Buổi 1

QT là gì?

Là khung phát triển ứng dụng có thể sử dụng trên các nền tảng khác nhau (window, linux…).

Ta có thể sử dụng tất cả các thư viện chuẩn của c++ cho QT,

Một bộ tiền xử lý, MOC (Meta-Object Compiler), được sử dụng để mở rộng ngôn ngữ C ++ với các tính năng như signal and slots. Trước khi biên dịch các file cpp -> file .o thì MOC sẽ biên dịch ra các file .moc trước.

Kiến trúc QT

* GUI Design
* Signal and slot: giao tiếp, tương tác giữa các object
* Meta Object System

QT module: QT core, QT GUI, QT multimedia, QT qml, QT quick…

QML là gì?

* QML là một ngôn ngữ khai báo được thiết kế để mô tả giao diện của một chương trình: cả hiển thị và hành vi.
* Javascript là ngôn ngữ kịch bản (scripting language) trong QML.
* Ngôn ngữ kịch bản không đòi hỏi phải biên dịch mà cần thông dịch. Viết code tới đâu nó chạy tới đó, gặp những lỗi nhỏ thì nó bỏ qua lỗi và chạy tiếp, nếu gặp lỗi nghiêm trọng thì nó đứng lại tại đó.

QT Quick là gi?

Bao gồm:

* Ngôn ngữ khai báo: QML
* Ngôn ngữ mệnh lệnh: JavaScript

QML component:

Là các QML được đóng gói, có thể tái sử dụng với các giao diện được xác định rõ.

* Khai báo import thư viện
* Root element: {property, sub qml component, scripts (java script function)

Các phần tử (element) được khai báo với nội dung chi tiết bên trong {}

Các phần tử có các thuộc tính (properties) : name : value, mỗi thuộc tính ta có thể set các value cho nó, nếu k set value thì nó sẽ lấy giá trị mặc định của các thuộc tính đó

Ta có thể thêm các thuộc tính tự định nghĩa vào: format: **Property** <type> <name> : <value>

Với mỗi thuộc tính ta có thể thêm các signal vào để xử lý chúng, những tín hiệu này sử được gọi mỗi khi value của thuộc tính thay đổi ví dụ ta muốn gọi hàm console.log(“height: “ + height) mỗi khi height thay đổi thì ta xử lý như sau:

**onHeightChanged**: console.log(“height: “ + height)

2 cách import file qml vào project:

* add vào qml.src
* để riêng file .qml và truyền đường dẫn tuyệt đối vào khi sử dụng

Cơ chế binding trong qml:

* biểu thức binding (qml): có dấu ':' thì là biểu thức binding. Mỗi property sẽ tự động thay đổi theo khi vế phải thay đổi
* gán giá trị (javascript) có dấu '=' , khi gán property bằng cách này thì nó sẽ remove tính binding của property (nghĩa là giá trị của nó sẽ được gián tĩnh và không thay đổi khi 1 biểu thức binding gán bằng dấu : . Nhưng vẫn có cách giải quyết vấn đề remove binding này vd:
* height = Qt.binding(function() { return width \* 3 })
* remove binding (biểu thức binding sẽ bị loss khi phép gán giá trị được thực hiện)

Signal handler

// SquareButton.qml

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: root

signal btnPressed()

signal btnRelease()

property [int](https://doc.qt.io/qt-5/qml-int.html) side: 100

width: side; height: side

[MouseArea](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

anchors.fill: parent

onPressed: {

btnPressed ()

}

onReleased: : {

btnRelease ()

}

}

}

Các thành phần cơ bản

Item:

* là thành phần cơ bản mà hầu hết các loại phần tử khác đều kế thừa hết các thuộc tính của item.
* Item bao gồm các nhóm thuộc tính:
* Geometry: x,y là top – left position; width, height là xác định kích thước
* Layout handing: anchors (left, right, top, bottom, vertical and horizontal center) kết nối tương quan giữa các object.
* Key handling: control key pressed (focus property…)
* Transformation: scale and rotate object
* Visual: opacity: để kiểm soát sự ẩn hiện của object.

Rectangle

Mở rộng từ item, kế thừa hết các thuộc tính của item, nó hỗ trợ thêm các thuộc tính khác như : border.color, border.width, radius (vát các góc), gradient (rectangle nếu k set width và height thì k thể sử dụng gradient) ….

