having been unvisited.
- void resetDistances ()

  Sets all the blocks of the grid as having infinite distance to the sheep.
- void resetPrevious ()

  Sets all the blocks of the grid as having the previous node null.
- Tile ∗ tileAt (int i, int j)

  retrieves the tile from the grid at the indices given
- bool win (Tile ∗tile)

  determines if the user sucessfully trapped the sheep
- Sheep2 ∗ getSheep ()

  gets the sheep of the game
- bool getUserTurn ()

  retrieves whether or not it is the user's turn
- void setUserTurn (bool userTurn)

  sets the turn of the user
- void computerTurn ()

  delays the computer turn
- void gameOver (bool win)

  Ends the game.
- void decrementScore ()

  Increments the number of blocks on click of a tile.
- QLCDNumber ∗ getScoreDisplay ()

  retrieves the lcd display
- int getBlockCount ()

  retrieves the number of blocked tiles
- Tile ∗ findNextTile ()

  Finds the shortest path and returns the next tile accordingly.
- void computeDistances (Tile ∗current)

  Computes the distances from the sheep to every other tile.
- QVector< Tile ∗ > ∗ getNonBlockedBorders ()

  Retrieves the non blocked border tiles.
- Difficulty getDifficulty ()

Returns the difficulty of the game.
- int getScore ()

  Returns the current score of the game.
- QString getSheepPos ()

  Returns a string representation of the sheep row and column.
- QString getBlockedTilesPos ()

  Returns the string representation of the blocked tiles' positions.
- void addToBlockedTiles (Tile ∗tile)

  Adds a tile to the list of blocked tiles.
- void placeLCD ()

  Places the LCD on the screen.

7.11.1 Constructor & Destructor Documentation

7.11.1.1 Game2Scene::Game2Scene (Difficulty *difficulty*, bool *resume*, QObject ∗ *parent* = 0) [explicit]

Default constructor.

Parameters

difficulty	Difficulty of the game
resume	Whether this is a new game

Places the items on the scene and sets the user turn.

7.11.1.2 Game2Scene::∼Game2Scene ( ) [virtual]

Destructor

Frees allocated memory.

7.11.2 Member Function Documentation

7.11.2.1 void Game2Scene::addToBlockedTiles (Tile ∗ *tile*)

Adds a tile to the list of blocked tiles.

Parameters

tile	The tile to add to the list

Adds a tile to the list of blocked tiles

7.11.2.2 void Game2Scene::computerTurn ( )

delays the computer turn

Starts the timer to delay the computer move.

7.11.2.3 void Game2Scene::decrementScore ( )

Increments the number of blocks on click of a tile.

Increments the number of blocks by one on click of a tile.

7.11.2.4 void Game2Scene::gameOver (bool *win*)

Ends the game.

Parameters

win	whether or not the user won

Ends the game and displays the GameOver item.

7.11.2.5 int Game2Scene::getBlockCount ( )

retrieves the number of blocked tiles

Returns

number of blocked tiles

Returns the number of blocked tiles.

7.11.2.6 QString Game2Scene::getBlockedTilesPos ( )

Returns the string representation of the blocked tiles' positions.

Returns

string representing the blocked tiles' positions

7.11.2.7 Difficulty Game2Scene::getDifficulty ( )

Returns the difficulty of the game.

Returns

the difficulty of the game

7.11.2.8 QVector< Tile * > * Game2Scene::getNeighbors (Tile * *center*)

Gets the left, right, upper and lower non-blocked neighbors of the given tile.

Parameters

center	The tile that we're getting the neighbors of

Returns

Pointer to a vector of pointers to the neighboring tiles

Gets the left, right, upper and bottom neighbors for the given tile. Memory is allocated in this function and should be freed by the caller.

7.11.2.9 QVector< Tile * > * Game2Scene::getNonBlockedBorders ( )

Retrieves the non blocked border tiles.

Retrieves the non blocked border tiles. Memory is allocated in this function and should be freed by the caller.

7.11.2.10 int Game2Scene::getScore ( )

Returns the current score of the game.

Returns

the score of the game

7.11.2.11 QLCDNumber * Game2Scene::getScoreDisplay ( )

retrieves the lcd display

Returns

the lcd display

Returns the score lcd.

7.11.2.12 Sheep2 * Game2Scene::getSheep ( )

gets the sheep of the game

Returns

the sheep of the game

Retrieves the sheep.

7.11.2.13 QString **Game2Scene::getSheepPos ( )**

Returns a string representation of the sheep row and column.

Returns

string representing the sheep's current row and column

7.11.2.14 bool **Game2Scene::getUserTurn ( )**

retrieves whether or not it is the user's turn

Returns

boolean indicating if it's the user's turn

Retrieves the value that indicates if it's the user's turn.

7.11.2.15 void **Game2Scene::moveSheep ( )** `[slot]`

moves the sheep according to the difficulty of the game

Moves the sheep according to the difficulty when it's the computer's turn.

7.11.2.16 void **Game2Scene::placeSheepInitial ( )**

places the sheep on a random unblocked tile at the beginning of the game

Chooses a random tile to place the sheep on at the beginning of the game.

7.11.2.17 void **Game2Scene::placeSheepInitialResumed (Tile ∗ *tile*)**

places the sheep on a the given unblocked tile at the beginning of the game

Parameters

tile	The tile to place the sheep on

Chooses a random tile to place the sheep on at the beginning of the game.

7.11.2.18 void **Game2Scene::placeTiles ( )**

Places the tiles to create the full grid.

Places the tiles on the grid of the game.

7.11.2.19 void **Game2Scene::placeTilesResumed** (QStringList *positions*)

Places tiles with blocks loaded from prevously saved game.

Parameters

positions	Loaded positions

Places tiles with blocks loaded from previously saved game

7.11.2.20 void **Game2Scene::resetDistances** ( )

Sets all the blocks of the grid as having infinite distance to the sheep.

Resets the status of the grid tiles as having infinite distance to the sheep.

7.11.2.21 void **Game2Scene::resetPrevious** ( )

Sets all the blocks of the grid as having the previous node null.

Resets the status of the grid tiles as having their previous tile NULL.

7.11.2.22 void **Game2Scene::resetVisited** ( )

Sets all the blocks of the grid as having been unvisited.

Resets the status of the grid tiles as not visited.

7.11.2.23 void **Game2Scene::setUserTurn** (bool *userTurn*)

sets the turn of the user

Parameters

userTurn	whether or not it's the user's turn

Sets the user's turn as true or false.

7.11.2.24 **Tile** ∗ **Game2Scene::tileAt** (int *i*, int *j*)

retrieves the tile from the grid at the indices given

Parameters

i	the row of the tile
j	the column of the tile

Returns

pointer to the tile at the indices

Gets the tile at the given indices.

7.11.2.25 QVector< int > Game2Scene::tilesToBlock ( )

Sets the layout of the level buttons.

Returns

The indices of the tiles to block initially

chooses distinct random tiles to flag for blocking initially at the start of the game.

7.11.2.26 bool Game2Scene::win (Tile * *tile*)

determines if the user sucessfully trapped the sheep

Parameters

tile	the tile to check the neighbors of

Returns

the state of the game if win or loss

Determines a win by checking if from the current sheep position it's possible to get to the border without encountering a blocked tile.

The documentation for this class was generated from the following files:

- game2/game2scene.h
- bin/moc_game2scene.cpp
- game2/game2scene.cpp

7.12 Game3 Class Reference

Public Slots

- void endGame ()

  Removes save and exit button and adds replay and back buttons.
- void replay ()

  Removes save and exit button and adds replay and back buttons.
- void goToMainMenu ()

Slot to go back to the games main menu when pressing Exit.
- void save ()

  Saves and exits.
- void closeEvent (QCloseEvent ∗bar)

  Deletes the saved game on close.

Public Member Functions

- Game3 (Difficulty difficulty, Size size, bool resume=false, QWidget ∗parent=0)

  Constructor.
- virtual ∼Game3 ()

  Destrucor.

7.12.1 Constructor & Destructor Documentation

7.12.1.1 Game3::Game3 (Difficulty *difficulty*, Size *size*, bool *resume* = `false`, QWidget ∗ *parent* = `0`) `[explicit]`

Constructor.

Parameters

difficulty	Difficulty of the game
size	Size of the game
resume	Whether this is a new game

Sets the size of the window, initializes the graphic items, sets the layouts and connects buttons to their slots.

7.12.1.2 Game3::∼Game3 ( ) `[virtual]`

Destrucor.

Frees allocated memory.

7.12.2 Member Function Documentation

7.12.2.1 void Game3::closeEvent (QCloseEvent ∗ *bar*) `[slot]`

Deletes the saved game on close.

Parameters

bar	The event triggered

Deletes the saved game on close unless it is a newly saved game.

7.12.2.2 void Game3::endGame ( ) [slot]

Removes save and exit button and adds replay and back buttons.

Removes the save and exitbutton and adds the go back and replay buttons along with their connections.

7.12.2.3 void Game3::goToMainMenu ( ) [slot]

Slot to go back to the games main menu when pressing Exit.

Goes back to the Main meny of Dots and Lines.

7.12.2.4 void Game3::replay ( ) [slot]

Removes save and exit button and adds replay and back buttons.

Reloads game.

7.12.2.5 void Game3::save ( ) [slot]

Saves and exits.

Saves and exits

The documentation for this class was generated from the following files:

- game3/game3.h
- game3/game3.cpp

7.13 Game3Options Class Reference

Public Slots

- void gotoGameMainMenu ()

  Takes the user to the game main menu.
- void gotoGame ()

  Takes the user to game 3.
- void setEasy ()

  Sets the game difficulty to Easy.
- void setModerate ()

  Sets the game difficulty to Moderate.
- void setHard ()

  Sets the game difficulty to Hard.
- void setSizeFour ()

  Sets the game size to 4x4.

- void setSizeEight ()

  Sets the game size to 8x8.
- void setSizeSixteen ()

  Sets the game size to 16x16.

Public Member Functions

- Game3Options (QWidget ∗parent=0)

  Constructor.
- virtual ∼Game3Options ()

  Destructor.

7.13.1 Constructor & Destructor Documentation

7.13.1.1 Game3Options::Game3Options (QWidget ∗ *parent* = 0) [explicit]

Constructor.

Initializes all buttons and text and shows them on the screen. Also initializes connections.

7.13.1.2 Game3Options::∼Game3Options ( ) [virtual]

Destructor.

Frees allocated memory.

7.13.2 Member Function Documentation

7.13.2.1 void Game3Options::gotoGame ( ) [slot]

Takes the user to game 3.

Takes the user to game 3. Called after clicking any level button.

7.13.2.2 void Game3Options::gotoGameMainMenu ( ) [slot]

Takes the user to the game main menu.

Takes the user to the game main menu that corresponds to game 3. Called after clicking the corresponding button.

7.13.2.3 void Game3Options::setEasy ( ) [slot]

Sets the game difficulty to Easy.

Sets the game difficulty to Easy. Called after clicking the corresponding button.

7.13.2.4 void Game3Options::setHard ( ) [slot]

Sets the game difficulty to Hard.

Sets the game difficulty to Hard. Called after clicking the corresponding button.

7.13.2.5 void Game3Options::setModerate ( ) [slot]

Sets the game difficulty to Moderate.

Sets the game difficulty to Moderate. Called after clicking the corresponding button.

7.13.2.6 void Game3Options::setSizeEight ( ) [slot]

Sets the game size to 8x8.

Sets the game size to 8x8. Called after clicking the corresponding button.

7.13.2.7 void Game3Options::setSizeFour ( ) [slot]

Sets the game size to 4x4.

Sets the game size to 4x4. Called after clicking the corresponding button.

7.13.2.8 void Game3Options::setSizeSixteen ( ) [slot]

Sets the game size to 16x16.

Sets the game size to 16x16. Called after clicking the corresponding button.

The documentation for this class was generated from the following files:

- game3/game3options.h
- game3/game3options.cpp

7.14 Game3Scene Class Reference

Public Slots

- void computerMove ()

  Computer move.

Signals

- void done ()

  Signals Game3 that the game is over.

Public Member Functions

- Game3Scene (Difficulty difficulty, Size size, bool resume=false, QObject ∗parent=0)

  Constructor.
- virtual ∼Game3Scene ()

  Destructor.
- void computerTurn ()

  Computer turn.
- bool isUserTurn ()

  Returns whose turn it is to play.
- void addNewlyDrawnLine (Line ∗line)

  Remembers newly drawn line.
- void clearNewLines ()

  Clears new lines by turning them grey.
- void gameOver ()

  Triggers finishing the game.
- bool noMoreMoves ()

  Returns whether there are any moves left.
- void closeBoxByUser ()

  Declares one more box as closed by user.
- void closeBoxByComputer ()

  Declares one more box as closed by computer.
- Line ∗ getLineThatClosesBox () const

  Finds and returns a non-clicked line that closes at least one box.
- Line ∗ getSmartLine () const

  Finds and returns a non-clicked line that does not let the user close a box.
- void freeze ()

  Freezes gameplay.
- int getScore () const

  Returns the game score.
- QString getBoxesClosedByUser () const

  Returns which boxes have been closed by the user.
- QString getBoxesClosedByPC () const

  Returns which boxes have been closed by the PC.
- QString getDrawnHorizontalLines () const

  Returns which horizontal lines have already been drawn.
- QString getDrawnVerticalLines () const

  Returns which vertical lines have already been drawn.

7.14.1 Constructor & Destructor Documentation

7.14.1.1 Game3Scene::Game3Scene (Difficulty *difficulty*, Size *size*, bool *resume* = `false`, QObject ∗ *parent* = `0`) `[explicit]`

Constructor.

Parameters

difficulty	Difficulty of the game
size	Size of the game
resume	Whether this is a new game

Initializes the difficulty, size, dots, lines and boxes of the game.

7.14.1.2 Game3Scene::∼Game3Scene ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.14.2 Member Function Documentation

7.14.2.1 void Game3Scene::addNewlyDrawnLine (Line ∗ *line*)

Remembers newly drawn line.

Parameters

line	Newly drawn line

Remembers newly drawn line so it can be turned grey later. Also removes the line from the list of unmarked lines.

7.14.2.2 void Game3Scene::clearNewLines ( )

Clears new lines by turning them grey.

Clears new lines by turning them grey and removing them from the vector.

7.14.2.3 void Game3Scene::computerMove ( ) `[slot]`

Computer move.

Picks a line to select according to difficulty, and plays the turn.

7.14.2.4 void Game3Scene::computerTurn ( )

Computer turn.

Starts a delay to call computerMove().

7.14.2.5 void Game3Scene::freeze ( )

Freezes gameplay.

Freezes gameplay. Called when the user stops the game mid-play.

7.14.2.6 void Game3Scene::gameOver ( )

Triggers finishing the game.

Ends the game.

7.14.2.7 QString Game3Scene::getBoxesClosedByPC ( ) const

Returns which boxes have been closed by the PC.

Returns

QString listing boxes that have been closed by the PC

Returns a QString with information on which boxes have been closed by the PC. Called when saving the game.

7.14.2.8 QString Game3Scene::getBoxesClosedByUser ( ) const

Returns which boxes have been closed by the user.

Returns

QString listing boxes that have been closed by the user

Returns a QString with information on which boxes have been closed by the user. Called when saving the game.

7.14.2.9 QString Game3Scene::getDrawnHorizontalLines ( ) const

Returns which horizontal lines have already been drawn.

Returns

QString listing horizontal lines that have been drawn

Returns a QString with information on which horizontal lines have been drawn. Called when saving the game.

7.14.2.10 QString **Game3Scene::getDrawnVerticalLines** ( ) const

Returns which vertical lines have already been drawn.

Returns

QString listing vertical lines that have been drawn

Returns a QString with information on which vertical lines have been drawn. Called when saving the game.

7.14.2.11 **Line** ∗ **Game3Scene::getLineThatClosesBox** ( ) const

Finds and returns a non-clicked line that closes at least one box.

Returns

Non-clicked line that closes a box

Finds and returns a non-clicked line that closes at least one box. Returns NULL if not found.

7.14.2.12 int **Game3Scene::getScore** ( ) const

Returns the game score.

Returns

Game score

7.14.2.13 **Line** ∗ **Game3Scene::getSmartLine** ( ) const

Finds and returns a non-clicked line that does not let the user close a box.

Returns

Non-clicked line that does not let the user close a box next

Finds and returns a non-clicked line that does not let the user close a box. It does so by checking that the returned line is not the third line to be drawn around any box. If such a line is not found, it returns NULL.

7.14.2.14 bool **Game3Scene::isUserTurn** ( )

Returns whose turn it is to play.

Returns

Whether it is the user's turn to play

Returns whether it is the user's turn to play.

7.14.2.15 bool Game3Scene::noMoreMoves ( )

Returns whether there are any moves left.

Returns

Whether there are any unmarked lines left

Returns whether there are any unmarked lines left.

The documentation for this class was generated from the following files:

- game3/game3scene.h
- bin/moc_game3scene.cpp
- game3/game3scene.cpp

7.15 GameMainMenu Class Reference

Public Slots

- void gotoGameOptions ()

  Takes the user to the game options.
- void gotoGameSelection ()

  Takes the user to the game selection menu.
- void resumeSavedGame ()

  Takes the user to his/her saved game.

Public Member Functions

- GameMainMenu (int gameChoice, QWidget ∗parent=0)

  Constructor.
- virtual ∼GameMainMenu ()

  Destructor.

7.15.1 Constructor & Destructor Documentation

7.15.1.1 GameMainMenu::GameMainMenu (int *gameChoice*, QWidget ∗ *parent* = 0) [explicit]

Constructor.

Initializes all buttons and the game title and instructions and shows them on the screen. Also initializes connections.

7.15.1.2 GameMainMenu::∼GameMainMenu ( ) [virtual]

Destructor.

Frees allocated memory.

7.15.2 Member Function Documentation

7.15.2.1 void GameMainMenu::gotoGameOptions ( ) [slot]

Takes the user to the game options.

Takes the user to the game option widget that corresponds to the game choice. Called after clicking "Play".

7.15.2.2 void GameMainMenu::gotoGameSelection ( ) [slot]

Takes the user to the game selection menu.

Takes the user back to the game selection menu. Called after clicking the corresponding button.

7.15.2.3 void GameMainMenu::resumeSavedGame ( ) [slot]

Takes the user to his/her saved game.

Takes the user to his/her saved game. Called after calling resume.

The documentation for this class was generated from the following files:

- gui/gamemainmenu.h
- gui/gamemainmenu.cpp

7.16 GameOver Class Reference

Public Member Functions

- GameOver (bool win, QObject ∗parent=0)

    Default constructor.
- virtual ∼GameOver ()

    Destructor.

7.16.1 Constructor & Destructor Documentation

7.16.1.1 GameOver::GameOver (bool *win*, QObject ∗ *parent* = 0) [explicit]

Default constructor.

Parameters

win	Indicates if the game has been won

Sets origin of the image depending on win state.

7.16.1.2 GameOver::∼GameOver ( ) [virtual]

Destructor.

Frees allocated memory.

The documentation for this class was generated from the following files:

- gameover.h
- gameover.cpp

7.17 GameSelection Class Reference

Public Slots

- void goToMyAccount ()
  Takes the user from the game selection menu to the My Account page.
- void goToMain ()
  Takes the user from the game selection menu to the main welcome window.
- void goToGame1 ()
  Takes the user from the game selection menu to the Sheep Line game.
- void goToGame2 ()
  Takes the user from the game selection menu to the Trap the Sheep game.
- void goToGame3 ()
  Takes the user from the game selection menu to the Dots and Lines game.

Public Member Functions

- GameSelection (QWidget ∗parent=0)
  Default constructor.
- virtual ∼GameSelection ()
  Destructor.

7.17.1 Constructor & Destructor Documentation

7.17.1.1 GameSelection::GameSelection (QWidget ∗ *parent* = 0) [explicit]

Default constructor.

Initializes all buttons and labels and shows them on the game selection menu.

7.17.1.2 GameSelection::∼GameSelection ( ) [virtual]

Destructor.

Frees allocated memory.

7.17.2 Member Function Documentation

7.17.2.1 void GameSelection::goToGame1 ( ) [slot]

Takes the user from the game selection menu to the Sheep Line game.

Takes the user to game 1 when the appropriate button is clicked.

7.17.2.2 void GameSelection::goToGame2 ( ) [slot]

Takes the user from the game selection menu to the Trap the Sheep game.

Takes the user to game 2 when the appropriate button is clicked.

7.17.2.3 void GameSelection::goToGame3 ( ) [slot]

Takes the user from the game selection menu to the Dots and Lines game.

Takes the user to game 3 when the appropriate button is clicked.

7.17.2.4 void GameSelection::goToMain ( ) [slot]

Takes the user from the game selection menu to the main welcome window.

Takes the user back to the login widget. Called when the logout button is clicked.

7.17.2.5 void GameSelection::goToMyAccount ( ) [slot]

Takes the user from the game selection menu to the My Account page.

Takes the user to his/her account page. Called when the "Account" button is clicked.

The documentation for this class was generated from the following files:

- gui/gameselection.h
- gui/gameselection.cpp

7.18 Helper Class Reference

Static Public Member Functions

- static void makeWidgetSmall (QWidget ∗widget)

Makes given widget small.
- static void makeWidgetLarge (QWidget ∗widget)

  Makes given widget large.
- static double toRadians (double degrees)

  Takes an angle in degrees, turns it into radians.
- static void deleteSavedGame (int gameNumber)

  Removes saved game for given game.
- static int getUserId ()

  Retrieves the user ID.
- static void setUserId (int id)

  Sets the user ID.
- static void initialize ()

  Initializes class attributes.

Static Public Attributes

- static const double PI = 3.14159265

  PI.
- static QSqlDatabase shaunDB

  The database for the game.

7.18.1 Member Function Documentation

7.18.1.1 void **Helper::deleteSavedGame (** int *gameNumber* **)** `[static]`

Removes saved game for given game.

Parameters

game-Number	The game whose entry we wish to delete

7.18.1.2 int **Helper::getUserId (** **)** `[static]`

Retrieves the user ID.

Returns

The user ID

7.18.1.3 void **Helper::initialize (** **)** `[static]`

Initializes class attributes.

Initializes static members. Called only once.

7.18.1.4 void **Helper::makeWidgetLarge (** QWidget ∗ *widget* **)** `[static]`

Makes given widget large.

Parameters

widget	Widget whose size to change

Makes the given widget large.

7.18.1.5 void **Helper::makeWidgetSmall (** QWidget ∗ *widget* **)** `[static]`

Makes given widget small.

Parameters

widget	Widget whose size to change

Makes the given widget small.

7.18.1.6 void **Helper::setUserId (** int *id* **)** `[static]`

Sets the user ID.

Parameters

id	The user ID

7.18.1.7 double **Helper::toRadians (** double *degrees* **)** `[static]`

Takes an angle in degrees, turns it into radians.

Parameters

degrees	Angle in degrees

Returns

Angle in radians

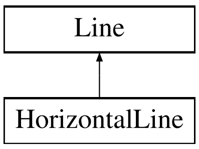
Converts angle from degrees to radians.

The documentation for this class was generated from the following files:

- helper.h
- helper.cpp

7.19 HorizontalLine Class Reference

Inheritance diagram for HorizontalLine:



Public Member Functions

- HorizontalLine (Box ∗above, Box ∗under, QObject ∗parent=0)

  Default constructor.
- virtual ∼HorizontalLine ()

  Destructor.
- void turnGrey ()

  Makes the line grey.
- bool playTurn (bool userTurn)

  Called when a line is selected.
- Box ∗ getAbove () const

  Gets the box above the line.
- Box ∗ getUnder () const

  Gets the box under the line.
- void simpleDraw ()

  Draws the line as grey and edits its corresponding boxes.

7.19.1 Constructor & Destructor Documentation

7.19.1.1 HorizontalLine::HorizontalLine (Box ∗ *above,* Box ∗ *under,* QObject ∗ *parent* = 0) `[explicit]`

Default constructor.

Parameters

above	Box above line
under	Box under line

Sets HorizontalLine properties.

7.19.1.2 HorizontalLine::∼HorizontalLine ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.19.2 Member Function Documentation

7.19.2.1 Box ∗ HorizontalLine::getAbove ( ) const

Gets the box above the line.

Returns

box above the line

Returns the box above the line.

7.19.2.2 Box ∗ HorizontalLine::getUnder ( ) const

Gets the box under the line.

Returns

box under the line

Returns the box under the line.

7.19.2.3 bool HorizontalLine::playTurn (bool *userTurn*) [virtual]

Called when a line is selected.

Parameters

userTurn	Whether it is the user's turn

Returns

Whether it is still the player's turn

Called when a line is drawn. Returns whether it is still the same player's turn.

Implements Line.

7.19.2.4 void HorizontalLine::turnGrey ( ) [virtual]

Makes the line grey.

Changes the object image to make it grey.

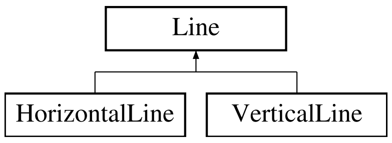
Implements Line.

The documentation for this class was generated from the following files:

- game3/horizontalline.h
- game3/horizontalline.cpp

7.20 Line Class Reference

Inheritance diagram for Line:



Public Member Functions

- Line (bool isHorizontal, QObject ∗parent=0)

  Default constructor.
- virtual void turnGrey ()=0

  Makes the line grey.
- void mousePressEvent (QGraphicsSceneMouseEvent ∗event)

  Called when the user clicks on the line.
- virtual bool playTurn (bool userTurn)=0

  Called when a line is selected.
- bool isDrawn () const

  Returns whether the line has been drawn already.
- void draw ()

  Sets the line as drawn.
- bool isHorizontal () const

  Returns whether the line is horizontal or vertical.
- virtual void simpleDraw ()=0

  Draws the line as grey and edits its corresponding boxes.

7.20.1 Constructor & Destructor Documentation

7.20.1.1 Line::Line (bool *isHorizontal,* QObject ∗ *parent* = 0) [explicit]

Default constructor.

Parameters

isHorizontal	Whether the line is horizontal or not (vertical)

Initializes the object and marks it as not drawn.

7.20.2 Member Function Documentation

7.20.2.1 bool **Line::isDrawn** ( ) const

Returns whether the line has been drawn already.

Returns

Whether the line has been drawn already

7.20.2.2 bool **Line::isHorizontal** ( ) const

Returns whether the line is horizontal or vertical.

Returns

Whether the line is horizontal

Returns whether the line is horizontal or vertical

7.20.2.3 void **Line::mousePressEvent** (QGraphicsSceneMouseEvent ∗ *event*)

Called when the user clicks on the line.

Called when the user clicks on the line. The function changes the states of corresponding lines and boxes. It then checks for a win.

7.20.2.4 virtual bool **Line::playTurn** (bool *userTurn*) `[pure virtual]`

Called when a line is selected.

Parameters

userTurn	Whether it is the user's turn

Returns

Whether it is still the player's turn

Implemented in HorizontalLine, and VerticalLine.

The documentation for this class was generated from the following files:

- game3/line.h
- game3/line.cpp

7.21 MainWidget Class Reference

Public Slots

- void goToGameSelection ()

  Slot that closes widget and opens the game selection menu.
- void signIn ()

  Signs in.
- void signUp ()

  Signs up.

Public Member Functions

- MainWidget (QWidget ∗parent=0)

  Default constructor.
- virtual ∼MainWidget ()

  Destructor.

7.21.1 Constructor & Destructor Documentation

7.21.1.1 MainWidget::MainWidget (QWidget ∗ *parent* = `0`) `[explicit]`

Default constructor.

Initializes all buttons, input fields and labels and shows them on the screen.

7.21.1.2 MainWidget::∼MainWidget ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.21.2 Member Function Documentation

7.21.2.1 void MainWidget::goToGameSelection ( ) `[slot]`

Slot that closes widget and opens the game selection menu.

Goes to Game Selection menu.

7.21.2.2 void MainWidget::signIn ( ) `[slot]`

Signs in.

Signs in by retrieving the account with the given username and password

7.21.2.3 void MainWidget::signUp ( ) [slot]

Signs up.

Signs up by adding an entry to the database

The documentation for this class was generated from the following files:

- gui/mainwidget.h
- gui/mainwidget.cpp

7.22 MyAccount Class Reference

Public Slots

- void goToGames ()
  Goes back to the games selection menu.
- void showGameGraph (int gamenb)
  Displays the graph of the selected game.

Public Member Functions

- MyAccount (QWidget ∗parent=0)
  Default constructor.
- virtual ∼MyAccount ()
  Destructor.

7.22.1 Constructor & Destructor Documentation

7.22.1.1 MyAccount::MyAccount (QWidget ∗ *parent* = 0) [explicit]

Default constructor.

Initializes all buttons and labels and shows them on the game selection menu.

7.22.1.2 MyAccount::∼MyAccount ( ) [virtual]

Destructor.

Frees allocated memory.

7.22.2 Member Function Documentation

7.22.2.1 void MyAccount::goToGames ( ) [slot]

Goes back to the games selection menu.

Takes the user back to the game selection menu. Called when the user clicks the corresponding button.

7.22.2.2 void **MyAccount::showGameGraph** (int *gamenb*) [slot]

Displays the graph of the selected game.

Parameters

gamenb	The number of the game

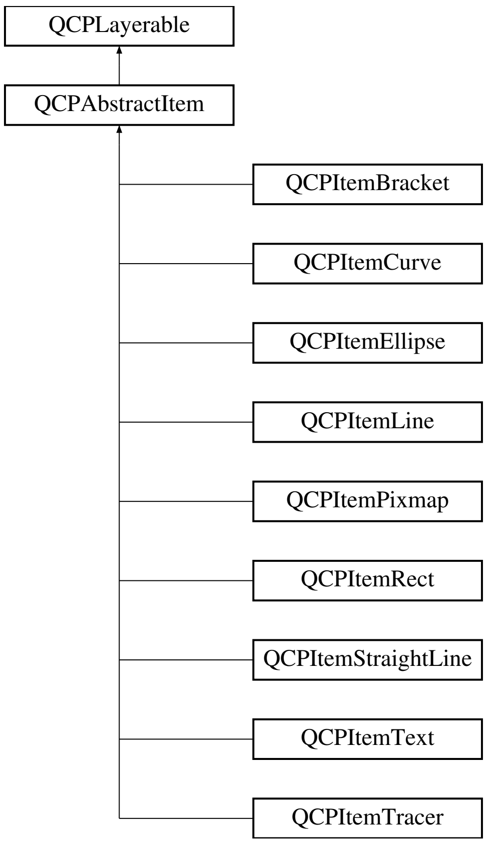
The documentation for this class was generated from the following files:

- account/myaccount.h
- account/myaccount.cpp

7.23 QCPAbstractItem Class Reference

The abstract base class for all items in a plot.

Inheritance diagram for QCPAbstractItem:



Signals

- void selectionChanged (bool selected)
- void **selectableChanged** (bool selectable)

Public Member Functions

- QCPAbstractItem (QCustomPlot ∗parentPlot)
- bool **clipToAxisRect** () const
- QCPAxisRect ∗ **clipAxisRect** () const
- bool **selectable** () const
- bool **selected** () const
- void setClipToAxisRect (bool clip)
- void setClipAxisRect (QCPAxisRect ∗rect)
- Q_SLOT void setSelectable (bool selectable)
- Q_SLOT void setSelected (bool selected)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const =0
- QList< QCPItemPosition ∗ > positions () const

- QList< QCPItemAnchor ∗ > anchors () const
- QCPItemPosition ∗ position (const QString &name) const
- QCPItemAnchor ∗ anchor (const QString &name) const
- bool hasAnchor (const QString &name) const

Protected Member Functions

- virtual QCP::Interaction **selectionCategory** () const
- virtual QRect **clipRect** () const
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)=0
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- double **distSqrToLine** (const QPointF &start, const QPointF &end, const QPointF &point) const
- double **rectSelectTest** (const QRectF &rect, const QPointF &pos, bool filledRect) const
- QCPItemPosition ∗ **createPosition** (const QString &name)
- QCPItemAnchor ∗ **createAnchor** (const QString &name, int anchorId)

Protected Attributes

- bool **mClipToAxisRect**
- QPointer< QCPAxisRect > **mClipAxisRect**
- QList< QCPItemPosition ∗ > **mPositions**
- QList< QCPItemAnchor ∗ > **mAnchors**
- bool **mSelectable**
- bool **mSelected**

Friends

- class **QCustomPlot**
- class **QCPItemAnchor**

7.23.1 Detailed Description

The abstract base class for all items in a plot.

In QCustomPlot, items are supplemental graphical elements that are neither plottables (QCPAbstractPlottable) nor axes (QCPAxis). While plottables are always tied to two axes and thus plot coordinates, items can also be placed in absolute coordinates independent of any axes. Each specific item has at least one QCPItemPosition member which controls the positioning. Some items are defined by more than one coordinate and thus have two or more QCPItemPosition members (For example, QCPItemRect has *topLeft* and *bottomRight*).

This abstract base class defines a very basic interface like visibility and clipping. Since this class is abstract, it can't be instantiated. Use one of the subclasses or create a subclass yourself to create new items.

The built-in items are:

QCPItemLine	A line defined by a start and an end point. May have different ending styles on each side (e.g. arrows).
QCPItemStraightLine	A straight line defined by a start and a direction point. Unlike QCPItemLine, the straight line is infinitely long and has no endings.
QCPItemCurve	A curve defined by start, end and two intermediate control points. May have different ending styles on each side (e.g. arrows).
QCPItemRect	A rectangle
QCPItemEllipse	An ellipse
QCPItemPixmap	An arbitrary pixmap
QCPItemText	A text label
QCPItemBracket	A bracket which may be used to reference/highlight certain parts in the plot.
QCPItemTracer	An item that can be attached to a QCPGraph and sticks to its data points, given a key coordinate.

7.23.2 Clipping

Items are by default clipped to the main axis rect (they are only visible inside the axis rect). To make an item visible outside that axis rect, disable clipping via setClipToAxisRect(false).

On the other hand if you want the item to be clipped to a different axis rect, specify it via setClipAxisRect. This clipAxisRect property of an item is only used for clipping behaviour, and in principle is independent of the coordinate axes the item might be tied to via its position members (QCPItemPosition::setAxes). However, it is common that the axis rect for clipping also contains the axes used for the item positions.

7.23.3 Using items

First you instantiate the item you want to use and add it to the plot:

by default, the positions of the item are bound to the x- and y-Axis of the plot. So we can just set the plot coordinates where the line should start/end:

If we don't want the line to be positioned in plot coordinates but a different coordinate system, e.g. absolute pixel positions on the QCustomPlot surface, we need to change the position type like this:

Then we can set the coordinates, this time in pixels:

and make the line visible on the entire QCustomPlot, by disabling clipping to the axis rect:

For more advanced plots, it is even possible to set different types and parent anchors per X/Y coordinate of an item position, using for example QCPItemPosition::setTypeX or QCPItemPosition::setParentAnchorX. For details, see the documentation of QCPItem-Position.

7.23.4 Creating own items

To create an own item, you implement a subclass of QCPAbstractItem. These are the pure virtual functions, you must implement:

- selectTest
- draw

See the documentation of those functions for what they need to do.

7.23.4.1 Allowing the item to be positioned

As mentioned, item positions are represented by QCPItemPosition members. Let's assume the new item shall have only one point as its position (as opposed to two like a rect or multiple like a polygon). You then add a public member of type QCPItemPosition like so:

```
QCPItemPosition * const myPosition;
```

the const makes sure the pointer itself can't be modified from the user of your new item (the QCPItemPosition instance it points to, can be modified, of course). The initialization of this pointer is made easy with the createPosition function. Just assign the return value of this function to each QCPItemPosition in the constructor of your item. createPosition takes a string which is the name of the position, typically this is identical to the variable name. For example, the constructor of QCPItemExample could look like this:

```
QCPItemExample::QCPItemExample(QCustomPlot *parentPlot) :
  QCPAbstractItem(parentPlot),
  myPosition(createPosition("myPosition"))
{
  // other constructor code
}
```

7.23.4.2 The draw function

To give your item a visual representation, reimplement the draw function and use the passed QCPPainter to draw the item. You can retrieve the item position in pixel coordinates from the position member(s) via QCPItemPosition::pixelPoint.

To optimize performance you should calculate a bounding rect first (don't forget to take the pen width into account), check whether it intersects the clipRect, and only draw the item at all if this is the case.

7.23.4.3 The selectTest function

Your implementation of the selectTest function may use the helpers distSqrToLine and rectSelectTest. With these, the implementation of the selection test becomes significantly simpler for most items. See the documentation of selectTest for what the function parameters mean and what the function should return.

7.23.4.4 Providing anchors

Providing anchors (QCPItemAnchor) starts off like adding a position. First you create a public member, e.g.

```
QCPItemAnchor * const bottom;
```

and create it in the constructor with the createAnchor function, assigning it a name and an anchor id (an integer enumerating all anchors on the item, you may create an own enum for this). Since anchors can be placed anywhere, relative to the item's position(s), your item needs to provide the position of every anchor with the reimplementation of the anchorPixelPoint(int anchorId) function.

In essence the QCPItemAnchor is merely an intermediary that itself asks your item for the pixel position when anything attached to the anchor needs to know the coordinates.

7.23.5 Constructor & Destructor Documentation

7.23.5.1 QCPAbstractItem::QCPAbstractItem (QCustomPlot * *parentPlot*)

Base class constructor which initializes base class members.

7.23.6 Member Function Documentation

7.23.6.1 QCPItemAnchor ∗ QCPAbstractItem::anchor (const QString & *name*) const

Returns the QCPItemAnchor with the specified *name*. If this item doesn't have an anchor by that name, returns 0.

This function provides an alternative way to access item anchors. Normally, you access anchors direcly by their member pointers (which typically have the same variable name as *name*).

See also

anchors, position

7.23.6.2 QList< QCPItemAnchor ∗ > QCPAbstractItem::anchors ( ) const [inline]

Returns all anchors of the item in a list. Note that since a position (QCPItemPosition) is always also an anchor, the list will also contain the positions of this item.

See also

positions, anchor

7.23.6.3 bool QCPAbstractItem::hasAnchor (const QString & *name*) const

Returns whether this item has an anchor with the specified *name*.

Note that you can check for positions with this function, too. This is because every position is also an anchor (QCPItemPosition inherits from QCPItemAnchor).

See also

anchor, position

7.23.6.4 QCPItemPosition ∗ QCPAbstractItem::position (const QString & *name*) const

Returns the QCPItemPosition with the specified *name*. If this item doesn't have a position by that name, returns 0.

This function provides an alternative way to access item positions. Normally, you access positions direcly by their member pointers (which typically have the same variable name as *name*).

See also

positions, anchor

7.23.6.5 QList< QCPItemPosition ∗ > QCPAbstractItem::positions ( ) const `[inline]`

Returns all positions of the item in a list.

See also

anchors, position

7.23.6.6 void QCPAbstractItem::selectionChanged (bool *selected*) `[signal]`

This signal is emitted when the selection state of this item has changed, either by user interaction or by a direct call to setSelected.

7.23.6.7 virtual double QCPAbstractItem::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = 0) const `[pure virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Reimplemented from QCPLayerable.

Implemented in QCPItemBracket, QCPItemTracer, QCPItemPixmap, QCPItemEllipse, QCPItemText, QCPItemRect, QCPItemCurve, QCPItemLine, and QCPItemStraight-Line.

7.23.6.8 void QCPAbstractItem::setClipAxisRect (QCPAxisRect ∗ *rect*)

Sets the clip axis rect. It defines the rect that will be used to clip the item when setClip-ToAxisRect is set to true.

See also

setClipToAxisRect

7.23.6.9 void QCPAbstractItem::setClipToAxisRect (bool *clip*)

Sets whether the item shall be clipped to an axis rect or whether it shall be visible on the entire QCustomPlot. The axis rect can be set with setClipAxisRect.

See also

setClipAxisRect

7.23.6.10 void QCPAbstractItem::setSelectable (bool *selectable*)

Sets whether the user can (de-)select this item by clicking on the QCustomPlot surface. (When QCustomPlot::setInteractions contains QCustomPlot::iSelectItems.)

However, even when *selectable* was set to false, it is possible to set the selection manually, by calling setSelected.

See also

QCustomPlot::setInteractions, setSelected

7.23.6.11 void QCPAbstractItem::setSelected (bool *selected*)

Sets whether this item is selected or not. When selected, it might use a different visual appearance (e.g. pen and brush), this depends on the specific item though.

The entire selection mechanism for items is handled automatically when QCustomPlot-::setInteractions contains QCustomPlot::iSelectItems. You only need to call this function when you wish to change the selection state manually.

This function can change the selection state even when setSelectable was set to false.

emits the selectionChanged signal when *selected* is different from the previous selection state.

See also

setSelectable, selectTest

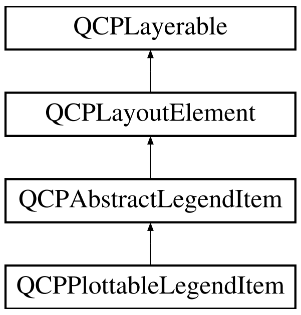
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.24 QCPAbstractLegendItem Class Reference

The abstract base class for all entries in a QCPLegend.

Inheritance diagram for QCPAbstractLegendItem:



Signals

- void selectionChanged (bool selected)
- void **selectableChanged** (bool selectable)

Public Member Functions

- QCPAbstractLegendItem (QCPLegend ∗parent)
- QCPLegend ∗ **parentLegend** () const
- QFont **font** () const
- QColor **textColor** () const
- QFont **selectedFont** () const
- QColor **selectedTextColor** () const
- bool **selectable** () const

- bool **selected** () const
- void setFont (const QFont &font)
- void setTextColor (const QColor &color)
- void setSelectedFont (const QFont &font)
- void setSelectedTextColor (const QColor &color)
- Q_SLOT void setSelectable (bool selectable)
- Q_SLOT void setSelected (bool selected)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual QCP::Interaction **selectionCategory** () const
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual QRect **clipRect** () const
- virtual void **draw** (QCPPainter ∗painter)=0
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)

Protected Attributes

- QCPLegend ∗ **mParentLegend**
- QFont **mFont**
- QColor **mTextColor**
- QFont **mSelectedFont**
- QColor **mSelectedTextColor**
- bool **mSelectable**
- bool **mSelected**

Friends

- class **QCPLegend**

7.24.1 Detailed Description

The abstract base class for all entries in a QCPLegend.

It defines a very basic interface for entries in a QCPLegend. For representing plottables in the legend, the subclass QCPPlottableLegendItem is more suitable.

Only derive directly from this class when you need absolute freedom (e.g. a custom legend entry that's not even associated with a plottable).

You must implement the following pure virtual functions:

- draw (from QCPLayerable)

You inherit the following members you may use:

QCPLegend ∗**mParentLegend**	A pointer to the parent QCPLegend.
QFont **mFont**	The generic font of the item. You should use this font for all or at least the most prominent text of the item.

7.24.2 Constructor & Destructor Documentation

7.24.2.1 QCPAbstractLegendItem::QCPAbstractLegendItem (QCPLegend ∗ *parent*) [explicit]

Constructs a QCPAbstractLegendItem and associates it with the QCPLegend *parent*. This does not cause the item to be added to *parent*, so QCPLegend::addItem must be called separately.

7.24.3 Member Function Documentation

7.24.3.1 void QCPAbstractLegendItem::selectionChanged (bool *selected*) [signal]

This signal is emitted when the selection state of this legend item has changed, either by user interaction or by a direct call to setSelected.

7.24.3.2 double QCPAbstractLegendItem::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

Layout elements are sensitive to events inside their outer rect. If *pos* is within the outer rect, this method returns a value corresponding to 0.99 times the parent plot's selection tolerance. However, layout elements are not selectable by default. So if *onlySelectable* is true, -1.0 is returned.

See QCPLayerable::selectTest for a general explanation of this virtual method.

QCPLayoutElement subclasses may reimplement this method to provide more specific selection test behaviour.

Reimplemented from QCPLayoutElement.

7.24.3.3 void QCPAbstractLegendItem::setFont (const QFont & *font*)

Sets the default font of this specific legend item to *font*.

See also

setTextColor, QCPLegend::setFont

7.24.3.4 void **QCPAbstractLegendItem::setSelectable** (bool *selectable*)

Sets whether this specific legend item is selectable.

See also

setSelectedParts, QCustomPlot::setInteractions

7.24.3.5 void **QCPAbstractLegendItem::setSelected** (bool *selected*)

Sets whether this specific legend item is selected.

It is possible to set the selection state of this item by calling this function directly, even if setSelectable is set to false.

See also

setSelectableParts, QCustomPlot::setInteractions

7.24.3.6 void **QCPAbstractLegendItem::setSelectedFont** (const QFont & *font*)

When this legend item is selected, *font* is used to draw generic text, instead of the normal font set with setFont.

See also

setFont, QCPLegend::setSelectedFont

7.24.3.7 void **QCPAbstractLegendItem::setSelectedTextColor** (const QColor & *color*)

When this legend item is selected, *color* is used to draw generic text, instead of the normal color set with setTextColor.

See also

setTextColor, QCPLegend::setSelectedTextColor

7.24.3.8 void **QCPAbstractLegendItem::setTextColor** (const QColor & *color*)

Sets the default text color of this specific legend item to *color*.

See also

setFont, QCPLegend::setTextColor

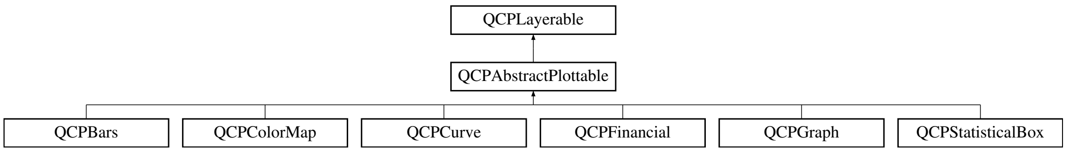
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.25 QCPAbstractPlottable Class Reference

The abstract base class for all data representing objects in a plot.

Inheritance diagram for QCPAbstractPlottable:



Signals

- void selectionChanged (bool selected)
- void selectableChanged (bool selectable)

Public Member Functions

- QCPAbstractPlottable (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- QString **name** () const
- bool **antialiasedFill** () const
- bool **antialiasedScatters** () const
- bool **antialiasedErrorBars** () const
- QPen **pen** () const
- QPen **selectedPen** () const
- QBrush **brush** () const
- QBrush **selectedBrush** () const
- QCPAxis ∗ **keyAxis** () const
- QCPAxis ∗ **valueAxis** () const
- bool **selectable** () const
- bool **selected** () const
- void setName (const QString &name)
- void setAntialiasedFill (bool enabled)
- void setAntialiasedScatters (bool enabled)
- void setAntialiasedErrorBars (bool enabled)
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setSelectedBrush (const QBrush &brush)
- void setKeyAxis (QCPAxis ∗axis)
- void setValueAxis (QCPAxis ∗axis)
- Q_SLOT void setSelectable (bool selectable)
- Q_SLOT void setSelected (bool selected)
- virtual void clearData ()=0

- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const =0
- virtual bool addToLegend ()
- virtual bool removeFromLegend () const
- void rescaleAxes (bool onlyEnlarge=false) const
- void rescaleKeyAxis (bool onlyEnlarge=false) const
- void rescaleValueAxis (bool onlyEnlarge=false) const

Protected Types

- enum SignDomain { sdNegative, sdBoth, sdPositive }

Protected Member Functions

- virtual QRect **clipRect** () const
- virtual void **draw** (QCPPainter ∗painter)=0
- virtual QCP::Interaction **selectionCategory** () const
- void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const =0
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const =0
- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const =0
- void **coordsToPixels** (double key, double value, double &x, double &y) const
- const QPointF **coordsToPixels** (double key, double value) const
- void **pixelsToCoords** (double x, double y, double &key, double &value) const
- void **pixelsToCoords** (const QPointF &pixelPos, double &key, double &value) const
- QPen **mainPen** () const
- QBrush **mainBrush** () const
- void **applyFillAntialiasingHint** (QCPPainter ∗painter) const
- void **applyScattersAntialiasingHint** (QCPPainter ∗painter) const
- void **applyErrorBarsAntialiasingHint** (QCPPainter ∗painter) const
- double **distSqrToLine** (const QPointF &start, const QPointF &end, const QPointF &point) const

Protected Attributes

- QString **mName**
- bool **mAntialiasedFill**
- bool **mAntialiasedScatters**

- bool **mAntialiasedErrorBars**
- QPen **mPen**
- QPen **mSelectedPen**
- QBrush **mBrush**
- QBrush **mSelectedBrush**
- QPointer< QCPAxis > **mKeyAxis**
- QPointer< QCPAxis > **mValueAxis**
- bool **mSelectable**
- bool **mSelected**

Friends

- class **QCustomPlot**
- class **QCPAxis**
- class **QCPPlottableLegendItem**

7.25.1 Detailed Description

The abstract base class for all data representing objects in a plot.

It defines a very basic interface like name, pen, brush, visibility etc. Since this class is abstract, it can't be instantiated. Use one of the subclasses or create a subclass yourself to create new ways of displaying data (see "Creating own plottables" below).

All further specifics are in the subclasses, for example:

- A normal graph with possibly a line, scatter points and error bars: QCPGraph (typically created with QCustomPlot::addGraph)
- A parametric curve: QCPCurve
- A bar chart: QCPBars
- A statistical box plot: QCPStatisticalBox
- A color encoded two-dimensional map: QCPColorMap
- An OHLC/Candlestick chart: QCPFinancial

7.25.2 Creating own plottables

To create an own plottable, you implement a subclass of QCPAbstractPlottable. These are the pure virtual functions, you must implement:

- clearData
- selectTest
- draw

- drawLegendIcon
- getKeyRange
- getValueRange

See the documentation of those functions for what they need to do.

For drawing your plot, you can use the coordsToPixels functions to translate a point in plot coordinates to pixel coordinates. This function is quite convenient, because it takes the orientation of the key and value axes into account for you (x and y are swapped when the key axis is vertical and the value axis horizontal). If you are worried about performance (i.e. you need to translate many points in a loop like QCPGraph), you can directly use QCPAxis::coordToPixel. However, you must then take care about the orientation of the axis yourself.

Here are some important members you inherit from QCPAbstractPlottable:

QCustomPlot ∗**mParentPlot**	A pointer to the parent QCustomPlot instance. The parent plot is inferred from the axes that are passed in the constructor.
QString **mName**	The name of the plottable.
QPen **mPen**	The generic pen of the plottable. You should use this pen for the most prominent data representing lines in the plottable (e.g QCPGraph uses this pen for its graph lines and scatters)
QPen **mSelectedPen**	The generic pen that should be used when the plottable is selected (hint: mainPen gives you the right pen, depending on selection state).
QBrush **mBrush**	The generic brush of the plottable. You should use this brush for the most prominent fillable structures in the plottable (e.g. QCPGraph uses this brush to control filling under the graph)
QBrush **mSelectedBrush**	The generic brush that should be used when the plottable is selected (hint: mainBrush gives you the right brush, depending on selection state).
QPointer<QCPAxis>**mKeyAxis**, **mValueAxis**	The key and value axes this plottable is attached to. Call their QCPAxis::coordToPixel functions to translate coordinates to pixels in either the key or value dimension. Make sure to check whether the pointer is null before using it. If one of the axes is null, don't draw the plottable.
bool **mSelected**	indicates whether the plottable is selected or not.

7.25.3 Member Enumeration Documentation

7.25.3.1 enum **QCPAbstractPlottable::SignDomain** `[protected]`

Represents negative and positive sign domain for passing to getKeyRange and getValueRange.

Enumerator:

sdNegative The negative sign domain, i.e. numbers smaller than zero.

sdBoth Both sign domains, including zero, i.e. all (rational) numbers.

sdPositive The positive sign domain, i.e. numbers greater than zero.

7.25.4 Constructor & Destructor Documentation

7.25.4.1 **QCPAbstractPlottable::QCPAbstractPlottable (QCPAxis** ∗ ***keyAxis,*** **QCPAxis** ∗ ***valueAxis*** **)**

Constructs an abstract plottable which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and have perpendicular orientations. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

Since QCPAbstractPlottable is an abstract class that defines the basic interface to plottables, it can't be directly instantiated.

You probably want one of the subclasses like QCPGraph or QCPCurve instead.

7.25.5 Member Function Documentation

7.25.5.1 bool **QCPAbstractPlottable::addToLegend ( )** `[virtual]`

Adds this plottable to the legend of the parent QCustomPlot (QCustomPlot::legend).

Normally, a QCPPlottableLegendItem is created and inserted into the legend. If the plottable needs a more specialized representation in the legend, this function will take this into account and instead create the specialized subclass of QCPAbstractLegendItem.

Returns true on success, i.e. when the legend exists and a legend item associated with this plottable isn't already in the legend.

See also

removeFromLegend, QCPLegend::addItem

7.25.5.2 void QCPAbstractPlottable::clearData ( ) [pure virtual]

Clears all data in the plottable.

Implemented in QCPFinancial, QCPColorMap, QCPStatisticalBox, QCPBars, QCP-Curve, and QCPGraph.

7.25.5.3 bool QCPAbstractPlottable::removeFromLegend ( ) const [virtual]

Removes the plottable from the legend of the parent QCustomPlot. This means the QCPAbstractLegendItem (usually a QCPPlottableLegendItem) that is associated with this plottable is removed.

Returns true on success, i.e. if the legend exists and a legend item associated with this plottable was found and removed.

See also

addToLegend, QCPLegend::removeItem

7.25.5.4 void QCPAbstractPlottable::rescaleAxes (bool *onlyEnlarge* = `false`) const

Rescales the key and value axes associated with this plottable to contain all displayed data, so the whole plottable is visible. If the scaling of an axis is logarithmic, rescale-Axes will make sure not to rescale to an illegal range i.e. a range containing different signs and/or zero. Instead it will stay in the current sign domain and ignore all parts of the plottable that lie outside of that domain.

onlyEnlarge makes sure the ranges are only expanded, never reduced. So it's possible to show multiple plottables in their entirety by multiple calls to rescaleAxes where the first call has *onlyEnlarge* set to false (the default), and all subsequent set to true.

See also

rescaleKeyAxis, rescaleValueAxis, QCustomPlot::rescaleAxes, QCPAxis::rescale

7.25.5.5 void QCPAbstractPlottable::rescaleKeyAxis (bool *onlyEnlarge* = `false`) const

Rescales the key axis of the plottable so the whole plottable is visible.

See rescaleAxes for detailed behaviour.

7.25.5.6 void QCPAbstractPlottable::rescaleValueAxis (bool *onlyEnlarge* = `false`) const

Rescales the value axis of the plottable so the whole plottable is visible.

Returns true if the axis was actually scaled. This might not be the case if this plottable has an invalid range, e.g. because it has no data points.

See rescaleAxes for detailed behaviour.

7.25.5.7 void QCPAbstractPlottable::selectableChanged (bool *selectable*) [signal]

This signal is emitted when the selectability of this plottable has changed.

See also

setSelectable

7.25.5.8 void QCPAbstractPlottable::selectionChanged (bool *selected*) [signal]

This signal is emitted when the selection state of this plottable has changed, either by user interaction or by a direct call to setSelected.

7.25.5.9 virtual double QCPAbstractPlottable::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [pure virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Reimplemented from QCPLayerable.

Implemented in QCPFinancial, QCPColorMap, QCPStatisticalBox, QCPBars, QCPCurve, and QCPGraph.

7.25.5.10 void **QCPAbstractPlottable::setAntialiasedErrorBars** (bool *enabled*)

Sets whether the error bars of this plottable are drawn antialiased or not.

Note that this setting may be overridden by QCustomPlot::setAntialiasedElements and QCustomPlot::setNotAntialiasedElements.

7.25.5.11 void **QCPAbstractPlottable::setAntialiasedFill** (bool *enabled*)

Sets whether fills of this plottable are drawn antialiased or not.

Note that this setting may be overridden by QCustomPlot::setAntialiasedElements and QCustomPlot::setNotAntialiasedElements.

7.25.5.12 void **QCPAbstractPlottable::setAntialiasedScatters** (bool *enabled*)

Sets whether the scatter symbols of this plottable are drawn antialiased or not.

Note that this setting may be overridden by QCustomPlot::setAntialiasedElements and QCustomPlot::setNotAntialiasedElements.

7.25.5.13 void **QCPAbstractPlottable::setBrush** (const QBrush & *brush*)

The brush is used to draw basic fills of the plottable representation in the plot. The Fill can be a color, gradient or texture, see the usage of QBrush.

For example, the QCPGraph subclass draws the fill under the graph with this brush, when it's not set to Qt::NoBrush.

See also

setPen

7.25.5.14 void QCPAbstractPlottable::setKeyAxis (QCPAxis ∗ *axis*)

The key axis of a plottable can be set to any axis of a QCustomPlot, as long as it is orthogonal to the plottable's value axis. This function performs no checks to make sure this is the case. The typical mathematical choice is to use the x-axis (QCustomPlot::x-Axis) as key axis and the y-axis (QCustomPlot::yAxis) as value axis.

Normally, the key and value axes are set in the constructor of the plottable (or QCustom-Plot::addGraph when working with QCPGraphs through the dedicated graph interface).

See also

setValueAxis

7.25.5.15 void QCPAbstractPlottable::setName (const QString & *name*)

The name is the textual representation of this plottable as it is displayed in the legend (QCPLegend). It may contain any UTF-8 characters, including newlines.

7.25.5.16 void QCPAbstractPlottable::setPen (const QPen & *pen*)

The pen is used to draw basic lines that make up the plottable representation in the plot.

For example, the QCPGraph subclass draws its graph lines with this pen.

See also

setBrush

7.25.5.17 void QCPAbstractPlottable::setSelectable (bool *selectable*)

Sets whether the user can (de-)select this plottable by clicking on the QCustomPlot surface. (When QCustomPlot::setInteractions contains iSelectPlottables.)

However, even when *selectable* was set to false, it is possible to set the selection manually, by calling setSelected directly.

See also

setSelected

7.25.5.18 void QCPAbstractPlottable::setSelected (bool *selected*)

Sets whether this plottable is selected or not. When selected, it uses a different pen and brush to draw its lines and fills, see setSelectedPen and setSelectedBrush.

