GAME PROGRAMMING USING QT 6

BEGINNER'S GUIDE

Create GUI's using Qt Designer

Create Amazing Games With QT 6, C++ AND Qt Quick

Qt GUI Programming

This chapter will help you learn how to use Qt to develop applications with a graphical user interface using the Qt Creator IDE. We will get familiar with the core Qt functionality, widgets, layouts, and the signals and slots mechanism that we will later use to create complex systems such as games. We will also cover the various actions and resource systems of Qt. By the end of this chapter, you will be able to write your own programs that communicate with the user through windows and widgets.

The main topics covered in this chapter are as listed:

- Windows and widgets
- Creating a Qt Widgets project and implementing a tic-tac-toe game
- Creating widgets with or without the visual form editor
- Using layouts to automatically position widgets
- Creating and using signals and slots
- Using the Qt resource system

Creating GUI in Qt

In this chapter, you will learn how to use the Qt Widgets module. It allows you to create classic desktop applications. The **user interface** (**UI**) of these applications consists of *widgets*.

A widget is a fragment of the UI with a specific look and behavior. Qt provides a lot of built-in widgets that are widely used in applications: labels, text boxes, checkboxes, buttons, and so on. Each of these widgets is represented as an instance of a C++ class derived from `QWidget` and provides methods for reading and writing the widget's content. You may also create your own widgets with custom content and behavior.

The base class of `QWidget` is `QObject`—the most important Qt class that contains multiple useful features. In particular, it implements parent–child relationships between objects, allowing you to organize a collection of objects in your program. Each object can have a parent object and an arbitrary number of children. Making a parent–child relationship between two objects has multiple consequences. When an object is deleted, all its children will be automatically deleted as well. For widgets, there is also a rule that a child occupies an area within the boundaries of its parent. For example, a typical form includes multiple labels, input fields, and buttons. Each of the form's elements is a widget, and the form is their parent widget.

Each widget has a separate coordinate system that is used for painting and event handling within the widget. By default, the origin of this coordinate system is placed in its top-left corner. The child's coordinate system is relative to its parent.

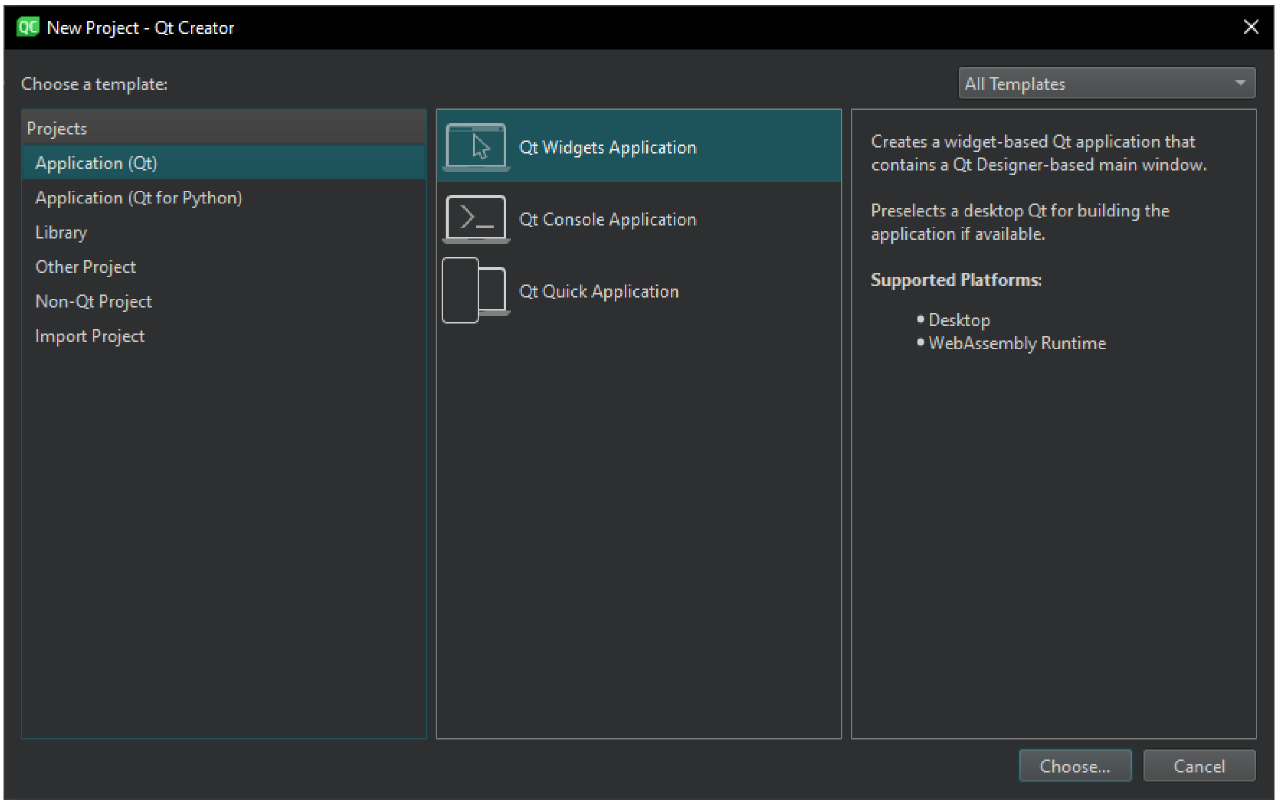
Any widget that is not included into another widget (that is, any *top-level widget*) becomes a window, and the desktop operating system will provide it with a window frame, which usually usually allows the user to drag around, resize, and close the window (although the presence and content of the window frame can be configured).

Time for action – Creating a Qt Widgets project

The first step to develop an application with Qt Creator is to create a project using one of the templates provided by the IDE.

From the File menu of Qt Creator, choose New File or Project. There are a number of project types to choose from. Follow the given steps for creating a Qt Desktop project:

1. For a widget-based application, choose the Application group and the Qt Widgets Application template, as shown in the following screenshot:


New Project - Qt Creator

Choose a template:

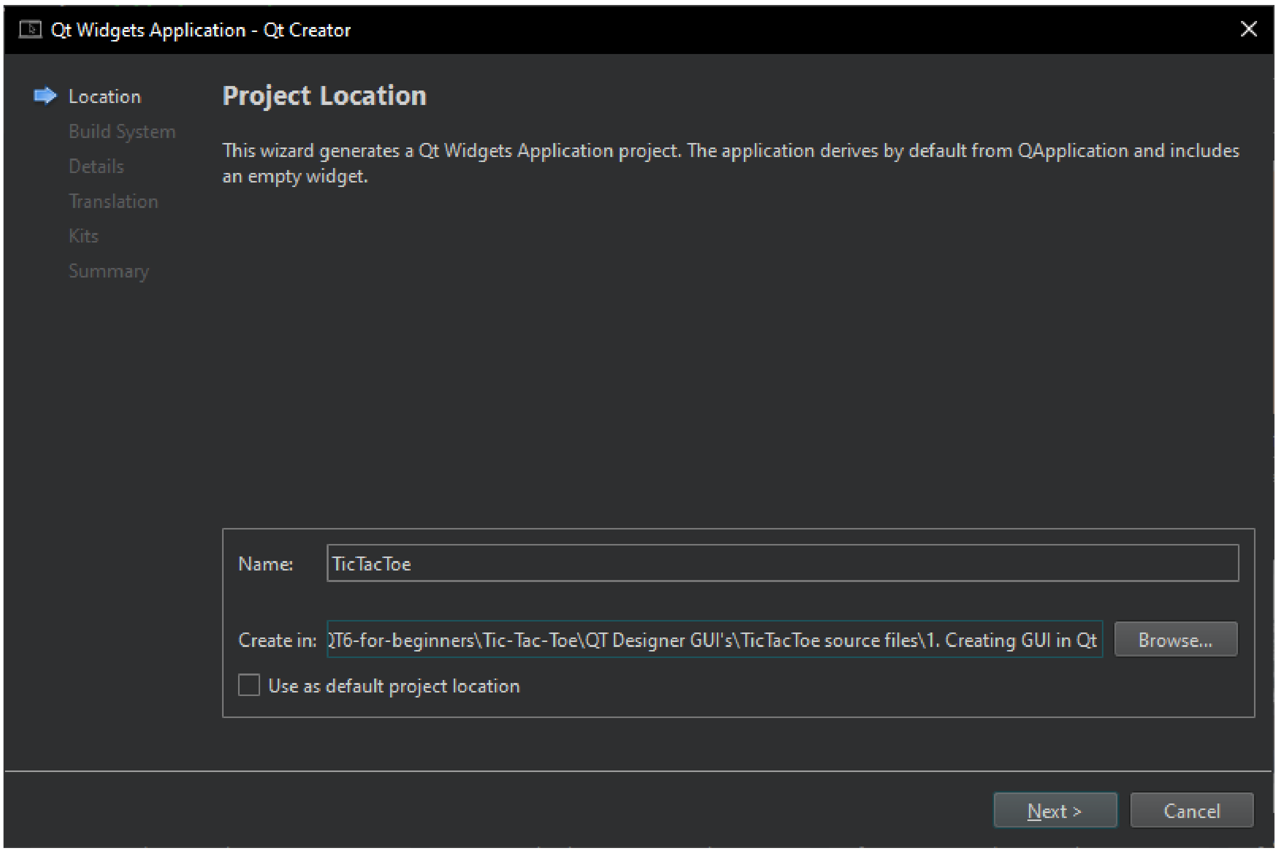
Creates a widget-based Qt application that contains a Qt Designer-based main window.

Preselects a desktop Qt for building the application if available.

Supported Platforms:

- Desktop
- WebAssembly Runtime

2. The next step is to choose a name and location for your new project:


Qt Widgets Application - Qt Creator

Project Location

This wizard generates a Qt Widgets Application project. The application derives by default from QApplication and includes an empty widget.

Name: TicTacToe

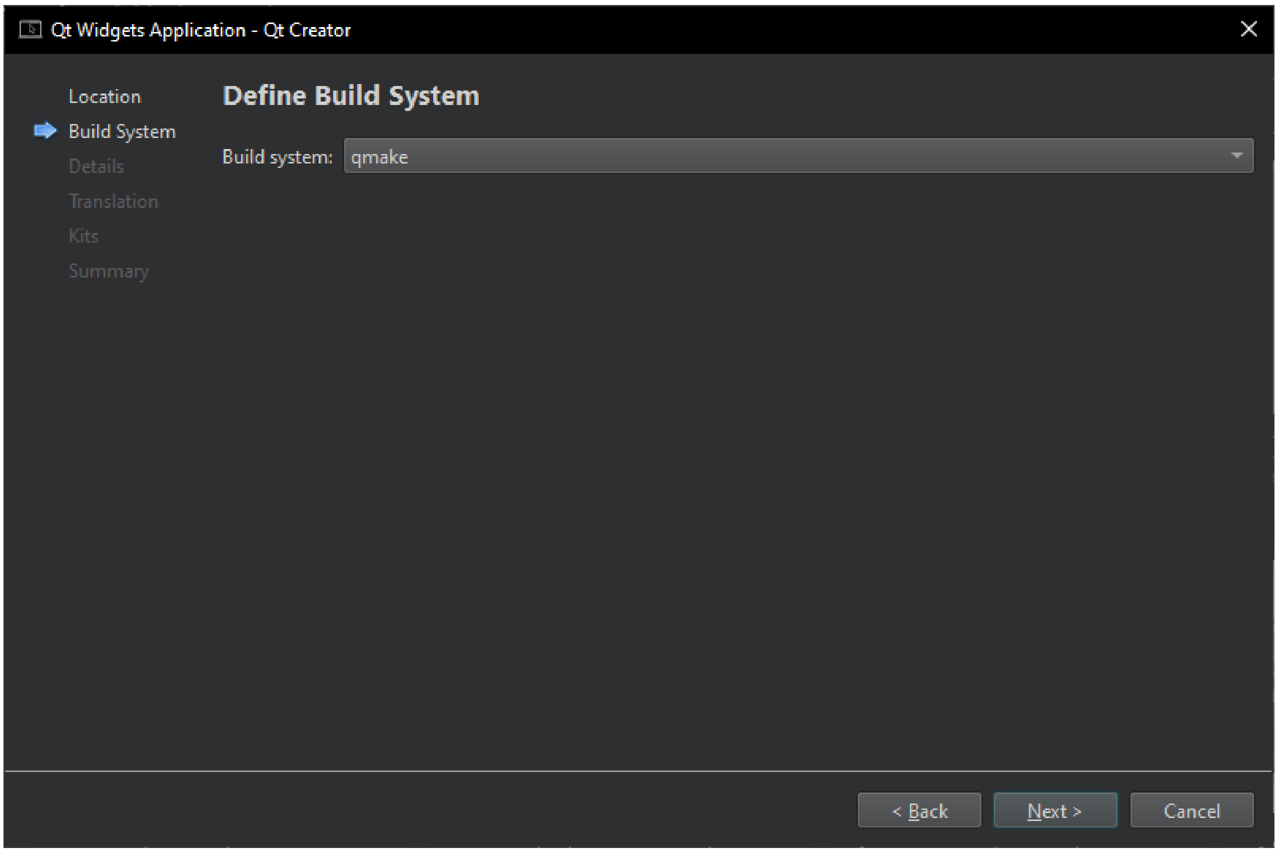
Create in: ...QT6-for-beginners\Tic-Tac-Toe\QT Designer GUI's\TicTacToe source files\1. Creating GUI in Qt

Use as default project location

3. We will create a simple tic-tac-toe game, so we will name our project `tictactoe` and provide a nice location for it, then next.

If you have a common directory where you put all your projects, you can tick the Use as default project location checkbox for Qt Creator to remember the location and suggest it the next time you start a new project.

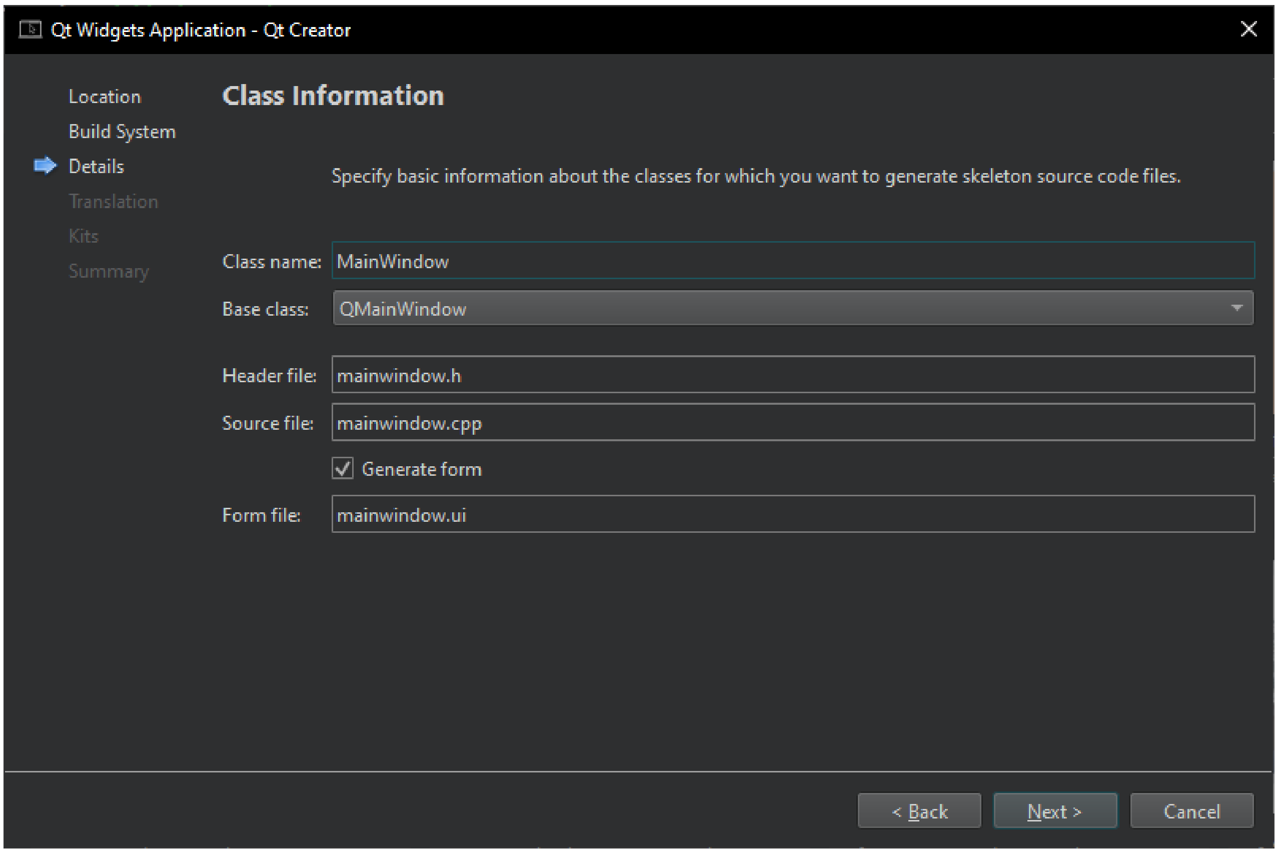
4. Now we choose the build system, we will be using qmake, for our projects. Choose qmake then click next.