Text

* Được dùng để hiện các đoạn text được kế thừa từ Item, có 1 vài thuộc tính hay sử dụng như:
* Các thuộc tính lưu trữ vị trí kích thước của text được kế thừa từ item: x,y, width, height, anchors…
* **Text**: nội dung text hiển thị
* **Color**: tùy chọn màu chữ
* **contentWidth**: chiều dài của nội dung text
* **clip**: khi set là true thì nó sẽ cắt nội dung text dài hơn width được set
* **elide**: Khi text dài hơn width được set thì phần text dài hơn sẽ co lại bởi ba dấu chấm. (chỉ enable khi set width cho text)
* **truncated**: (chỉ đọc) khi sử dụng elide nếu text dài quá phạm vi hiển thị thì truncated có giá trị true.
* **Wrapmode**: Tự động xuống hàng khi text dài hơn phạm vi hiển thị
* **HorizontalAlignment**: Text.AlignRight, Text.AlignHCenter (phải set width cho text)
* **VerticalAlignment**: Text.AlignTop, Text.AlignBottom, Text.AlignVCenter. (phải set height cho text)
* Vd:

Text {

id: *name*

color: "black"

font.pointSize: 24

clip: true

wrapMode: Text.Wrap

anchors.fill: *parent*

horizontalAlignment: Text.AlignHCenter

}

Image

* Cho phép load và hiển thị 1 image với các định dạng khác nhau như PNG, JPEG, SVG
* Các các thuộc tính quan trọng như:
* Các thuộc tính lưu trữ vị trí kích thước của text được kế thừa từ item: x,y, width, height, anchors…
* **source**: “url” ta add ảnh vào trong qml.prc như 1 file .qml khác. Khi đó ta chỉ cần truyền tên ảnh vào path. (note: khi dùng đường dẫn tuyệt đối thì ta phải thêm file:/// vào trước path)
* **fillMode**: Cho phép căn chỉnh ảnh phù hợp với kích thước đã set cho Image.

MouseArea

Sử dụng để tương tác khi click chuột

Vd:

signal btnClicked()

signal btnRelease()

MouseArea {

id: *mouseArea*

anchors.fill: *parent*

onClicked: {

*btnClicked*()

}

onReleased: {

*btnRelease*()

}

}

**Textinput và textEdit**: **text input** chỉ hiển thị 1 dòng văn bản thuần nhập từ bàn phím, còn **text edit** ta có thể hiển thị 1 khối văn bản có thể chỉnh sửa và **định dạng.**

Textdit thì ta cần set textFormat là TextEdit.AutoText thì sẽ sử dụng được các định dạng cho rich text.

Thuộc tính hay cho cả 2 loại nè: selectByMouse: true (bôi đen copy)

Buổi 2 JavaScripts

javaCripts là ngôn ngữ kịch bản động.

* không đòi hỏi biên dịch nhưng nó đòi hỏi thông dịch (thông dịch thực hiện trong môi trường ảo) mỗi lần chương trình chạy lại phải thông dịch.
* Ngôn ngữ kịch bản không đòi hỏi phải biên dịch mà cần thông dịch. Viết code tới đâu nó chạy tới đó, gặp những lỗi nhỏ thì nó bỏ qua lỗi và chạy tiếp, nếu gặp lỗi nghiêm trọng thì nó đứng lại tại đó.

Static là các lần chạy khác nhau đều như nhau.

**Creating objects dynamically**

Mỗi khi chạy runtime thì tùy vào điều kiện thiết lập (tùy theo logic) mà tạo ra các object khác nhau.

**Các kiểu dữ liệu cơ bản js:** number, string, boolean, null, **undefined, object**

Để sử dụng 1 file js trong file qml thì ta cần import nó vào, cú pháp import giống như trong python được học

Cách gọi hàm cũng tương tự trong python, ví dụ khi **Import “DemoJs.js” as Demo** Khi đó muốn gọi các hàm được định nghĩa trong file DemoJs.js thì ta gọi như sau : **demo.ten\_ham(ds thamso)**

Khi import giữa các file js khác nhau thì ta thêm dấu . vào trước từ khóa import

**Ta có thể tạo 1 object động** bằng cách gọi hàm Qt.createComponent() và Qt.createObject() hoặc bằng cách gọi duy nhất 1 hàm Qt.createQmlObject()

**Vd: Bổ sung đi bro (Xem ảnh nha)**

**Trong qml những phần sử dụng js:** Trong các slots xử lý các signal, khai báo define hàm js trong qml

**Code-Behind Implementation Resources** : khi ta tạo 1 object của file qml khi ta import 1 file js thì nó sẽ tạo 1 bản copy riêng object đó.

**Shared JavaScript Resources:** Khi từ khóa **.pragma library** được thêm vào file .js khi khi load các object file qml mà import file js kia, file js chỉ được load 1 lần.

Buổi 3 Animation

Animation không cho tạo đối tượng do nó là 1 abtract class

Các property:

* alwayRunToEnd: bool,
* loops: int
* paused: bool
* running: bool

signal :

* started(), : emit khi bắt đầu
* stopped(): emit khi kết thúc
* finished(): emit khi hoàn thành **1 cách tự nhiên**

Method

* complete(): hoàn thành animation với giá trị aniamtion được set là kết quả cuối cùng
* stop(): dừng
* pause(), resume()
* restart(): bắt đầu sau khi stopped
* start()

Các loại

* **Number animation:** dùng để animate các thuộc tính có giá trị là số: ví dụ: width, height, x,y
* **ColorAnimation**
* **rotationAnimation**
* **parentAnimation**
* **AnchorAnimation**
* **Vector3dAnimation**
* **SequentialAnimation**
* **ParallelAnimation**
* **PauseAniamtion**