The entire selection mechanism for plottables is handled automatically when QCustom-Plot::setInteractions contains iSelectPlottables. You only need to call this function when you wish to change the selection state manually.

This function can change the selection state even when setSelectable was set to false.

emits the selectionChanged signal when *selected* is different from the previous selection state.

See also

setSelectable, selectTest

7.25.5.19 void QCPAbstractPlottable::setSelectedBrush (const QBrush & *brush*)

When the plottable is selected, this brush is used to draw fills instead of the normal brush set via setBrush.

See also

setSelected, setSelectable, setSelectedPen, selectTest

7.25.5.20 void QCPAbstractPlottable::setSelectedPen (const QPen & *pen*)

When the plottable is selected, this pen is used to draw basic lines instead of the normal pen set via setPen.

See also

setSelected, setSelectable, setSelectedBrush, selectTest

7.25.5.21 void QCPAbstractPlottable::setValueAxis (QCPAxis ∗ *axis*)

The value axis of a plottable can be set to any axis of a QCustomPlot, as long as it is orthogonal to the plottable's key axis. This function performs no checks to make sure this is the case. The typical mathematical choice is to use the x-axis (QCustomPlot::x-Axis) as key axis and the y-axis (QCustomPlot::yAxis) as value axis.

Normally, the key and value axes are set in the constructor of the plottable (or QCustom-Plot::addGraph when working with QCPGraphs through the dedicated graph interface).

See also

setKeyAxis

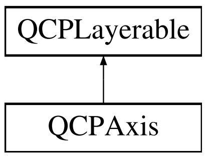
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.26 QCPAxis Class Reference

Manages a single axis inside a QCustomPlot.

Inheritance diagram for QCPAxis:



Public Types

- enum AxisType { atLeft = 0x01, atRight = 0x02, atTop = 0x04, atBottom = 0x08 }
- enum LabelType { ltNumber, ltDateTime }
- enum LabelSide { lsInside, lsOutside }
- enum ScaleType { stLinear, stLogarithmic }
- enum SelectablePart { spNone = 0, spAxis = 0x001, spTickLabels = 0x002, spAxisLabel = 0x004 }

Signals

- void ticksRequest ()
- void rangeChanged (const QCPRange &newRange)
- void rangeChanged (const QCPRange &newRange, const QCPRange &oldRange)
- void scaleTypeChanged (QCPAxis::ScaleType scaleType)
- void selectionChanged (const QCPAxis::SelectableParts &parts)
- void selectableChanged (const QCPAxis::SelectableParts &parts)

Public Member Functions

- QCPAxis (QCPAxisRect ∗parent, AxisType type)
- AxisType **axisType** () const
- QCPAxisRect ∗ **axisRect** () const
- ScaleType **scaleType** () const
- double **scaleLogBase** () const
- const QCPRange **range** () const
- bool **rangeReversed** () const
- bool **autoTicks** () const
- int **autoTickCount** () const
- bool **autoTickLabels** () const
- bool **autoTickStep** () const
- bool **autoSubTicks** () const

- bool **ticks** () const
- bool **tickLabels** () const
- int **tickLabelPadding** () const
- LabelType **tickLabelType** () const
- QFont **tickLabelFont** () const
- QColor **tickLabelColor** () const
- double **tickLabelRotation** () const
- LabelSide **tickLabelSide** () const
- QString **dateTimeFormat** () const
- Qt::TimeSpec **dateTimeSpec** () const
- QString **numberFormat** () const
- int **numberPrecision** () const
- double **tickStep** () const
- QVector< double > **tickVector** () const
- QVector< QString > **tickVectorLabels** () const
- int **tickLengthIn** () const
- int **tickLengthOut** () const
- int **subTickCount** () const
- int **subTickLengthIn** () const
- int **subTickLengthOut** () const
- QPen **basePen** () const
- QPen **tickPen** () const
- QPen **subTickPen** () const
- QFont **labelFont** () const
- QColor **labelColor** () const
- QString **label** () const
- int **labelPadding** () const
- int **padding** () const
- int **offset** () const
- SelectableParts **selectedParts** () const
- SelectableParts **selectableParts** () const
- QFont **selectedTickLabelFont** () const
- QFont **selectedLabelFont** () const
- QColor **selectedTickLabelColor** () const
- QColor **selectedLabelColor** () const
- QPen **selectedBasePen** () const
- QPen **selectedTickPen** () const
- QPen **selectedSubTickPen** () const
- QCPLineEnding **lowerEnding** () const
- QCPLineEnding **upperEnding** () const
- QCPGrid ∗ grid () const
- Q_SLOT void setScaleType (QCPAxis::ScaleType type)
- void setScaleLogBase (double base)
- Q_SLOT void setRange (const QCPRange &range)
- void setRange (double lower, double upper)
- void setRange (double position, double size, Qt::AlignmentFlag alignment)

- void setRangeLower (double lower)
- void setRangeUpper (double upper)
- void setRangeReversed (bool reversed)
- void setAutoTicks (bool on)
- void setAutoTickCount (int approximateCount)
- void setAutoTickLabels (bool on)
- void setAutoTickStep (bool on)
- void setAutoSubTicks (bool on)
- void setTicks (bool show)
- void setTickLabels (bool show)
- void setTickLabelPadding (int padding)
- void setTickLabelType (LabelType type)
- void setTickLabelFont (const QFont &font)
- void setTickLabelColor (const QColor &color)
- void setTickLabelRotation (double degrees)
- void setTickLabelSide (LabelSide side)
- void setDateTimeFormat (const QString &format)
- void setDateTimeSpec (const Qt::TimeSpec &timeSpec)
- void setNumberFormat (const QString &formatCode)
- void setNumberPrecision (int precision)
- void setTickStep (double step)
- void setTickVector (const QVector< double > &vec)
- void setTickVectorLabels (const QVector< QString > &vec)
- void setTickLength (int inside, int outside=0)
- void setTickLengthIn (int inside)
- void setTickLengthOut (int outside)
- void setSubTickCount (int count)
- void setSubTickLength (int inside, int outside=0)
- void setSubTickLengthIn (int inside)
- void setSubTickLengthOut (int outside)
- void setBasePen (const QPen &pen)
- void setTickPen (const QPen &pen)
- void setSubTickPen (const QPen &pen)
- void setLabelFont (const QFont &font)
- void setLabelColor (const QColor &color)
- void setLabel (const QString &str)
- void setLabelPadding (int padding)
- void setPadding (int padding)
- void setOffset (int offset)
- void setSelectedTickLabelFont (const QFont &font)
- void setSelectedLabelFont (const QFont &font)
- void setSelectedTickLabelColor (const QColor &color)
- void setSelectedLabelColor (const QColor &color)
- void setSelectedBasePen (const QPen &pen)
- void setSelectedTickPen (const QPen &pen)
- void setSelectedSubTickPen (const QPen &pen)

- Q_SLOT void setSelectableParts (const QCPAxis::SelectableParts &selectable-Parts)
- Q_SLOT void setSelectedParts (const QCPAxis::SelectableParts &selected-Parts)
- void setLowerEnding (const QCPLineEnding &ending)
- void setUpperEnding (const QCPLineEnding &ending)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- Qt::Orientation orientation () const
- void moveRange (double diff)
- void scaleRange (double factor, double center)
- void setScaleRatio (const QCPAxis ∗otherAxis, double ratio=1.0)
- void rescale (bool onlyVisiblePlottables=false)
- double pixelToCoord (double value) const
- double coordToPixel (double value) const
- SelectablePart getPartAt (const QPointF &pos) const
- QList< QCPAbstractPlottable ∗ > plottables () const
- QList< QCPGraph ∗ > graphs () const
- QList< QCPAbstractItem ∗ > items () const

Static Public Member Functions

- static AxisType marginSideToAxisType (QCP::MarginSide side)
- static Qt::Orientation orientation (AxisType type)
- static AxisType opposite (AxisType type)

Protected Member Functions

- virtual void **setupTickVectors** ()
- virtual void **generateAutoTicks** ()
- virtual int **calculateAutoSubTickCount** (double tickStep) const
- virtual int **calculateMargin** ()
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- virtual QCP::Interaction **selectionCategory** () const
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- void **visibleTickBounds** (int &lowIndex, int &highIndex) const
- double **baseLog** (double value) const
- double **basePow** (double value) const
- QPen **getBasePen** () const
- QPen **getTickPen** () const
- QPen **getSubTickPen** () const
- QFont **getTickLabelFont** () const
- QFont **getLabelFont** () const
- QColor **getTickLabelColor** () const
- QColor **getLabelColor** () const

Protected Attributes

- AxisType **mAxisType**
- QCPAxisRect ∗ **mAxisRect**
- int **mPadding**
- Qt::Orientation **mOrientation**
- SelectableParts **mSelectableParts**
- SelectableParts **mSelectedParts**
- QPen **mBasePen**
- QPen **mSelectedBasePen**
- QString **mLabel**
- QFont **mLabelFont**
- QFont **mSelectedLabelFont**
- QColor **mLabelColor**
- QColor **mSelectedLabelColor**
- bool **mTickLabels**
- bool **mAutoTickLabels**
- LabelType **mTickLabelType**
- QFont **mTickLabelFont**
- QFont **mSelectedTickLabelFont**
- QColor **mTickLabelColor**
- QColor **mSelectedTickLabelColor**
- QString **mDateTimeFormat**
- Qt::TimeSpec **mDateTimeSpec**
- int **mNumberPrecision**
- QLatin1Char **mNumberFormatChar**
- bool **mNumberBeautifulPowers**
- bool **mTicks**
- double **mTickStep**
- int **mSubTickCount**
- int **mAutoTickCount**
- bool **mAutoTicks**
- bool **mAutoTickStep**
- bool **mAutoSubTicks**
- QPen **mTickPen**
- QPen **mSelectedTickPen**
- QPen **mSubTickPen**
- QPen **mSelectedSubTickPen**
- QCPRange **mRange**
- bool **mRangeReversed**
- ScaleType **mScaleType**
- double **mScaleLogBase**
- double **mScaleLogBaseLogInv**
- QCPGrid ∗ **mGrid**
- QCPAxisPainterPrivate ∗ **mAxisPainter**
- int **mLowestVisibleTick**
- int **mHighestVisibleTick**

- QVector< double > **mTickVector**
- QVector< QString > **mTickVectorLabels**
- QVector< double > **mSubTickVector**
- bool **mCachedMarginValid**
- int **mCachedMargin**

Friends

- class **QCustomPlot**
- class **QCPGrid**
- class **QCPAxisRect**

7.26.1 Detailed Description

Manages a single axis inside a QCustomPlot.

Usually doesn't need to be instantiated externally. Access QCustomPlot's default four axes via QCustomPlot::xAxis (bottom), QCustomPlot::yAxis (left), QCustomPlot::xAxis2 (top) and QCustomPlot::yAxis2 (right).

Axes are always part of an axis rect, see QCPAxisRect.

Naming convention of axis parts

Overview of the spacings and paddings that define the geometry of an axis. The dashed gray line on the left represents the QCustomPlot widget border.

7.26.2 Member Enumeration Documentation

7.26.2.1 enum QCPAxis::AxisType

Defines at which side of the axis rect the axis will appear. This also affects how the tick marks are drawn, on which side the labels are placed etc.

Enumerator:

atLeft	`0x01` Axis is vertical and on the left side of the axis rect
atRight	`0x02` Axis is vertical and on the right side of the axis rect
atTop	`0x04` Axis is horizontal and on the top side of the axis rect
atBottom	`0x08` Axis is horizontal and on the bottom side of the axis rect

7.26.2.2 enum QCPAxis::LabelSide

Defines on which side of the axis the tick labels (numbers) shall appear.

See also

setTickLabelSide

Enumerator:

lsInside Tick labels will be displayed inside the axis rect and clipped to the inner axis rect.

lsOutside Tick labels will be displayed outside the axis rect.

7.26.2.3 enum **QCPAxis::LabelType**

When automatic tick label generation is enabled (setAutoTickLabels), defines how the coordinate of the tick is interpreted, i.e. translated into a string.

See also

setTickLabelType

Enumerator:

ltNumber Tick coordinate is regarded as normal number and will be displayed as such. (see setNumberFormat)

ltDateTime Tick coordinate is regarded as a date/time (seconds since 1970-01-01T00:00:00 UTC) and will be displayed and formatted as such. (for details, see setDateTimeFormat)

7.26.2.4 enum **QCPAxis::ScaleType**

Defines the scale of an axis.

See also

setScaleType

Enumerator:

stLinear Linear scaling.

stLogarithmic Logarithmic scaling with correspondingly transformed plots and (major) tick marks at every base power (see setScaleLogBase).

7.26.2.5 enum **QCPAxis::SelectablePart**

Defines the selectable parts of an axis.

See also

setSelectableParts, setSelectedParts

Enumerator:

spNone None of the selectable parts.

spAxis The axis backbone and tick marks.

spTickLabels Tick labels (numbers) of this axis (as a whole, not individually)

spAxisLabel The axis label.

7.26.3 Constructor & Destructor Documentation

7.26.3.1 QCPAxis::QCPAxis (QCPAxisRect ∗ *parent,* AxisType *type*) [explicit]

Constructs an Axis instance of Type *type* for the axis rect *parent*.

Usually it isn't necessary to instantiate axes directly, because you can let QCustomPlot create them for you with QCPAxisRect::addAxis. If you want to use own QCPAxis-subclasses however, create them manually and then inject them also via QCPAxisRect::addAxis.

7.26.4 Member Function Documentation

7.26.4.1 double QCPAxis::coordToPixel (double *value*) const

Transforms *value*, in coordinates of the axis, to pixel coordinates of the QCustomPlot widget.

7.26.4.2 QCPAxis::SelectablePart QCPAxis::getPartAt (const QPointF & *pos*) const

Returns the part of the axis that is hit by *pos* (in pixels). The return value of this function is independent of the user-selectable parts defined with setSelectableParts. Further, this function does not change the current selection state of the axis.

If the axis is not visible (setVisible), this function always returns spNone.

See also

setSelectedParts, setSelectableParts, QCustomPlot::setInteractions

7.26.4.3 QList< QCPGraph ∗ > QCPAxis::graphs ( ) const

Returns a list of all the graphs that have this axis as key or value axis.

See also

plottables, items

7.26.4.4 QCPGrid ∗ QCPAxis::grid ( ) const [inline]

Returns the QCPGrid instance belonging to this axis. Access it to set details about the way the grid is displayed.

7.26.4.5 QList< QCPAbstractItem ∗ > QCPAxis::items ( ) const

Returns a list of all the items that are associated with this axis. An item is considered associated with an axis if at least one of its positions uses the axis as key or value axis.

See also

plottables, graphs

7.26.4.6 QCPAxis::AxisType QCPAxis::marginSideToAxisType (QCP::MarginSide *side*) [static]

Transforms a margin side to the logically corresponding axis type. (QCP::msLeft to QCPAxis::atLeft, QCP::msRight to QCPAxis::atRight, etc.)

7.26.4.7 void QCPAxis::moveRange (double *diff*)

If the scale type (setScaleType) is stLinear, *diff* is added to the lower and upper bounds of the range. The range is simply moved by *diff*.

If the scale type is stLogarithmic, the range bounds are multiplied by *diff*. This corresponds to an apparent "linear" move in logarithmic scaling by a distance of log(diff).

7.26.4.8 QCPAxis::AxisType QCPAxis::opposite (QCPAxis::AxisType *type*) [static]

Returns the axis type that describes the opposite axis of an axis with the specified *type*.

7.26.4.9 Qt::Orientation QCPAxis::orientation ( ) const [inline]

Returns the orientation of this axis. The axis orientation (horizontal or vertical) is deduced from the axis type (left, top, right or bottom).

See also

orientation(AxisType type)

7.26.4.10 static Qt::Orientation QCPAxis::orientation (AxisType *type*) [inline, static]

Returns the orientation of the specified axis type

See also

orientation()

7.26.4.11 double QCPAxis::pixelToCoord (double *value*) const

Transforms *value*, in pixel coordinates of the QCustomPlot widget, to axis coordinates.

7.26.4.12 QList< QCPAbstractPlottable ∗ > QCPAxis::plottables ( ) const

Returns a list of all the plottables that have this axis as key or value axis.

If you are only interested in plottables of type QCPGraph, see graphs.

See also

graphs, items

7.26.4.13 void QCPAxis::rangeChanged (const QCPRange & *newRange*) [signal]

This signal is emitted when the range of this axis has changed. You can connect it to the setRange slot of another axis to communicate the new range to the other axis, in order for it to be synchronized.

You may also manipulate/correct the range with setRange in a slot connected to this signal. This is useful if for example a maximum range span shall not be exceeded, or if the lower/upper range shouldn't go beyond certain values. For example, the following slot would limit the x axis to only positive ranges:

```
if (newRange.lower < 0)
  plot->xAxis->setRange(0, newRange.size());
```

7.26.4.14 void QCPAxis::rangeChanged (const QCPRange & *newRange*, const QCPRange & *oldRange*) [signal]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Additionally to the new range, this signal also provides the previous range held by the axis as *oldRange*.

7.26.4.15 void QCPAxis::rescale (bool *onlyVisiblePlottables* = `false`)

Changes the axis range such that all plottables associated with this axis are fully visible in that dimension.

See also

QCPAbstractPlottable::rescaleAxes, QCustomPlot::rescaleAxes

7.26.4.16 void QCPAxis::scaleRange (double *factor,* double *center*)

Scales the range of this axis by *factor* around the coordinate *center*. For example, if *factor* is 2.0, *center* is 1.0, then the axis range will double its size, and the point at coordinate 1.0 won't have changed its position in the QCustomPlot widget (i.e. coordinates around 1.0 will have moved symmetrically closer to 1.0).

7.26.4.17 void QCPAxis::scaleTypeChanged (QCPAxis::ScaleType *scaleType*) `[signal]`

This signal is emitted when the scale type changes, by calls to setScaleType

7.26.4.18 void QCPAxis::selectableChanged (const QCPAxis::SelectableParts & *parts*) `[signal]`

This signal is emitted when the selectability changes, by calls to setSelectableParts

7.26.4.19 void QCPAxis::selectionChanged (const QCPAxis::SelectableParts & *parts*) `[signal]`

This signal is emitted when the selection state of this axis has changed, either by user interaction or by a direct call to setSelectedParts.

7.26.4.20 double QCPAxis::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Reimplemented from QCPLayerable.

7.26.4.21 void QCPAxis::setAutoSubTicks (bool *on*)

Sets whether the number of sub ticks in one tick interval is determined automatically. This works, as long as the tick step mantissa is a multiple of 0.5. When setAutoTickStep is enabled, this is always the case.

When *on* is set to false, you may set the sub tick count with setSubTickCount manually.

See also

setAutoTickCount, setAutoTicks, setAutoTickStep

7.26.4.22 void QCPAxis::setAutoTickCount (int *approximateCount*)

When setAutoTickStep is true, *approximateCount* determines how many ticks should be generated in the visible range, approximately.

It's not guaranteed that this number of ticks is met exactly, but approximately within a tolerance of about two.

Only values greater than zero are accepted as *approximateCount*.

See also

setAutoTickStep, setAutoTicks, setAutoSubTicks

7.26.4.23 void QCPAxis::setAutoTickLabels (bool *on*)

Sets whether the tick labels are generated automatically. Depending on the tick label type (ltNumber or ltDateTime), the labels will either show the coordinate as floating point number (setNumberFormat), or a date/time formatted according to setDateTimeFormat.

If *on* is set to false, you should provide the tick labels via setTickVectorLabels. This is usually used in a combination with setAutoTicks set to false for complete control over tick positions and labels, e.g. when the ticks should be at multiples of pi and show "2pi", "3pi" etc. as tick labels.

If you need dynamically calculated tick vectors (and possibly tick label vectors), set the vectors in a slot connected to the ticksRequest signal.

See also

setAutoTicks

7.26.4.24 void QCPAxis::setAutoTicks (bool *on*)

Sets whether the tick positions should be calculated automatically (either from an automatically generated tick step or a tick step provided manually via setTickStep, see setAutoTickStep).

If *on* is set to false, you must provide the tick positions manually via setTickVector. For these manual ticks you may let QCPAxis generate the appropriate labels automatically by leaving setAutoTickLabels set to true. If you also wish to control the displayed labels manually, set setAutoTickLabels to false and provide the label strings with setTickVectorLabels.

If you need dynamically calculated tick vectors (and possibly tick label vectors), set the vectors in a slot connected to the ticksRequest signal.

See also

setAutoTickLabels, setAutoSubTicks, setAutoTickCount, setAutoTickStep

7.26.4.25 void QCPAxis::setAutoTickStep (bool *on*)

Sets whether the tick step, i.e. the interval between two (major) ticks, is calculated automatically. If *on* is set to true, the axis finds a tick step that is reasonable for human readable plots.

The number of ticks the algorithm aims for within the visible range can be specified with setAutoTickCount.

If *on* is set to false, you may set the tick step manually with setTickStep.

See also

setAutoTicks, setAutoSubTicks, setAutoTickCount

7.26.4.26 void QCPAxis::setBasePen (const QPen & *pen*)

Sets the pen, the axis base line is drawn with.

See also

setTickPen, setSubTickPen

7.26.4.27 void QCPAxis::setDateTimeFormat (const QString & *format*)

Sets the format in which dates and times are displayed as tick labels, if setTickLabelType is ltDateTime. for details about the *format* string, see the documentation of QDateTime::toString().

Newlines can be inserted with "\n".

See also

setDateTimeSpec

7.26.4.28 void QCPAxis::setDateTimeSpec (const Qt::TimeSpec & *timeSpec*)

Sets the time spec that is used for the date time values when setTickLabelType is ltDateTime.

The default value of QDateTime objects (and also QCustomPlot) is `Qt::LocalTime`. However, if the date time values passed to QCustomPlot are given in the UTC spec, set *timeSpec* to `Qt::UTC` to get the correct axis labels.

See also

setDateTimeFormat

7.26.4.29 void QCPAxis::setLabel (const QString & *str*)

Sets the text of the axis label that will be shown below/above or next to the axis, depending on its orientation. To disable axis labels, pass an empty string as *str*.

7.26.4.30 void QCPAxis::setLabelColor (const QColor & *color*)

Sets the color of the axis label.

See also

setLabelFont

7.26.4.31 void QCPAxis::setLabelFont (const QFont & *font*)

Sets the font of the axis label.

See also

setLabelColor

7.26.4.32 void QCPAxis::setLabelPadding (int *padding*)

Sets the distance between the tick labels and the axis label.

See also

setTickLabelPadding, setPadding

7.26.4.33 void QCPAxis::setLowerEnding (const QCPLineEnding & *ending*)

Sets the style for the lower axis ending. See the documentation of QCPLineEnding for available styles.

For horizontal axes, this method refers to the left ending, for vertical axes the bottom ending. Note that this meaning does not change when the axis range is reversed with setRangeReversed.

See also

setUpperEnding

7.26.4.34 void QCPAxis::setNumberFormat (const QString & *formatCode*)

Sets the number format for the numbers drawn as tick labels (if tick label type is It-Number). This *formatCode* is an extended version of the format code used e.g. by Q-String::number() and QLocale::toString(). For reference about that, see the "Argument Formats" section in the detailed description of the QString class. *formatCode* is a string of one, two or three characters. The first character is identical to the normal format code used by Qt. In short, this means: 'e'/'E' scientific format, 'f' fixed format, 'g'/'G' scientific or fixed, whichever is shorter.

The second and third characters are optional and specific to QCustomPlot:

If the first char was 'e' or 'g', numbers are/might be displayed in the scientific format, e.g. "5.5e9", which is ugly in a plot. So when the second char of *formatCode* is set to 'b' (for "beautiful"), those exponential numbers are formatted in a more natural way, i.e. "5.5 [multiplication sign] 10 [superscript] 9". By default, the multiplication sign is a centered dot. If instead a cross should be shown (as is usual in the USA), the third char of *formatCode* can be set to 'c'. The inserted multiplication signs are the UTF-8 characters 215 (0xD7) for the cross and 183 (0xB7) for the dot.

If the scale type (setScaleType) is stLogarithmic and the *formatCode* uses the 'b' option (beautifully typeset decimal powers), the display usually is "1 [multiplication sign] 10 [superscript] n", which looks unnatural for logarithmic scaling (the "1 [multiplication sign]" part). To only display the decimal power, set the number precision to zero with setNumberPrecision.

Examples for *formatCode:*

- `g` normal format code behaviour. If number is small, fixed format is used, if number is large, normal scientific format is used
- `gb` If number is small, fixed format is used, if number is large, scientific format is used with beautifully typeset decimal powers and a dot as multiplication sign
- `ebc` All numbers are in scientific format with beautifully typeset decimal power and a cross as multiplication sign
- `fb` illegal format code, since fixed format doesn't support (or need) beautifully typeset decimal powers. Format code will be reduced to 'f'.
- `hello` illegal format code, since first char is not 'e', 'E', 'f', 'g' or 'G'. Current format code will not be changed.

7.26.4.35 void **QCPAxis::setNumberPrecision** (int *precision*)

Sets the precision of the tick label numbers. See QLocale::toString(double i, char f, int prec) for details. The effect of precisions are most notably for number Formats starting with 'e', see setNumberFormat

If the scale type (setScaleType) is stLogarithmic and the number format (setNumberFormat) uses the 'b' format code (beautifully typeset decimal powers), the display usually is "1 [multiplication sign] 10 [superscript] n", which looks unnatural for logarithmic scaling (the redundant "1 [multiplication sign]" part). To only display the decimal power "10 [superscript] n", set *precision* to zero.

7.26.4.36 void **QCPAxis::setOffset** (int *offset*)

Sets the offset the axis has to its axis rect side.

If an axis rect side has multiple axes and automatic margin calculation is enabled for that side, only the offset of the inner most axis has meaning (even if it is set to be invisible). The offset of the other, outer axes is controlled automatically, to place them at appropriate positions.

7.26.4.37 void **QCPAxis::setPadding** (int *padding*)

Sets the padding of the axis.

When QCPAxisRect::setAutoMargins is enabled, the padding is the additional outer most space, that is left blank.

The axis padding has no meaning if QCPAxisRect::setAutoMargins is disabled.

See also

setLabelPadding, setTickLabelPadding

7.26.4.38 void QCPAxis::setRange (const QCPRange & *range*)

Sets the range of the axis.

This slot may be connected with the rangeChanged signal of another axis so this axis is always synchronized with the other axis range, when it changes.

To invert the direction of an axis, use setRangeReversed.

7.26.4.39 void QCPAxis::setRange (double *lower,* double *upper*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the lower and upper bound of the axis range.

To invert the direction of an axis, use setRangeReversed.

There is also a slot to set a range, see setRange(const QCPRange &range).

7.26.4.40 void QCPAxis::setRange (double *position,* double *size,* Qt::AlignmentFlag *alignment*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the range of the axis.

The *position* coordinate indicates together with the *alignment* parameter, where the new range will be positioned. *size* defines the size of the new axis range. *alignment* may be Qt::AlignLeft, Qt::AlignRight or Qt::AlignCenter. This will cause the left border, right border, or center of the range to be aligned with *position*. Any other values of *alignment* will default to Qt::AlignCenter.

7.26.4.41 void QCPAxis::setRangeLower (double *lower*)

Sets the lower bound of the axis range. The upper bound is not changed.

See also

setRange

7.26.4.42 void QCPAxis::setRangeReversed (bool *reversed*)

Sets whether the axis range (direction) is displayed reversed. Normally, the values on horizontal axes increase left to right, on vertical axes bottom to top. When *reversed* is set to true, the direction of increasing values is inverted.

Note that the range and data interface stays the same for reversed axes, e.g. the *lower* part of the setRange interface will still reference the mathematically smaller number than the *upper* part.

7.26.4.43 void QCPAxis::setRangeUpper (double *upper*)

Sets the upper bound of the axis range. The lower bound is not changed.

See also

setRange

7.26.4.44 void QCPAxis::setScaleLogBase (double *base*)

If setScaleType is set to stLogarithmic, *base* will be the logarithm base of the scaling. In logarithmic axis scaling, major tick marks appear at all powers of *base*.

Properties like tick step (setTickStep) don't apply in logarithmic scaling. If you wish a decimal base but less major ticks, consider choosing *base* 100, 1000 or even higher.

7.26.4.45 void QCPAxis::setScaleRatio (const QCPAxis ∗ *otherAxis,* double *ratio* = `1.0`)

Scales the range of this axis to have a certain scale *ratio* to *otherAxis*. The scaling will be done around the center of the current axis range.

For example, if *ratio* is 1, this axis is the *yAxis* and *otherAxis* is *xAxis*, graphs plotted with those axes will appear in a 1:1 aspect ratio, independent of the aspect ratio the axis rect has.

This is an operation that changes the range of this axis once, it doesn't fix the scale ratio indefinitely. Note that calling this function in the constructor of the QCustomPlot's parent won't have the desired effect, since the widget dimensions aren't defined yet, and a resizeEvent will follow.

7.26.4.46 void QCPAxis::setScaleType (QCPAxis::ScaleType *type*)

Sets whether the axis uses a linear scale or a logarithmic scale. If *type* is set to stLogarithmic, the logarithm base can be set with setScaleLogBase. In logarithmic axis scaling, major tick marks appear at all powers of the logarithm base. Properties like tick step (setTickStep) don't apply in logarithmic scaling. If you wish a decimal base but less major ticks, consider choosing a logarithm base of 100, 1000 or even higher.

If *type* is stLogarithmic and the number format (setNumberFormat) uses the 'b' option (beautifully typeset decimal powers), the display usually is "1 [multiplication sign] 10 [superscript] n", which looks unnatural for logarithmic scaling (the "1 [multiplication sign]" part). To only display the decimal power, set the number precision to zero with setNumberPrecision.

7.26.4.47 void **QCPAxis::setSelectableParts** (const QCPAxis::SelectableParts & *selectableParts*)

Sets whether the user can (de-)select the parts in *selectable* by clicking on the QCustomPlot surface. (When QCustomPlot::setInteractions contains iSelectAxes.)

However, even when *selectable* is set to a value not allowing the selection of a specific part, it is still possible to set the selection of this part manually, by calling setSelectedParts directly.

See also

SelectablePart, setSelectedParts

7.26.4.48 void **QCPAxis::setSelectedBasePen** (const QPen & *pen*)

Sets the pen that is used to draw the axis base line when selected.

See also

setBasePen, setSelectableParts, setSelectedParts, QCustomPlot::setInteractions

7.26.4.49 void **QCPAxis::setSelectedLabelColor** (const QColor & *color*)

Sets the color that is used for the axis label when it is selected.

See also

setLabelColor, setSelectableParts, setSelectedParts, QCustomPlot::setInteractions

7.26.4.50 void **QCPAxis::setSelectedLabelFont** (const QFont & *font*)

Sets the font that is used for the axis label when it is selected.

See also

setLabelFont, setSelectableParts, setSelectedParts, QCustomPlot::setInteractions

7.26.4.51 void **QCPAxis::setSelectedParts** (const QCPAxis::SelectableParts & *selectedParts*)

Sets the selected state of the respective axis parts described by SelectablePart. When a part is selected, it uses a different pen/font.

The entire selection mechanism for axes is handled automatically when QCustomPlot::setInteractions contains iSelectAxes. You only need to call this function when you wish to change the selection state manually.

This function can change the selection state of a part, independent of the setSelectable-Parts setting.

emits the selectionChanged signal when *selected* is different from the previous selection state.

See also

SelectablePart, setSelectableParts, selectTest, setSelectedBasePen, setSelected-TickPen, setSelectedSubTickPen, setSelectedTickLabelFont, setSelectedLabel-Font, setSelectedTickLabelColor, setSelectedLabelColor

7.26.4.52 void **QCPAxis::setSelectedSubTickPen** (const QPen & *pen*)

Sets the pen that is used to draw the subticks when selected.

See also

setSubTickPen, setSelectableParts, setSelectedParts, QCustomPlot::set-Interactions

7.26.4.53 void **QCPAxis::setSelectedTickLabelColor** (const QColor & *color*)

Sets the color that is used for tick labels when they are selected.

See also

setTickLabelColor, setSelectableParts, setSelectedParts, QCustomPlot::set-Interactions

7.26.4.54 void **QCPAxis::setSelectedTickLabelFont** (const QFont & *font*)

Sets the font that is used for tick labels when they are selected.

See also

setTickLabelFont, setSelectableParts, setSelectedParts, QCustomPlot::set-Interactions

7.26.4.55 void **QCPAxis::setSelectedTickPen** (const QPen & *pen*)

Sets the pen that is used to draw the (major) ticks when selected.

See also

setTickPen, setSelectableParts, setSelectedParts, QCustomPlot::setInteractions

7.26.4.56 void QCPAxis::setSubTickCount (int *count*)

Sets the number of sub ticks in one (major) tick step. A sub tick count of three for example, divides the tick intervals in four sub intervals.

By default, the number of sub ticks is chosen automatically in a reasonable manner as long as the mantissa of the tick step is a multiple of 0.5. When setAutoTickStep is enabled, this is always the case.

If you want to disable automatic sub tick count and use this function to set the count manually, see setAutoSubTicks.

7.26.4.57 void QCPAxis::setSubTickLength (int *inside,* int *outside =* 0)

Sets the length of the subticks in pixels. *inside* is the length the subticks will reach inside the plot and *outside* is the length they will reach outside the plot. If *outside* is greater than zero, the tick labels and axis label will increase their distance to the axis accordingly, so they won't collide with the ticks.

See also

setTickLength, setSubTickLengthIn, setSubTickLengthOut

7.26.4.58 void QCPAxis::setSubTickLengthIn (int *inside*)

Sets the length of the inward subticks in pixels. *inside* is the length the subticks will reach inside the plot.

See also

setSubTickLengthOut, setSubTickLength, setTickLength

7.26.4.59 void QCPAxis::setSubTickLengthOut (int *outside*)

Sets the length of the outward subticks in pixels. *outside* is the length the subticks will reach outside the plot. If *outside* is greater than zero, the tick labels will increase their distance to the axis accordingly, so they won't collide with the ticks.

See also

setSubTickLengthIn, setSubTickLength, setTickLength

7.26.4.60 void QCPAxis::setSubTickPen (const QPen & *pen*)

Sets the pen, subtick marks will be drawn with.

See also

setSubTickCount, setSubTickLength, setBasePen

7.26.4.61 void QCPAxis::setTickLabelColor (const QColor & *color*)

Sets the color of the tick labels.

See also

setTickLabels, setTickLabelFont

7.26.4.62 void QCPAxis::setTickLabelFont (const QFont & *font*)

Sets the font of the tick labels.

See also

setTickLabels, setTickLabelColor

7.26.4.63 void QCPAxis::setTickLabelPadding (int *padding*)

Sets the distance between the axis base line (including any outward ticks) and the tick labels.

See also

setLabelPadding, setPadding

7.26.4.64 void QCPAxis::setTickLabelRotation (double *degrees*)

Sets the rotation of the tick labels. If *degrees* is zero, the labels are drawn normally. Else, the tick labels are drawn rotated by *degrees* clockwise. The specified angle is bound to values from -90 to 90 degrees.

If *degrees* is exactly -90, 0 or 90, the tick labels are centered on the tick coordinate. For other angles, the label is drawn with an offset such that it seems to point toward or away from the tick mark.

7.26.4.65 void QCPAxis::setTickLabels (bool *show*)

Sets whether tick labels are displayed. Tick labels are the numbers drawn next to tick marks.

7.26.4.66 void QCPAxis::setTickLabelSide (LabelSide *side*)

Sets whether the tick labels (numbers) shall appear inside or outside the axis rect.

The usual and default setting is lsOutside. Very compact plots sometimes require tick labels to be inside the axis rect, to save space. If *side* is set to lsInside, the tick labels appear on the inside are additionally clipped to the axis rect.

7.26.4.67 void QCPAxis::setTickLabelType (LabelType *type*)

Sets whether the tick labels display numbers or dates/times.

If *type* is set to ltNumber, the format specifications of setNumberFormat apply.

If *type* is set to ltDateTime, the format specifications of setDateTimeFormat apply.

In QCustomPlot, date/time coordinates are `double` numbers representing the seconds since 1970-01-01T00:00:00 UTC. This format can be retrieved from QDateTime objects with the QDateTime::toTime_t() function. Since this only gives a resolution of one second, there is also the QDateTime::toMSecsSinceEpoch() function which returns the timespan described above in milliseconds. Divide its return value by 1000.0 to get a value with the format needed for date/time plotting, with a resolution of one millisecond.

Using the toMSecsSinceEpoch function allows dates that go back to 2nd January 4713 B.C. (represented by a negative number), unlike the toTime_t function, which works with unsigned integers and thus only goes back to 1st January 1970. So both for range and accuracy, use of toMSecsSinceEpoch()/1000.0 should be preferred as key coordinate for date/time axes.

See also

setTickLabels

7.26.4.68 void QCPAxis::setTickLength (int *inside*, int *outside* = `0`)

Sets the length of the ticks in pixels. *inside* is the length the ticks will reach inside the plot and *outside* is the length they will reach outside the plot. If *outside* is greater than zero, the tick labels and axis label will increase their distance to the axis accordingly, so they won't collide with the ticks.

See also

setSubTickLength, setTickLengthIn, setTickLengthOut

7.26.4.69 void QCPAxis::setTickLengthIn (int *inside*)

Sets the length of the inward ticks in pixels. *inside* is the length the ticks will reach inside the plot.

See also

setTickLengthOut, setTickLength, setSubTickLength

7.26.4.70 void QCPAxis::setTickLengthOut (int *outside*)

Sets the length of the outward ticks in pixels. *outside* is the length the ticks will reach outside the plot. If *outside* is greater than zero, the tick labels and axis label will increase their distance to the axis accordingly, so they won't collide with the ticks.

See also

setTickLengthIn, setTickLength, setSubTickLength

7.26.4.71 void QCPAxis::setTickPen (const QPen & *pen*)

Sets the pen, tick marks will be drawn with.

See also

setTickLength, setBasePen

7.26.4.72 void QCPAxis::setTicks (bool *show*)

Sets whether tick marks are displayed.

Note that setting *show* to false does not imply that tick labels are invisible, too. To achieve that, see setTickLabels.

7.26.4.73 void QCPAxis::setTickStep (double *step*)

If setAutoTickStep is set to false, use this function to set the tick step manually. The tick step is the interval between (major) ticks, in plot coordinates.

See also

setSubTickCount

7.26.4.74 void QCPAxis::setTickVector (const QVector< double > & *vec*)

If you want full control over what ticks (and possibly labels) the axes show, this function is used to set the coordinates at which ticks will appear.setAutoTicks must be disabled, else the provided tick vector will be overwritten with automatically generated tick coordinates upon replot. The labels of the ticks can be generated automatically when setAutoTickLabels is left enabled. If it is disabled, you can set the labels manually with setTickVectorLabels.

vec is a vector containing the positions of the ticks, in plot coordinates.

Warning

vec must be sorted in ascending order, no additional checks are made to ensure this.

See also

setTickVectorLabels

7.26.4.75 void QCPAxis::setTickVectorLabels (const QVector< QString > & *vec*)

If you want full control over what ticks and labels the axes show, this function is used to set a number of QStrings that will be displayed at the tick positions which you need to provide with setTickVector. These two vectors should have the same size. (Note that you need to disable setAutoTicks and setAutoTickLabels first.)

vec is a vector containing the labels of the ticks. The entries correspond to the respective indices in the tick vector, passed via setTickVector.

See also

setTickVector

7.26.4.76 void QCPAxis::setUpperEnding (const QCPLineEnding & *ending*)

Sets the style for the upper axis ending. See the documentation of QCPLineEnding for available styles.

For horizontal axes, this method refers to the right ending, for vertical axes the top ending. Note that this meaning does not change when the axis range is reversed with setRangeReversed.

See also

setLowerEnding

7.26.4.77 void QCPAxis::ticksRequest ( ) `[signal]`

This signal is emitted when setAutoTicks is false and the axis is about to generate tick labels for a replot.

Modifying the tick positions can be done with setTickVector. If you also want to control the tick labels, set setAutoTickLabels to false and also provide the labels with setTick-VectorLabels.

If you only want static ticks you probably don't need this signal, since you can just set the tick vector (and possibly tick label vector) once. However, if you want to provide ticks (and maybe labels) dynamically, e.g. depending on the current axis range, connect a slot to this signal and set the vector/vectors there.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.27 QCPAxisPainterPrivate Class Reference

Classes

- struct CachedLabel
- struct TickLabelData

Public Member Functions

- QCPAxisPainterPrivate (QCustomPlot ∗parentPlot)
- virtual void **draw** (QCPPainter ∗painter)
- virtual int **size** () const
- void **clearCache** ()
- QRect **axisSelectionBox** () const
- QRect **tickLabelsSelectionBox** () const
- QRect **labelSelectionBox** () const

Public Attributes

- QCPAxis::AxisType **type**
- QPen **basePen**
- QCPLineEnding **lowerEnding**
- QCPLineEnding **upperEnding**
- int **labelPadding**
- QFont **labelFont**
- QColor **labelColor**
- QString **label**
- int **tickLabelPadding**
- double **tickLabelRotation**
- QCPAxis::LabelSide **tickLabelSide**
- bool **substituteExponent**
- bool **numberMultiplyCross**
- int **tickLengthIn**
- int **tickLengthOut**
- int **subTickLengthIn**
- int **subTickLengthOut**
- QPen **tickPen**
- QPen **subTickPen**
- QFont **tickLabelFont**
- QColor **tickLabelColor**
- QRect **axisRect**
- QRect **viewportRect**
- double **offset**
- bool **abbreviateDecimalPowers**
- bool **reversedEndings**
- QVector< double > **subTickPositions**
- QVector< double > **tickPositions**
- QVector< QString > **tickLabels**

Protected Member Functions

- virtual QByteArray **generateLabelParameterHash** () const
- virtual void **placeTickLabel** (QCPPainter ∗painter, double position, int distance-ToAxis, const QString &text, QSize ∗tickLabelsSize)
- virtual void **drawTickLabel** (QCPPainter ∗painter, double x, double y, const Tick-LabelData &labelData) const
- virtual TickLabelData **getTickLabelData** (const QFont &font, const QString &text) const
- virtual QPointF **getTickLabelDrawOffset** (const TickLabelData &labelData) const
- virtual void **getMaxTickLabelSize** (const QFont &font, const QString &text, Q-Size ∗tickLabelsSize) const

Protected Attributes

- QCustomPlot ∗ **mParentPlot**
- QByteArray **mLabelParameterHash**
- QCache< QString, CachedLabel > **mLabelCache**
- QRect **mAxisSelectionBox**
- QRect **mTickLabelsSelectionBox**
- QRect **mLabelSelectionBox**

7.27.1 Constructor & Destructor Documentation

7.27.1.1 QCPAxisPainterPrivate::QCPAxisPainterPrivate (QCustomPlot ∗ *parentPlot*) [explicit]

Constructs a QCPAxisPainterPrivate instance. Make sure to not create a new instance on every redraw, to utilize the caching mechanisms.

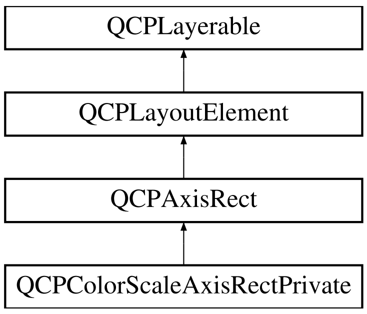
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.28 QCPAxisRect Class Reference

Holds multiple axes and arranges them in a rectangular shape.

Inheritance diagram for QCPAxisRect:



Public Member Functions

- QCPAxisRect (QCustomPlot ∗parentPlot, bool setupDefaultAxes=true)
- QPixmap **background** () const
- bool **backgroundScaled** () const
- Qt::AspectRatioMode **backgroundScaledMode** () const
- Qt::Orientations **rangeDrag** () const
- Qt::Orientations **rangeZoom** () const
- QCPAxis ∗ rangeDragAxis (Qt::Orientation orientation)
- QCPAxis ∗ rangeZoomAxis (Qt::Orientation orientation)
- double rangeZoomFactor (Qt::Orientation orientation)
- void setBackground (const QPixmap &pm)
- void setBackground (const QPixmap &pm, bool scaled, Qt::AspectRatioMode mode=Qt::KeepAspectRatioByExpanding)
- void setBackground (const QBrush &brush)
- void setBackgroundScaled (bool scaled)
- void setBackgroundScaledMode (Qt::AspectRatioMode mode)
- void setRangeDrag (Qt::Orientations orientations)
- void setRangeZoom (Qt::Orientations orientations)
- void setRangeDragAxes (QCPAxis ∗horizontal, QCPAxis ∗vertical)
- void setRangeZoomAxes (QCPAxis ∗horizontal, QCPAxis ∗vertical)
- void setRangeZoomFactor (double horizontalFactor, double verticalFactor)
- void setRangeZoomFactor (double factor)
- int axisCount (QCPAxis::AxisType type) const
- QCPAxis ∗ axis (QCPAxis::AxisType type, int index=0) const
- QList< QCPAxis ∗ > axes (QCPAxis::AxisTypes types) const
- QList< QCPAxis ∗ > axes () const
- QCPAxis ∗ addAxis (QCPAxis::AxisType type, QCPAxis ∗axis=0)
- QList< QCPAxis ∗ > addAxes (QCPAxis::AxisTypes types)
- bool removeAxis (QCPAxis ∗axis)
- QCPLayoutInset ∗ insetLayout () const
- void setupFullAxesBox (bool connectRanges=false)
- QList< QCPAbstractPlottable ∗ > plottables () const
- QList< QCPGraph ∗ > graphs () const
- QList< QCPAbstractItem ∗ > items () const

- int left () const
- int right () const
- int top () const
- int bottom () const
- int width () const
- int height () const
- QSize size () const
- QPoint topLeft () const
- QPoint topRight () const
- QPoint bottomLeft () const
- QPoint bottomRight () const
- QPoint center () const
- virtual void update (UpdatePhase phase)
- virtual QList< QCPLayoutElement ∗ > elements (bool recursive) const

Protected Member Functions

- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- virtual int **calculateAutoMargin** (QCP::MarginSide side)
- virtual void mousePressEvent (QMouseEvent ∗event)
- virtual void mouseMoveEvent (QMouseEvent ∗event)
- virtual void mouseReleaseEvent (QMouseEvent ∗event)
- virtual void wheelEvent (QWheelEvent ∗event)
- void **drawBackground** (QCPPainter ∗painter)
- void **updateAxesOffset** (QCPAxis::AxisType type)

Protected Attributes

- QBrush **mBackgroundBrush**
- QPixmap **mBackgroundPixmap**
- QPixmap **mScaledBackgroundPixmap**
- bool **mBackgroundScaled**
- Qt::AspectRatioMode **mBackgroundScaledMode**
- QCPLayoutInset ∗ **mInsetLayout**
- Qt::Orientations **mRangeDrag**
- Qt::Orientations **mRangeZoom**
- QPointer< QCPAxis > **mRangeDragHorzAxis**
- QPointer< QCPAxis > **mRangeDragVertAxis**
- QPointer< QCPAxis > **mRangeZoomHorzAxis**
- QPointer< QCPAxis > **mRangeZoomVertAxis**
- double **mRangeZoomFactorHorz**
- double **mRangeZoomFactorVert**
- QCPRange **mDragStartHorzRange**
- QCPRange **mDragStartVertRange**

- QCP::AntialiasedElements **mAADragBackup**
- QCP::AntialiasedElements **mNotAADragBackup**
- QPoint **mDragStart**
- bool **mDragging**
- QHash< QCPAxis::AxisType, QList< QCPAxis ∗ > > **mAxes**

Friends

- class **QCustomPlot**

7.28.1 Detailed Description

Holds multiple axes and arranges them in a rectangular shape.

This class represents an axis rect, a rectangular area that is bounded on all sides with an arbitrary number of axes.

Initially QCustomPlot has one axis rect, accessible via QCustomPlot::axisRect(). - However, the layout system allows to have multiple axis rects, e.g. arranged in a grid layout (QCustomPlot::plotLayout).

By default, QCPAxisRect comes with four axes, at bottom, top, left and right. They can be accessed via axis by providing the respective axis type (QCPAxis::AxisType) and index. If you need all axes in the axis rect, use axes. The top and right axes are set to be invisible initially (QCPAxis::setVisible). To add more axes to a side, use addAxis or addAxes. To remove an axis, use removeAxis.

The axis rect layerable itself only draws a background pixmap or color, if specified (set-Background). It is placed on the "background" layer initially (see QCPLayer for an explanation of the QCustomPlot layer system). The axes that are held by the axis rect can be placed on other layers, independently of the axis rect.

Every axis rect has a child layout of type QCPLayoutInset. It is accessible via inset-Layout and can be used to have other layout elements (or even other layouts with multiple elements) hovering inside the axis rect.

If an axis rect is clicked and dragged, it processes this by moving certain axis ranges. The behaviour can be controlled with setRangeDrag and setRangeDragAxes. If the mouse wheel is scrolled while the cursor is on the axis rect, certain axes are scaled. This is controllable via setRangeZoom, setRangeZoomAxes and setRangeZoomFactor. These interactions are only enabled if QCustomPlot::setInteractions contains QCP::i-RangeDrag and QCP::iRangeZoom.

Overview of the spacings and paddings that define the geometry of an axis. The dashed line on the far left indicates the viewport/widget border.

7.28.2 Constructor & Destructor Documentation

7.28.2.1 QCPAxisRect::QCPAxisRect (QCustomPlot ∗ *parentPlot,* bool *setupDefaultAxes* = `true`) `[explicit]`

Creates a QCPAxisRect instance and sets default values. An axis is added for each of the four sides, the top and right axes are set invisible initially.

7.28.3 Member Function Documentation

7.28.3.1 QList< QCPAxis ∗ > QCPAxisRect::addAxes (QCPAxis::AxisTypes *types*)

Adds a new axis with addAxis to each axis rect side specified in *types*. This may be an `or`-combination of QCPAxis::AxisType, so axes can be added to multiple sides at once.

Returns a list of the added axes.

See also

addAxis, setupFullAxesBox

7.28.3.2 QCPAxis ∗ QCPAxisRect::addAxis (QCPAxis::AxisType *type,* QCPAxis ∗ *axis* = `0`)

Adds a new axis to the axis rect side specified with *type*, and returns it. If *axis* is 0, a new QCPAxis instance is created internally.

You may inject QCPAxis instances (or sublasses of QCPAxis) by setting *axis* to an axis that was previously created outside QCustomPlot. It is important to note that QCustom-Plot takes ownership of the axis, so you may not delete it afterwards. Further, the *axis* must have been created with this axis rect as parent and with the same axis type as specified in *type*. If this is not the case, a debug output is generated, the axis is not added, and the method returns 0.

This method can not be used to move *axis* between axis rects. The same *axis* instance must not be added multiple times to the same or different axis rects.

If an axis rect side already contains one or more axes, the lower and upper endings of the new axis (QCPAxis::setLowerEnding, QCPAxis::setUpperEnding) are set to QCP-LineEnding::esHalfBar.

See also

addAxes, setupFullAxesBox

7.28.3.3 QList< QCPAxis ∗ > QCPAxisRect::axes (QCPAxis::AxisTypes *types*) const

Returns all axes on the axis rect sides specified with *types*.

types may be a single QCPAxis::AxisType or an `or`-combination, to get the axes of multiple sides.

See also

axis

7.28.3.4 QList< QCPAxis * > QCPAxisRect::axes ( ) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Returns all axes of this axis rect.

7.28.3.5 QCPAxis * QCPAxisRect::axis (QCPAxis::AxisType *type,* int *index* = 0) const

Returns the axis with the given *index* on the axis rect side specified with *type*.

See also

axisCount, axes

7.28.3.6 int QCPAxisRect::axisCount (QCPAxis::AxisType *type*) const

Returns the number of axes on the axis rect side specified with *type*.

See also

axis

7.28.3.7 int QCPAxisRect::bottom ( ) const [inline]

Returns the pixel position of the bottom border of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.8 QPoint QCPAxisRect::bottomLeft ( ) const [inline]

Returns the bottom left corner of this axis rect in pixels. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.9 QPoint QCPAxisRect::bottomRight ( ) const [inline]

Returns the bottom right corner of this axis rect in pixels. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.10 QPoint QCPAxisRect::center ( ) const [inline]

Returns the center of this axis rect in pixels. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.11 QList< QCPLayoutElement * > QCPAxisRect::elements (bool *recursive*) const [virtual]

Returns a list of all child elements in this layout element. If *recursive* is true, all sub-child elements are included in the list, too.

Warning

There may be entries with value 0 in the returned list. (For example, QCPLayout-Grid may have empty cells which yield 0 at the respective index.)

Reimplemented from QCPLayoutElement.

7.28.3.12 QList< QCPGraph * > QCPAxisRect::graphs ( ) const

Returns a list of all the graphs that are associated with this axis rect.

A graph is considered associated with an axis rect if its key or value axis (or both) is in this axis rect.

See also

plottables, items

7.28.3.13 int QCPAxisRect::height ( ) const [inline]

Returns the pixel height of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.14 QCPLayoutInset * QCPAxisRect::insetLayout ( ) const [inline]

Returns the inset layout of this axis rect. It can be used to place other layout elements (or even layouts with multiple other elements) inside/on top of an axis rect.

See also

QCPLayoutInset

7.28.3.15 QList< QCPAbstractItem * > QCPAxisRect::items ( ) const

Returns a list of all the items that are associated with this axis rect.

An item is considered associated with an axis rect if any of its positions has key or value axis set to an axis that is in this axis rect, or if any of its positions has QCPItemPosition::setAxisRect set to the axis rect, or if the clip axis rect (QCPAbstractItem::setClipAxisRect) is set to this axis rect.

See also

plottables, graphs

7.28.3.16 int QCPAxisRect::left ( ) const [inline]

Returns the pixel position of the left border of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.17 void QCPAxisRect::mouseMoveEvent (QMouseEvent ∗ *event*) [protected, virtual]

This event is called, if the mouse is moved inside the outer rect of this layout element.

Reimplemented from QCPLayoutElement.

7.28.3.18 void QCPAxisRect::mousePressEvent (QMouseEvent ∗ *event*) [protected, virtual]

This event is called, if the mouse was pressed while being inside the outer rect of this layout element.

Reimplemented from QCPLayoutElement.

7.28.3.19 void QCPAxisRect::mouseReleaseEvent (QMouseEvent ∗ *event*) [protected, virtual]

This event is called, if the mouse was previously pressed inside the outer rect of this layout element and is now released.

Reimplemented from QCPLayoutElement.

7.28.3.20 QList< QCPAbstractPlottable ∗ > QCPAxisRect::plottables ( ) const

Returns a list of all the plottables that are associated with this axis rect.

A plottable is considered associated with an axis rect if its key or value axis (or both) is in this axis rect.

See also

graphs, items

7.28.3.21 QCPAxis ∗ QCPAxisRect::rangeDragAxis (Qt::Orientation *orientation*)

Returns the range drag axis of the *orientation* provided.

See also

setRangeDragAxes

7.28.3.22 QCPAxis ∗ QCPAxisRect::rangeZoomAxis (Qt::Orientation *orientation*)

Returns the range zoom axis of the *orientation* provided.

See also

setRangeZoomAxes

7.28.3.23 double QCPAxisRect::rangeZoomFactor (Qt::Orientation *orientation*)

Returns the range zoom factor of the *orientation* provided.

See also

setRangeZoomFactor

7.28.3.24 bool QCPAxisRect::removeAxis (QCPAxis ∗ *axis*)

Removes the specified *axis* from the axis rect and deletes it.

Returns true on success, i.e. if *axis* was a valid axis in this axis rect.

See also

addAxis

7.28.3.25 int QCPAxisRect::right ( ) const [inline]

Returns the pixel position of the right border of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.26 void QCPAxisRect::setBackground (const QPixmap & *pm*)

Sets *pm* as the axis background pixmap. The axis background pixmap will be drawn inside the axis rect. Since axis rects place themselves on the "background" layer by default, the axis rect backgrounds are usually drawn below everything else.

For cases where the provided pixmap doesn't have the same size as the axis rect, scaling can be enabled with setBackgroundScaled and the scaling mode (i.e. whether and how the aspect ratio is preserved) can be set with setBackgroundScaledMode. To set all these options in one call, consider using the overloaded version of this function.

Below the pixmap, the axis rect may be optionally filled with a brush, if specified with setBackground(const QBrush &brush).

See also

setBackgroundScaled, setBackgroundScaledMode, setBackground(const QBrush &brush)

7.28.3.27 void QCPAxisRect::setBackground (const QPixmap & *pm*, bool *scaled*, Qt::AspectRatioMode *mode* = `Qt::KeepAspectRatioByExpanding`)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows setting the background pixmap of the axis rect, whether it shall be scaled and how it shall be scaled in one call.

See also

setBackground(const QPixmap &pm), setBackgroundScaled, setBackgroundScaledMode

7.28.3.28 void QCPAxisRect::setBackground (const QBrush & *brush*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets *brush* as the background brush. The axis rect background will be filled with this brush. Since axis rects place themselves on the "background" layer by default, the axis rect backgrounds are usually drawn below everything else.

The brush will be drawn before (under) any background pixmap, which may be specified with setBackground(const QPixmap &pm).

To disable drawing of a background brush, set *brush* to Qt::NoBrush.

See also

setBackground(const QPixmap &pm)

7.28.3.29 void QCPAxisRect::setBackgroundScaled (bool *scaled*)

Sets whether the axis background pixmap shall be scaled to fit the axis rect or not. If *scaled* is set to true, you may control whether and how the aspect ratio of the original pixmap is preserved with setBackgroundScaledMode.

Note that the scaled version of the original pixmap is buffered, so there is no performance penalty on replots. (Except when the axis rect dimensions are changed continuously.)

See also

setBackground, setBackgroundScaledMode

7.28.3.30 void QCPAxisRect::setBackgroundScaledMode (Qt::AspectRatioMode *mode*)

If scaling of the axis background pixmap is enabled (setBackgroundScaled), use this function to define whether and how the aspect ratio of the original pixmap passed to setBackground is preserved.