Qt Widgets Application - Qt Creator

Define Build System

Build system: qmake

5. Now you will be presented with the option of creating the first widget for your project. We want to create a widget that will represent the main window of our application, so we can leave the Class name and Base class fields unchanged. We also want to use the visual form editor to edit the content of the main window, so Generate form should also be left checked:

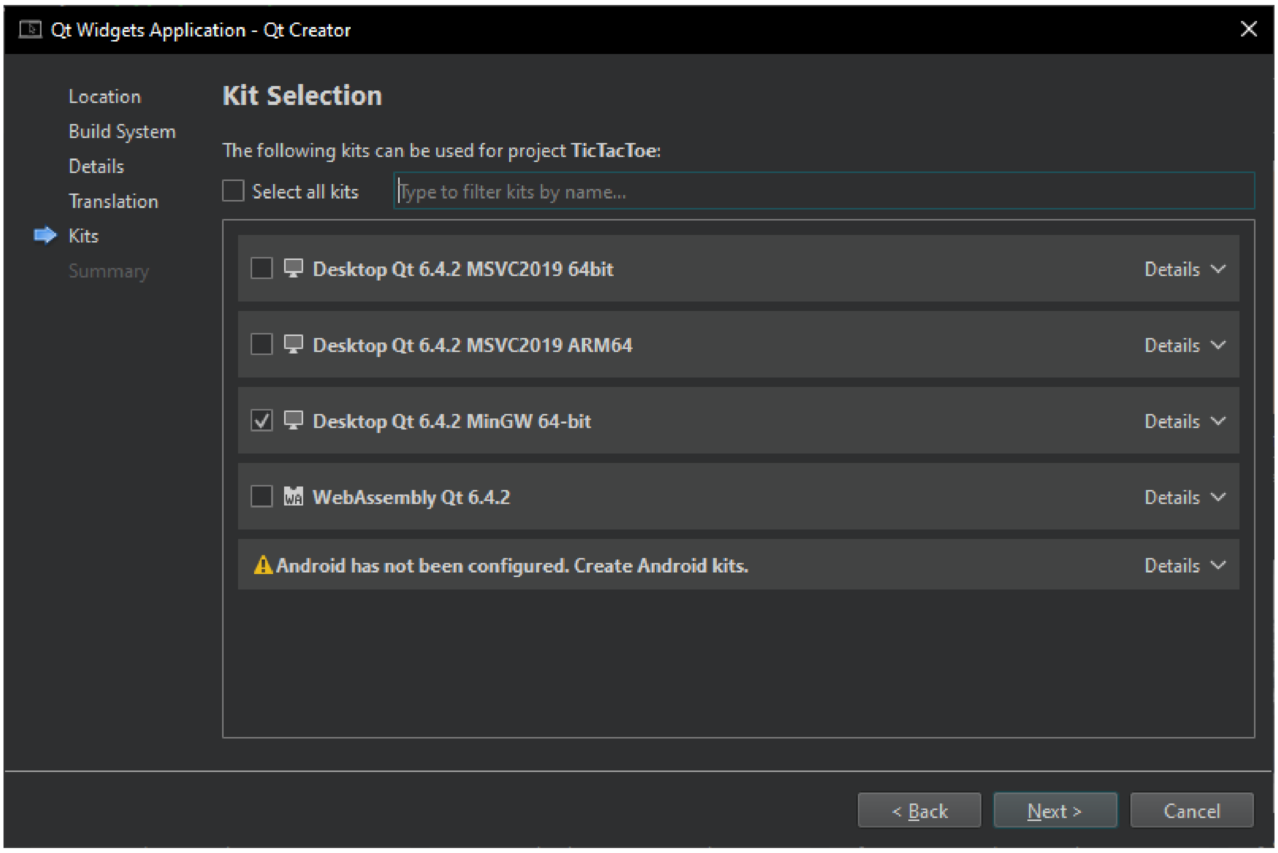

Qt Widgets Application - Qt Creator

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name: MainWindow
Base class: QMainWindow
Header file: mainwindow.h
Source file: mainwindow.cpp
✓ Generate form
Form file: mainwindow.ui

4. Click next until you get to the kits section, you need to select the kit (or multiple kits) you want to use with the project. Select the Desktop Qt kit corresponding to the Qt version you want to use:


Qt Widgets Application - Qt Creator

Location
Build System
Details
Translation
Kits
Summary

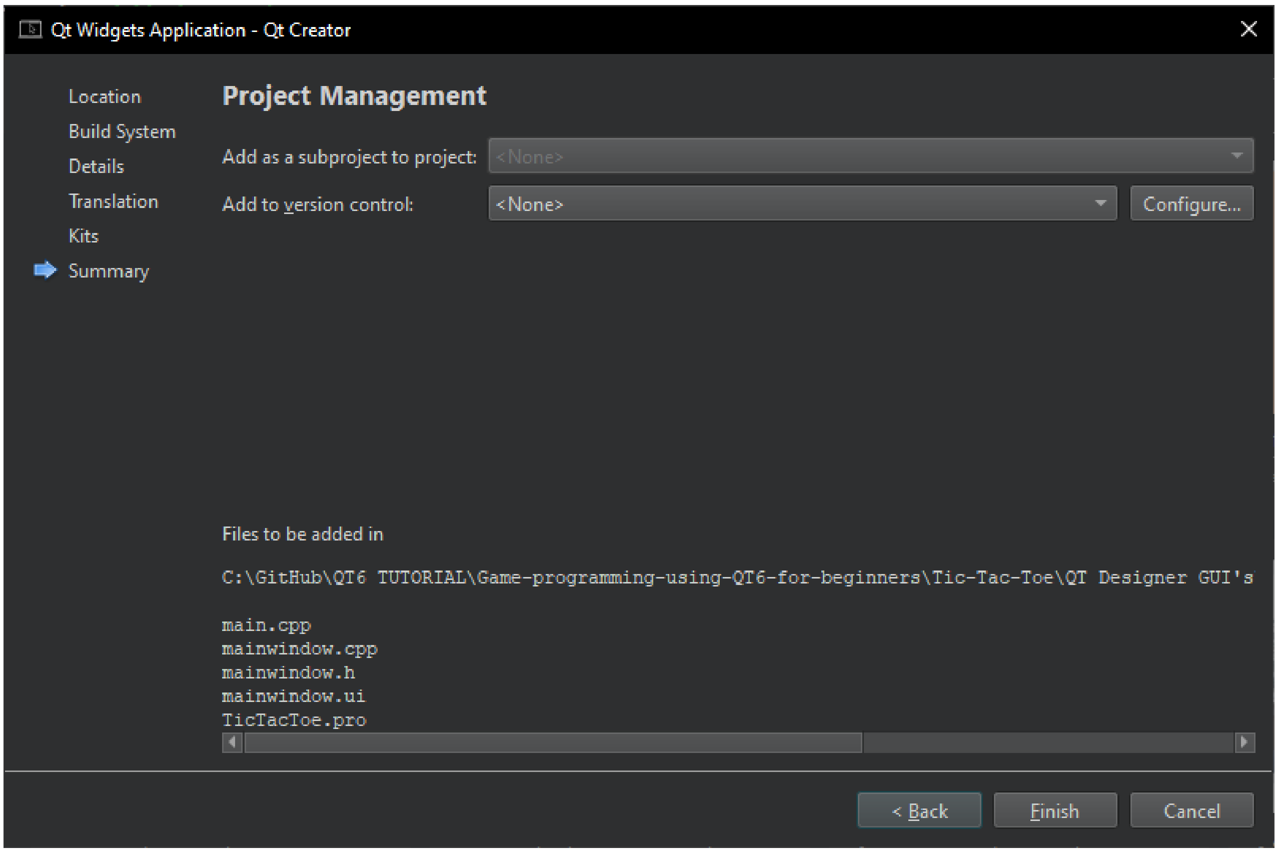
Kit Selection

The following kits can be used for project **TicTacToe**:

- [ ] Select all kits — Type to filter kits by name...
- [ ] Desktop Qt 6.4.2 MSVC2019 64bit — Details
- [ ] Desktop Qt 6.4.2 MSVC2019 ARM64 — Details
- [✓] Desktop Qt 6.4.2 MinGW 64-bit — Details
- [ ] WebAssembly Qt 6.4.2 — Details
- ⚠ Android has not been configured. Create Android kits. — Details

< Back | Next > | Cancel

6. Then, click on Next and Finish.


Qt Widgets Application - Qt Creator

Location
Build System
Details
Translation
Kits
Summary

Project Management

Add as a subproject to project: <None>
Add to version control: <None> Configure...

Files to be added in

```
C:\GitHub\QT6 TUTORIAL\Game-programming-using-QT6-for-beginners\Tic-Tac-Toe\QT Designer GUI's

main.cpp
mainwindow.cpp
mainwindow.h
mainwindow.ui
TicTacToe.pro
```

< Back | Finish | Cancel

What just happened?

Creator created a new subdirectory in the directory that you previously chose for the location of the project. This new directory (the **project directory**) now contains a number of files. You can use the Projects pane of Qt Creator to list and open these files. Let's go through these files.