**Transition ( k kế thừa animation)**

Khi nào 1 đối tượng có thể trigger để chạy animation:

* Gọi hàm start()
* Khi 1 đối tượng được tạo
* Khi 1 property change( behavior)
* Khi thay đổi state( transition)

State

là 1 tập các thay đổi khác so với cấu hình ban đầu (khi mới tạo đối tượng) **:Xem ảnh ví dụ đi**

**Transition**: xác định các animations sẽ được áp dụng khi xảy ra thay đổi **state**

Các property của transition:

* List các animation: vd: Numberanimation, ….
* Running:
* From (string): state1
* To(strin): state2
* Enabled: kích hoạt animation khi có from và to

Timer:

Trigger 1 hành động một trong 1 khoảng thời gian cho trước.

Property:

* Interval: set time cho sự kiện trigger
* Repeat: lặp lại không?
* Running:

Signal: triggered()

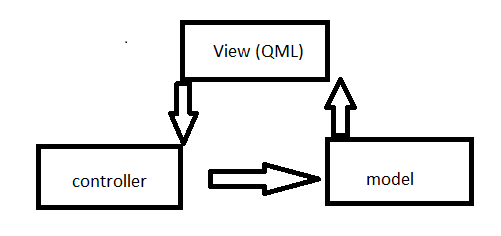
Ví dụ: number animation : ảnh code demo **RunningText.png**

Ví dụ: animation with transition and state: **Animation\_With\_Transition\_State.png**

Ví dụ minh họa về state: **State\_Qml\_demo.png**

Ví dụ minh họa về timer: **Timer\_Demo.png**

Buổi 4 Model view



**Model**: nơi lưu trữ data và các logic xử lý khai thác sử dụng data, cung cấp data cho view

**View** là hộp chứa để hiển thị model (dữ liệu): Repeater, ListView, GridView, PathView, TableView

**Controller**: phản hồi từ người dùng, điều hướng dữ liệu thông qua model

**Delegates**: cách để render cho 1 instance trong view ( cách hiển thị ra 1 item riêng biệt) , nó có thể là 1 component or 1 file qml

**Listview và repeater**

**Listview** không đồng thời vẽ hết tất cả các phần tử của list model, nó chỉ vẽ số lượng item hiển thị đủ trong kích thước list view và 1 số item buffer. (property cacheBuffer)

**Repeater**: vẽ tất cả item đồng thời: tất cả item giống nhau về kích thước, và số item ít

Note: Flickable + repeater tính năng tương tự như listview (có thể cuộn lên xuống được)

Mỗi 1 item sẽ có 1 index

Tạo model: Tạo model trong qml, Tạo model trong c++

*Tạo model trong qml:*

* **1 số integer: ModelViewDemo project file main.qml**
* Array: **ModelViewDemo project file main.qml**
* **ListModel**: item – simple hierarchi of QML type **ModelViewDemo project file main.qml**
* **XmlListModel** – XML data source
* **ObjectModel** – Visula Items, no delegate needed

Buổi 5. Layout, Container

Layout:

Nơi đặt ra các bố cục, quy tắc để sắp xếp các đối tượng của của nó trong layout (Các quy tắc có thể như các ràng buộc về kích thước tối đa và kích thước tối thiểu, các tín hiệu (fillWidth, fillHeight…) cho phép thay đổi kích thước object bên trong layout khi layout resize)

* **prefereWidth**: kích thước ưa thích của object, default -1, khi đó nó sẽ sử dụng implicitWidth thay thì.
* **fillWidth**: default(false): fix chiều rộng component ở giá trị prefered, nếu true thì cho phép co giản nếu layout bị resize.

Nó k thể tạo được object riêng để sử dụng mà nó chỉ được sử dụng khi đặt trong 4 loại layout sau

* Rowlayout
* ColumnLayout
* Gridlayout: lưu ý: khi set rows thì phải set rows thì phải set flow: topToBottom, khi set colume thì mặc định flow se là leftToRight
* Rows:int
* Columns: int
* Flow: enum: leftToRight (khi set Columns), topTobottom (khi set Rows)…
* LayoutDirection: hướng xếp các item vào lưới: leftToRight, rightToLeft
* Stack layout

Container

Vùng chứa cho phép người dùng có thể chèn thêm, hoặc loại bỏ bớt đi các mục 1 cách dynamic.

Thêm 1 item: tabBar(id).addItem(object) vd: tabBar.addItem(tabButton*(id\_component*).createObject(tabBar(id\_parent))

Xóa 1 item: *tabBar*.removeItem(*tabBar*.itemAt(1)) (tabBar là id của component TabBar{}) itemAt(index).

Di 1 item: tabBar.moveItem(from, to), from, to là int(index)

Chèn thêm 1 item tại ví trí bất kỳ: *tabBar*.insertItem(3,*tabButton*.createObject(*tabBar*)) // 3 là index cần chèn, cái kia là item cần chèn

Lấy ra thông tin 1 item: ta dùng tabBar.itemAt(index)

Mỗi container lại có 1 mục đích khác nhau

Các loại phổ biến: (ít dùng, vì sao ít dùng?)

