COMP2811 Lab 3: Introduction to Qt

Goals for this lab:

- Introduce the Qt UI toolkit
- Learn to create widgets
- Understand message passing in Qt
- Explore Qt to support courseworks

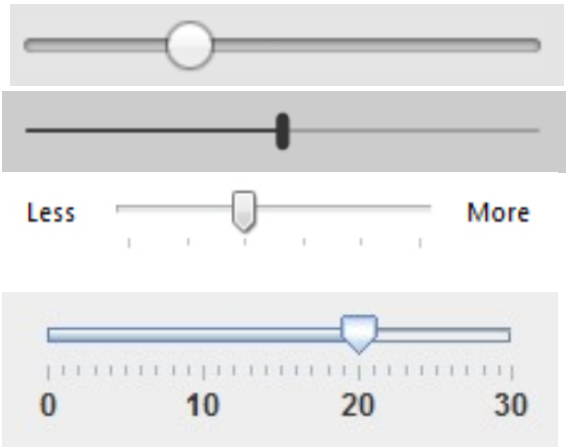
Definitions

- Toolkit (or Framework)
  - Library for GUI creation; Other examples include:
    - Windows 11 Universal Windows Platform
    - Java's Swing and FX
    - Apple's H.I.G.
    - Bootstrap or jQuery for the web.
  - Each of these has a different API and approach.

- Widget (or Component)
  - Visual element of a GUI; examples:

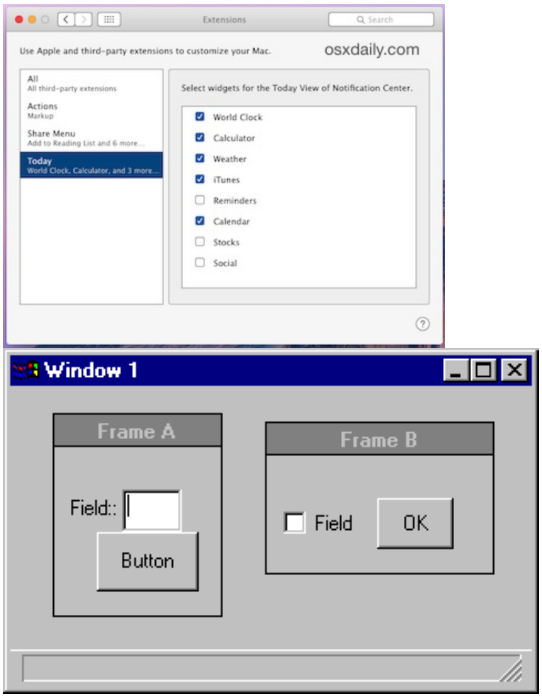
Button


Text Field

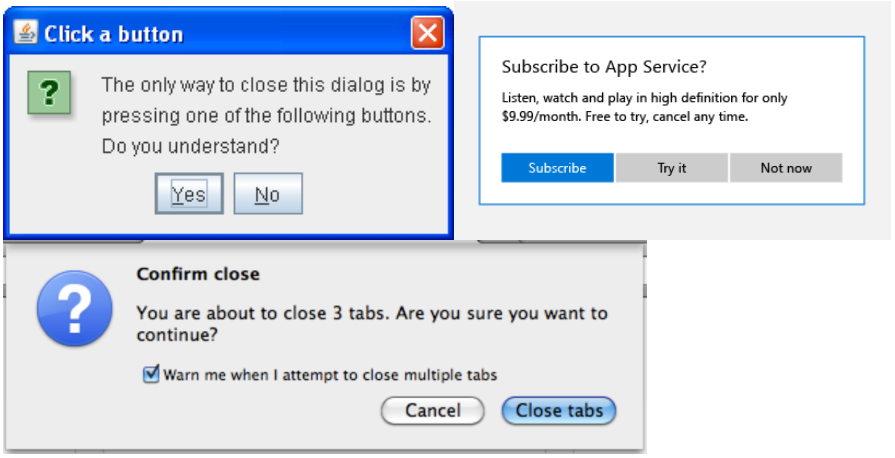

Slider


- Container
  - A widget whose role is to hold other widgets; examples:

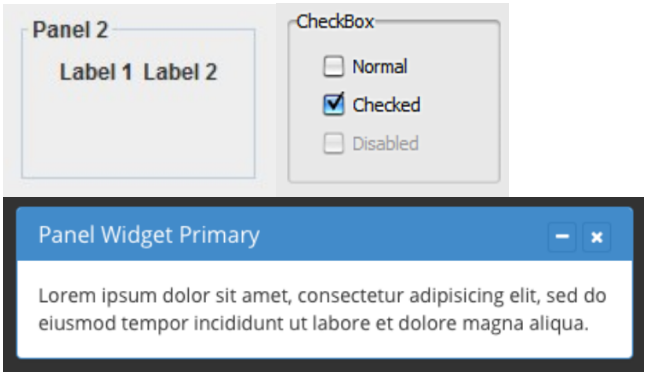
Windows



Dialog Box



Panel



The Qt toolkit

- A *framework* for *cross-platform* application development, emphasising but not exclusively focused on GUIs
- Dual-licensed for commercial and open source projects. We use the open-source license in 2811.
- Well-established (available since early 1990s)
- Used by the Linux KDE desktop, Google Earth, Sky Q, VirtualBox, VLC…
- See https://www.qt.io for more information
- Why are we using Qt in comp2811?
    - Cross platform (Linux, Android, Windows, and more)
    - It will be used in next year's *Introduction to Graphics* course

"Hello World!" in Qt

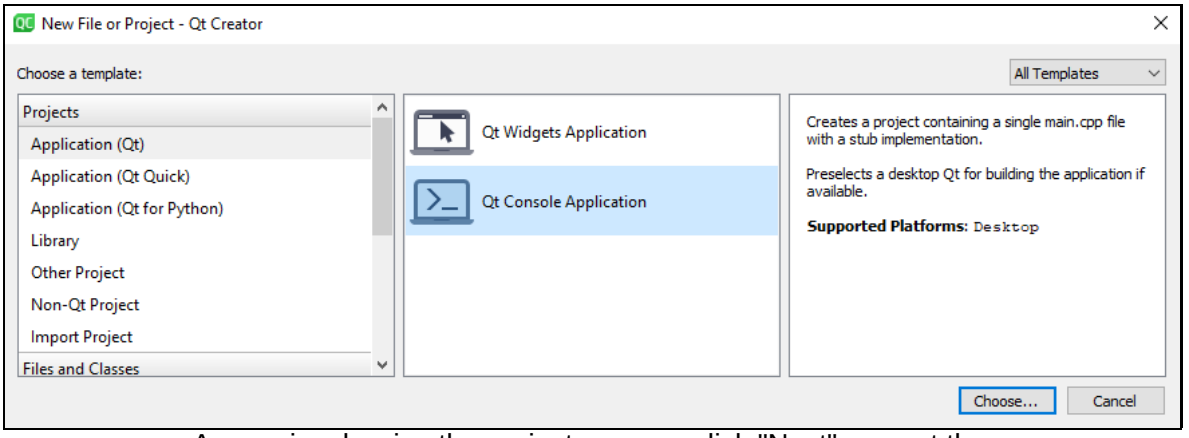
```
#include<QtWidgets>
int main(int argc, char* argv[])
{
     qDebug() << QT_VERSION_STR;
      QApplication app(argc, argv);
      QLabel* label = new QLabel("<h1>Hello World!
</h1>");
      label -> setFixedSize(QSize(500,500));
      label-> show();

      return app.exec();
}
```