The `main.cpp` file contains an implementation of the `main()` function, the entry point of the application, as the following code shows:

```
#include "mainwindow.h"
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

The `main()` function creates an instance of the `QApplication` class and feeds it with variables containing the command-line arguments. Then, it instantiates our `MainWindow` class, calls its `show` method, and finally, returns a value returned by the `exec` method of the application object.

`QApplication` is a singleton class that manages the whole application. In particular, it is responsible for processing events that come from within the application or from external sources. For events to be processed, an event loop needs to be running. The loop waits for incoming events and dispatches them to proper routines. Most things in Qt are done through events: input handling, redrawing, receiving data over the network, triggering timers, and so on. This is the reason we say that Qt is an event-oriented framework. Without an active event loop, the event handling would not function properly. The `exec()` call in `QApplication` (or, to be more specific, in its base class—`QCoreApplication`) is responsible for entering the main event loop of the application. The function does not return until your application requests the event loop to be terminated. When that eventually happens, the `main` function returns and your application ends.

The `mainwindow.h` and the `mainwindow.cpp` files implement the `MainWindow` class. For now, there is almost no code in it. The class is derived from `QMainWindow` (which, in turn, is derived from `QWidget`), so it inherits a lot of methods and behavior from its base class. It also contains a `Ui::MainWindow *ui` field, which is initialized in the constructor and deleted in the destructor. The constructor also calls the `ui->setupUi(this);` function.

`Ui::MainWindow` is an *automatically generated* class, so there is no declaration of it in the source code. It will be created in the build directory when the project is built. The purpose of this class is to set up our widget and fill it with content based on changes in the form editor. The automatically generated class is not a `QWidget`. In fact, it contains only two methods: `setupUi`, which performs the initial setup, and `retranslateUi`, which updates visible text when the UI language is changed. All widgets and other objects added in the form editor are available as public fields of the `Ui::MainWindow` class, so we can access them from within the `MainWindow` method as `ui->objectName`.

`mainwindow.ui` is a form file that can be edited in the visual form editor. If you open it in Qt Creator by double-clicking on it in the Projects pane, Qt Creator will switch to the Design mode. If you switch back to the Edit mode, you will see that this file is actually an XML file containing the hierarchy and properties of all objects edited in Design mode. During the building of the project, a special tool called the User Interface Compiler converts this XML file to the implementation of the `Ui::MainWindow` class used in the `MainWindow` class.

Note that you don't need to edit the XML file by hand or edit any code in the `Ui::MainWindow` class. Making changes in the visual editor is enough to apply them to your `MainWindow` class and make the form's objects available to it.

The final file that was generated is called `tictactoe.pro` and is the project configuration file. It contains all the information that is required to build your project using the tools that Qt provides. Let's analyze this file (less important directives are omitted):

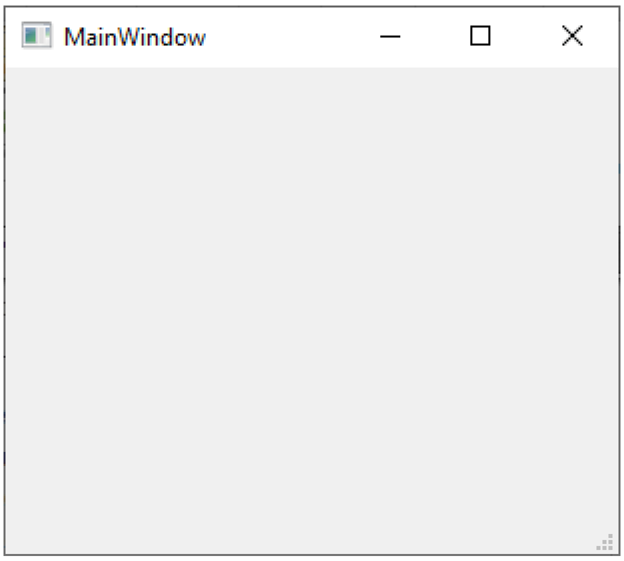
```
QT += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
CONFIG += c++17
SOURCES += \
    main.cpp \
    mainwindow.cpp
HEADERS += \
    mainwindow.h
FORMS   += \
    mainwindow.ui
```

The first two lines enable Qt's `core`, `gui`, and `widgets` modules. The last seven lines list all files that should be used to build the project.

In fact, `qmake` enables Qt Core and Qt GUI modules by default, even if you don't specify them explicitly in the project file. You can opt out of using a default module if you want. For example, you can disable Qt GUI by adding `QT -= gui` to the project file.

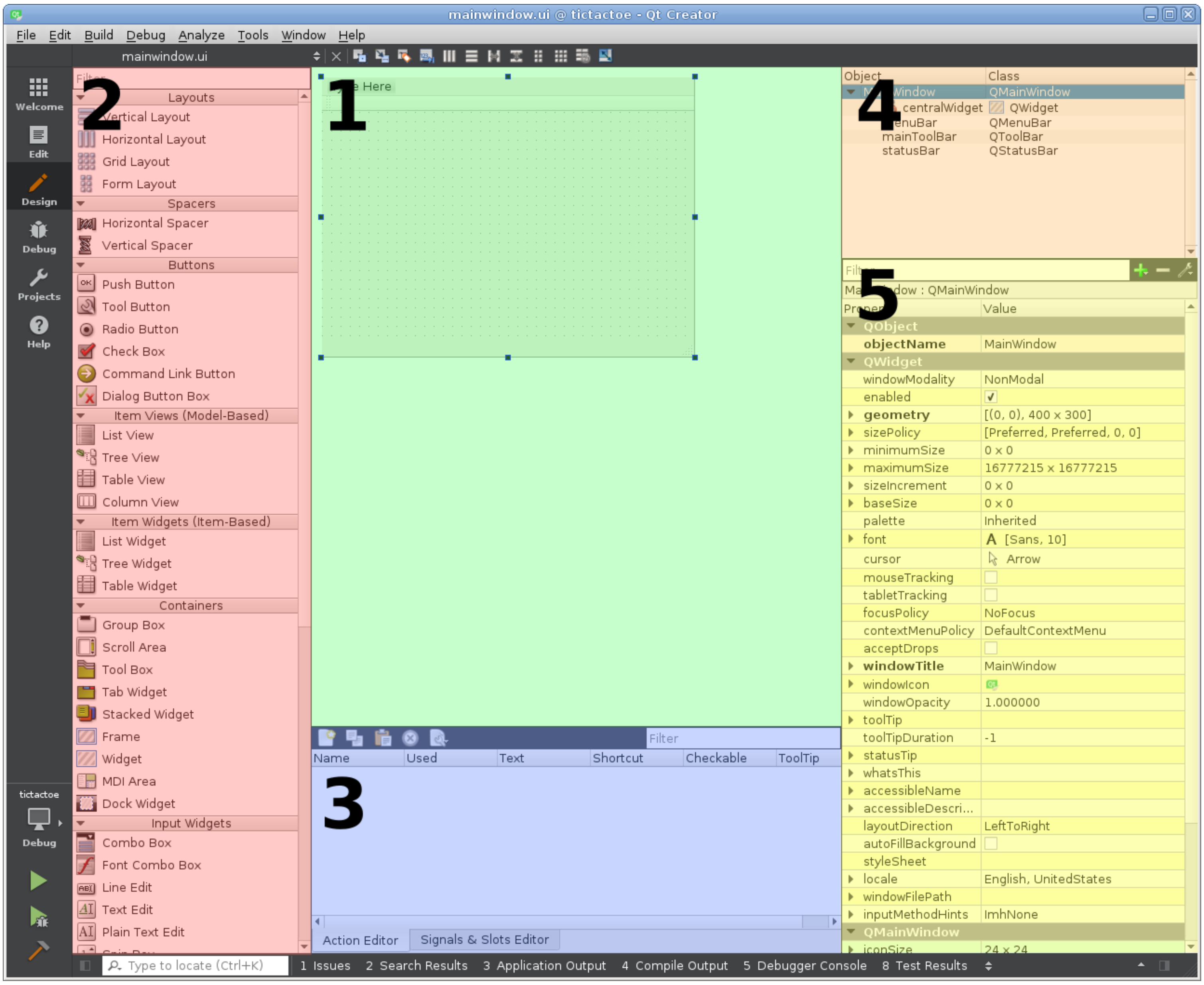
The third line tells the build system that we want to use C++17 features in our project. If it's not there the C++ compiler will receive a flag indicating that C++17 support should be enabled. This may not be needed if your compiler has C++17 support enabled by default, or you wont be using its features. If you wish to use C++14 instead, use `CONFIG += c++14`.

What we have now is a complete Qt Widgets project. To build and run it, simply choose the Run entry from the Build drop-down menu or click on the green triangle icon on the left-hand side of the Qt Creator window. After a while, you should see a window pop up. Since we didn't add anything to the window, it is blank:



Design mode interface

Open the `mainwindow.ui` file and examine Qt Creator's Design mode:



Object	Class
Window	QMainWindow
centralWidget	QWidget
enuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

Ma dow : QMainWindow

Pro	Value
QObject	
objectName	MainWindow
QWidget	
windowModality	NonModal
enabled	✓
geometry	[(0, 0), 400 x 300]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Sans, 10]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
windowTitle	MainWindow
windowIcon	
windowOpacity	1.000000
toolTip	
toolTipDuration	-1
statusTip	
whatsThis	
accessibleName	
accessibleDescri...	
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	English, UnitedStates
windowFilePath	
inputMethodHints	ImhNone
QMainWindow	
iconSize	24 x 24

The Design mode consists of five major parts (they are marked on this screenshot):

- The central area **(1)** is the main worksheet. It contains a graphical representation of the form being designed where you can move widgets around, compose them into layouts, and see how they react. It also allows further manipulation of the form using the point-and-click method that we will learn later.
- The toolbox **(2)** is located in the left part of the window. It contains a list of available types of widget that are arranged into groups containing items with a related or similar functionality. Over the list, you can see a box that lets you filter widgets that are displayed in the list to show only those that match the entered expression. At the beginning of the list, there are also items that are not really widgets—one group contains layouts, and the other one contains so-called spacers, which are a way to push other items away from each other or create an empty space in layouts. The main purpose of the toolbox is to add items to the form in the worksheet. You can do that by grabbing a widget from the list with the mouse, dragging it to the widget in the central area, and releasing the mouse button.
- The two tabs **(3)** in the lower part of the window—Action Editor and Signal/Slot Editor—allow us to create helper entities such as actions for the menus and toolbars or signal-slot connections between widgets.
- The object tree **(4)** is situated in the top-right corner and contains the hierarchy tree of the form's items. The object name and class name of each item added to the form is displayed in the tree. The topmost item corresponds to the form itself. You can use both the central area and the object tree to select the existing items and access their context menu (for example, if you want to delete an item, you can select the Remove... option in the context menu).
- The property editor **(5)** is located in the bottom-right corner. It allows you to view and change the values of all the properties of the item currently selected in the central area and the object tree. Properties are grouped by their classes that they have been declared in, starting from `QObject` (the base class implementing properties), which declares only one, but an important,

property—`objectName`. Following `QObject`, there are properties declared in `QWidget`, which is a direct descendant of `QObject`. They are mainly related to the geometry and layout policies of the widget. Further down the list, you can find properties that come from further derivations of `QWidget`, down to the concrete class of the selected widget. The Filter field above the properties can help you find the needed property quickly.

Taking a closer look at the property editor, you can see that some of them have ▸ arrows, which reveal new rows when clicked. These are composed properties where the complete property value is determined from more than one subproperty value; for example, if there is a property called `geometry` that defines a rectangle, it can be expanded to show four subproperties: `x`, `y`, `width`, and `height`. Another thing that you may quickly note is that some property names are displayed in bold. This means that the property value was modified and is different from the default value for this property. This lets you quickly find the properties that you have modified.

If you changed a property's value but decided to stick to the default value later, you should click on the corresponding input field and then click on the small button with an arrow to its right: ↶. This is not the same as setting the original value by hand. For example, if you examine the `spacing` property of some layouts, it would appear as if it had some constant default value for (example, 6). However, the actual default value depends on the style the application uses and may be different on a different operating system, so the only way to set the default value is to use the dedicated button and ensure that the property is not displayed in bold anymore.

If you prefer a purely alphabetical order where properties are not grouped by their class, you can switch the view using a pop-up menu that becomes available after you click on the wrench icon positioned over the property list; however, once you get familiar with the hierarchy of Qt classes, it will be much easier to navigate the list when it is sorted by class affinity.

What was described here is the basic tool layout. If you don't like it, you can invoke the context menu from the main worksheet, uncheck the Automatically Hide View Title Bars entry, and use the title bars that appear to re-arrange all the panes to your liking, or even close the ones you don't currently need.

Now that you are familiar with the structure of the visual form editor, you can finally add some content to our widget. We are making a tic-tac-toe game with local multiplayer, so we need some way of displaying which of the two players currently moves. Let's put the game board in the center of the window and display the names of the players above and below the board. When a player needs to move, we will make the corresponding name's font bold. We also need a button that will start a new game.

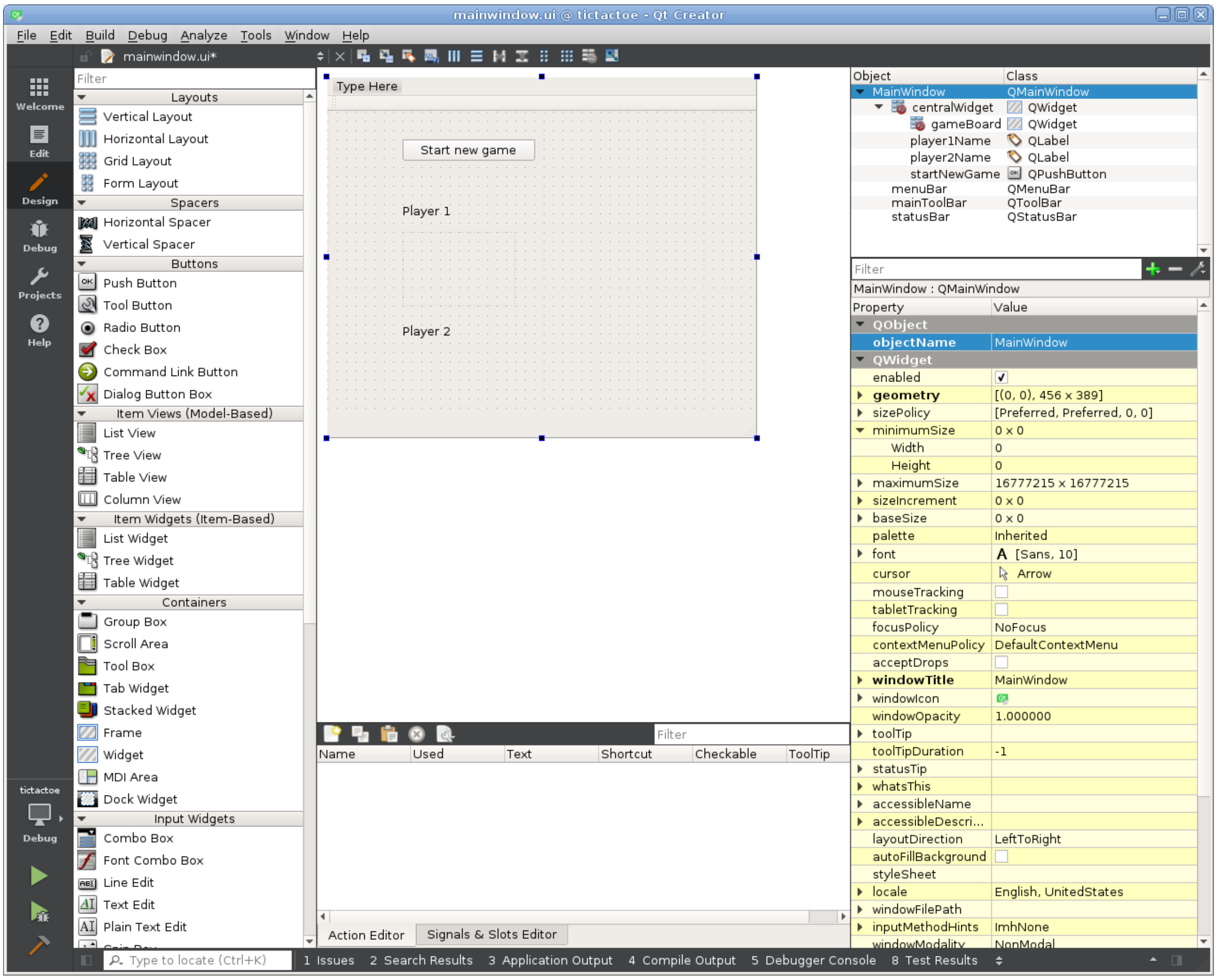
Time for action – Adding widgets to the form

Locate the Label item in the toolbox (it's in the Display Widgets category) and drag it to our form. Use the property editor to set the `objectName` property of the label to `player1Name`. `objectName` is a unique identifier of a form item. The object name is used as the name of the public field in the `Ui::MainWindow` class, so the label will be available as `ui->player1Name` in the `MainWindow` class (and will have a `QLabel *` type). Then, locate the `text` property in the property editor (it will be in the `QLabel` group, as it is the class that introduces the property) and set it to `Player 1`. You will see that the text in the central area will be updated accordingly. Add another label, set its `objectName` to `player2Name` and its `text` to `Player 2`.

You can select a widget in the central area and press the F2 key to edit the text in place. Another way is to double-click on the widget in the form. It works for any widget that can display text.

Drag a Push Button (from the Buttons group) to the form and use the *F2* key to rename it to `Start new game`. If the name does not fit in the button, you can resize it using the blue rectangles on its edges. Set the `objectName` of the button to `startNewGame`.

There is no built-in widget for our game board, so we will need to create a custom widget for it later. For now, we will use an empty widget. Locate Widget in the Containers group of the toolbox and drag it to the form inbetween Player1 and Player2 text labes. Set its `objectName` to `gameBoard`:



Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
gameBoard	QWidget
player1Name	QLabel
player2Name	QLabel
startNewGame	QPushButton
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

MainWindow : QMainWindow

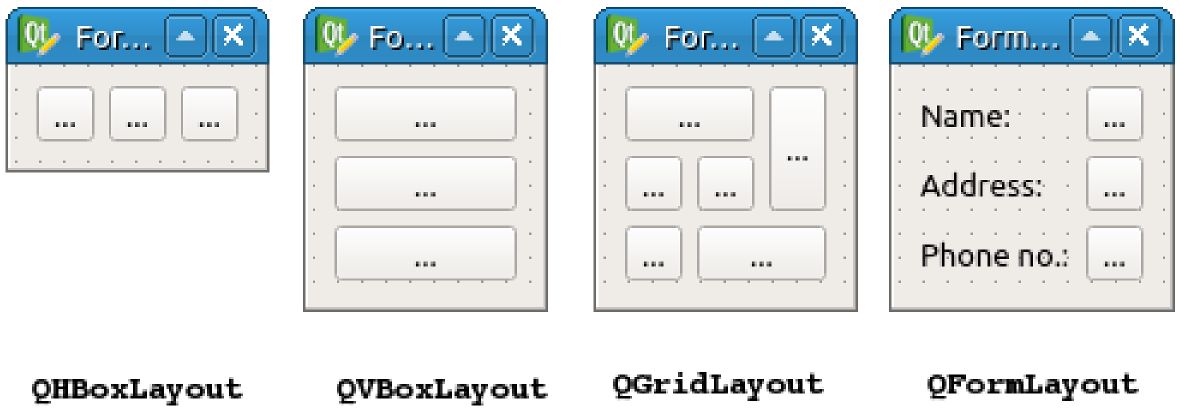
Property	Value
QObject	
objectName	MainWindow
QWidget	
enabled	✓
geometry	[(0, 0), 456 x 389]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
Width	0
Height	0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Sans, 10]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
windowTitle	MainWindow
windowIcon	
windowOpacity	1.000000
toolTip	
toolTipDuration	-1
statusTip	
whatsThis	
accessibleName	
accessibleDescri...	
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	English, UnitedStates
windowFilePath	
inputMethodHints	ImhNone
windowModality	NonModal

Layouts

If you build and run the project now, you will see the window with two labels and a button, but they will remain in the exact positions you left them. This is what you almost never want. Usually, it is desired that widgets are automatically resized based on their content and the size of their neighbors. They need to adjust to the changes of the window's size (or, in contrast, the window size may need to be restricted based on possible sizes of the widgets inside of it). This is a very important feature for a cross-platform application, as you cannot assume any particular screen resolution or size of controls. In Qt, all of this requires us to use a special mechanism called **layouts**.

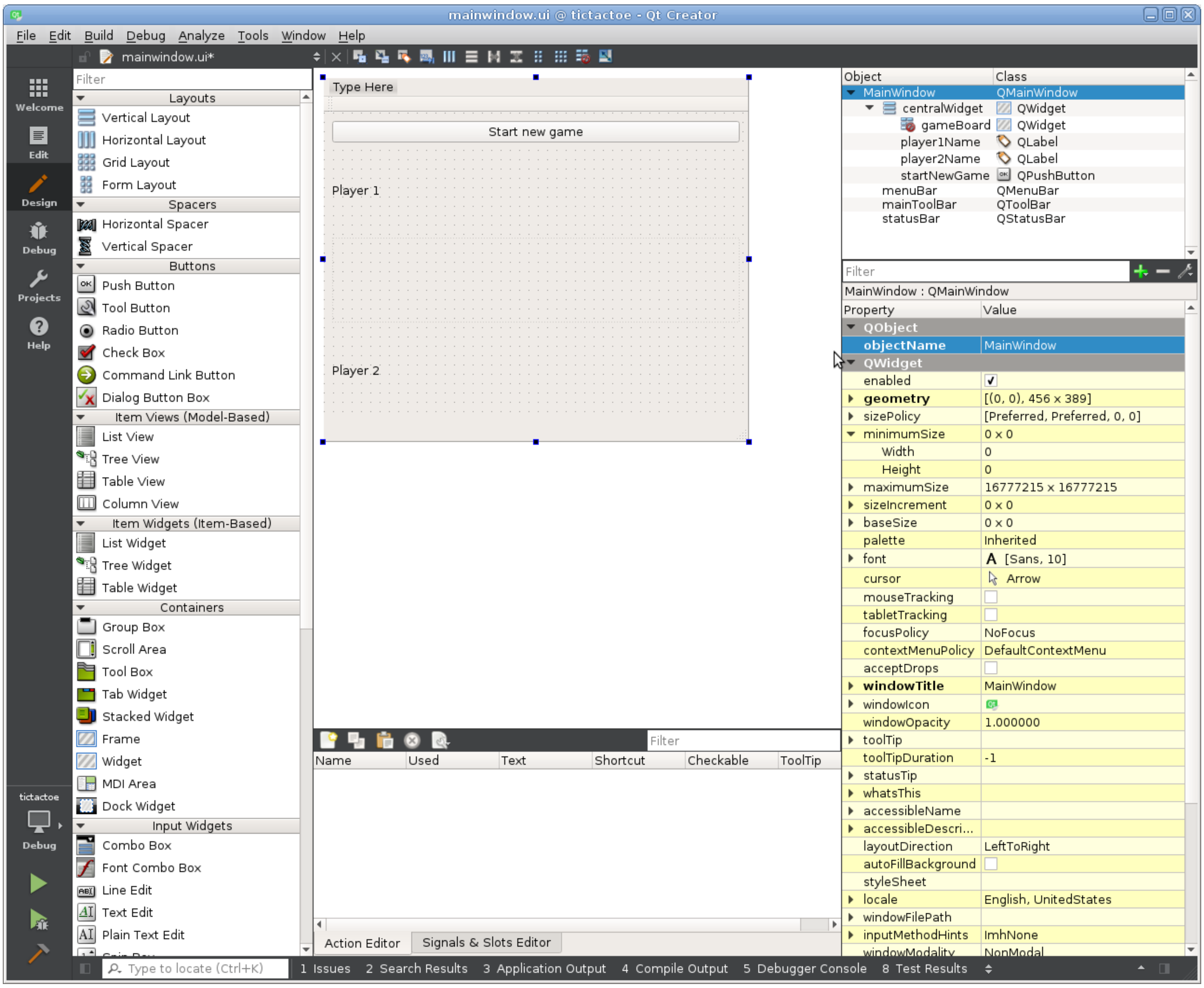
Layouts allow us to arrange the content of a widget, ensuring that its space is used efficiently. When we set a layout on a widget, we can start adding widgets, and even other layouts, and the mechanism will resize and reposition them according to the rules that we specify. When something happens in the user interface that influences how widgets should be displayed (for example, the label text is replaced with longer text, which makes the label require more space to show its content), the layout is triggered again, which recalculates all positions and sizes and updates widgets, as necessary.

Qt comes with a predefined set of layouts that are derived from the `QLayout` class, but you can also create your own. The ones that we already have at our disposal are `QHBoxLayout` and `QVBoxLayout`, which position items horizontally and vertically; `QGridLayout`, which arranges items in a grid so that an item can span across columns or rows; and `QFormLayout`, which creates two columns of items with item descriptions in one column and item content in the other. There is also `QStackedLayout`, which is rarely used directly and which makes one of the items assigned to it possess all the available space. You can see the most common layouts in action in the following figure:



Time for action – Adding a layout to the form

Select the MainWindow top-level item in the object tree and click on , the Lay Out Vertically icon in the upper toolbar. The button, labels, and the empty widget will be automatically resized to take all the available space of the form in the central area:



Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
gameBoard	QWidget
player1Name	QLabel
player2Name	QLabel
startNewGame	QPushButton
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

MainWindow : QMainWindow

Property	Value
QObject	
objectName	MainWindow
QWidget	
enabled	✓
geometry	[(0, 0), 456 x 389]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
Width	0
Height	0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Sans, 10]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
windowTitle	MainWindow
windowIcon	
windowOpacity	1.000000
toolTip	
toolTipDuration	-1
statusTip	
whatsThis	
accessibleName	
accessibleDescri...	
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	English, UnitedStates
windowFilePath	
inputMethodHints	ImhNone
windowModality	NonModal

If the items were arranged in a different order, you can drag and drop them to change the order.

Run the application and check that the window's contents are automatically positioned and resized to use all the available space when the window is resized. Unfortunately, the labels take more vertical space than they really require, resulting in an empty space in the application window. We will fix this issue later in this chapter when we learn about size policies.

You can test the layouts of your form without building and running the whole application. Open the Tools menu, go to the Form Editor submenu, and choose the Preview entry. You will see a new window open that looks exactly like the form we just designed. You can resize the window and interact with the objects inside to monitor the behavior of the layouts and widgets. What really happened here is that Qt Creator built a real window for us based on the description that we provided in all the areas of the design mode. Without *any compilation, in a blink of an eye, we received a fully working window with all the layouts working and all the properties adjusted to our liking. This is a very important tool, so ensure that you use it often to verify that your layouts are controlling all the widgets as you intended them to—it is much faster than compiling and running the whole application just to check whether the widgets stretch or squeeze properly. You can also resize the form in the central area of the form editor by dragging its bottom-right corner, and if the layouts are set up correctly, the contents should be resized and repositioned.*

Now that you can create and display a form, two important operations need to be implemented. First, you need to receive notifications when the user interacts with your form (for example, presses a button) to perform some actions in the code. Second, you need to change the properties of the form's contents programmatically, and fill it with real data (for example, set player names from the code).

Signals and slots

To trigger functionality as a response to something that happens in an application, Qt uses a mechanism of signals and slots. This is another important feature of the `QObject` class. It's based on connecting a notification (which Qt calls a **signal**) about a change of state in some object with a function or method (called a **slot**) that is executed when such a notification arises. For example, if a button is pressed, it **emits** (sends) a `clicked()` signal. If some method is connected to this signal, the method will be called whenever the button is pressed.

Signals can have arguments that serve as a payload. For example, an input box widget (`QLineEdit`) has a `textEdited(const QString &text)` signal that's emitted when the user edits the text in the input box. A slot connected to this signal will receive the new text in the input box as its argument (provided it has an argument).

Signals and slots can be used with all classes that inherit `QObject` (including all widgets). A signal can be connected to a slot, member function, or functor (which includes a regular global function). When an object emits a signal, any of these entities that are connected to that signal will be called. A signal can also be connected to another signal, in which case emitting the first signal will make the other signal be emitted as well. You can connect any number of slots to a single signal and any number of signals to a single slot.

Creating signals and slots

If you create a `QObject` subclass (or a `QWidget` subclass, as QWidget inherits QObject), you can mark a method of this class as a signal or a slot. If the parent class had any signals or non-private slots, your class will also inherit them.

In order for signals and slots to work properly, the class declaration must contain the `Q_OBJECT` macro in a private section of its definition (Qt Creator has generated it for us). When the project is built, a special tool called **Meta-Object Compiler** (**moc**) will examine the class's header and generate some extra code necessary for signals and slots to work properly.

Keep in mind that **moc** *and all other Qt build tools do not edit the project files. Your C++ files are passed to the compiler without any changes. All special effects are achieved by generating separate C++ files and adding them to the compilation process.*

A signal can be created by declaring a class method in the `signals` section of the class declaration:

```
signals:
    void valueChanged(int newValue);
```

However, we don't implement such a method; this will be done automatically by **moc**. We can send (`emit`) the signal by calling the method. There is a convention that a signal call should be preceded by the `emit` macro. This macro has no effect (it's actually a blank macro), but it helps us clarify our intent to emit the signal:

```
void MyClass::setValue(int newValue) {
    m_value = newValue;
    emit valueChanged(newValue);
}
```

You should only emit signals from within the class methods, as if it were a protected function.

Slots are class methods declared in the `private slots`, `protected slots`, or `public slots` section of the class declaration. Contrary to signals, slots need to be implemented. Qt will call the slot when a signal connected to it is emitted. The visibility of the slot (private, protected, or public) should be chosen using the same principles as for normal methods.

The C++ standard only describes three types of sections of the class definition (`private`, `protected`, and `public`), so you may wonder how these special sections work. They are actually simple macros: the `signals` macro expands to `public`, and `slots` is a blank macro. So, the compiler treats them as normal methods. These keywords are, however, used by **moc** *to determine how to generate the extra code.*

Connecting signals and slots

Signals and slots can be connected and disconnected dynamically using the `QObject::connect()` and `QObject::disconnect()` functions. A regular, signal-slot connection is defined by the following four attributes:

- An object that changes its state (sender)
- A signal in the sender object
- An object that contains the function to be called (receiver)
- A slot in the receiver

If you want to make the connection, you need to call the `QObject::connect` function and pass these four parameters to it. For example, the following code can be used to clear the input box whenever the button is clicked on:

```
connect(button,   &QPushButton::clicked,
        lineEdit, &QLineEdit::clear);
```

Signals and slots in this code are specified using a standard C++ feature called pointers to member functions. Such a pointer contains the name of the class and the name of the method (in our case, signal or slot) in that class. Qt Creator's code autocompletion will help you write connect statements. In particular, if you press *Ctrl* + *Space* after
`connect(button, &`, it will insert the name of the class, and if you do that after `connect(button, &QPushButton::`, it will suggest one of the available signals (in another context, it would suggest all the existing methods of the class).

Note that you can't set the arguments of signals or slots when making a connection. Arguments of the source signal are always determined by the function that emits the signal. Arguments of the receiving slot (or signal) are always the same as the arguments of the source signal, with two exceptions:

- If the receiving slot or signal has fewer arguments than the source signal, the remaining arguments are ignored. For example, if you want to use the `valueChanged(int)` signal but don't care about the passed value, you can connect this signal to a slot without arguments.
- If the types of the corresponding arguments are not the same, but an implicit conversion between them exists, that conversion is performed. This means that you can, for example, connect a signal carrying a `double` value with a slot taking an `int` parameter.

If the signal and the slot do not have compatible signatures, you will get a compile-time error.

An existing connection is automatically destroyed after the sender or the receiver objects are deleted. Manual disconnection is rarely needed. The `connect()` function returns a connection handle that can be passed to `disconnect()`. Alternatively, you can call `disconnect()` with the same arguments the `connect()` was called with to undo the connection.

You don't always need to declare a slot to perform a connection. It's possible to connect a signal to a standalone function:

```
connect(button, &QPushButton::clicked, someFunction);
```

The function can also be a lambda expression, in which case it is possible to write the code directly in the `connect` statement:

```
connect(pushButton, &QPushButton::clicked, []()
{
    qDebug() << "clicked!";
});
```

It can be useful if you want to invoke a slot with a fixed argument value that can't be carried by a signal because it has less arguments. A solution is to invoke the slot from a lambda function (or a standalone function):

```
connect(pushButton, &QPushButton::clicked, [label]()
{
    label->setText("button was clicked");
});
```

A function can even be replaced with a function object (functor). To do this, we create a class, for which we overload the call operator that is compatible with the signal that we wish to connect to, as shown in the following code snippet:

```
class Functor {
public:
    Functor(const QString &name) : m_name(name) {}
    void operator()(bool toggled) const {
        qDebug() << m_name << ": button state changed to" << toggled
    }
private:
    QString m_name;
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton *button = new QPushButton();
    button->setCheckable(true);
    QObject::connect(button, &QPushButton::toggled,
                     Functor("my functor"));
    button->show();
    return a.exec();
}
```

This is often a nice way to execute a slot with an additional parameter that is not carried by the signal, as this is much cleaner than using a lambda expression. However, keep in mind that automatic disconnection will not happen when the object referenced in the lambda expression or the functor is deleted. This can lead to a use-after-free bug.

While it is actually possible to connect a signal to a method of a `QObject`-based class that is not a slot, doing this is not recommended. Declaring the method as a slot shows your intent better. Additionally, methods that are not slots are not available to Qt at runtime, which is required in some cases.

Old connect syntax

Before Qt 5, the old connect syntax was the only option. It looks as follows:

```
connect(spinBox, SIGNAL(valueChanged(int)),
        dial,    SLOT(setValue(int)));
```

This statement establishes a connection between the signal of the `spinBox` object called `valueChanged` that carries an `int` parameter and a `setValue` slot in the `dial` object that accepts an `int` parameter. It is forbidden to put argument names or values in a
`connect` statement. Qt Creator is usually able to suggest all possible inputs in this context if you press *Ctrl* + *Space* after `SIGNAL(` or `SLOT(`.

While this syntax is still available, we discourage its wide use, because it has the following drawbacks:

- If the signal or the slot is incorrectly referenced (for example, its name or argument types are incorrect) or if argument types of the signals and the slot are not compatible, there will be no compile-time error, only a runtime warning. The new syntax approach performs all the necessary checks at compile time.
- The old syntax doesn't support casting argument values to another type (for example, connect a signal carrying a `double` value with a slot taking an `int` parameter).
- The old syntax doesn't support connecting a signal to a standalone function, a lambda expression, or a functor.

The old syntax also uses macros and may look unclear to developers not familiar with Qt. It's hard to say which syntax is easier to read (the old syntax displays argument types, while the new syntax displays the class name instead). However, the new syntax has a big disadvantage when using overloaded signals or slots. The only way to resolve the overloaded function type is to use an explicit cast:

```
connect(spinBox,
        static_cast<void (QSpinBox::*)(int)>(&QSpinBox::valueChanged),
        ...);
```

The old connect syntax includes argument types, so it doesn't have this issue. In this case, the old syntax may look more acceptable, but compile-time checks may still be considered more valuable than shorter code. In this book, we prefer the new syntax, but use the old syntax when working with overloaded methods for the sake of clarity.

Signal and slot access specifiers

As mentioned earlier, you should only emit signals from the class that owns it or from its subclasses. However, if signals were really protected or private, you would not be able to connect to them using the pointer-to-member function syntax. To make such connections possible, signals are made public functions. This means that the compiler won't stop you from calling the signal from outside. If you want to prevent such calls, you can declare `QPrivateSignal` as the last argument of the signal:

```
signals:
    void valueChanged(int value, QPrivateSignal);
```

`QPrivateSignal` is a private struct created in each `QObject` subclass by the `Q_OBJECT` macro, so you can only create `QPrivateSignal` objects in the current class.

Slots can be public, protected, or private, depending on how you want to restrict access to them. When using the pointer to a member function syntax for connection, you will only be able to create pointers to slots if you have access to them. It's also correct to call a slot directly from any other location as long as you have access to it.

That being said, Qt doesn't really support restricting access to signals and slots. Regardless of how a signal or a slot is declared, you can always access it using the old connect syntax. You can also call any signal or slot using the `QMetaObject::invokeMethod` method. While you can restrict direct C++ calls to reduce the possibility of errors, keep in mind that the users of your API still can access any signal or slot if they really want to.

Time for action – Receiving the button-click signal from the form

Open the `mainwindow.h` file and create a `private slots` section in the class declaration, then declare the `startNewGame()` private slot, as shown in the following code:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private slots:
    void startNewGame();
}
```

To quickly implement a freshly declared method, we can ask Qt Creator to create the skeleton code for us by positioning the text cursor at the method declaration, pressing *Alt* + *Enter* on the keyboard, and choosing Add definition in mainwindow.cpp from the popup.

It also works the other way round. You can write the method body first and then position the cursor on the method signature, press Alt + Enter, and choose Add (...) declaration from the quick-fix menu. There are also various other context-dependent fixes that are available in Creator.

Write the highlighted code in the implementation of this method:

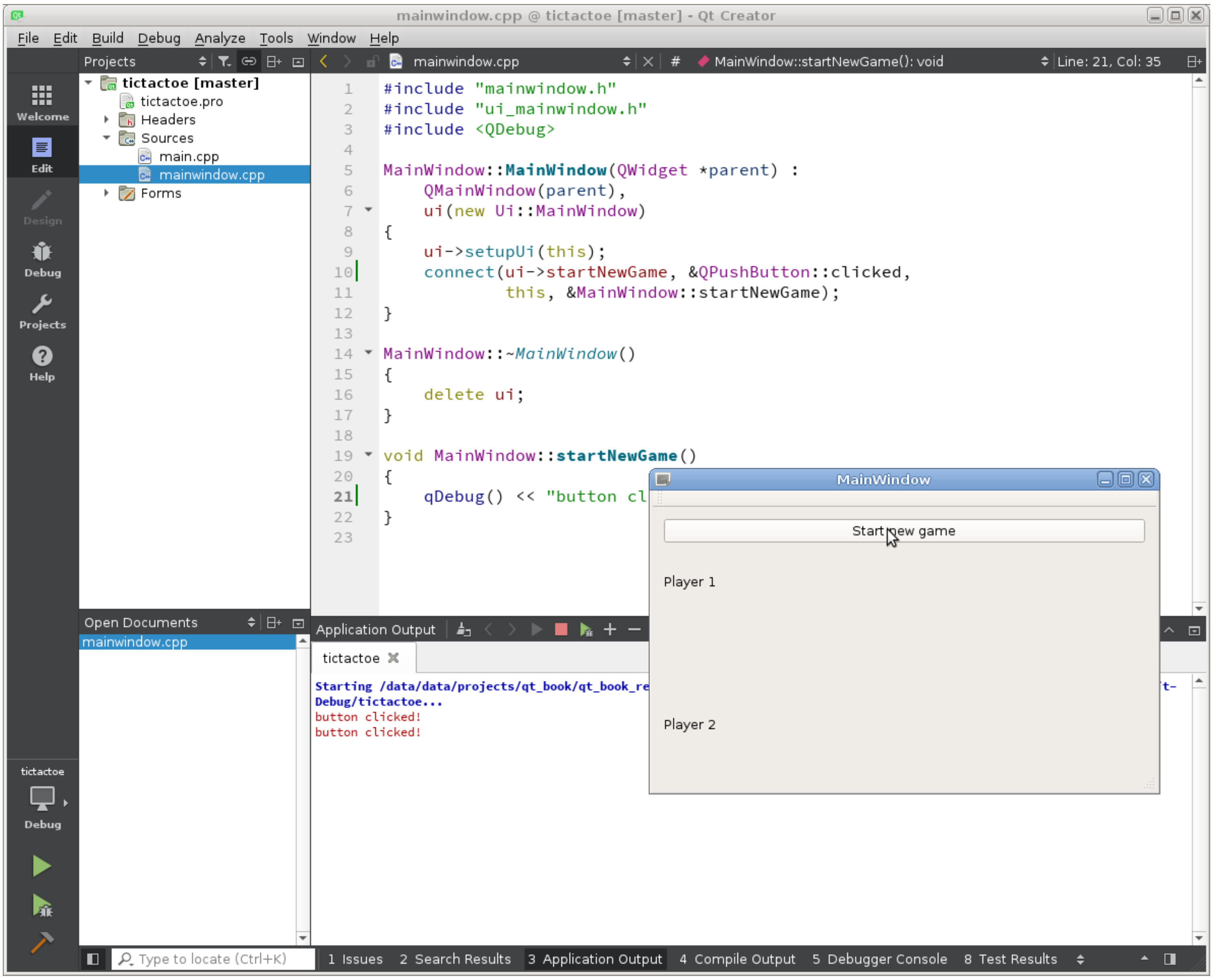
```
void MainWindow::startNewGame()
{
    qDebug() << "button clicked!";
}
```

Add `#include <QDebug>` to the top section of the `mainwindow.cpp` file to make the `qDebug()` macro available.