* dialogButtonBox
* MenuBar
* SwipeView
* TabBar

Loader (quan trọng)

* Tạo object qml động. Nó có thể load 1 file qml, hoặc 1 componet object. Nó hữu ích khi chỉ muốn tạo đối tượng khi được yêu cầu.
* Nếu size loader mà không được set thì nó sẽ lấy kích thước của object được load lên
* Nếu size loader được chủ ý set thì kích thước object được load sẽ phải lấy kích thước được set từ loader, áp dụng cho cả object được load., nếu không thì nó sẽ lấy size được định nghĩa trong đối tượng được load lên
* Có 2 cách load:
* C1: idloader.source: “tên\_file\_qml”

// file: LoaderSample.qml

import QtQuick 2.0

Rectangle {

color: "red"

width: 200

height: 200

}

// file main.qml

Loader {

id: *idLoader*

anchors.centerIn: *parent*

}

MouseArea {

anchors.fill: *parent*

onClicked: {

*idLoader*.source = "LoaderSample.qml"

}

}

* C2: tạo 1 component và loader trong cùng 1 file qml.

Loader {

id: *idLoader*

anchors.centerIn: *parent*

//Ta cũng có thể load source trực tiếp ở đây, khi đó object sẽ được load mà k cần tín hiệu kích hoạt load

//sourceComponent: comp

}

Component {

id: *comp*

//Chỉ có thể có 1 object trực tiếp là con của Component

Rectangle {

color: "red"

width: 200

height: 200

}

}

MouseArea {

anchors.fill: *parent*

onClicked: {

*idLoader*.sourceComponent = *comp*

}

}

**Để lấy nôi dung của 1 item được loaded ta dùng:**

myLoader.item //myLoader là id của component Loader{}

Khi đó **mọi signal** được emit từ object được load thì đều có thể được nhận bởi Connections vd:

**Connections** { target: myLoader.item…. tên\_slot\_ứng\_với signal từ item được load ….}

**Connections** : dùng để tương tác với các signal từ đối tượng được load

Buổi 6 Meta object Signal slot (Quan trọng pro ơi)

* The meta-object system
* Qobject: class core, các class khác đều phải kế thừa từ Qobject để sử dụng signal slot
* Meta object compiler
* Signal and slot

…

Meta object: cung cấp cơ chế signal và slot cho phép giao tiếp giữa các đối tượng, nó dựa trên 3 thứ:

* Qobject class base của các class khác muốn sử dụng cơ chế signal slot phải kế thừa.
* Q\_OBJECT: bắt buộc nằm trong mục privated của class (kế thừa class Qobject)
* Quá trình compiler: file.cpp sẽ compiler ra file .o tương ứng, file.h sẽ tạo ra file moc.cpp -> sau đó lại compiler file moc.cpp tạo ra file .o tương ứng tiếp

3 điều kiện để tạo 1 class có thể sử dụng signal slot

* Kế thừa **Qobject**
* Có **Q\_OBJECT** marco
* Built with **moc** tool

Những feature được cung cấp bởi meta object system:

* Signal slot,
* Q\_property: get set property object bằng property name.
* Tr(): translate strings

Bản chất của slot là các function.

Khi 1 signal được phát ra thì các slot kết nối với nó sẽ được thực thi ngay lập tức, giống như 1 lệnh gọi hàm thông thường.

Slot sẽ được gọi khi một signal kết nối với nó được phát ra.

1 signal có thể kết nối nhiều slot

Nhiều signal có thể kết nối 1 slot

1 signal có thể kết nối 1 signal khác

Khi muốn ngắt kết nối signal slot thì ta sử dụng Qobject::disconnect(… khác tương tự như hàm connect())

Chú ý: Thứ tự kết nối luôn là signal -> slot

2 Cách kết nối: cũ và mới.

Cách cũ: k được kiểm tra (tên signal slot, matching parament) khi compiler mà chỉ check khi runtime

Cách mới

Kết nối giữa signal và slot trong C++: Xem chi tiết code demo trong project **SignalSlot**

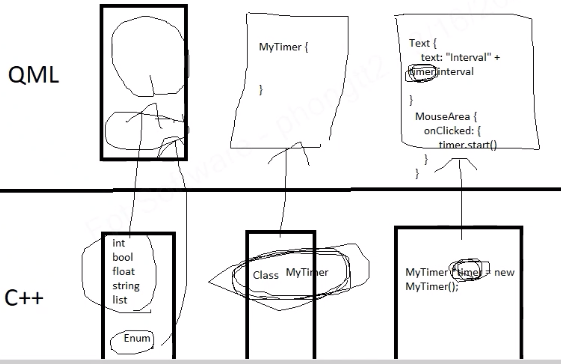
Kết nối giữa qml và C++: Xem chi tiết code demo trong project **SignalSlot2**

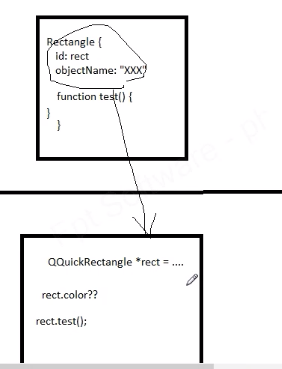
**Téo sẽ xem lại**

Buổi 6 Tương tác giữa C++ và Qml

Có cái gì đó ở C++ ta muốn sử dụng ở qml:

3 trường hợp:

Có cái gì đó ở qml ta lại muốn sử dụng ở C++



Q\_ENUM(ten\_enum).