Let's analyse this code:

- `#include<QtWidgets>`
    - need to include header for Qt classes
- `qDebug() << QT_VERSION_STR;`
    - print out the Qt version to the console (a good check that the library is installed and up to date).
- `QApplication app(argc, argv);`
    - a single QApplication object is created at the start of `main`, to manage resources and control interaction
- `QLabel* label = new QLabel("<h1>Hello World</h1>");`
    - note the use of pointer and creation of object with new
- `label -> setFixedSize(QSize(500,500));`
    - set the size of the label - the size is width and height in pixels.
    - note the use of the arrow operator (`->`) to call a function on a pointer to an object
- `label-> show();`
    - display the widget in a window
- `return app.exec();`
    - main should end with a call to `QApplication`'s `exec` function, to start the *main event loop.*

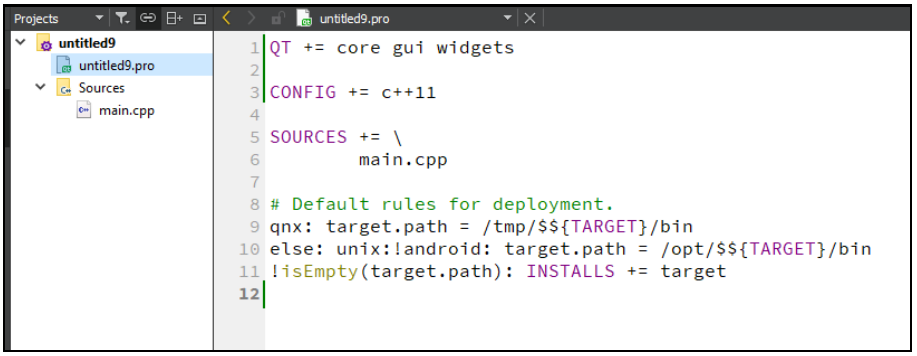
Exercise 1. Follow these instructions:

1. Start Qt Creator.
2. We will use the console template to give us a clean project configuration:
    a. go "File" → "New Project" as previously
    b. On the next screen select "Application (Qt)" and "Qt Console Application", then "Choose…".



c. As previously, give the project a name, click "Next", accept the default "qmake" build system, Translation File, Qt Kit, and Project Management settings by clicking "Next" four more times.

3. The *.pro* file is the description used by qmake to build our executable.
   a. Open and update the .pro file to add support for a UI.
   It will be named after your project name:



   b. We should include the Qt Modules for creating GUIs and Widgets.
   Set this content (if you copy and paste from here, delete any blank lines after `SOURCES += \`):

```
QT += core gui widgets

CONFIG += c++11

SOURCES += \
        main.cpp

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

4. Delete the code already in *main.cpp*, and paste the *Hello World* code from above.
5. Run your program. If there are problems, try doing a *clean* ("Build" menu → "Clean project XXX") before running again.
6. Your running program should show something like the following in a new window (if it isn't visible, check the taskbar/*Alt-Tab* for the new window):

7. Change the program so that the label says something rude.
8. What happens if you create the `QLabel` before the `QApplication`?

Event loops

- These are sometimes referred to as the ‘main loop’
- They are entered after constructing a GUI and making it visible
- The program runs a loop which:
  - waits for events
  - dispatches these events to appropriate pieces of event handling code
  - if it receives a quit event then delete resources and quit…
  - ...otherwise start the loop again
- Examples of events:
  - mouse click
  - key press
  - message from the network
- *Semantic events* (button clicks, menu selection, etc) are handled using a *signals* and *slots* mechanism
- The `app.exec()` code in the previous exercise starts the event-loop.

Creating widget Instances

- Widgets are *objects* (instances of classes)
- Recall that C++ objects can be created using automatic allocation:

```
foo f;
```

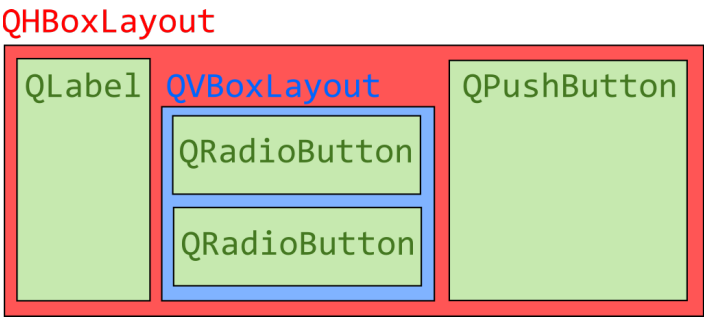
- ...or they can be created using dynamic allocation:

```
foo* f = new foo();
 ...
delete f; // to release storage
```

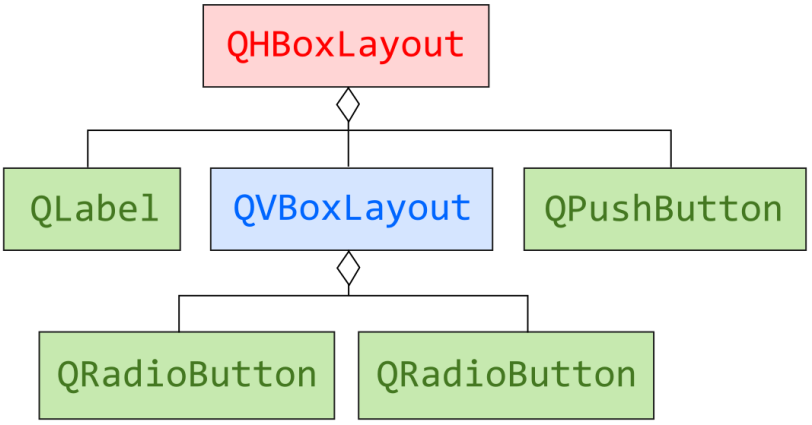
- Normal practice is to use dynamic allocation for Qt widgets and manipulate them using *pointers*

Pointers allow representation of the user-interface as an object tree:

- Containers hold pointers to widgets inside them:



  - In the above example, we have a panel with a box layout (`QHBoxLayout`)...
  - ...inside of which are a label (`QLabel`), a button (`QPushButton`), and another layout (`QVBoxLayout`) ...
  - ...inside that layout there are two radio buttons (`QRadioButton`):



    - Each of these (`QHBoxLayout, QLabel, QPushButton` etc...) are C++ objects with pointers to the widgets inside them.
- When the parent is `deleted`, it walks through its list of children and deletes each in turn; children `delete` their own children, as well as other other heap resources.
- The children are also used when a widget is enabled / disabled
    - An example of an enabled cancel button, and a disabled apply button:

    - When a parent widget is disabled, it disables all of its children.
    - When a parent widget is enabled, it enables all of its children.

Parent-child relationships

- These can be established during widget creation:

```
QLabel* label = new QLabel(parentWidget);
```

- Or, can be established when a widget is added to a container, via the `addWidget` method:

```
QLabel* label = new QLabel();
label -> setParent(parentWidget);
```

- (recall that the `->` is a C++ operator that calls a method on a pointer to an object)

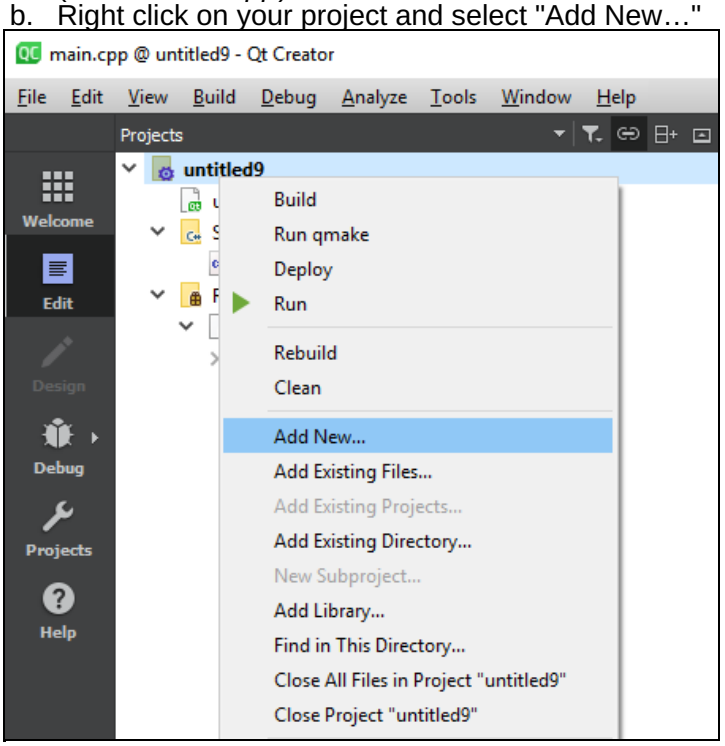
Exercise 2. Complete these steps

1. You may find it useful to have the Qt Widgets documentation open in a separate browser tab before you begin.
2. Overwrite the previous *main.cpp* or setup a new project and *.pro* file as above. (Remember you can only have one `main` function in a project.)
3. Copy the following code into *main.cpp:*

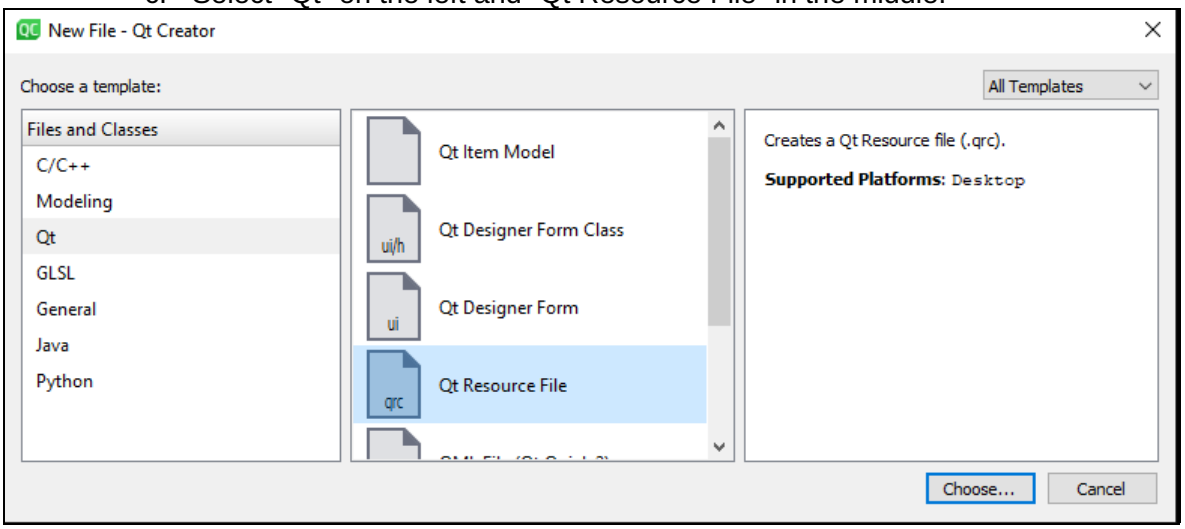
```
#include<QtWidgets>
int main(int argc, char* argv[])
{
      QApplication app(argc, argv);
      QPushButton* widget = new QPushButton("Click
Here");
      widget-> show();
```

```
    return app.exec();
}
```

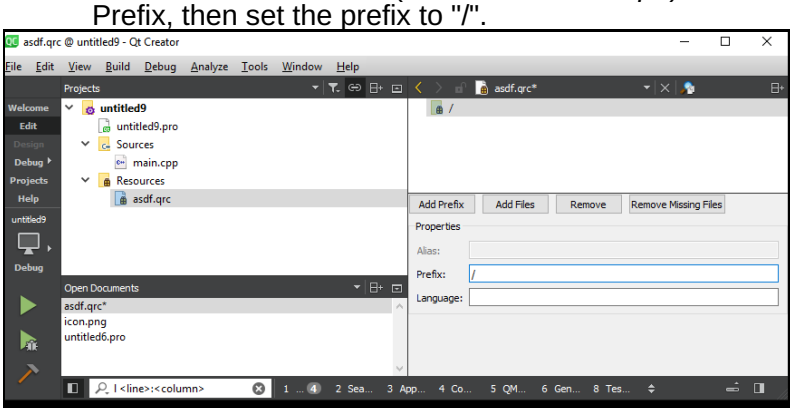
4. Let's now decorate the button:
    a. From the resource zip copy *icon.png* into your source folder (next to *main.cpp*).
    b. Right click on your project and select "Add New…"



    c. Select "Qt" on the left and "Qt Resource File" in the middle:



    d. Add a name for the resource files, but don't change the location.
    Click "Finish" on the Project Management screen.
    e. Select the resource file (its name ends in *.qrc*), select Add Prefix, then set the prefix to "/".



f. Within the resource file you can now select "Add Files" and select your *icon.png.*
g. Note that your new resource file has been automatically added to your *.pro* file.
h. Then add the following line to the main program, after the line creating the `QPushButton` object and before the line calling its `show` function:

```
widget->setIcon(QIcon(":/icon.png"));
```

i. This line creates a `QIcon` object from the given icon file and then immediately passes this object to the widget's `setIcon` function, thereby ensuring that the icon will be used by the widget.
j. Recompile using the make command, then run.
k. Try changing the font used for button text by adding a line like the one below to the main program (assuming that `"Times"` is a font on your system).

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

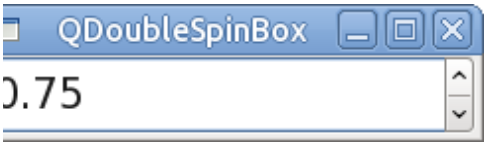
l. Recompile and rerun as before.
m. Add a line that calls `setWindowTitle` on `widget`, setting the title to "QPushButton". Then add another line that calls `setMinimumSize`, specifying 240 and 60 as the width and height arguments. Both of these function calls should be placed before the line that calls the widget's show function.
n. Now run the program, and you should see something like this:

h. Try disabling the button. You will need to call the `setEnabled` function on the widget, with an argument of `false`.
i. Remove or comment out the code that disables the button, then add the following to the program, just before the call to the button's `show` function:

```
QObject::connect(widget, SIGNAL(clicked()), &app,
SLOT(quit()));
```

j. Run your program. Click the button and the program should quit. This is an example of the *signals and slots* that we will learn about next.

5. Now let's create a spinner widget:
   a. Create a new file with a `main` function (you may need to rename your old `main` function). In this file, write a program that creates a `QDoubleSpinBox` widget. As in the previous exercise, this widget should act as the application's window. You can read the class' API here.
   b. Follow the same steps as before to create files into your project. Fix any compilation errors before continuing.
   c. Configure your `QDoubleSpinBox` widget in the following ways:
      i. Use `setRange` to set lower and upper limits of 0.0 and 1.0, respectively.
      ii. Use `setSingleStep` to specify a step size of 0.05.
      iii. Use `setFont` as above to specify 16-point as the font.
      iv. Use `setMinimumSize` to specify a minimum width and height of 250 pixels and 40 pixels, respectively.
      v. Use `setWindowTitle` to specify the window title as "QDoubleSpinBox".
   d. Your application now should look something like this when run:



e. Try entering values into the spin box. Try altering the value using the small buttons at the right of the spin box.

6. Now let's create a single-line text input:
   a. Create a new file with a `main` function (you may need to rename your old main function). In this file, write a program that creates a `QLineEdit` widget. As in the previous exercise, this widget should act as the application's window. Add a line that calls `setMinimumSize` on the widget, specifying 200 and 40 as the arguments of the function call.
   b. Run the program and you should see a small window containing an input field suitable for entering a single line of text.
   c. Try changing text alignment. You can do this using the `setAlignment` function and one of the alignment constants, e.g.

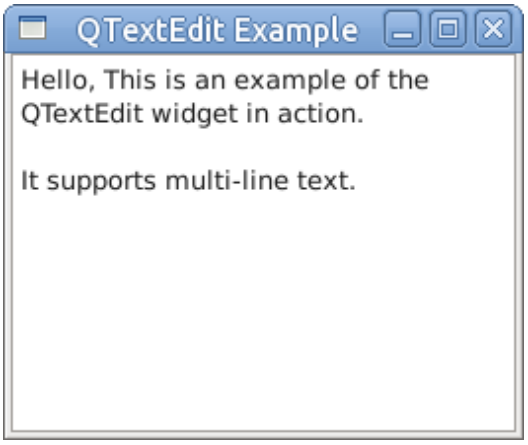
```
widget->setAlignment(Qt::AlignCenter);
```

   d. (Note: This assumes that `widget` is the name of your `QLineEdit` variable, and that it is a pointer variable.)
   e. Try disabling text input by calling `setEnabled` on the object, with an argument of `false`. Rerun the program and try entering some text.
   You should find that this isn't possible.
   f. Remove the call to `setEnabled`.
   Try restricting input to three-digit integers, using an *input mask*:
   g. You can do this by calling `setInputMask` on the object and using `"999"` as the mask.
   The `QLineEdit` documentation has details of what this means and how masks are specified.
   (Click on `inputMask` in the list of properties to see this information.)
   h. Run the program again and try entering different kinds of input.
   i. Remove or comment out the call to `setInputMask`.
   Set a *validator* that limits input to an integer in the range 1-20:

```
QIntValidator* validator = new QIntValidator(1, 99);
widget->setValidator(validator);
```

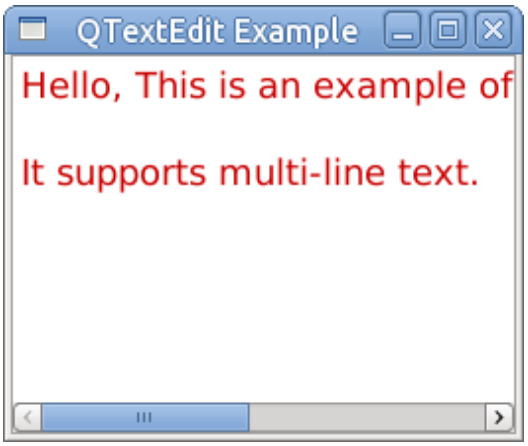
   j. Run the program again and verify that input is restricted to integers in this range.

7. Finally, let's play with a multi-line text input widget:
   a. Study the `QTextEdit` documentation, then write a small program in C++ that creates and displays a `QTextEdit` widget in a window by itself. Use the widget's `setWindowTitle` function to give the window a title of "QTextEdit Example". The final result should look something like this:



b. Try resizing the window, both horizontally and vertically. Notice how the text rewraps in response to changing dimensions, and how scroll bars appear when necessary.

c. Call functions on your widget to change the font to something different (e.g., 14-point "DejaVu Sans") and change the text colour to a shade of red. Add another function call to disable word wrapping. You should end up with something like this:



- Hint 1*:* you will need to study the documentation for `QColor`, `QFont` and `QTextOption` to determine which function calls to use.
- Hint 2: set the text colour and font *before* setting the text to show using the `setText` function.

Widget configuration

- The configuration of a widget describes its properties, for example
  - a button's text
  - a slider's value
  - whether a checkbox is selected

- We can change the configuration of widgets in 3 different ways:
  1. With constructor parameters:

```
QLabel* label = new QLabel("Name");
```

  2. With function calls on the created widget:

```
QPushButton* button = new QPushButton();
button->setIcon(QIcon(":/icon.png"));
```

  3. Declaratively, with style sheets when starting the application:

```
> ./app -stylesheet app.qss
```

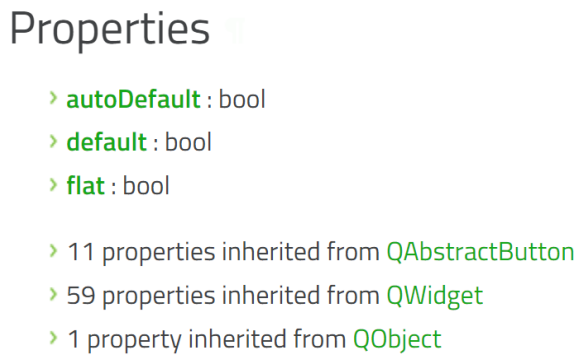
  where *app.qss* contains the configuration:

```
QLineEdit {
      font: bold 16pt;
      color: red;
}
```

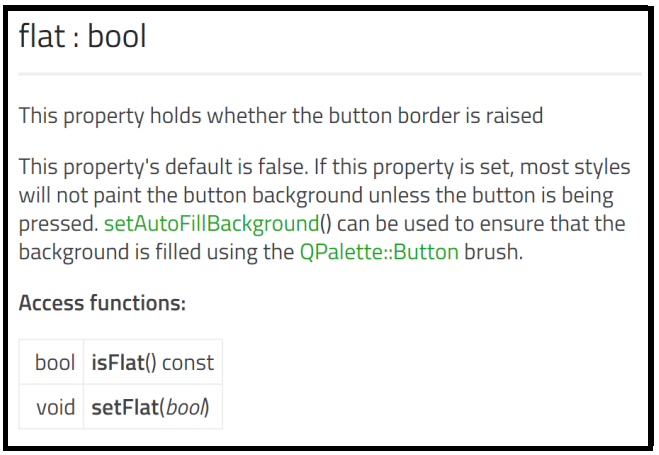
4. You can set the stylesheet in code with the function:

```
widget->setStyleSheet("font: bold 16pt; color: red; ");
```

- A list of properties for each widget can be found online. For example, the API for QPushButton shows 74 properties, some of which are inherited from parent classes `QAbstractButton`, `QWidget` and `QObject`:



- This is a lot of properties! But don't worry, we'll only use a few.
- We can see how to get and set a property by reading the API. For example, if we click on *flat*, we see:



- By reading the above, we see that if we have a QPushButton

```
QPushButton* button = new QPushButton("push me");
```

  - we can find out if the flat property is set using:

```
bool foo = button -> isFlat();
```

  - and can set the flat property using:

```
button -> setFlat(true);
```

Exercise 3. Follow these instructions to explore some simple Qt projects.

1. Explore the projects in the *eg* folder of the resource zip.
2. You should be able to go "File" → "Open File or Project..." and select one of the *.pro* files. You might have to select which Qt Kit to use (but there is usually only one) and click "Configure Project".
3. Open 4 examples (you can ignore the readme files). For each of the examples:
   a. Run the project. You might have to make some small changes.
   b. Read the source code.

    c. Identify the properties set and read on each widget.
    d. Read the Qt documentation to understand what each property does.
4. Remember to look at these examples later if you need to see example code for a widget!

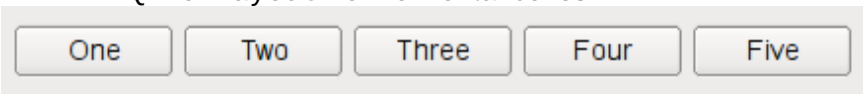
Widget	**filename**
`QLabel`	*label, pixmap, gif*
`QLCDNumber`	*number*
`QPushButton`	*button*
`QComboBox`	*combo*
`QLineEdit`	*textfield, postcode*
`QTextBrowser`	*browser*
`QSlider`	*slider*
`QSpinBox`	*spinbox*
`QScrollArea`	*imgview*
`QTabWidget`	*tabs*

Qt containers

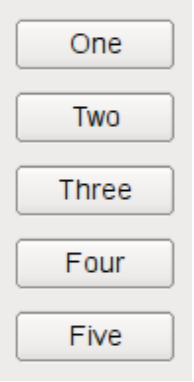
- Recall that a container is a widget that contains other widgets.
- Here are some common Qt containers:
    - `QWidget` - superclass of all widgets; can also act as a generic container for other widgets
    - `QFrame` - like `QWidget` but with an optional frame
    - `QMainWindow` - an application window with menus, etc
    - `QDialog` - a secondary window (child of main window), typically on screen for a short time
    - `QMessageBox` - a simple modal dialog for display of messages or confirmation of actions

Layout classes

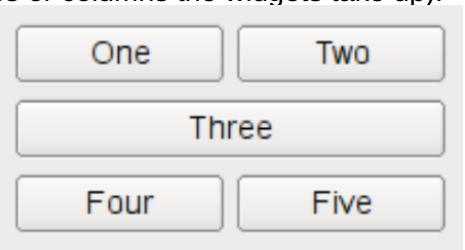
- These act as containers with a built-in layout algorithm
- There are several options provided by Qt
    - `QHBoxLayout` for horizontal boxes:



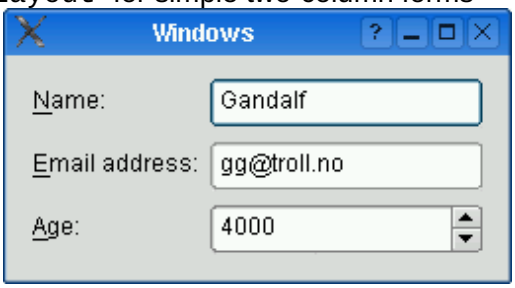
    - `QVBoxLayout` for vertical boxes:



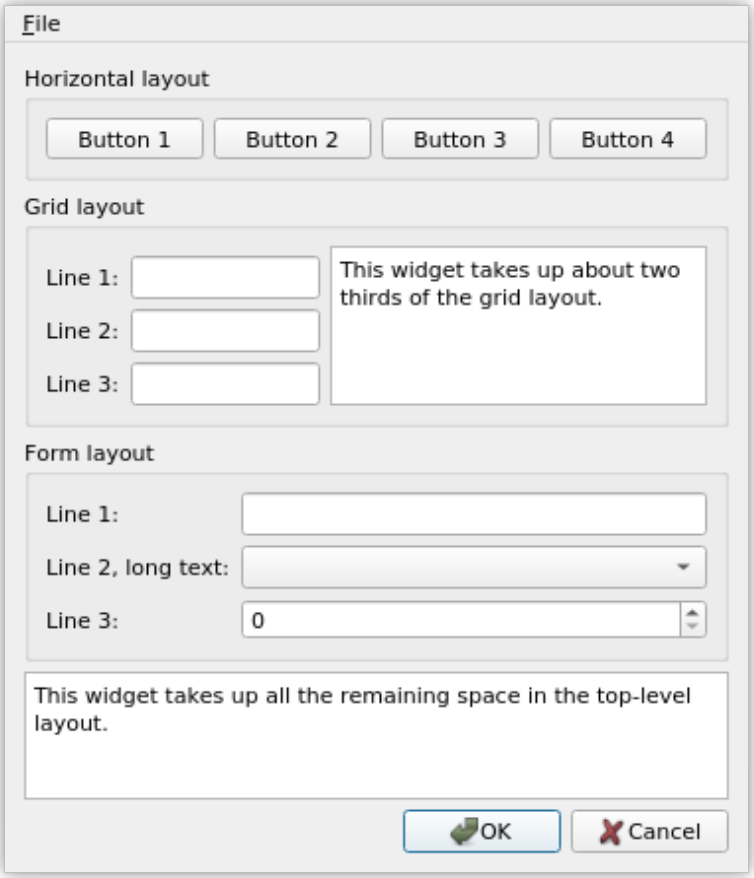
- `QGridLayout` for a 2D grid of widgets (you can specify how many rows or columns the widgets take up):



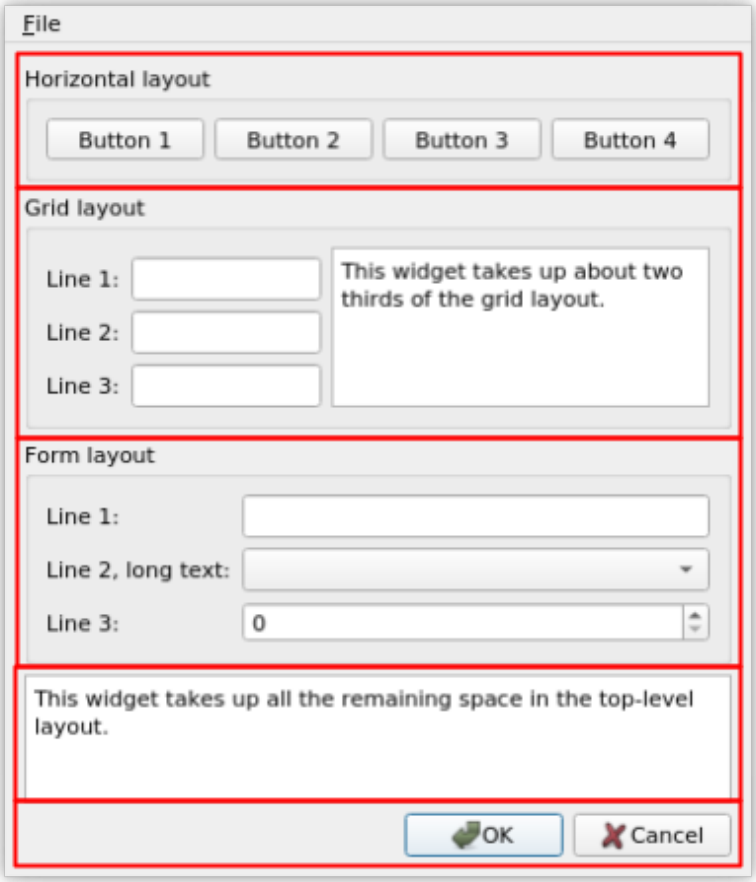
- `QFormLayout` for simple two-column forms



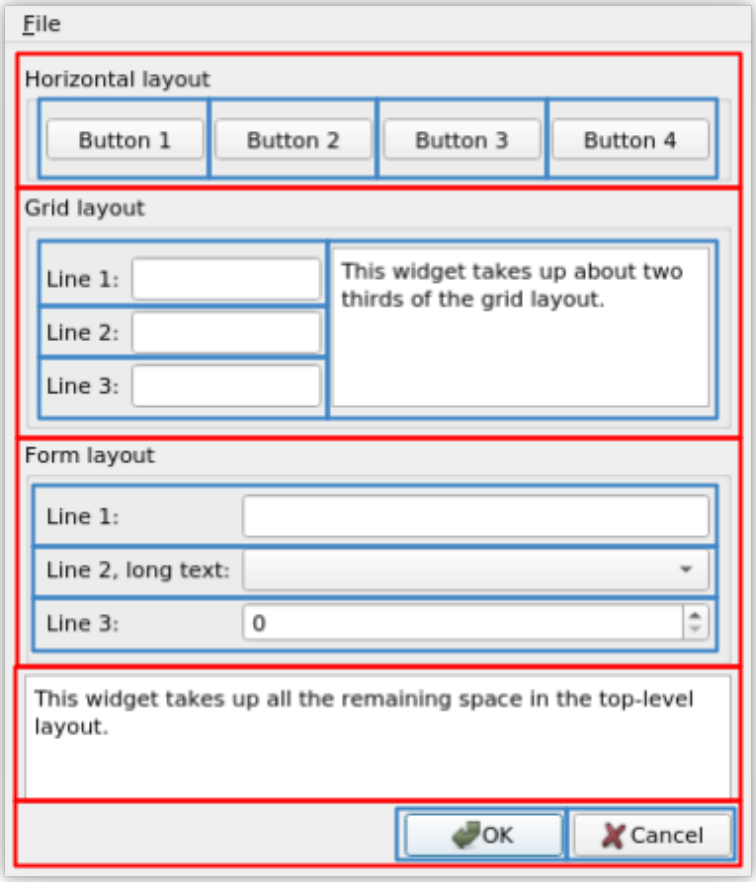
- Remember that most of the *containers* are subclasses of `QWidget`! Therefore, more complex layouts can be achieved by nesting simple layouts inside each other:



- Here a `QVBoxLayout` contains 5 different elements:



- Each of which contains another layout (`QHBoxLayout, QGridLayout, QFormLayout etc...`):



`QVBoxLayout` example

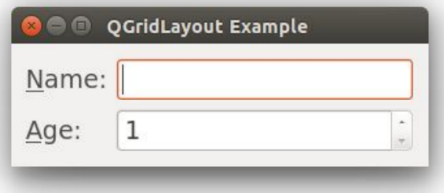
- Here is the example of `QVBoxLayout` from the lecture