Finally, add a connect statement to the constructor after the `setupUi()` call:

```
ui->setupUi(this);
connect(ui->startNewGame, &QPushButton::clicked,
        this, &MainWindow::startNewGame);
```

Run the application and try clicking on the button. The `button clicked!` text should appear in the Application Output pane in the bottom part of Qt Creator's window (if the pane isn't activated, use the Application Output button in the bottom panel to open it):



```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect(ui->startNewGame, &QPushButton::clicked,
            this, &MainWindow::startNewGame);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::startNewGame()
{
    qDebug() << "button cl
}
```

```
Starting /data/data/projects/qt_book/qt_book_re                t-
Debug/tictactoe...
button clicked!
button clicked!
```

What just happened?

We created a new private slot in the `MainWindow` class and connected the `clicked()` signal of the Start new game button to the slot. When the user clicks on the button, Qt will call our slot, and the code we wrote inside it gets executed.

Ensure that you put any operations with the form elements after the `setupUi()` call. This function creates the elements, so `ui->startNewGame` will simply be uninitialized before `setupUi()` is called, and attempting to use it will result in undefined behavior.

`qDebug() << ...` is a convenient way to print debug information to the `stderr` (standard error output) of the application process. It's quite similar to the `std::cerr << ...` method available in the standard library, but it separates supplied values with spaces and appends a new line at the end.

Putting debug outputs everywhere quickly becomes inconvenient. Luckily, Qt Creator has powerful integration with C++ debuggers, so you can use Debug mode to check whether some particular line is executing, see the current values of the local variables at that location, and so on. For example, try setting a break point at the line containing `qDebug()` by clicking on the space to the left of the line number (a red circle indicating the break point should appear). Click on the Start Debugging button (a green triangle with a bug at the bottom-left corner of Qt Creator), wait for the application to launch, and press the Start new game button. When the application enters the break point location, it will pause, and Qt Creator's window will be brought to the front. The yellow arrow over the break point circle will indicate the current step of the execution. You can use the buttons below the code editor to continue execution, stop, or execute the process in steps. Learning to use the debugger becomes very important when developing large applications. We will talk more about using the debugger later,

Automatic slot connection and its drawbacks

Qt also offers an easier way to make a connection between signals of the form's elements and the slots of the class. You can right-click on the button in the central area of the form editor and select the Go to slot... option. You will be prompted to select one of the signals available in the button's class (`QPushButton`). After you select the `clicked()` signal, Qt Creator will automatically add a new `on_startNewGame_clicked` slot to our `MainWindow` class.

The tricky part is that there is no `connect()` call that enforces the connection. How is the button's signal connected to this slot, then? The answer is Qt's automatic slot connection feature. When the constructor calls the `ui->setupUi(this)` function, it creates the widgets and other objects in the form and then calls the `QMetaObject::connectSlotsByName` method. This method looks at the list of slots existing in the widget class (in our case, `MainWindow`) and searches for ones that have their name in an `on_<object name>_<signal name>` pattern, where `<object name>` is the `objectName` of an existing child widget and `<signal name>` is the name of one of this widget's signals. In our case, a button called `startNewGame` is a child widget of our widget, and it has a `clicked` signal, so this signal is automatically connected to an `on_startNewGame_clicked` slot.

While this is a really convenient feature, it has many drawbacks:

- It makes your application harder to maintain. If you rename or remove the form element, you have to update or remove the slot manually. If you forget to do that, the application will only produce a warning at runtime when the automatic connection fails. In a large application, especially when not all forms are instantiated at the start of the application, there is a significant risk that you will miss the warning and the application will not work as intended.
- You have to use a specific name for the slot (for example, `on_startNewGame_clicked()` instead of a clean-looking `startNewGame()`).
- Sometimes you want to connect signals from multiple objects to the same slot. Automatic slot connection doesn't provide a way to do this, and creating multiple slots just to call a single function will lead to unnecessary code bloat.
- Automatic slot connection has a runtime cost, because it needs to examine the available children and slots and find the matching ones, but it's usually insignificant since it only runs when the form object is created.

The basic approach shown in the previous section is much more maintainable. Making an explicit `connect()` call with pointers to member functions will ensure that both signal and slot are specified properly. If you rename or remove the button, it will immediately result in a compilation error that is impossible to miss. You are also free to choose a meaningful name for the slot, so you can make it part of your public API, if desired.

Considering all this, we advise against using the automatic slot connection feature, as the convenience does not outweigh the drawbacks.

Time for action – Changing the texts on the labels from the code

Printing text to the console is not as impressive as changing the text in our form. We don't have GUI for letting users enter their names yet, so we'll hardcode some names for now. Let's change the implementation of our slot to the following:

```
void MainWindow::startNewGame()
{
    ui->player1Name->setText(tr("Alice"));
    ui->player2Name->setText(tr("Bob"));
}
```

Now, when you run the application and click on the button, the labels in the form will change. Let's break down this code into pieces:

- As mentioned earlier, the first label's object is accessible in our class as `ui->player1Name` and has the `QLabel *` type.
- We're calling the `setText` method of the `QLabel` class. This is the setter of the `text` property of `QLabel` (the same property that we edited in the property editor of the Design mode). As per Qt's naming convention, getters should have

  followed by the property name. You can set the text cursor on `setText` and press *F1* to learn more about the property and its access functions.
- The `tr()` function (which is short for "translate") is used to translate the text to the current UI language of the application. By default, this function returns the passed string unchanged, but it's a good habit to wrap any and all string literals that are displayed to the user in this function. Any user-visible text that you enter in the form editor is also subject to translation and is passed through a similar function automatically. Only strings that should not be affected by translation (for example, object names that are used as identifiers) should be created without the `tr()` function.

Creating a widget for the tic-tac-toe board

Let's move on to implementing the board. It should contain nine buttons that can display "X" or "O" and allow the players to make their moves. We could add the button directly to the empty widget of our form. However, the behavior of the board is fairly separate from the rest of the form, and it will have quite a bit of logic inside. Following the encapsulation principle, we prefer implementing the board as a separate widget class. Then, we'll replace the empty widget in our main window with the board widget we created.

Choosing between designer forms and plain C++ classes

One way of creating a custom widget is by adding a Designer Form Class to the project. Designer Form Class is a template provided by Qt Creator. It consists of a C++ class that inherits `QWidget` (directly or indirectly) and a designer form (`.ui` file), tied together by some automatically generated code. Our `MainWindow` class also follows this template.

However, if you try to use the visual form editor to create our tic-tac-toe board, you may find it quite inconvenient for this task. One problem is that you need to add nine identical buttons to the form manually. Another issue is accessing these buttons from the code when you need to make a signal connection or change the button's text. The `ui->objectName` approach is not applicable here because you can only access a concrete widget this way, so you'd have to resort to other means, such as the `findChild()` method that allows you to search for a child object by its name.

In this case, we prefer to add the buttons in the code, where we can make a loop, set up each button, and put them into an array for easy addressing. The process is pretty similar to how the designer forms operate, but we'll do it by hand. Of course, anything that the form editor can do is accessible through the API.

> *After you build the project, you can hold Ctrl and click on `ui_mainwindow.h` at the beginning of `mainwindow.cpp` to see the code that actually sets up our main window. You should not edit this file, because your changes will not be persistent.*

Time for action – Creating a game board widget

Locate the `tictactoe` folder in the project tree (it's the top-level entry corresponding to our whole project), open its context menu, and select Add New... Select C++ in the left list and C++ Class in the central list. Click on the Choose button, input `TicTacToeWidget` in the Class name field, and select QWidget in the Base class drop-down list. Click on Next and Finish. Qt Creator will create header and source files for our new class and add them to the project.

Open the `tictactoewidget.h` file in Creator and update it by adding the highlighted code:

```
#ifndef TICTACTOEWIDGET_H
#define TICTACTOEWIDGET_H
#include <QWidget>
class TicTacToeWidget : public QWidget
{
    Q_OBJECT
public:
    TicTacToeWidget(QWidget *parent = nullptr);
private:
    QVector<QPushButton*> m_board;
signals:
};
#endif // TICTACTOEWIDGET_H
```

Our additions create a `QVector` object (a container similar to `std::vector`) that can hold pointers to instances of the `QPushButton` class, which is the most commonly used button class in Qt. We have to include the Qt header containing the `QPushButton` declaration. Qt Creator can help us do this quickly. Set the text cursor on `QPushButton`, press *Alt* + *Enter*, and select Add #include <QPushButton>. The include directive will appear at the beginning of the file. As you may have noted, each Qt class is declared in the header file that is called exactly the same as the class itself.

From now on, this book will not remind you about adding the include directives to your source code—you will have to take care of this by yourself. This is really easy; just remember that to use a Qt class you need to include a file named after that class.

The next step is to create all the buttons and use a layout to manage their geometries. Switch to the `tictactoewidget.cpp` file and locate the constructor.

> *You can use the F4 key to switch between the corresponding header and the source files. You can also use the F2 key to navigate from the definition of a method to its implementation, and back.*

First, let's create a layout that will hold our buttons:

```
QGridLayout *gridLayout = new QGridLayout(this);
```

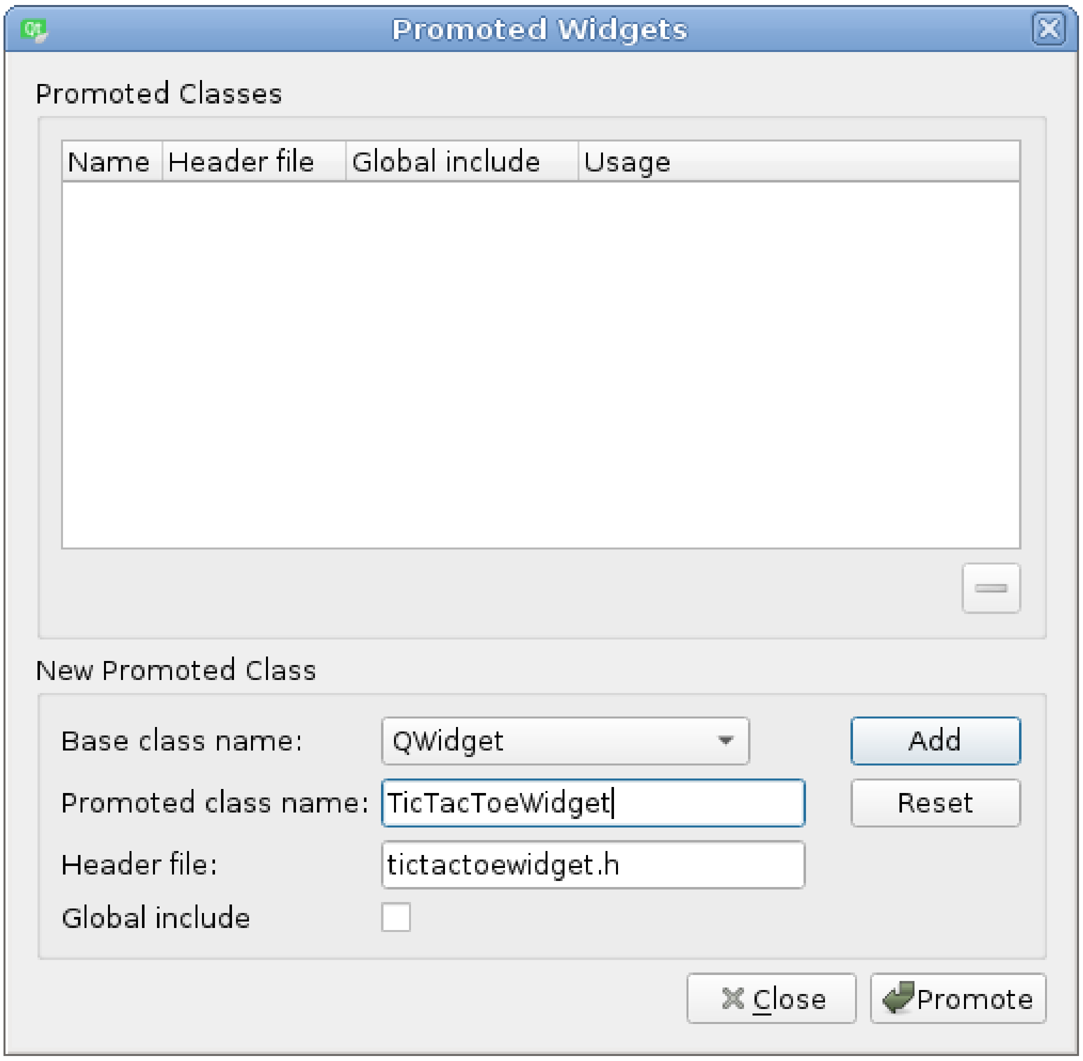
By passing the `this` pointer to the layout's constructor, we attached the layout to our widget. Then, we can start adding buttons to the layout:

```
for(int row = 0; row < 3; ++row) {
    for(int column = 0; column < 3; ++column) {
        QPushButton *button = new QPushButton(" ");
        gridLayout->addWidget(button, row, column);
        m_board.append(button);
    }
}
```

The code creates a loop over rows and columns of the board. In each iteration, it creates an instance of the `QPushButton` class. The content of each button is set to a single space so that it gets the correct initial size. Then, we add the button to the layout in `row` and `column`. At the end, we store the pointer to the button in the vector that was declared earlier. This lets us reference any of the buttons later on. They are stored in the vector in such an order that the first three buttons of the first row are stored first, then the buttons from the second row, and finally those from the last row.

This should be enough for testing the widget. Let's add it to our main window. Open the `mainwindow.ui` file. Invoke the context menu of the empty widget called `gameBoard` and choose Promote to. This allows us to **promote** a widget to another class, that is, substitute a widget in the form with an instance of another class.

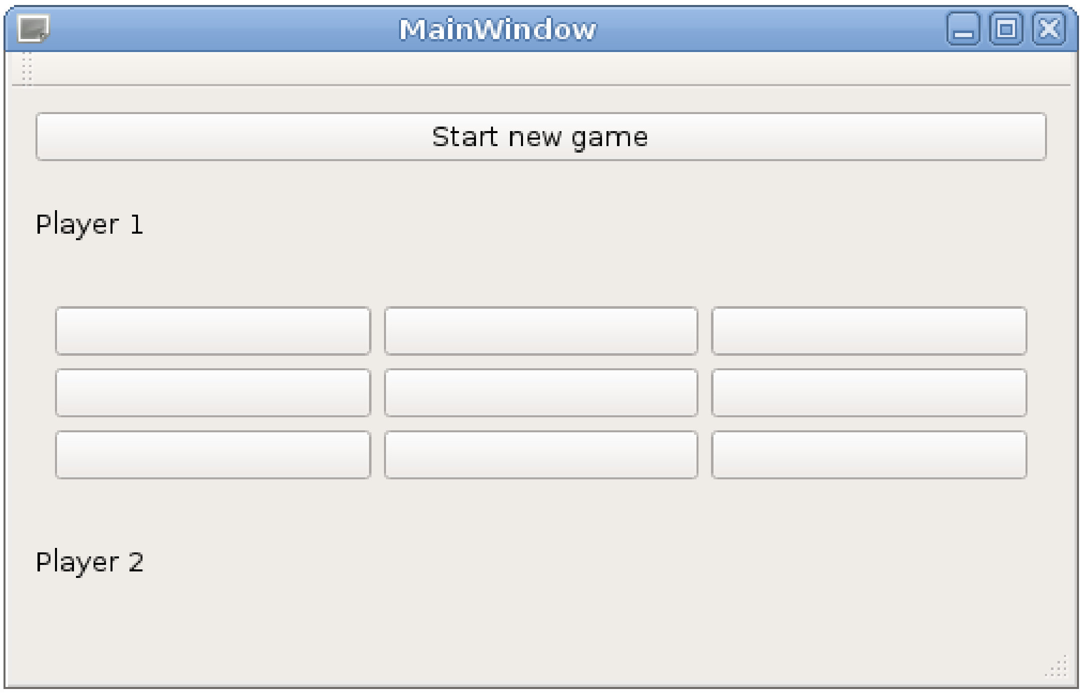
In our case, we will want to replace the empty widget with our game board. Select QWidget in the Base class name list, because our `TicTacToeWidget` is inherited from `QWidget`. Input `TicTacToeWidget` into the Promoted class name field and verify that the Header file field contains the correct name of the class's header file, as illustrated:



Base class name: QWidget
Promoted class name: TicTacToeWidget
Header file: tictactoewidget.h
Global include ☐

Then, click on the button labeled Add and then Promote, to close the dialog and confirm the promotion. You will not note any changes in the form, because the replacement only takes place at runtime (however, you will see the `TicTacToeWidget` class name next to `gameBoard` in the object tree).