See also

setBackground, setBackgroundScaled

7.28.3.31 void QCPAxisRect::setRangeDrag (Qt::Orientations *orientations*)

Sets which axis orientation may be range dragged by the user with mouse interaction. What orientation corresponds to which specific axis can be set with setRangeDragAxes(QCPAxis ∗horizontal, QCPAxis ∗vertical). By default, the horizontal axis is the bottom axis (xAxis) and the vertical axis is the left axis (yAxis).

To disable range dragging entirely, pass 0 as *orientations* or remove QCP::iRangeDrag from QCustomPlot::setInteractions. To enable range dragging for both directions, pass `Qt::Horizontal | Qt::Vertical` as *orientations*.

In addition to setting *orientations* to a non-zero value, make sure QCustomPlot::setInteractions contains QCP::iRangeDrag to enable the range dragging interaction.

See also

setRangeZoom, setRangeDragAxes, QCustomPlot::setNoAntialiasingOnDrag

7.28.3.32 void QCPAxisRect::setRangeDragAxes (QCPAxis ∗ *horizontal*, QCPAxis ∗ *vertical*)

Sets the axes whose range will be dragged when setRangeDrag enables mouse range dragging on the QCustomPlot widget.

See also

setRangeZoomAxes

7.28.3.33 void QCPAxisRect::setRangeZoom (Qt::Orientations *orientations*)

Sets which axis orientation may be zoomed by the user with the mouse wheel. What orientation corresponds to which specific axis can be set with setRangeZoomAxes(QCPAxis ∗horizontal, QCPAxis ∗vertical). By default, the horizontal axis is the bottom axis (xAxis) and the vertical axis is the left axis (yAxis).

To disable range zooming entirely, pass 0 as *orientations* or remove QCP::iRangeZoom from QCustomPlot::setInteractions. To enable range zooming for both directions, pass `Qt::Horizontal | Qt::Vertical` as *orientations*.

In addition to setting *orientations* to a non-zero value, make sure QCustomPlot::setInteractions contains QCP::iRangeZoom to enable the range zooming interaction.

See also

setRangeZoomFactor, setRangeZoomAxes, setRangeDrag

7.28.3.34 void **QCPAxisRect::setRangeZoomAxes** (**QCPAxis** ∗ *horizontal*, **QCPAxis** ∗ *vertical*)

Sets the axes whose range will be zoomed when setRangeZoom enables mouse wheel zooming on the QCustomPlot widget. The two axes can be zoomed with different strengths, when different factors are passed to setRangeZoomFactor(double horizontalFactor, double verticalFactor).

See also

setRangeDragAxes

7.28.3.35 void **QCPAxisRect::setRangeZoomFactor** (double *horizontalFactor*, double *verticalFactor*)

Sets how strong one rotation step of the mouse wheel zooms, when range zoom was activated with setRangeZoom. The two parameters *horizontalFactor* and *verticalFactor* provide a way to let the horizontal axis zoom at different rates than the vertical axis. Which axis is horizontal and which is vertical, can be set with setRangeZoomAxes.

When the zoom factor is greater than one, scrolling the mouse wheel backwards (towards the user) will zoom in (make the currently visible range smaller). For zoom factors smaller than one, the same scrolling direction will zoom out.

7.28.3.36 void **QCPAxisRect::setRangeZoomFactor** (double *factor*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets both the horizontal and vertical zoom *factor*.

7.28.3.37 void **QCPAxisRect::setupFullAxesBox** (bool *connectRanges* = `false`)

Convenience function to create an axis on each side that doesn't have any axes yet and set their visibility to true. Further, the top/right axes are assigned the following properties of the bottom/left axes:

- range (QCPAxis::setRange)
- range reversed (QCPAxis::setRangeReversed)

- scale type (QCPAxis::setScaleType)
- scale log base (QCPAxis::setScaleLogBase)
- ticks (QCPAxis::setTicks)
- auto (major) tick count (QCPAxis::setAutoTickCount)
- sub tick count (QCPAxis::setSubTickCount)
- auto sub ticks (QCPAxis::setAutoSubTicks)
- tick step (QCPAxis::setTickStep)
- auto tick step (QCPAxis::setAutoTickStep)
- number format (QCPAxis::setNumberFormat)
- number precision (QCPAxis::setNumberPrecision)
- tick label type (QCPAxis::setTickLabelType)
- date time format (QCPAxis::setDateTimeFormat)
- date time spec (QCPAxis::setDateTimeSpec)

Tick labels (QCPAxis::setTickLabels) of the right and top axes are set to false.

If *connectRanges* is true, the rangeChanged signals of the bottom and left axes are connected to the QCPAxis::setRange slots of the top and right axes.

7.28.3.38 QSize QCPAxisRect::size ( ) const [inline]

Returns the pixel size of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.39 int QCPAxisRect::top ( ) const [inline]

Returns the pixel position of the top border of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.40 QPoint QCPAxisRect::topLeft ( ) const [inline]

Returns the top left corner of this axis rect in pixels. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.41 QPoint QCPAxisRect::topRight ( ) const [inline]

Returns the top right corner of this axis rect in pixels. Margins are not taken into account here, so the returned value is with respect to the inner rect.

7.28.3.42 void QCPAxisRect::update (UpdatePhase *phase*) [virtual]

This method is called automatically upon replot and doesn't need to be called by users of QCPAxisRect.

Calls the base class implementation to update the margins (see QCPLayoutElement::update), and finally passes the rect to the inset layout (insetLayout) and calls its QCP-InsetLayout::update function.

Reimplemented from QCPLayoutElement.

7.28.3.43 void QCPAxisRect::wheelEvent (QWheelEvent ∗ *event*) [protected, virtual]

This event is called, if the mouse wheel is scrolled while the cursor is inside the rect of this layout element.

Reimplemented from QCPLayoutElement.

7.28.3.44 int QCPAxisRect::width ( ) const [inline]

Returns the pixel width of this axis rect. Margins are not taken into account here, so the returned value is with respect to the inner rect.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.29 QCPBarData Class Reference

Holds the data of one single data point (one bar) for QCPBars.

Public Member Functions

- QCPBarData ()
- QCPBarData (double key, double value)

Public Attributes

- double **key**
- double **value**

7.29.1 Detailed Description

Holds the data of one single data point (one bar) for QCPBars.

The container for storing multiple data points is QCPBarDataMap.

The stored data is:

- *key:* coordinate on the key axis of this bar
- *value:* height coordinate on the value axis of this bar

See also

QCPBarDataaMap

7.29.2 Constructor & Destructor Documentation

7.29.2.1 QCPBarData::QCPBarData ( )

Constructs a bar data point with key and value set to zero.

7.29.2.2 QCPBarData::QCPBarData (double *key*, double *value*)

Constructs a bar data point with the specified *key* and *value*.

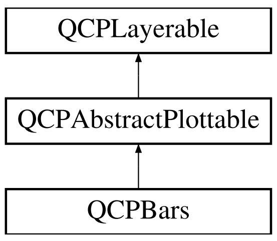
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.30 QCPBars Class Reference

A plottable representing a bar chart in a plot.

Inheritance diagram for QCPBars:



Public Types

- enum WidthType { wtAbsolute, wtAxisRectRatio, wtPlotCoords }

Public Member Functions

- QCPBars (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- double **width** () const
- WidthType **widthType** () const
- QCPBarsGroup ∗ **barsGroup** () const
- double **baseValue** () const
- QCPBars ∗ barBelow () const
- QCPBars ∗ barAbove () const
- QCPBarDataMap ∗ **data** () const
- void setWidth (double width)
- void setWidthType (WidthType widthType)
- void setBarsGroup (QCPBarsGroup ∗barsGroup)
- void setBaseValue (double baseValue)
- void setData (QCPBarDataMap ∗data, bool copy=false)
- void setData (const QVector< double > &key, const QVector< double > &value)
- void moveBelow (QCPBars ∗bars)
- void moveAbove (QCPBars ∗bars)
- void addData (const QCPBarDataMap &dataMap)
- void addData (const QCPBarData &data)
- void addData (double key, double value)
- void addData (const QVector< double > &keys, const QVector< double > &values)
- void removeDataBefore (double key)
- void removeDataAfter (double key)
- void removeData (double fromKey, double toKey)
- void removeData (double key)
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- void **getVisibleDataBounds** (QCPBarDataMap::const_iterator &lower, QCPBarDataMap::const_iterator &upperEnd) const
- QPolygonF **getBarPolygon** (double key, double value) const
- void **getPixelWidth** (double key, double &lower, double &upper) const
- double **getStackedBaseValue** (double key, bool positive) const

Static Protected Member Functions

- static void **connectBars** (QCPBars ∗lower, QCPBars ∗upper)

Protected Attributes

- QCPBarDataMap ∗ **mData**
- double **mWidth**
- WidthType **mWidthType**
- QCPBarsGroup ∗ **mBarsGroup**
- double **mBaseValue**
- QPointer< QCPBars > **mBarBelow**
- QPointer< QCPBars > **mBarAbove**

Friends

- class **QCustomPlot**
- class **QCPLegend**
- class **QCPBarsGroup**

7.30.1 Detailed Description

A plottable representing a bar chart in a plot.

To plot data, assign it with the setData or addData functions.

7.30.2 Changing the appearance

The appearance of the bars is determined by the pen and the brush (setPen, setBrush). The width of the individual bars can be controlled with setWidthType and setWidth.

Bar charts are stackable. This means, two QCPBars plottables can be placed on top of each other (see QCPBars::moveAbove). So when two bars are at the same key position, they will appear stacked.

If you would like to group multiple QCPBars plottables together so they appear side by side as shown below, use QCPBarsGroup.

7.30.3 Usage

Like all data representing objects in QCustomPlot, the QCPBars is a plottable (QCP-AbstractPlottable). So the plottable-interface of QCustomPlot applies (QCustomPlot-::plottable, QCustomPlot::addPlottable, QCustomPlot::removePlottable, etc.)

Usually, you first create an instance:

add it to the m_customPlot with QCustomPlot::addPlottable:

and then modify the properties of the newly created plottable, e.g.:

7.30.4 Member Enumeration Documentation

7.30.4.1 enum QCPBars::WidthType

Defines the ways the width of the bar can be specified. Thus it defines what the number passed to setWidth actually means.

See also

setWidthType, setWidth

Enumerator:

wtAbsolute Bar width is in absolute pixels.

wtAxisRectRatio Bar width is given by a fraction of the axis rect size.

wtPlotCoords Bar width is in key coordinates and thus scales with the key axis range.

7.30.5 Constructor & Destructor Documentation

7.30.5.1 QCPBars::QCPBars (QCPAxis ∗ *keyAxis,* QCPAxis ∗ *valueAxis*) [explicit]

Constructs a bar chart which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and not have the same orientation. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

The constructed QCPBars can be added to the plot with QCustomPlot::addPlottable, QCustomPlot then takes ownership of the bar chart.

7.30.6 Member Function Documentation

7.30.6.1 void QCPBars::addData (const QCPBarDataMap & *dataMap*)

Adds the provided data points in *dataMap* to the current data.

See also

removeData

7.30.6.2 void QCPBars::addData (const QCPBarData & *data*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point in *data* to the current data.

See also

removeData

7.30.6.3 void QCPBars::addData (double *key,* double *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point as *key* and *value* tuple to the current data

See also

removeData

7.30.6.4 void QCPBars::addData (const QVector< double > & *keys,* const QVector< double > & *values*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided data points as *key* and *value* tuples to the current data.

See also

removeData

7.30.6.5 QCPBars ∗ QCPBars::barAbove ( ) const [inline]

Returns the bars plottable that is directly above this bars plottable. If there is no such plottable, returns 0.

See also

barBelow, moveBelow, moveAbove

7.30.6.6 QCPBars ∗ QCPBars::barBelow ( ) const [inline]

Returns the bars plottable that is directly below this bars plottable. If there is no such plottable, returns 0.

See also

barAbove, moveBelow, moveAbove

7.30.6.7 void QCPBars::clearData ( ) [virtual]

Removes all data points.

See also

removeData, removeDataAfter, removeDataBefore

Implements QCPAbstractPlottable.

7.30.6.8 void QCPBars::moveAbove (QCPBars ∗ *bars*)

Moves this bars plottable above *bars*. In other words, the bars of this plottable will appear above the bars of *bars*. The move target *bars* must use the same key and value axis as this plottable.

Inserting into and removing from existing bar stacking is handled gracefully. If *bars* already has a bars object above itself, this bars object is inserted between the two. If this bars object is already between two other bars, the two other bars will be stacked on top of each other after the operation.

To remove this bars plottable from any stacking, set *bars* to 0.

See also

moveBelow, barBelow, barAbove

7.30.6.9 void QCPBars::moveBelow (QCPBars ∗ *bars*)

Moves this bars plottable below *bars*. In other words, the bars of this plottable will appear below the bars of *bars*. The move target *bars* must use the same key and value axis as this plottable.

Inserting into and removing from existing bar stacking is handled gracefully. If *bars* already has a bars object below itself, this bars object is inserted between the two. If this bars object is already between two other bars, the two other bars will be stacked on top of each other after the operation.

To remove this bars plottable from any stacking, set *bars* to 0.

See also

moveBelow, barAbove, barBelow

7.30.6.10 void QCPBars::removeData (double *fromKey,* double *toKey*)

Removes all data points with key between *fromKey* and *toKey*. if *fromKey* is greater or equal to *toKey*, the function does nothing. To remove a single data point with known key, use removeData(double key).

See also

addData, clearData

7.30.6.11 void QCPBars::removeData (double *key*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes a single data point at *key*. If the position is not known with absolute precision, consider using removeData(double fromKey, double toKey) with a small fuzziness interval around the suspected position, depeding on the precision with which the key is known.

See also

addData, clearData

7.30.6.12 void QCPBars::removeDataAfter (double *key*)

Removes all data points with key greater than *key*.

See also

addData, clearData

7.30.6.13 void QCPBars::removeDataBefore (double *key*)

Removes all data points with key smaller than *key*.

See also

addData, clearData

7.30.6.14 double QCPBars::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.30.6.15 void QCPBars::setBarsGroup (QCPBarsGroup ∗ *barsGroup*)

Sets to which QCPBarsGroup this QCPBars instance belongs to. Alternatively, you can also use QCPBarsGroup::append.

To remove this QCPBars from any group, set *barsGroup* to 0.

7.30.6.16 void QCPBars::setBaseValue (double *baseValue*)

Sets the base value of this bars plottable.

The base value defines where on the value coordinate the bars start. How far the bars extend from the base value is given by their individual value data. For example, if the base value is set to 1, a bar with data value 2 will have its lowest point at value coordinate 1 and highest point at 3.

For stacked bars, only the base value of the bottom-most QCPBars has meaning.

The default base value is 0.

7.30.6.17 void QCPBars::setData (QCPBarDataMap ∗ *data*, bool *copy* = `false`)

Replaces the current data with the provided *data*.

If *copy* is set to true, data points in *data* will only be copied. if false, the plottable takes ownership of the passed data and replaces the internal data pointer with it. This is significantly faster than copying for large datasets.

7.30.6.18 void QCPBars::setData (const QVector< double > & *key*, const QVector< double > & *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *key* and *value* tuples. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.30.6.19 void QCPBars::setWidth (double *width*)

Sets the width of the bars.

How the number passed as *width* is interpreted (e.g. screen pixels, plot coordinates,...), depends on the currently set width type, see setWidthType and WidthType.

7.30.6.20 void QCPBars::setWidthType (QCPBars::WidthType *widthType*)

Sets how the width of the bars is defined. See the documentation of WidthType for an explanation of the possible values for *widthType*.

The default value is wtPlotCoords.

See also

setWidth

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.31 QCPBarsGroup Class Reference

Groups multiple QCPBars together so they appear side by side.

Public Types

- enum SpacingType { stAbsolute, stAxisRectRatio, stPlotCoords }

Public Member Functions

- QCPBarsGroup (QCustomPlot ∗parentPlot)
- SpacingType **spacingType** () const
- double **spacing** () const
- void setSpacingType (SpacingType spacingType)
- void setSpacing (double spacing)
- QList< QCPBars ∗ > bars () const
- QCPBars ∗ bars (int index) const
- int size () const
- bool isEmpty () const
- void clear ()
- bool contains (QCPBars ∗bars) const
- void append (QCPBars ∗bars)
- void insert (int i, QCPBars ∗bars)
- void remove (QCPBars ∗bars)

Protected Member Functions

- void **registerBars** (QCPBars ∗bars)
- void **unregisterBars** (QCPBars ∗bars)
- double **keyPixelOffset** (const QCPBars ∗bars, double keyCoord)
- double **getPixelSpacing** (const QCPBars ∗bars, double keyCoord)

Protected Attributes

- QCustomPlot ∗ **mParentPlot**
- SpacingType **mSpacingType**
- double **mSpacing**
- QList< QCPBars ∗ > **mBars**

Friends

- class **QCPBars**

7.31.1 Detailed Description

Groups multiple QCPBars together so they appear side by side.

When showing multiple QCPBars in one plot which have bars at identical keys, it may be desirable to have them appearing next to each other at each key. This is what adding the respective QCPBars plottables to a QCPBarsGroup achieves. (An alternative approach is to stack them on top of each other, see QCPBars::moveAbove.)

7.31.2 Usage

To add a QCPBars plottable to the group, create a new group and then add the respective bars intances:

Alternatively to appending to the group like shown above, you can also set the group on the QCPBars plottable via QCPBars::setBarsGroup.

The spacing between the bars can be configured via setSpacingType and setSpacing. The bars in this group appear in the plot in the order they were appended. To insert a bars plottable at a certain index position, or to reposition a bars plottable which is already in the group, use insert.

To remove specific bars from the group, use either remove or call QCPBars::setBarsGroup(0) on the respective bars plottable.

To clear the entire group, call clear, or simply delete the group.

7.31.3 Example

The image above is generated with the following code:

7.31.4 Member Enumeration Documentation

7.31.4.1 enum QCPBarsGroup::SpacingType

Defines the ways the spacing between bars in the group can be specified. Thus it defines what the number passed to setSpacing actually means.

See also

setSpacingType, setSpacing

Enumerator:

stAbsolute Bar spacing is in absolute pixels.

stAxisRectRatio Bar spacing is given by a fraction of the axis rect size.

stPlotCoords Bar spacing is in key coordinates and thus scales with the key axis range.

7.31.5 Constructor & Destructor Documentation

7.31.5.1 QCPBarsGroup::QCPBarsGroup (QCustomPlot ∗ *parentPlot*)

Constructs a new bars group for the specified QCustomPlot instance.

7.31.6 Member Function Documentation

7.31.6.1 void **QCPBarsGroup::append** (**QCPBars** ∗ *bars*)

Adds the specified *bars* plottable to this group. Alternatively, you can also use QCP-Bars::setBarsGroup on the *bars* instance.

See also

insert, remove

7.31.6.2 QList< **QCPBars** ∗ > **QCPBarsGroup::bars** ( ) const [inline]

Returns all bars currently in this group.

See also

bars(int index)

7.31.6.3 **QCPBars** ∗ **QCPBarsGroup::bars** (int *index*) const

Returns the QCPBars instance with the specified *index* in this group. If no such QCP-Bars exists, returns 0.

See also

bars(), size

7.31.6.4 void **QCPBarsGroup::clear** ( )

Removes all QCPBars plottables from this group.

See also

isEmpty

7.31.6.5 bool **QCPBarsGroup::contains** (**QCPBars** ∗ *bars*) const [inline]

Returns whether the specified *bars* plottable is part of this group.

7.31.6.6 void **QCPBarsGroup::insert** (int *i*, **QCPBars** ∗ *bars*)

Inserts the specified *bars* plottable into this group at the specified index position *i*. This gives you full control over the ordering of the bars.

bars may already be part of this group. In that case, *bars* is just moved to the new index position.

See also

append, remove

7.31.6.7 bool QCPBarsGroup::isEmpty ( ) const [inline]

Returns whether this bars group is empty.

See also

size

7.31.6.8 void QCPBarsGroup::remove (QCPBars ∗ *bars*)

Removes the specified *bars* plottable from this group.

See also

contains, clear

7.31.6.9 void QCPBarsGroup::setSpacing (double *spacing*)

Sets the spacing between adjacent bars. What the number passed as *spacing* actually means, is defined by the current SpacingType, which can be set with setSpacingType.

See also

setSpacingType

7.31.6.10 void QCPBarsGroup::setSpacingType (SpacingType *spacingType*)

Sets how the spacing between adjacent bars is interpreted. See SpacingType.

The actual spacing can then be specified with setSpacing.

See also

setSpacing

7.31.6.11 int QCPBarsGroup::size ( ) const [inline]

Returns the number of QCPBars plottables that are part of this group.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.32 QCPColorGradient Class Reference

Defines a color gradient for use with e.g. QCPColorMap.

Public Types

- enum ColorInterpolation { ciRGB, ciHSV }
- enum GradientPreset { gpGrayscale, gpHot, gpCold, gpNight, gpCandy, gpGeography, gpIon, gpThermal, gpPolar, gpSpectrum, gpJet, gpHues }

Public Member Functions

- QCPColorGradient (GradientPreset preset=gpCold)
- bool **operator==** (const QCPColorGradient &other) const
- bool **operator!=** (const QCPColorGradient &other) const
- int **levelCount** () const
- QMap< double, QColor > **colorStops** () const
- ColorInterpolation **colorInterpolation** () const
- bool **periodic** () const
- void setLevelCount (int n)
- void setColorStops (const QMap< double, QColor > &colorStops)
- void setColorStopAt (double position, const QColor &color)
- void setColorInterpolation (ColorInterpolation interpolation)
- void setPeriodic (bool enabled)
- void colorize (const double *data, const QCPRange &range, QRgb *scanLine, int n, int dataIndexFactor=1, bool logarithmic=false)
- QRgb **color** (double position, const QCPRange &range, bool logarithmic=false)
- void loadPreset (GradientPreset preset)
- void clearColorStops ()
- QCPColorGradient inverted () const

Protected Member Functions

- void **updateColorBuffer** ()

Protected Attributes

- int **mLevelCount**
- QMap< double, QColor > **mColorStops**
- ColorInterpolation **mColorInterpolation**
- bool **mPeriodic**
- QVector< QRgb > **mColorBuffer**
- bool **mColorBufferInvalidated**

7.32.1 Detailed Description

Defines a color gradient for use with e.g. QCPColorMap.

This class describes a color gradient which can be used to encode data with color. For example, QCPColorMap and QCPColorScale have setGradient methods which take an instance of this class. Colors are set with setColorStopAt(double position, const QColor &color) with a *position* from 0 to 1. In between these defined color positions, the color will be interpolated linearly either in RGB or HSV space, see setColorInterpolation.

Alternatively, load one of the preset color gradients shown in the image below, with loadPreset, or by directly specifying the preset in the constructor.

The fact that the QCPColorGradient(GradientPreset preset) constructor allows directly converting a GradientPreset to a QCPColorGradient, you can also directly pass -GradientPreset to all the *setGradient* methods, e.g.:

The total number of levels used in the gradient can be set with setLevelCount. Whether the color gradient shall be applied periodically (wrapping around) to data values that lie outside the data range specified on the plottable instance can be controlled with setPeriodic.

7.32.2 Member Enumeration Documentation

7.32.2.1 enum QCPColorGradient::ColorInterpolation

Defines the color spaces in which color interpolation between gradient stops can be performed.

See also

setColorInterpolation

Enumerator:

ciRGB Color channels red, green and blue are linearly interpolated.

ciHSV Color channels hue, saturation and value are linearly interpolated (The hue is interpolated over the shortest angle distance)

7.32.2.2 enum QCPColorGradient::GradientPreset

Defines the available presets that can be loaded with loadPreset. See the documentation there for an image of the presets.

Enumerator:

gpGrayscale Continuous lightness from black to white (suited for non-biased data representation)

gpHot Continuous lightness from black over firey colors to white (suited for non-biased data representation)

gpCold Continuous lightness from black over icey colors to white (suited for non-biased data representation)

gpNight Continuous lightness from black over weak blueish colors to white (suited for non-biased data representation)

gpCandy Blue over pink to white.

gpGeography Colors suitable to represent different elevations on geographical maps.

gpIon Half hue spectrum from black over purple to blue and finally green (creates banding illusion but allows more precise magnitude estimates)

gpThermal Colors suitable for thermal imaging, ranging from dark blue over purple to orange, yellow and white.

gpPolar Colors suitable to emphasize polarity around the center, with blue for negative, black in the middle and red for positive values.

gpSpectrum An approximation of the visible light spectrum (creates banding illusion but allows more precise magnitude estimates)

gpJet Hue variation similar to a spectrum, often used in numerical visualization (creates banding illusion but allows more precise magnitude estimates)

gpHues Full hue cycle, with highest and lowest color red (suitable for periodic data, such as angles and phases, see setPeriodic)

7.32.3 Constructor & Destructor Documentation

7.32.3.1 QCPColorGradient::QCPColorGradient (GradientPreset *preset* = gpCold)

Constructs a new QCPColorGradient initialized with the colors and color interpolation according to *preset*.

The color level count is initialized to 350.

7.32.4 Member Function Documentation

7.32.4.1 void QCPColorGradient::clearColorStops ( )

Clears all color stops.

See also

setColorStops, setColorStopAt

7.32.4.2 void QCPColorGradient::colorize (const double ∗ *data*, const QCPRange & *range*, QRgb ∗ *scanLine*, int *n*, int *dataIndexFactor* = `1`, bool *logarithmic* = `false`)

This method is used to quickly convert a *data* array to colors. The colors will be output in the array *scanLine*. Both *data* and *scanLine* must have the length *n* when passed to this function. The data range that shall be used for mapping the data value to the gradient is passed in *range*. *logarithmic* indicates whether the data values shall be mapped to colors logarithmically.

if *data* actually contains 2D-data linearized via `[row*columnCount + column]`, you can set *dataIndexFactor* to `columnCount` to convert a column instead of a row of the data array, in *scanLine*. *scanLine* will remain a regular (1D) array. This works because *data* is addressed `data[i*dataIndexFactor]`.

7.32.4.3 QCPColorGradient QCPColorGradient::inverted ( ) const

Returns an inverted gradient. The inverted gradient has all properties as this QCPColorGradient, but the order of the color stops is inverted.

See also

setColorStops, setColorStopAt

7.32.4.4 void QCPColorGradient::loadPreset (GradientPreset *preset*)

Clears the current color stops and loads the specified *preset*. A preset consists of predefined color stops and the corresponding color interpolation method.

The available presets are:

7.32.4.5 void QCPColorGradient::setColorInterpolation (QCPColorGradient::ColorInterpolation *interpolation*)

Sets whether the colors in between the configured color stops (see setColorStopAt) shall be interpolated linearly in RGB or in HSV color space.

For example, a sweep in RGB space from red to green will have a muddy brown intermediate color, whereas in HSV space the intermediate color is yellow.

7.32.4.6 void QCPColorGradient::setColorStopAt (double *position*, const QColor & *color*)

Sets the *color* the gradient will have at the specified *position* (from 0 to 1). In between these color stops, the color is interpolated according to setColorInterpolation.

See also

setColorStops, clearColorStops

7.32.4.7 void QCPColorGradient::setColorStops (const QMap< double, QColor > & *colorStops*)

Sets at which positions from 0 to 1 which color shall occur. The positions are the keys, the colors are the values of the passed QMap *colorStops*. In between these color stops, the color is interpolated according to setColorInterpolation.

A more convenient way to create a custom gradient may be to clear all color stops with clearColorStops and then adding them one by one with setColorStopAt.

See also

clearColorStops

7.32.4.8 void QCPColorGradient::setLevelCount (int *n*)

Sets the number of discretization levels of the color gradient to *n*. The default is 350 which is typically enough to create a smooth appearance.

7.32.4.9 void QCPColorGradient::setPeriodic (bool *enabled*)

Sets whether data points that are outside the configured data range (e.g. QCPColorMap::setDataRange) are colored by periodically repeating the color gradient or whether they all have the same color, corresponding to the respective gradient boundary color.

As shown in the image above, gradients that have the same start and end color are especially suitable for a periodic gradient mapping, since they produce smooth color transitions throughout the color map. A preset that has this property is gpHues.

In practice, using periodic color gradients makes sense when the data corresponds to a periodic dimension, such as an angle or a phase. If this is not the case, the color encoding might become ambiguous, because multiple different data values are shown as the same color.

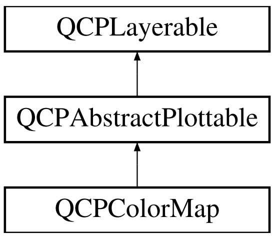
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.33 QCPColorMap Class Reference

A plottable representing a two-dimensional color map in a plot.

Inheritance diagram for QCPColorMap:



Signals

- void dataRangeChanged (QCPRange newRange)
- void dataScaleTypeChanged (QCPAxis::ScaleType scaleType)
- void gradientChanged (QCPColorGradient newGradient)

Public Member Functions

- QCPColorMap (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- QCPColorMapData ∗ data () const
- QCPRange **dataRange** () const
- QCPAxis::ScaleType **dataScaleType** () const
- bool **interpolate** () const
- bool **tightBoundary** () const
- QCPColorGradient **gradient** () const
- QCPColorScale ∗ **colorScale** () const
- void setData (QCPColorMapData ∗data, bool copy=false)
- Q_SLOT void setDataRange (const QCPRange &dataRange)
- Q_SLOT void setDataScaleType (QCPAxis::ScaleType scaleType)
- Q_SLOT void setGradient (const QCPColorGradient &gradient)
- void setInterpolate (bool enabled)
- void setTightBoundary (bool enabled)
- void setColorScale (QCPColorScale ∗colorScale)
- void rescaleDataRange (bool recalculateDataBounds=false)
- Q_SLOT void updateLegendIcon (Qt::TransformationMode transformMode=Qt::-SmoothTransformation, const QSize &thumbSize=QSize(32, 18))
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual void **updateMapImage** ()
- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSign-Domain=sdBoth) const

- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const

Protected Attributes

- QCPRange **mDataRange**
- QCPAxis::ScaleType **mDataScaleType**
- QCPColorMapData $*$ **mMapData**
- QCPColorGradient **mGradient**
- bool **mInterpolate**
- bool **mTightBoundary**
- QPointer$<$ QCPColorScale $>$ **mColorScale**
- QImage **mMapImage**
- QImage **mUndersampledMapImage**
- QPixmap **mLegendIcon**
- bool **mMapImageInvalidated**

Friends

- class **QCustomPlot**
- class **QCPLegend**

7.33.1 Detailed Description

A plottable representing a two-dimensional color map in a plot.

The data is stored in the class QCPColorMapData, which can be accessed via the data() method.

A color map has three dimensions to represent a data point: The *key* dimension, the *value* dimension and the *data* dimension. As with other plottables such as graphs, *key* and *value* correspond to two orthogonal axes on the QCustomPlot surface that you specify in the QCPColorMap constructor. The *data* dimension however is encoded as the color of the point at (*key*, *value*).

Set the number of points (or *cells*) in the key/value dimension via QCPColorMapData::setSize. The plot coordinate range over which these points will be displayed is specified via QCPColorMapData::setRange. The first cell will be centered on the lower range boundary and the last cell will be centered on the upper range boundary. The data can be set by either accessing the cells directly with QCPColorMapData::setCell or by addressing the cells via their plot coordinates with QCPColorMapData::setData. If possible, you should prefer setCell, since it doesn't need to do any coordinate transformation and thus performs a bit better.

The cell with index (0, 0) is at the bottom left, if the color map uses normal (i.e. not reversed) key and value axes.

To show the user which colors correspond to which *data* values, a QCPColorScale is typically placed to the right of the axis rect. See the documentation there for details on how to add and use a color scale.

7.33.2 Changing the appearance

The central part of the appearance is the color gradient, which can be specified via setGradient. See the documentation of QCPColorGradient for details on configuring a color gradient.

The *data* range that is mapped to the colors of the gradient can be specified with setDataRange. To make the data range encompass the whole data set minimum to maximum, call rescaleDataRange.

7.33.3 Usage

Like all data representing objects in QCustomPlot, the QCPColorMap is a plottable (QCPAbstractPlottable). So the plottable-interface of QCustomPlot applies (QCustomPlot::plottable, QCustomPlot::addPlottable, QCustomPlot::removePlottable, etc.)

Usually, you first create an instance and add it to the m_customPlot:

and then modify the properties of the newly created color map, e.g.:

Note

The QCPColorMap always displays the data at equal key/value intervals, even if the key or value axis is set to a logarithmic scaling. If you want to use QCPColorMap with logarithmic axes, you shouldn't use the QCPColorMapData::setData method as it uses a linear transformation to determine the cell index. Rather directly access the cell index with QCPColorMapData::setCell.

7.33.4 Constructor & Destructor Documentation

7.33.4.1 QCPColorMap::QCPColorMap (QCPAxis ∗ *keyAxis,* QCPAxis ∗ *valueAxis*) `[explicit]`

Constructs a color map with the specified *keyAxis* and *valueAxis*.

The constructed QCPColorMap can be added to the plot with QCustomPlot::addPlottable, QCustomPlot then takes ownership of the color map.

7.33.5 Member Function Documentation

7.33.5.1 void QCPColorMap::clearData ( ) `[virtual]`

Clears the colormap data by calling QCPColorMapData::clear() on the internal data. This also resizes the map to 0x0 cells.

Implements QCPAbstractPlottable.

7.33.5.2 QCPColorMapData ∗ QCPColorMap::data ( ) const [inline]

Returns a pointer to the internal data storage of type QCPColorMapData. Access this to modify data points (cells) and the color map key/value range.

See also

setData

7.33.5.3 void QCPColorMap::dataRangeChanged (QCPRange *newRange*) [signal]

This signal is emitted when the data range changes.

See also

setDataRange

7.33.5.4 void QCPColorMap::dataScaleTypeChanged (QCPAxis::ScaleType *scaleType*) [signal]

This signal is emitted when the data scale type changes.

See also

setDataScaleType

7.33.5.5 void QCPColorMap::gradientChanged (QCPColorGradient *newGradient*) [signal]

This signal is emitted when the gradient changes.

See also

setGradient

7.33.5.6 void QCPColorMap::rescaleDataRange (bool *recalculateDataBounds* = false)

Sets the data range (setDataRange) to span the minimum and maximum values that occur in the current data set. This corresponds to the rescaleKeyAxis or rescaleValueAxis methods, only for the third data dimension of the color map.

The minimum and maximum values of the data set are buffered in the internal QCPColorMapData instance (data). As data is updated via its QCPColorMapData::setCell or QCPColorMapData::setData, the buffered minimum and maximum values are updated, too. For performance reasons, however, they are only updated in an expanding fashion. So the buffered maximum can only increase and the buffered minimum can only decrease. In consequence, changes to the data that actually lower the maximum of the data set (by overwriting the cell holding the current maximum with a smaller value), aren't recognized and the buffered maximum overestimates the true maximum of the data set. The same happens for the buffered minimum. To recalculate the true minimum and maximum by explicitly looking at each cell, the method QCPColorMap-Data::recalculateDataBounds can be used. For convenience, setting the parameter *recalculateDataBounds* calls this method before setting the data range to the buffered minimum and maximum.

See also

setDataRange

7.33.5.7 double QCPColorMap::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.33.5.8 void QCPColorMap::setColorScale (QCPColorScale ∗ *colorScale*)

Associates the color scale *colorScale* with this color map.

This means that both the color scale and the color map synchronize their gradient, data range and data scale type (setGradient, setDataRange, setDataScaleType). Multiple color maps can be associated with one single color scale. This causes the color maps to also synchronize those properties, via the mutual color scale.

This function causes the color map to adopt the current color gradient, data range and data scale type of *colorScale*. After this call, you may change these properties at either the color map or the color scale, and the setting will be applied to both.

Pass 0 as *colorScale* to disconnect the color scale from this color map again.

7.33.5.9 void QCPColorMap::setData (QCPColorMapData ∗ *data*, bool *copy* = `false`)

Replaces the current data with the provided *data*.

If *copy* is set to true, the *data* object will only be copied. if false, the color map takes ownership of the passed data and replaces the internal data pointer with it. This is significantly faster than copying for large datasets.

7.33.5.10 void QCPColorMap::setDataRange (const QCPRange & *dataRange*)

Sets the data range of this color map to *dataRange*. The data range defines which data values are mapped to the color gradient.

To make the data range span the full range of the data set, use rescaleDataRange.

See also

QCPColorScale::setDataRange

7.33.5.11 void QCPColorMap::setDataScaleType (QCPAxis::ScaleType *scaleType*)

Sets whether the data is correlated with the color gradient linearly or logarithmically.

See also

QCPColorScale::setDataScaleType

7.33.5.12 void QCPColorMap::setGradient (const QCPColorGradient & *gradient*)

Sets the color gradient that is used to represent the data. For more details on how to create an own gradient or use one of the preset gradients, see QCPColorGradient.

The colors defined by the gradient will be used to represent data values in the currently set data range, see setDataRange. Data points that are outside this data range will either be colored uniformly with the respective gradient boundary color, or the gradient will repeat, depending on QCPColorGradient::setPeriodic.

See also

QCPColorScale::setGradient

7.33.5.13 void QCPColorMap::setInterpolate (bool *enabled*)

Sets whether the color map image shall use bicubic interpolation when displaying the color map shrinked or expanded, and not at a 1:1 pixel-to-data scale.

7.33.5.14 void QCPColorMap::setTightBoundary (bool *enabled*)

Sets whether the outer most data rows and columns are clipped to the specified key and value range (see QCPColorMapData::setKeyRange, QCPColorMapData::setValue-Range).

if *enabled* is set to false, the data points at the border of the color map are drawn with the same width and height as all other data points. Since the data points are represented by rectangles of one color centered on the data coordinate, this means that the shown color map extends by half a data point over the specified key/value range in each direction.

7.33.5.15 void QCPColorMap::updateLegendIcon (Qt::TransformationMode *transformMode* = `Qt::SmoothTransformation`, const QSize & *thumbSize* = `QSize(32, 18)`)

Takes the current appearance of the color map and updates the legend icon, which is used to represent this color map in the legend (see QCPLegend).

The *transformMode* specifies whether the rescaling is done by a faster, low quality image scaling algorithm (Qt::FastTransformation) or by a slower, higher quality algorithm (Qt::SmoothTransformation).

The current color map appearance is scaled down to *thumbSize*. Ideally, this should be equal to the size of the legend icon (see QCPLegend::setIconSize). If it isn't exactly the configured legend icon size, the thumb will be rescaled during drawing of the legend item.

See also

setDataRange

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.34 QCPColorMapData Class Reference

Holds the two-dimensional data of a QCPColorMap plottable.

Public Member Functions

- QCPColorMapData (int keySize, int valueSize, const QCPRange &keyRange, const QCPRange &valueRange)
- QCPColorMapData (const QCPColorMapData &other)
- QCPColorMapData & operator= (const QCPColorMapData &other)
- int **keySize** () const
- int **valueSize** () const
- QCPRange **keyRange** () const
- QCPRange **valueRange** () const
- QCPRange **dataBounds** () const
- double **data** (double key, double value)
- double **cell** (int keyIndex, int valueIndex)
- void setSize (int keySize, int valueSize)
- void setKeySize (int keySize)
- void setValueSize (int valueSize)
- void setRange (const QCPRange &keyRange, const QCPRange &valueRange)
- void setKeyRange (const QCPRange &keyRange)
- void setValueRange (const QCPRange &valueRange)
- void setData (double key, double value, double z)
- void setCell (int keyIndex, int valueIndex, double z)
- void recalculateDataBounds ()
- void clear ()
- void fill (double z)
- bool isEmpty () const
- void coordToCell (double key, double value, int ∗keyIndex, int ∗valueIndex) const
- void cellToCoord (int keyIndex, int valueIndex, double ∗key, double ∗value) const

Protected Attributes

- int **mKeySize**
- int **mValueSize**
- QCPRange **mKeyRange**
- QCPRange **mValueRange**
- bool **mIsEmpty**
- double $*$ **mData**
- QCPRange **mDataBounds**
- bool **mDataModified**

Friends

- class **QCPColorMap**

7.34.1 Detailed Description

Holds the two-dimensional data of a QCPColorMap plottable.

This class is a data storage for QCPColorMap. It holds a two-dimensional array, which QCPColorMap then displays as a 2D image in the plot, where the array values are represented by a color, depending on the value.

The size of the array can be controlled via setSize (or setKeySize, setValueSize). - Which plot coordinates these cells correspond to can be configured with setRange (or setKeyRange, setValueRange).

The data cells can be accessed in two ways: They can be directly addressed by an integer index with setCell. This is the fastest method. Alternatively, they can be addressed by their plot coordinate with setData. plot coordinate to cell index transformations and vice versa are provided by the functions coordToCell and cellToCoord.

This class also buffers the minimum and maximum values that are in the data set, to provide QCPColorMap::rescaleDataRange with the necessary information quickly. Setting a cell to a value that is greater than the current maximum increases this maximum to the new value. However, setting the cell that currently holds the maximum value to a smaller value doesn't decrease the maximum again, because finding the true new maximum would require going through the entire data array, which might be time consuming. The same holds for the data minimum. This functionality is given by recalculateDataBounds, such that you can decide when it is sensible to find the true current minimum and maximum. The method QCPColorMap::rescaleDataRange offers a convenience parameter *recalculateDataBounds* which may be set to true to automatically call recalculateDataBounds internally.

7.34.2 Constructor & Destructor Documentation

7.34.2.1 QCPColorMapData::QCPColorMapData (int *keySize*, int *valueSize*, const QCPRange & *keyRange*, const QCPRange & *valueRange*)

Constructs a new QCPColorMapData instance. The instance has *keySize* cells in the key direction and *valueSize* cells in the value direction. These cells will be displayed by the QCPColorMap at the coordinates *keyRange* and *valueRange*.

See also

setSize, setKeySize, setValueSize, setRange, setKeyRange, setValueRange

7.34.2.2 QCPColorMapData::QCPColorMapData (const QCPColorMapData & *other*)

Constructs a new QCPColorMapData instance copying the data and range of *other*.

7.34.3 Member Function Documentation

7.34.3.1 void QCPColorMapData::cellToCoord (int *keyIndex*, int *valueIndex*, double ∗ *key*, double ∗ *value*) const

Transforms cell indices given by *keyIndex* and *valueIndex* to cell indices of this QCP-ColorMapData instance. The resulting coordinates are returned via the output parameters *key* and *value*.

If you are only interested in a key or value coordinate, you may pass 0 as *key* or *value*.

Note

The QCPColorMap always displays the data at equal key/value intervals, even if the key or value axis is set to a logarithmic scaling. If you want to use QCPColor-Map with logarithmic axes, you shouldn't use the QCPColorMapData::cellToCoord method as it uses a linear transformation to determine the cell index.

See also

coordToCell, QCPAxis::pixelToCoord

7.34.3.2 void QCPColorMapData::clear ( )

Frees the internal data memory.

This is equivalent to calling setSize(0, 0).

7.34.3.3 void QCPColorMapData::coordToCell (double *key,* double *value,* int ∗ *keyIndex,* int ∗ *valueIndex*) const

Transforms plot coordinates given by *key* and *value* to cell indices of this QCPColorMapData instance. The resulting cell indices are returned via the output parameters *keyIndex* and *valueIndex*.

The retrieved key/value cell indices can then be used for example with setCell.

If you are only interested in a key or value index, you may pass 0 as *valueIndex* or *keyIndex*.

Note

The QCPColorMap always displays the data at equal key/value intervals, even if the key or value axis is set to a logarithmic scaling. If you want to use QCPColorMap with logarithmic axes, you shouldn't use the QCPColorMapData::coordToCell method as it uses a linear transformation to determine the cell index.

See also

cellToCoord, QCPAxis::coordToPixel

7.34.3.4 void QCPColorMapData::fill (double *z*)

Sets all cells to the value *z*.

7.34.3.5 bool QCPColorMapData::isEmpty ( ) const [inline]

Returns whether this instance carries no data. This is equivalent to having a size where at least one of the dimensions is 0 (see setSize).

7.34.3.6 QCPColorMapData & QCPColorMapData::operator= (const QCPColorMapData & *other*)

Overwrites this color map data instance with the data stored in *other*.

7.34.3.7 void QCPColorMapData::recalculateDataBounds ( )

Goes through the data and updates the buffered minimum and maximum data values.

Calling this method is only advised if you are about to call QCPColorMap::rescaleDataRange and can not guarantee that the cells holding the maximum or minimum data haven't been overwritten with a smaller or larger value respectively, since the buffered maximum/minimum values have been updated the last time. Why this is the case is explained in the class description (QCPColorMapData).

Note that the method QCPColorMap::rescaleDataRange provides a parameter *recalculateDataBounds* for convenience. Setting this to true will call this method for you, before doing the rescale.

7.34.3.8 void QCPColorMapData::setCell (int *keyIndex*, int *valueIndex*, double *z*)

Sets the data of the cell with indices *keyIndex* and *valueIndex* to *z*. The indices enumerate the cells starting from zero, up to the map's size-1 in the respective dimension (see setSize).

In the standard plot configuration (horizontal key axis and vertical value axis, both not range-reversed), the cell with indices (0, 0) is in the bottom left corner and the cell with indices (keySize-1, valueSize-1) is in the top right corner of the color map.

See also

setData, setSize

7.34.3.9 void QCPColorMapData::setData (double *key*, double *value*, double *z*)

Sets the data of the cell, which lies at the plot coordinates given by *key* and *value*, to *z*.

Note

The QCPColorMap always displays the data at equal key/value intervals, even if the key or value axis is set to a logarithmic scaling. If you want to use QCPColorMap with logarithmic axes, you shouldn't use the QCPColorMapData::setData method as it uses a linear transformation to determine the cell index. Rather directly access the cell index with QCPColorMapData::setCell.

See also

setCell, setRange

7.34.3.10 void QCPColorMapData::setKeyRange (const QCPRange & *keyRange*)

Sets the coordinate range the data shall be distributed over in the key dimension. - Together with the value range, This defines the rectangular area covered by the color map in plot coordinates.

The outer cells will be centered on the range boundaries given to this function. For example, if the key size (setKeySize) is 3 and *keyRange* is set to `QCPRange(2, 3)` there will be cells centered on the key coordinates 2, 2.5 and 3.

See also

setRange, setValueRange, setSize

7.34.3.11 void QCPColorMapData::setKeySize (int *keySize*)

Resizes the data array to have *keySize* cells in the key dimension.

The current data is discarded and the map cells are set to 0, unless the map had already the requested size.

Setting *keySize* to zero frees the internal data array and isEmpty returns true.

See also

setKeyRange, setSize, setValueSize

7.34.3.12 void QCPColorMapData::setRange (const QCPRange & *keyRange*, const QCPRange & *valueRange*)

Sets the coordinate ranges the data shall be distributed over. This defines the rectangular area covered by the color map in plot coordinates.

The outer cells will be centered on the range boundaries given to this function. For example, if the key size (setKeySize) is 3 and *keyRange* is set to `QCPRange(2, 3)` there will be cells centered on the key coordinates 2, 2.5 and 3.

See also

setSize

7.34.3.13 void QCPColorMapData::setSize (int *keySize*, int *valueSize*)

Resizes the data array to have *keySize* cells in the key dimension and *valueSize* cells in the value dimension.

The current data is discarded and the map cells are set to 0, unless the map had already the requested size.

Setting at least one of *keySize* or *valueSize* to zero frees the internal data array and isEmpty returns true.

See also

setRange, setKeySize, setValueSize

7.34.3.14 void QCPColorMapData::setValueRange (const QCPRange & *valueRange*)

Sets the coordinate range the data shall be distributed over in the value dimension. - Together with the key range, This defines the rectangular area covered by the color map in plot coordinates.

The outer cells will be centered on the range boundaries given to this function. For example, if the value size (setValueSize) is 3 and *valueRange* is set to `QCPRange(2, 3)` there will be cells centered on the value coordinates 2, 2.5 and 3.

See also

setRange, setKeyRange, setSize

7.34.3.15 void QCPColorMapData::setValueSize (int *valueSize*)

Resizes the data array to have *valueSize* cells in the value dimension.

The current data is discarded and the map cells are set to 0, unless the map had already the requested size.

Setting *valueSize* to zero frees the internal data array and isEmpty returns true.

See also

setValueRange, setSize, setKeySize

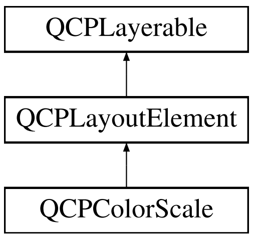
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.35 QCPColorScale Class Reference

A color scale for use with color coding data such as QCPColorMap.

Inheritance diagram for QCPColorScale:



Signals

- void dataRangeChanged (QCPRange newRange)
- void dataScaleTypeChanged (QCPAxis::ScaleType scaleType)
- void gradientChanged (QCPColorGradient newGradient)

Public Member Functions

- QCPColorScale (QCustomPlot ∗parentPlot)
- QCPAxis ∗ axis () const
- QCPAxis::AxisType **type** () const
- QCPRange **dataRange** () const
- QCPAxis::ScaleType **dataScaleType** () const
- QCPColorGradient **gradient** () const

- QString **label** () const
- int **barWidth** () const
- bool **rangeDrag** () const
- bool **rangeZoom** () const
- void setType (QCPAxis::AxisType type)
- Q_SLOT void setDataRange (const QCPRange &dataRange)
- Q_SLOT void setDataScaleType (QCPAxis::ScaleType scaleType)
- Q_SLOT void setGradient (const QCPColorGradient &gradient)
- void setLabel (const QString &str)
- void setBarWidth (int width)
- void setRangeDrag (bool enabled)
- void setRangeZoom (bool enabled)
- QList< QCPColorMap ∗ > colorMaps () const
- void rescaleDataRange (bool onlyVisibleMaps)
- virtual void update (UpdatePhase phase)

Protected Member Functions

- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void mousePressEvent (QMouseEvent ∗event)
- virtual void mouseMoveEvent (QMouseEvent ∗event)
- virtual void mouseReleaseEvent (QMouseEvent ∗event)
- virtual void wheelEvent (QWheelEvent ∗event)

Protected Attributes

- QCPAxis::AxisType **mType**
- QCPRange **mDataRange**
- QCPAxis::ScaleType **mDataScaleType**
- QCPColorGradient **mGradient**
- int **mBarWidth**
- QPointer < QCPColorScaleAxisRectPrivate > **mAxisRect**
- QPointer< QCPAxis > **mColorAxis**

Friends

- class **QCPColorScaleAxisRectPrivate**

7.35.1 Detailed Description

A color scale for use with color coding data such as QCPColorMap.

This layout element can be placed on the plot to correlate a color gradient with data values. It is usually used in combination with one or multiple QCPColorMaps.

The color scale can be either horizontal or vertical, as shown in the image above. The orientation and the side where the numbers appear is controlled with setType.

Use QCPColorMap::setColorScale to connect a color map with a color scale. Once they are connected, they share their gradient, data range and data scale type (setGradient, setDataRange, setDataScaleType). Multiple color maps may be associated with a single color scale, to make them all synchronize these properties.

To have finer control over the number display and axis behaviour, you can directly access the axis. See the documentation of QCPAxis for details about configuring axes. For example, if you want to change the number of automatically generated ticks, call

Placing a color scale next to the main axis rect works like with any other layout element:

In this case we have placed it to the right of the default axis rect, so it wasn't necessary to call setType, since QCPAxis::atRight is already the default. The text next to the color scale can be set with setLabel.

For optimum appearance (like in the image above), it may be desirable to line up the axis rect and the borders of the color scale. Use a QCPMarginGroup to achieve this:

Color scales are initialized with a non-zero minimum top and bottom margin (setMinimumMargins), because vertical color scales are most common and the minimum top/bottom margin makes sure it keeps some distance to the top/bottom widget border. So if you change to a horizontal color scale by setting setType to QCPAxis::atBottom or QCPAxis::atTop, you might want to also change the minimum margins accordingly, e.g. `setMinimumMargins(QMargins(6, 0, 6, 0))`.

7.35.2 Constructor & Destructor Documentation

7.35.2.1 QCPColorScale::QCPColorScale (QCustomPlot ∗ *parentPlot*) [explicit]

Constructs a new QCPColorScale.

7.35.3 Member Function Documentation

7.35.3.1 QCPAxis ∗ QCPColorScale::axis ( ) const [inline]

Returns the internal QCPAxis instance of this color scale. You can access it to alter the appearance and behaviour of the axis. QCPColorScale duplicates some properties in its interface for convenience. Those are setDataRange (QCPAxis::setRange), setDataScaleType (QCPAxis::setScaleType), and the method setLabel (QCPAxis::setLabel). - As they each are connected, it does not matter whether you use the method on the QCPColorScale or on its QCPAxis.

If the type of the color scale is changed with setType, the axis returned by this method will change, too, to either the left, right, bottom or top axis, depending on which type was set.

7.35.3.2 QList< QCPColorMap ∗ > QCPColorScale::colorMaps ( ) const

Returns a list of all the color maps associated with this color scale.

7.35.3.3 void QCPColorScale::dataRangeChanged (QCPRange *newRange*) [signal]

This signal is emitted when the data range changes.

See also

setDataRange

7.35.3.4 void QCPColorScale::dataScaleTypeChanged (QCPAxis::ScaleType *scaleType*) [signal]

This signal is emitted when the data scale type changes.

See also

setDataScaleType

7.35.3.5 void QCPColorScale::gradientChanged (QCPColorGradient *newGradient*) [signal]

This signal is emitted when the gradient changes.

See also

setGradient

7.35.3.6 void QCPColorScale::mouseMoveEvent (QMouseEvent ∗ *event*) `[protected, virtual]`

This event is called, if the mouse is moved inside the outer rect of this layout element.

Reimplemented from QCPLayoutElement.

7.35.3.7 void QCPColorScale::mousePressEvent (QMouseEvent ∗ *event*) `[protected, virtual]`

This event is called, if the mouse was pressed while being inside the outer rect of this layout element.

Reimplemented from QCPLayoutElement.

7.35.3.8 void QCPColorScale::mouseReleaseEvent (QMouseEvent ∗ *event*) `[protected, virtual]`

This event is called, if the mouse was previously pressed inside the outer rect of this layout element and is now released.

Reimplemented from QCPLayoutElement.

7.35.3.9 void QCPColorScale::rescaleDataRange (bool *onlyVisibleMaps*)

Changes the data range such that all color maps associated with this color scale are fully mapped to the gradient in the data dimension.

See also

setDataRange

7.35.3.10 void QCPColorScale::setBarWidth (int *width*)

Sets the width (or height, for horizontal color scales) the bar where the gradient is displayed will have.

7.35.3.11 void QCPColorScale::setDataRange (const QCPRange & *dataRange*)

Sets the range spanned by the color gradient and that is shown by the axis in the color scale.

It is equivalent to calling QCPColorMap::setDataRange on any of the connected color maps. It is also equivalent to directly accessing the axis and setting its range with QCPAxis::setRange.

See also

setDataScaleType, setGradient, rescaleDataRange

7.35.3.12 void QCPColorScale::setDataScaleType (QCPAxis::ScaleType *scaleType*)

Sets the scale type of the color scale, i.e. whether values are linearly associated with colors or logarithmically.

It is equivalent to calling QCPColorMap::setDataScaleType on any of the connected color maps. It is also equivalent to directly accessing the axis and setting its scale type with QCPAxis::setScaleType.

See also

setDataRange, setGradient

7.35.3.13 void QCPColorScale::setGradient (const QCPColorGradient & *gradient*)

Sets the color gradient that will be used to represent data values.

It is equivalent to calling QCPColorMap::setGradient on any of the connected color maps.

See also

setDataRange, setDataScaleType

7.35.3.14 void QCPColorScale::setLabel (const QString & *str*)

Sets the axis label of the color scale. This is equivalent to calling QCPAxis::setLabel on the internal axis.

7.35.3.15 void QCPColorScale::setRangeDrag (bool *enabled*)

Sets whether the user can drag the data range (setDataRange).

Note that QCP::iRangeDrag must be in the QCustomPlot's interactions (QCustomPlot::setInteractions) to allow range dragging.

7.35.3.16 void QCPColorScale::setRangeZoom (bool *enabled*)

Sets whether the user can zoom the data range (setDataRange) by scrolling the mouse wheel.

Note that QCP::iRangeZoom must be in the QCustomPlot's interactions (QCustomPlot::setInteractions) to allow range dragging.

7.35.3.17 void QCPColorScale::setType (QCPAxis::AxisType *type*)

Sets at which side of the color scale the axis is placed, and thus also its orientation.

Note that after setting *type* to a different value, the axis returned by axis() will be a different one. The new axis will adopt the following properties from the previous axis: The range, scale type, log base and label.

7.35.3.18 void QCPColorScale::update (UpdatePhase *phase*) [virtual]

Updates the layout element and sub-elements. This function is automatically called before every replot by the parent layout element. It is called multiple times, once for every UpdatePhase. The phases are run through in the order of the enum values. For details about what happens at the different phases, see the documentation of Update-Phase.

Layout elements that have child elements should call the update method of their child elements, and pass the current *phase* unchanged.

The default implementation executes the automatic margin mechanism in the upMargins phase. Subclasses should make sure to call the base class implementation.

Reimplemented from QCPLayoutElement.

7.35.3.19 void QCPColorScale::wheelEvent (QWheelEvent * *event*) [protected, virtual]

This event is called, if the mouse wheel is scrolled while the cursor is inside the rect of this layout element.

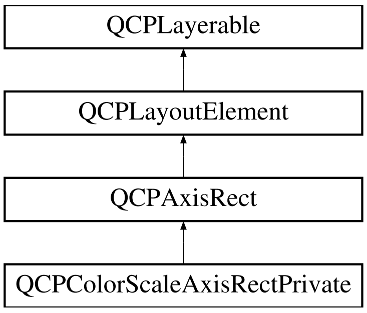
Reimplemented from QCPLayoutElement.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.36 QCPColorScaleAxisRectPrivate Class Reference

Inheritance diagram for QCPColorScaleAxisRectPrivate:



Public Member Functions

- QCPColorScaleAxisRectPrivate (QCPColorScale ∗parentColorScale)

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- void **updateGradientImage** ()
- Q_SLOT void **axisSelectionChanged** (QCPAxis::SelectableParts selectedParts)
- Q_SLOT void **axisSelectableChanged** (QCPAxis::SelectableParts selectableParts)

Protected Attributes

- QCPColorScale ∗ **mParentColorScale**
- QImage **mGradientImage**
- bool **mGradientImageInvalidated**

Friends

- class **QCPColorScale**

7.36.1 Constructor & Destructor Documentation

7.36.1.1 QCPColorScaleAxisRectPrivate::QCPColorScaleAxisRectPrivate (QCPColorScale ∗ *parentColorScale*) [explicit]

Creates a new instance, as a child of *parentColorScale*.