```
QRadioButton* apple = new QRadioButton("Apple");
QRadioButton* banana = new QRadioButton("Banana");
QRadioButton* kiwi = new QRadioButton("Kiwi")
...
QVBoxLayout* layout = new QVBoxLayout();
layout->addWidget(apple);
layout->addWidget(banana);
layout->addWidget(kiwi);
layout->addStretch(1);
...
window->setLayout(layout);
```

- The result looks like this on Ubuntu. Note that the `addStretch` command adds additional space below *kiwi.*

`QGridLayout` example

```
...
QGridLayout* layout = new QGridLayout();
layout->addWidget(nameLabel, 0, 0);
layout->addWidget(name, 0, 1);
layout->addWidget(ageLabel, 1, 0);
layout->addWidget(age, 1, 1);
...
window->setLayout(layout);
```

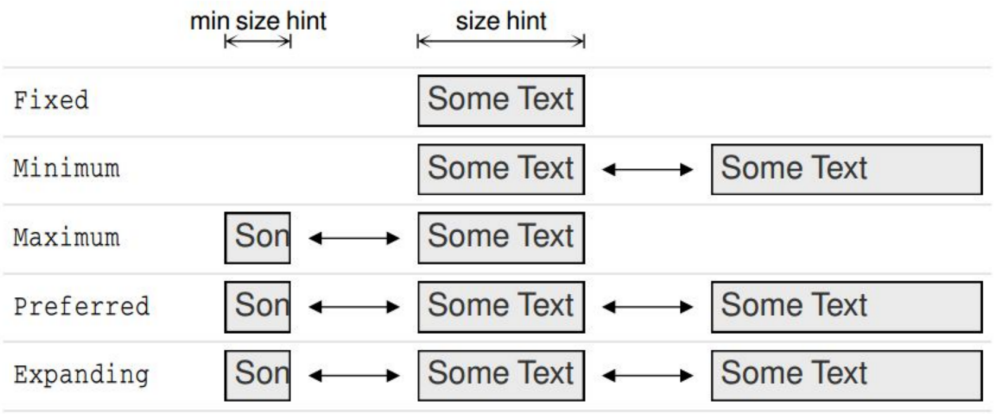


- Note that when you use addWidget with QGridLayout, you can add integer column and row arguments, for example: `(nameLabel, 0, 0)`

Constraining widgets in layouts

There are several techniques for widgets to define how much space they take up in a layout:

- Adding *stretchable space* to the layout, using `addStretch` (as we saw in the `QVBoxLayout` example)
- Setting an appropriate size policy:
  - The built-in widgets provide a `sizeHint()` function - this gives the "best" rectangular size for a widget so that it can show all its content without wasting space.
  - The built-in widgets also provide a `minimumSizeHint()`function - this gives the smallest rectangular size for a widget.
  - The size policy uses the `sizeHint` and `minimumSizeHint` to set the size of the widget in the layout (details here):
    - `Fixed` - the `sizeHint` is used
    - `Minimum` - the widget is sized greater than `sizeHint`
    - `Maximum` - the widget is sized smaller than `sizeHint`, but no smaller than `minimumSizeHint`
    - `Expanding` - the widget is sized greater than `minimumSizeHint`, with no maximum size
  - This is summarised in the following diagram:



- We can also set "hard" maximum and minimum sizes. These functions guarantee that the size of a widget in a layout will be above a minimum or maximum:
  - `setMinimumWidth()`

- `setMinimumHeight()`
- `setMaximumWidth()`
- `setMaximumHeight ()`

Qt Layout Workflow

There are many ways to approach creating layouts. However, a typical approach is to:

- Figure out the nesting for layout objects.
- Create the innermost layouts first. *Test each one as you create it!*
- Call `addWidget` to add a widget to these layouts, or `addStretch` to add stretchable space
- Move outwards, creating new layouts and calling addLayout to add an inner layout to an outer layout
- Call setLayout on the 'root widget' (`window, dialog or QWidget`), passing it the outermost layout object

---

Exercise 4. Follow the following instructions to experiment with `QHBoxLayout`

1. Create a new file this exercise called *hbox.cpp* containing the following `main` function (remember to remove or rename old `main` functions):