Run the application and check whether the game board appears in the main window:



What just happened?

Not all widget types are directly available in the form designer. Sometimes, we need to use widget classes that will only be created in the project that is being built. The simplest way to be able to put a custom widget on a form is to ask the designer to replace the class name of a standard widget with a custom name. By promoting an object to a different class, we saved a lot of work trying to otherwise fit our game board into the user interface.

You are now familiar with two ways of creating custom widgets: you can use the form editor or add widgets from the code. Both approaches are valuable. When creating a new widget class in your project, choose the most convenient way depending on your current task.

Automatic deletion of objects

You might have noted that although we created a number of objects in the constructor using the `new` operator, we didn't destroy those objects anywhere (for example, in the destructor). This is because of the way the memory is managed by Qt. Qt doesn't do any garbage collecting (as C# or Java does), but it has this nice feature related to `QObject` parent–child hierarchies. The rule is that whenever a `QObject` instance is destroyed, it also deletes all of its children. This is another reason to set parents to the objects that we create—if we do this, we don't have to care about explicitly freeing any memory.

Since all layouts and widgets inside our top-level widget (an instance of `MainWindow` class) are its direct or indirect children, they will all be deleted when the main window is destroyed. The `MainWindow` object is created in the `main()` function without the `new` keyword, so it will be deleted at the end of the application after `a.exec()` returns.

When working with widgets, it's pretty easy to verify that every object has a proper parent. You can assume that anything that is displayed inside the window is a direct or indirect child of that window. However, the parent–child relationship becomes less apparent when working with invisible objects, so you should always check that each object has a proper parent and therefore will be deleted at some point. For example, in our `TicTacToeWidget` class, the `gridLayout` object receives its parent through a constructor argument (`this`). The button objects are initially created without a parent, but the `addWidget()` function assigns a parent widget to them.

Time for action – Functionality of a tic-tac-toe board

We need to implement a function that will be called upon by clicking on any of the nine buttons on the board. It has to change the text of the button that was clicked on—either "X" or "O"—based on which player made the move. It then has to check whether the move resulted in the game being won by the player (or a draw if no more moves are possible), and if the game ended, it should emit an appropriate signal, informing the environment about the event.

When the user clicks on a button, the `clicked()` signal is emitted. Connecting this signal to a custom slot lets us implement the mentioned functionality, but since the signal doesn't carry any parameters, how do we tell which button caused the slot to be triggered? We could connect each button to a separate slot, but that's an ugly solution. Fortunately, there are two ways of working around this problem. When a slot is invoked, a pointer to the object that caused the signal to be sent is accessible through a special method in `QObject`, called `sender()`. We can use that pointer to find out which of the nine buttons stored in the board list is the one that caused the signal to fire:

```
void TicTacToeWidget::someSlot() {
    QPushButton *button = static_cast<QPushButton*>(sender());
    int buttonIndex = m_board.indexOf(button);
    // ...
}
```

While `sender()` is a useful call, we should try to avoid it in our own code as it breaks some principles of object-oriented programming. Moreover, there are situations where calling this function is not safe. A better way is to use a dedicated class called `QSignalMapper`, which lets us achieve a similar result without using `sender()` directly. Modify the constructor of `TicTacToeWidget`, as follows:

```
QGridLayout *gridLayout = new QGridLayout(this);
QSignalMapper *mapper = new QSignalMapper(this);
for(int row = 0; row < 3; ++row) {
    for(int column = 0; column < 3; ++column) {
        QPushButton *button = new QPushButton(" ");
        gridLayout->addWidget(button, row, column);
        m_board.append(button);
        mapper->setMapping(button, m_board.count() - 1);
        connect(button, SIGNAL(clicked()), mapper, SLOT(map()));
    }
}
connect(mapper, SIGNAL(mappedInt(int)),
        this,   SLOT(handleButtonClick(int)));
```

Here, we first created an instance of `QSignalMapper` and passed a pointer to the board widget as its parent so that the mapper is deleted when the widget is deleted.

Almost all subclasses of `QObject` can receive a pointer to the parent object in the constructor. In fact, our `MainWindow` and `TicTacToeWidget` classes can also do that, thanks to the code Qt Creator generated in their constructors. Following this rule in custom `QObject`-based classes is recommended. While the parent argument is often optional, it's a good idea to pass it when possible, because objects will be automatically deleted when the parent is deleted. However, there are a few cases where this is redundant, for example, when you add a widget to a layout, the layout will automatically set the parent widget for it.

Then, when we create buttons, we "teach" the mapper that each of the buttons has a number associated with it—the first button will have the number `0`, the second one will be bound to the number `1`, and so on. By connecting the `clicked()` signal from the button to the mapper's `map()` slot, we tell the mapper to process that signal. When the mapper receives the signal from any of the buttons, it will find the mapping of the sender of the signal and emit another signal—`mapped()`—with the mapped number as its parameter. This allows us to connect to that signal with a new slot (`handleButtonClick()`) that takes the index of the button in the board list.

Before we create and implement the slot, we need to create a useful enum type and a few helper methods. First, add the following code to the public section of the class declaration in the `tictactoewidget.h` file:

```
enum class Player {
    Invalid, Player1, Player2, Draw
};
Q_ENUM(Player)
```

This enum lets us specify information about players in the game. The `Q_ENUM` macro will make Qt recognize the enum (for example, it will allow you to pass the values of this type to `qDebug()` and also make serialization easier). Generally, it's a good idea to use `Q_ENUM` for any enum in a `QObject`-based class.

We can use the `Player` enum immediately to mark whose move it is now. To do so, add a private field to the class:

```
Player m_currentPlayer;
```

Don't forget to give the new field an initial value in the constructor:

```
m_currentPlayer = Player::Invalid;
```

Then, add the two public methods to manipulate the value of this field:

```
Player currentPlayer() const
{
    return m_currentPlayer;
}
void setCurrentPlayer(Player p)
{
    if(m_currentPlayer == p) {
        return;
    }
    m_currentPlayer = p;
    emit currentPlayerChanged(p);
}
```

The last method emits a signal, so we have to add the signal declaration to the class definition along with another signal that we will use:

```
signals:
    void currentPlayerChanged(TicTacToeWidget::Player);
    void gameOver(TicTacToeWidget::Player);
```

We only emit the `currentPlayerChanged` signal when the current player really changes. You always have to pay attention that you don't emit a "changed" signal when you set a value to a field to the same value that it had before the function was called. Users of your classes expect that if a signal is called changed, it is emitted when the value really changes. Otherwise, this can lead to an infinite loop in signal emissions if you have two objects that connect their value setters to the other object's changed signal.

Now it is time to implement the slot itself. First, declare it in the header file:

```
private slots:
    void handleButtonClick(int index);
```

Use *Alt* + *Enter* to quickly generate a definition for the new method, as we did earlier.

When any of the buttons is pressed, the `handleButtonClick()` slot will be called. The index of the button clicked on will be received as the argument. We can now implement the slot in the `.cpp` file:

```
void TicTacToeWidget::handleButtonClick(int index)
{
    if (m_currentPlayer == Player::Invalid) {
        return; // game is not started
    }
    if(index < 0 || index >= m_board.size()) {
        return; // out of bounds check
    }
    QPushButton *button = m_board[index];
    if(button->text() != " ") return; // invalid move
    button->setText(currentPlayer() == Player::Player1 ? "X" : "O");
    Player winner = checkWinCondition();
    if(winner == Player::Invalid) {
        setCurrentPlayer(currentPlayer() == Player::Player1 ?
                         Player::Player2 : Player::Player1);
        return;
    } else {
        emit gameOver(winner);
    }
}
```

Here, we first retrieve a pointer to the button based on its index. Then, we check whether the button contains an empty space—if not, then it's already occupied, so we return from the method so that the player can pick another field in the board. Next, we set the current player's mark on the button. Then, we check whether the player has won the game. If the game didn't end, we switch the current player and return; otherwise, we emit a `gameOver()` signal, telling our environment who won the game. The `checkWinCondition()` method returns `Player1`, `Player2`, or `Draw` if the game has ended, and `Invalid` otherwise. We will not show the implementation of this method here, as it is quite lengthy. Try implementing it on your own, and if you encounter problems, you can see the solution in the code bundle that accompanies this project.

The last thing we need to do in this class is to add another public method for starting a new game. It will clear the board and set the current player:

```
void TicTacToeWidget::initNewGame() {
    for(QPushButton *button: m_board) {
        button->setText(" ");
    }
    setCurrentPlayer(Player::Player1);
}
```

Now we only need to call this method in the `MainWindow::startNewGame` method:

```
void MainWindow::startNewGame()
{
    ui->player1Name->setText(tr("Alice"));
    ui->player2Name->setText(tr("Bob"));
    ui->gameBoard->initNewGame();
}
```

Note that `ui->gameBoard` actually has a `TicTacToeWidget *` type, and we can call its methods even though the form editor doesn't know anything specific about our custom class. This is the result of the *promoting* that we did earlier.

It's time to see how all this works together! Run the application, click on the Start new game button, and you should be able to play some tic-tac-toe.

Time for action – Reacting to the game board's signals

While writing a turn-based board game, it is a good idea to always clearly mark whose turn it is now to make a move. We will do this by marking the moving player's name in bold. There is already a signal in the board class that tells us that the current player has changed, which we can react to update the labels.

We need to connect the board's `currentPlayerChanged` signal to a new slot in the `MainWindow` class. Let's add appropriate code into the `MainWindow` constructor:

```
ui->setupUi(this);
connect(ui->gameBoard, &TicTacToeWidget::currentPlayerChanged,
        this, &MainWindow::updateNameLabels);
```

Now, for the slot itself, declare the following methods in the `MainWindow` class:

```
private:
    void setLabelBold(QLabel *label, bool isBold);
private slots:
    void updateNameLabels();
```

Now implement them using the following code:

```
void MainWindow::setLabelBold(QLabel *label, bool isBold)
{
    QFont f = label->font();
    f.setBold(isBold);
    label->setFont(f);
}

void MainWindow::updateNameLabels()
{
    setLabelBold(ui->player1Name,
        ui->gameBoard->currentPlayer() ==
            TicTacToeWidget::Player::Player1);
    setLabelBold(ui->player2Name,
        ui->gameBoard->currentPlayer() ==
            TicTacToeWidget::Player::Player2);
}
```

What just happened?

`QWidget` (and, by extension, any widget class) has a `font` property that determines the properties of the font this widget uses. This property has the `QFont` type. We can't just write `label->font()->setBold(isBold);`, because `font()` returns a const reference, so we have to make a copy of the `QFont` object. That copy has no connection to the label, so we need to call `label->setFont(f)` to apply our changes. To avoid repetition of this procedure, we created a helper function, called `setLabelBold`.

The last thing that needs to be done is to handle the situation when the game ends. Connect the `gameOver()` signal from the board to a new slot in the main window class. Implement the slot as follows:

```
void MainWindow::handleGameOver(TicTacToeWidget::Player winner) {
    QString message;
    if(winner == TicTacToeWidget::Player::Draw) {
        message = tr("Game ended with a draw.");
    } else {
        QString winnerName = winner == TicTacToeWidget::Player::Player1 ?
                    ui->player1Name->text() : ui->player2Name->text();
        message = tr("%1 wins").arg(winnerName);
    }
    QMessageBox::information(this, tr("Info"), message);
}
```

This code checks who won the game, assembles the message and shows it using a static method `QMessageBox::information()` that shows a modal dialog containing the message and a button that allows us to close the dialog.

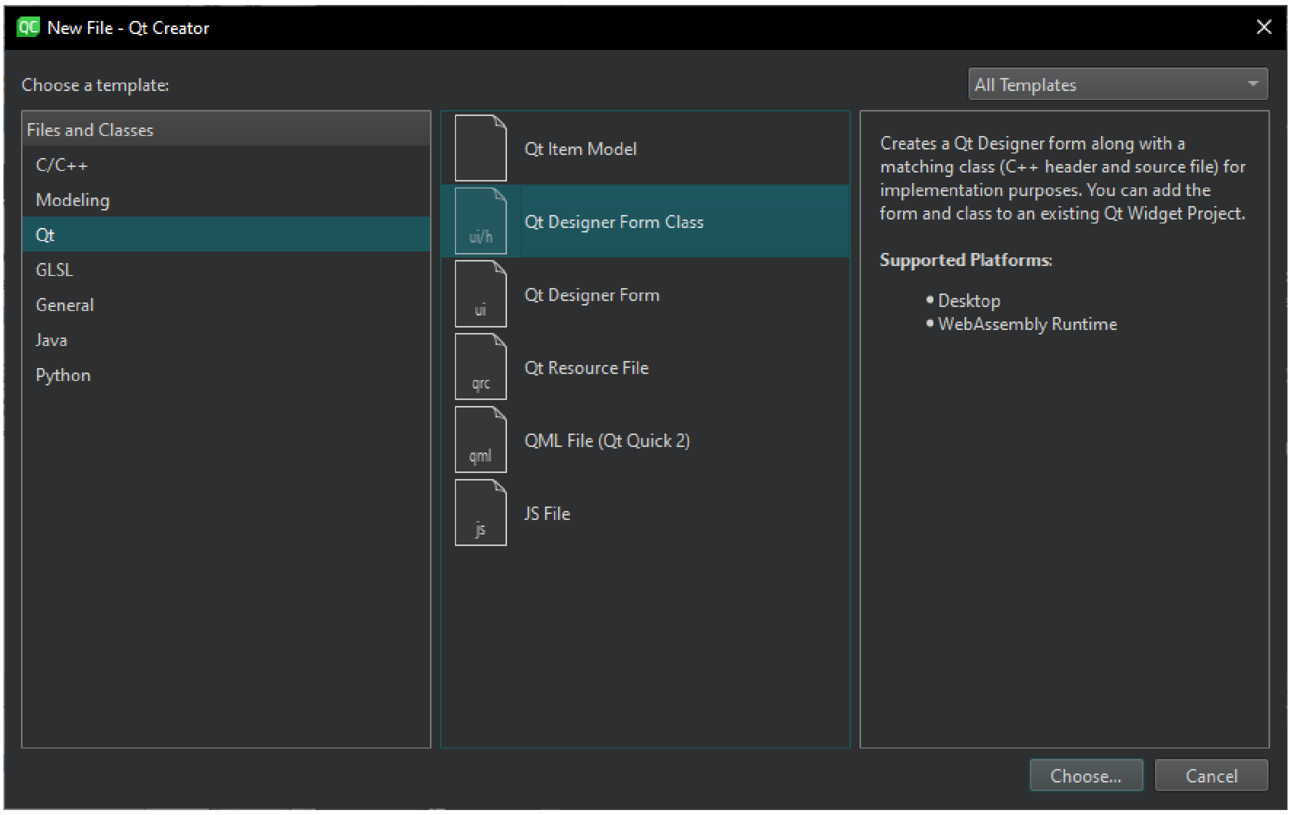
Run the game and check that it now highlights the current player and shows the message when the game ends.

Advanced form editor usage

Now it's time to give the players a way to input their names. We will do that by adding a game configuration dialog that will appear when starting a new game.