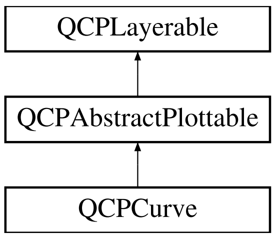
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.37 QCPCurve Class Reference

A plottable representing a parametric curve in a plot.

Inheritance diagram for QCPCurve:



Public Types

- enum LineStyle { IsNone, IsLine }

Public Member Functions

- QCPCurve (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- QCPCurveDataMap ∗ **data** () const
- QCPScatterStyle **scatterStyle** () const
- LineStyle **lineStyle** () const
- void setData (QCPCurveDataMap ∗data, bool copy=false)
- void setData (const QVector< double > &t, const QVector< double > &key, const QVector< double > &value)
- void setData (const QVector< double > &key, const QVector< double > &value)
- void setScatterStyle (const QCPScatterStyle &style)
- void setLineStyle (LineStyle style)
- void addData (const QCPCurveDataMap &dataMap)
- void addData (const QCPCurveData &data)
- void addData (double t, double key, double value)
- void addData (double key, double value)
- void addData (const QVector< double > &ts, const QVector< double > &keys, const QVector< double > &values)
- void removeDataBefore (double t)
- void removeDataAfter (double t)
- void removeData (double fromt, double tot)
- void removeData (double t)
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual void **drawScatterPlot** (QCPPainter ∗painter, const QVector< QPointF > ∗pointData) const
- void **getCurveData** (QVector< QPointF > ∗lineData) const
- int **getRegion** (double x, double y, double rectLeft, double rectTop, double rectRight, double rectBottom) const
- QPointF **getOptimizedPoint** (int prevRegion, double prevKey, double prevValue, double key, double value, double rectLeft, double rectTop, double rectRight, double rectBottom) const
- QVector< QPointF > **getOptimizedCornerPoints** (int prevRegion, int currentRegion, double prevKey, double prevValue, double key, double value, double rectLeft, double rectTop, double rectRight, double rectBottom) const
- bool **mayTraverse** (int prevRegion, int currentRegion) const
- bool **getTraverse** (double prevKey, double prevValue, double key, double value, double rectLeft, double rectTop, double rectRight, double rectBottom, QPointF &crossA, QPointF &crossB) const
- void **getTraverseCornerPoints** (int prevRegion, int currentRegion, double rectLeft, double rectTop, double rectRight, double rectBottom, QVector< QPointF > &beforeTraverse, QVector< QPointF > &afterTraverse) const
- double **pointDistance** (const QPointF &pixelPoint) const

Protected Attributes

- QCPCurveDataMap ∗ **mData**
- QCPScatterStyle **mScatterStyle**
- LineStyle **mLineStyle**

Friends

- class **QCustomPlot**
- class **QCPLegend**

7.37.1 Detailed Description

A plottable representing a parametric curve in a plot.

Unlike QCPGraph, plottables of this type may have multiple points with the same key coordinate, so their visual representation can have *loops*. This is realized by introducing a third coordinate *t*, which defines the order of the points described by the other two coordinates *x* and *y*.

To plot data, assign it with the setData or addData functions.

Gaps in the curve can be created by adding data points with NaN as key and value (`qQNaN()` or `std::numeric_limits<double>::quiet_NaN()`) in between the two data points that shall be separated.

7.37.2 Changing the appearance

The appearance of the curve is determined by the pen and the brush (setPen, setBrush).

7.37.3 Usage

Like all data representing objects in QCustomPlot, the QCPCurve is a plottable (QCPAbstractPlottable). So the plottable-interface of QCustomPlot applies (QCustomPlot::plottable, QCustomPlot::addPlottable, QCustomPlot::removePlottable, etc.)

Usually, you first create an instance and add it to the m_customPlot:

and then modify the properties of the newly created plottable, e.g.:

7.37.4 Member Enumeration Documentation

7.37.4.1 enum QCPCurve::LineStyle

Defines how the curve's line is represented visually in the plot. The line is drawn with the current pen of the curve (setPen).

See also

setLineStyle

Enumerator:

lsNone No line is drawn between data points (e.g. only scatters)

lsLine Data points are connected with a straight line.

7.37.5 Constructor & Destructor Documentation

7.37.5.1 QCPCurve::QCPCurve (QCPAxis ∗ *keyAxis*, QCPAxis ∗ *valueAxis*) `[explicit]`

Constructs a curve which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and not have the same orientation. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

The constructed QCPCurve can be added to the plot with QCustomPlot::addPlottable, QCustomPlot then takes ownership of the graph.

7.37.6 Member Function Documentation

7.37.6.1 void **QCPCurve::addData** (const **QCPCurveDataMap** & *dataMap*)

Adds the provided data points in *dataMap* to the current data.

See also

removeData

7.37.6.2 void **QCPCurve::addData** (const **QCPCurveData** & *data*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point in *data* to the current data.

See also

removeData

7.37.6.3 void **QCPCurve::addData** (double *t*, double *key*, double *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point as *t*, *key* and *value* tuple to the current data

See also

removeData

7.37.6.4 void **QCPCurve::addData** (double *key*, double *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point as *key* and *value* pair to the current data The t parameter of the data point is set to the t of the last data point plus 1. If there is no last data point, t will be set to 0.

See also

removeData

7.37.6.5 void **QCPCurve::addData** (const QVector< double > & *ts,* const QVector< double > & *keys,* const QVector< double > & *values*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided data points as *t*, *key* and *value* tuples to the current data.

See also

removeData

7.37.6.6 void **QCPCurve::clearData** ( ) `[virtual]`

Removes all data points.

See also

removeData, removeDataAfter, removeDataBefore

Implements QCPAbstractPlottable.

7.37.6.7 void **QCPCurve::removeData** (double *fromt,* double *tot*)

Removes all data points with curve parameter t between *fromt* and *tot*. if *fromt* is greater or equal to *tot*, the function does nothing. To remove a single data point with known t, use removeData(double t).

See also

addData, clearData

7.37.6.8 void **QCPCurve::removeData** (double *t*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes a single data point at curve parameter *t*. If the position is not known with absolute precision, consider using removeData(double fromt, double tot) with a small fuzziness interval around the suspected position, depeding on the precision with which the curve parameter is known.

See also

addData, clearData

7.37.6.9 void QCPCurve::removeDataAfter (double *t*)

Removes all data points with curve parameter t greater than *t*.

See also

addData, clearData

7.37.6.10 void QCPCurve::removeDataBefore (double *t*)

Removes all data points with curve parameter t smaller than *t*.

See also

addData, clearData

7.37.6.11 double QCPCurve::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.37.6.12 void QCPCurve::setData (QCPCurveDataMap $*$ *data,* bool *copy* = `false`)

Replaces the current data with the provided *data*.

If *copy* is set to true, data points in *data* will only be copied. if false, the plottable takes ownership of the passed data and replaces the internal data pointer with it. This is significantly faster than copying for large datasets.

7.37.6.13 void QCPCurve::setData (const QVector$<$ double $>$ & *t,* const QVector$<$ double $>$ & *key,* const QVector$<$ double $>$ & *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *t*, *key* and *value* tuples. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.37.6.14 void QCPCurve::setData (const QVector$<$ double $>$ & *key,* const QVector$<$ double $>$ & *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided *key* and *value* pairs. The t parameter of each data point will be set to the integer index of the respective key/value pair.

7.37.6.15 void QCPCurve::setLineStyle (QCPCurve::LineStyle *style*)

Sets how the single data points are connected in the plot or how they are represented visually apart from the scatter symbol. For scatter-only plots, set *style* to lsNone and setScatterStyle to the desired scatter style.

See also

setScatterStyle

7.37.6.16 void QCPCurve::setScatterStyle (const QCPScatterStyle & *style*)

Sets the visual appearance of single data points in the plot. If set to QCPScatterStyle::ssNone, no scatter points are drawn (e.g. for line-only plots with appropriate line style).

See also

QCPScatterStyle, setLineStyle

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.38 QCPCurveData Class Reference

Holds the data of one single data point for QCPCurve.

Public Member Functions

- QCPCurveData ()
- QCPCurveData (double t, double key, double value)

Public Attributes

- double **t**
- double **key**
- double **value**

7.38.1 Detailed Description

Holds the data of one single data point for QCPCurve.

The container for storing multiple data points is QCPCurveDataMap.

The stored data is:

- *t:* the free parameter of the curve at this curve point (cp. the mathematical vector *(x(t), y(t))*)
- *key:* coordinate on the key axis of this curve point
- *value:* coordinate on the value axis of this curve point

See also

QCPCurveDataMap

7.38.2 Constructor & Destructor Documentation

7.38.2.1 QCPCurveData::QCPCurveData ( )

Constructs a curve data point with t, key and value set to zero.

7.38.2.2 QCPCurveData::QCPCurveData (double *t*, double *key*, double *value*)

Constructs a curve data point with the specified *t*, *key* and *value*.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.39 QCPData Class Reference

Holds the data of one single data point for QCPGraph.

Public Member Functions

- QCPData ()
- QCPData (double key, double value)

Public Attributes

- double **key**
- double **value**
- double **keyErrorPlus**
- double **keyErrorMinus**
- double **valueErrorPlus**
- double **valueErrorMinus**

7.39.1 Detailed Description

Holds the data of one single data point for QCPGraph.

The container for storing multiple data points is QCPDataMap.

The stored data is:

- *key:* coordinate on the key axis of this data point
- *value:* coordinate on the value axis of this data point
- *keyErrorMinus:* negative error in the key dimension (for error bars)
- *keyErrorPlus:* positive error in the key dimension (for error bars)
- *valueErrorMinus:* negative error in the value dimension (for error bars)
- *valueErrorPlus:* positive error in the value dimension (for error bars)

See also

QCPDataMap

7.39.2 Constructor & Destructor Documentation

7.39.2.1 QCPData::QCPData ( )

Constructs a data point with key, value and all errors set to zero.

7.39.2.2 QCPData::QCPData (double *key*, double *value*)

Constructs a data point with the specified *key* and *value*. All errors are set to zero.

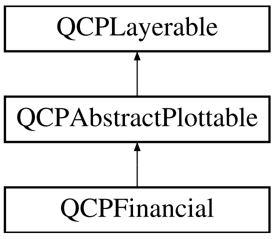
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.40 QCPFinancial Class Reference

A plottable representing a financial stock chart.

Inheritance diagram for QCPFinancial:



Public Types

- enum ChartStyle { csOhlc, csCandlestick }

Public Member Functions

- QCPFinancial (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- QCPFinancialDataMap ∗ data () const
- ChartStyle **chartStyle** () const
- double **width** () const
- bool **twoColored** () const
- QBrush **brushPositive** () const
- QBrush **brushNegative** () const
- QPen **penPositive** () const
- QPen **penNegative** () const

- void setData (QCPFinancialDataMap ∗data, bool copy=false)
- void setData (const QVector< double > &key, const QVector< double > &open, const QVector< double > &high, const QVector< double > &low, const Q-Vector< double > &close)
- void setChartStyle (ChartStyle style)
- void setWidth (double width)
- void setTwoColored (bool twoColored)
- void setBrushPositive (const QBrush &brush)
- void setBrushNegative (const QBrush &brush)
- void setPenPositive (const QPen &pen)
- void setPenNegative (const QPen &pen)
- void addData (const QCPFinancialDataMap &dataMap)
- void addData (const QCPFinancialData &data)
- void addData (double key, double open, double high, double low, double close)
- void addData (const QVector< double > &key, const QVector< double > &open, const QVector< double > &high, const QVector< double > &low, const Q-Vector< double > &close)
- void removeDataBefore (double key)
- void removeDataAfter (double key)
- void removeData (double fromKey, double toKey)
- void removeData (double key)
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Static Public Member Functions

- static QCPFinancialDataMap timeSeriesToOhlc (const QVector< double > &time, const QVector< double > &value, double timeBinSize, double timeBin-Offset=0)

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSign-Domain=sdBoth) const
- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSign-Domain=sdBoth) const
- void **drawOhlcPlot** (QCPPainter ∗painter, const QCPFinancialDataMap::const_-iterator &begin, const QCPFinancialDataMap::const_iterator &end)
- void **drawCandlestickPlot** (QCPPainter ∗painter, const QCPFinancialDataMap-::const_iterator &begin, const QCPFinancialDataMap::const_iterator &end)
- double **ohlcSelectTest** (const QPointF &pos, const QCPFinancialDataMap-::const_iterator &begin, const QCPFinancialDataMap::const_iterator &end) const

- double **candlestickSelectTest** (const QPointF &pos, const QCPFinancialDataMap::const_iterator &begin, const QCPFinancialDataMap::const_iterator &end) const
- void **getVisibleDataBounds** (QCPFinancialDataMap::const_iterator &lower, QCPFinancialDataMap::const_iterator &upper) const

Protected Attributes

- QCPFinancialDataMap $*$ **mData**
- ChartStyle **mChartStyle**
- double **mWidth**
- bool **mTwoColored**
- QBrush **mBrushPositive**
- QBrush **mBrushNegative**
- QPen **mPenPositive**
- QPen **mPenNegative**

Friends

- class **QCustomPlot**
- class **QCPLegend**

7.40.1 Detailed Description

A plottable representing a financial stock chart.

This plottable represents time series data binned to certain intervals, mainly used for stock charts. The two common representations OHLC (Open-High-Low-Close) bars and Candlesticks can be set via setChartStyle.

The data is passed via setData as a set of open/high/low/close values at certain keys (typically times). This means the data must be already binned appropriately. If data is only available as a series of values (e.g. *price* against *time*), you can use the static convenience function timeSeriesToOhlc to generate binned OHLC-data which can then be passed to setData.

The width of the OHLC bars/candlesticks can be controlled with setWidth and is given in plot key coordinates. A typical choice is to set it to (or slightly less than) one bin interval width.

7.40.2 Changing the appearance

Charts can be either single- or two-colored (setTwoColored). If set to be single-colored, lines are drawn with the plottable's pen (setPen) and fills with the brush (setBrush).

If set to two-colored, positive changes of the value during an interval (*close* $>=$ *open*) are represented with a different pen and brush than negative changes (*close* $<$ *open*). These can be configured with setPenPositive, setPenNegative, setBrushPositive, and setBrushNegative. In two-colored mode, the normal plottable pen/brush is ignored. Upon selection however, the normal selected pen/brush (setSelectedPen, setSelected-Brush) is used, irrespective of whether the chart is single- or two-colored.

7.40.3 Member Enumeration Documentation

7.40.3.1 enum **QCPFinancial::ChartStyle**

Defines the possible representations of OHLC data in the plot.

See also

setChartStyle

Enumerator:

csOhlc Open-High-Low-Close bar representation.

csCandlestick Candlestick representation.

7.40.4 Constructor & Destructor Documentation

7.40.4.1 QCPFinancial::QCPFinancial (QCPAxis ∗ *keyAxis*, QCPAxis ∗ *valueAxis*) `[explicit]`

Constructs a financial chart which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and not have the same orientation. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

The constructed QCPFinancial can be added to the plot with QCustomPlot::add-Plottable, QCustomPlot then takes ownership of the financial chart.

7.40.5 Member Function Documentation

7.40.5.1 void **QCPFinancial::addData** (const **QCPFinancialDataMap** & *dataMap*)

Adds the provided data points in *dataMap* to the current data.

Alternatively, you can also access and modify the data via the data method, which returns a pointer to the internal QCPFinancialDataMap.

See also

removeData

7.40.5.2 void QCPFinancial::addData (const QCPFinancialData & *data*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point in *data* to the current data.

Alternatively, you can also access and modify the data via the data method, which returns a pointer to the internal QCPFinancialData.

See also

removeData

7.40.5.3 void QCPFinancial::addData (double *key,* double *open,* double *high,* double *low,* double *close*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point given by *key*, *open*, *high*, *low*, and *close* to the current data.

Alternatively, you can also access and modify the data via the data method, which returns a pointer to the internal QCPFinancialData.

See also

removeData

7.40.5.4 void QCPFinancial::addData (const QVector< double > & *key,* const QVector< double > & *open,* const QVector< double > & *high,* const QVector< double > & *low,* const QVector< double > & *close*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided open/high/low/-close data to the current data.

Alternatively, you can also access and modify the data via the data method, which returns a pointer to the internal QCPFinancialData.

See also

removeData

7.40.5.5 void QCPFinancial::clearData ( ) `[virtual]`

Removes all data points.

See also

removeData, removeDataAfter, removeDataBefore

Implements QCPAbstractPlottable.

7.40.5.6 QCPFinancialDataMap ∗ QCPFinancial::data ( ) const [inline]

Returns a pointer to the internal data storage of type QCPFinancialDataMap. You may use it to directly manipulate the data, which may be more convenient and faster than using the regular setData or addData methods, in certain situations.

7.40.5.7 void QCPFinancial::removeData (double *fromKey*, double *toKey*)

Removes all data points with keys between *fromKey* and *toKey*. if *fromKey* is greater or equal to *toKey*, the function does nothing. To remove a single data point with known key, use removeData(double key).

See also

addData, clearData

7.40.5.8 void QCPFinancial::removeData (double *key*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes a single data point at *key*. If the position is not known with absolute precision, consider using removeData(double fromKey, double toKey) with a small fuzziness interval around the suspected position, depeding on the precision with which the key is known.

See also

addData, clearData

7.40.5.9 void QCPFinancial::removeDataAfter (double *key*)

Removes all data points with keys greater than *key*.

See also

addData, clearData

7.40.5.10 void QCPFinancial::removeDataBefore (double *key*)

Removes all data points with keys smaller than *key*.

See also

addData, clearData

7.40.5.11 double QCPFinancial::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = `0`) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.40.5.12 void QCPFinancial::setBrushNegative (const QBrush & *brush*)

If setTwoColored is set to true, this function controls the brush that is used to draw fills of data points with a negative trend (i.e. bars/candlesticks with close $<$ open).

If *twoColored* is false, the normal plottable's pen and brush are used (setPen, setBrush).

See also

setBrushPositive, setPenNegative, setPenPositive

7.40.5.13 void QCPFinancial::setBrushPositive (const QBrush & *brush*)

If setTwoColored is set to true, this function controls the brush that is used to draw fills of data points with a positive trend (i.e. bars/candlesticks with close $>=$ open).

If *twoColored* is false, the normal plottable's pen and brush are used (setPen, setBrush).

See also

setBrushNegative, setPenPositive, setPenNegative

7.40.5.14 void QCPFinancial::setChartStyle (QCPFinancial::ChartStyle *style*)

Sets which representation style shall be used to display the OHLC data.

7.40.5.15 void QCPFinancial::setData (QCPFinancialDataMap $*$ *data*, bool *copy* = `false`)

Replaces the current data with the provided *data*.

If *copy* is set to true, data points in *data* will only be copied. if false, the plottable takes ownership of the passed data and replaces the internal data pointer with it. This is significantly faster than copying for large datasets.

Alternatively, you can also access and modify the plottable's data via the data method, which returns a pointer to the internal QCPFinancialDataMap.

See also

timeSeriesToOhlc

7.40.5.16 void QCPFinancial::setData (const QVector$<$ double $>$ & *key*, const QVector$<$ double $>$ & *open*, const QVector$<$ double $>$ & *high*, const QVector$<$ double $>$ & *low*, const QVector$<$ double $>$ & *close*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided open/high/low/close data. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

See also

timeSeriesToOhlc

7.40.5.17 void QCPFinancial::setPenNegative (const QPen & *pen*)

If setTwoColored is set to true, this function controls the pen that is used to draw outlines of data points with a negative trend (i.e. bars/candlesticks with close $<$ open).

If *twoColored* is false, the normal plottable's pen and brush are used (setPen, setBrush).

See also

setPenPositive, setBrushNegative, setBrushPositive

7.40.5.18 void QCPFinancial::setPenPositive (const QPen & *pen*)

If setTwoColored is set to true, this function controls the pen that is used to draw outlines of data points with a positive trend (i.e. bars/candlesticks with close >= open).

If *twoColored* is false, the normal plottable's pen and brush are used (setPen, setBrush).

See also

setPenNegative, setBrushPositive, setBrushNegative

7.40.5.19 void QCPFinancial::setTwoColored (bool *twoColored*)

Sets whether this chart shall contrast positive from negative trends per data point by using two separate colors to draw the respective bars/candlesticks.

If *twoColored* is false, the normal plottable's pen and brush are used (setPen, setBrush).

See also

setPenPositive, setPenNegative, setBrushPositive, setBrushNegative

7.40.5.20 void QCPFinancial::setWidth (double *width*)

Sets the width of the individual bars/candlesticks to *width* in plot key coordinates.

A typical choice is to set it to (or slightly less than) one bin interval width.

7.40.5.21 QCPFinancialDataMap QCPFinancial::timeSeriesToOhlc (const QVector< double > & *time*, const QVector< double > & *value*, double *timeBinSize*, double *timeBinOffset* = 0) [static]

A convenience function that converts time series data (*value* against *time*) to OHLC binned data points. The return value can then be passed on to setData.

The size of the bins can be controlled with *timeBinSize* in the same units as *time* is given. For example, if the unit of *time* is seconds and single OHLC/Candlesticks should span an hour each, set *timeBinSize* to 3600.

timeBinOffset allows to control precisely at what *time* coordinate a bin should start. The value passed as *timeBinOffset* doesn't need to be in the range encompassed by the *time* keys. It merely defines the mathematical offset/phase of the bins that will be used to process the data.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.41 QCPFinancialData Class Reference

Holds the data of one single data point for QCPFinancial.

Public Member Functions

- QCPFinancialData ()
- QCPFinancialData (double key, double open, double high, double low, double close)

Public Attributes

- double **key**
- double **open**
- double **high**
- double **low**
- double **close**

7.41.1 Detailed Description

Holds the data of one single data point for QCPFinancial.

The container for storing multiple data points is QCPFinancialDataMap.

The stored data is:

- *key:* coordinate on the key axis of this data point
- *open:* The opening value at the data point
- *high:* The high/maximum value at the data point
- *low:* The low/minimum value at the data point
- *close:* The closing value at the data point

See also

QCPFinancialDataMap

7.41.2 Constructor & Destructor Documentation

7.41.2.1 QCPFinancialData::QCPFinancialData ( )

Constructs a data point with key and all values set to zero.

7.41.2.2 QCPFinancialData::QCPFinancialData (double *key,* double *open,* double *high,* double *low,* double *close*)

Constructs a data point with the specified *key* and OHLC values.

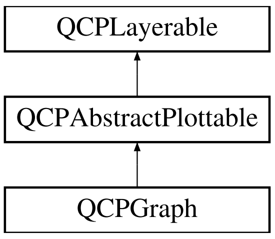
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.42 QCPGraph Class Reference

A plottable representing a graph in a plot.

Inheritance diagram for QCPGraph:



Public Types

- enum LineStyle { lsNone, lsLine, lsStepLeft, lsStepRight, lsStepCenter, ls-Impulse }
- enum ErrorType { etNone, etKey, etValue, etBoth }

Public Member Functions

- QCPGraph (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- QCPDataMap ∗ data () const
- LineStyle **lineStyle** () const
- QCPScatterStyle **scatterStyle** () const
- ErrorType **errorType** () const
- QPen **errorPen** () const
- double **errorBarSize** () const
- bool **errorBarSkipSymbol** () const
- QCPGraph ∗ **channelFillGraph** () const
- bool **adaptiveSampling** () const
- void setData (QCPDataMap ∗data, bool copy=false)
- void setData (const QVector< double > &key, const QVector< double > &value)

- void setDataKeyError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &keyError)
- void setDataKeyError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &keyErrorMinus, const QVector< double > &keyErrorPlus)
- void setDataValueError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &valueError)
- void setDataValueError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &valueErrorMinus, const QVector< double > &valueErrorPlus)
- void setDataBothError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &keyError, const QVector< double > &valueError)
- void setDataBothError (const QVector< double > &key, const QVector< double > &value, const QVector< double > &keyErrorMinus, const QVector< double > &keyErrorPlus, const QVector< double > &valueErrorMinus, const QVector< double > &valueErrorPlus)
- void setLineStyle (LineStyle ls)
- void setScatterStyle (const QCPScatterStyle &style)
- void setErrorType (ErrorType errorType)
- void setErrorPen (const QPen &pen)
- void setErrorBarSize (double size)
- void setErrorBarSkipSymbol (bool enabled)
- void setChannelFillGraph (QCPGraph ∗targetGraph)
- void setAdaptiveSampling (bool enabled)
- void addData (const QCPDataMap &dataMap)
- void addData (const QCPData &data)
- void addData (double key, double value)
- void addData (const QVector< double > &keys, const QVector< double > &values)
- void removeDataBefore (double key)
- void removeDataAfter (double key)
- void removeData (double fromKey, double toKey)
- void removeData (double key)
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- void rescaleAxes (bool onlyEnlarge, bool includeErrorBars) const
- void rescaleKeyAxis (bool onlyEnlarge, bool includeErrorBars) const
- void rescaleValueAxis (bool onlyEnlarge, bool includeErrorBars) const

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const

- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual QCPRange getKeyRange (bool &foundRange, SignDomain inSignDomain, bool includeErrors) const
- virtual QCPRange getValueRange (bool &foundRange, SignDomain inSignDomain, bool includeErrors) const
- virtual void **drawFill** (QCPPainter ∗painter, QVector< QPointF > ∗lineData) const
- virtual void **drawScatterPlot** (QCPPainter ∗painter, QVector< QCPData > ∗scatterData) const
- virtual void **drawLinePlot** (QCPPainter ∗painter, QVector< QPointF > ∗lineData) const
- virtual void **drawImpulsePlot** (QCPPainter ∗painter, QVector< QPointF > ∗lineData) const
- void **getPreparedData** (QVector< QCPData > ∗lineData, QVector< QCPData > ∗scatterData) const
- void **getPlotData** (QVector< QPointF > ∗lineData, QVector< QCPData > ∗scatterData) const
- void **getScatterPlotData** (QVector< QCPData > ∗scatterData) const
- void **getLinePlotData** (QVector< QPointF > ∗linePixelData, QVector< QCPData > ∗scatterData) const
- void **getStepLeftPlotData** (QVector< QPointF > ∗linePixelData, QVector< QCPData > ∗scatterData) const
- void **getStepRightPlotData** (QVector< QPointF > ∗linePixelData, QVector< QCPData > ∗scatterData) const
- void **getStepCenterPlotData** (QVector< QPointF > ∗linePixelData, QVector< QCPData > ∗scatterData) const
- void **getImpulsePlotData** (QVector< QPointF > ∗linePixelData, QVector< QCPData > ∗scatterData) const
- void **drawError** (QCPPainter ∗painter, double x, double y, const QCPData &data) const
- void **getVisibleDataBounds** (QCPDataMap::const_iterator &lower, QCPDataMap::const_iterator &upper) const
- int **countDataInBounds** (const QCPDataMap::const_iterator &lower, const QCPDataMap::const_iterator &upper, int maxCount) const
- void **addFillBasePoints** (QVector< QPointF > ∗lineData) const
- void **removeFillBasePoints** (QVector< QPointF > ∗lineData) const
- QPointF **lowerFillBasePoint** (double lowerKey) const
- QPointF **upperFillBasePoint** (double upperKey) const
- const QPolygonF **getChannelFillPolygon** (const QVector< QPointF > ∗lineData) const
- int **findIndexBelowX** (const QVector< QPointF > ∗data, double x) const
- int **findIndexAboveX** (const QVector< QPointF > ∗data, double x) const
- int **findIndexBelowY** (const QVector< QPointF > ∗data, double y) const
- int **findIndexAboveY** (const QVector< QPointF > ∗data, double y) const
- double **pointDistance** (const QPointF &pixelPoint) const

Protected Attributes

- QCPDataMap ∗ **mData**
- QPen **mErrorPen**
- LineStyle **mLineStyle**
- QCPScatterStyle **mScatterStyle**
- ErrorType **mErrorType**
- double **mErrorBarSize**
- bool **mErrorBarSkipSymbol**
- QPointer< QCPGraph > **mChannelFillGraph**
- bool **mAdaptiveSampling**

Friends

- class **QCustomPlot**
- class **QCPLegend**

7.42.1 Detailed Description

A plottable representing a graph in a plot.

Usually you create new graphs by calling QCustomPlot::addGraph. The resulting instance can be accessed via QCustomPlot::graph.

To plot data, assign it with the setData or addData functions. Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

Graphs are used to display single-valued data. Single-valued means that there should only be one data point per unique key coordinate. In other words, the graph can't have *loops*. If you do want to plot non-single-valued curves, rather use the QCPCurve plottable.

Gaps in the graph line can be created by adding data points with NaN as value (`qQNaN()` or `std::numeric_limits<double>::quiet_NaN()`) in between the two data points that shall be separated.

7.42.2 Changing the appearance

The appearance of the graph is mainly determined by the line style, scatter style, brush and pen of the graph (setLineStyle, setScatterStyle, setBrush, setPen).

7.42.2.1 Filling under or between graphs

QCPGraph knows two types of fills: Normal graph fills towards the zero-value-line parallel to the key axis of the graph, and fills between two graphs, called channel fills. To enable a fill, just set a brush with setBrush which is neither Qt::NoBrush nor fully transparent.

By default, a normal fill towards the zero-value-line will be drawn. To set up a channel fill between this graph and another one, call setChannelFillGraph with the other graph as parameter.

See also

QCustomPlot::addGraph, QCustomPlot::graph

7.42.3 Member Enumeration Documentation

7.42.3.1 enum QCPGraph::ErrorType

Defines what kind of error bars are drawn for each data point

Enumerator:

etNone No error bars are shown.

etKey Error bars for the key dimension of the data point are shown.

etValue Error bars for the value dimension of the data point are shown.

etBoth Error bars for both key and value dimensions of the data point are shown.

7.42.3.2 enum QCPGraph::LineStyle

Defines how the graph's line is represented visually in the plot. The line is drawn with the current pen of the graph (setPen).

See also

setLineStyle

Enumerator:

lsNone data points are not connected with any lines (e.g. data only represented with symbols according to the scatter style, see setScatterStyle)

lsLine data points are connected by a straight line

lsStepLeft line is drawn as steps where the step height is the value of the left data point

lsStepRight line is drawn as steps where the step height is the value of the right data point

lsStepCenter line is drawn as steps where the step is in between two data points

lsImpulse each data point is represented by a line parallel to the value axis, which reaches from the data point to the zero-value-line

7.42.4 Constructor & Destructor Documentation

7.42.4.1 QCPGraph::QCPGraph (QCPAxis $*$ *keyAxis*, QCPAxis $*$ *valueAxis*) [explicit]

Constructs a graph which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and not have the same orientation. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

The constructed QCPGraph can be added to the plot with QCustomPlot::addPlottable, QCustomPlot then takes ownership of the graph.

To directly create a graph inside a plot, you can also use the simpler QCustomPlot::addGraph function.

7.42.5 Member Function Documentation

7.42.5.1 void QCPGraph::addData (const QCPDataMap & *dataMap*)

Adds the provided data points in *dataMap* to the current data.

Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

See also

removeData

7.42.5.2 void QCPGraph::addData (const QCPData & *data*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point in *data* to the current data.

Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

See also

removeData

7.42.5.3 void QCPGraph::addData (double *key*, double *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided single data point as *key* and *value* pair to the current data.

Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

See also

removeData

7.42.5.4 void QCPGraph::addData (const QVector< double > & *keys,* const QVector< double > & *values*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Adds the provided data points as *key* and *value* pairs to the current data.

Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

See also

removeData

7.42.5.5 void QCPGraph::clearData ( ) `[virtual]`

Removes all data points.

See also

removeData, removeDataAfter, removeDataBefore

Implements QCPAbstractPlottable.

7.42.5.6 QCPDataMap ∗ QCPGraph::data ( ) const `[inline]`

Returns a pointer to the internal data storage of type QCPDataMap. You may use it to directly manipulate the data, which may be more convenient and faster than using the regular setData or addData methods, in certain situations.

7.42.5.7 QCPRange QCPGraph::getKeyRange (bool & *foundRange,* SignDomain *inSignDomain,* bool *includeErrors*) const `[protected, virtual]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows to specify whether the error bars should be included in the range calculation.

See also

getKeyRange(bool &foundRange, SignDomain inSignDomain)

7.42.5.8 QCPRange QCPGraph::getValueRange (bool & *foundRange,* SignDomain *inSignDomain,* bool *includeErrors*) const [protected, virtual]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows to specify whether the error bars should be included in the range calculation.

See also

getValueRange(bool &foundRange, SignDomain inSignDomain)

7.42.5.9 void QCPGraph::removeData (double *fromKey,* double *toKey*)

Removes all data points with keys between *fromKey* and *toKey*. if *fromKey* is greater or equal to *toKey*, the function does nothing. To remove a single data point with known key, use removeData(double key).

See also

addData, clearData

7.42.5.10 void QCPGraph::removeData (double *key*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes a single data point at *key*. If the position is not known with absolute precision, consider using removeData(double fromKey, double toKey) with a small fuzziness interval around the suspected position, depeding on the precision with which the key is known.

See also

addData, clearData

7.42.5.11 void QCPGraph::removeDataAfter (double *key*)

Removes all data points with keys greater than *key*.

See also

addData, clearData

7.42.5.12 void QCPGraph::removeDataBefore (double *key*)

Removes all data points with keys smaller than *key*.

See also

addData, clearData

7.42.5.13 void **QCPGraph::rescaleAxes** (bool *onlyEnlarge,* bool *includeErrorBars*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows to define whether error bars are taken into consideration when determining the new axis range.

See also

rescaleKeyAxis, rescaleValueAxis, QCPAbstractPlottable::rescaleAxes, QCustom-Plot::rescaleAxes

7.42.5.14 void **QCPGraph::rescaleKeyAxis** (bool *onlyEnlarge,* bool *includeErrorBars*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows to define whether error bars (of kind QCPGraph::etKey) are taken into consideration when determining the new axis range.

See also

rescaleAxes, QCPAbstractPlottable::rescaleKeyAxis

7.42.5.15 void **QCPGraph::rescaleValueAxis** (bool *onlyEnlarge,* bool *includeErrorBars*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows to define whether error bars (of kind QCPGraph::etValue) are taken into consideration when determining the new axis range.

See also

rescaleAxes, QCPAbstractPlottable::rescaleValueAxis

7.42.5.16 double **QCPGraph::selectTest** (const QPointF & *pos,* bool *onlySelectable,* QVariant * *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.42.5.17 void **QCPGraph::setAdaptiveSampling** (bool *enabled*)

Sets whether adaptive sampling shall be used when plotting this graph. QCustomPlot's adaptive sampling technique can drastically improve the replot performance for graphs with a larger number of points (e.g. above 10,000), without notably changing the appearance of the graph.

By default, adaptive sampling is enabled. Even if enabled, QCustomPlot decides whether adaptive sampling shall actually be used on a per-graph basis. So leaving adaptive sampling enabled has no disadvantage in almost all cases.

As can be seen, line plots experience no visual degradation from adaptive sampling. Outliers are reproduced reliably, as well as the overall shape of the data set. The replot time reduces dramatically though. This allows QCustomPlot to display large amounts of data in realtime.

Care must be taken when using high-density scatter plots in combination with adaptive sampling. The adaptive sampling algorithm treats scatter plots more carefully than line plots which still gives a significant reduction of replot times, but not quite as much as for line plots. This is because scatter plots inherently need more data points to be preserved in order to still resemble the original, non-adaptive-sampling plot. As shown above, the results still aren't quite identical, as banding occurs for the outer data points. This is in fact intentional, such that the boundaries of the data cloud stay visible to the viewer. How strong the banding appears, depends on the point density, i.e. the number of points in the plot.

For some situations with scatter plots it might thus be desirable to manually turn adaptive sampling off. For example, when saving the plot to disk. This can be achieved by setting *enabled* to false before issuing a command like QCustomPlot::savePng, and setting *enabled* back to true afterwards.

7.42.5.18 void QCPGraph::setChannelFillGraph (QCPGraph ∗ *targetGraph*)

Sets the target graph for filling the area between this graph and *targetGraph* with the current brush (setBrush).

When *targetGraph* is set to 0, a normal graph fill to the zero-value-line will be shown. To disable any filling, set the brush to Qt::NoBrush.

See also

setBrush

7.42.5.19 void QCPGraph::setData (QCPDataMap ∗ *data*, bool *copy* = `false`)

Replaces the current data with the provided *data*.

If *copy* is set to true, data points in *data* will only be copied. if false, the graph takes ownership of the passed data and replaces the internal data pointer with it. This is significantly faster than copying for large datasets.

Alternatively, you can also access and modify the graph's data via the data method, which returns a pointer to the internal QCPDataMap.

7.42.5.20 void QCPGraph::setData (const QVector< double > & *key*, const QVector< double > & *value*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *key* and *value* pairs. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.42.5.21 void QCPGraph::setDataBothError (const QVector< double > & *key*, const QVector< double > & *value*, const QVector< double > & *keyError*, const QVector< double > & *valueError*)

Replaces the current data with the provided points in *key* and *value* pairs. Additionally the symmetrical key and value errors of the data points are set to the values in *keyError* and *valueError*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

For asymmetrical errors (plus different from minus), see the overloaded version of this function.

7.42.5.22 void **QCPGraph::setDataBothError** (const QVector< double > & *key,* const QVector< double > & *value,* const QVector< double > & *keyErrorMinus,* const QVector< double > & *keyErrorPlus,* const QVector< double > & *valueErrorMinus,* const QVector< double > & *valueErrorPlus*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *key* and *value* pairs. Additionally the negative key and value errors of the data points are set to the values in *keyErrorMinus* and *valueErrorMinus*. The positive key and value errors are set to the values in *keyErrorPlus* *valueErrorPlus*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.42.5.23 void **QCPGraph::setDataKeyError** (const QVector< double > & *key,* const QVector< double > & *value,* const QVector< double > & *keyError*)

Replaces the current data with the provided points in *key* and *value* pairs. Additionally the symmetrical key error of the data points are set to the values in *keyError*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

For asymmetrical errors (plus different from minus), see the overloaded version of this function.

7.42.5.24 void **QCPGraph::setDataKeyError** (const QVector< double > & *key,* const QVector< double > & *value,* const QVector< double > & *keyErrorMinus,* const QVector< double > & *keyErrorPlus*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *key* and *value* pairs. Additionally the negative key error of the data points are set to the values in *keyErrorMinus*, the positive key error to *keyErrorPlus*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.42.5.25 void **QCPGraph::setDataValueError** (const QVector< double > & *key,* const QVector< double > & *value,* const QVector< double > & *valueError*)

Replaces the current data with the provided points in *key* and *value* pairs. Additionally the symmetrical value error of the data points are set to the values in *valueError*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

For asymmetrical errors (plus different from minus), see the overloaded version of this function.

7.42.5.26 void QCPGraph::setDataValueError (const QVector< double > & *key,* const QVector< double > & *value,* const QVector< double > & *valueErrorMinus,* const QVector< double > & *valueErrorPlus*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Replaces the current data with the provided points in *key* and *value* pairs. Additionally the negative value error of the data points are set to the values in *valueErrorMinus*, the positive value error to *valueErrorPlus*. For error bars to show appropriately, see setErrorType. The provided vectors should have equal length. Else, the number of added points will be the size of the smallest vector.

7.42.5.27 void QCPGraph::setErrorBarSize (double *size*)

Sets the width of the handles at both ends of an error bar in pixels.

7.42.5.28 void QCPGraph::setErrorBarSkipSymbol (bool *enabled*)

If *enabled* is set to true, the error bar will not be drawn as a solid line under the scatter symbol but leave some free space around the symbol.

This feature uses the current scatter size (QCPScatterStyle::setSize) to determine the size of the area to leave blank. So when drawing Pixmaps as scatter points (QCPScatterStyle::ssPixmap), the scatter size must be set manually to a value corresponding to the size of the Pixmap, if the error bars should leave gaps to its boundaries.

setErrorType, setErrorBarSize, setScatterStyle

7.42.5.29 void QCPGraph::setErrorPen (const QPen & *pen*)

Sets the pen with which the error bars will be drawn.

See also

setErrorBarSize, setErrorType

7.42.5.30 void QCPGraph::setErrorType (ErrorType *errorType*)

Sets which kind of error bars (Key Error, Value Error or both) should be drawn on each data point. If you set *errorType* to something other than etNone, make sure to actually pass error data via the specific setData functions along with the data points (e.g. setDataValueError, setDataKeyError, setDataBothError).

See also

ErrorType

7.42.5.31 void QCPGraph::setLineStyle (LineStyle *ls*)

Sets how the single data points are connected in the plot. For scatter-only plots, set *ls* to lsNone and setScatterStyle to the desired scatter style.

See also

setScatterStyle

7.42.5.32 void QCPGraph::setScatterStyle (const QCPScatterStyle & *style*)

Sets the visual appearance of single data points in the plot. If set to QCPScatterStyle::ssNone, no scatter points are drawn (e.g. for line-only-plots with appropriate line style).

See also

QCPScatterStyle, setLineStyle

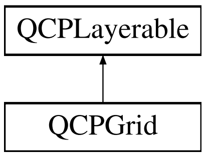
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.43 QCPGrid Class Reference

Responsible for drawing the grid of a QCPAxis.

Inheritance diagram for QCPGrid:



Public Member Functions

- QCPGrid (QCPAxis ∗parentAxis)
- bool **subGridVisible** () const
- bool **antialiasedSubGrid** () const
- bool **antialiasedZeroLine** () const
- QPen **pen** () const
- QPen **subGridPen** () const
- QPen **zeroLinePen** () const
- void setSubGridVisible (bool visible)

- void setAntialiasedSubGrid (bool enabled)
- void setAntialiasedZeroLine (bool enabled)
- void setPen (const QPen &pen)
- void setSubGridPen (const QPen &pen)
- void setZeroLinePen (const QPen &pen)

Protected Member Functions

- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- void **drawGridLines** (QCPPainter ∗painter) const
- void **drawSubGridLines** (QCPPainter ∗painter) const

Protected Attributes

- bool **mSubGridVisible**
- bool **mAntialiasedSubGrid**
- bool **mAntialiasedZeroLine**
- QPen **mPen**
- QPen **mSubGridPen**
- QPen **mZeroLinePen**
- QCPAxis ∗ **mParentAxis**

Friends

- class **QCPAxis**

7.43.1 Detailed Description

Responsible for drawing the grid of a QCPAxis.

This class is tightly bound to QCPAxis. Every axis owns a grid instance and uses it to draw the grid lines, sub grid lines and zero-line. You can interact with the grid of an axis via QCPAxis::grid. Normally, you don't need to create an instance of QCPGrid yourself.

The axis and grid drawing was split into two classes to allow them to be placed on different layers (both QCPAxis and QCPGrid inherit from QCPLayerable). Thus it is possible to have the grid in the background and the axes in the foreground, and any plottables/items in between. This described situation is the default setup, see the QCP-Layer documentation.

7.43.2 Constructor & Destructor Documentation

7.43.2.1 QCPGrid::QCPGrid (QCPAxis $*$ *parentAxis*)

Creates a QCPGrid instance and sets default values.

You shouldn't instantiate grids on their own, since every QCPAxis brings its own QCP-Grid.

7.43.3 Member Function Documentation

7.43.3.1 void QCPGrid::setAntialiasedSubGrid (bool *enabled*)

Sets whether sub grid lines are drawn antialiased.

7.43.3.2 void QCPGrid::setAntialiasedZeroLine (bool *enabled*)

Sets whether zero lines are drawn antialiased.

7.43.3.3 void QCPGrid::setPen (const QPen & *pen*)

Sets the pen with which (major) grid lines are drawn.

7.43.3.4 void QCPGrid::setSubGridPen (const QPen & *pen*)

Sets the pen with which sub grid lines are drawn.

7.43.3.5 void QCPGrid::setSubGridVisible (bool *visible*)

Sets whether grid lines at sub tick marks are drawn.

See also

setSubGridPen

7.43.3.6 void QCPGrid::setZeroLinePen (const QPen & *pen*)

Sets the pen with which zero lines are drawn.

Zero lines are lines at value coordinate 0 which may be drawn with a different pen than other grid lines. To disable zero lines and just draw normal grid lines at zero, set *pen* to Qt::NoPen.

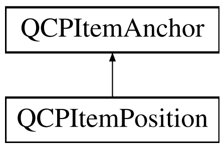
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.44 QCPItemAnchor Class Reference

An anchor of an item to which positions can be attached to.

Inheritance diagram for QCPItemAnchor:



Public Member Functions

- QCPItemAnchor (QCustomPlot ∗parentPlot, QCPAbstractItem ∗parentItem, const QString name, int anchorId=-1)
- QString **name** () const
- virtual QPointF pixelPoint () const

Protected Member Functions

- virtual QCPItemPosition ∗ toQCPItemPosition ()
- void **addChildX** (QCPItemPosition ∗pos)
- void **removeChildX** (QCPItemPosition ∗pos)
- void **addChildY** (QCPItemPosition ∗pos)
- void **removeChildY** (QCPItemPosition ∗pos)

Protected Attributes

- QString **mName**
- QCustomPlot ∗ **mParentPlot**
- QCPAbstractItem ∗ **mParentItem**
- int **mAnchorId**
- QSet< QCPItemPosition ∗ > **mChildrenX**
- QSet< QCPItemPosition ∗ > **mChildrenY**

Friends

- class **QCPItemPosition**

7.44.1 Detailed Description

An anchor of an item to which positions can be attached to.

An item (QCPAbstractItem) may have one or more anchors. Unlike QCPItemPosition, an anchor doesn't control anything on its item, but provides a way to tie other items via their positions to the anchor.

For example, a QCPItemRect is defined by its positions *topLeft* and *bottomRight*. - Additionally it has various anchors like *top*, *topRight* or *bottomLeft* etc. So you can attach the *start* (which is a QCPItemPosition) of a QCPItemLine to one of the anchors by calling QCPItemPosition::setParentAnchor on *start*, passing the wanted anchor of the QCPItemRect. This way the start of the line will now always follow the respective anchor location on the rect item.

Note that QCPItemPosition derives from QCPItemAnchor, so every position can also serve as an anchor to other positions.

To learn how to provide anchors in your own item subclasses, see the subclassing section of the QCPAbstractItem documentation.

7.44.2 Constructor & Destructor Documentation

7.44.2.1 QCPItemAnchor::QCPItemAnchor (QCustomPlot ∗ *parentPlot*, QCPAbstractItem ∗ *parentItem*, const QString *name*, int *anchorId* = `-1`)

Creates a new QCPItemAnchor. You shouldn't create QCPItemAnchor instances directly, even if you want to make a new item subclass. Use QCPAbstractItem::createAnchor instead, as explained in the subclassing section of the QCPAbstractItem documentation.

7.44.3 Member Function Documentation

7.44.3.1 QPointF QCPItemAnchor::pixelPoint ( ) const `[virtual]`

Returns the final absolute pixel position of the QCPItemAnchor on the QCustomPlot surface.

The pixel information is internally retrieved via QCPAbstractItem::anchorPixelPosition of the parent item, QCPItemAnchor is just an intermediary.

Reimplemented in QCPItemPosition.

7.44.3.2 QCPItemPosition ∗ QCPItemAnchor::toQCPItemPosition ( ) `[inline, protected, virtual]`

Returns 0 if this instance is merely a QCPItemAnchor, and a valid pointer of type QCPItemPosition∗ if it actually is a QCPItemPosition (which is a subclass of QCPItemAnchor).

This safe downcast functionality could also be achieved with a dynamic_cast. However, QCustomPlot avoids dynamic_cast to work with projects that don't have RTTI support enabled (e.g. -fno-rtti flag with gcc compiler).

Reimplemented in QCPItemPosition.

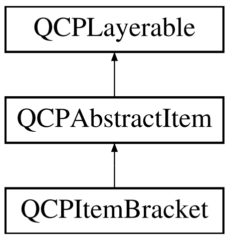
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.45 QCPItemBracket Class Reference

A bracket for referencing/highlighting certain parts in the plot.

Inheritance diagram for QCPItemBracket:



Public Types

- enum BracketStyle { bsSquare, bsRound, bsCurly, bsCalligraphic }

Public Member Functions

- QCPItemBracket (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- double **length** () const
- BracketStyle **style** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setLength (double length)
- void setStyle (BracketStyle style)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **left**
- QCPItemPosition ∗const **right**
- QCPItemAnchor ∗const **center**

Protected Types

- enum **AnchorIndex** { **aiCenter** }

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- QPen **mainPen** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- double **mLength**
- BracketStyle **mStyle**

7.45.1 Detailed Description

A bracket for referencing/highlighting certain parts in the plot.

It has two positions, *left* and *right*, which define the span of the bracket. If *left* is actually farther to the left than *right*, the bracket is opened to the bottom, as shown in the example image.

The bracket supports multiple styles via setStyle. The length, i.e. how far the bracket stretches away from the embraced span, can be controlled with setLength.

Demonstrating the effect of different values for setLength, for styles bsCalligraphic and bsSquare. Anchors and positions are displayed for reference.

It provides an anchor *center*, to allow connection of other items, e.g. an arrow (QCPItemLine or QCPItemCurve) or a text label (QCPItemText), to the bracket.

7.45.2 Member Enumeration Documentation

7.45.2.1 enum QCPItemBracket::BracketStyle

Enumerator:

bsSquare A brace with angled edges.

bsRound A brace with round edges.

bsCurly A curly brace.

bsCalligraphic A curly brace with varying stroke width giving a calligraphic impression.

7.45.3 Constructor & Destructor Documentation

7.45.3.1 QCPItemBracket::QCPItemBracket (QCustomPlot ∗ *parentPlot*)

Creates a bracket item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.45.4 Member Function Documentation

7.45.4.1 double QCPItemBracket::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.45.4.2 void **QCPItemBracket::setLength** (double *length*)

Sets the *length* in pixels how far the bracket extends in the direction towards the embraced span of the bracket (i.e. perpendicular to the *left*-*right*-direction)

Demonstrating the effect of different values for setLength, for styles bsCalligraphic and bsSquare. Anchors and positions are displayed for reference.

7.45.4.3 void **QCPItemBracket::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw the bracket.

Note that when the style is bsCalligraphic, only the color will be taken from the pen, the stroke and width are ignored. To change the apparent stroke width of a calligraphic bracket, use setLength, which has a similar effect.

See also

setSelectedPen

7.45.4.4 void **QCPItemBracket::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used to draw the bracket when selected

See also

setPen, setSelected

7.45.4.5 void **QCPItemBracket::setStyle** (**QCPItemBracket::BracketStyle** *style*)

Sets the style of the bracket, i.e. the shape/visual appearance.

See also

setPen

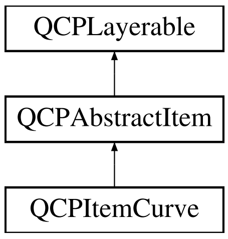
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.46 QCPItemCurve Class Reference

A curved line from one point to another.

Inheritance diagram for QCPItemCurve:



Public Member Functions

- QCPItemCurve (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- QCPLineEnding **head** () const
- QCPLineEnding **tail** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setHead (const QCPLineEnding &head)
- void setTail (const QCPLineEnding &tail)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **start**
- QCPItemPosition ∗const **startDir**
- QCPItemPosition ∗const **endDir**
- QCPItemPosition ∗const **end**

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- QPen **mainPen** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- QCPLineEnding **mHead**
- QCPLineEnding **mTail**

7.46.1 Detailed Description

A curved line from one point to another.

It has four positions, *start* and *end*, which define the end points of the line, and two control points which define the direction the line exits from the start and the direction from which it approaches the end: *startDir* and *endDir*.

With setHead and setTail you may set different line ending styles, e.g. to create an arrow.

Often it is desirable for the control points to stay at fixed relative positions to the start/end point. This can be achieved by setting the parent anchor e.g. of *startDir* simply to *start*, and then specify the desired pixel offset with QCPItemPosition::setCoords on *startDir*.

7.46.2 Constructor & Destructor Documentation

7.46.2.1 QCPItemCurve::QCPItemCurve (QCustomPlot ∗ *parentPlot*)

Creates a curve item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.46.3 Member Function Documentation

7.46.3.1 double QCPItemCurve::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.46.3.2 void **QCPItemCurve::setHead** (const **QCPLineEnding** & *head*)

Sets the line ending style of the head. The head corresponds to the *end* position.

Note that due to the overloaded QCPLineEnding constructor, you may directly specify a QCPLineEnding::EndingStyle here, e.g.

```
setHead(QCPLineEnding::esSpikeArrow)
```

See also

setTail

7.46.3.3 void **QCPItemCurve::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line

See also

setSelectedPen

7.46.3.4 void **QCPItemCurve::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line when selected

See also

setPen, setSelected

7.46.3.5 void QCPItemCurve::setTail (const QCPLineEnding & *tail*)

Sets the line ending style of the tail. The tail corresponds to the *start* position.

Note that due to the overloaded QCPLineEnding constructor, you may directly specify a QCPLineEnding::EndingStyle here, e.g.

```
setTail(QCPLineEnding::esSpikeArrow)
```

See also

setHead

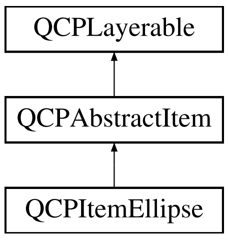
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.47 QCPItemEllipse Class Reference

An ellipse.

Inheritance diagram for QCPItemEllipse:



Public Member Functions

- QCPItemEllipse (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- QBrush **brush** () const
- QBrush **selectedBrush** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setSelectedBrush (const QBrush &brush)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **topLeft**
- QCPItemPosition ∗const **bottomRight**
- QCPItemAnchor ∗const **topLeftRim**
- QCPItemAnchor ∗const **top**
- QCPItemAnchor ∗const **topRightRim**
- QCPItemAnchor ∗const **right**
- QCPItemAnchor ∗const **bottomRightRim**
- QCPItemAnchor ∗const **bottom**
- QCPItemAnchor ∗const **bottomLeftRim**
- QCPItemAnchor ∗const **left**
- QCPItemAnchor ∗const **center**

Protected Types

- enum **AnchorIndex** { **aiTopLeftRim**, **aiTop**, **aiTopRightRim**, **aiRight**, **aiBottomRightRim**, **aiBottom**, **aiBottomLeftRim**, **aiLeft**, **aiCenter** }

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- QPen **mainPen** () const
- QBrush **mainBrush** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- QBrush **mBrush**
- QBrush **mSelectedBrush**

7.47.1 Detailed Description

An ellipse.

It has two positions, *topLeft* and *bottomRight*, which define the rect the ellipse will be drawn in.

7.47.2 Constructor & Destructor Documentation

7.47.2.1 QCPItemEllipse::QCPItemEllipse (QCustomPlot ∗ *parentPlot*)

Creates an ellipse item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.47.3 Member Function Documentation

7.47.3.1 double QCPItemEllipse::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.47.3.2 void QCPItemEllipse::setBrush (const QBrush & *brush*)

Sets the brush that will be used to fill the ellipse. To disable filling, set *brush* to Qt::NoBrush.

See also

setSelectedBrush, setPen

7.47.3.3 void QCPItemEllipse::setPen (const QPen & *pen*)

Sets the pen that will be used to draw the line of the ellipse

See also

setSelectedPen, setBrush

7.47.3.4 void QCPItemEllipse::setSelectedBrush (const QBrush & *brush*)

Sets the brush that will be used to fill the ellipse when selected. To disable filling, set *brush* to Qt::NoBrush.

See also

setBrush

7.47.3.5 void QCPItemEllipse::setSelectedPen (const QPen & *pen*)

Sets the pen that will be used to draw the line of the ellipse when selected

See also

setPen, setSelected

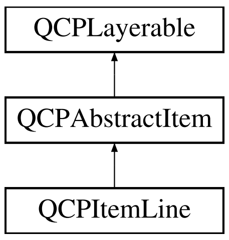
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.48 QCPItemLine Class Reference

A line from one point to another.

Inheritance diagram for QCPItemLine:



Public Member Functions

- QCPItemLine (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- QCPLineEnding **head** () const
- QCPLineEnding **tail** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setHead (const QCPLineEnding &head)
- void setTail (const QCPLineEnding &tail)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **start**
- QCPItemPosition ∗const **end**

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- QLineF **getRectClippedLine** (const QVector2D &start, const QVector2D &end, const QRect &rect) const
- QPen **mainPen** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- QCPLineEnding **mHead**
- QCPLineEnding **mTail**

7.48.1 Detailed Description

A line from one point to another.

It has two positions, *start* and *end*, which define the end points of the line.

With setHead and setTail you may set different line ending styles, e.g. to create an arrow.

7.48.2 Constructor & Destructor Documentation

7.48.2.1 QCPItemLine::QCPItemLine (QCustomPlot ∗ *parentPlot*)

Creates a line item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.48.3 Member Function Documentation

7.48.3.1 double QCPItemLine::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.48.3.2 void **QCPItemLine::setHead** (const **QCPLineEnding** & *head*)

Sets the line ending style of the head. The head corresponds to the *end* position.

Note that due to the overloaded QCPLineEnding constructor, you may directly specify a QCPLineEnding::EndingStyle here, e.g.

```
setHead(QCPLineEnding::esSpikeArrow)
```

See also

setTail

7.48.3.3 void **QCPItemLine::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line

See also

setSelectedPen

7.48.3.4 void **QCPItemLine::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line when selected

See also

setPen, setSelected

7.48.3.5 void **QCPItemLine::setTail** (const **QCPLineEnding** & *tail*)

Sets the line ending style of the tail. The tail corresponds to the *start* position.