```
#include <QtWidgets>

int main(int argc, char* argv[])
{
 QApplication app(argc, argv);

 QWidget* window = new QWidget();
 window->setWindowTitle("QHBoxLayout Test");

 window->show();

 return app.exec();
}
```

   a. Notice the use of `QWidget` to represent the application's window. Qt does have a 'proper' window class called `QMainWindow` that supports the use of menu bars, etc, but we aren't using any of those features here, so `QWidget` will suffice.

   b. Run the program. An empty window should be displayed.

   c. Add code to *hbox.cpp*, to create three `QPushButton` objects. Remember to define these using pointers. Use `button1, button2, button3` as the names of these pointer variables.

   d. Now create a `QHBoxLayout` object and add the three buttons to it. The following code, put *before* the call to the `show` function, should do the trick:

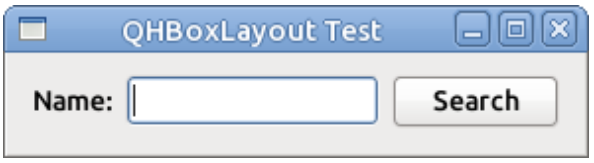
```
QHBoxLayout* layout = new QHBoxLayout();
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
```

   e. Add code to make this the layout used by the `QWidget` object representing the application window:

```
window->setLayout(layout);
```

f. Compile and run the program again. Try resizing the window both horizontally and vertically in various ways and observe how it behaves.

g. Change `hbox.cpp` so that it creates a label, a text field and a button, instead of three buttons. Use `QLabel` for the label and `QLineEdit` for the text field. Don't forget to add the `#include` directives for these components as required. The final result should look a little like this:



h. Recompile and rerun your program. Try resizing the window horizontally and vertically, as you did before. How has the resizing behaviour changed?

i. Add the following between the `addWidget` function calls for the text field and button:

```
layout->addStretch(1);
```

j. Recompile and rerun your program.
How has its resizing behaviour changed?

k. Move the call to `addStretch` to that it is before the line that adds the `QLabel` widget to the layout. Add a second call to `addStretch`, after the line that adds the button to the layout. Recompile and rerun your program. How has its resizing behaviour changed?

l. Change the integer passed to one of the `addStretch` function calls. How has the layout behaviour changed?

2. **Now**, start a new project or file to experiment with the `QLineEdit` widget

a. Copy the file *nested.cpp* from the resource zip into your project. Remove any other main functions you have in your project.

b. When you run the project, a blank window should appear.

c. Edit the file and, underneath the `// Create widgets` comment, add code to create a label and a text field:

```
QLabel* label = new QLabel("&Name:");
QLineEdit* nameField = new QLineEdit();
label->setBuddy(nameField);
```

d. Notice the `&N` in the label text and how the label's `setBuddy` function is called to associate the label with the text field. These two steps will allow users to shift focus to the text field using a keyboard shortcut (Alt+N).

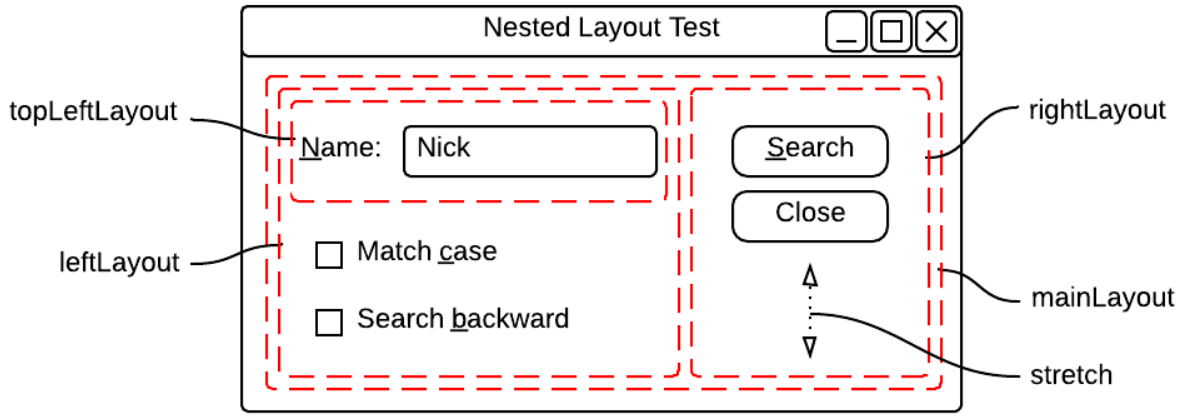
e. Run `make` to check whether the program still compiles. If you get any errors, double-check that you've entered the code correctly.

f. Now add code to create two buttons, labelled `Search` & `Close`, and two checkboxes, labelled "Match case" & "Search backward":

```
QPushButton* searchButton = new QPushButton("&Search");
QPushButton* closeButton = new QPushButton("Close");
QCheckBox* caseCheck = new QCheckBox("Match &case");
QCheckBox* backwardCheck = new QCheckBox("Search
```

```
&backward");
```

g. The next task is to arrange these components in a set of nested layouts, using the `QHBoxLayout` and `QVBoxLayout` objects shown below:



h. Start by adding the layout code for the left of the window, below the `// Arrange widgets` comment:

```
QHBoxLayout* topLeftLayout = new QHBoxLayout();
topLeftLayout->addWidget(label);
topLeftLayout->addWidget(nameField);