Time for action – Designing the game configuration dialog

First, select Add New... in the context menu of the tictactoe project and choose to create a new Qt Designer Form Class, as shown in the following screenshot:

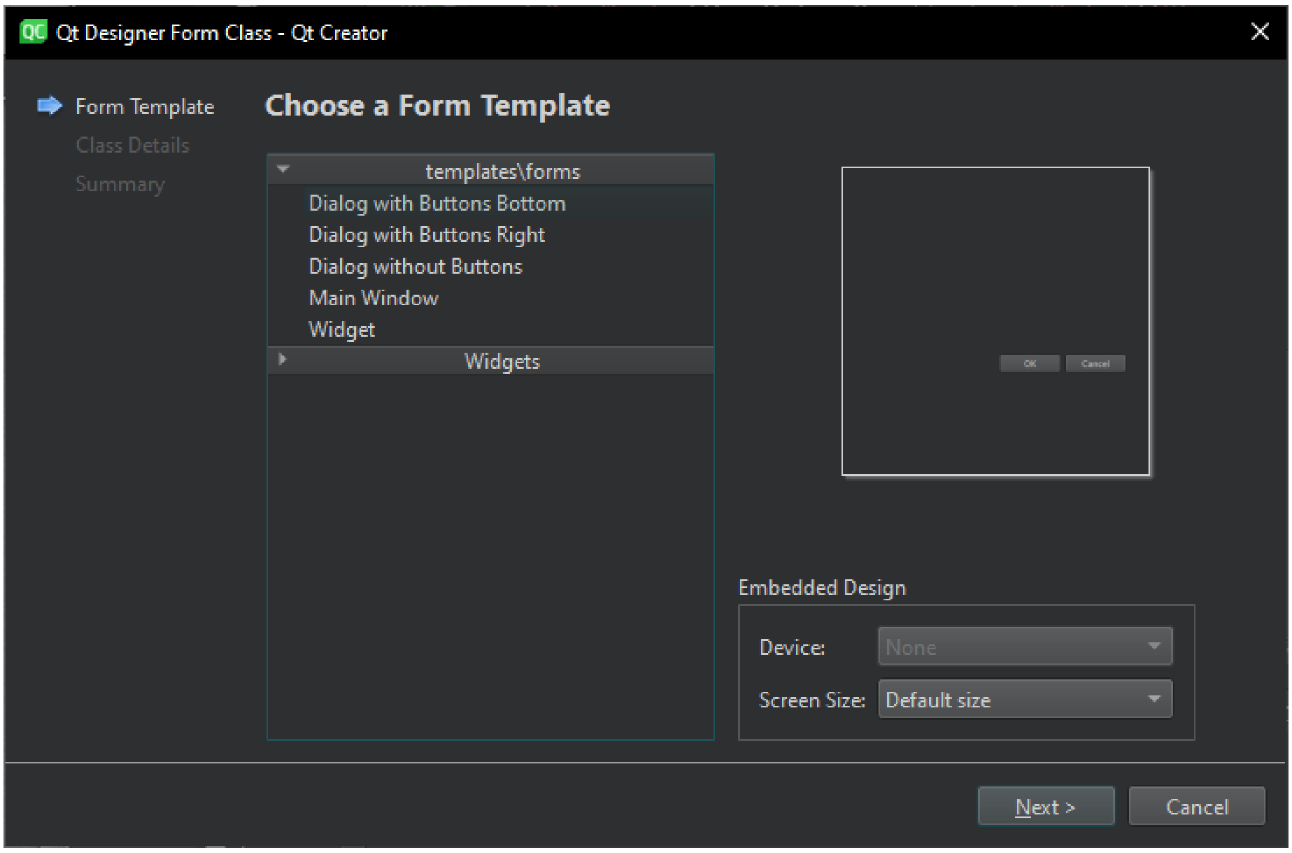


Creates a Qt Designer form along with a matching class (C++ header and source file) for implementation purposes. You can add the form and class to an existing Qt Widget Project.

Supported Platforms:

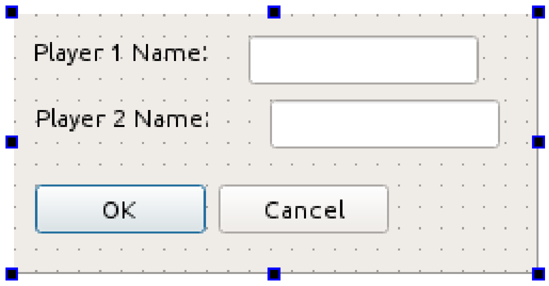
- Desktop
- WebAssembly Runtime

In the window that appears, choose Dialog with Buttons Bottom:



Adjust the class name to `ConfigurationDialog`, leave the rest of the settings at their default values, and complete the wizard. The files that appear in the project (`.cpp`, `.h`, and `.ui`) are very similar to the files generated for the `MainWindow` class when we created our project. The only difference is that `MainWindow` uses `QMainWindow` as its base class, and `ConfigurationDialog` uses `QDialog`. Also, a `MainWindow` instance is created in the `main` function, so it shows when the application is started, while we'll need to create a `ConfigurationDialog` instance somewhere else in the code. `QDialog` implements behavior that is common for dialogs; in addition to the main content, it displays one or multiple buttons. When the dialog is selected, the user can interact with the dialog and then press one of the buttons. After this, the dialog is usually destroyed. `QDialog` has a convenient `exec()` method that doesn't return until the user makes a choice, and then it returns information about the pressed button. We will see that in action after we finish creating the dialog.

Drag and drop two labels and two line edits on the form, position them roughly in a grid, double-click on each of the labels, and adjust their captions to receive a result similar to the following:



Change the `objectName` property of the line edits to `player1Name` and `player2Name`. Then, click on some empty space in the form and choose the Layout in a grid entry in the upper toolbar. You should see the widgets snap into place—that's because you have just applied a layout to the form. Open the Tools menu, go to the Form Editor submenu, and choose the Preview entry to preview the form.

Accelerators and label buddies

Now, we will focus on giving the dialog some more polish. The first thing we will do is add accelerators to our widgets. These are keyboard shortcuts that, when activated, cause particular widgets to gain keyboard focus or perform a predetermined action (for example, toggle a checkbox or push a button). Accelerators are usually marked by underlining them, as follows:

We will set accelerators to our line edits so that when the user activates an accelerator for the first field, it will gain focus. Through this, we can enter the name of the first player, and, similarly, when the accelerator for the second line edit is triggered, we can start typing in the name for the second player.

Start by selecting the first label on the left-hand side of the first line edit. Press *F2* and change the text to `Player &A Name:`. The `&` character marks the character directly after it as an accelerator for the widget. Accelerators may not work with digits on some platforms, so we decided to use a letter instead. Similarly, rename the second label to `Player &B Name:`.

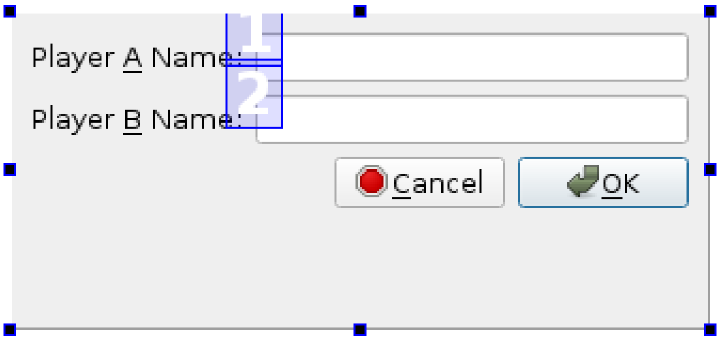
For widgets that are composed of both text and the actual functionality (for example, a button), this is enough to make accelerators work. However, since `QLineEdit` does not have any text associated with it, we have to use a separate widget for that. This is why we have set the accelerator on the label. Now we need to associate the label with the line edit so that the activation of the label's accelerator will forward it to the widget of our choice. This is done by setting a so-called **buddy** for the label. You can do this in code using the `setBuddy` method of the `QLabel` class or using Creator's form designer. Since we're already in the Design mode, we'll use the latter approach. For that, we need to activate a dedicated mode in the form designer.

Look at the upper part of Creator's window; directly above the form, you will find a toolbar containing a couple of icons. Click on the one labeled Edit buddies. Now, move the mouse cursor over the label, press the mouse button, and drag from the label toward the line edit. When you drag the label over the line edit, you'll see a graphical visualization of a connection being set between the label and the line edit. If you release the button now, the association will be made permanent. You should note that when such an association is made, the ampersand character (&) vanishes from the label, and the character behind it gets an underscore. Repeat this for the other label and corresponding line edit. Click on the Edit widgets button above the form to return the form editor to the default mode. Now, you can preview the form again and check whether accelerators work as expected; pressing *Alt* + *A* and *Alt* + *B* should set the text cursor to the first and second text field, respectively.

The tab order

While you're previewing the form, you can check another aspect of the UI design. Note which line edit receives the focus when the form is open. There is a chance that the second line edit will be activated first. To check and modify the order of focus, close the preview and switch to the tab order editing mode by clicking on the icon called Edit Tab Order in the toolbar.

This mode associates a box with a number to each focusable widget. By clicking on the rectangle in the order you wish the widgets to gain focus, you can reorder values, thus re-ordering focus. Now make it so that the order is as shown here:



Our form only has two widgets that can receive focus (except for the dialog's buttons, but their tab order is managed automatically). If you create a form with multiple controls, there is a good chance that when you press the *Tab* key repeatedly, the focus will start jumping back and forth between buttons and line edits instead of a linear progress from top to bottom (which is an intuitive order for this particular dialog). You can use this mode to correct the tab order.

Enter the preview again and check whether the focus changes according to what you've set.

When deciding about the tab order, it is good to consider which fields in the dialog are mandatory and which are optional. It is a good idea to allow the user to tab through all the mandatory fields first, then to the dialog confirmation button (for example, one that says OK or Accept), and then cycle through all the optional fields. Thanks to this, the user will be able to quickly fill all the mandatory fields and accept the dialog without the need to cycle through all the optional fields that the user wants to leave as their default values.

Time for action – Public interface of the dialog

The next thing to do is to allow to store and read player names from outside the dialog—since the `ui` component is private, there is no access to it from outside the class code. This is a common situation and one that Qt is also compliant with. Each data field in almost every Qt class is private and may contain accessors (a getter and optionally a setter), which are public methods that allow us to read and store values for data fields. Our dialog has two such fields—the names for the two players.

Names of setter methods in Qt are usually started with `set`, followed by the name of the property with the first letter converted to uppercase. In our situation, the two setters will be called `setPlayer1Name` and `setPlayer2Name`, and they will both accept `QString` and return `void`. Declare them in the class header, as shown in the following code snippet:

```
void setPlayer1Name(const QString &p1name);
void setPlayer2Name(const QString &p2name);
```

Implement their bodies in the `.cpp` file:

```
void ConfigurationDialog::setPlayer1Name(const QString &p1name)
{
    ui->player1Name->setText(p1name);
}
void ConfigurationDialog::setPlayer2Name(const QString &p2name)
{
    ui->player2Name->setText(p2name);
}
```

Getter methods in Qt are usually called the same as the property that they are related to—`player1Name` and `player2Name`. Put the following code in the header file:

```
QString player1Name() const;
QString player2Name() const;
```

Put the following code in the implementation file:

```
QString ConfigurationDialog::player1Name() const
{
    return ui->player1Name->text();
}
QString ConfigurationDialog::player2Name() const
{
    return ui->player2Name->text();
}
```

Our dialog is now ready. Let's use it in the `MainWindow::startNewGame` function to request player names before starting the game:

```
ConfigurationDialog dialog(this);
if(dialog.exec() == QDialog::Rejected) {
    return; // do nothing if dialog rejected
}
ui->player1Name->setText(dialog.player1Name());
ui->player2Name->setText(dialog.player2Name());
ui->gameBoard->initNewGame();
```

In this slot, we create the settings dialog and show it to the user, forcing them to enter player names. The `exec()` function doesn't return until the dialog is accepted or cancelled. If the dialog was canceled, we abandon the creation of a new game. Otherwise, we ask the dialog for player names and set them on appropriate labels. Finally, we initialize the board so that users can play the game. The dialog object was created without the `new` keyword, so it will be deleted immediately after this.

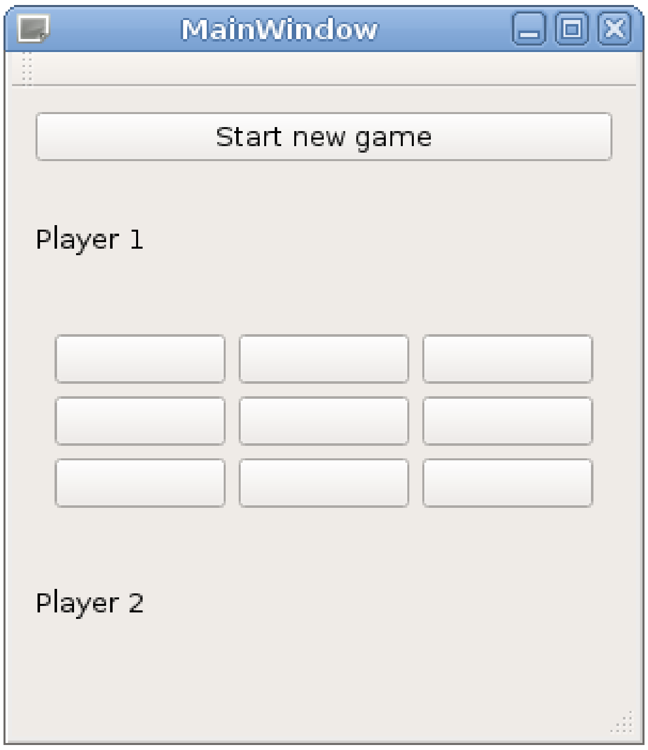
Now you can run the application and see how the configuration dialog works.

Polishing the application

We have implemented all the important functionalities of our game, and now we will start improving it by exploring other Qt features.

Size policies

If you change the height of the main window of our game, you will note that different widgets are resized in a different way. In particular, buttons retain their original height, and labels gain empty fields to the top and bottom of the text:



This is because each widget has a property called `sizePolicy`, which decides how a widget is to be resized by a layout. You can set separate size policies for horizontal and vertical directions. A button has a vertical size policy of `Fixed` by default, which means that the height of the widget will not change from the default height regardless of how much space there is available. A label has a `Preferred` size policy by default. The following are the available size policies:

- `Ignored`: In this, the default size of the widget is ignored and the widget can freely grow and shrink
- `Fixed`: In this, the default size is the only allowed size of the widget
- `Preferred`: In this, the default size is the desired size, but both smaller and bigger sizes are acceptable
- `Minimum`: In this, the default size is the smallest acceptable size for the widget, but the widget can be made larger without hurting its functionality
- `Maximum`: In this, the default size is the largest size of the widget, and the widget can be shrunk (even to nothing) without hurting its functionality
- `Expanding`: In this, the default size is the desired size; a smaller size (even zero) is acceptable, but the widget is able to increase its usefulness when more and more space is assigned to it
- `MinimumExpanding`: This is a combination of `Minimum` and `Expanding`—the widget is greedy in terms of space, and it cannot be made smaller than its default size

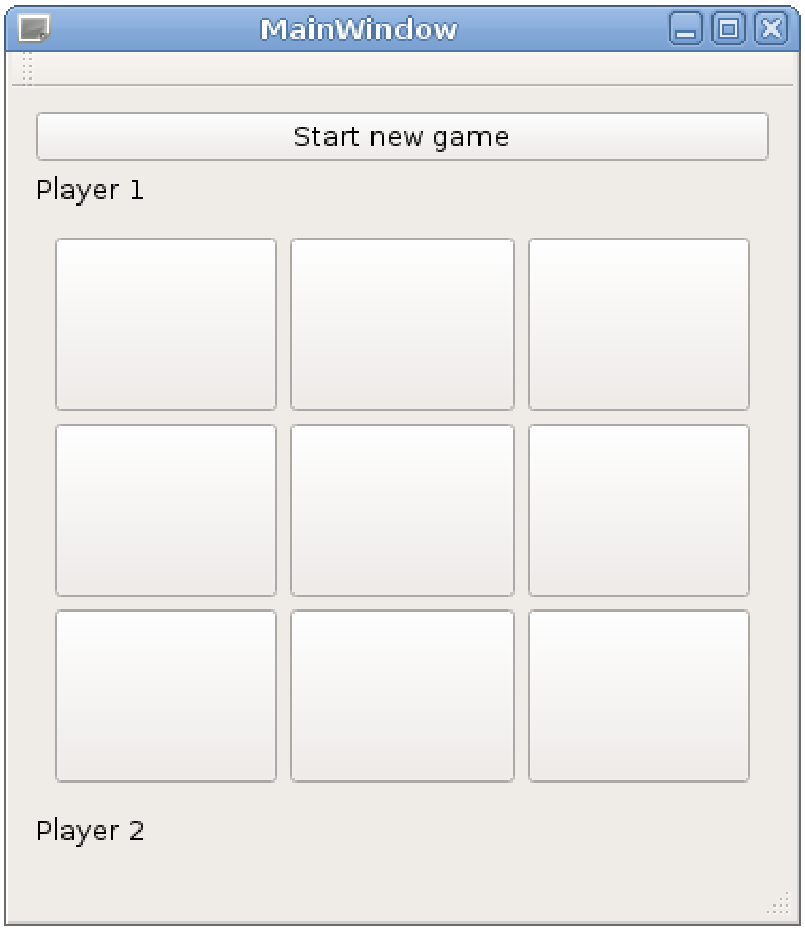
How do we determine the default size? The answer is by the size returned by the `sizeHint` virtual method. For layouts, the size is calculated based on the sizes and size policies of their child widgets and nested layouts. For basic widgets, the value returned by `sizeHint` depends on the content of the widget. In the case of a button, if it holds a line of text and an icon, `sizeHint` will return the size that is required to fully encompass the text, icon, some space between them, the button frame, and the padding between the frame and content itself.

In our form, we prefer that when the main window is resized, the labels will keep their height, and the game board buttons will grow. To do this, open `mainwindow.ui` in the form editor, select the first label, and then hold *Ctrl* and click on the second label. Now both labels are selected, so we can edit their properties at the same time. Locate `sizePolicy` in the property editor (if you're having trouble locating a property, use the Filter field above the property editor) and expand it by clicking on the triangle to its left. Set Vertical Policy to Fixed. You will see the changes in the form's layout immediately.