Note that due to the overloaded QCPLineEnding constructor, you may directly specify a QCPLineEnding::EndingStyle here, e.g.

```
setTail(QCPLineEnding::esSpikeArrow)
```

See also

setHead

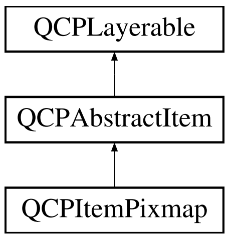
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.49 QCPItemPixmap Class Reference

An arbitrary pixmap.

Inheritance diagram for QCPItemPixmap:



Public Member Functions

- QCPItemPixmap (QCustomPlot ∗parentPlot)
- QPixmap **pixmap** () const
- bool **scaled** () const
- Qt::AspectRatioMode **aspectRatioMode** () const
- Qt::TransformationMode **transformationMode** () const
- QPen **pen** () const
- QPen **selectedPen** () const
- void setPixmap (const QPixmap &pixmap)
- void setScaled (bool scaled, Qt::AspectRatioMode aspectRatioMode=Qt::KeepAspectRatio, Qt::TransformationMode transformationMode=Qt::SmoothTransformation)
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **topLeft**
- QCPItemPosition ∗const **bottomRight**
- QCPItemAnchor ∗const **top**
- QCPItemAnchor ∗const **topRight**
- QCPItemAnchor ∗const **right**
- QCPItemAnchor ∗const **bottom**
- QCPItemAnchor ∗const **bottomLeft**
- QCPItemAnchor ∗const **left**

Protected Types

- enum **AnchorIndex** { **aiTop**, **aiTopRight**, **aiRight**, **aiBottom**, **aiBottomLeft**, **aiLeft** }

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- void **updateScaledPixmap** (QRect finalRect=QRect(), bool flipHorz=false, bool flipVert=false)
- QRect **getFinalRect** (bool ∗flippedHorz=0, bool ∗flippedVert=0) const
- QPen **mainPen** () const

Protected Attributes

- QPixmap **mPixmap**
- QPixmap **mScaledPixmap**
- bool **mScaled**
- bool **mScaledPixmapInvalidated**
- Qt::AspectRatioMode **mAspectRatioMode**
- Qt::TransformationMode **mTransformationMode**
- QPen **mPen**
- QPen **mSelectedPen**

7.49.1 Detailed Description

An arbitrary pixmap.

It has two positions, *topLeft* and *bottomRight*, which define the rectangle the pixmap will be drawn in. Depending on the scale setting (setScaled), the pixmap will be either scaled to fit the rectangle or be drawn aligned to the topLeft position.

If scaling is enabled and *topLeft* is further to the bottom/right than *bottomRight* (as shown on the right side of the example image), the pixmap will be flipped in the respective orientations.

7.49.2 Constructor & Destructor Documentation

7.49.2.1 QCPItemPixmap::QCPItemPixmap (QCustomPlot ∗ *parentPlot*)

Creates a rectangle item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.49.3 Member Function Documentation

7.49.3.1 double **QCPItemPixmap::selectTest** (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.49.3.2 void **QCPItemPixmap::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw a border around the pixmap.

See also

setSelectedPen, setBrush

7.49.3.3 void QCPItemPixmap::setPixmap (const QPixmap & *pixmap*)

Sets the pixmap that will be displayed.

7.49.3.4 void QCPItemPixmap::setScaled (bool *scaled*, Qt::AspectRatioMode *aspectRatioMode* = `Qt::KeepAspectRatio`, Qt::TransformationMode *transformationMode* = `Qt::SmoothTransformation`)

Sets whether the pixmap will be scaled to fit the rectangle defined by the *topLeft* and *bottomRight* positions.

7.49.3.5 void QCPItemPixmap::setSelectedPen (const QPen & *pen*)

Sets the pen that will be used to draw a border around the pixmap when selected

See also

setPen, setSelected

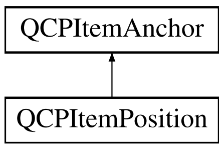
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.50 QCPItemPosition Class Reference

Manages the position of an item.

Inheritance diagram for QCPItemPosition:



Public Types

- enum PositionType { ptAbsolute, ptViewportRatio, ptAxisRectRatio, ptPlotCoords }

Public Member Functions

- QCPItemPosition (QCustomPlot ∗parentPlot, QCPAbstractItem ∗parentItem, const QString name)

- PositionType type () const
- PositionType **typeX** () const
- PositionType **typeY** () const
- QCPItemAnchor ∗ parentAnchor () const
- QCPItemAnchor ∗ **parentAnchorX** () const
- QCPItemAnchor ∗ **parentAnchorY** () const
- double **key** () const
- double **value** () const
- QPointF **coords** () const
- QCPAxis ∗ **keyAxis** () const
- QCPAxis ∗ **valueAxis** () const
- QCPAxisRect ∗ **axisRect** () const
- virtual QPointF pixelPoint () const
- void setType (PositionType type)
- void setTypeX (PositionType type)
- void setTypeY (PositionType type)
- bool setParentAnchor (QCPItemAnchor ∗parentAnchor, bool keepPixelPosition=false)
- bool setParentAnchorX (QCPItemAnchor ∗parentAnchor, bool keepPixelPosition=false)
- bool setParentAnchorY (QCPItemAnchor ∗parentAnchor, bool keepPixelPosition=false)
- void setCoords (double key, double value)
- void setCoords (const QPointF &coords)
- void setAxes (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- void setAxisRect (QCPAxisRect ∗axisRect)
- void setPixelPoint (const QPointF &pixelPoint)

Protected Member Functions

- virtual QCPItemPosition ∗ toQCPItemPosition ()

Protected Attributes

- PositionType **mPositionTypeX**
- PositionType **mPositionTypeY**
- QPointer< QCPAxis > **mKeyAxis**
- QPointer< QCPAxis > **mValueAxis**
- QPointer< QCPAxisRect > **mAxisRect**
- double **mKey**
- double **mValue**
- QCPItemAnchor ∗ **mParentAnchorX**
- QCPItemAnchor ∗ **mParentAnchorY**

7.50.1 Detailed Description

Manages the position of an item.

Every item has at least one public QCPItemPosition member pointer which provides ways to position the item on the QCustomPlot surface. Some items have multiple positions, for example QCPItemRect has two: *topLeft* and *bottomRight*.

QCPItemPosition has a type (PositionType) that can be set with setType. This type defines how coordinates passed to setCoords are to be interpreted, e.g. as absolute pixel coordinates, as plot coordinates of certain axes, etc. For more advanced plots it is also possible to assign different types per X/Y coordinate of the position (see setTypeX, setTypeY). This way an item could be positioned at a fixed pixel distance from the top in the Y direction, while following a plot coordinate in the X direction.

A QCPItemPosition may have a parent QCPItemAnchor, see setParentAnchor. This way you can tie multiple items together. If the QCPItemPosition has a parent, its coordinates (setCoords) are considered to be absolute pixels in the reference frame of the parent anchor, where (0, 0) means directly ontop of the parent anchor. For example, You could attach the *start* position of a QCPItemLine to the *bottom* anchor of a QCPItemText to make the starting point of the line always be centered under the text label, no matter where the text is moved to. For more advanced plots, it is possible to assign different parent anchors per X/Y coordinate of the position, see setParentAnchorX, setParentAnchorY. This way an item could follow another item in the X direction but stay at a fixed position in the Y direction. Or even follow item A in X, and item B in Y.

Note that every QCPItemPosition inherits from QCPItemAnchor and thus can itself be used as parent anchor for other positions.

To set the apparent pixel position on the QCustomPlot surface directly, use setPixelPoint. This works no matter what type this QCPItemPosition is or what parent-child situation it is in, as setPixelPoint transforms the coordinates appropriately, to make the position appear at the specified pixel values.

7.50.2 Member Enumeration Documentation

7.50.2.1 enum QCPItemPosition::PositionType

Defines the ways an item position can be specified. Thus it defines what the numbers passed to setCoords actually mean.

See also

setType

Enumerator:

ptAbsolute Static positioning in pixels, starting from the top left corner of the viewport/widget.

ptViewportRatio Static positioning given by a fraction of the viewport size. For example, if you call setCoords(0, 0), the position will be at the top left corner of the viewport/widget. setCoords(1, 1) will be at the bottom right corner, setCoords(0.5, 0) will be horizontally centered and vertically at the top of the viewport/widget, etc.

ptAxisRectRatio Static positioning given by a fraction of the axis rect size (see setAxisRect). For example, if you call setCoords(0, 0), the position will be at the top left corner of the axis rect. setCoords(1, 1) will be at the bottom right corner, setCoords(0.5, 0) will be horizontally centered and vertically at the top of the axis rect, etc. You can also go beyond the axis rect by providing negative coordinates or coordinates larger than 1.

ptPlotCoords Dynamic positioning at a plot coordinate defined by two axes (see setAxes).

7.50.3 Constructor & Destructor Documentation

7.50.3.1 QCPItemPosition::QCPItemPosition (QCustomPlot ∗ *parentPlot,* QCPAbstractItem ∗ *parentItem,* const QString *name*)

Creates a new QCPItemPosition. You shouldn't create QCPItemPosition instances directly, even if you want to make a new item subclass. Use QCPAbstractItem::createPosition instead, as explained in the subclassing section of the QCPAbstractItem documentation.

7.50.4 Member Function Documentation

7.50.4.1 QCPItemAnchor ∗ QCPItemPosition::parentAnchor ( ) const [inline]

Returns the current parent anchor.

If different parent anchors were set for X and Y (setParentAnchorX, setParentAnchorY), this method returns the parent anchor of the Y coordinate. In that case rather use *parentAnchorX()* and *parentAnchorY()*.

See also

setParentAnchor

7.50.4.2 QPointF QCPItemPosition::pixelPoint ( ) const [virtual]

Returns the final absolute pixel position of the QCPItemPosition on the QCustomPlot surface. It includes all effects of type (setType) and possible parent anchors (setParentAnchor).

See also

setPixelPoint

Reimplemented from QCPItemAnchor.

7.50.4.3 void QCPItemPosition::setAxes (QCPAxis ∗ *keyAxis,* QCPAxis ∗ *valueAxis*)

When setType is ptPlotCoords, this function may be used to specify the axes the coordinates set with setCoords relate to. By default they are set to the initial xAxis and yAxis of the QCustomPlot.

7.50.4.4 void QCPItemPosition::setAxisRect (QCPAxisRect ∗ *axisRect*)

When setType is ptAxisRectRatio, this function may be used to specify the axis rect the coordinates set with setCoords relate to. By default this is set to the main axis rect of the QCustomPlot.

7.50.4.5 void QCPItemPosition::setCoords (double *key,* double *value*)

Sets the coordinates of this QCPItemPosition. What the coordinates mean, is defined by the type (setType, setTypeX, setTypeY).

For example, if the type is ptAbsolute, *key* and *value* mean the x and y pixel position on the QCustomPlot surface. In that case the origin (0, 0) is in the top left corner of the QCustomPlot viewport. If the type is ptPlotCoords, *key* and *value* mean a point in the plot coordinate system defined by the axes set by setAxes. By default those are the QCustomPlot's xAxis and yAxis. See the documentation of setType for other available coordinate types and their meaning.

If different types were configured for X and Y (setTypeX, setTypeY), *key* and *value* must also be provided in the different coordinate systems. Here, the X type refers to *key*, and the Y type refers to *value*.

See also

setPixelPoint

7.50.4.6 void QCPItemPosition::setCoords (const QPointF & *pos*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the coordinates as a QPointF *pos* where pos.x has the meaning of *key* and pos.y the meaning of *value* of the setCoords(double key, double value) method.

7.50.4.7 bool QCPItemPosition::setParentAnchor (QCPItemAnchor ∗ *parentAnchor,* bool *keepPixelPosition* = `false`)

Sets the parent of this QCPItemPosition to *parentAnchor*. This means the position will now follow any position changes of the anchor. The local coordinate system of positions with a parent anchor always is absolute pixels, with (0, 0) being exactly on top of the parent anchor. (Hence the type shouldn't be set to ptPlotCoords for positions with parent anchors.)

if *keepPixelPosition* is true, the current pixel position of the QCPItemPosition is preserved during reparenting. If it's set to false, the coordinates are set to (0, 0), i.e. the position will be exactly on top of the parent anchor.

To remove this QCPItemPosition from any parent anchor, set *parentAnchor* to 0.

If the QCPItemPosition previously had no parent and the type is ptPlotCoords, the type is set to ptAbsolute, to keep the position in a valid state.

This method sets the parent anchor for both X and Y directions. It is also possible to set different parents for X and Y, see setParentAnchorX, setParentAnchorY.

7.50.4.8 bool QCPItemPosition::setParentAnchorX (QCPItemAnchor ∗ *parentAnchor,* bool *keepPixelPosition* = `false`)

This method sets the parent anchor of the X coordinate to *parentAnchor*.

For a detailed description of what a parent anchor is, see the documentation of setParentAnchor.

See also

setParentAnchor, setParentAnchorY

7.50.4.9 bool QCPItemPosition::setParentAnchorY (QCPItemAnchor ∗ *parentAnchor,* bool *keepPixelPosition* = `false`)

This method sets the parent anchor of the Y coordinate to *parentAnchor*.

For a detailed description of what a parent anchor is, see the documentation of setParentAnchor.

See also

setParentAnchor, setParentAnchorX

7.50.4.10 void QCPItemPosition::setPixelPoint (const QPointF & *pixelPoint*)

Sets the apparent pixel position. This works no matter what type (setType) this QCPItemPosition is or what parent-child situation it is in, as coordinates are transformed appropriately, to make the position finally appear at the specified pixel values.

Only if the type is ptAbsolute and no parent anchor is set, this function's effect is identical to that of setCoords.

See also

pixelPoint, setCoords

7.50.4.11 void QCPItemPosition::setType (QCPItemPosition::PositionType *type*)

Sets the type of the position. The type defines how the coordinates passed to setCoords should be handled and how the QCPItemPosition should behave in the plot.

The possible values for *type* can be separated in two main categories:

- The position is regarded as a point in plot coordinates. This corresponds to ptPlotCoords and requires two axes that define the plot coordinate system. They can be specified with setAxes. By default, the QCustomPlot's x- and yAxis are used.

- The position is fixed on the QCustomPlot surface, i.e. independent of axis ranges. This corresponds to all other types, i.e. ptAbsolute, ptViewportRatio and ptAxisRectRatio. They differ only in the way the absolute position is described, see the documentation of PositionType for details. For ptAxisRectRatio, note that you can specify the axis rect with setAxisRect. By default this is set to the main axis rect.

Note that the position type ptPlotCoords is only available (and sensible) when the position has no parent anchor (setParentAnchor).

If the type is changed, the apparent pixel position on the plot is preserved. This means the coordinates as retrieved with coords() and set with setCoords may change in the process.

This method sets the type for both X and Y directions. It is also possible to set different types for X and Y, see setTypeX, setTypeY.

7.50.4.12 void QCPItemPosition::setTypeX (QCPItemPosition::PositionType *type*)

This method sets the position type of the X coordinate to *type*.

For a detailed description of what a position type is, see the documentation of setType.

See also

setType, setTypeY

7.50.4.13 void QCPItemPosition::setTypeY (QCPItemPosition::PositionType *type*)

This method sets the position type of the Y coordinate to *type*.

For a detailed description of what a position type is, see the documentation of setType.

See also

setType, setTypeX

7.50.4.14 virtual QCPItemPosition∗ QCPItemPosition::toQCPItemPosition ( ) [inline, protected, virtual]

Returns 0 if this instance is merely a QCPItemAnchor, and a valid pointer of type QCPItemPosition∗ if it actually is a QCPItemPosition (which is a subclass of QCPItemAnchor).

This safe downcast functionality could also be achieved with a dynamic_cast. However, QCustomPlot avoids dynamic_cast to work with projects that don't have RTTI support enabled (e.g. -fno-rtti flag with gcc compiler).

Reimplemented from QCPItemAnchor.

7.50.4.15 QCPItemPosition::PositionType ∗ QCPItemPosition::type ( ) const [inline]

Returns the current position type.

If different types were set for X and Y (setTypeX, setTypeY), this method returns the type of the X coordinate. In that case rather use *typeX()* and *typeY()*.

See also

setType

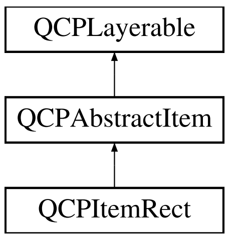
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.51 QCPItemRect Class Reference

A rectangle.

Inheritance diagram for QCPItemRect:



Public Member Functions

- QCPItemRect (QCustomPlot ∗parentPlot)

- QPen **pen** () const
- QPen **selectedPen** () const
- QBrush **brush** () const
- QBrush **selectedBrush** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setSelectedBrush (const QBrush &brush)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **topLeft**
- QCPItemPosition ∗const **bottomRight**
- QCPItemAnchor ∗const **top**
- QCPItemAnchor ∗const **topRight**
- QCPItemAnchor ∗const **right**
- QCPItemAnchor ∗const **bottom**
- QCPItemAnchor ∗const **bottomLeft**
- QCPItemAnchor ∗const **left**

Protected Types

- enum **AnchorIndex** { **aiTop**, **aiTopRight**, **aiRight**, **aiBottom**, **aiBottomLeft**, **aiLeft** }

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- QPen **mainPen** () const
- QBrush **mainBrush** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- QBrush **mBrush**
- QBrush **mSelectedBrush**

7.51.1 Detailed Description

A rectangle.

It has two positions, *topLeft* and *bottomRight*, which define the rectangle.

7.51.2 Constructor & Destructor Documentation

7.51.2.1 QCPItemRect::QCPItemRect (QCustomPlot * *parentPlot*)

Creates a rectangle item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.51.3 Member Function Documentation

7.51.3.1 double QCPItemRect::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant * *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99*selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.51.3.2 void **QCPItemRect::setBrush** (const QBrush & *brush*)

Sets the brush that will be used to fill the rectangle. To disable filling, set *brush* to Qt::NoBrush.

See also

setSelectedBrush, setPen

7.51.3.3 void **QCPItemRect::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line of the rectangle

See also

setSelectedPen, setBrush

7.51.3.4 void **QCPItemRect::setSelectedBrush** (const QBrush & *brush*)

Sets the brush that will be used to fill the rectangle when selected. To disable filling, set *brush* to Qt::NoBrush.

See also

setBrush

7.51.3.5 void **QCPItemRect::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line of the rectangle when selected

See also

setPen, setSelected

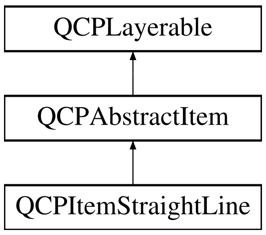
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.52 QCPItemStraightLine Class Reference

A straight line that spans infinitely in both directions.

Inheritance diagram for QCPItemStraightLine:



Public Member Functions

- QCPItemStraightLine (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **point1**
- QCPItemPosition ∗const **point2**

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- double **distToStraightLine** (const QVector2D &point1, const QVector2D &vec, const QVector2D &point) const
- QLineF **getRectClippedStraightLine** (const QVector2D &point1, const Q-Vector2D &vec, const QRect &rect) const
- QPen **mainPen** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**

7.52.1 Detailed Description

A straight line that spans infinitely in both directions.

It has two positions, *point1* and *point2*, which define the straight line.

7.52.2 Constructor & Destructor Documentation

7.52.2.1 QCPItemStraightLine::QCPItemStraightLine (QCustomPlot ∗ *parentPlot*)

Creates a straight line item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.52.3 Member Function Documentation

7.52.3.1 double QCPItemStraightLine::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = `0`) const `[virtual]`

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.52.3.2 void QCPItemStraightLine::setPen (const QPen & *pen*)

Sets the pen that will be used to draw the line

See also

setSelectedPen

7.52.3.3 void QCPItemStraightLine::setSelectedPen (const QPen & *pen*)

Sets the pen that will be used to draw the line when selected

See also

setPen, setSelected

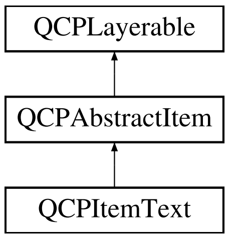
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.53 QCPItemText Class Reference

A text label.

Inheritance diagram for QCPItemText:



Public Member Functions

- QCPItemText (QCustomPlot ∗parentPlot)
- QColor **color** () const
- QColor **selectedColor** () const
- QPen **pen** () const
- QPen **selectedPen** () const
- QBrush **brush** () const
- QBrush **selectedBrush** () const
- QFont **font** () const

- QFont **selectedFont** () const
- QString **text** () const
- Qt::Alignment **positionAlignment** () const
- Qt::Alignment **textAlignment** () const
- double **rotation** () const
- QMargins **padding** () const
- void setColor (const QColor &color)
- void setSelectedColor (const QColor &color)
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setSelectedBrush (const QBrush &brush)
- void setFont (const QFont &font)
- void setSelectedFont (const QFont &font)
- void setText (const QString &text)
- void setPositionAlignment (Qt::Alignment alignment)
- void setTextAlignment (Qt::Alignment alignment)
- void setRotation (double degrees)
- void setPadding (const QMargins &padding)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Public Attributes

- QCPItemPosition ∗const **position**
- QCPItemAnchor ∗const **topLeft**
- QCPItemAnchor ∗const **top**
- QCPItemAnchor ∗const **topRight**
- QCPItemAnchor ∗const **right**
- QCPItemAnchor ∗const **bottomRight**
- QCPItemAnchor ∗const **bottom**
- QCPItemAnchor ∗const **bottomLeft**
- QCPItemAnchor ∗const **left**

Protected Types

- enum **AnchorIndex** { **aiTopLeft**, **aiTop**, **aiTopRight**, **aiRight**, **aiBottomRight**, **aiBottom**, **aiBottomLeft**, **aiLeft** }

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QPointF **anchorPixelPoint** (int anchorId) const
- QPointF **getTextDrawPoint** (const QPointF &pos, const QRectF &rect, Qt::-Alignment positionAlignment) const

- QFont **mainFont** () const
- QColor **mainColor** () const
- QPen **mainPen** () const
- QBrush **mainBrush** () const

Protected Attributes

- QColor **mColor**
- QColor **mSelectedColor**
- QPen **mPen**
- QPen **mSelectedPen**
- QBrush **mBrush**
- QBrush **mSelectedBrush**
- QFont **mFont**
- QFont **mSelectedFont**
- QString **mText**
- Qt::Alignment **mPositionAlignment**
- Qt::Alignment **mTextAlignment**
- double **mRotation**
- QMargins **mPadding**

7.53.1 Detailed Description

A text label.

Its position is defined by the member *position* and the setting of setPositionAlignment. The latter controls which part of the text rect shall be aligned with *position*.

The text alignment itself (i.e. left, center, right) can be controlled with setTextAlignment.

The text may be rotated around the *position* point with setRotation.

7.53.2 Constructor & Destructor Documentation

7.53.2.1 QCPItemText::QCPItemText (QCustomPlot ∗ *parentPlot*)

Creates a text item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.53.3 Member Function Documentation

7.53.3.1 double QCPItemText::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.53.3.2 void **QCPItemText::setBrush** (const QBrush & *brush*)

Sets the brush that will be used do fill the background of the text. To disable the background, set *brush* to Qt::NoBrush.

See also

setSelectedBrush, setPen, setPadding

7.53.3.3 void **QCPItemText::setColor** (const QColor & *color*)

Sets the color of the text.

7.53.3.4 void **QCPItemText::setFont** (const QFont & *font*)

Sets the font of the text.

See also

setSelectedFont, setColor

7.53.3.5 void **QCPItemText::setPadding** (const QMargins & *padding*)

Sets the distance between the border of the text rectangle and the text. The appearance (and visibility) of the text rectangle can be controlled with setPen and setBrush.

7.53.3.6 void **QCPItemText::setPen** (const QPen & *pen*)

Sets the pen that will be used do draw a rectangular border around the text. To disable the border, set *pen* to Qt::NoPen.

See also

setSelectedPen, setBrush, setPadding

7.53.3.7 void **QCPItemText::setPositionAlignment** (Qt::Alignment *alignment*)

Sets which point of the text rect shall be aligned with *position*.

Examples:

- If *alignment* is `Qt::AlignHCenter | Qt::AlignTop`, the text will be positioned such that the top of the text rect will be horizontally centered on *position*.
- If *alignment* is `Qt::AlignLeft | Qt::AlignBottom`, *position* will indicate the bottom left corner of the text rect.

If you want to control the alignment of (multi-lined) text within the text rect, use setText-Alignment.

7.53.3.8 void **QCPItemText::setRotation** (double *degrees*)

Sets the angle in degrees by which the text (and the text rectangle, if visible) will be rotated around *position*.

7.53.3.9 void **QCPItemText::setSelectedBrush** (const QBrush & *brush*)

Sets the brush that will be used do fill the background of the text, when the item is selected. To disable the background, set *brush* to Qt::NoBrush.

See also

setBrush

7.53.3.10 void **QCPItemText::setSelectedColor** (const QColor & *color*)

Sets the color of the text that will be used when the item is selected.

7.53.3.11 void **QCPItemText::setSelectedFont** (const QFont & *font*)

Sets the font of the text that will be used when the item is selected.

See also

setFont

7.53.3.12 void **QCPItemText::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used do draw a rectangular border around the text, when the item is selected. To disable the border, set *pen* to Qt::NoPen.

See also

setPen

7.53.3.13 void **QCPItemText::setText** (const QString & *text*)

Sets the text that will be displayed. Multi-line texts are supported by inserting a line break character, e.g. '

'.

See also

setFont, setColor, setTextAlignment

7.53.3.14 void **QCPItemText::setTextAlignment** (Qt::Alignment *alignment*)

Controls how (multi-lined) text is aligned inside the text rect (typically Qt::AlignLeft, Qt::-AlignCenter or Qt::AlignRight).

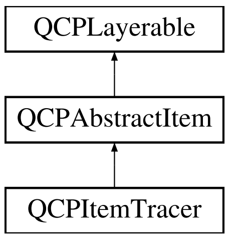
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.54 QCPItemTracer Class Reference

Item that sticks to QCPGraph data points.

Inheritance diagram for QCPItemTracer:



Public Types

- enum TracerStyle { tsNone, tsPlus, tsCrosshair, tsCircle, tsSquare }

Public Member Functions

- QCPItemTracer (QCustomPlot ∗parentPlot)
- QPen **pen** () const
- QPen **selectedPen** () const
- QBrush **brush** () const
- QBrush **selectedBrush** () const
- double **size** () const
- TracerStyle **style** () const
- QCPGraph ∗ **graph** () const
- double **graphKey** () const
- bool **interpolating** () const
- void setPen (const QPen &pen)
- void setSelectedPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setSelectedBrush (const QBrush &brush)
- void setSize (double size)
- void setStyle (TracerStyle style)
- void setGraph (QCPGraph ∗graph)
- void setGraphKey (double key)
- void setInterpolating (bool enabled)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- void updatePosition ()

Public Attributes

- QCPItemPosition ∗const **position**

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- QPen **mainPen** () const
- QBrush **mainBrush** () const

Protected Attributes

- QPen **mPen**
- QPen **mSelectedPen**
- QBrush **mBrush**
- QBrush **mSelectedBrush**
- double **mSize**
- TracerStyle **mStyle**
- QCPGraph ∗ **mGraph**
- double **mGraphKey**
- bool **mInterpolating**

7.54.1 Detailed Description

Item that sticks to QCPGraph data points.

The tracer can be connected with a QCPGraph via setGraph. Then it will automatically adopt the coordinate axes of the graph and update its *position* to be on the graph's data. This means the key stays controllable via setGraphKey, but the value will follow the graph data. If a QCPGraph is connected, note that setting the coordinates of the tracer item directly via *position* will have no effect because they will be overriden in the next redraw (this is when the coordinate update happens).

If the specified key in setGraphKey is outside the key bounds of the graph, the tracer will stay at the corresponding end of the graph.

With setInterpolating you may specify whether the tracer may only stay exactly on data points or whether it interpolates data points linearly, if given a key that lies between two data points of the graph.

The tracer has different visual styles, see setStyle. It is also possible to make the tracer have no own visual appearance (set the style to tsNone), and just connect other item positions to the tracer *position* (used as an anchor) via QCPItemPosition::setParent-Anchor.

Note

The tracer position is only automatically updated upon redraws. So when the data of the graph changes and immediately afterwards (without a redraw) the a position coordinates of the tracer are retrieved, they will not reflect the updated data of the graph. In this case updatePosition must be called manually, prior to reading the tracer coordinates.

7.54.2 Member Enumeration Documentation

7.54.2.1 enum QCPItemTracer::TracerStyle

The different visual appearances a tracer item can have. Some styles size may be controlled with setSize.

See also

setStyle

Enumerator:

tsNone The tracer is not visible.

tsPlus A plus shaped crosshair with limited size.

tsCrosshair A plus shaped crosshair which spans the complete axis rect.

tsCircle A circle.

tsSquare A square.

7.54.3 Constructor & Destructor Documentation

7.54.3.1 QCPItemTracer::QCPItemTracer (QCustomPlot ∗ *parentPlot*)

Creates a tracer item and sets default values.

The constructed item can be added to the plot with QCustomPlot::addItem.

7.54.4 Member Function Documentation

7.54.4.1 double QCPItemTracer::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractItem.

7.54.4.2 void **QCPItemTracer::setBrush** (const QBrush & *brush*)

Sets the brush that will be used to draw any fills of the tracer

See also

setSelectedBrush, setPen

7.54.4.3 void **QCPItemTracer::setGraph** (**QCPGraph** ∗ *graph*)

Sets the QCPGraph this tracer sticks to. The tracer *position* will be set to type QCPItemPosition::ptPlotCoords and the axes will be set to the axes of *graph*.

To free the tracer from any graph, set *graph* to 0. The tracer *position* can then be placed freely like any other item position. This is the state the tracer will assume when its graph gets deleted while still attached to it.

See also

setGraphKey

7.54.4.4 void **QCPItemTracer::setGraphKey** (double *key*)

Sets the key of the graph's data point the tracer will be positioned at. This is the only free coordinate of a tracer when attached to a graph.

Depending on setInterpolating, the tracer will be either positioned on the data point closest to *key*, or will stay exactly at *key* and interpolate the value linearly.

See also

setGraph, setInterpolating

7.54.4.5 void **QCPItemTracer::setInterpolating** (bool *enabled*)

Sets whether the value of the graph's data points shall be interpolated, when positioning the tracer.

If *enabled* is set to false and a key is given with setGraphKey, the tracer is placed on the data point of the graph which is closest to the key, but which is not necessarily exactly there. If *enabled* is true, the tracer will be positioned exactly at the specified key, and the appropriate value will be interpolated from the graph's data points linearly.

See also

setGraph, setGraphKey

7.54.4.6 void **QCPItemTracer::setPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line of the tracer

See also

setSelectedPen, setBrush

7.54.4.7 void **QCPItemTracer::setSelectedBrush** (const QBrush & *brush*)

Sets the brush that will be used to draw any fills of the tracer, when selected.

See also

setBrush, setSelected

7.54.4.8 void **QCPItemTracer::setSelectedPen** (const QPen & *pen*)

Sets the pen that will be used to draw the line of the tracer when selected

See also

setPen, setSelected

7.54.4.9 void **QCPItemTracer::setSize** (double *size*)

Sets the size of the tracer in pixels, if the style supports setting a size (e.g. tsSquare does, tsCrosshair does not).

7.54.4.10 void QCPItemTracer::setStyle (QCPItemTracer::TracerStyle *style*)

Sets the style/visual appearance of the tracer.

If you only want to use the tracer *position* as an anchor for other items, set *style* to tsNone.

7.54.4.11 void QCPItemTracer::updatePosition ( )

If the tracer is connected with a graph (setGraph), this function updates the tracer's *position* to reside on the graph data, depending on the configured key (setGraphKey).

It is called automatically on every redraw and normally doesn't need to be called manually. One exception is when you want to read the tracer coordinates via *position* and are not sure that the graph's data (or the tracer key with setGraphKey) hasn't changed since the last redraw. In that situation, call this function before accessing *position*, to make sure you don't get out-of-date coordinates.

If there is no graph set on this tracer, this function does nothing.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.55 QCPLayer Class Reference

A layer that may contain objects, to control the rendering order.

Public Member Functions

- QCPLayer (QCustomPlot ∗parentPlot, const QString &layerName)
- QCustomPlot ∗ **parentPlot** () const
- QString **name** () const
- int index () const
- QList< QCPLayerable ∗ > children () const
- bool **visible** () const
- void setVisible (bool visible)

Protected Member Functions

- void **addChild** (QCPLayerable ∗layerable, bool prepend)
- void **removeChild** (QCPLayerable ∗layerable)

Protected Attributes

- QCustomPlot $*$ **mParentPlot**
- QString **mName**
- int **mIndex**
- QList$<$ QCPLayerable $*$ $>$ **mChildren**
- bool **mVisible**

Friends

- class **QCustomPlot**
- class **QCPLayerable**

7.55.1 Detailed Description

A layer that may contain objects, to control the rendering order.

The Layering system of QCustomPlot is the mechanism to control the rendering order of the elements inside the plot.

It is based on the two classes QCPLayer and QCPLayerable. QCustomPlot holds an ordered list of one or more instances of QCPLayer (see QCustomPlot::addLayer, -QCustomPlot::layer, QCustomPlot::moveLayer, etc.). When replotting, QCustomPlot goes through the list of layers bottom to top and successively draws the layerables of the layers.

A QCPLayer contains an ordered list of QCPLayerable instances. QCPLayerable is an abstract base class from which almost all visible objects derive, like axes, grids, graphs, items, etc.

Initially, QCustomPlot has five layers: "background", "grid", "main", "axes" and "legend" (in that order). The top two layers "axes" and "legend" contain the default axes and legend, so they will be drawn on top. In the middle, there is the "main" layer. It is initially empty and set as the current layer (see QCustomPlot::setCurrentLayer). This means, all new plottables, items etc. are created on this layer by default. Then comes the "grid" layer which contains the QCPGrid instances (which belong tightly to QCPAxis, see Q-CPAxis::grid). The Axis rect background shall be drawn behind everything else, thus the default QCPAxisRect instance is placed on the "background" layer. Of course, the layer affiliation of the individual objects can be changed as required (QCPLayerable-::setLayer).

Controlling the ordering of objects is easy: Create a new layer in the position you want it to be, e.g. above "main", with QCustomPlot::addLayer. Then set the current layer with QCustomPlot::setCurrentLayer to that new layer and finally create the objects normally. They will be placed on the new layer automatically, due to the current layer setting. -Alternatively you could have also ignored the current layer setting and just moved the objects with QCPLayerable::setLayer to the desired layer after creating them.

It is also possible to move whole layers. For example, If you want the grid to be shown in front of all plottables/items on the "main" layer, just move it above "main" with QCustom-Plot::moveLayer.

The rendering order within one layer is simply by order of creation or insertion. The item created last (or added last to the layer), is drawn on top of all other objects on that layer.

When a layer is deleted, the objects on it are not deleted with it, but fall on the layer below the deleted layer, see QCustomPlot::removeLayer.

7.55.2 Constructor & Destructor Documentation

7.55.2.1 QCPLayer::QCPLayer (QCustomPlot ∗ *parentPlot,* const QString & *layerName*)

Creates a new QCPLayer instance.

Normally you shouldn't directly instantiate layers, use QCustomPlot::addLayer instead.

Warning

It is not checked that *layerName* is actually a unique layer name in *parentPlot*. This check is only performed by QCustomPlot::addLayer.

7.55.3 Member Function Documentation

7.55.3.1 QList< QCPLayerable ∗ > QCPLayer::children ( ) const [inline]

Returns a list of all layerables on this layer. The order corresponds to the rendering order: layerables with higher indices are drawn above layerables with lower indices.

7.55.3.2 int QCPLayer::index ( ) const [inline]

Returns the index this layer has in the QCustomPlot. The index is the integer number by which this layer can be accessed via QCustomPlot::layer.

Layers with higher indices will be drawn above layers with lower indices.

7.55.3.3 void QCPLayer::setVisible (bool *visible*)

Sets whether this layer is visible or not. If *visible* is set to false, all layerables on this layer will be invisible.

This function doesn't change the visibility property of the layerables (QCPLayerable::setVisible), but the QCPLayerable::realVisibility of each layerable takes the visibility of the parent layer into account.

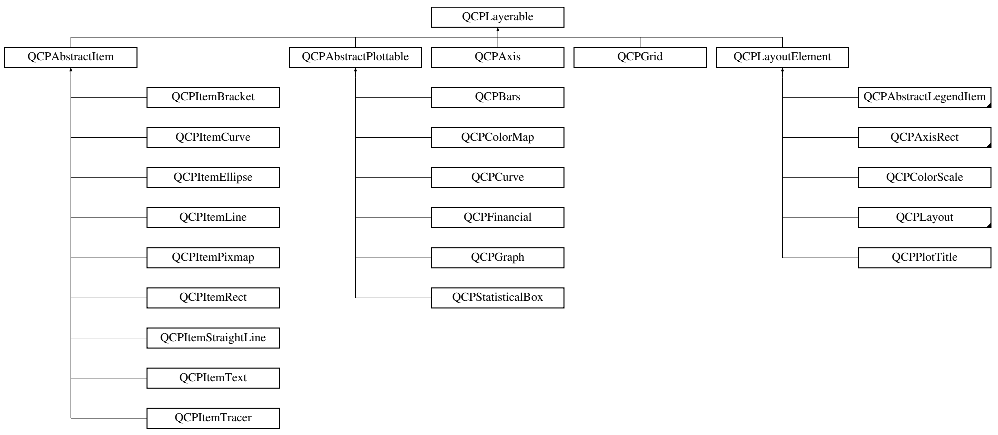
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.56 QCPLayerable Class Reference

Base class for all drawable objects.

Inheritance diagram for QCPLayerable:



Signals

- void layerChanged (QCPLayer ∗newLayer)

Public Member Functions

- QCPLayerable (QCustomPlot ∗plot, QString targetLayer=QString(), QCP-Layerable ∗parentLayerable=0)
- bool **visible** () const
- QCustomPlot ∗ **parentPlot** () const
- QCPLayerable ∗ parentLayerable () const
- QCPLayer ∗ **layer** () const
- bool **antialiased** () const
- void setVisible (bool on)
- Q_SLOT bool setLayer (QCPLayer ∗layer)
- bool setLayer (const QString &layerName)
- void setAntialiased (bool enabled)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- bool realVisibility () const

Protected Member Functions

- virtual void **parentPlotInitialized** (QCustomPlot ∗parentPlot)
- virtual QCP::Interaction **selectionCategory** () const

- virtual QRect **clipRect** () const
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const =0
- virtual void **draw** (QCPPainter ∗painter)=0
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- void **initializeParentPlot** (QCustomPlot ∗parentPlot)
- void **setParentLayerable** (QCPLayerable ∗parentLayerable)
- bool **moveToLayer** (QCPLayer ∗layer, bool prepend)
- void **applyAntialiasingHint** (QCPPainter ∗painter, bool localAntialiased, QCP::AntialiasedElement overrideElement) const

Protected Attributes

- bool **mVisible**
- QCustomPlot ∗ **mParentPlot**
- QPointer< QCPLayerable > **mParentLayerable**
- QCPLayer ∗ **mLayer**
- bool **mAntialiased**

Friends

- class **QCustomPlot**
- class **QCPAxisRect**

7.56.1 Detailed Description

Base class for all drawable objects.

This is the abstract base class most visible objects derive from, e.g. plottables, axes, grid etc.

Every layerable is on a layer (QCPLayer) which allows controlling the rendering order by stacking the layers accordingly.

For details about the layering mechanism, see the QCPLayer documentation.

7.56.2 Constructor & Destructor Documentation

7.56.2.1 QCPLayerable::QCPLayerable (QCustomPlot ∗ *plot,* QString *targetLayer =* `QString()`*,* QCPLayerable ∗ *parentLayerable =* `0`)

Creates a new QCPLayerable instance.

Since QCPLayerable is an abstract base class, it can't be instantiated directly. Use one of the derived classes.

If *plot* is provided, it automatically places itself on the layer named *targetLayer*. If *targetLayer* is an empty string, it places itself on the current layer of the plot (see QCustomPlot::setCurrentLayer).

It is possible to provide 0 as *plot*. In that case, you should assign a parent plot at a later time with initializeParentPlot.

The layerable's parent layerable is set to *parentLayerable*, if provided. Direct layerable parents are mainly used to control visibility in a hierarchy of layerables. This means a layerable is only drawn, if all its ancestor layerables are also visible. Note that *parentLayerable* does not become the QObject-parent (for memory management) of this layerable, *plot* does. It is not uncommon to set the QObject-parent to something else in the constructors of QCPLayerable subclasses, to guarantee a working destruction hierarchy.

7.56.3 Member Function Documentation

7.56.3.1 void **QCPLayerable::layerChanged** (**QCPLayer** ∗ *newLayer*) `[signal]`

This signal is emitted when the layer of this layerable changes, i.e. this layerable is moved to a different layer.

See also

setLayer

7.56.3.2 **QCPLayerable** ∗ **QCPLayerable::parentLayerable** ( ) const `[inline]`

Returns the parent layerable of this layerable. The parent layerable is used to provide visibility hierarchies in conjunction with the method realVisibility. This way, layerables only get drawn if their parent layerables are visible, too.

Note that a parent layerable is not necessarily also the QObject parent for memory management. Further, a layerable doesn't always have a parent layerable, so this function may return 0.

A parent layerable is set implicitly with when placed inside layout elements and doesn't need to be set manually by the user.

7.56.3.3 bool **QCPLayerable::realVisibility** ( ) const

Returns whether this layerable is visible, taking the visibility of the layerable parent and the visibility of the layer this layerable is on into account. This is the method that is consulted to decide whether a layerable shall be drawn or not.

If this layerable has a direct layerable parent (usually set via hierarchies implemented in subclasses, like in the case of QCPLayoutElement), this function returns true only if this layerable has its visibility set to true and the parent layerable's realVisibility returns true.

If this layerable doesn't have a direct layerable parent, returns the state of this layerable's visibility.

7.56.3.4 double QCPLayerable::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Reimplemented in QCPItemBracket, QCPItemTracer, QCPItemPixmap, QCPItemEllipse, QCPItemText, QCPItemRect, QCPItemCurve, QCPItemLine, QCPItemStraightLine, QCPFinancial, QCPColorMap, QCPStatisticalBox, QCPBars, QCPCurve, QCPGraph, QCPPlotTitle, QCPLegend, QCPAbstractLegendItem, QCPAbstractItem, QCPAbstractPlottable, QCPAxis, QCPLayoutInset, and QCPLayoutElement.

7.56.3.5 void QCPLayerable::setAntialiased (bool *enabled*)

Sets whether this object will be drawn antialiased or not.

Note that antialiasing settings may be overridden by QCustomPlot::setAntialiasedElements and QCustomPlot::setNotAntialiasedElements.

7.56.3.6 bool QCPLayerable::setLayer (QCPLayer ∗ *layer*)

Sets the *layer* of this layerable object. The object will be placed on top of the other objects already on *layer*.

If *layer* is 0, this layerable will not be on any layer and thus not appear in the plot (or interact/receive events).

Returns true if the layer of this layerable was successfully changed to *layer*.

7.56.3.7 bool QCPLayerable::setLayer (const QString & *layerName*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the layer of this layerable object by name

Returns true on success, i.e. if *layerName* is a valid layer name.

7.56.3.8 void QCPLayerable::setVisible (bool *on*)

Sets the visibility of this layerable object. If an object is not visible, it will not be drawn on the QCustomPlot surface, and user interaction with it (e.g. click and selection) is not possible.

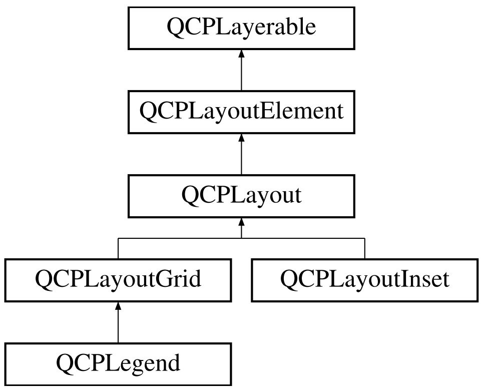
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.57 QCPLayout Class Reference

The abstract base class for layouts.

Inheritance diagram for QCPLayout:



Public Member Functions

- QCPLayout ()
- virtual void update (UpdatePhase phase)
- virtual QList< QCPLayoutElement ∗ > elements (bool recursive) const
- virtual int elementCount () const =0
- virtual QCPLayoutElement ∗ elementAt (int index) const =0
- virtual QCPLayoutElement ∗ takeAt (int index)=0
- virtual bool take (QCPLayoutElement ∗element)=0
- virtual void simplify ()
- bool removeAt (int index)
- bool remove (QCPLayoutElement ∗element)
- void clear ()

Protected Member Functions

- virtual void **updateLayout** ()
- void sizeConstraintsChanged () const
- void **adoptElement** (QCPLayoutElement ∗el)
- void **releaseElement** (QCPLayoutElement ∗el)
- QVector< int > **getSectionSizes** (QVector< int > maxSizes, QVector< int > minSizes, QVector< double > stretchFactors, int totalSize) const

Friends

- class **QCPLayoutElement**

7.57.1 Detailed Description

The abstract base class for layouts.

This is an abstract base class for layout elements whose main purpose is to define the position and size of other child layout elements. In most cases, layouts don't draw anything themselves (but there are exceptions to this, e.g. QCPLegend).

QCPLayout derives from QCPLayoutElement, and thus can itself be nested in other layouts.

QCPLayout introduces a common interface for accessing and manipulating the child elements. Those functions are most notably elementCount, elementAt, takeAt, take, simplify, removeAt, remove and clear. Individual subclasses may add more functions to this interface which are more specialized to the form of the layout. For example, Q-CPLayoutGrid adds functions that take row and column indices to access cells of the layout grid more conveniently.

Since this is an abstract base class, you can't instantiate it directly. Rather use one of its subclasses like QCPLayoutGrid or QCPLayoutInset.

For a general introduction to the layout system, see the dedicated documentation page The Layout System.

7.57.2 Constructor & Destructor Documentation

7.57.2.1 QCPLayout::QCPLayout ( ) [explicit]

Creates an instance of QCPLayout and sets default values. Note that since QCPLayout is an abstract base class, it can't be instantiated directly.

7.57.3 Member Function Documentation

7.57.3.1 void QCPLayout::clear ( )

Removes and deletes all layout elements in this layout. Finally calls simplify to make sure all empty cells are collapsed.

See also

remove, removeAt

7.57.3.2 QCPLayoutElement ∗ QCPLayout::elementAt (int *index*) const [pure virtual]

Returns the element in the cell with the given *index*. If *index* is invalid, returns 0.

Note that even if *index* is valid, the respective cell may be empty in some layouts (e.g. QCPLayoutGrid), so this function may return 0 in those cases. You may use this function to check whether a cell is empty or not.

See also

elements, elementCount, takeAt

Implemented in QCPLayoutInset, and QCPLayoutGrid.

7.57.3.3 int QCPLayout::elementCount ( ) const [pure virtual]

Returns the number of elements/cells in the layout.

See also

elements, elementAt

Implemented in QCPLayoutInset, and QCPLayoutGrid.

7.57.3.4 QList< QCPLayoutElement ∗ > QCPLayout::elements (bool *recursive*) const [virtual]

Returns a list of all child elements in this layout element. If *recursive* is true, all sub-child elements are included in the list, too.

Warning

There may be entries with value 0 in the returned list. (For example, QCPLayout-Grid may have empty cells which yield 0 at the respective index.)

Reimplemented from QCPLayoutElement.

Reimplemented in QCPLayoutGrid.

7.57.3.5 bool QCPLayout::remove (QCPLayoutElement ∗ *element*)

Removes and deletes the provided *element*. Returns true on success. If *element* is not in the layout, returns false.

This function internally uses takeAt to remove the element from the layout and then deletes the element. Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

removeAt, take

7.57.3.6 bool QCPLayout::removeAt (int *index*)

Removes and deletes the element at the provided *index*. Returns true on success. If *index* is invalid or points to an empty cell, returns false.

This function internally uses takeAt to remove the element from the layout and then deletes the returned element. Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

remove, takeAt

7.57.3.7 void QCPLayout::simplify ( ) `[virtual]`

Simplifies the layout by collapsing empty cells. The exact behavior depends on subclasses, the default implementation does nothing.

Not all layouts need simplification. For example, QCPLayoutInset doesn't use explicit simplification while QCPLayoutGrid does.

Reimplemented in QCPLayoutInset, and QCPLayoutGrid.

7.57.3.8 void QCPLayout::sizeConstraintsChanged ( ) const [protected]

Subclasses call this method to report changed (minimum/maximum) size constraints.

If the parent of this layout is again a QCPLayout, forwards the call to the parent's size-ConstraintsChanged. If the parent is a QWidget (i.e. is the QCustomPlot::plotLayout of QCustomPlot), calls QWidget::updateGeometry, so if the QCustomPlot widget is inside a Qt QLayout, it may update itself and resize cells accordingly.

7.57.3.9 bool QCPLayout::take (QCPLayoutElement * *element*) [pure virtual]

Removes the specified *element* from the layout and returns true on success.

If the *element* isn't in this layout, returns false.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

takeAt

Implemented in QCPLayoutInset, and QCPLayoutGrid.

7.57.3.10 QCPLayoutElement * QCPLayout::takeAt (int *index*) [pure virtual]

Removes the element with the given *index* from the layout and returns it.

If the *index* is invalid or the cell with that index is empty, returns 0.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

elementAt, take

Implemented in QCPLayoutInset, and QCPLayoutGrid.

7.57.3.11 void QCPLayout::update (UpdatePhase *phase*) [virtual]

First calls the QCPLayoutElement::update base class implementation to update the margins on this layout.

Then calls updateLayout which subclasses reimplement to reposition and resize their cells.

Finally, update is called on all child elements.

Reimplemented from QCPLayoutElement.

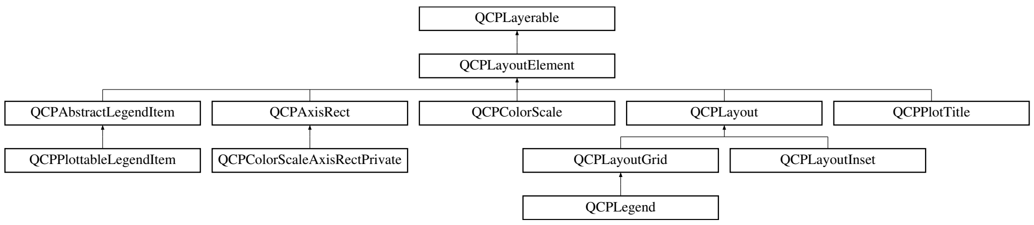
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.58 QCPLayoutElement Class Reference

The abstract base class for all objects that form the layout system.

Inheritance diagram for QCPLayoutElement:



Public Types

- enum UpdatePhase { upPreparation, upMargins, upLayout }

Public Member Functions

- QCPLayoutElement (QCustomPlot ∗parentPlot=0)
- QCPLayout ∗ layout () const
- QRect rect () const
- QRect **outerRect** () const
- QMargins **margins** () const
- QMargins **minimumMargins** () const
- QCP::MarginSides **autoMargins** () const
- QSize **minimumSize** () const
- QSize **maximumSize** () const
- QCPMarginGroup ∗ **marginGroup** (QCP::MarginSide side) const
- QHash< QCP::MarginSide, QCPMarginGroup ∗ > **marginGroups** () const
- void setOuterRect (const QRect &rect)
- void setMargins (const QMargins &margins)
- void setMinimumMargins (const QMargins &margins)
- void setAutoMargins (QCP::MarginSides sides)
- void setMinimumSize (const QSize &size)
- void setMinimumSize (int width, int height)
- void setMaximumSize (const QSize &size)
- void setMaximumSize (int width, int height)

- void setMarginGroup (QCP::MarginSides sides, QCPMarginGroup ∗group)
- virtual void update (UpdatePhase phase)
- virtual QSize minimumSizeHint () const
- virtual QSize maximumSizeHint () const
- virtual QList< QCPLayoutElement ∗ > elements (bool recursive) const
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual int **calculateAutoMargin** (QCP::MarginSide side)
- virtual void mousePressEvent (QMouseEvent ∗event)
- virtual void mouseMoveEvent (QMouseEvent ∗event)
- virtual void mouseReleaseEvent (QMouseEvent ∗event)
- virtual void mouseDoubleClickEvent (QMouseEvent ∗event)
- virtual void wheelEvent (QWheelEvent ∗event)
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- virtual void **parentPlotInitialized** (QCustomPlot ∗parentPlot)

Protected Attributes

- QCPLayout ∗ **mParentLayout**
- QSize **mMinimumSize**
- QSize **mMaximumSize**
- QRect **mRect**
- QRect **mOuterRect**
- QMargins **mMargins**
- QMargins **mMinimumMargins**
- QCP::MarginSides **mAutoMargins**
- QHash< QCP::MarginSide, QCPMarginGroup ∗ > **mMarginGroups**

Friends

- class **QCustomPlot**
- class **QCPLayout**
- class **QCPMarginGroup**

7.58.1 Detailed Description

The abstract base class for all objects that form the layout system.

This is an abstract base class. As such, it can't be instantiated directly, rather use one of its subclasses.

A Layout element is a rectangular object which can be placed in layouts. It has an outer rect (QCPLayoutElement::outerRect) and an inner rect (QCPLayoutElement::rect). The difference between outer and inner rect is called its margin. The margin can either be set to automatic or manual (setAutoMargins) on a per-side basis. If a side is set to manual, that margin can be set explicitly with setMargins and will stay fixed at that value. If it's set to automatic, the layout element subclass will control the value itself (via calculateAutoMargin).

Layout elements can be placed in layouts (base class QCPLayout) like QCPLayoutGrid. The top level layout is reachable via QCustomPlot::plotLayout, and is a QCPLayoutGrid. Since QCPLayout itself derives from QCPLayoutElement, layouts can be nested.

Thus in QCustomPlot one can divide layout elements into two categories: The ones that are invisible by themselves, because they don't draw anything. Their only purpose is to manage the position and size of other layout elements. This category of layout elements usually use QCPLayout as base class. Then there is the category of layout elements which actually draw something. For example, QCPAxisRect, QCPLegend and QCPPlotTitle are of this category. This does not necessarily mean that the latter category can't have child layout elements. QCPLegend for instance, actually derives from QCPLayoutGrid and the individual legend items are child layout elements in the grid layout.

7.58.2 Member Enumeration Documentation

7.58.2.1 enum QCPLayoutElement::UpdatePhase

Defines the phases of the update process, that happens just before a replot. At each phase, update is called with the according UpdatePhase value.

Enumerator:

upPreparation Phase used for any type of preparation that needs to be done before margin calculation and layout.

upMargins Phase in which the margins are calculated and set.

upLayout Final phase in which the layout system places the rects of the elements.

7.58.3 Constructor & Destructor Documentation

7.58.3.1 QCPLayoutElement::QCPLayoutElement (QCustomPlot ∗ *parentPlot* = `0`) `[explicit]`

Creates an instance of QCPLayoutElement and sets default values.

7.58.4 Member Function Documentation

7.58.4.1 QList< QCPLayoutElement * > QCPLayoutElement::elements (bool *recursive*) const [virtual]

Returns a list of all child elements in this layout element. If *recursive* is true, all sub-child elements are included in the list, too.

Warning

There may be entries with value 0 in the returned list. (For example, QCPLayout-Grid may have empty cells which yield 0 at the respective index.)

Reimplemented in QCPAxisRect, QCPLayoutGrid, and QCPLayout.

7.58.4.2 QCPLayout * QCPLayoutElement::layout ( ) const [inline]

Returns the parent layout of this layout element.

7.58.4.3 QSize QCPLayoutElement::maximumSizeHint ( ) const [virtual]

Returns the maximum size this layout element (the inner rect) may be expanded to.

if a maximum size (setMaximumSize) was not set manually, parent layouts consult this function to determine the maximum allowed size of this layout element. (A manual maximum size is considered set if it is smaller than Qt's QWIDGETSIZE_MAX.)

Reimplemented in QCPPlotTitle, and QCPLayoutGrid.

7.58.4.4 QSize QCPLayoutElement::minimumSizeHint ( ) const [virtual]

Returns the minimum size this layout element (the inner rect) may be compressed to.

if a minimum size (setMinimumSize) was not set manually, parent layouts consult this function to determine the minimum allowed size of this layout element. (A manual minimum size is considered set if it is non-zero.)

Reimplemented in QCPPlotTitle, QCPPlottableLegendItem, and QCPLayoutGrid.

7.58.4.5 void QCPLayoutElement::mouseDoubleClickEvent (QMouseEvent * *event*) [inline, protected, virtual]

This event is called, if the mouse is double-clicked inside the outer rect of this layout element.

7.58.4.6 void QCPLayoutElement::mouseMoveEvent (QMouseEvent * *event*) [inline, protected, virtual]

This event is called, if the mouse is moved inside the outer rect of this layout element.

Reimplemented in QCPColorScale, and QCPAxisRect.

7.58.4.7 void QCPLayoutElement::mousePressEvent (QMouseEvent ∗ *event*) [inline, protected, virtual]

This event is called, if the mouse was pressed while being inside the outer rect of this layout element.

Reimplemented in QCPColorScale, and QCPAxisRect.

7.58.4.8 void QCPLayoutElement::mouseReleaseEvent (QMouseEvent ∗ *event*) [inline, protected, virtual]

This event is called, if the mouse was previously pressed inside the outer rect of this layout element and is now released.

Reimplemented in QCPColorScale, and QCPAxisRect.

7.58.4.9 QRect QCPLayoutElement::rect ( ) const [inline]

Returns the inner rect of this layout element. The inner rect is the outer rect (setOuterRect) shrinked by the margins (setMargins, setAutoMargins).

In some cases, the area between outer and inner rect is left blank. In other cases the margin area is used to display peripheral graphics while the main content is in the inner rect. This is where automatic margin calculation becomes interesting because it allows the layout element to adapt the margins to the peripheral graphics it wants to draw. - For example, QCPAxisRect draws the axis labels and tick labels in the margin area, thus needs to adjust the margins (if setAutoMargins is enabled) according to the space required by the labels of the axes.