**qmlRegisterType**<tenclass>(“ten\_de\_import”, 1,0, “ten\_sử\_dung\_tạo\_object tren qml)

**qmlRegisterUncreatableType**<tenclass>(>(“ten\_de\_import”, 1,0, “ten\_sử\_dung\_tạo\_object tren qml”, “tin\_nhin\_bao\_loi”)

**setContextProperty**(…)

Truy nhập object trên qml từ c++: ảnh demo ok.

Code demo: Xem project **InteractCPP\_Qml**

Day 9: List model C++

**Model** nơi lưu trữ data và các logic xử lý khai thác sử dụng data.

**Controler** chỉ yêu cầu điều hướng sử dụng dữ liệu, xử lý dữ liệu thông qua model

**View**: bố cục, cách mà dữ liệu hiển thị.

**Qvariant**: là 1 kiểu biến trung gian, có thể cast qua lại giữa các kiểu dữ liệu khác nhau. Khi viết hàm ta sử dụng parameter kiểu Qvariant thì không cần phải overload cho các kiểu dữ liệu khác nhau.

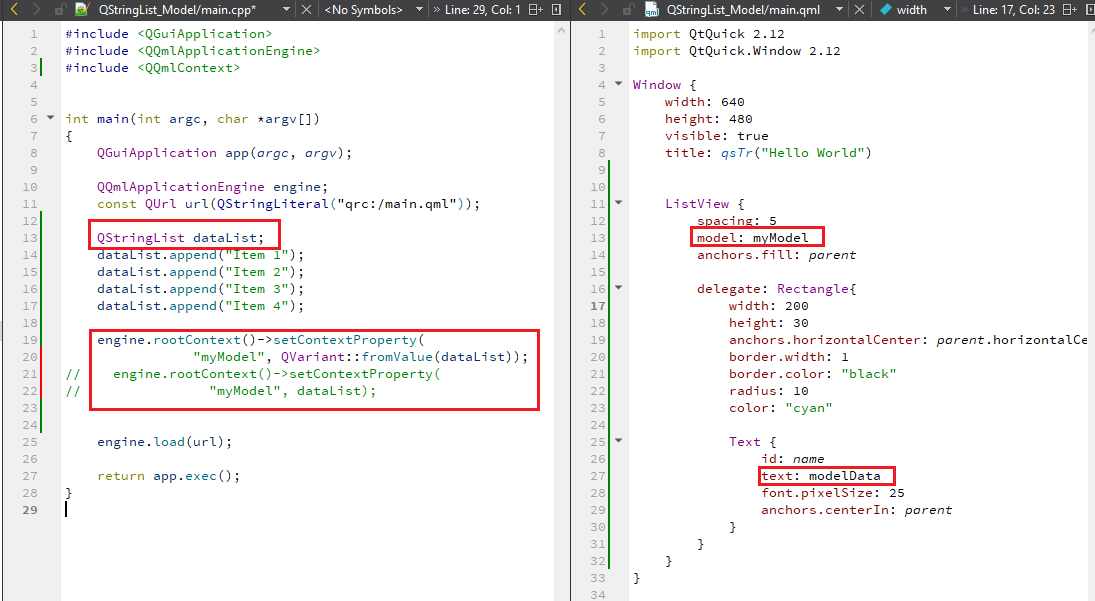
QstringList model

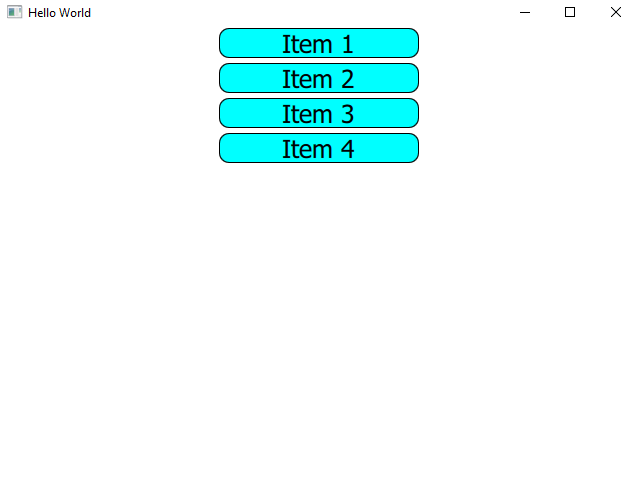
QStringList và 1 class được kế thừa từ QList với object là QString (QList<QString>)

QStringList có thể được sử dụng để cung cấp data cho model bên file .qml

Để sử dụng 1 object của QStringList được tạo từ bên C++ sang file .qml thì bước đầu ta cần setContextProperty vd:

* QStringList myModel
* // hàm setContextProperty cần 2 tham số là "ten su du ben qml" và địa chỉ object
* engine.rootContext()->setContextProperty("SongModel", &myModel);
* chý ý: ta cần setContextProperty trước khi engine load.