The buttons on the game board are created in the code, so navigate to the constructor of `TicTacToeWidget` class and set the size policy using the following code:

```
QPushButton *button = new QPushButton(" ");
button->setSizePolicy(QSizePolicy::Preferred,
                      QSizePolicy::Preferred);
```

This will change both the horizontal and vertical policy of buttons to `Preferred`. Run the game and observe the changes:



Protecting against invalid input

The configuration dialog did not have any validation until now. Let's make it such that the button to accept the dialog is only enabled when neither of the two line edits is empty (that is, when both the fields contain player names). To do this, we need to connect the `textChanged` signal of each line edit to a slot that will perform the task.

First, go to the `configurationdialog.h` file and create a private slot `void updateOKButtonState();` in the `ConfigurationDialog` class (you will need to add the `private slots` section manually). Use the following code to implement this slot:

```
void ConfigurationDialog::updateOKButtonState()
{
    QPushButton *okButton = ui->buttonBox->button(QDialogButtonBox::Ok);
    okButton->setEnabled(!ui->player1Name->text().isEmpty() &&
                         !ui->player2Name->text().isEmpty());
}
```

This code asks the button box that currently contains the OK and Cancel buttons to give a pointer to the button that accepts the dialog (we have to do that because the buttons are not contained in the form directly, so there are no fields for them in `ui`). Then, we set the button's `enabled` property based on whether both player names contain valid values or not.

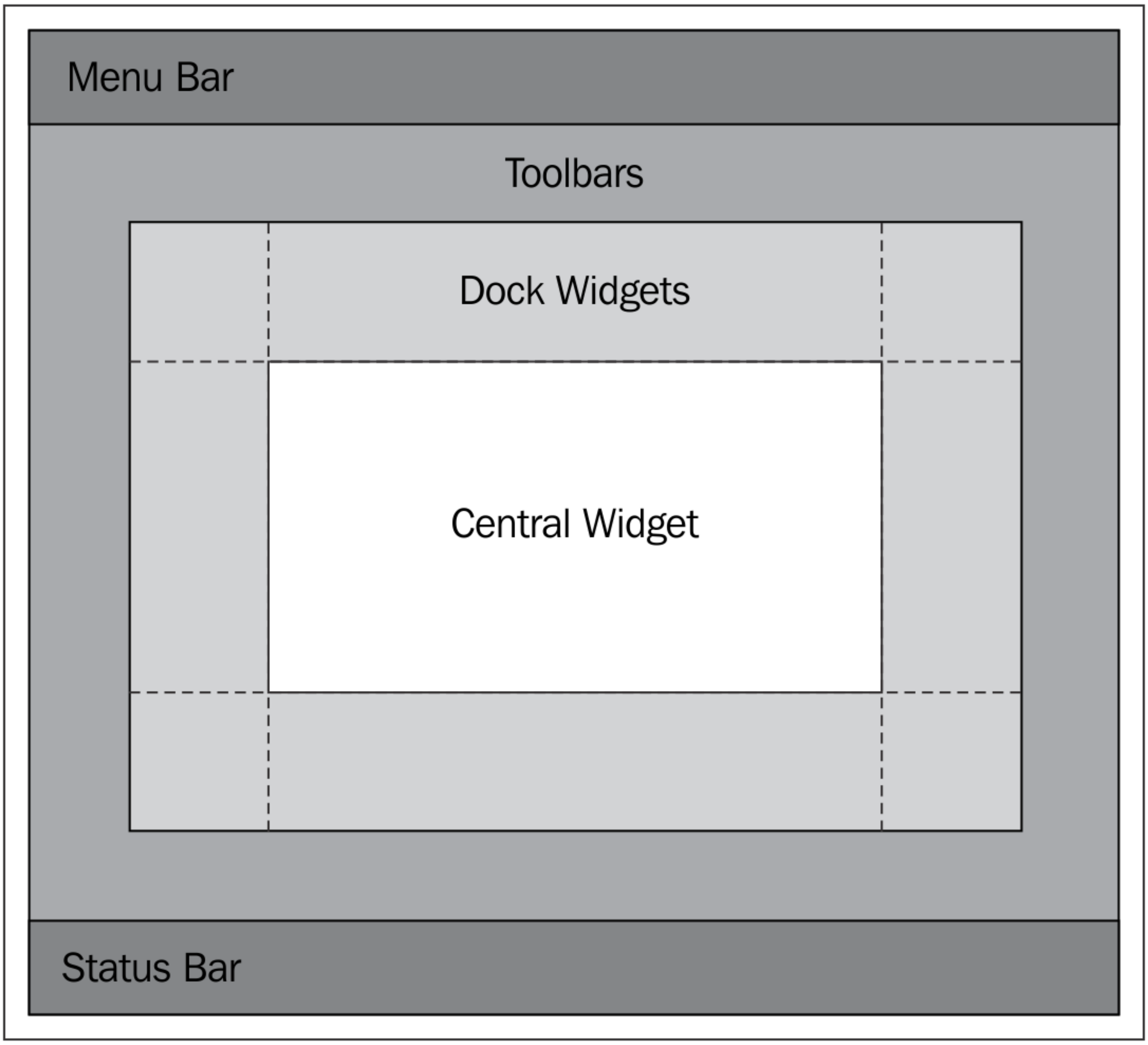
Next, edit the constructor of the dialog to connect two signals to our new slot. The button state also needs to be updated when we first create the dialog, so add an invocation of `updateOKButtonState()` to the constructor:

```
ui->setupUi(this);
connect(ui->player1Name, &QLineEdit::textChanged,
        this, &ConfigurationDialog::updateOKButtonState);
connect(ui->player2Name, &QLineEdit::textChanged,
        this, &ConfigurationDialog::updateOKButtonState);
updateOKButtonState();
```

Main menu and toolbars

As you may remember, any widget that has no parent will be displayed as a window. However, when we created our main window, we selected `QMainWindow` as the base class. If we had selected `QWidget` instead, we would still be able to do everything we did up to this point. However, the `QMainWindow` class provides some unique functionality that we will now use.

A main window represents the control center of an application. It can contain menus, toolbars, docking widgets, a status bar, and the *central widget* that contains the main content of the window, as shown in the following diagram:



If you open the `mainwindow.ui` file and take a look at the object tree, you will see the mandatory `centralWidget` that actually contains our form. There are also optional `menuBar` and `statusBar` that were added automatically when Qt Creator generated the form.

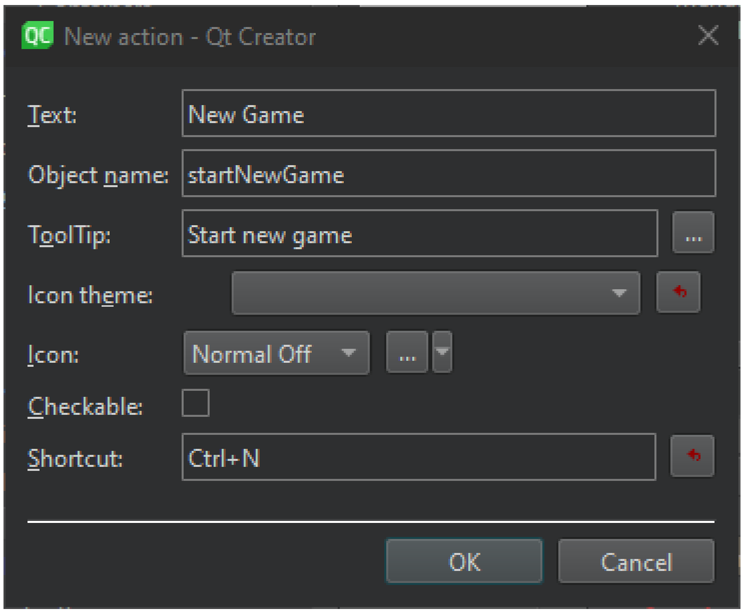
The central widget part doesn't need any extra explanation; it is a regular widget like any other. We will also not focus on dock widgets or the status bar here. They are useful components, but you can learn about them yourself. Instead, we will spend some time mastering menus and toolbars. You have surely seen and used toolbars and menus in many applications, and you know how important they are for a good user experience.

The main menu has a bit of unusual behavior. It's usually positioned in the top part of the window, but in macOS and some Linux environments, the main menu is separated from the window and displayed in the top area of the screen. Toolbars, on the other hand, can be moved freely by the user and docked horizontally or vertically to the sides of the main window.

The main class shared by both these concepts is `QAction`, which represents a functionality that can be invoked by a user. A single action can be used in multiple places—it can be an entry in a menu (the `QMenu` instances) or in a toolbar (`QToolBar`), a button, or a keyboard shortcut (`QShortcut`). Manipulating the action (for example, changing its text) causes all its incarnations to update. For example, if you have a Save entry in the menu (with a keyboard shortcut bound to it), a Save icon in the toolbar, and maybe also a Save button somewhere else in your user interface and you want to disallow saving the document (for example, a map in your dungeons and dragons game level editor) because its contents haven't changed since the document was last loaded. In this case, if the menu entry, toolbar icon, and button are all linked to the same `QAction` instance, then, once you set the `enabled` property of the action to `false`, all the three entities will become disabled as well. This is an easy way to keep different parts of your application in sync—if you disable an action object, you can be sure that all entries that trigger the functionality represented by the action are also disabled. Actions can be instantiated in code or created graphically using Action Editor in Qt Creator. An action can have different pieces of data associated with it—a text, tooltip, status bar tip, icons, and others that are less often used. All these are used by incarnations of your actions.

Time for action – Creating a menu and a toolbar

Let's replace our boring Start new game button with a menu entry and a toolbar icon. First, select the button and press the *Delete* key to delete it. Then, locate Action Editor in the bottom-center part of the form editor and click on the New button on its toolbar. Enter the following values in the dialog (you can fill the Shortcut field by pressing the key combination you want to use):


New action - Qt Creator

Text:	New Game
Object name:	startNewGame
ToolTip:	Start new game
Icon theme:	
Icon:	Normal Off
Checkable:	☐
Shortcut:	Ctrl+N

In the central area (between the Type Here text and the first label: "Player 1") open the context menu and select Add Tool Bar then drag the line containing the New Game action from the action editor to the toolbar, which results in a button appearing in the toolbar.

To create a menu for the window, double-click on the Type Here text on the top of the form and replace the text with `&File` (although our application doesn't work with files, we will follow this tradition). Then, drag the New Game action from the action editor over the newly created menu, but do not drop it there yet. The menu should open now, and you can drag the action so that a red bar appears in the submenu in the position where you want the menu entry to appear; now you can release the mouse button to create the entry.

Now we should restore the functionality that was broken when we deleted the button. Navigate to the constructor of the `MainWindow` class and adjust the `connect()` call:

```
connect(ui->startNewGame, &QAction::triggered,
        this, &MainWindow::startNewGame);
```

Actions, like widgets, are accessible through the `ui` object. The `ui->startNewGame` object is now a `QAction` instead of a `QPushButton`, and we use its `triggered()` signal to detect whether the action was selected in some way.

Now, if you run the application, you can select the menu entry, press a button on the toolbar, or press the *Ctrl* + *N* keys. Either of these operations will cause the action to emit the `triggered()` signal, and the game configuration dialog should appear.

Like widgets, `QAction` objects have some useful methods that are accessible in our form class. For example, executing `ui->startNewGame->setEnabled(false)` will disable all ways to trigger the New Game action.

Let's add another action for quitting the application (although the user can already do it just by closing the main window). Use the action editor to add a new action with text `Quit`, object name `quit`, and shortcut *Ctrl* + *Q*. Add it to the menu and the toolbar, like the first action.

We can add a new slot that stops the application, but such a slot already exists in `QApplication`, so let's just reuse it. Locate the constructor of our form in `mainwindow.cpp` and append the following code:

```
connect(ui->quit, &QAction::triggered,
        qApp,     &QApplication::quit);
```

What just happened?

The `qApp` macro is a shortcut for a function that returns a pointer to the application singleton object, so when the action is triggered, Qt will call the `quit()` slot on the `QApplication` object created in `main()`, which, in turn, will cause the application to end.

The Qt resource system

Buttons in the toolbar usually display icons instead of text. To implement this, we need to add icon files to our project and assign them to the actions we created.

One way of creating icons is by loading images from the filesystem. The problem with this is that you have to install a bunch of files along with your application, and you need to always know where they are located to be able to provide paths to access them. Fortunately, Qt provides a convenient and portable way to embed arbitrary files (such as images for icons) directly in the executable file. This is done by preparing resource files that are later compiled in the binary. Qt Creator provides a graphical tool for this as well.

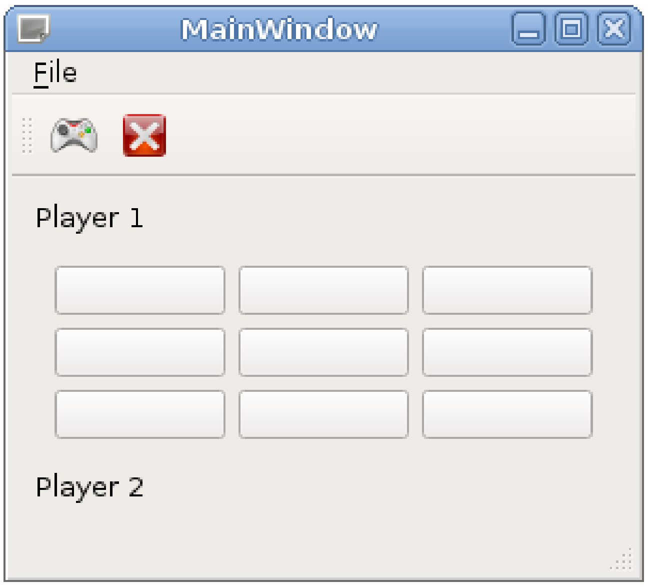
Time for action – Adding icons to the project

We will add icons to our Start new game and Quit actions. First, use your file manager to create a new subdirectory called `icons` in the project directory. Place two icon files in the `icons` directory. You can use icons from the files provided with the book.

Click on Add New... in the context menu of the tictactoe project and select Qt Resource File (located in Qt category). Name it `resources`, and finish the wizard. Qt Creator will add a new `resources.qrc` file to the project (it will be displayed under the Resources category in the project tree).

Locate the new `resources.qrc` file in the project tree of Qt Creator and choose Add Existing Files... in its context menu. Select both icons, and confirm their addition to the resources.

Open the `mainwindow.ui` form, and double-click on one of the actions in the action editor. Click on the "..." button next to the Icon field, select icons in the left part of the window, and select the appropriate icon in the right part of the window. Once you confirm changes in the dialogs, the corresponding button on the toolbar will switch to displaying the icon instead of the text. The menu entry will also gain the selected icon. Repeat this operation for the second action. Our game should now look like this:



Have a go hero – Extending the game

There are a lot of subtle improvements you can make in the project. For example, you can change the title of the main window (by editing its `windowTitle` property), add accelerators to the actions, disable the board buttons that do nothing on click, remove the status bar, or use it for displaying the game status.

As an additional exercise, you can try to modify the code we wrote in this chapter to allow playing the game on boards bigger than 3 × 3. Let the user decide the size of the board (you can modify the game options dialog for that and use `QSlider` and `QSpinBox` to allow the user to choose the size of the board), and you can then instruct `TicTacToeWidget` to build the board based on the size it gets. Remember to adjust the game-winning logic! If at any point you run into a dead end and do not know which classes and functions to use, consult the reference manual.

Pop quiz

Q1. Which classes can have signals?

1. All classes derived from `QWidget`.
2. All classes derived from `QObject`.
3. All classes.

Q2. For which of the following do you have to provide your own implementation?

1. A signal.
2. A slot.
3. Both.

Q3. A method that returns the preferred size of a widget is called which of these?

1. `preferredSize`.
2. `sizeHint`.
3. `defaultSize`.

Q4. What is the purpose of the `QAction` object?

1. It represents a functionality that a user can invoke in the program.
2. It holds a key sequence to move the focus on a widget.
3. It is a base class for all forms generated using the form editor.

Summary

In this chapter, you learned how to create simple graphical user interfaces with Qt. We went through two approaches: designing the user interface with a graphical tool that generates most of the code for us, and creating user interface classes by writing all the code directly. None of them is better than the other. The form designer allows you to avoid boilerplate code and helps you handle large forms with a lot of controls. On the other hand, the code writing approach gives you more control over the process and allows you to create automatically populated and dynamic interfaces.

We also learned how to use signals and slots in Qt. You should now be able to create simple user interfaces and fill them with logic by connecting signals to slots —predefined ones as well as custom ones that you now know how to define and fill with code.

Qt contains many widget types, but we didn't introduce them to you one by one. There is a really nice explanation of many widget types in the Qt manual called Qt Widget Gallery, which shows most of them in action. If you have any doubts about using any of those widgets, you can check the example code and also look up the appropriate class in the Qt reference manual to learn more about them.

As you already saw, Qt allows you to create custom widget classes, but in this chapter our custom classes mostly reused the default widgets. It's also possible to modify how the widget responds to events and implement custom painting. However, if you want to implement a game with custom 2D graphics, there is a simpler alternative —the Graphics View Framework that we'll use in the next chapter.