7.58.4.10 double QCPLayoutElement::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = 0) const [virtual]

Layout elements are sensitive to events inside their outer rect. If *pos* is within the outer rect, this method returns a value corresponding to 0.99 times the parent plot's selection tolerance. However, layout elements are not selectable by default. So if *onlySelectable* is true, -1.0 is returned.

See QCPLayerable::selectTest for a general explanation of this virtual method.

QCPLayoutElement subclasses may reimplement this method to provide more specific selection test behaviour.

Reimplemented from QCPLayerable.

Reimplemented in QCPPlotTitle, QCPLegend, QCPAbstractLegendItem, and QCPLayoutInset.

7.58.4.11 void QCPLayoutElement::setAutoMargins (QCP::MarginSides *sides*)

Sets on which sides the margin shall be calculated automatically. If a side is calculated automatically, a minimum margin value may be provided with setMinimumMargins. If a side is set to be controlled manually, the value may be specified with setMargins.

Margin sides that are under automatic control may participate in a QCPMarginGroup (see setMarginGroup), to synchronize (align) it with other layout elements in the plot.

See also

setMinimumMargins, setMargins

7.58.4.12 void QCPLayoutElement::setMarginGroup (QCP::MarginSides *sides*, QCPMarginGroup ∗ *group*)

Sets the margin *group* of the specified margin *sides*.

Margin groups allow synchronizing specified margins across layout elements, see the documentation of QCPMarginGroup.

To unset the margin group of *sides*, set *group* to 0.

Note that margin groups only work for margin sides that are set to automatic (setAuto-Margins).

7.58.4.13 void QCPLayoutElement::setMargins (const QMargins & *margins*)

Sets the margins of this layout element. If setAutoMargins is disabled for some or all sides, this function is used to manually set the margin on those sides. Sides that are still set to be handled automatically are ignored and may have any value in *margins*.

The margin is the distance between the outer rect (controlled by the parent layout via setOuterRect) and the inner rect (which usually contains the main content of this layout element).

See also

setAutoMargins

7.58.4.14 void QCPLayoutElement::setMaximumSize (const QSize & *size*)

Sets the maximum size for the inner rect of this layout element. A parent layout tries to respect the *size* here by changing row/column sizes in the layout accordingly.

7.58.4.15 void QCPLayoutElement::setMaximumSize (int *width*, int *height*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the maximum size for the inner rect of this layout element.

7.58.4.16 void QCPLayoutElement::setMinimumMargins (const QMargins & *margins*)

If setAutoMargins is enabled on some or all margins, this function is used to provide minimum values for those margins.

The minimum values are not enforced on margin sides that were set to be under manual control via setAutoMargins.

See also

setAutoMargins

7.58.4.17 void QCPLayoutElement::setMinimumSize (const QSize & *size*)

Sets the minimum size for the inner rect of this layout element. A parent layout tries to respect the *size* here by changing row/column sizes in the layout accordingly.

If the parent layout size is not sufficient to satisfy all minimum size constraints of its child layout elements, the layout may set a size that is actually smaller than *size*. QCustomPlot propagates the layout's size constraints to the outside by setting its own minimum QWidget size accordingly, so violations of *size* should be exceptions.

7.58.4.18 void QCPLayoutElement::setMinimumSize (int *width,* int *height*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the minimum size for the inner rect of this layout element.

7.58.4.19 void QCPLayoutElement::setOuterRect (const QRect & *rect*)

Sets the outer rect of this layout element. If the layout element is inside a layout, the layout sets the position and size of this layout element using this function.

Calling this function externally has no effect, since the layout will overwrite any changes to the outer rect upon the next replot.

The layout element will adapt its inner rect by applying the margins inward to the outer rect.

See also

rect

7.58.4.20 void QCPLayoutElement::update (UpdatePhase *phase*) [virtual]

Updates the layout element and sub-elements. This function is automatically called before every replot by the parent layout element. It is called multiple times, once for every UpdatePhase. The phases are run through in the order of the enum values. For details about what happens at the different phases, see the documentation of Update-Phase.

Layout elements that have child elements should call the update method of their child elements, and pass the current *phase* unchanged.

The default implementation executes the automatic margin mechanism in the upMargins phase. Subclasses should make sure to call the base class implementation.

Reimplemented in QCPColorScale, QCPAxisRect, and QCPLayout.

7.58.4.21 void QCPLayoutElement::wheelEvent (QWheelEvent ∗ *event*) [inline, protected, virtual]

This event is called, if the mouse wheel is scrolled while the cursor is inside the rect of this layout element.

Reimplemented in QCPColorScale, and QCPAxisRect.

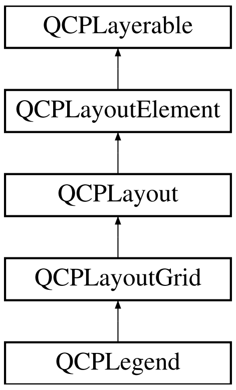
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.59 QCPLayoutGrid Class Reference

A layout that arranges child elements in a grid.

Inheritance diagram for QCPLayoutGrid:



Public Member Functions

- QCPLayoutGrid ()
- int rowCount () const
- int columnCount () const

- QList< double > **columnStretchFactors** () const
- QList< double > **rowStretchFactors** () const
- int **columnSpacing** () const
- int **rowSpacing** () const
- void setColumnStretchFactor (int column, double factor)
- void setColumnStretchFactors (const QList< double > &factors)
- void setRowStretchFactor (int row, double factor)
- void setRowStretchFactors (const QList< double > &factors)
- void setColumnSpacing (int pixels)
- void setRowSpacing (int pixels)
- virtual void **updateLayout** ()
- virtual int elementCount () const
- virtual QCPLayoutElement ∗ elementAt (int index) const
- virtual QCPLayoutElement ∗ takeAt (int index)
- virtual bool take (QCPLayoutElement ∗element)
- virtual QList< QCPLayoutElement ∗ > elements (bool recursive) const
- virtual void simplify ()
- virtual QSize minimumSizeHint () const
- virtual QSize maximumSizeHint () const
- QCPLayoutElement ∗ element (int row, int column) const
- bool addElement (int row, int column, QCPLayoutElement ∗element)
- bool hasElement (int row, int column)
- void expandTo (int newRowCount, int newColumnCount)
- void insertRow (int newIndex)
- void insertColumn (int newIndex)

Protected Member Functions

- void **getMinimumRowColSizes** (QVector< int > ∗minColWidths, QVector< int > ∗minRowHeights) const
- void **getMaximumRowColSizes** (QVector< int > ∗maxColWidths, QVector< int > ∗maxRowHeights) const

Protected Attributes

- QList< QList< QCPLayoutElement ∗ > > **mElements**
- QList< double > **mColumnStretchFactors**
- QList< double > **mRowStretchFactors**
- int **mColumnSpacing**
- int **mRowSpacing**

7.59.1 Detailed Description

A layout that arranges child elements in a grid.

Elements are laid out in a grid with configurable stretch factors (setColumnStretchFactor, setRowStretchFactor) and spacing (setColumnSpacing, setRowSpacing).

Elements can be added to cells via addElement. The grid is expanded if the specified row or column doesn't exist yet. Whether a cell contains a valid layout element can be checked with hasElement, that element can be retrieved with element. If rows and columns that only have empty cells shall be removed, call simplify. Removal of elements is either done by just adding the element to a different layout or by using the QCPLayout interface take or remove.

Row and column insertion can be performed with insertRow and insertColumn.

7.59.2 Constructor & Destructor Documentation

7.59.2.1 QCPLayoutGrid::QCPLayoutGrid ( ) `[explicit]`

Creates an instance of QCPLayoutGrid and sets default values.

7.59.3 Member Function Documentation

7.59.3.1 bool QCPLayoutGrid::addElement (int *row*, int *column*, QCPLayoutElement ∗ *element*)

Adds the *element* to cell with *row* and *column*. If *element* is already in a layout, it is first removed from there. If *row* or *column* don't exist yet, the layout is expanded accordingly.

Returns true if the element was added successfully, i.e. if the cell at *row* and *column* didn't already have an element.

See also

element, hasElement, take, remove

7.59.3.2 int QCPLayoutGrid::columnCount ( ) const

Returns the number of columns in the layout.

See also

rowCount

7.59.3.3 QCPLayoutElement ∗ QCPLayoutGrid::element (int *row*, int *column*) const

Returns the element in the cell in *row* and *column*.

Returns 0 if either the row/column is invalid or if the cell is empty. In those cases, a qDebug message is printed. To check whether a cell exists and isn't empty, use hasElement.

See also

addElement, hasElement

7.59.3.4 QCPLayoutElement ∗ QCPLayoutGrid::elementAt (int *index*) const `[virtual]`

Returns the element in the cell with the given *index*. If *index* is invalid, returns 0.

Note that even if *index* is valid, the respective cell may be empty in some layouts (e.g. QCPLayoutGrid), so this function may return 0 in those cases. You may use this function to check whether a cell is empty or not.

See also

elements, elementCount, takeAt

Implements QCPLayout.

7.59.3.5 int QCPLayoutGrid::elementCount ( ) const `[virtual]`

Returns the number of elements/cells in the layout.

See also

elements, elementAt

Implements QCPLayout.

7.59.3.6 QList< QCPLayoutElement ∗ > QCPLayoutGrid::elements (bool *recursive*) const `[virtual]`

Returns a list of all child elements in this layout element. If *recursive* is true, all sub-child elements are included in the list, too.

Warning

There may be entries with value 0 in the returned list. (For example, QCPLayoutGrid may have empty cells which yield 0 at the respective index.)

Reimplemented from QCPLayout.

7.59.3.7 void **QCPLayoutGrid::expandTo** (int *newRowCount,* int *newColumnCount*)

Expands the layout to have *newRowCount* rows and *newColumnCount* columns. So the last valid row index will be *newRowCount-1*, the last valid column index will be *newColumnCount-1*.

If the current column/row count is already larger or equal to *newColumnCount/newRowCount,* this function does nothing in that dimension.

Newly created cells are empty, new rows and columns have the stretch factor 1.

Note that upon a call to addElement, the layout is expanded automatically to contain the specified row and column, using this function.

See also

simplify

7.59.3.8 bool **QCPLayoutGrid::hasElement** (int *row,* int *column*)

Returns whether the cell at *row* and *column* exists and contains a valid element, i.e. isn't empty.

See also

element

7.59.3.9 void **QCPLayoutGrid::insertColumn** (int *newIndex*)

Inserts a new column with empty cells at the column index *newIndex*. Valid values for *newIndex* range from 0 (inserts a row at the left) to *rowCount* (appends a row at the right).

See also

insertRow

7.59.3.10 void **QCPLayoutGrid::insertRow** (int *newIndex*)

Inserts a new row with empty cells at the row index *newIndex*. Valid values for *newIndex* range from 0 (inserts a row at the top) to *rowCount* (appends a row at the bottom).

See also

insertColumn

7.59.3.11 QSize QCPLayoutGrid::maximumSizeHint ( ) const [virtual]

Returns the maximum size this layout element (the inner rect) may be expanded to.

if a maximum size (setMaximumSize) was not set manually, parent layouts consult this function to determine the maximum allowed size of this layout element. (A manual maximum size is considered set if it is smaller than Qt's QWIDGETSIZE_MAX.)

Reimplemented from QCPLayoutElement.

7.59.3.12 QSize QCPLayoutGrid::minimumSizeHint ( ) const [virtual]

Returns the minimum size this layout element (the inner rect) may be compressed to.

if a minimum size (setMinimumSize) was not set manually, parent layouts consult this function to determine the minimum allowed size of this layout element. (A manual minimum size is considered set if it is non-zero.)

Reimplemented from QCPLayoutElement.

7.59.3.13 int QCPLayoutGrid::rowCount ( ) const

Returns the number of rows in the layout.

See also

columnCount

7.59.3.14 void QCPLayoutGrid::setColumnSpacing (int *pixels*)

Sets the gap that is left blank between columns to *pixels*.

See also

setRowSpacing

7.59.3.15 void QCPLayoutGrid::setColumnStretchFactor (int *column*, double *factor*)

Sets the stretch *factor* of *column*.

Stretch factors control the relative sizes of rows and columns. Cells will not be resized beyond their minimum and maximum widths/heights (QCPLayoutElement::setMinimumSize, QCPLayoutElement::setMaximumSize), regardless of the stretch factor.

The default stretch factor of newly created rows/columns is 1.

See also

setColumnStretchFactors, setRowStretchFactor

7.59.3.16 void QCPLayoutGrid::setColumnStretchFactors (const QList< double > & *factors*)

Sets the stretch *factors* of all columns. *factors* must have the size columnCount.

Stretch factors control the relative sizes of rows and columns. Cells will not be resized beyond their minimum and maximum widths/heights (QCPLayoutElement::setMinimumSize, QCPLayoutElement::setMaximumSize), regardless of the stretch factor.

The default stretch factor of newly created rows/columns is 1.

See also

setColumnStretchFactor, setRowStretchFactors

7.59.3.17 void QCPLayoutGrid::setRowSpacing (int *pixels*)

Sets the gap that is left blank between rows to *pixels*.

See also

setColumnSpacing

7.59.3.18 void QCPLayoutGrid::setRowStretchFactor (int *row*, double *factor*)

Sets the stretch *factor* of *row*.

Stretch factors control the relative sizes of rows and columns. Cells will not be resized beyond their minimum and maximum widths/heights (QCPLayoutElement::setMinimumSize, QCPLayoutElement::setMaximumSize), regardless of the stretch factor.

The default stretch factor of newly created rows/columns is 1.

See also

setColumnStretchFactors, setRowStretchFactor

7.59.3.19 void QCPLayoutGrid::setRowStretchFactors (const QList< double > & *factors*)

Sets the stretch *factors* of all rows. *factors* must have the size rowCount.

Stretch factors control the relative sizes of rows and columns. Cells will not be resized beyond their minimum and maximum widths/heights (QCPLayoutElement::setMinimumSize, QCPLayoutElement::setMaximumSize), regardless of the stretch factor.

The default stretch factor of newly created rows/columns is 1.

See also

setRowStretchFactor, setColumnStretchFactors

7.59.3.20 void QCPLayoutGrid::simplify ( ) [virtual]

Simplifies the layout by collapsing rows and columns which only contain empty cells.

Reimplemented from QCPLayout.

7.59.3.21 bool QCPLayoutGrid::take (QCPLayoutElement ∗ *element*) [virtual]

Removes the specified *element* from the layout and returns true on success.

If the *element* isn't in this layout, returns false.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

takeAt

Implements QCPLayout.

7.59.3.22 QCPLayoutElement ∗ QCPLayoutGrid::takeAt (int *index*) [virtual]

Removes the element with the given *index* from the layout and returns it.

If the *index* is invalid or the cell with that index is empty, returns 0.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

elementAt, take

Implements QCPLayout.

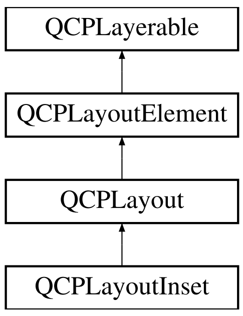
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.60 QCPLayoutInset Class Reference

A layout that places child elements aligned to the border or arbitrarily positioned.

Inheritance diagram for QCPLayoutInset:



Public Types

- enum InsetPlacement { ipFree, ipBorderAligned }

Public Member Functions

- QCPLayoutInset ()
- InsetPlacement insetPlacement (int index) const
- Qt::Alignment insetAlignment (int index) const
- QRectF insetRect (int index) const
- void setInsetPlacement (int index, InsetPlacement placement)
- void setInsetAlignment (int index, Qt::Alignment alignment)
- void setInsetRect (int index, const QRectF &rect)
- virtual void **updateLayout** ()
- virtual int elementCount () const
- virtual QCPLayoutElement ∗ elementAt (int index) const
- virtual QCPLayoutElement ∗ takeAt (int index)
- virtual bool take (QCPLayoutElement ∗element)
- virtual void simplify ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- void addElement (QCPLayoutElement ∗element, Qt::Alignment alignment)
- void addElement (QCPLayoutElement ∗element, const QRectF &rect)

Protected Attributes

- QList< QCPLayoutElement ∗ > **mElements**
- QList< InsetPlacement > **mInsetPlacement**
- QList< Qt::Alignment > **mInsetAlignment**
- QList< QRectF > **mInsetRect**

7.60.1 Detailed Description

A layout that places child elements aligned to the border or arbitrarily positioned.

Elements are placed either aligned to the border or at arbitrary position in the area of the layout. Which placement applies is controlled with the InsetPlacement (setInset-Placement).

Elements are added via addElement(QCPLayoutElement ∗element, Qt::Alignment alignment) or addElement(QCPLayoutElement ∗element, const QRectF &rect). If the first method is used, the inset placement will default to ipBorderAligned and the element will be aligned according to the *alignment* parameter. The second method defaults to ipFree and allows placing elements at arbitrary position and size, defined by *rect*.

The alignment or rect can be set via setInsetAlignment or setInsetRect, respectively.

This is the layout that every QCPAxisRect has as QCPAxisRect::insetLayout.

7.60.2 Member Enumeration Documentation

7.60.2.1 enum QCPLayoutInset::InsetPlacement

Defines how the placement and sizing is handled for a certain element in a QCPLayout-Inset.

Enumerator:

ipFree The element may be positioned/sized arbitrarily, see setInsetRect.

ipBorderAligned The element is aligned to one of the layout sides, see setInset-Alignment.

7.60.3 Constructor & Destructor Documentation

7.60.3.1 QCPLayoutInset::QCPLayoutInset ( ) [explicit]

Creates an instance of QCPLayoutInset and sets default values.

7.60.4 Member Function Documentation

7.60.4.1 void QCPLayoutInset::addElement (QCPLayoutElement ∗ *element,* Qt::Alignment *alignment*)

Adds the specified *element* to the layout as an inset aligned at the border (setInset-Alignment is initialized with ipBorderAligned). The alignment is set to *alignment*.

alignment is an or combination of the following alignment flags: Qt::AlignLeft, Qt::Align-HCenter, Qt::AlighRight, Qt::AlignTop, Qt::AlignVCenter, Qt::AlignBottom. Any other alignment flags will be ignored.

See also

addElement(QCPLayoutElement ∗element, const QRectF &rect)

7.60.4.2 void QCPLayoutInset::addElement (QCPLayoutElement ∗ *element,* const QRectF & *rect*)

Adds the specified *element* to the layout as an inset with free positioning/sizing (setInsetAlignment is initialized with ipFree). The position and size is set to *rect*.

rect is given in fractions of the whole inset layout rect. So an inset with rect (0, 0, 1, 1) will span the entire layout. An inset with rect (0.6, 0.1, 0.35, 0.35) will be in the top right corner of the layout, with 35% width and height of the parent layout.

See also

addElement(QCPLayoutElement ∗element, Qt::Alignment alignment)

7.60.4.3 QCPLayoutElement ∗ QCPLayoutInset::elementAt (int *index*) const [virtual]

Returns the element in the cell with the given *index*. If *index* is invalid, returns 0.

Note that even if *index* is valid, the respective cell may be empty in some layouts (e.g. QCPLayoutGrid), so this function may return 0 in those cases. You may use this function to check whether a cell is empty or not.

See also

elements, elementCount, takeAt

Implements QCPLayout.

7.60.4.4 int QCPLayoutInset::elementCount ( ) const [virtual]

Returns the number of elements/cells in the layout.

See also

elements, elementAt

Implements QCPLayout.

7.60.4.5 Qt::Alignment QCPLayoutInset::insetAlignment (int *index*) const

Returns the alignment of the element with the specified *index*. The alignment only has a meaning, if the inset placement (setInsetPlacement) is ipBorderAligned.

7.60.4.6 QCPLayoutInset::InsetPlacement QCPLayoutInset::insetPlacement (int *index*) const

Returns the placement type of the element with the specified *index*.

7.60.4.7 QRectF QCPLayoutInset::insetRect (int *index*) const

Returns the rect of the element with the specified *index*. The rect only has a meaning, if the inset placement (setInsetPlacement) is ipFree.

7.60.4.8 double QCPLayoutInset::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = 0) const [virtual]

The inset layout is sensitive to events only at areas where its (visible) child elements are sensitive. If the selectTest method of any of the child elements returns a positive number for *pos*, this method returns a value corresponding to 0.99 times the parent plot's selection tolerance. The inset layout is not selectable itself by default. So if *onlySelectable* is true, -1.0 is returned.

See QCPLayerable::selectTest for a general explanation of this virtual method.

Reimplemented from QCPLayoutElement.

7.60.4.9 void QCPLayoutInset::setInsetAlignment (int *index,* Qt::Alignment *alignment*)

If the inset placement (setInsetPlacement) is ipBorderAligned, this function is used to set the alignment of the element with the specified *index* to *alignment*.

alignment is an or combination of the following alignment flags: Qt::AlignLeft, Qt::AlignHCenter, Qt::AlighRight, Qt::AlignTop, Qt::AlignVCenter, Qt::AlignBottom. Any other alignment flags will be ignored.

7.60.4.10 void QCPLayoutInset::setInsetPlacement (int *index,* QCPLayoutInset::InsetPlacement *placement*)

Sets the inset placement type of the element with the specified *index* to *placement*.

See also

InsetPlacement

7.60.4.11 void QCPLayoutInset::setInsetRect (int *index,* const QRectF & *rect*)

If the inset placement (setInsetPlacement) is ipFree, this function is used to set the position and size of the element with the specified *index* to *rect*.

rect is given in fractions of the whole inset layout rect. So an inset with rect (0, 0, 1, 1) will span the entire layout. An inset with rect (0.6, 0.1, 0.35, 0.35) will be in the top right corner of the layout, with 35% width and height of the parent layout.

Note that the minimum and maximum sizes of the embedded element (QCPLayoutElement::setMinimumSize, QCPLayoutElement::setMaximumSize) are enforced.

7.60.4.12 void **QCPLayoutInset::simplify** ( ) `[inline, virtual]`

The QCPInsetLayout does not need simplification since it can never have empty cells due to its linear index structure. This method does nothing.

Reimplemented from QCPLayout.

7.60.4.13 bool **QCPLayoutInset::take** (**QCPLayoutElement** ∗ *element*) `[virtual]`

Removes the specified *element* from the layout and returns true on success.

If the *element* isn't in this layout, returns false.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

takeAt

Implements QCPLayout.

7.60.4.14 **QCPLayoutElement** ∗ **QCPLayoutInset::takeAt** (int *index*) `[virtual]`

Removes the element with the given *index* from the layout and returns it.

If the *index* is invalid or the cell with that index is empty, returns 0.

Note that some layouts don't remove the respective cell right away but leave an empty cell after successful removal of the layout element. To collapse empty cells, use simplify.

See also

elementAt, take

Implements QCPLayout.

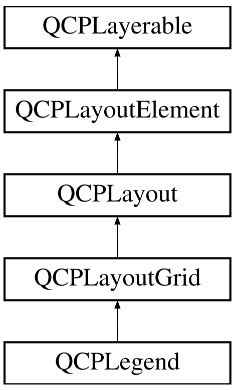
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.61 QCPLegend Class Reference

Manages a legend inside a QCustomPlot.

Inheritance diagram for QCPLegend:



Public Types

- enum SelectablePart { spNone = 0x000, spLegendBox = 0x001, spItems = 0x002 }

Signals

- void selectionChanged (QCPLegend::SelectableParts parts)
- void **selectableChanged** (QCPLegend::SelectableParts parts)

Public Member Functions

- QCPLegend ()
- QPen **borderPen** () const
- QBrush **brush** () const
- QFont **font** () const
- QColor **textColor** () const
- QSize **iconSize** () const
- int **iconTextPadding** () const
- QPen **iconBorderPen** () const
- SelectableParts **selectableParts** () const
- SelectableParts **selectedParts** () const
- QPen **selectedBorderPen** () const
- QPen **selectedIconBorderPen** () const
- QBrush **selectedBrush** () const
- QFont **selectedFont** () const

- QColor **selectedTextColor** () const
- void setBorderPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setFont (const QFont &font)
- void setTextColor (const QColor &color)
- void setIconSize (const QSize &size)
- void setIconSize (int width, int height)
- void setIconTextPadding (int padding)
- void setIconBorderPen (const QPen &pen)
- Q_SLOT void setSelectableParts (const SelectableParts &selectableParts)
- Q_SLOT void setSelectedParts (const SelectableParts &selectedParts)
- void setSelectedBorderPen (const QPen &pen)
- void setSelectedIconBorderPen (const QPen &pen)
- void setSelectedBrush (const QBrush &brush)
- void setSelectedFont (const QFont &font)
- void setSelectedTextColor (const QColor &color)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const
- QCPAbstractLegendItem ∗ item (int index) const
- QCPPlottableLegendItem ∗ itemWithPlottable (const QCPAbstractPlottable ∗plottable) const
- int itemCount () const
- bool hasItem (QCPAbstractLegendItem ∗item) const
- bool hasItemWithPlottable (const QCPAbstractPlottable ∗plottable) const
- bool addItem (QCPAbstractLegendItem ∗item)
- bool removeItem (int index)
- bool removeItem (QCPAbstractLegendItem ∗item)
- void clearItems ()
- QList< QCPAbstractLegendItem ∗ > selectedItems () const

Protected Member Functions

- virtual void **parentPlotInitialized** (QCustomPlot ∗parentPlot)
- virtual QCP::Interaction **selectionCategory** () const
- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- QPen **getBorderPen** () const
- QBrush **getBrush** () const

Protected Attributes

- QPen **mBorderPen**
- QPen **mIconBorderPen**
- QBrush **mBrush**
- QFont **mFont**
- QColor **mTextColor**
- QSize **mIconSize**
- int **mIconTextPadding**
- SelectableParts **mSelectedParts**
- SelectableParts **mSelectableParts**
- QPen **mSelectedBorderPen**
- QPen **mSelectedIconBorderPen**
- QBrush **mSelectedBrush**
- QFont **mSelectedFont**
- QColor **mSelectedTextColor**

Friends

- class **QCustomPlot**
- class **QCPAbstractLegendItem**

7.61.1 Detailed Description

Manages a legend inside a QCustomPlot.

A legend is a small box somewhere in the plot which lists plottables with their name and icon.

Normally, the legend is populated by calling QCPAbstractPlottable::addToLegend. - The respective legend item can be removed with QCPAbstractPlottable::removeFromLegend. However, QCPLegend also offers an interface to add and manipulate legend items directly: item, itemWithPlottable, itemCount, addItem, removeItem, etc.

The QCPLegend derives from QCPLayoutGrid and as such can be placed in any position a QCPLayoutElement may be positioned. The legend items are themselves QCPLayoutElements which are placed in the grid layout of the legend. QCPLegend only adds an interface specialized for handling child elements of type QCPAbstractLegendItem, as mentioned above. In principle, any other layout elements may also be added to a legend via the normal QCPLayoutGrid interface. However, the QCPAbstractLegendItem-Interface will ignore those elements (e.g. itemCount will only return the number of items with QCPAbstractLegendItems type).

By default, every QCustomPlot has one legend (QCustomPlot::legend) which is placed in the inset layout of the main axis rect (QCPAxisRect::insetLayout). To move the legend to another position inside the axis rect, use the methods of the QCPLayoutInset. To move the legend outside of the axis rect, place it anywhere else with the QCPLayout/-QCPLayoutElement interface.

7.61.2 Member Enumeration Documentation

7.61.2.1 enum QCPLegend::SelectablePart

Defines the selectable parts of a legend

See also

setSelectedParts, setSelectableParts

Enumerator:

spNone `0x000` None

spLegendBox `0x001` The legend box (frame)

spItems `0x002` Legend items individually (see selectedItems)

7.61.3 Constructor & Destructor Documentation

7.61.3.1 QCPLegend::QCPLegend ( ) `[explicit]`

Constructs a new QCPLegend instance with *parentPlot* as the containing plot and default values.

Note that by default, QCustomPlot already contains a legend ready to be used as Q-CustomPlot::legend

7.61.4 Member Function Documentation

7.61.4.1 bool QCPLegend::addItem (QCPAbstractLegendItem ∗ *item*)

Adds *item* to the legend, if it's not present already.

Returns true on sucess, i.e. if the item wasn't in the list already and has been successfuly added.

The legend takes ownership of the item.

7.61.4.2 void QCPLegend::clearItems ( )

Removes all items from the legend.

7.61.4.3 bool QCPLegend::hasItem (QCPAbstractLegendItem ∗ *item*) const

Returns whether the legend contains *itm*.

7.61.4.4 bool QCPLegend::hasItemWithPlottable (const QCPAbstractPlottable ∗ *plottable*) const

Returns whether the legend contains a QCPPlottableLegendItem which is associated with *plottable* (e.g. a QCPGraph∗). If such an item isn't in the legend, returns false.

See also

itemWithPlottable

7.61.4.5 QCPAbstractLegendItem ∗ QCPLegend::item (int *index*) const

Returns the item with index *i*.

See also

itemCount

7.61.4.6 int QCPLegend::itemCount ( ) const

Returns the number of items currently in the legend.

See also

item

7.61.4.7 QCPPlottableLegendItem ∗ QCPLegend::itemWithPlottable (const QCPAbstractPlottable ∗ *plottable*) const

Returns the QCPPlottableLegendItem which is associated with *plottable* (e.g. a QCP-Graph∗). If such an item isn't in the legend, returns 0.

See also

hasItemWithPlottable

7.61.4.8 bool QCPLegend::removeItem (int *index*)

Removes the item with index *index* from the legend.

Returns true, if successful.

See also

itemCount, clearItems

7.61.4.9 bool QCPLegend::removeItem (QCPAbstractLegendItem $*$ *item*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes *item* from the legend.

Returns true, if successful.

See also

clearItems

7.61.4.10 QList$<$ QCPAbstractLegendItem $*$ $>$ QCPLegend::selectedItems ( ) const

Returns the legend items that are currently selected. If no items are selected, the list is empty.

See also

QCPAbstractLegendItem::setSelected, setSelectable

7.61.4.11 void QCPLegend::selectionChanged (QCPLegend::SelectableParts *selection*) `[signal]`

This signal is emitted when the selection state of this legend has changed.

See also

setSelectedParts, setSelectableParts

7.61.4.12 double QCPLegend::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant $*$ *details* = `0`) const `[virtual]`

Layout elements are sensitive to events inside their outer rect. If *pos* is within the outer rect, this method returns a value corresponding to 0.99 times the parent plot's selection tolerance. However, layout elements are not selectable by default. So if *onlySelectable* is true, -1.0 is returned.

See QCPLayerable::selectTest for a general explanation of this virtual method.

QCPLayoutElement subclasses may reimplement this method to provide more specific selection test behaviour.

Reimplemented from QCPLayoutElement.

7.61.4.13 void QCPLegend::setBorderPen (const QPen & *pen*)

Sets the pen, the border of the entire legend is drawn with.

7.61.4.14 void **QCPLegend::setBrush** (const QBrush & *brush*)

Sets the brush of the legend background.

7.61.4.15 void **QCPLegend::setFont** (const QFont & *font*)

Sets the default font of legend text. Legend items that draw text (e.g. the name of a graph) will use this font by default. However, a different font can be specified on a per-item-basis by accessing the specific legend item.

This function will also set *font* on all already existing legend items.

See also

QCPAbstractLegendItem::setFont

7.61.4.16 void **QCPLegend::setIconBorderPen** (const QPen & *pen*)

Sets the pen used to draw a border around each legend icon. Legend items that draw an icon (e.g. a visual representation of the graph) will use this pen by default.

If no border is wanted, set this to *Qt::NoPen*.

7.61.4.17 void **QCPLegend::setIconSize** (const QSize & *size*)

Sets the size of legend icons. Legend items that draw an icon (e.g. a visual representation of the graph) will use this size by default.

7.61.4.18 void **QCPLegend::setIconSize** (int *width,* int *height*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

7.61.4.19 void **QCPLegend::setIconTextPadding** (int *padding*)

Sets the horizontal space in pixels between the legend icon and the text next to it. - Legend items that draw an icon (e.g. a visual representation of the graph) and text (e.g. the name of the graph) will use this space by default.

7.61.4.20 void **QCPLegend::setSelectableParts** (const SelectableParts & *selectable*)

Sets whether the user can (de-)select the parts in *selectable* by clicking on the - QCustomPlot surface. (When QCustomPlot::setInteractions contains QCP::iSelect-Legend.)

However, even when *selectable* is set to a value not allowing the selection of a specific part, it is still possible to set the selection of this part manually, by calling setSelected-Parts directly.

See also

SelectablePart, setSelectedParts

7.61.4.21 void QCPLegend::setSelectedBorderPen (const QPen & *pen*)

When the legend box is selected, this pen is used to draw the border instead of the normal pen set via setBorderPen.

See also

setSelectedParts, setSelectableParts, setSelectedBrush

7.61.4.22 void QCPLegend::setSelectedBrush (const QBrush & *brush*)

When the legend box is selected, this brush is used to draw the legend background instead of the normal brush set via setBrush.

See also

setSelectedParts, setSelectableParts, setSelectedBorderPen

7.61.4.23 void QCPLegend::setSelectedFont (const QFont & *font*)

Sets the default font that is used by legend items when they are selected.

This function will also set *font* on all already existing legend items.

See also

setFont, QCPAbstractLegendItem::setSelectedFont

7.61.4.24 void QCPLegend::setSelectedIconBorderPen (const QPen & *pen*)

Sets the pen legend items will use to draw their icon borders, when they are selected.

See also

setSelectedParts, setSelectableParts, setSelectedFont

7.61.4.25 void QCPLegend::setSelectedParts (const SelectableParts & *selected*)

Sets the selected state of the respective legend parts described by SelectablePart. - When a part is selected, it uses a different pen/font and brush. If some legend items are selected and *selected* doesn't contain spItems, those items become deselected.

The entire selection mechanism is handled automatically when QCustomPlot::set-Interactions contains iSelectLegend. You only need to call this function when you wish to change the selection state manually.

This function can change the selection state of a part even when setSelectableParts was set to a value that actually excludes the part.

emits the selectionChanged signal when *selected* is different from the previous selection state.

Note that it doesn't make sense to set the selected state spItems here when it wasn't set before, because there's no way to specify which exact items to newly select. Do this by calling QCPAbstractLegendItem::setSelected directly on the legend item you wish to select.

See also

SelectablePart, setSelectableParts, selectTest, setSelectedBorderPen, set-SelectedIconBorderPen, setSelectedBrush, setSelectedFont

7.61.4.26 void QCPLegend::setSelectedTextColor (const QColor & *color*)

Sets the default text color that is used by legend items when they are selected.

This function will also set *color* on all already existing legend items.

See also

setTextColor, QCPAbstractLegendItem::setSelectedTextColor

7.61.4.27 void QCPLegend::setTextColor (const QColor & *color*)

Sets the default color of legend text. Legend items that draw text (e.g. the name of a graph) will use this color by default. However, a different colors can be specified on a per-item-basis by accessing the specific legend item.

This function will also set *color* on all already existing legend items.

See also

QCPAbstractLegendItem::setTextColor

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.62 QCPLineEnding Class Reference

Handles the different ending decorations for line-like items.

Public Types

- enum EndingStyle { esNone, esFlatArrow, esSpikeArrow, esLineArrow, esDisc, esSquare, esDiamond, esBar, esHalfBar, esSkewedBar }

Public Member Functions

- QCPLineEnding ()
- QCPLineEnding (EndingStyle style, double width=8, double length=10, bool inverted=false)
- EndingStyle **style** () const
- double **width** () const
- double **length** () const
- bool **inverted** () const
- void setStyle (EndingStyle style)
- void setWidth (double width)
- void setLength (double length)
- void setInverted (bool inverted)
- double **boundingDistance** () const
- double realLength () const
- void **draw** (QCPPainter ∗painter, const QVector2D &pos, const QVector2D &dir) const
- void **draw** (QCPPainter ∗painter, const QVector2D &pos, double angle) const

Protected Attributes

- EndingStyle **mStyle**
- double **mWidth**
- double **mLength**
- bool **mInverted**

7.62.1 Detailed Description

Handles the different ending decorations for line-like items.

For every ending a line-like item has, an instance of this class exists. For example, QCPItemLine has two endings which can be set with QCPItemLine::setHead and QCPItemLine::setTail.

The styles themselves are defined via the enum QCPLineEnding::EndingStyle. Most decorations can be modified regarding width and length, see setWidth and setLength. The direction of the ending decoration (e.g. direction an arrow is pointing) is controlled by the line-like item. For example, when both endings of a QCPItemLine are set to be arrows, they will point to opposite directions, e.g. "outward". This can be changed by setInverted, which would make the respective arrow point inward.

Note that due to the overloaded QCPLineEnding constructor, you may directly specify a QCPLineEnding::EndingStyle where actually a QCPLineEnding is expected, e.g.

7.62.2 Member Enumeration Documentation

7.62.2.1 enum QCPLineEnding::EndingStyle

Defines the type of ending decoration for line-like items, e.g. an arrow.

The width and length of these decorations can be controlled with the functions setWidth and setLength. Some decorations like esDisc, esSquare, esDiamond and esBar only support a width, the length property is ignored.

See also

QCPItemLine::setHead, QCPItemLine::setTail, QCPItemCurve::setHead, QCPItemCurve::setTail, QCPAxis::setLowerEnding, QCPAxis::setUpperEnding

Enumerator:

esNone No ending decoration.

esFlatArrow A filled arrow head with a straight/flat back (a triangle)

esSpikeArrow A filled arrow head with an indented back.

esLineArrow A non-filled arrow head with open back.

esDisc A filled circle.

esSquare A filled square.

esDiamond A filled diamond (45° rotated square)

esBar A bar perpendicular to the line.

esHalfBar A bar perpendicular to the line, pointing out to only one side (to which side can be changed with setInverted)

esSkewedBar A bar that is skewed (skew controllable via setLength)

7.62.3 Constructor & Destructor Documentation

7.62.3.1 QCPLineEnding::QCPLineEnding ( )

Creates a QCPLineEnding instance with default values (style esNone).

7.62.3.2 QCPLineEnding::QCPLineEnding (QCPLineEnding::EndingStyle *style*, double *width* = `8`, double *length* = `10`, bool *inverted* = `false`)

Creates a QCPLineEnding instance with the specified values.

7.62.4 Member Function Documentation

7.62.4.1 double **QCPLineEnding::realLength** ( ) const

Starting from the origin of this line ending (which is style specific), returns the length covered by the line ending symbol, in backward direction.

For example, the esSpikeArrow has a shorter real length than a esFlatArrow, even if both have the same setLength value, because the spike arrow has an inward curved back, which reduces the length along its center axis (the drawing origin for arrows is at the tip).

This function is used for precise, style specific placement of line endings, for example in QCPAxes.

7.62.4.2 void **QCPLineEnding::setInverted** (bool *inverted*)

Sets whether the ending decoration shall be inverted. For example, an arrow decoration will point inward when *inverted* is set to true.

Note that also the *width* direction is inverted. For symmetrical ending styles like arrows or discs, this doesn't make a difference. However, asymmetric styles like esHalfBar are affected by it, which can be used to control to which side the half bar points to.

7.62.4.3 void **QCPLineEnding::setLength** (double *length*)

Sets the length of the ending decoration, if the style supports it. On arrows, for example, the length defines the size in pointing direction.

See also

setWidth

7.62.4.4 void **QCPLineEnding::setStyle** (**QCPLineEnding::EndingStyle** *style*)

Sets the style of the ending decoration.

7.62.4.5 void **QCPLineEnding::setWidth** (double *width*)

Sets the width of the ending decoration, if the style supports it. On arrows, for example, the width defines the size perpendicular to the arrow's pointing direction.

See also

setLength

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.63 QCPMarginGroup Class Reference

A margin group allows synchronization of margin sides if working with multiple layout elements.

Public Member Functions

- QCPMarginGroup (QCustomPlot ∗parentPlot)
- QList< QCPLayoutElement ∗ > elements (QCP::MarginSide side) const
- bool isEmpty () const
- void clear ()

Protected Member Functions

- int **commonMargin** (QCP::MarginSide side) const
- void **addChild** (QCP::MarginSide side, QCPLayoutElement ∗element)
- void **removeChild** (QCP::MarginSide side, QCPLayoutElement ∗element)

Protected Attributes

- QCustomPlot ∗ **mParentPlot**
- QHash< QCP::MarginSide, QList < QCPLayoutElement ∗ > > **mChildren**

Friends

- class **QCPLayoutElement**

7.63.1 Detailed Description

A margin group allows synchronization of margin sides if working with multiple layout elements.

QCPMarginGroup allows you to tie a margin side of two or more layout elements together, such that they will all have the same size, based on the largest required margin in the group.

In certain situations it is desirable that margins at specific sides are synchronized across layout elements. For example, if one QCPAxisRect is below another one in a grid layout, it will provide a cleaner look to the user if the left and right margins of the two axis rects are of the same size. The left axis of the top axis rect will then be at the same horizontal position as the left axis of the lower axis rect, making them appear aligned. The same applies for the right axes. This is what QCPMarginGroup makes possible.

To add/remove a specific side of a layout element to/from a margin group, use the QCPLayoutElement::setMarginGroup method. To completely break apart the margin group, either call clear, or just delete the margin group.

7.63.2 Example

First create a margin group:

Then set this group on the layout element sides:

Here, we've used the first two axis rects of the plot and synchronized their left margins with each other and their right margins with each other.

7.63.3 Constructor & Destructor Documentation

7.63.3.1 QCPMarginGroup::QCPMarginGroup (QCustomPlot ∗ *parentPlot*)

Creates a new QCPMarginGroup instance in *parentPlot*.

7.63.4 Member Function Documentation

7.63.4.1 void QCPMarginGroup::clear ( )

Clears this margin group. The synchronization of the margin sides that use this margin group is lifted and they will use their individual margin sizes again.

7.63.4.2 QList< QCPLayoutElement ∗ > QCPMarginGroup::elements (QCP::MarginSide *side*) const [inline]

Returns a list of all layout elements that have their margin *side* associated with this margin group.

7.63.4.3 bool QCPMarginGroup::isEmpty ( ) const

Returns whether this margin group is empty. If this function returns true, no layout elements use this margin group to synchronize margin sides.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.64 QCPPainter Class Reference

QPainter subclass used internally.

Public Types

- enum PainterMode { pmDefault = 0x00, pmVectorized = 0x01, pmNoCaching = 0x02, pmNonCosmetic = 0x04 }

Public Member Functions

- QCPPainter ()
- QCPPainter (QPaintDevice ∗device)
- bool **antialiasing** () const
- PainterModes **modes** () const
- void setAntialiasing (bool enabled)
- void setMode (PainterMode mode, bool enabled=true)
- void setModes (PainterModes modes)
- bool begin (QPaintDevice ∗device)
- void setPen (const QPen &pen)
- void setPen (const QColor &color)
- void setPen (Qt::PenStyle penStyle)
- void drawLine (const QLineF &line)
- void **drawLine** (const QPointF &p1, const QPointF &p2)
- void save ()
- void restore ()
- void makeNonCosmetic ()

Protected Attributes

- PainterModes **mModes**
- bool **mIsAntialiasing**
- QStack< bool > **mAntialiasingStack**

7.64.1 Detailed Description

QPainter subclass used internally.

This QPainter subclass is used to provide some extended functionality e.g. for tweaking position consistency between antialiased and non-antialiased painting. Further it provides workarounds for QPainter quirks.

Warning

This class intentionally hides non-virtual functions of QPainter, e.g. setPen, save and restore. So while it is possible to pass a QCPPainter instance to a function that expects a QPainter pointer, some of the workarounds and tweaks will be unavailable to the function (because it will call the base class implementations of the functions actually hidden by QCPPainter).

7.64.2 Member Enumeration Documentation

7.64.2.1 enum QCPPainter::PainterMode

Defines special modes the painter can operate in. They disable or enable certain subsets of features/fixes/workarounds, depending on whether they are wanted on the respective output device.

Enumerator:

pmDefault `0x00` Default mode for painting on screen devices

pmVectorized `0x01` Mode for vectorized painting (e.g. PDF export). For example, this prevents some antialiasing fixes.

pmNoCaching `0x02` Mode for all sorts of exports (e.g. PNG, PDF,...). For example, this prevents using cached pixmap labels

pmNonCosmetic `0x04` Turns pen widths 0 to 1, i.e. disables cosmetic pens. (A cosmetic pen is always drawn with width 1 pixel in the vector image/pdf viewer, independent of zoom.)

7.64.3 Constructor & Destructor Documentation

7.64.3.1 QCPPainter::QCPPainter ( )

Creates a new QCPPainter instance and sets default values

7.64.3.2 QCPPainter::QCPPainter (QPaintDevice * *device*)

Creates a new QCPPainter instance on the specified paint *device* and sets default values. Just like the analogous QPainter constructor, begins painting on *device* immediately.

Like begin, this method sets QPainter::NonCosmeticDefaultPen in Qt versions before Qt5.

7.64.4 Member Function Documentation

7.64.4.1 bool QCPPainter::begin (QPaintDevice * *device*)

Sets the QPainter::NonCosmeticDefaultPen in Qt versions before Qt5 after beginning painting on *device*. This is necessary to get cosmetic pen consistency across Qt versions, because since Qt5, all pens are non-cosmetic by default, and in Qt4 this render hint must be set to get that behaviour.

The Constructor QCPPainter(QPaintDevice *device) which directly starts painting also sets the render hint as appropriate.

Note

this function hides the non-virtual base class implementation.

7.64.4.2 void QCPPainter::drawLine (const QLineF & *line*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Works around a Qt bug introduced with Qt 4.8 which makes drawing QLineF unpredictable when antialiasing is disabled. Thus when antialiasing is disabled, it rounds the *line* to integer coordinates and then passes it to the original drawLine.

Note

this function hides the non-virtual base class implementation.

7.64.4.3 void QCPPainter::makeNonCosmetic ( )

Changes the pen width to 1 if it currently is 0. This function is called in the setPen overrides when the pmNonCosmetic mode is set.

7.64.4.4 void QCPPainter::restore ( )

Restores the painter (see QPainter::restore). Since QCPPainter adds some new internal state to QPainter, the save/restore functions are reimplemented to also save/restore those members.

Note

this function hides the non-virtual base class implementation.

See also

save

7.64.4.5 void **QCPPainter::save** ( )

Saves the painter (see QPainter::save). Since QCPPainter adds some new internal state to QPainter, the save/restore functions are reimplemented to also save/restore those members.

Note

this function hides the non-virtual base class implementation.

See also

restore

7.64.4.6 void **QCPPainter::setAntialiasing** (bool *enabled*)

Sets whether painting uses antialiasing or not. Use this method instead of using setRenderHint with QPainter::Antialiasing directly, as it allows QCPPainter to regain pixel exactness between antialiased and non-antialiased painting (Since Qt $<$ 5.0 uses slightly different coordinate systems for AA/Non-AA painting).

7.64.4.7 void **QCPPainter::setMode** (**QCPPainter::PainterMode** *mode,* bool *enabled* = `true`)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the mode of the painter. - This controls whether the painter shall adjust its fixes/workarounds optimized for certain output devices.

7.64.4.8 void **QCPPainter::setModes** (PainterModes *modes*)

Sets the mode of the painter. This controls whether the painter shall adjust its fixes/- workarounds optimized for certain output devices.

7.64.4.9 void **QCPPainter::setPen** (const QPen & *pen*)

Sets the pen of the painter and applies certain fixes to it, depending on the mode of this QCPPainter.

Note

this function hides the non-virtual base class implementation.

7.64.4.10 void QCPPainter::setPen (const QColor & *color*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the pen (by color) of the painter and applies certain fixes to it, depending on the mode of this QCPPainter.

Note

this function hides the non-virtual base class implementation.

7.64.4.11 void QCPPainter::setPen (Qt::PenStyle *penStyle*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the pen (by style) of the painter and applies certain fixes to it, depending on the mode of this QCPPainter.

Note

this function hides the non-virtual base class implementation.

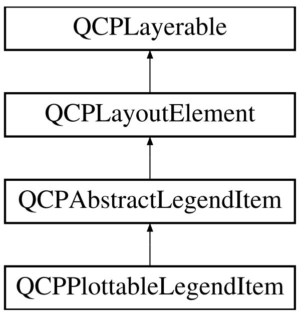
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.65 QCPPlottableLegendItem Class Reference

A legend item representing a plottable with an icon and the plottable name.

Inheritance diagram for QCPPlottableLegendItem:



Public Member Functions

- QCPPlottableLegendItem (QCPLegend ∗parent, QCPAbstractPlottable ∗plottable)
- QCPAbstractPlottable ∗ **plottable** ()

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual QSize minimumSizeHint () const
- QPen **getIconBorderPen** () const
- QColor **getTextColor** () const
- QFont **getFont** () const

Protected Attributes

- QCPAbstractPlottable ∗ **mPlottable**

7.65.1 Detailed Description

A legend item representing a plottable with an icon and the plottable name.

This is the standard legend item for plottables. It displays an icon of the plottable next to the plottable name. The icon is drawn by the respective plottable itself (QCPAbstract-Plottable::drawLegendIcon), and tries to give an intuitive symbol for the plottable. For example, the QCPGraph draws a centered horizontal line and/or a single scatter point in the middle.

Legend items of this type are always associated with one plottable (retrievable via the plottable() function and settable with the constructor). You may change the font of the plottable name with setFont. Icon padding and border pen is taken from the parent QC-PLegend, see QCPLegend::setIconBorderPen and QCPLegend::setIconTextPadding.

The function QCPAbstractPlottable::addToLegend/QCPAbstractPlottable::remove-FromLegend creates/removes legend items of this type in the default implementation. However, these functions may be reimplemented such that a different kind of legend item (e.g a direct subclass of QCPAbstractLegendItem) is used for that plottable.

Since QCPLegend is based on QCPLayoutGrid, a legend item itself is just a subclass of QCPLayoutElement. While it could be added to a legend (or any other layout) via the normal layout interface, QCPLegend has specialized functions for handling legend items conveniently, see the documentation of QCPLegend.

7.65.2 Constructor & Destructor Documentation

7.65.2.1 QCPPlottableLegendItem::QCPPlottableLegendItem (QCPLegend ∗ *parent,* QCPAbstractPlottable ∗ *plottable*)

Creates a new legend item associated with *plottable*.

Once it's created, it can be added to the legend via QCPLegend::addItem.

A more convenient way of adding/removing a plottable to/from the legend is via the functions QCPAbstractPlottable::addToLegend and QCPAbstractPlottable::remove-FromLegend.

7.65.3 Member Function Documentation

7.65.3.1 QSize QCPPlottableLegendItem::minimumSizeHint ( ) const [protected, virtual]

Returns the minimum size this layout element (the inner rect) may be compressed to.

if a minimum size (setMinimumSize) was not set manually, parent layouts consult this function to determine the minimum allowed size of this layout element. (A manual minimum size is considered set if it is non-zero.)

Reimplemented from QCPLayoutElement.

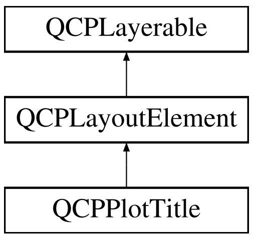
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.66 QCPPlotTitle Class Reference

A layout element displaying a plot title text.

Inheritance diagram for QCPPlotTitle:



Signals

- void selectionChanged (bool selected)
- void **selectableChanged** (bool selectable)

Public Member Functions

- QCPPlotTitle (QCustomPlot ∗parentPlot)
- QCPPlotTitle (QCustomPlot ∗parentPlot, const QString &text)
- QString **text** () const
- QFont **font** () const
- QColor **textColor** () const
- QFont **selectedFont** () const
- QColor **selectedTextColor** () const

- bool **selectable** () const
- bool **selected** () const
- void setText (const QString &text)
- void setFont (const QFont &font)
- void setTextColor (const QColor &color)
- void setSelectedFont (const QFont &font)
- void setSelectedTextColor (const QColor &color)
- Q_SLOT void setSelectable (bool selectable)
- Q_SLOT void setSelected (bool selected)
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual void **applyDefaultAntialiasingHint** (QCPPainter ∗painter) const
- virtual void **draw** (QCPPainter ∗painter)
- virtual QSize minimumSizeHint () const
- virtual QSize maximumSizeHint () const
- virtual void **selectEvent** (QMouseEvent ∗event, bool additive, const QVariant &details, bool ∗selectionStateChanged)
- virtual void **deselectEvent** (bool ∗selectionStateChanged)
- QFont **mainFont** () const
- QColor **mainTextColor** () const

Protected Attributes

- QString **mText**
- QFont **mFont**
- QColor **mTextColor**
- QFont **mSelectedFont**
- QColor **mSelectedTextColor**
- QRect **mTextBoundingRect**
- bool **mSelectable**
- bool **mSelected**

7.66.1 Detailed Description

A layout element displaying a plot title text.

The text may be specified with setText, theformatting can be controlled with setFont and setTextColor.

A plot title can be added as follows:

Since a plot title is a common requirement, QCustomPlot offers specialized selection signals for easy interaction with QCPPlotTitle. If a layout element of type QCPPlot-Title is clicked, the signal QCustomPlot::titleClick is emitted. A double click emits the QCustomPlot::titleDoubleClick signal.

7.66.2 Constructor & Destructor Documentation

7.66.2.1 QCPPlotTitle::QCPPlotTitle (QCustomPlot ∗ *parentPlot*) [explicit]

Creates a new QCPPlotTitle instance and sets default values. The initial text is empty (setText).

To set the title text in the constructor, rather use QCPPlotTitle(QCustomPlot ∗parentPlot, const QString &text).

7.66.2.2 QCPPlotTitle::QCPPlotTitle (QCustomPlot ∗ *parentPlot,* const QString & *text*) [explicit]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Creates a new QCPPlotTitle instance and sets default values. The initial text is set to *text*.

7.66.3 Member Function Documentation

7.66.3.1 QSize QCPPlotTitle::maximumSizeHint ( ) const [protected, virtual]

Returns the maximum size this layout element (the inner rect) may be expanded to.

if a maximum size (setMaximumSize) was not set manually, parent layouts consult this function to determine the maximum allowed size of this layout element. (A manual maximum size is considered set if it is smaller than Qt's QWIDGETSIZE_MAX.)

Reimplemented from QCPLayoutElement.

7.66.3.2 QSize QCPPlotTitle::minimumSizeHint ( ) const [protected, virtual]

Returns the minimum size this layout element (the inner rect) may be compressed to.

if a minimum size (setMinimumSize) was not set manually, parent layouts consult this function to determine the minimum allowed size of this layout element. (A manual minimum size is considered set if it is non-zero.)

Reimplemented from QCPLayoutElement.

7.66.3.3 void QCPPlotTitle::selectionChanged (bool *selected*) `[signal]`

This signal is emitted when the selection state has changed to *selected*, either by user interaction or by a direct call to setSelected.

See also

setSelected, setSelectable

7.66.3.4 double QCPPlotTitle::selectTest (const QPointF & *pos*, bool *onlySelectable*, QVariant ∗ *details* = `0`) const `[virtual]`

Layout elements are sensitive to events inside their outer rect. If *pos* is within the outer rect, this method returns a value corresponding to 0.99 times the parent plot's selection tolerance. However, layout elements are not selectable by default. So if *onlySelectable* is true, -1.0 is returned.

See QCPLayerable::selectTest for a general explanation of this virtual method.

QCPLayoutElement subclasses may reimplement this method to provide more specific selection test behaviour.

Reimplemented from QCPLayoutElement.

7.66.3.5 void QCPPlotTitle::setFont (const QFont & *font*)

Sets the *font* of the title text.

See also

setTextColor, setSelectedFont

7.66.3.6 void QCPPlotTitle::setSelectable (bool *selectable*)

Sets whether the user may select this plot title to *selectable*.

Note that even when *selectable* is set to `false`, the selection state may be changed programmatically via setSelected.

7.66.3.7 void QCPPlotTitle::setSelected (bool *selected*)

Sets the selection state of this plot title to *selected*. If the selection has changed, selectionChanged is emitted.

Note that this function can change the selection state independently of the current set-Selectable state.

7.66.3.8 void **QCPPlotTitle::setSelectedFont** (const QFont & *font*)

Sets the *font* of the title text that will be used if the plot title is selected (setSelected).

See also

setFont

7.66.3.9 void **QCPPlotTitle::setSelectedTextColor** (const QColor & *color*)

Sets the *color* of the title text that will be used if the plot title is selected (setSelected).

See also

setTextColor

7.66.3.10 void **QCPPlotTitle::setText** (const QString & *text*)

Sets the text that will be displayed to *text*. Multiple lines can be created by insertion of "\n".

See also

setFont, setTextColor

7.66.3.11 void **QCPPlotTitle::setTextColor** (const QColor & *color*)

Sets the *color* of the title text.

See also

setFont, setSelectedTextColor

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.67 QCPRange Class Reference

Represents the range an axis is encompassing.

Public Member Functions

- QCPRange ()
- QCPRange (double lower, double upper)
- bool **operator==** (const QCPRange &other) const
- bool **operator!=** (const QCPRange &other) const
- QCPRange & operator+= (const double &value)
- QCPRange & operator-= (const double &value)
- QCPRange & operator∗= (const double &value)
- QCPRange & operator/= (const double &value)
- double size () const
- double center () const
- void normalize ()
- void expand (const QCPRange &otherRange)
- QCPRange expanded (const QCPRange &otherRange) const
- QCPRange sanitizedForLogScale () const
- QCPRange sanitizedForLinScale () const
- bool contains (double value) const

Static Public Member Functions

- static bool validRange (double lower, double upper)
- static bool validRange (const QCPRange &range)

Public Attributes

- double **lower**
- double **upper**

Static Public Attributes

- static const double minRange = 1e-280
- static const double maxRange = 1e250

Friends

- const QCPRange operator+ (const QCPRange &, double)
- const QCPRange operator+ (double, const QCPRange &)
- const QCPRange operator- (const QCPRange &range, double value)
- const QCPRange operator∗ (const QCPRange &range, double value)
- const QCPRange operator∗ (double value, const QCPRange &range)
- const QCPRange operator/ (const QCPRange &range, double value)

7.67.1 Detailed Description

Represents the range an axis is encompassing.

contains a *lower* and *upper* double value and provides convenience input, output and modification functions.