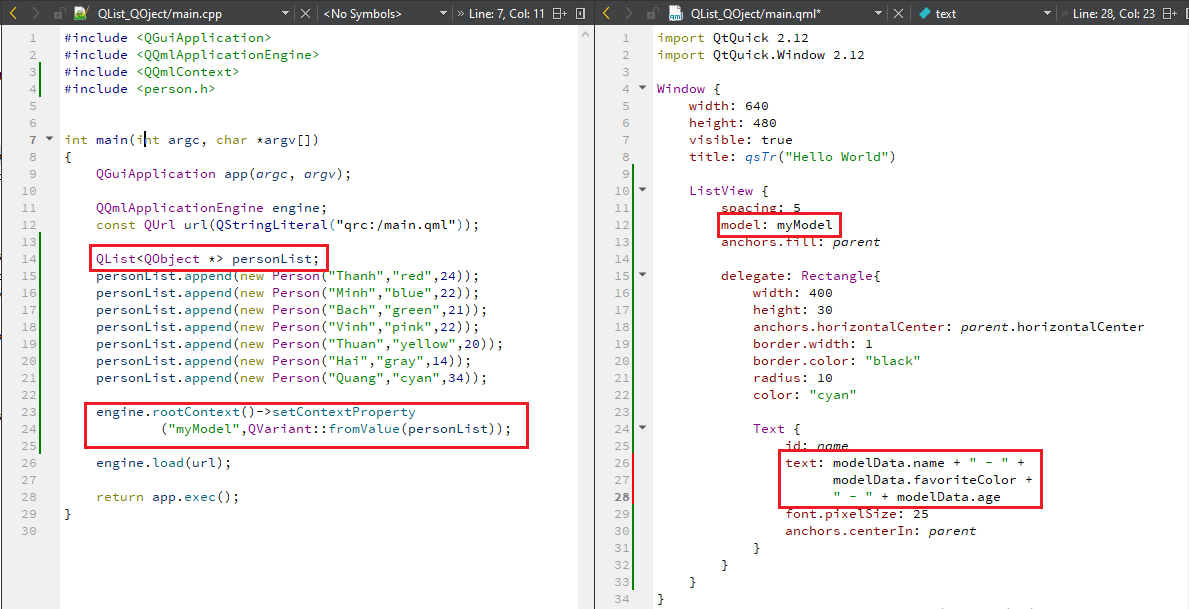
Qobject List

QList <Object\*>

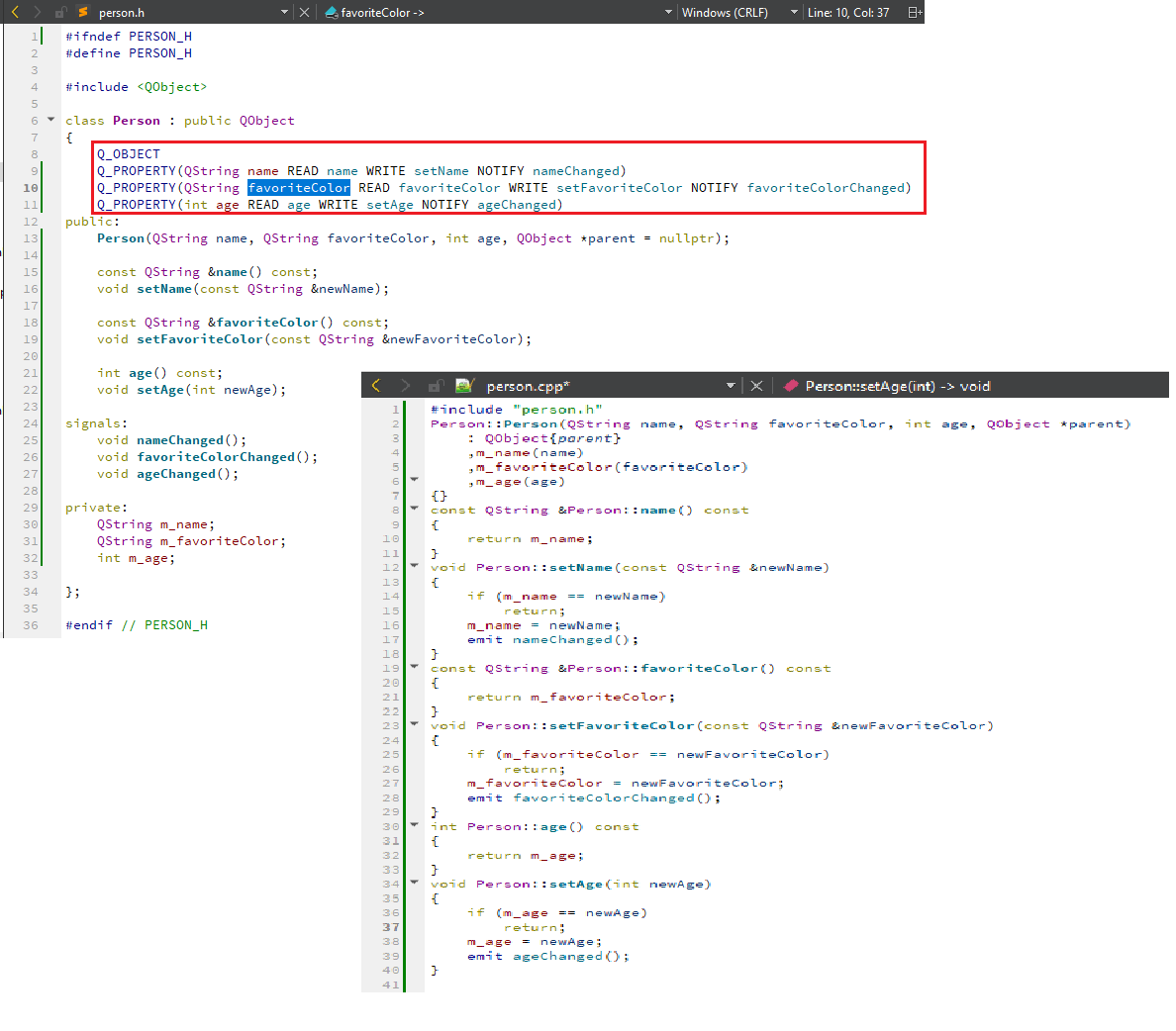
* Tổng quát hơn từ QString list chỉ lưu trữ danh sách các object là kiểu QString
* QList <object \*> lưu trữ danh sách các object có kiểu tùy chọn.
* Cách sử dụng tương tự như QStringList nhưng phần setContextProperty thì tham số thứ 2 cần cast sang Qvariant

vd:

* QList<Object\*> myModel
* myModel.append(new myClass("abc", "xyz,...))
* engine.rootContext()->setContextProperty("SongModel", QVariant::fromValue(myModel));



Class person:



Editable List Object \*

Abstract List Model

Class phải kế thừa QabstractListModel và phải định nghĩa lại các hàm rowCount, data, và Qhask roleName;

Ảnh demo ok.

Load tên các file với một đuôi xác định vào 1 list ta sử dụng Hàm entryList

Để gọi và sử dụng hàm entryList thì ta cần tạo 1 object thuộc class QDir (do vậy cần include <QDir>.

vd:

QDir myDirect("folderPath...");

//Load tất cả file có đuối là png và jpg

QStringList myList = myDirect.entryList(QStringList() << "\*png" << "\*jpg");

Buổi 10 Chuyển đổi các kiểu giữa Qml và C++

Data ownership

Khi 1 object được tạo từ c++ sau đó truyền lên qml để sử dụng thì ta cần phải setObjectOwnership để nó xác định rõ đối tượng đó là cppOwnership hay để khi sử dụng trên qml nếu đối tượng bị hủy thì nó sẽ k hủy ở c++.