QVBoxLayout* leftLayout = new QVBoxLayout();
leftLayout->addLayout(topLeftLayout);
leftLayout->addWidget(caseCheck);
leftLayout->addWidget(backwardCheck);
```

i. Check the program still compiles and runs.

j. When you are placing a layout inside another layout, you must use the `addLayout` function rather than the `addWidget` function.

k. Next, add the layout code for the right-hand side of the window. See if you can figure this out for yourself! Use `rightLayout` as the name for the required layout object. Note that you should add a call to `addStretch` on this object to add a stretch element to the layout.

l. Finally, add the code that places `leftLayout` and `rightLayoutinto mainLayout` and set this as the layout of the window:

```
QHBoxLayout* mainLayout = new QHBoxLayout();
mainLayout->addLayout(leftLayout);
mainLayout->addLayout(rightLayout);
window->setLayout(mainLayout);
```

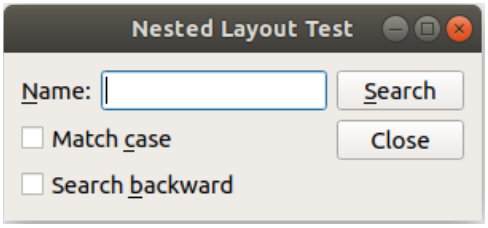
m. Run the program. Try resizing the window horizontally, then vertically, to see the effect this has on layout.

n. To stop vertical resizing from spoiling the layout, add this line before the line that shows `window`:

```
window->setFixedHeight(window->sizeHint().height());
```

o. Here, the chained function call `window->sizeHint().height()` returns the optimum height for the window's layout.

p. Let us pause and admire our glorious finished layout:



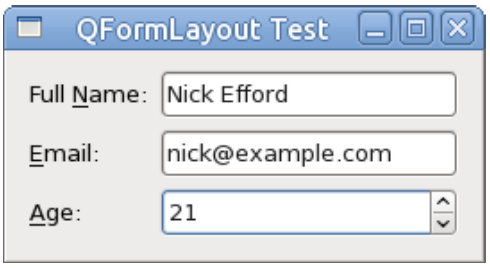
Let's try the `QFormLayout` next.

You've seen some examples of putting labels alongside corresponding input fields using `QHBoxLayout`. This is such a common requirement that Qt provides a specific layout class called `QFormLayout` to support it. `QFormLayout` can create the labels and buddy them with the associated input fields for you:

```
QFormLayout* form = new QFormLayout();
form->addRow("&Name", nameField);
```

`QFormLayout` will also align labels and input fields if there are multiple rows in the form, creating a neat two-column layout. Rows of the layout can be wrapped so that the labels appear above the corresponding input fields when horizontal space is limited.

3. Copy *form.cpp* from the resources zip and study it, set it up, and run the program. You should see something like this:



   a. Try using the Tab key to move between input fields. Check that the keyboard shortcuts work (`Alt+N`, `Alt+E`, `Alt+A` on Linux machines).
   b. Try changing the label alignment by adding the following line to the program:

```
form->setLabelAlignment(Qt::AlignRight);
```

   c. (If labels were already right-aligned on the platform on which you are running the program, use `Qt::AlignLeft` instead.)
   d. Remove the line that modified label alignment, then try changing the wrapping behaviour by adding the following line:

```
form->setRowWrapPolicy(QFormLayout::WrapAllRows);
```

   e. When you rerun the program, you should see that the labels are now above their corresponding input fields.

Exercise 5. This exercise introduces a cleaner and more scalable way of structuring your Qt application, as a *class* rather than dumping everything in the `main` function.

We focus here on creating subclasses of `QWidget`, which is a suitable choice for simple UIs that don't need menus, toolbars, etc. A better approach for more complex UIs is to subclass `QMainWindow` - which we will look at later.

1. Let's try a new way of structuring your UI using three files: *window.h*, which defines the class; *window.cpp*, which implements the functions of the class; and *main.cpp*, which provides the main program. *window.h* can have this content:

```
#ifndef LAB3_WINDOW_H
#define LAB3_WINDOW_H
#include <QWidget> // just the supertype of all widgets
class QPushButton; // forward-declare QPushButton so we
                   // don't need to import

class MyWindow: public QWidget // extending QWidget!
{
 public:
   MyWindow();

 private:
   void createWidgets();
   void arrangeWidgets();

   // Pointers to widgets are created here
   QPushButton* aButton;
};

#endif //LAB3_WINDOW_H
```

The `private` functions specified here should be called from within the constructor. This isn't a requirement; it is done purely to make the constructor smaller and improve the structure and readability of the code by keeping widget configuration and layout code cleanly separated.

It is polite to keep header files lightweight and concise. To avoid `include`ing the whole Qt widget definitions in header file we can use a forward-declaration:

```
class QPushButton;
```

You would then specify the widget instance variable in the `private` section of the class definition like so:

```
QLineEdit* nameField;
```

2. Create *window.cpp* with the above instructions. It should look something like this:

```
#include <QtWidgets> // include all widgets, including
push-buttons
#include "window.h"

MyWindow::MyWindow()
{
 createWidgets();
 arrangeWidgets();
 setWindowTitle("My Window");
 setMinimumSize(320, 200);
}

void MyWindow::createWidgets()
{
 // Create widgets using new
```

```
    aButton = new QPushButton("push me, i dare you");
}

void MyWindow::arrangeWidgets()
{
   // Create layout for widgets here
   QHBoxLayout* layout = new QHBoxLayout();
    layout->addWidget(aButton);
    setLayout(layout);
}
```

Notice how the constructor calls the functions to create widgets and layout. This is also a logical place to set the title for the window and impose any constraints on window size.

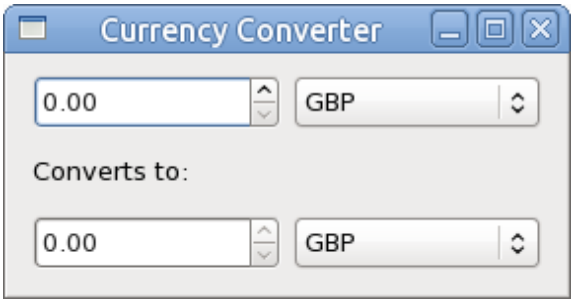
3. Finally your main function can take this simple form. Add it to the bottom of *window.cpp*:

```
int main(int argc, char* argv[])
{
 QApplication app(argc, argv);

 MyWindow window;
 window.show();

 return app.exec();
}
```

4. In this case, the top-level UI object is created and manipulated with automatic allocation, but you could create it with dynamic allocation instead if you wanted to:

```
QWidget* window = new MyWindow();
window->show();
```

5. Now follow these instructions:
    a. Modify the template described above to create a simple currency converter layout in a class, using the two files *window.h, window.cpp.*
    b. The application should look something like this:



    c. Use `QDoubleSpinBox` widgets for the amounts and `QComboBoxwidgets` for the corresponding currencies. Use a `QHBoxLayout` to pair up each spin box with the corresponding combo box. Use a `QVBoxLayout` for the overall layout of the window.

Further Reading / Useful links

- Qt API

Acknowledgements:

- sizeHint/ minimumSizeHint diagram: from Blanchette & Summerfield, *C++ GUI Programming With Qt 4*
- Layout examples: https://doc.qt.io/qt-5/examples-layouts.html
- Nick Efford
- Tom Kelly
- Rafael de Kuffner Anjos