See also

QCPAxis::setRange

7.67.2 Constructor & Destructor Documentation

7.67.2.1 QCPRange::QCPRange ( )

Constructs a range with *lower* and *upper* set to zero.

7.67.2.2 QCPRange::QCPRange (double *lower,* double *upper*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Constructs a range with the specified *lower* and *upper* values.

7.67.3 Member Function Documentation

7.67.3.1 double QCPRange::center ( ) const

Returns the center of the range, i.e. (*upper*+*lower*)$*$*0*.5

7.67.3.2 bool QCPRange::contains (double *value*) const

Returns true when *value* lies within or exactly on the borders of the range.

7.67.3.3 void QCPRange::expand (const QCPRange & *otherRange*)

Expands this range such that *otherRange* is contained in the new range. It is assumed that both this range and *otherRange* are normalized (see normalize).

If *otherRange* is already inside the current range, this function does nothing.

See also

expanded

7.67.3.4 QCPRange QCPRange::expanded (const QCPRange & *otherRange*) const

Returns an expanded range that contains this and *otherRange*. It is assumed that both this range and *otherRange* are normalized (see normalize).

See also

expand

7.67.3.5 void QCPRange::normalize ( )

Makes sure *lower* is numerically smaller than *upper*. If this is not the case, the values are swapped.

7.67.3.6 QCPRange & QCPRange::operator*= (const double & *value*) [inline]

Multiplies both boundaries of the range by *value*.

7.67.3.7 QCPRange & QCPRange::operator+= (const double & *value*) [inline]

Adds *value* to both boundaries of the range.

7.67.3.8 QCPRange & QCPRange::operator-= (const double & *value*) [inline]

Subtracts *value* from both boundaries of the range.

7.67.3.9 QCPRange & QCPRange::operator/= (const double & *value*) [inline]

Divides both boundaries of the range by *value*.

7.67.3.10 QCPRange QCPRange::sanitizedForLinScale ( ) const

Returns a sanitized version of the range. Sanitized means for linear scales, that *lower* will always be numerically smaller (or equal) to *upper*.

7.67.3.11 QCPRange QCPRange::sanitizedForLogScale ( ) const

Returns a sanitized version of the range. Sanitized means for logarithmic scales, that the range won't span the positive and negative sign domain, i.e. contain zero. Further *lower* will always be numerically smaller (or equal) to *upper*.

If the original range does span positive and negative sign domains or contains zero, the returned range will try to approximate the original range as good as possible. If the positive interval of the original range is wider than the negative interval, the returned range will only contain the positive interval, with lower bound set to *rangeFac* or *rangeFac* $*$*upper*, whichever is closer to zero. Same procedure is used if the negative interval is wider than the positive interval, this time by changing the *upper* bound.

7.67.3.12 double QCPRange::size ( ) const

Returns the size of the range, i.e. *upper-lower*

7.67.3.13 bool QCPRange::validRange (double *lower,* double *upper*) `[static]`

Checks, whether the specified range is within valid bounds, which are defined as QCPRange::maxRange and QCPRange::minRange. A valid range means:

- range bounds within -maxRange and maxRange
- range size above minRange
- range size below maxRange

7.67.3.14 bool QCPRange::validRange (const QCPRange & *range*) `[static]`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Checks, whether the specified range is within valid bounds, which are defined as QCPRange::maxRange and QCPRange::minRange. A valid range means:

- range bounds within -maxRange and maxRange
- range size above minRange
- range size below maxRange

7.67.4 Friends And Related Function Documentation

7.67.4.1 const QCPRange operator$*$ (const QCPRange & *range,* double *value*) `[friend]`

Multiplies both boundaries of the range by *value*.

7.67.4.2 const QCPRange operator$*$ (double *value,* const QCPRange & *range*) `[friend]`

Multiplies both boundaries of the range by *value*.

7.67.4.3 const QCPRange operator+ (const QCPRange & *range,* double *value*) `[friend]`

Adds *value* to both boundaries of the range.

7.67.4.4 const QCPRange operator+ (double *value,* const QCPRange & *range*) `[friend]`

Adds *value* to both boundaries of the range.

7.67.4.5 const QCPRange operator- (const QCPRange & *range,* double *value*) `[friend]`

Subtracts *value* from both boundaries of the range.

7.67.4.6 const QCPRange operator/ (const QCPRange & *range,* double *value*) `[friend]`

Divides both boundaries of the range by *value*.

7.67.5 Member Data Documentation

7.67.5.1 const double QCPRange::maxRange = 1e250 `[static]`

Maximum values (negative and positive) the range will accept in range-changing functions. Larger absolute values would cause errors due to the 11-bit exponent of double precision numbers, corresponding to a maximum magnitude of roughly 1e308. Since the number of planck-volumes in the entire visible universe is only ∼1e183, this should be enough.

See also

validRange, minRange

7.67.5.2 const double QCPRange::minRange = 1e-280 `[static]`

Minimum range size (*upper* - *lower*) the range changing functions will accept. Smaller intervals would cause errors due to the 11-bit exponent of double precision numbers, corresponding to a minimum magnitude of roughly 1e-308.

See also

validRange, maxRange

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.68 QCPScatterStyle Class Reference

Represents the visual appearance of scatter points.

Public Types

- enum ScatterShape { ssNone, ssDot, ssCross, ssPlus, ssCircle, ssDisc, ssSquare, ssDiamond, ssStar, ssTriangle, ssTriangleInverted, ssCrossSquare, ssPlusSquare, ssCrossCircle, ssPlusCircle, ssPeace, ssPixmap, ssCustom }

Public Member Functions

- QCPScatterStyle ()
- QCPScatterStyle (ScatterShape shape, double size=6)
- QCPScatterStyle (ScatterShape shape, const QColor &color, double size)
- QCPScatterStyle (ScatterShape shape, const QColor &color, const QColor &fill, double size)
- QCPScatterStyle (ScatterShape shape, const QPen &pen, const QBrush &brush, double size)
- QCPScatterStyle (const QPixmap &pixmap)
- QCPScatterStyle (const QPainterPath &customPath, const QPen &pen, const Q-Brush &brush=Qt::NoBrush, double size=6)
- double **size** () const
- ScatterShape **shape** () const
- QPen **pen** () const
- QBrush **brush** () const
- QPixmap **pixmap** () const
- QPainterPath **customPath** () const
- void setSize (double size)
- void setShape (ScatterShape shape)
- void setPen (const QPen &pen)
- void setBrush (const QBrush &brush)
- void setPixmap (const QPixmap &pixmap)
- void setCustomPath (const QPainterPath &customPath)
- bool isNone () const
- bool isPenDefined () const
- void applyTo (QCPPainter ∗painter, const QPen &defaultPen) const
- void drawShape (QCPPainter ∗painter, QPointF pos) const
- void drawShape (QCPPainter ∗painter, double x, double y) const

Protected Attributes

- double **mSize**
- ScatterShape **mShape**
- QPen **mPen**
- QBrush **mBrush**
- QPixmap **mPixmap**
- QPainterPath **mCustomPath**
- bool **mPenDefined**

7.68.1 Detailed Description

Represents the visual appearance of scatter points.

This class holds information about shape, color and size of scatter points. In plottables like QCPGraph it is used to store how scatter points shall be drawn. For example, QCPGraph::setScatterStyle takes a QCPScatterStyle instance.

A scatter style consists of a shape (setShape), a line color (setPen) and possibly a fill (setBrush), if the shape provides a fillable area. Further, the size of the shape can be controlled with setSize.

7.68.2 Specifying a scatter style

You can set all these configurations either by calling the respective functions on an instance:

Or you can use one of the various constructors that take different parameter combinations, making it easy to specify a scatter style in a single call, like so:

7.68.3 Leaving the color/pen up to the plottable

There are two constructors which leave the pen undefined: QCPScatterStyle() and - QCPScatterStyle(ScatterShape shape, double size). If those constructors are used, a call to isPenDefined will return false. It leads to scatter points that inherit the pen from the plottable that uses the scatter style. Thus, if such a scatter style is passed to Q- CPGraph, the line color of the graph (QCPGraph::setPen) will be used by the scatter points. This makes it very convenient to set up typical scatter settings:

Notice that it wasn't even necessary to explicitly call a QCPScatterStyle constructor. This works because QCPScatterStyle provides a constructor that can transform a - ScatterShape directly into a QCPScatterStyle instance (that's the QCPScatterStyle(- ScatterShape shape, double size) constructor with a default for *size*). In those cases, C++ allows directly supplying a ScatterShape, where actually a QCPScatterStyle is expected.

7.68.4 Custom shapes and pixmaps

QCPScatterStyle supports drawing custom shapes and arbitrary pixmaps as scatter points.

For custom shapes, you can provide a QPainterPath with the desired shape to the setCustomPath function or call the constructor that takes a painter path. The scatter shape will automatically be set to ssCustom.

For pixmaps, you call setPixmap with the desired QPixmap. Alternatively you can use the constructor that takes a QPixmap. The scatter shape will automatically be set to ssPixmap. Note that setSize does not influence the appearance of the pixmap.

7.68.5 Member Enumeration Documentation

7.68.5.1 enum QCPScatterStyle::ScatterShape

Defines the shape used for scatter points.

On plottables/items that draw scatters, the sizes of these visualizations (with exception of ssDot and ssPixmap) can be controlled with the setSize function. Scatters are drawn with the pen and brush specified with setPen and setBrush.

Enumerator:

ssNone no scatter symbols are drawn (e.g. in QCPGraph, data only represented with lines)

ssDot {ssDot.png} a single pixel (use ssDisc or ssCircle if you want a round shape with a certain radius)

ssCross {ssCross.png} a cross

ssPlus {ssPlus.png} a plus

ssCircle {ssCircle.png} a circle

ssDisc {ssDisc.png} a circle which is filled with the pen's color (not the brush as with ssCircle)

ssSquare {ssSquare.png} a square

ssDiamond {ssDiamond.png} a diamond

ssStar {ssStar.png} a star with eight arms, i.e. a combination of cross and plus

ssTriangle {ssTriangle.png} an equilateral triangle, standing on baseline

ssTriangleInverted {ssTriangleInverted.png} an equilateral triangle, standing on corner

ssCrossSquare {ssCrossSquare.png} a square with a cross inside

ssPlusSquare {ssPlusSquare.png} a square with a plus inside

ssCrossCircle {ssCrossCircle.png} a circle with a cross inside

ssPlusCircle {ssPlusCircle.png} a circle with a plus inside

ssPeace {ssPeace.png} a circle, with one vertical and two downward diagonal lines

ssPixmap a custom pixmap specified by setPixmap, centered on the data point coordinates

ssCustom custom painter operations are performed per scatter (As QPainter-Path, see setCustomPath)

7.68.6 Constructor & Destructor Documentation

7.68.6.1 QCPScatterStyle::QCPScatterStyle ( )

Creates a new QCPScatterStyle instance with size set to 6. No shape, pen or brush is defined.

Since the pen is undefined (isPenDefined returns false), the scatter color will be inherited from the plottable that uses this scatter style.

7.68.6.2 QCPScatterStyle::QCPScatterStyle (ScatterShape *shape*, double *size* = 6)

Creates a new QCPScatterStyle instance with shape set to *shape* and size to *size*. No pen or brush is defined.

Since the pen is undefined (isPenDefined returns false), the scatter color will be inherited from the plottable that uses this scatter style.

7.68.6.3 QCPScatterStyle::QCPScatterStyle (ScatterShape *shape*, const QColor & *color*, double *size*)

Creates a new QCPScatterStyle instance with shape set to *shape*, the pen color set to *color*, and size to *size*. No brush is defined, i.e. the scatter point will not be filled.

7.68.6.4 QCPScatterStyle::QCPScatterStyle (ScatterShape *shape*, const QColor & *color*, const QColor & *fill*, double *size*)

Creates a new QCPScatterStyle instance with shape set to *shape*, the pen color set to *color*, the brush color to *fill* (with a solid pattern), and size to *size*.

7.68.6.5 QCPScatterStyle::QCPScatterStyle (ScatterShape *shape*, const QPen & *pen*, const QBrush & *brush*, double *size*)

Creates a new QCPScatterStyle instance with shape set to *shape*, the pen set to *pen*, the brush to *brush*, and size to *size*.

Warning

In some cases it might be tempting to directly use a pen style like `Qt::NoPen` as *pen* and a color like `Qt::blue` as *brush*. Notice however, that the corresponding call
`QCPScatterStyle(QCPScatterShape::ssCircle, Qt::NoPen, Qt::blue, 5)`
doesn't necessarily lead C++ to use this constructor in some cases, but might mistake `Qt::NoPen` for a QColor and use the QCPScatterStyle(ScatterShape shape, const QColor &color, const QColor &fill, double size) constructor instead (which will lead to an unexpected look of the scatter points). To prevent this, be more explicit with the parameter types. For example, use `QBrush(Qt::blue)` instead of just `Qt::blue`, to clearly point out to the compiler that this constructor is wanted.

7.68.6.6 QCPScatterStyle::QCPScatterStyle (const QPixmap & *pixmap*)

Creates a new QCPScatterStyle instance which will show the specified *pixmap*. The scatter shape is set to ssPixmap.

7.68.6.7 QCPScatterStyle::QCPScatterStyle (const QPainterPath & *customPath,* const QPen & *pen,* const QBrush & *brush* = `Qt::NoBrush`, double *size* = `6`)

Creates a new QCPScatterStyle instance with a custom shape that is defined via *customPath*. The scatter shape is set to ssCustom.

The custom shape line will be drawn with *pen* and filled with *brush*. The size has a slightly different meaning than for built-in scatter points: The custom path will be drawn scaled by a factor of *size/6.0*. Since the default *size* is 6, the custom path will appear at a its natural size by default. To double the size of the path for example, set *size* to 12.

7.68.7 Member Function Documentation

7.68.7.1 void QCPScatterStyle::applyTo (QCPPainter ∗ *painter,* const QPen & *defaultPen*) const

Applies the pen and the brush of this scatter style to *painter*. If this scatter style has an undefined pen (isPenDefined), sets the pen of *painter* to *defaultPen* instead.

This function is used by plottables (or any class that wants to draw scatters) just before a number of scatters with this style shall be drawn with the *painter*.

See also

drawShape

7.68.7.2 void QCPScatterStyle::drawShape (QCPPainter ∗ *painter,* QPointF *pos*) const

Draws the scatter shape with *painter* at position *pos*.

This function does not modify the pen or the brush on the painter, as applyTo is meant to be called before scatter points are drawn with drawShape.

See also

applyTo

7.68.7.3 void QCPScatterStyle::drawShape (QCPPainter ∗ *painter,* double *x,* double *y*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Draws the scatter shape with *painter* at position *x* and *y*.

7.68.7.4 bool QCPScatterStyle::isNone ( ) const [inline]

Returns whether the scatter shape is ssNone.

See also

setShape

7.68.7.5 bool QCPScatterStyle::isPenDefined ( ) const [inline]

Returns whether a pen has been defined for this scatter style.

The pen is undefined if a constructor is called that does not carry *pen* as parameter. Those are QCPScatterStyle() and QCPScatterStyle(ScatterShape shape, double size). If the pen is left undefined, the scatter color will be inherited from the plottable that uses this scatter style.

See also

setPen

7.68.7.6 void QCPScatterStyle::setBrush (const QBrush & *brush*)

Sets the brush that will be used to fill scatter points to *brush*. Note that not all scatter shapes have fillable areas. For example, ssPlus does not while ssCircle does.

See also

setPen

7.68.7.7 void QCPScatterStyle::setCustomPath (const QPainterPath & *customPath*)

Sets the custom shape that will be drawn as scatter point to *customPath*.

The scatter shape is automatically set to ssCustom.

7.68.7.8 void QCPScatterStyle::setPen (const QPen & *pen*)

Sets the pen that will be used to draw scatter points to *pen*.

If the pen was previously undefined (see isPenDefined), the pen is considered defined after a call to this function, even if *pen* is `Qt::NoPen`.

See also

setBrush

7.68.7.9 void QCPScatterStyle::setPixmap (const QPixmap & *pixmap*)

Sets the pixmap that will be drawn as scatter point to *pixmap*.

Note that setSize does not influence the appearance of the pixmap.

The scatter shape is automatically set to ssPixmap.

7.68.7.10 void QCPScatterStyle::setShape (QCPScatterStyle::ScatterShape *shape*)

Sets the shape to *shape*.

Note that the calls setPixmap and setCustomPath automatically set the shape to ssPixmap and ssCustom, respectively.

See also

setSize

7.68.7.11 void QCPScatterStyle::setSize (double *size*)

Sets the size (pixel diameter) of the drawn scatter points to *size*.

See also

setShape

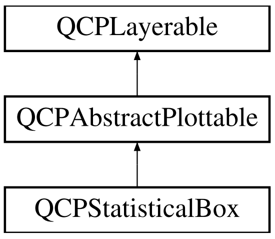
The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.69 QCPStatisticalBox Class Reference

A plottable representing a single statistical box in a plot.

Inheritance diagram for QCPStatisticalBox:



Public Member Functions

- QCPStatisticalBox (QCPAxis ∗keyAxis, QCPAxis ∗valueAxis)
- double **key** () const
- double **minimum** () const
- double **lowerQuartile** () const
- double **median** () const
- double **upperQuartile** () const
- double **maximum** () const
- QVector< double > **outliers** () const
- double **width** () const
- double **whiskerWidth** () const
- QPen **whiskerPen** () const
- QPen **whiskerBarPen** () const
- QPen **medianPen** () const
- QCPScatterStyle **outlierStyle** () const
- void setKey (double key)
- void setMinimum (double value)
- void setLowerQuartile (double value)
- void setMedian (double value)
- void setUpperQuartile (double value)
- void setMaximum (double value)
- void setOutliers (const QVector< double > &values)
- void setData (double key, double minimum, double lowerQuartile, double median, double upperQuartile, double maximum)
- void setWidth (double width)
- void setWhiskerWidth (double width)
- void setWhiskerPen (const QPen &pen)
- void setWhiskerBarPen (const QPen &pen)
- void setMedianPen (const QPen &pen)
- void setOutlierStyle (const QCPScatterStyle &style)
- virtual void clearData ()
- virtual double selectTest (const QPointF &pos, bool onlySelectable, QVariant ∗details=0) const

Protected Member Functions

- virtual void **draw** (QCPPainter ∗painter)
- virtual void **drawLegendIcon** (QCPPainter ∗painter, const QRectF &rect) const
- virtual QCPRange **getKeyRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual QCPRange **getValueRange** (bool &foundRange, SignDomain inSignDomain=sdBoth) const
- virtual void **drawQuartileBox** (QCPPainter ∗painter, QRectF ∗quartileBox=0) const
- virtual void **drawMedian** (QCPPainter ∗painter) const
- virtual void **drawWhiskers** (QCPPainter ∗painter) const
- virtual void **drawOutliers** (QCPPainter ∗painter) const

Protected Attributes

- QVector< double > **mOutliers**
- double **mKey**
- double **mMinimum**
- double **mLowerQuartile**
- double **mMedian**
- double **mUpperQuartile**
- double **mMaximum**
- double **mWidth**
- double **mWhiskerWidth**
- QPen **mWhiskerPen**
- QPen **mWhiskerBarPen**
- QPen **mMedianPen**
- QCPScatterStyle **mOutlierStyle**

Friends

- class **QCustomPlot**
- class **QCPLegend**

7.69.1 Detailed Description

A plottable representing a single statistical box in a plot.

To plot data, assign it with the individual parameter functions or use setData to set all parameters at once. The individual functions are:

- setMinimum
- setLowerQuartile
- setMedian

- setUpperQuartile
- setMaximum

Additionally you can define a list of outliers, drawn as scatter datapoints:

- setOutliers

7.69.2 Changing the appearance

The appearance of the box itself is controlled via setPen and setBrush. You may change the width of the box with setWidth in plot coordinates (not pixels).

Analog functions exist for the minimum/maximum-whiskers: setWhiskerPen, setWhiskerBarPen, setWhiskerWidth. The whisker width is the width of the bar at the top (maximum) and bottom (minimum).

The median indicator line has its own pen, setMedianPen.

If the whisker backbone pen is changed, make sure to set the capStyle to Qt::FlatCap. Else, the backbone line might exceed the whisker bars by a few pixels due to the pen cap being not perfectly flat.

The Outlier data points are drawn as normal scatter points. Their look can be controlled with setOutlierStyle

7.69.3 Usage

Like all data representing objects in QCustomPlot, the QCPStatisticalBox is a plottable. Usually, you first create an instance and add it to the m_customPlot:

and then modify the properties of the newly created plottable, e.g.:

7.69.4 Constructor & Destructor Documentation

7.69.4.1 QCPStatisticalBox::QCPStatisticalBox (QCPAxis ∗ *keyAxis,* QCPAxis ∗ *valueAxis*) [explicit]

Constructs a statistical box which uses *keyAxis* as its key axis ("x") and *valueAxis* as its value axis ("y"). *keyAxis* and *valueAxis* must reside in the same QCustomPlot instance and not have the same orientation. If either of these restrictions is violated, a corresponding message is printed to the debug output (qDebug), the construction is not aborted, though.

The constructed statistical box can be added to the plot with QCustomPlot::addPlottable, QCustomPlot then takes ownership of the statistical box.

7.69.5 Member Function Documentation

7.69.5.1 void QCPStatisticalBox::clearData ( ) [virtual]

Clears all data in the plottable.

Implements QCPAbstractPlottable.

7.69.5.2 double QCPStatisticalBox::selectTest (const QPointF & *pos,* bool *onlySelectable,* QVariant ∗ *details* = 0) const [virtual]

This function is used to decide whether a click hits a layerable object or not.

pos is a point in pixel coordinates on the QCustomPlot surface. This function returns the shortest pixel distance of this point to the object. If the object is either invisible or the distance couldn't be determined, -1.0 is returned. Further, if *onlySelectable* is true and the object is not selectable, -1.0 is returned, too.

If the object is represented not by single lines but by an area like a QCPItemText or the bars of a QCPBars plottable, a click inside the area should also be considered a hit. In these cases this function thus returns a constant value greater zero but still below the parent plot's selection tolerance. (typically the selectionTolerance multiplied by 0.99).

Providing a constant value for area objects allows selecting line objects even when they are obscured by such area objects, by clicking close to the lines (i.e. closer than 0.-99∗selectionTolerance).

The actual setting of the selection state is not done by this function. This is handled by the parent QCustomPlot when the mouseReleaseEvent occurs, and the finally selected object is notified via the selectEvent/deselectEvent methods.

details is an optional output parameter. Every layerable subclass may place any information in *details*. This information will be passed to selectEvent when the parent QCustomPlot decides on the basis of this selectTest call, that the object was successfully selected. The subsequent call to selectEvent will carry the *details*. This is useful for multi-part objects (like QCPAxis). This way, a possibly complex calculation to decide which part was clicked is only done once in selectTest. The result (i.e. the actually clicked part) can then be placed in *details*. So in the subsequent selectEvent, the decision which part was selected doesn't have to be done a second time for a single selection operation.

You may pass 0 as *details* to indicate that you are not interested in those selection details.

See also

selectEvent, deselectEvent, QCustomPlot::setInteractions

Implements QCPAbstractPlottable.

7.69.5.3 void QCPStatisticalBox::setData (double *key,* double *minimum,* double *lowerQuartile,* double *median,* double *upperQuartile,* double *maximum*)

Sets all parameters of the statistical box plot at once.

See also

setKey, setMinimum, setLowerQuartile, setMedian, setUpperQuartile, setMaximum

7.69.5.4 void QCPStatisticalBox::setKey (double *key*)

Sets the key coordinate of the statistical box.

7.69.5.5 void QCPStatisticalBox::setLowerQuartile (double *value*)

Sets the parameter "lower Quartile" of the statistical box plot. This is the lower end of the box. The lower and the upper quartiles are the two statistical quartiles around the median of the sample, they contain 50% of the sample data.

See also

setUpperQuartile, setPen, setBrush, setWidth

7.69.5.6 void QCPStatisticalBox::setMaximum (double *value*)

Sets the parameter "maximum" of the statistical box plot. This is the position of the upper whisker, typically the maximum measurement of the sample that's not considered an outlier.

See also

setMinimum, setWhiskerPen, setWhiskerBarPen, setWhiskerWidth

7.69.5.7 void QCPStatisticalBox::setMedian (double *value*)

Sets the parameter "median" of the statistical box plot. This is the value of the median mark inside the quartile box. The median separates the sample data in half (50% of the sample data is below/above the median).

See also

setMedianPen

7.69.5.8 void QCPStatisticalBox::setMedianPen (const QPen & *pen*)

Sets the pen used for drawing the median indicator line inside the statistical box.

7.69.5.9 void QCPStatisticalBox::setMinimum (double *value*)

Sets the parameter "minimum" of the statistical box plot. This is the position of the lower whisker, typically the minimum measurement of the sample that's not considered an outlier.

See also

setMaximum, setWhiskerPen, setWhiskerBarPen, setWhiskerWidth

7.69.5.10 void QCPStatisticalBox::setOutliers (const QVector< double > & *values*)

Sets a vector of outlier values that will be drawn as scatters. Any data points in the sample that are not within the whiskers (setMinimum, setMaximum) should be considered outliers and displayed as such.

See also

setOutlierStyle

7.69.5.11 void QCPStatisticalBox::setOutlierStyle (const QCPScatterStyle & *style*)

Sets the appearance of the outlier data points.

See also

setOutliers

7.69.5.12 void QCPStatisticalBox::setUpperQuartile (double *value*)

Sets the parameter "upper Quartile" of the statistical box plot. This is the upper end of the box. The lower and the upper quartiles are the two statistical quartiles around the median of the sample, they contain 50% of the sample data.

See also

setLowerQuartile, setPen, setBrush, setWidth

7.69.5.13 void QCPStatisticalBox::setWhiskerBarPen (const QPen & *pen*)

Sets the pen used for drawing the whisker bars (Those are the lines parallel to the key axis at each end of the whisker backbone).

See also

setWhiskerPen

7.69.5.14 void QCPStatisticalBox::setWhiskerPen (const QPen & *pen*)

Sets the pen used for drawing the whisker backbone (That's the line parallel to the value axis).

Make sure to set the *pen* capStyle to Qt::FlatCap to prevent the whisker backbone from reaching a few pixels past the whisker bars, when using a non-zero pen width.

See also

setWhiskerBarPen

7.69.5.15 void QCPStatisticalBox::setWhiskerWidth (double *width*)

Sets the width of the whiskers (setMinimum, setMaximum) in key coordinates.

See also

setWidth

7.69.5.16 void QCPStatisticalBox::setWidth (double *width*)

Sets the width of the box in key coordinates.

See also

setWhiskerWidth

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp

7.70 QCustomPlot Class Reference

The central class of the library. This is the QWidget which displays the plot and interacts with the user.

Public Types

- enum LayerInsertMode { limBelow, limAbove }
- enum RefreshPriority { rpImmediate, rpQueued, rpHint }

Signals

- void mouseDoubleClick (QMouseEvent ∗event)
- void mousePress (QMouseEvent ∗event)
- void mouseMove (QMouseEvent ∗event)
- void mouseRelease (QMouseEvent ∗event)
- void mouseWheel (QWheelEvent ∗event)
- void plottableClick (QCPAbstractPlottable ∗plottable, QMouseEvent ∗event)
- void plottableDoubleClick (QCPAbstractPlottable ∗plottable, QMouseEvent ∗event)
- void itemClick (QCPAbstractItem ∗item, QMouseEvent ∗event)
- void itemDoubleClick (QCPAbstractItem ∗item, QMouseEvent ∗event)
- void axisClick (QCPAxis ∗axis, QCPAxis::SelectablePart part, QMouseEvent ∗event)
- void axisDoubleClick (QCPAxis ∗axis, QCPAxis::SelectablePart part, QMouse-Event ∗event)
- void legendClick (QCPLegend ∗legend, QCPAbstractLegendItem ∗item, Q-MouseEvent ∗event)
- void legendDoubleClick (QCPLegend ∗legend, QCPAbstractLegendItem ∗item, QMouseEvent ∗event)
- void titleClick (QMouseEvent ∗event, QCPPlotTitle ∗title)
- void titleDoubleClick (QMouseEvent ∗event, QCPPlotTitle ∗title)
- void selectionChangedByUser ()
- void beforeReplot ()
- void afterReplot ()

Public Member Functions

- QCustomPlot (QWidget ∗parent=0)
- QRect viewport () const
- QPixmap **background** () const
- bool **backgroundScaled** () const
- Qt::AspectRatioMode **backgroundScaledMode** () const
- QCPLayoutGrid ∗ plotLayout () const
- QCP::AntialiasedElements **antialiasedElements** () const
- QCP::AntialiasedElements **notAntialiasedElements** () const
- bool **autoAddPlottableToLegend** () const
- const QCP::Interactions **interactions** () const
- int **selectionTolerance** () const
- bool **noAntialiasingOnDrag** () const
- QCP::PlottingHints **plottingHints** () const
- Qt::KeyboardModifier **multiSelectModifier** () const
- void setViewport (const QRect &rect)
- void setBackground (const QPixmap &pm)
- void setBackground (const QPixmap &pm, bool scaled, Qt::AspectRatioMode mode=Qt::KeepAspectRatioByExpanding)

- void setBackground (const QBrush &brush)
- void setBackgroundScaled (bool scaled)
- void setBackgroundScaledMode (Qt::AspectRatioMode mode)
- void setAntialiasedElements (const QCP::AntialiasedElements &antialiased-Elements)
- void setAntialiasedElement (QCP::AntialiasedElement antialiasedElement, bool enabled=true)
- void setNotAntialiasedElements (const QCP::AntialiasedElements &not-AntialiasedElements)
- void setNotAntialiasedElement (QCP::AntialiasedElement notAntialiased-Element, bool enabled=true)
- void setAutoAddPlottableToLegend (bool on)
- void setInteractions (const QCP::Interactions &interactions)
- void setInteraction (const QCP::Interaction &interaction, bool enabled=true)
- void setSelectionTolerance (int pixels)
- void setNoAntialiasingOnDrag (bool enabled)
- void setPlottingHints (const QCP::PlottingHints &hints)
- void setPlottingHint (QCP::PlottingHint hint, bool enabled=true)
- void setMultiSelectModifier (Qt::KeyboardModifier modifier)
- QCPAbstractPlottable ∗ plottable (int index)
- QCPAbstractPlottable ∗ plottable ()
- bool addPlottable (QCPAbstractPlottable ∗plottable)
- bool removePlottable (QCPAbstractPlottable ∗plottable)
- bool removePlottable (int index)
- int clearPlottables ()
- int plottableCount () const
- QList< QCPAbstractPlottable ∗ > selectedPlottables () const
- QCPAbstractPlottable ∗ plottableAt (const QPointF &pos, bool onlySelectable=false) const
- bool hasPlottable (QCPAbstractPlottable ∗plottable) const
- QCPGraph ∗ graph (int index) const
- QCPGraph ∗ graph () const
- QCPGraph ∗ addGraph (QCPAxis ∗keyAxis=0, QCPAxis ∗valueAxis=0)
- bool removeGraph (QCPGraph ∗graph)
- bool removeGraph (int index)
- int clearGraphs ()
- int graphCount () const
- QList< QCPGraph ∗ > selectedGraphs () const
- QCPAbstractItem ∗ item (int index) const
- QCPAbstractItem ∗ item () const
- bool addItem (QCPAbstractItem ∗item)
- bool removeItem (QCPAbstractItem ∗item)
- bool removeItem (int index)
- int clearItems ()
- int itemCount () const
- QList< QCPAbstractItem ∗ > selectedItems () const
- QCPAbstractItem ∗ itemAt (const QPointF &pos, bool onlySelectable=false) const

- bool hasItem (QCPAbstractItem ∗item) const
- QCPLayer ∗ layer (const QString &name) const
- QCPLayer ∗ layer (int index) const
- QCPLayer ∗ currentLayer () const
- bool setCurrentLayer (const QString &name)
- bool setCurrentLayer (QCPLayer ∗layer)
- int layerCount () const
- bool addLayer (const QString &name, QCPLayer ∗otherLayer=0, LayerInsert-Mode insertMode=limAbove)
- bool removeLayer (QCPLayer ∗layer)
- bool moveLayer (QCPLayer ∗layer, QCPLayer ∗otherLayer, LayerInsertMode insertMode=limAbove)
- int axisRectCount () const
- QCPAxisRect ∗ axisRect (int index=0) const
- QList< QCPAxisRect ∗ > axisRects () const
- QCPLayoutElement ∗ layoutElementAt (const QPointF &pos) const
- Q_SLOT void rescaleAxes (bool onlyVisiblePlottables=false)
- QList< QCPAxis ∗ > selectedAxes () const
- QList< QCPLegend ∗ > selectedLegends () const
- Q_SLOT void deselectAll ()
- bool savePdf (const QString &fileName, bool noCosmeticPen=false, int width=0, int height=0, const QString &pdfCreator=QString(), const QString &pdfTitle=Q-String())
- bool savePng (const QString &fileName, int width=0, int height=0, double scale=1.0, int quality=-1)
- bool saveJpg (const QString &fileName, int width=0, int height=0, double scale=1.0, int quality=-1)
- bool saveBmp (const QString &fileName, int width=0, int height=0, double scale=1.0)
- bool saveRastered (const QString &fileName, int width, int height, double scale, const char ∗format, int quality=-1)
- QPixmap toPixmap (int width=0, int height=0, double scale=1.0)
- void toPainter (QCPPainter ∗painter, int width=0, int height=0)
- Q_SLOT void replot (QCustomPlot::RefreshPriority refreshPriority=QCustom-Plot::rpHint)

Public Attributes

- QCPAxis ∗ xAxis
- QCPAxis ∗ yAxis
- QCPAxis ∗ xAxis2
- QCPAxis ∗ yAxis2
- QCPLegend ∗ legend

Protected Member Functions

- virtual QSize **minimumSizeHint** () const
- virtual QSize **sizeHint** () const
- virtual void **paintEvent** (QPaintEvent ∗event)
- virtual void **resizeEvent** (QResizeEvent ∗event)
- virtual void **mouseDoubleClickEvent** (QMouseEvent ∗event)
- virtual void **mousePressEvent** (QMouseEvent ∗event)
- virtual void **mouseMoveEvent** (QMouseEvent ∗event)
- virtual void **mouseReleaseEvent** (QMouseEvent ∗event)
- virtual void **wheelEvent** (QWheelEvent ∗event)
- virtual void **draw** (QCPPainter ∗painter)
- virtual void **axisRemoved** (QCPAxis ∗axis)
- virtual void **legendRemoved** (QCPLegend ∗legend)
- void **updateLayerIndices** () const
- QCPLayerable ∗ **layerableAt** (const QPointF &pos, bool onlySelectable, QVariant ∗selectionDetails=0) const
- void **drawBackground** (QCPPainter ∗painter)

Protected Attributes

- QRect **mViewport**
- QCPLayoutGrid ∗ **mPlotLayout**
- bool **mAutoAddPlottableToLegend**
- QList< QCPAbstractPlottable ∗ > **mPlottables**
- QList< QCPGraph ∗ > **mGraphs**
- QList< QCPAbstractItem ∗ > **mItems**
- QList< QCPLayer ∗ > **mLayers**
- QCP::AntialiasedElements **mAntialiasedElements**
- QCP::AntialiasedElements **mNotAntialiasedElements**
- QCP::Interactions **mInteractions**
- int **mSelectionTolerance**
- bool **mNoAntialiasingOnDrag**
- QBrush **mBackgroundBrush**
- QPixmap **mBackgroundPixmap**
- QPixmap **mScaledBackgroundPixmap**
- bool **mBackgroundScaled**
- Qt::AspectRatioMode **mBackgroundScaledMode**
- QCPLayer ∗ **mCurrentLayer**
- QCP::PlottingHints **mPlottingHints**
- Qt::KeyboardModifier **mMultiSelectModifier**
- QPixmap **mPaintBuffer**
- QPoint **mMousePressPos**
- QPointer< QCPLayoutElement > **mMouseEventElement**
- bool **mReplotting**

Friends

- class **QCPLegend**
- class **QCPAxis**
- class **QCPLayer**
- class **QCPAxisRect**

7.70.1 Detailed Description

The central class of the library. This is the QWidget which displays the plot and interacts with the user.

For tutorials on how to use QCustomPlot, see the website

`http://www.qcustomplot.com/`

7.70.2 Member Enumeration Documentation

7.70.2.1 enum QCustomPlot::LayerInsertMode

Defines how a layer should be inserted relative to an other layer.

See also

addLayer, moveLayer

Enumerator:

limBelow Layer is inserted below other layer.

limAbove Layer is inserted above other layer.

7.70.2.2 enum QCustomPlot::RefreshPriority

Defines with what timing the QCustomPlot surface is refreshed after a replot.

See also

replot

Enumerator:

rpImmediate The QCustomPlot surface is immediately refreshed, by calling Q-Widget::repaint() after the replot.

rpQueued Queues the refresh such that it is performed at a slightly delayed point in time after the replot, by calling QWidget::update() after the replot.

rpHint Whether to use immediate repaint or queued update depends on whether the plotting hint QCP::phForceRepaint is set, see setPlottingHints.

7.70.3 Constructor & Destructor Documentation

7.70.3.1 QCustomPlot::QCustomPlot (QWidget * *parent* = `0`) `[explicit]`

Constructs a QCustomPlot and sets reasonable default values.

7.70.4 Member Function Documentation

7.70.4.1 QCPGraph * QCustomPlot::addGraph (QCPAxis * *keyAxis* = `0`, QCPAxis * *valueAxis* = `0`)

Creates a new graph inside the plot. If *keyAxis* and *valueAxis* are left unspecified (0), the bottom (xAxis) is used as key and the left (yAxis) is used as value axis. If specified, *keyAxis* and *valueAxis* must reside in this QCustomPlot.

keyAxis will be used as key axis (typically "x") and *valueAxis* as value axis (typically "y") for the graph.

Returns a pointer to the newly created graph, or 0 if adding the graph failed.

See also

graph, graphCount, removeGraph, clearGraphs

7.70.4.2 bool QCustomPlot::addItem (QCPAbstractItem * *item*)

Adds the specified item to the plot. QCustomPlot takes ownership of the item.

Returns true on success, i.e. when *item* wasn't already in the plot and the parent plot of *item* is this QCustomPlot.

See also

item, itemCount, removeItem, clearItems

7.70.4.3 bool QCustomPlot::addLayer (const QString & *name*, QCPLayer * *otherLayer* = `0`, QCustomPlot::LayerInsertMode *insertMode* = limAbove)

Adds a new layer to this QCustomPlot instance. The new layer will have the name *name*, which must be unique. Depending on *insertMode*, it is positioned either below or above *otherLayer*.

Returns true on success, i.e. if there is no other layer named *name* and *otherLayer* is a valid layer inside this QCustomPlot.

If *otherLayer* is 0, the highest layer in the QCustomPlot will be used.

For an explanation of what layers are in QCustomPlot, see the documentation of QCP-Layer.

See also

layer, moveLayer, removeLayer

7.70.4.4 bool QCustomPlot::addPlottable (QCPAbstractPlottable ∗ *plottable*)

Adds the specified plottable to the plot and, if setAutoAddPlottableToLegend is enabled, to the legend (QCustomPlot::legend). QCustomPlot takes ownership of the plottable.

Returns true on success, i.e. when *plottable* isn't already in the plot and the parent plot of *plottable* is this QCustomPlot (the latter is controlled by what axes were passed in the plottable's constructor).

See also

plottable, plottableCount, removePlottable, clearPlottables

7.70.4.5 void QCustomPlot::afterReplot ( ) `[signal]`

This signal is emitted immediately after a replot has taken place (caused by a call to the slot replot).

It is safe to mutually connect the replot slot with this signal on two QCustomPlots to make them replot synchronously, it won't cause an infinite recursion.

See also

replot, beforeReplot

7.70.4.6 void QCustomPlot::axisClick (QCPAxis ∗ *axis*, QCPAxis::SelectablePart *part*, QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when an axis is clicked.

event is the mouse event that caused the click, *axis* is the axis that received the click and *part* indicates the part of the axis that was clicked.

See also

axisDoubleClick

7.70.4.7 void QCustomPlot::axisDoubleClick (QCPAxis ∗ *axis*, QCPAxis::SelectablePart *part*, QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when an axis is double clicked.

event is the mouse event that caused the click, *axis* is the axis that received the click and *part* indicates the part of the axis that was clicked.

See also

axisClick

7.70.4.8 QCPAxisRect ∗ QCustomPlot::axisRect (int *index* = `0`) const

Returns the axis rect with *index*.

Initially, only one axis rect (with index 0) exists in the plot. If multiple axis rects were added, all of them may be accessed with this function in a linear fashion (even when they are nested in a layout hierarchy or inside other axis rects via QCPAxisRect::insetLayout).

See also

axisRectCount, axisRects

7.70.4.9 int QCustomPlot::axisRectCount ( ) const

Returns the number of axis rects in the plot.

All axis rects can be accessed via QCustomPlot::axisRect().

Initially, only one axis rect exists in the plot.

See also

axisRect, axisRects

7.70.4.10 QList< QCPAxisRect ∗ > QCustomPlot::axisRects ( ) const

Returns all axis rects in the plot.

See also

axisRectCount, axisRect

7.70.4.11 void QCustomPlot::beforeReplot ( ) `[signal]`

This signal is emitted immediately before a replot takes place (caused by a call to the slot replot).

It is safe to mutually connect the replot slot with this signal on two QCustomPlots to make them replot synchronously, it won't cause an infinite recursion.

See also

replot, afterReplot

7.70.4.12 int QCustomPlot::clearGraphs ( )

Removes all graphs from the plot (and the QCustomPlot::legend, if necessary).

Returns the number of graphs removed.

See also

removeGraph

7.70.4.13 int QCustomPlot::clearItems ( )

Removes all items from the plot.

Returns the number of items removed.

See also

removeItem

7.70.4.14 int QCustomPlot::clearPlottables ( )

Removes all plottables from the plot (and the QCustomPlot::legend, if necessary).

Returns the number of plottables removed.

See also

removePlottable

7.70.4.15 QCPLayer ∗ QCustomPlot::currentLayer ( ) const

Returns the layer that is set as current layer (see setCurrentLayer).

7.70.4.16 void QCustomPlot::deselectAll ( )

Deselects all layerables (plottables, items, axes, legends,...) of the QCustomPlot.

Since calling this function is not a user interaction, this does not emit the selection-ChangedByUser signal. The individual selectionChanged signals are emitted though, if the objects were previously selected.

See also

setInteractions, selectedPlottables, selectedItems, selectedAxes, selectedLegends

7.70.4.17 QCPGraph ∗ QCustomPlot::graph (int *index*) const

Returns the graph with *index*. If the index is invalid, returns 0.

There is an overloaded version of this function with no parameter which returns the last created graph, see QCustomPlot::graph()

See also

graphCount, addGraph

7.70.4.18 QCPGraph ∗ QCustomPlot::graph ( ) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Returns the last graph, that was created with addGraph. If there are no graphs in the plot, returns 0.

See also

graphCount, addGraph

7.70.4.19 int QCustomPlot::graphCount ( ) const

Returns the number of currently existing graphs in the plot

See also

graph, addGraph

7.70.4.20 bool QCustomPlot::hasItem (QCPAbstractItem ∗ *item*) const

Returns whether this QCustomPlot contains the *item*.

See also

addItem

7.70.4.21 bool QCustomPlot::hasPlottable (QCPAbstractPlottable ∗ *plottable*) const

Returns whether this QCustomPlot instance contains the *plottable*.

See also

addPlottable

7.70.4.22 QCPAbstractItem ∗ QCustomPlot::item (int *index*) const

Returns the item with *index*. If the index is invalid, returns 0.

There is an overloaded version of this function with no parameter which returns the last added item, see QCustomPlot::item()

See also

itemCount, addItem

7.70.4.23 QCPAbstractItem ∗ QCustomPlot::item ( ) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Returns the last item, that was added with addItem. If there are no items in the plot, returns 0.

See also

itemCount, addItem

7.70.4.24 QCPAbstractItem ∗ QCustomPlot::itemAt (const QPointF & *pos*, bool *onlySelectable* = `false`) const

Returns the item at the pixel position *pos*. Items that only consist of single lines (e.g. QCPItemLine or QCPItemCurve) have a tolerance band around them, see setSelectionTolerance. If multiple items come into consideration, the one closest to *pos* is returned.

If *onlySelectable* is true, only items that are selectable (QCPAbstractItem::setSelectable) are considered.

If there is no item at *pos*, the return value is 0.

See also

plottableAt, layoutElementAt

7.70.4.25 void QCustomPlot::itemClick (QCPAbstractItem ∗ *item*, QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when an item is clicked.

event is the mouse event that caused the click and *item* is the item that received the click.

See also

itemDoubleClick

7.70.4.26 int QCustomPlot::itemCount ( ) const

Returns the number of currently existing items in the plot

See also

item, addItem

7.70.4.27 void QCustomPlot::itemDoubleClick (QCPAbstractItem ∗ *item*, QMouseEvent ∗ *event*) [signal]

This signal is emitted when an item is double clicked.

event is the mouse event that caused the click and *item* is the item that received the click.

See also

itemClick

7.70.4.28 QCPLayer ∗ QCustomPlot::layer (const QString & *name*) const

Returns the layer with the specified *name*. If there is no layer with the specified name, 0 is returned.

Layer names are case-sensitive.

See also

addLayer, moveLayer, removeLayer

7.70.4.29 QCPLayer ∗ QCustomPlot::layer (int *index*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Returns the layer by *index*. If the index is invalid, 0 is returned.

See also

addLayer, moveLayer, removeLayer

7.70.4.30 int QCustomPlot::layerCount ( ) const

Returns the number of currently existing layers in the plot

See also

layer, addLayer

7.70.4.31 QCPLayoutElement ∗ QCustomPlot::layoutElementAt (const QPointF & *pos*) const

Returns the layout element at pixel position *pos*. If there is no element at that position, returns 0.

Only visible elements are used. If QCPLayoutElement::setVisible on the element itself or on any of its parent elements is set to false, it will not be considered.

See also

itemAt, plottableAt

7.70.4.32 void QCustomPlot::legendClick (QCPLegend ∗ *legend*, QCPAbstractLegendItem ∗ *item*, QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when a legend (item) is clicked.

event is the mouse event that caused the click, *legend* is the legend that received the click and *item* is the legend item that received the click. If only the legend and no item is clicked, *item* is 0. This happens for a click inside the legend padding or the space between two items.

See also

legendDoubleClick

7.70.4.33 void QCustomPlot::legendDoubleClick (QCPLegend ∗ *legend*, QCPAbstractLegendItem ∗ *item*, QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when a legend (item) is double clicked.

event is the mouse event that caused the click, *legend* is the legend that received the click and *item* is the legend item that received the click. If only the legend and no item is clicked, *item* is 0. This happens for a click inside the legend padding or the space between two items.

See also

legendClick

7.70.4.34 void QCustomPlot::mouseDoubleClick (QMouseEvent ∗ *event*) `[signal]`

This signal is emitted when the QCustomPlot receives a mouse double click event.

7.70.4.35 void QCustomPlot::mouseMove (QMouseEvent ∗ *event*) [signal]

This signal is emitted when the QCustomPlot receives a mouse move event.

It is emitted before QCustomPlot handles any other mechanism like range dragging. So a slot connected to this signal can still influence the behaviour e.g. with QCPAxisRect::setRangeDrag or QCPAxisRect::setRangeDragAxes.

Warning

It is discouraged to change the drag-axes with QCPAxisRect::setRangeDragAxes here, because the dragging starting point was saved the moment the mouse was pressed. Thus it only has a meaning for the range drag axes that were set at that moment. If you want to change the drag axes, consider doing this in the mousePress signal instead.

7.70.4.36 void QCustomPlot::mousePress (QMouseEvent ∗ *event*) [signal]

This signal is emitted when the QCustomPlot receives a mouse press event.

It is emitted before QCustomPlot handles any other mechanism like range dragging. So a slot connected to this signal can still influence the behaviour e.g. with QCPAxisRect::setRangeDrag or QCPAxisRect::setRangeDragAxes.

7.70.4.37 void QCustomPlot::mouseRelease (QMouseEvent ∗ *event*) [signal]

This signal is emitted when the QCustomPlot receives a mouse release event.

It is emitted before QCustomPlot handles any other mechanisms like object selection. So a slot connected to this signal can still influence the behaviour e.g. with setInteractions or QCPAbstractPlottable::setSelectable.

7.70.4.38 void QCustomPlot::mouseWheel (QWheelEvent ∗ *event*) [signal]

This signal is emitted when the QCustomPlot receives a mouse wheel event.

It is emitted before QCustomPlot handles any other mechanisms like range zooming. So a slot connected to this signal can still influence the behaviour e.g. with QCPAxisRect::setRangeZoom, QCPAxisRect::setRangeZoomAxes or QCPAxisRect::setRangeZoomFactor.

7.70.4.39 bool QCustomPlot::moveLayer (QCPLayer ∗ *layer,* QCPLayer ∗ *otherLayer,* QCustomPlot::LayerInsertMode *insertMode* = limAbove)

Moves the specified *layer* either above or below *otherLayer*. Whether it's placed above or below is controlled with *insertMode*.

Returns true on success, i.e. when both *layer* and *otherLayer* are valid layers in the QCustomPlot.

See also

layer, addLayer, moveLayer

7.70.4.40 QCPLayoutGrid ∗ QCustomPlot::plotLayout ( ) const [inline]

Returns the top level layout of this QCustomPlot instance. It is a QCPLayoutGrid, initially containing just one cell with the main QCPAxisRect inside.

7.70.4.41 QCPAbstractPlottable ∗ QCustomPlot::plottable (int *index*)

Returns the plottable with *index*. If the index is invalid, returns 0.

There is an overloaded version of this function with no parameter which returns the last added plottable, see QCustomPlot::plottable()

See also

plottableCount, addPlottable

7.70.4.42 QCPAbstractPlottable ∗ QCustomPlot::plottable ( )

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Returns the last plottable that was added with addPlottable. If there are no plottables in the plot, returns 0.

See also

plottableCount, addPlottable

7.70.4.43 QCPAbstractPlottable ∗ QCustomPlot::plottableAt (const QPointF & *pos*, bool *onlySelectable* = `false`) const

Returns the plottable at the pixel position *pos*. Plottables that only consist of single lines (like graphs) have a tolerance band around them, see setSelectionTolerance. If multiple plottables come into consideration, the one closest to *pos* is returned.

If *onlySelectable* is true, only plottables that are selectable (QCPAbstractPlottable::setSelectable) are considered.

If there is no plottable at *pos*, the return value is 0.

See also

itemAt, layoutElementAt

7.70.4.44 void QCustomPlot::plottableClick (QCPAbstractPlottable ∗ *plottable,* QMouseEvent ∗ *event*) [signal]

This signal is emitted when a plottable is clicked.

event is the mouse event that caused the click and *plottable* is the plottable that received the click.

See also

plottableDoubleClick

7.70.4.45 int QCustomPlot::plottableCount ( ) const

Returns the number of currently existing plottables in the plot

See also

plottable, addPlottable

7.70.4.46 void QCustomPlot::plottableDoubleClick (QCPAbstractPlottable ∗ *plottable,* QMouseEvent ∗ *event*) [signal]

This signal is emitted when a plottable is double clicked.

event is the mouse event that caused the click and *plottable* is the plottable that received the click.

See also

plottableClick

7.70.4.47 bool QCustomPlot::removeGraph (QCPGraph ∗ *graph*)

Removes the specified *graph* from the plot and, if necessary, from the QCustomPlot::legend. If any other graphs in the plot have a channel fill set towards the removed graph, the channel fill property of those graphs is reset to zero (no channel fill).

Returns true on success.

See also

clearGraphs

7.70.4.48 bool QCustomPlot::removeGraph (int *index*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes the graph by its *index*.

7.70.4.49 bool QCustomPlot::removeItem (QCPAbstractItem ∗ *item*)

Removes the specified item from the plot.

Returns true on success.

See also

addItem, clearItems

7.70.4.50 bool QCustomPlot::removeItem (int *index*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes the item by its *index*.

7.70.4.51 bool QCustomPlot::removeLayer (QCPLayer ∗ *layer*)

Removes the specified *layer* and returns true on success.

All layerables (e.g. plottables and items) on the removed layer will be moved to the layer below *layer*. If *layer* is the bottom layer, the layerables are moved to the layer above. In both cases, the total rendering order of all layerables in the QCustomPlot is preserved.

If *layer* is the current layer (setCurrentLayer), the layer below (or above, if bottom layer) becomes the new current layer.

It is not possible to remove the last layer of the plot.

See also

layer, addLayer, moveLayer

7.70.4.52 bool QCustomPlot::removePlottable (QCPAbstractPlottable ∗ *plottable*)

Removes the specified plottable from the plot and, if necessary, from the legend (QCustomPlot::legend).

Returns true on success.

See also

addPlottable, clearPlottables

7.70.4.53 bool QCustomPlot::removePlottable (int *index*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes the plottable by its *index*.

7.70.4.54 void QCustomPlot::replot (QCustomPlot::RefreshPriority *refreshPriority* = QCustomPlot::rpHint)

Causes a complete replot into the internal buffer. Finally, update() is called, to redraw the buffer on the QCustomPlot widget surface. This is the method that must be called to make changes, for example on the axis ranges or data points of graphs, visible.

Under a few circumstances, QCustomPlot causes a replot by itself. Those are resize events of the QCustomPlot widget and user interactions (object selection and range dragging/zooming).

Before the replot happens, the signal beforeReplot is emitted. After the replot, afterReplot is emitted. It is safe to mutually connect the replot slot with any of those two signals on two QCustomPlots to make them replot synchronously, it won't cause an infinite recursion.

7.70.4.55 void QCustomPlot::rescaleAxes (bool *onlyVisiblePlottables* = `false`)

Rescales the axes such that all plottables (like graphs) in the plot are fully visible.

if *onlyVisiblePlottables* is set to true, only the plottables that have their visibility set to true (QCPLayerable::setVisible), will be used to rescale the axes.

See also

QCPAbstractPlottable::rescaleAxes, QCPAxis::rescale

7.70.4.56 bool QCustomPlot::saveBmp (const QString & *fileName*, int *width* = `0`, int *height* = `0`, double *scale* = `1.0`)

Saves a BMP image file to *fileName* on disc. The output plot will have the dimensions *width* and *height* in pixels. If either *width* or *height* is zero, the exported image will have the same dimensions as the QCustomPlot widget currently has. Line widths and texts etc. are not scaled up when larger widths/heights are used. If you want that effect, use the *scale* parameter.

For example, if you set both *width* and *height* to 100 and *scale* to 2, you will end up with an image file of size 200∗200 in which all graphical elements are scaled up by factor 2 (line widths, texts, etc.). This scaling is not done by stretching a 100∗100 image, the result will have full 200∗200 pixel resolution.

If you use a high scaling factor, it is recommended to enable antialiasing for all elements via temporarily setting QCustomPlot::setAntialiasedElements to QCP::aeAll as this allows QCustomPlot to place objects with sub-pixel accuracy.

Warning

If calling this function inside the constructor of the parent of the QCustomPlot widget (i.e. the MainWindow constructor, if QCustomPlot is inside the MainWindow), always provide explicit non-zero widths and heights. If you leave *width* or *height* as

0 (default), this function uses the current width and height of the QCustomPlot widget. However, in Qt, these aren't defined yet inside the constructor, so you would get an image that has strange widths/heights.

The objects of the plot will appear in the current selection state. If you don't want any selected objects to be painted in their selected look, deselect everything with deselectAll before calling this function.

Returns true on success. If this function fails, most likely the BMP format isn't supported by the system, see Qt docs about QImageWriter::supportedImageFormats().

See also

savePdf, savePng, saveJpg, saveRastered

7.70.4.57 bool QCustomPlot::saveJpg (const QString & *fileName,* int *width* = `0`, int *height* = `0`, double *scale* = `1.0`, int *quality* = `-1`)

Saves a JPG image file to *fileName* on disc. The output plot will have the dimensions *width* and *height* in pixels. If either *width* or *height* is zero, the exported image will have the same dimensions as the QCustomPlot widget currently has. Line widths and texts etc. are not scaled up when larger widths/heights are used. If you want that effect, use the *scale* parameter.

For example, if you set both *width* and *height* to 100 and *scale* to 2, you will end up with an image file of size 200∗200 in which all graphical elements are scaled up by factor 2 (line widths, texts, etc.). This scaling is not done by stretching a 100∗100 image, the result will have full 200∗200 pixel resolution.

If you use a high scaling factor, it is recommended to enable antialiasing for all elements via temporarily setting QCustomPlot::setAntialiasedElements to QCP::aeAll as this allows QCustomPlot to place objects with sub-pixel accuracy.

Warning

If calling this function inside the constructor of the parent of the QCustomPlot widget (i.e. the MainWindow constructor, if QCustomPlot is inside the MainWindow), always provide explicit non-zero widths and heights. If you leave *width* or *height* as 0 (default), this function uses the current width and height of the QCustomPlot widget. However, in Qt, these aren't defined yet inside the constructor, so you would get an image that has strange widths/heights.

The objects of the plot will appear in the current selection state. If you don't want any selected objects to be painted in their selected look, deselect everything with deselectAll before calling this function.

JPG compression can be controlled with the *quality* parameter which must be between 0 and 100 or -1 to use the default setting.

Returns true on success. If this function fails, most likely the JPG format isn't supported by the system, see Qt docs about QImageWriter::supportedImageFormats().

See also

savePdf, savePng, saveBmp, saveRastered

7.70.4.58 bool QCustomPlot::savePdf (const QString & *fileName*, bool *noCosmeticPen* = `false`, int *width* = `0`, int *height* = `0`, const QString & *pdfCreator* = `QString()`, const QString & *pdfTitle* = `QString()`)

Saves a PDF with the vectorized plot to the file *fileName*. The axis ratio as well as the scale of texts and lines will be derived from the specified *width* and *height*. This means, the output will look like the normal on-screen output of a QCustomPlot widget with the corresponding pixel width and height. If either *width* or *height* is zero, the exported image will have the same dimensions as the QCustomPlot widget currently has.

noCosmeticPen disables the use of cosmetic pens when drawing to the PDF file. - Cosmetic pens are pens with numerical width 0, which are always drawn as a one pixel wide line, no matter what zoom factor is set in the PDF-Viewer. For more information about cosmetic pens, see the QPainter and QPen documentation.