ĐỂ set Data Ownership cho 1 đối tượng thì ta sử dụng hàm

QqmlEngine::setObjectOwnership(value, QqmlEngine::CPPOwnershipe);

QqmlEngine::CppOwnerShip and QQmlEngine::JavaScriptOwnership

Ôn tập lại phần tương tác giữa C++ và Qml phần **qmlregistertypte** để tạo object của class c++ trên file qml và phần **qmlregisteruncreatobject** để sử dụng enum tử class c++ trong file qml

Buổi 11. Multi thread graphic and mô hình MVC

Multi thread grapthic

Nắm được có những cách nào để tạo 1 custom GUI element?

* tạo item và vẽ lên canvas sử dụng Qpainter
* tạo item và vẽ theo cơ chế scene graph
* tạo item và vẽ theo code openGL

Nắm được thế nào là main thread (UI thread) và render thread?

* Main Thread là thread main của chương trình. Mỗi chương trình có 1 thread chính gọi là main thread. Khi một ứng dụng có UI thì main thread thường được gọi là UI thread. Nếu main thread bị hang tại một thời điểm nào đó thì ứng dụng lúc đó gọi là bị hang và người dùng sẽ không thể tương tác trên giao diện được.
* Render thread là 1 sub thread để support render các UI item.

Trong qt làm thế nào để enable render thread loop?

* " set biến môi trường: QSG\_RENDER\_LOOP với value = "threaded"

Sự khác nhau animation và animator?