The objects of the plot will appear in the current selection state. If you don't want any selected objects to be painted in their selected look, deselect everything with deselectAll before calling this function.

Returns true on success.

Warning

- If you plan on editing the exported PDF file with a vector graphics editor like Inkscape, it is advised to set *noCosmeticPen* to true to avoid losing those cosmetic lines (which might be quite many, because cosmetic pens are the default for e.g. axes and tick marks).
- If calling this function inside the constructor of the parent of the QCustomPlot widget (i.e. the MainWindow constructor, if QCustomPlot is inside the Main-Window), always provide explicit non-zero widths and heights. If you leave *width* or *height* as 0 (default), this function uses the current width and height of the QCustomPlot widget. However, in Qt, these aren't defined yet inside the constructor, so you would get an image that has strange widths/heights.

pdfCreator and *pdfTitle* may be used to set the according metadata fields in the resulting PDF file.

Note

On Android systems, this method does nothing and issues an according qDebug warning message. This is also the case if for other reasons the define flag QT_N-O_PRINTER is set.

See also

savePng, saveBmp, saveJpg, saveRastered

7.70.4.59 bool QCustomPlot::savePng (const QString & *fileName*, int *width* = `0`, int *height* = `0`, double *scale* = `1.0`, int *quality* = `-1`)

Saves a PNG image file to *fileName* on disc. The output plot will have the dimensions *width* and *height* in pixels. If either *width* or *height* is zero, the exported image will have the same dimensions as the QCustomPlot widget currently has. Line widths and texts etc. are not scaled up when larger widths/heights are used. If you want that effect, use the *scale* parameter.

For example, if you set both *width* and *height* to 100 and *scale* to 2, you will end up with an image file of size 200∗200 in which all graphical elements are scaled up by factor 2 (line widths, texts, etc.). This scaling is not done by stretching a 100∗100 image, the result will have full 200∗200 pixel resolution.

If you use a high scaling factor, it is recommended to enable antialiasing for all elements via temporarily setting QCustomPlot::setAntialiasedElements to QCP::aeAll as this allows QCustomPlot to place objects with sub-pixel accuracy.

Warning

> If calling this function inside the constructor of the parent of the QCustomPlot widget (i.e. the MainWindow constructor, if QCustomPlot is inside the MainWindow), always provide explicit non-zero widths and heights. If you leave *width* or *height* as 0 (default), this function uses the current width and height of the QCustomPlot widget. However, in Qt, these aren't defined yet inside the constructor, so you would get an image that has strange widths/heights.

The objects of the plot will appear in the current selection state. If you don't want any selected objects to be painted in their selected look, deselect everything with deselectAll before calling this function.

If you want the PNG to have a transparent background, call setBackground(const Q-Brush &brush) with no brush (Qt::NoBrush) or a transparent color (Qt::transparent), before saving.

PNG compression can be controlled with the *quality* parameter which must be between 0 and 100 or -1 to use the default setting.

Returns true on success. If this function fails, most likely the PNG format isn't supported by the system, see Qt docs about QImageWriter::supportedImageFormats().

See also

> savePdf, saveBmp, saveJpg, saveRastered

7.70.4.60 bool QCustomPlot::saveRastered (const QString & *fileName*, int *width*, int *height*, double *scale*, const char ∗ *format*, int *quality* = `-1`)

Saves the plot to a rastered image file *fileName* in the image format *format*. The plot is sized to *width* and *height* in pixels and scaled with *scale*. (width 100 and scale 2.0 lead to a full resolution file with width 200.) If the *format* supports compression, *quality* may be between 0 and 100 to control it.

Returns true on success. If this function fails, most likely the given *format* isn't supported by the system, see Qt docs about QImageWriter::supportedImageFormats().

See also

saveBmp, saveJpg, savePng, savePdf

7.70.4.61 QList< QCPAxis * > QCustomPlot::selectedAxes ( ) const

Returns the axes that currently have selected parts, i.e. whose selection state is not QCPAxis::spNone.

See also

selectedPlottables, selectedLegends, setInteractions, QCPAxis::setSelectedParts, QCPAxis::setSelectableParts

7.70.4.62 QList< QCPGraph * > QCustomPlot::selectedGraphs ( ) const

Returns a list of the selected graphs. If no graphs are currently selected, the list is empty.

If you are not only interested in selected graphs but other plottables like QCPCurve, QCPBars, etc., use selectedPlottables.

See also

setInteractions, selectedPlottables, QCPAbstractPlottable::setSelectable, QCP-AbstractPlottable::setSelected

7.70.4.63 QList< QCPAbstractItem * > QCustomPlot::selectedItems ( ) const

Returns a list of the selected items. If no items are currently selected, the list is empty.

See also

setInteractions, QCPAbstractItem::setSelectable, QCPAbstractItem::setSelected

7.70.4.64 QList< QCPLegend * > QCustomPlot::selectedLegends ( ) const

Returns the legends that currently have selected parts, i.e. whose selection state is not QCPLegend::spNone.

See also

selectedPlottables, selectedAxes, setInteractions, QCPLegend::setSelectedParts, QCPLegend::setSelectableParts, QCPLegend::selectedItems

7.70.4.65 QList< QCPAbstractPlottable ∗ > QCustomPlot::selectedPlottables ( ) const

Returns a list of the selected plottables. If no plottables are currently selected, the list is empty.

There is a convenience function if you're only interested in selected graphs, see selectedGraphs.

See also

setInteractions, QCPAbstractPlottable::setSelectable, QCPAbstractPlottable::setSelected

7.70.4.66 void QCustomPlot::selectionChangedByUser ( ) [signal]

This signal is emitted after the user has changed the selection in the QCustomPlot, e.g. by clicking. It is not emitted when the selection state of an object has changed programmatically by a direct call to setSelected() on an object or by calling deselectAll.

In addition to this signal, selectable objects also provide individual signals, for example QCPAxis::selectionChanged or QCPAbstractPlottable::selectionChanged. Note that those signals are emitted even if the selection state is changed programmatically.

See the documentation of setInteractions for details about the selection mechanism.

See also

selectedPlottables, selectedGraphs, selectedItems, selectedAxes, selectedLegends

7.70.4.67 void QCustomPlot::setAntialiasedElement (QCP::AntialiasedElement *antialiasedElement,* bool *enabled* = true)

Sets whether the specified *antialiasedElement* is forcibly drawn antialiased.

See setAntialiasedElements for details.

See also

setNotAntialiasedElement

7.70.4.68 void QCustomPlot::setAntialiasedElements (const QCP::AntialiasedElements & *antialiasedElements*)

Sets which elements are forcibly drawn antialiased as an *or* combination of QCP::AntialiasedElement.

This overrides the antialiasing settings for whole element groups, normally controlled with the *setAntialiasing* function on the individual elements. If an element is neither specified in setAntialiasedElements nor in setNotAntialiasedElements, the antialiasing setting on each individual element instance is used.

For example, if *antialiasedElements* contains QCP::aePlottables, all plottables will be drawn antialiased, no matter what the specific QCPAbstractPlottable::setAntialiased value was set to.

if an element in *antialiasedElements* is already set in setNotAntialiasedElements, it is removed from there.

See also

setNotAntialiasedElements

7.70.4.69 void **QCustomPlot::setAutoAddPlottableToLegend** (bool *on*)

If set to true, adding a plottable (e.g. a graph) to the QCustomPlot automatically also adds the plottable to the legend (QCustomPlot::legend).

See also

addPlottable, addGraph, QCPLegend::addItem

7.70.4.70 void **QCustomPlot::setBackground** (const QPixmap & *pm*)

Sets *pm* as the viewport background pixmap (see setViewport). The pixmap is always drawn below all other objects in the plot.

For cases where the provided pixmap doesn't have the same size as the viewport, scaling can be enabled with setBackgroundScaled and the scaling mode (whether and how the aspect ratio is preserved) can be set with setBackgroundScaledMode. To set all these options in one call, consider using the overloaded version of this function.

If a background brush was set with setBackground(const QBrush &brush), the viewport will first be filled with that brush, before drawing the background pixmap. This can be useful for background pixmaps with translucent areas.

See also

setBackgroundScaled, setBackgroundScaledMode

7.70.4.71 void **QCustomPlot::setBackground** (const QPixmap & *pm,* bool *scaled,* Qt::AspectRatioMode *mode* = `Qt::KeepAspectRatioByExpanding`)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Allows setting the background pixmap of the viewport, whether it shall be scaled and how it shall be scaled in one call.

See also

setBackground(const QPixmap &pm), setBackgroundScaled, setBackgroundScaledMode

7.70.4.72 void QCustomPlot::setBackground (const QBrush & *brush*)

Sets the background brush of the viewport (see setViewport).

Before drawing everything else, the background is filled with *brush*. If a background pixmap was set with setBackground(const QPixmap &pm), this brush will be used to fill the viewport before the background pixmap is drawn. This can be useful for background pixmaps with translucent areas.

Set *brush* to Qt::NoBrush or Qt::Transparent to leave background transparent. This can be useful for exporting to image formats which support transparency, e.g. savePng.

See also

setBackgroundScaled, setBackgroundScaledMode

7.70.4.73 void QCustomPlot::setBackgroundScaled (bool *scaled*)

Sets whether the viewport background pixmap shall be scaled to fit the viewport. If *scaled* is set to true, control whether and how the aspect ratio of the original pixmap is preserved with setBackgroundScaledMode.

Note that the scaled version of the original pixmap is buffered, so there is no performance penalty on replots. (Except when the viewport dimensions are changed continuously.)

See also

setBackground, setBackgroundScaledMode

7.70.4.74 void QCustomPlot::setBackgroundScaledMode (Qt::AspectRatioMode *mode*)

If scaling of the viewport background pixmap is enabled (setBackgroundScaled), use this function to define whether and how the aspect ratio of the original pixmap is preserved.

See also

setBackground, setBackgroundScaled

7.70.4.75 bool QCustomPlot::setCurrentLayer (const QString & *name*)

Sets the layer with the specified *name* to be the current layer. All layerables (QCP-Layerable), e.g. plottables and items, are created on the current layer.

Returns true on success, i.e. if there is a layer with the specified *name* in the QCustom-Plot.

Layer names are case-sensitive.

See also

addLayer, moveLayer, removeLayer, QCPLayerable::setLayer

7.70.4.76 bool QCustomPlot::setCurrentLayer (QCPLayer ∗ *layer*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Sets the provided *layer* to be the current layer.

Returns true on success, i.e. when *layer* is a valid layer in the QCustomPlot.

See also

addLayer, moveLayer, removeLayer

7.70.4.77 void QCustomPlot::setInteraction (const QCP::Interaction & *interaction,* bool *enabled* = `true`)

Sets the single *interaction* of this QCustomPlot to *enabled*.

For details about the interaction system, see setInteractions.

See also

setInteractions

7.70.4.78 void QCustomPlot::setInteractions (const QCP::Interactions & *interactions*)

Sets the possible interactions of this QCustomPlot as an or-combination of QCP::-Interaction enums. There are the following types of interactions:

Axis range manipulation is controlled via QCP::iRangeDrag and QCP::iRangeZoom. When the respective interaction is enabled, the user may drag axes ranges and zoom with the mouse wheel. For details how to control which axes the user may drag/zoom and in what orientations, see QCPAxisRect::setRangeDrag, QCPAxisRect::setRange-Zoom, QCPAxisRect::setRangeDragAxes, QCPAxisRect::setRangeZoomAxes.

Plottable selection is controlled by QCP::iSelectPlottables. If QCP::iSelectPlottables is set, the user may select plottables (graphs, curves, bars,...) by clicking on them or in their vicinity (setSelectionTolerance). Whether the user can actually select a plottable can further be restricted with the QCPAbstractPlottable::setSelectable function on the specific plottable. To find out whether a specific plottable is selected, call QCPAbstractPlottable::selected(). To retrieve a list of all currently selected plottables, call selectedPlottables. If you're only interested in QCPGraphs, you may use the convenience function selectedGraphs.

Item selection is controlled by QCP::iSelectItems. If QCP::iSelectItems is set, the user may select items (QCPItemLine, QCPItemText,...) by clicking on them or in their vicinity. To find out whether a specific item is selected, call QCPAbstractItem::selected(). To retrieve a list of all currently selected items, call selectedItems.

Axis selection is controlled with QCP::iSelectAxes. If QCP::iSelectAxes is set, the user may select parts of the axes by clicking on them. What parts exactly (e.g. Axis base line, tick labels, axis label) are selectable can be controlled via QCPAxis::setSelectableParts for each axis. To retrieve a list of all axes that currently contain selected parts, call selectedAxes. Which parts of an axis are selected, can be retrieved with QCPAxis::selectedParts().

Legend selection is controlled with QCP::iSelectLegend. If this is set, the user may select the legend itself or individual items by clicking on them. What parts exactly are selectable can be controlled via QCPLegend::setSelectableParts. To find out whether the legend or any of its child items are selected, check the value of QCPLegend::selectedParts. To find out which child items are selected, call QCPLegend::selectedItems.

All other selectable elements The selection of all other selectable objects (e.g. QCPPlotTitle, or your own layerable subclasses) is controlled with QCP::iSelectOther. If set, the user may select those objects by clicking on them. To find out which are currently selected, you need to check their selected state explicitly.

If the selection state has changed by user interaction, the selectionChangedByUser signal is emitted. Each selectable object additionally emits an individual selectionChanged signal whenever their selection state has changed, i.e. not only by user interaction.

To allow multiple objects to be selected by holding the selection modifier (setMultiSelectModifier), set the flag QCP::iMultiSelect.

Note

In addition to the selection mechanism presented here, QCustomPlot always emits corresponding signals, when an object is clicked or double clicked. see plottableClick and plottableDoubleClick for example.

See also

setInteraction, setSelectionTolerance

7.70.4.79 void QCustomPlot::setMultiSelectModifier (Qt::KeyboardModifier *modifier*)

Sets the keyboard modifier that will be recognized as multi-select-modifier.

If QCP::iMultiSelect is specified in setInteractions, the user may select multiple objects by clicking on them one after the other while holding down *modifier*.

By default the multi-select-modifier is set to Qt::ControlModifier.

See also

setInteractions

7.70.4.80 void **QCustomPlot::setNoAntialiasingOnDrag** (bool *enabled*)

Sets whether antialiasing is disabled for this QCustomPlot while the user is dragging axes ranges. If many objects, especially plottables, are drawn antialiased, this greatly improves performance during dragging. Thus it creates a more responsive user experience. As soon as the user stops dragging, the last replot is done with normal antialiasing, to restore high image quality.

See also

setAntialiasedElements, setNotAntialiasedElements

7.70.4.81 void **QCustomPlot::setNotAntialiasedElement** (**QCP::AntialiasedElement** *notAntialiasedElement,* bool *enabled* = `true`)

Sets whether the specified *notAntialiasedElement* is forcibly drawn not antialiased.

See setNotAntialiasedElements for details.

See also

setAntialiasedElement

7.70.4.82 void **QCustomPlot::setNotAntialiasedElements** (const QCP::AntialiasedElements & *notAntialiasedElements*)

Sets which elements are forcibly drawn not antialiased as an *or* combination of QCP::-AntialiasedElement.

This overrides the antialiasing settings for whole element groups, normally controlled with the *setAntialiasing* function on the individual elements. If an element is neither specified in setAntialiasedElements nor in setNotAntialiasedElements, the antialiasing setting on each individual element instance is used.

For example, if *notAntialiasedElements* contains QCP::aePlottables, no plottables will be drawn antialiased, no matter what the specific QCPAbstractPlottable::setAntialiased value was set to.

if an element in *notAntialiasedElements* is already set in setAntialiasedElements, it is removed from there.

See also

setAntialiasedElements

7.70.4.83 void QCustomPlot::setPlottingHint (QCP::PlottingHint *hint*, bool *enabled* = `true`)

Sets the specified plotting *hint* to *enabled*.

See also

setPlottingHints

7.70.4.84 void QCustomPlot::setPlottingHints (const QCP::PlottingHints & *hints*)

Sets the plotting hints for this QCustomPlot instance as an *or* combination of QCP::-PlottingHint.

See also

setPlottingHint

7.70.4.85 void QCustomPlot::setSelectionTolerance (int *pixels*)

Sets the tolerance that is used to decide whether a click selects an object (e.g. a plottable) or not.

If the user clicks in the vicinity of the line of e.g. a QCPGraph, it's only regarded as a potential selection when the minimum distance between the click position and the graph line is smaller than *pixels*. Objects that are defined by an area (e.g. QCPBars) only react to clicks directly inside the area and ignore this selection tolerance. In other words, it only has meaning for parts of objects that are too thin to exactly hit with a click and thus need such a tolerance.

See also

setInteractions, QCPLayerable::selectTest

7.70.4.86 void QCustomPlot::setViewport (const QRect & *rect*)

Sets the viewport of this QCustomPlot. The Viewport is the area that the top level layout (QCustomPlot::plotLayout()) uses as its rect. Normally, the viewport is the entire widget rect.

This function is used to allow arbitrary size exports with toPixmap, savePng, savePdf, etc. by temporarily changing the viewport size.

7.70.4.87 void QCustomPlot::titleClick (QMouseEvent ∗ *event*, QCPPlotTitle ∗ *title*) `[signal]`

This signal is emitted when a plot title is clicked.

event is the mouse event that caused the click and *title* is the plot title that received the click.

See also

titleDoubleClick

7.70.4.88 void QCustomPlot::titleDoubleClick (QMouseEvent ∗ *event*, QCPPlotTitle ∗ *title*) [signal]

This signal is emitted when a plot title is double clicked.

event is the mouse event that caused the click and *title* is the plot title that received the click.

See also

titleClick

7.70.4.89 void QCustomPlot::toPainter (QCPPainter ∗ *painter*, int *width* = 0, int *height* = 0)

Renders the plot using the passed *painter*.

The plot is sized to *width* and *height* in pixels. If the *painter's* scale is not 1.0, the resulting plot will appear scaled accordingly.

Note

If you are restricted to using a QPainter (instead of QCPPainter), create a temporary QPicture and open a QCPPainter on it. Then call toPainter with this QCPPainter. After ending the paint operation on the picture, draw it with the QPainter. This will reproduce the painter actions the QCPPainter took, with a QPainter.

See also

toPixmap

7.70.4.90 QPixmap QCustomPlot::toPixmap (int *width* = 0, int *height* = 0, double *scale* = 1.0)

Renders the plot to a pixmap and returns it.

The plot is sized to *width* and *height* in pixels and scaled with *scale*. (width 100 and scale 2.0 lead to a full resolution pixmap with width 200.)

See also

toPainter, saveRastered, saveBmp, savePng, saveJpg, savePdf

7.70.4.91 QRect QCustomPlot::viewport ( ) const [inline]

Returns the viewport rect of this QCustomPlot instance. The viewport is the area the plot is drawn in, all mechanisms, e.g. margin caluclation take the viewport to be the outer border of the plot. The viewport normally is the rect() of the QCustomPlot widget, i.e. a rect with top left (0, 0) and size of the QCustomPlot widget.

Don't confuse the viewport with the axis rect (QCustomPlot::axisRect). An axis rect is typically an area enclosed by four axes, where the graphs/plottables are drawn in. - The viewport is larger and contains also the axes themselves, their tick numbers, their labels, the plot title etc.

Only when saving to a file (see savePng, savePdf etc.) the viewport is temporarily modified to allow saving plots with sizes independent of the current widget size.

7.70.5 Member Data Documentation

7.70.5.1 QCPLegend ∗ QCustomPlot::legend

A pointer to the default legend of the main axis rect. The legend is invisible by default. Use QCPLegend::setVisible to change this.

QCustomPlot offers convenient pointers to the axes (xAxis, yAxis, xAxis2, yAxis2) and the legend. They make it very easy working with plots that only have a single axis rect and at most one axis at each axis rect side. If you use the layout system to add multiple legends to the plot, use the layout system interface to access the new legend. For example, legends can be placed inside an axis rect's inset layout, and must then also be accessed via the inset layout. If the default legend is removed due to manipulation of the layout system (e.g. by removing the main axis rect), the corresponding pointer becomes 0.

7.70.5.2 QCPAxis ∗ QCustomPlot::xAxis

A pointer to the primary x Axis (bottom) of the main axis rect of the plot.

QCustomPlot offers convenient pointers to the axes (xAxis, yAxis, xAxis2, yAxis2) and the legend. They make it very easy working with plots that only have a single axis rect and at most one axis at each axis rect side. If you use the layout system to add multiple axis rects or multiple axes to one side, use the QCPAxisRect::axis interface to access the new axes. If one of the four default axes or the default legend is removed due to manipulation of the layout system (e.g. by removing the main axis rect), the corresponding pointers become 0.

7.70.5.3 QCPAxis ∗ QCustomPlot::xAxis2

A pointer to the secondary x Axis (top) of the main axis rect of the plot. Secondary axes are invisible by default. Use QCPAxis::setVisible to change this (or use QCPAxisRect::setupFullAxesBox).

QCustomPlot offers convenient pointers to the axes (xAxis, yAxis, xAxis2, yAxis2) and the legend. They make it very easy working with plots that only have a single axis rect and at most one axis at each axis rect side. If you use the layout system to add multiple axis rects or multiple axes to one side, use the QCPAxisRect::axis interface to access the new axes. If one of the four default axes or the default legend is removed due to manipulation of the layout system (e.g. by removing the main axis rect), the corresponding pointers become 0.

7.70.5.4 QCPAxis ∗ QCustomPlot::yAxis

A pointer to the primary y Axis (left) of the main axis rect of the plot.

QCustomPlot offers convenient pointers to the axes (xAxis, yAxis, xAxis2, yAxis2) and the legend. They make it very easy working with plots that only have a single axis rect and at most one axis at each axis rect side. If you use the layout system to add multiple axis rects or multiple axes to one side, use the QCPAxisRect::axis interface to access the new axes. If one of the four default axes or the default legend is removed due to manipulation of the layout system (e.g. by removing the main axis rect), the corresponding pointers become 0.

7.70.5.5 QCPAxis ∗ QCustomPlot::yAxis2

A pointer to the secondary y Axis (right) of the main axis rect of the plot. Secondary axes are invisible by default. Use QCPAxis::setVisible to change this (or use QCPAxisRect::setupFullAxesBox).

QCustomPlot offers convenient pointers to the axes (xAxis, yAxis, xAxis2, yAxis2) and the legend. They make it very easy working with plots that only have a single axis rect and at most one axis at each axis rect side. If you use the layout system to add multiple axis rects or multiple axes to one side, use the QCPAxisRect::axis interface to access the new axes. If one of the four default axes or the default legend is removed due to manipulation of the layout system (e.g. by removing the main axis rect), the corresponding pointers become 0.

The documentation for this class was generated from the following files:

- account/qcustomplot.h
- account/qcustomplot.cpp
- bin/moc_qcustomplot.cpp

7.71 ShaunGamesTest Class Reference

Public Member Functions

- void **setUp** ()
- void **tearDown** ()

Protected Member Functions

- void **toRadiansTest** ()
- void **getRandomSheepNumberTest** ()

The documentation for this class was generated from the following files:

- tests/ShaunGamesTest.h
- tests/ShaunGamesTest.cpp

7.72 Sheep1 Class Reference

Public Slots

- void firedMove ()

  Moves the sheep in the direction of the firing.

Public Member Functions

- Sheep1 (int number, bool inLine, QObject ∗parent=0)

  Constructor.
- virtual ∼Sheep1 ()

  Destructor.
- double getAngle () const

  Gets the angle of the sheep in degrees.
- void setAngle (double angle)

  Sets the angle of the sheep in degrees.
- bool isInLine () const

  Checks whether the sheep is part of the moving line.
- void setInLine (bool inLine)

  Changes the status of the sheep as in or out of the moving line.
- void fire (double angle)

  Fires the sheep in a straight line.
- void moveInLine (double distance)

  Moves the sheep in line by given distance.
- double inLineDistanceTo (const Sheep1 ∗other) const

  Returns the in-line distance between object and given sheep.
- int getNumber () const

  Returns the sheep number.

Static Public Member Functions

- static int getRandomSheepNumber ()

  Returns a number between 1 and 9.

7.72.1 Constructor & Destructor Documentation

7.72.1.1 Sheep1::Sheep1 (int *number,* bool *inLine,* QObject ∗ *parent =* `0`) `[explicit]`

Constructor.

Parameters

number	Sheep number
inLine	Whether sheep is in line

Sets the properties of the sheep.

7.72.1.2 Sheep1::∼Sheep1 ( ) `[virtual]`

Destructor.

Frees allocated memory.

7.72.2 Member Function Documentation

7.72.2.1 void Sheep1::fire (double *angle*)

Fires the sheep in a straight line.

Parameters

angle	Angle at which to fire the sheep

Fires sheep at an angle. Called when the user fires the cannon.

7.72.2.2 void Sheep1::firedMove ( ) `[slot]`

Moves the sheep in the direction of the firing.

Moves the sheep in the distance of the firing of the cannon.

7.72.2.3 double Sheep1::getAngle ( ) const

Gets the angle of the sheep in degrees.

Returns

The angle of the sheep in the circle

7.72.2.4 int Sheep1::getNumber ( ) const

Returns the sheep number.

Returns

Sheep number

7.72.2.5 int Sheep1::getRandomSheepNumber ( ) [static]

Returns a number between 1 and 9.

Returns

Number between 1 and 9

Returns a random sheep number between 1 and 9.

7.72.2.6 double Sheep1::inLineDistanceTo (const Sheep1 * *other*) const

Returns the in-line distance between object and given sheep.

Parameters

other	Other sheep

Returns

Distance in pixels

Calculates the distance between the two in-line sheep.

7.72.2.7 bool Sheep1::isInLine ( ) const

Checks whether the sheep is part of the moving line.

Returns

Whether sheep is in line

7.72.2.8 void Sheep1::moveInLine (double *distance*)

Moves the sheep in line by given distance.

Parameters

distance	Distance by which to move the sheep in pixels

Moves sheep in line by given distance.

7.72.2.9 void Sheep1::setAngle (double *angle*)

Sets the angle of the sheep in degrees.

Parameters

angle	The angle of the sheep in the circle

7.72.2.10 void Sheep1::setInLine (bool *inLine*)

Changes the status of the sheep as in or out of the moving line.

Parameters

inLine	Status of the sheep (inside or outside the line)

The documentation for this class was generated from the following files:

- game1/sheep1.h
- game1/sheep1.cpp

7.73 Sheep2 Class Reference

Public Member Functions

- Sheep2 (Tile ∗tile, QObject ∗parent=0)

  default constructor
- void setCurrent (Tile ∗tile)

  sets the current tile of the sheep
- Tile ∗ getCurrent ()

  gets the current tile of the sheep

7.73.1 Constructor & Destructor Documentation

7.73.1.1 Sheep2::Sheep2 (Tile ∗ *tile*, QObject ∗ *parent* = 0) [explicit]

default constructor

Initializes the sheep position and picture.

7.73.2 Member Function Documentation

7.73.2.1 Tile ∗ Sheep2::getCurrent ( )

gets the current tile of the sheep

Returns

the current tile

Gets the current tile of the sheep.

7.73.2.2 void Sheep2::setCurrent (Tile ∗ *tile*)

sets the current tile of the sheep

Parameters

tile	the tile to be set as current

Unsets current tile and sets the argument tile as current.

The documentation for this class was generated from the following files:

- game2/sheep2.h
- game2/sheep2.cpp

7.74 QCPAxisPainterPrivate::TickLabelData Struct Reference

Public Attributes

- QString **basePart**
- QString **expPart**
- QRect **baseBounds**
- QRect **expBounds**
- QRect **totalBounds**
- QRect **rotatedTotalBounds**
- QFont **baseFont**
- QFont **expFont**

The documentation for this struct was generated from the following file:

- account/qcustomplot.h

7.75 Tile Class Reference

Public Member Functions

- Tile (bool block, int row, int col, QObject ∗parent=0)

  Default constructor.
- void setBlock (bool block)

  set the status of the tile as blocked or not
- bool isBlocked ()

  retrieves the blocked status of the tile
- void mousePressEvent (QGraphicsSceneMouseEvent ∗event)

  what to do on the mouse press event
- virtual ∼Tile ()

  Destrictor.
- void setHasSheep (bool placed)

  sets the status of the tile as having a sheep on it or not
- int getRow ()

  retrieves the row index of the current tile
- int getCol ()

  retrieves the column index of the current tile
- bool isBorder ()

  checks if the current tile is on the border of the grid
- void setVisited (bool visit)

  sets the status of the tile as visited or not
- bool isVisited ()

  checks if the current tile has been visited
- int getDistance ()

  Retrieves the distance to the sheep so far.
- void setDistance (int distance)

  Sets the distance to the sheep.
- Tile ∗ getPrev ()

  Retrieves the previous tile.
- void setPrev (Tile ∗tile)

  Sets the previous tile.

7.75.1 Constructor & Destructor Documentation

7.75.1.1 Tile::Tile (bool *block*, int *row*, int *col*, QObject ∗ *parent* = 0) `[explicit]`

Default constructor.

Sets the block status, scale and initializes indices.

7.75.1.2 Tile::∼Tile ( ) [virtual]

Destrictor.

Frees allocated memory.

7.75.2 Member Function Documentation

7.75.2.1 int Tile::getCol ( )

retrieves the column index of the current tile

Returns

the column index

Retrieves the column of the tile.

7.75.2.2 int Tile::getDistance ( )

Retrieves the distance to the sheep so far.

Returns

The distance to the sheep

7.75.2.3 Tile ∗ Tile::getPrev ( )

Retrieves the previous tile.

Returns

The previous tile

7.75.2.4 int Tile::getRow ( )

retrieves the row index of the current tile

Returns

the row index

Retrieves the row of the tile.

7.75.2.5 bool Tile::isBlocked ( )

retrieves the blocked status of the tile

Returns

whether or not the tile is blocked

Retrieves the blocked status.

7.75.2.6 bool Tile::isBorder ( )

checks if the current tile is on the border of the grid

Returns

the border status of the tile

Checks if the column is a border.

7.75.2.7 bool Tile::isVisited ( )

checks if the current tile has been visited

Returns

the visited status of the tile

Returns the visited status of the tile.

7.75.2.8 void Tile::mousePressEvent (QGraphicsSceneMouseEvent ∗ *event*)

what to do on the mouse press event

Parameters

event	the mouse press event

On click, places a block on the tile and checks for win status, then gives the turn to the computer.

7.75.2.9 void Tile::setBlock (bool *block*)

set the status of the tile as blocked or not

Parameters

block	decides if the tile is blocked or not

Marks tile as selected and adds border to it.

7.75.2.10 void **Tile::setDistance** (int *distance*)

Sets the distance to the sheep.

Parameters

The	distance to the sheep

7.75.2.11 void **Tile::setHasSheep** (bool *placed*)

sets the status of the tile as having a sheep on it or not

Parameters

placed	boolean status of having a sheep placed on it or not

Sets the tile as having a sheep or not.

7.75.2.12 void **Tile::setPrev** (**Tile** ∗ *tile*)

Sets the previous tile.

Parameters

tile	The tile to be set as previous

7.75.2.13 void **Tile::setVisited** (bool *visit*)

sets the status of the tile as visited or not

Parameters

visit	visited status of the tile

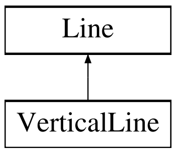
Sets the visited status of the tile.

The documentation for this class was generated from the following files:

- game2/tile.h
- game2/tile.cpp

7.76 VerticalLine Class Reference

Inheritance diagram for VerticalLine:



Public Member Functions

- VerticalLine (Box ∗left, Box ∗right, QObject ∗parent=0)

  Default constructor.
- virtual ∼VerticalLine ()

  Destructor.
- void turnGrey ()

  Makes the line grey.
- bool playTurn (bool userTurn)

  Called when a line is selected.
- Box ∗ getLeft () const

  Gets the box to the left of the line.
- Box ∗ getRight () const

  Gets the box to the right of the line.
- void simpleDraw ()

  Draws the line as grey and edits its corresponding boxes.

7.76.1 Constructor & Destructor Documentation

7.76.1.1 VerticalLine::VerticalLine (Box ∗ *left,* Box ∗ *right,* QObject ∗ *parent* = 0) [explicit]

Default constructor.

Parameters

left	Box on left of line
right	Box on right of line

Sets VerticalLine properties.

7.76.1.2 VerticalLine::∼VerticalLine ( ) [virtual]

Destructor.

Frees allocated memory.

7.76.2 Member Function Documentation

7.76.2.1 Box ∗ VerticalLine::getLeft ( ) const

Gets the box to the left of the line.

Returns

box to the left of the line

Returns the box to the left of the line.

7.76.2.2 Box ∗ VerticalLine::getRight ( ) const

Gets the box to the right of the line.

Returns

box to the right of the line

Returns the box to the right of the line.

7.76.2.3 bool VerticalLine::playTurn (bool *userTurn*) [virtual]

Called when a line is selected.

Parameters

userTurn	Whether it is the user's turn

Returns

Whether it is still the player's turn

Called when a line is drawn. Returns whether it is still the same player's turn.

Implements Line.

7.76.2.4 void VerticalLine::turnGrey ( ) [virtual]

Makes the line grey.

Changes the object image to make it grey.

Implements Line.

The documentation for this class was generated from the following files:

- game3/verticalline.h
- game3/verticalline.cpp

Chapter 8

File Documentation

8.1 account/myaccount.cpp File Reference

Contains MyAccount class definition.

`#include "myaccount.h" #include "helper.h" #include "gui/gameselection.-`
`h" #include <QSqlQuery>`

8.1.1 Detailed Description

Contains MyAccount class definition.

8.2 account/myaccount.h File Reference

Class representing the my account and performance history windows.

`#include <QtGui> #include "account/qcustomplot.h"`

Classes

- class MyAccount

8.2.1 Detailed Description

Class representing the my account and performance history windows. Menu that presents to the signed in user his history and performance statistics

Author

Rita Aoun
Rawan Moukalled

8.3 account/qcustomplot.cpp File Reference

```
#include "qcustomplot.h"
```

8.3.1 Detailed Description

8.4 account/qcustomplot.h File Reference

```
#include <QObject> #include <QPointer> #include <QWidget> ×
#include <QPainter> #include <QPaintEvent> #include <-
QMouseEvent> #include <QPixmap> #include <QVector> ×
#include <QString> #include <QDateTime> #include <QMulti-
Map> #include <QFlags> #include <QDebug> #include <Q-
Vector2D> #include <QStack> #include <QCache> #include <-
QMargins> #include <qmath.h> #include <limits> #include
<QtNumeric> #include <QtPrintSupport/QtPrintSupport>
```

Classes

- class QCPScatterStyle

  Represents the visual appearance of scatter points.
- class QCPPainter

  QPainter subclass used internally.
- class QCPLayer

  A layer that may contain objects, to control the rendering order.
- class QCPLayerable

  Base class for all drawable objects.
- class QCPRange

  Represents the range an axis is encompassing.
- class QCPMarginGroup

  A margin group allows synchronization of margin sides if working with multiple layout elements.
- class QCPLayoutElement

  The abstract base class for all objects that form the layout system.
- class QCPLayout

  The abstract base class for layouts.
- class QCPLayoutGrid

  A layout that arranges child elements in a grid.
- class QCPLayoutInset

  A layout that places child elements aligned to the border or arbitrarily positioned.
- class QCPLineEnding

  Handles the different ending decorations for line-like items.
- class QCPGrid

Responsible for drawing the grid of a QCPAxis.

- class QCPAxis

  Manages a single axis inside a QCustomPlot.
- class QCPAxisPainterPrivate
- struct QCPAxisPainterPrivate::CachedLabel
- struct QCPAxisPainterPrivate::TickLabelData
- class QCPAbstractPlottable

  The abstract base class for all data representing objects in a plot.
- class QCPItemAnchor

  An anchor of an item to which positions can be attached to.
- class QCPItemPosition

  Manages the position of an item.
- class QCPAbstractItem

  The abstract base class for all items in a plot.
- class QCustomPlot

  The central class of the library. This is the QWidget which displays the plot and interacts with the user.
- class QCPColorGradient

  Defines a color gradient for use with e.g. QCPColorMap.
- class QCPAxisRect

  Holds multiple axes and arranges them in a rectangular shape.
- class QCPAbstractLegendItem

  The abstract base class for all entries in a QCPLegend.
- class QCPPlottableLegendItem

  A legend item representing a plottable with an icon and the plottable name.
- class QCPLegend

  Manages a legend inside a QCustomPlot.
- class QCPPlotTitle

  A layout element displaying a plot title text.
- class QCPColorScaleAxisRectPrivate
- class QCPColorScale

  A color scale for use with color coding data such as QCPColorMap.
- class QCPData

  Holds the data of one single data point for QCPGraph.
- class QCPGraph

  A plottable representing a graph in a plot.
- class QCPCurveData

  Holds the data of one single data point for QCPCurve.
- class QCPCurve

  A plottable representing a parametric curve in a plot.
- class QCPBarsGroup

  Groups multiple QCPBars together so they appear side by side.
- class QCPBarData

  Holds the data of one single data point (one bar) for QCPBars.

- class QCPBars

  A plottable representing a bar chart in a plot.
- class QCPStatisticalBox

  A plottable representing a single statistical box in a plot.
- class QCPColorMapData

  Holds the two-dimensional data of a QCPColorMap plottable.
- class QCPColorMap

  A plottable representing a two-dimensional color map in a plot.
- class QCPFinancialData

  Holds the data of one single data point for QCPFinancial.
- class QCPFinancial

  A plottable representing a financial stock chart.
- class QCPItemStraightLine

  A straight line that spans infinitely in both directions.
- class QCPItemLine

  A line from one point to another.
- class QCPItemCurve

  A curved line from one point to another.
- class QCPItemRect

  A rectangle.
- class QCPItemText

  A text label.
- class QCPItemEllipse

  An ellipse.
- class QCPItemPixmap

  An arbitrary pixmap.
- class QCPItemTracer

  Item that sticks to QCPGraph data points.
- class QCPItemBracket

  A bracket for referencing/highlighting certain parts in the plot.

Namespaces

- namespace QCP

Typedefs

- typedef QMap< double, QCPData > QCPDataMap
- typedef QMapIterator< double, QCPData > **QCPDataMapIterator**
- typedef QMutableMapIterator < double, QCPData > **QCPDataMutableMap-Iterator**
- typedef QMap< double, QCPCurveData > QCPCurveDataMap
- typedef QMapIterator< double, QCPCurveData > **QCPCurveDataMapIterator**

- typedef QMutableMapIterator < double, QCPCurveData > **QCPCurveDataMutableMapIterator**
- typedef QMap< double, QCPBarData > QCPBarDataMap
- typedef QMapIterator< double, QCPBarData > **QCPBarDataMapIterator**
- typedef QMutableMapIterator < double, QCPBarData > **QCPBarDataMutableMapIterator**
- typedef QMap< double, QCPFinancialData > QCPFinancialDataMap
- typedef QMapIterator< double, QCPFinancialData > **QCPFinancialDataMapIterator**
- typedef QMutableMapIterator < double, QCPFinancialData > **QCPFinancialDataMutableMapIterator**

Enumerations

- enum QCP::MarginSide { QCP::msLeft = 0x01, QCP::msRight = 0x02, QCP::msTop = 0x04, QCP::msBottom = 0x08, QCP::msAll = 0xFF, QCP::msNone = 0x00 }
- enum QCP::AntialiasedElement { QCP::aeAxes = 0x0001, QCP::aeGrid = 0x0002, QCP::aeSubGrid = 0x0004, QCP::aeLegend = 0x0008, QCP::aeLegendItems = 0x0010, QCP::aePlottables = 0x0020, QCP::aeItems = 0x0040, QCP::aeScatters = 0x0080, QCP::aeErrorBars = 0x0100, QCP::aeFills = 0x0200, QCP::aeZeroLine = 0x0400, QCP::aeAll = 0xFFFF, QCP::aeNone = 0x0000 }
- enum QCP::PlottingHint { QCP::phNone = 0x000, QCP::phFastPolylines = 0x001, QCP::phForceRepaint = 0x002, QCP::phCacheLabels = 0x004 }
- enum QCP::Interaction { QCP::iRangeDrag = 0x001, QCP::iRangeZoom = 0x002, QCP::iMultiSelect = 0x004, QCP::iSelectPlottables = 0x008, QCP::iSelectAxes = 0x010, QCP::iSelectLegend = 0x020, QCP::iSelectItems = 0x040, QCP::iSelectOther = 0x080 }

Functions

- bool **QCP::isInvalidData** (double value)
- bool **QCP::isInvalidData** (double value1, double value2)
- void **QCP::setMarginValue** (QMargins &margins, QCP::MarginSide side, int value)
- int **QCP::getMarginValue** (const QMargins &margins, QCP::MarginSide side)
- **Q_DECLARE_TYPEINFO** (QCPScatterStyle, Q_MOVABLE_TYPE)
- **Q_DECLARE_TYPEINFO** (QCPRange, Q_MOVABLE_TYPE)
- const QCPRange operator+ (const QCPRange &range, double value)
- const QCPRange operator+ (double value, const QCPRange &range)
- const QCPRange operator- (const QCPRange &range, double value)
- const QCPRange operator∗ (const QCPRange &range, double value)
- const QCPRange operator∗ (double value, const QCPRange &range)
- const QCPRange operator/ (const QCPRange &range, double value)
- **Q_DECLARE_TYPEINFO** (QCPLineEnding, Q_MOVABLE_TYPE)

- **Q_DECLARE_TYPEINFO** (QCPData, Q_MOVABLE_TYPE)
- **Q_DECLARE_TYPEINFO** (QCPCurveData, Q_MOVABLE_TYPE)
- **Q_DECLARE_TYPEINFO** (QCPBarData, Q_MOVABLE_TYPE)
- **Q_DECLARE_TYPEINFO** (QCPFinancialData, Q_MOVABLE_TYPE)

8.4.1 Detailed Description

8.4.2 Typedef Documentation

8.4.2.1 QCPBarDataMap

Container for storing QCPBarData items in a sorted fashion. The key of the map is the key member of the QCPBarData instance.

This is the container in which QCPBars holds its data.

See also

QCPBarData, QCPBars::setData

8.4.2.2 QCPCurveDataMap

Container for storing QCPCurveData items in a sorted fashion. The key of the map is the t member of the QCPCurveData instance.

This is the container in which QCPCurve holds its data.

See also

QCPCurveData, QCPCurve::setData

8.4.2.3 QCPDataMap

Container for storing QCPData items in a sorted fashion. The key of the map is the key member of the QCPData instance.

This is the container in which QCPGraph holds its data.

See also

QCPData, QCPGraph::setData

8.4.2.4 QCPFinancialDataMap

Container for storing QCPFinancialData items in a sorted fashion. The key of the map is the key member of the QCPFinancialData instance.

This is the container in which QCPFinancial holds its data.

See also

QCPFinancial, QCPFinancial::setData

8.4.3 Function Documentation

8.4.3.1 const **QCPRange** operator∗ (const **QCPRange** & *range*, double *value*) `[inline]`

Multiplies both boundaries of the range by *value*.

8.4.3.2 const **QCPRange** operator∗ (double *value*, const **QCPRange** & *range*) `[inline]`

Multiplies both boundaries of the range by *value*.

8.4.3.3 const **QCPRange** operator+ (const **QCPRange** & *range*, double *value*) `[inline]`

Adds *value* to both boundaries of the range.

8.4.3.4 const **QCPRange** operator+ (double *value*, const **QCPRange** & *range*) `[inline]`

Adds *value* to both boundaries of the range.

8.4.3.5 const **QCPRange** operator- (const **QCPRange** & *range*, double *value*) `[inline]`

Subtracts *value* from both boundaries of the range.

8.4.3.6 const **QCPRange** operator/ (const **QCPRange** & *range*, double *value*) `[inline]`

Divides both boundaries of the range by *value*.

8.5 difficulty.h File Reference

Difficulty enum.

Enumerations

- enum **Difficulty** { **NO_DIFFICULTY**, **EASY**, **MODERATE**, **HARD**, **DIFFICULTY_END** }

8.5.1 Detailed Description

Difficulty enum. This enum lists the different possible difficulties for games 2 and 3.

Author

Rita Aoun
Rawan Moukalled

8.6 game1/barn.cpp File Reference

Contains Barn class definition.

`#include "barn.h"` `#include "game1scene.h"` `#include <QList>`

8.6.1 Detailed Description

Contains Barn class definition.

8.7 game1/barn.h File Reference

Barn class.

`#include <QtGui>` `#include <QTimer>`

Classes

- class Barn

8.7.1 Detailed Description

Barn class. Barn that terminates the game once a sheep from the line reaches it

Author

Rita Aoun
Rawan Moukalled

8.8 game1/cannon.cpp File Reference

Contains Cannon class definition.

`#include "cannon.h"` `#include "game1scene.h"`

8.8.1 Detailed Description

Contains Cannon class definition.

8.9 game1/cannon.h File Reference

Cannon class.

`#include <QtGui>`

Classes

- class Cannon

8.9.1 Detailed Description

Cannon class. Cannon objects rotate with mouse movements, and fire sheep on click.

Author

Rita Aoun
Rawan Moukalled

8.10 game1/game1.cpp File Reference

Contains the Sheep Line.

`#include "game1/game1.h"` `#include "helper.h"` `#include "gui/gamemainmenu.h"` `#include <QSqlQuery>`

8.10.1 Detailed Description

Contains the Sheep Line.

8.11 game1/game1.h File Reference

Sheep Line class.

`#include <QtGui>` `#include "game1/game1scene.h"`

Classes

- class Game1

8.11.1 Detailed Description

Sheep Line class. This is the class for the gameplay of the Sheep Line game.

Author

Rita Aoun
Rawan Moukalled

8.12 game1/game1options.cpp File Reference

Contains Game1Options class definition.

`#include "game1/game1options.h"` `#include "helper.h"` `#include "gui/gamemainmenu.h"` `#include "game1/game1.h"` `#include <Q-SqlQuery>`

8.12.1 Detailed Description

Contains Game1Options class definition.

8.13 game1/game1options.h File Reference

Game1Options class.

`#include <QtGui>`

Classes

- class Game1Options

8.13.1 Detailed Description

Game1Options class. This is the options page for game 1, where the user can choose the level with which to start the game. Only unlocked levels can be accessed.

Author

Rita Aoun
Rawan Moukalled

8.14 game1/game1scene.cpp File Reference

Contains Game1Scene class definition.

`#include "game1scene.h" #include "helper.h" #include <Q-Vector> #include <QSet> #include "game1options.h" #include <QSqlQuery>`

8.14.1 Detailed Description

Contains Game1Scene class definition.

8.15 game1/game1scene.h File Reference

Sheep Line class.

`#include <QtGui> #include <QLinkedList> #include <Q-Timer> #include "game1/cannon.h" #include "game1/sheep1.-h" #include "game1/barn.h" #include "gameover.h"`

Classes

- class Game1Scene

8.15.1 Detailed Description

Sheep Line class. Implements the scene of Game 1: Sheep Line

Author

Rita Aoun
Rawan Moukalled

8.16 game1/sheep1.cpp File Reference

Contains Sheep1 class definition.

`#include "sheep1.h" #include "helper.h" #include "game1scene.-h" #include <QString>`

8.16.1 Detailed Description

Contains Sheep1 class definition.

8.17 game1/sheep1.h File Reference

Sheep1 class.

```
#include <QtGui> #include <QTimer>
```

Classes

- class Sheep1

8.17.1 Detailed Description

Sheep1 class. Randomly numbered sheep that are used for Game 1: Sheep Line

Author

Rita Aoun
Rawan Moukalled

8.18 game2/game2.cpp File Reference

Contains Game2 class definition.

```
#include "game2/game2.h"   #include "helper.h"   #include
"gui/gamemainmenu.h" #include <QSqlQuery> #include <QSql-
Error>
```

8.18.1 Detailed Description

Contains Game2 class definition.

8.19 game2/game2.h File Reference

Trap the Sheep class.

```
#include <QtGui> #include "difficulty.h" #include "game2/game2scene.-
h"
```

Classes

- class Game2

8.19.1 Detailed Description

Trap the Sheep class. This is the class for the gameplay of the Trap the Sheep game.

Author

Rita Aoun
Rawan Moukalled

8.20 game2/game2options.cpp File Reference

Contains Game2Options class definition.

`#include "game2/game2options.h" #include "helper.h" #include "game2/game2.h" #include "gui/gamemainmenu.h"`

8.20.1 Detailed Description

Contains Game2Options class definition.

8.21 game2/game2options.h File Reference

Game2Options class.

`#include <QtGui> #include "difficulty.h"`

Classes

- class Game2Options

8.21.1 Detailed Description

Game2Options class. This is the options page for game 2, where the user can choose the level with which to start the game. Levels are: Easy, Moderate and Hard.

Author

Rita Aoun
Rawan Moukalled

8.22 game2/game2scene.cpp File Reference

Contains Game2Scene class definition.

`#include "game2scene.h"` `#include <climits>` `#include <Q-SqlQuery>` `#include "helper.h"`

8.22.1 Detailed Description

Contains Game2Scene class definition.

8.23 game2/game2scene.h File Reference

Trap the Sheep scene class.

`#include <QGraphicsScene>` `#include <QtGui>` `#include "difficulty.-h"` `#include "game2/tile.h"` `#include "game2/sheep2.h"` ×
`#include "gameover.h"`

Classes

- class Game2Scene

8.23.1 Detailed Description

Trap the Sheep scene class. This is the scene class for the gameplay of the Trap the Sheep game.

Author

Rita Aoun
Rawan Moukalled

8.24 game2/sheep2.cpp File Reference

Contains Sheep class definition.

`#include "sheep2.h"`

8.24.1 Detailed Description

Contains Sheep class definition.

8.25 game2/tile.cpp File Reference

Contains Tile class definition.

`#include "tile.h" #include "game2scene.h" #include "sheep2.-h"`

8.25.1 Detailed Description

Contains Tile class definition.

8.26 game2/tile.h File Reference

class for the tiles of game 2

`#include <QtGui> #include <QMouseEvent>`

Classes

- class Tile

8.26.1 Detailed Description

class for the tiles of game 2 This is class for the tiles of the grid in game 2

Author

Rita Aoun
Rawan Moukalled

8.27 game3/box.cpp File Reference

Contains Box class definition.

`#include "box.h"`

8.27.1 Detailed Description

Contains Box class definition.

8.28 game3/box.h File Reference

Box class.

```
#include <QtGui>
```

Classes

- class Box

8.28.1 Detailed Description

Box class. Box objects need to be bounded by a player for them to win points.

Author

Rita Aoun
Rawan Moukalled

8.29 game3/dot.cpp File Reference

Contains Dot class definition.

```
#include "dot.h"
```

8.29.1 Detailed Description

Contains Dot class definition.

8.30 game3/dot.h File Reference

Dot class.

```
#include <QtGui>
```

Classes

- class Dot

8.30.1 Detailed Description

Dot class. Dot objects delimit the game lines.

Author

Rita Aoun
Rawan Moukalled

8.31 game3/game3.cpp File Reference

Contains the Dots and Lines game.

`#include "game3/game3.h"` `#include "helper.h"` `#include "gui/gamemainmenu.h"` `#include <QSqlQuery>`

8.31.1 Detailed Description

Contains the Dots and Lines game.

8.32 game3/game3.h File Reference

Dots and Lines class.

`#include <QtGui>` `#include "difficulty.h"` `#include "game3/size.-h"` `#include "game3/game3scene.h"`

Classes

- class Game3

8.32.1 Detailed Description

Dots and Lines class. This is the class for the gameplay of the Dots and Lines game.

Author

Rita Aoun
Rawan Moukalled

8.33 game3/game3options.cpp File Reference

Contains Game3Options class definition.

`#include "game3options.h"` `#include "helper.h"` `#include "game3/game3.h"` `#include "gui/gamemainmenu.h"`

8.33.1 Detailed Description

Contains Game3Options class definition.

8.34 game3/game3options.h File Reference

Game3Options class.

`#include <QtGui> #include "difficulty.h" #include "game3/size.- h"`

Classes

- class Game3Options

8.34.1 Detailed Description

Game3Options class. This is the options page for game 3, where the user can choose the level and size with which to start the game. Levels are: Easy, Moderate and Hard. Sizes are: 4x4, 8x8, 16x16.

Author

Rita Aoun
Rawan Moukalled

8.35 game3/game3scene.cpp File Reference

Contains Game3Scene class definition.

`#include "game3scene.h" #include "helper.h" #include <Q- SqlQuery>`

8.35.1 Detailed Description

Contains Game3Scene class definition.

8.36 game3/game3scene.h File Reference

Game3Scene class.

`#include <QtGui> #include "difficulty.h" #include "game3/size.- h" #include "game3/dot.h" #include "game3/horizontalline.- h" #include "game3/verticalline.h" #include "gameover.h"`

Classes

- class Game3Scene

8.36.1 Detailed Description

Game3Scene class. This is the scene for game 3, Dots and Lines.

Author

Rita Aoun
Rawan Moukalled

8.37 game3/horizontalline.cpp File Reference

Contains HorizontalLine class definition.

```
#include "horizontalline.h"  #include "game3/game3scene.-
h"
```

8.37.1 Detailed Description

Contains HorizontalLine class definition.

8.38 game3/horizontalline.h File Reference

HorizontalLine class.

```
#include "game3/line.h" #include "game3/box.h"
```

Classes

- class HorizontalLine

8.38.1 Detailed Description

HorizontalLine class. Horizontal lines that delimit boxes from the top and bottom.

Author

Rita Aoun
Rawan Moukalled

8.39 game3/line.cpp File Reference

Contains Line class definition.

```
#include "line.h" #include "game3/game3scene.h"
```

8.39.1 Detailed Description

Contains Line class definition.

8.40 game3/line.h File Reference

Line class.

```
#include <QtGui>
```

Classes

- class Line

8.40.1 Detailed Description

Line class. Line is an interface for vertical and horizontal lines. It implements the on-click reaction of lines and it remembers whether a line has been clicked before.

Author

Rita Aoun
Rawan Moukalled

8.41 game3/size.h File Reference

Size enum.

Enumerations

- enum **Size** { **NO_SIZE**, **FOURBYFOUR** = 4, **EIGHTBYEIGHT** = 8, **SIXTEENBYSIXTEEN** = 16, **SIZE_END** }

8.41.1 Detailed Description

Size enum. This enum lists the different possible sizes of the game 3 grid.

Author

Rita Aoun
Rawan Moukalled

8.42 game3/verticalline.cpp File Reference

Contains VerticalLine class definition.

`#include "verticalline.h" #include "game3/game3scene.h"`

8.42.1 Detailed Description

Contains VerticalLine class definition.

8.43 game3/verticalline.h File Reference

VerticalLine class.

`#include <game3/line.h> #include "game3/box.h"`

Classes

- class VerticalLine

8.43.1 Detailed Description

VerticalLine class. Vertical lines that delimit boxes from the left and right.

Author

Rita Aoun
Rawan Moukalled

8.44 gameover.cpp File Reference

Contains GameOver class definition.

`#include "gameover.h"`

8.44.1 Detailed Description

Contains GameOver class definition.

8.45 gameover.h File Reference

Game Over class.

`#include <QtGui>`

Classes

- class GameOver

8.45.1 Detailed Description

Game Over class. Image overlayed on the screen when game is over

Author

Rita Aoun
Rawan Moukalled

8.46 gui/gamemainmenu.cpp File Reference

Contains GameMainMenu class definition.

```
#include "gui/gamemainmenu.h" #include "helper.h" #include
"game1/game1options.h"   #include "game2/game2options.h" ×
#include "game3/game3options.h"   #include "game1/game1.h"
#include "game2/game2.h" #include "game3/game3.h" #include
"gui/gameselection.h"   #include "difficulty.h"   #include
"game3/size.h" #include <QSqlQuery>
```

8.46.1 Detailed Description

Contains GameMainMenu class definition.

8.47 gui/gamemainmenu.h File Reference

GameMainMenu class.

```
#include <QtGui>
```

Classes

- class GameMainMenu

8.47.1 Detailed Description

GameMainMenu class. This is the main game menu, where the user choose between resuming a previous game or starting a new one. The instructions are also shown on this menu.

Author

Rita Aoun
Rawan Moukalled

8.48 gui/gameselection.cpp File Reference

Contains GameSelection class definition.

```
#include "gui/gameselection.h" #include "helper.h" #include
"gui/mainwidget.h" #include "gui/gamemainmenu.h" #include
"account/myaccount.h"
```

8.48.1 Detailed Description

Contains GameSelection class definition.

8.49 gui/gameselection.h File Reference

Game selection menu class.

```
#include <QtGui>
```

Classes

- class GameSelection

8.49.1 Detailed Description

Game selection menu class. Game selection menu, where the user can select one of the three games available.

Author

Rita Aoun
Rawan Moukalled

8.50 gui/mainwidget.cpp File Reference

Contains MainWidget class definition.

```
#include "gui/mainwidget.h" #include "gui/gameselection.-
h" #include "helper.h" #include <QSqlQuery>
```

8.50.1 Detailed Description

Contains MainWidget class definition.

8.51 gui/mainwidget.h File Reference

MainWidget class.

`#include <QtGui>`

Classes

- class MainWidget

8.51.1 Detailed Description

MainWidget class. This is the main sign in window, where the user is given the chance to go on as a guest or to login/sign up.

Author

Rita Aoun
Rawan Moukalled

8.52 helper.cpp File Reference

Contains Helper class definition.

```
#include "helper.h"  #include <ctime>  #include <QTime> ×
#include <QSqlQuery> #include <QSqlError>
```

8.52.1 Detailed Description

Contains Helper class definition.

8.53 helper.h File Reference

Helper class.

`#include <QtGui> #include <QSqlDatabase>`

Classes

- class Helper

8.53.1 Detailed Description

Helper class. This class provides various helper functions that are needed across windows.

Author

Rita Aoun
Rawan Moukalled