* Animations chỉ hỗ trợ chạy trên main thread vì vậy nếu main thread bị block thì animation sẽ bị dừng lại cho tới khi main thread free. Animator support chạy trên render thread loop vì vậy nó sẽ không block UI.
* Animations có thể chạy trên 1 hoặc nhiều property. Animator chí có danh sách cố định các property.

MVC là gì?

* MVC là một design parten cho một ứng dụng UI, nó chia app thành 3 khối chính : M-model (các class dùng để lưu data), V- view (các class hay Item define cách vẽ các thành phần giao diện), C- controller (các class control để handle xử lý bussiness logic).

Animation advance

Animaton: binding data mỗi khi giá trị animation thay đổi bước nhảy. thực hiện ở main thread

Animator: thực hiện trên render thread, k bị blog UI, chỉ update property khi hoàn thành animation

* OpacityAnimator
* ScaleAnimator
* RotationAnimator
* UniformAnimator
* XAnimator
* YAnimator

Nhược điểm animation:

* Khi animation chạy thì ngốn cpu do nó cần tính toán nhiều trong khoảng thời gian ngắn
* Chạy trên main thread: mỗi lần timer trigger thì nó cần set property (tính toán biding data) của các đối tượng qml, mà việc này cần thực hiện trên main thread

Animator: giải quyết các vấn đề sảy ra khi animation bị giật lag.

Nó thi hành ở Render thread

Nhược điểm của animator:

Do nó chỉ update 1 chiều từ qml 🡪 sence graph mà k có chiều ngược lại do đó, property của qml chỉ update khi animator chạy xong (property của qml không biding data khi chạy animator)

Vấn đề Khi ta muốn truy xuất 1 object thuộc class này trong 1 class khác, Nếu số lượng class lớn, sảy ra vấn đề khó quản lý, phức tạp do có thể sảy ra trường hợp 1 class này chứa object của class kia, và class kia lại chứa object của class này.

Do vậy để giải quyết vấn đề đó thì có 2 cách:

Cách 1: chia tầng thiết kế, class tầng cao hơn, bên trong nó có thể tạo được object của class tầng thấp hơn, các class cùng 1 tầng thì class này k thể trực tiếp truy xuất đến class kia, mà phải thông qua class ở tầng cao hơn.

Cách 2: Ví dụ 1 class thì ta sẽ tạo 1 object của chính class đó bên trong trong nó. Object này là duy nhất (instance, static) và 1 hàm để truy xuất đến object static đó; Trong hàm này sẽ kiểm tra nếu object này null thì cấp phát cho nó và return về object đó, nếu không thì sẽ chỉ return về object đó. Khi đó 1 class khác muốn tạo object của class kia, thì nó chỉ việc truy xuất tên\_class\_kia::getInstance().

Ưu điểm của sington là nó có thể gọi instance bất kì chỗ nào, nhưng nó cũng là nhược điểm của nó, do có thể gọi được bất kì chỗ nào, do đó nó phá vỡ thiết kế hệ thống ban đầu (hệ thống có phân cấp)

Để đảm bào đối tượng sington là duy nhất thì ta để hàm constructor của nó ở private, và hàm dựng copy là cũng ở private và delete nó đi.

Buổi 12 Đa ngôn ngữ

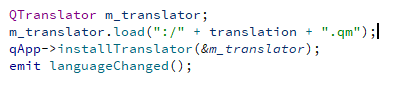
Qt Localization WorkFlow

* Cấu hình file .pro : TRANSLATIONS += lang\_en.ts \

lang\_vn.ts

* Khai báo tên các **file .ts** sẽ tạo ra khi chạy lupdate: nội dung các **file .ts** sẽ được **lupdate** sẽ tìm kiếm tất cả các string ở trong hàm **tr()** **ở file .cpp và qsTr() ở file .qml,** các string này như 1 key id: Các key này sẽ được translate ra nội dung tương ứng ở thẻ <**translation**> trong **file .ts.**
* Sau khi chạy **lupdate** và đã có file .ts với tên đã được khai báo thì ta chạy tiếp **lrelease** để tạo các **file .qm.** (mỗi lần cập nhật file .ts thì ta cần phải chạy lại lrelease)
* Khi đã có các **file .qm** (mỗi file tương ứng với 1 loại ngôn ngữ khác nhau cần translate của các key), Khi đó khi gọi hàm tr(“string\_id”) trong file .cpp hoặc qsTr(“string\_id”) trong file .qml, thì nó sẽ search các key này trong file .qm, **nếu có nội dung đã được translate thì nó hàm sẽ trả về nội dung này, nếu k có thì hàm sẽ trả về chính nội dung string\_id đã truyền vào.**

Các bước triển khai multi language

* Cấu hình file .pro
* Tạo 1 class để với các key\_string\_id được khai báo với **cùng 1 signal để modify**: do đó, khi thay đổi thiết đặt ngôn ngữ thì tất cả các nội dung của key\_string\_id sẽ đều được cập nhật lại với chỉ duy nhất 1 signal cần emit khi thiết đặt lại ngôn ngữ
* Hàm thiết đặt lại ngôn ngữ: 

*Tip:* Sau khi ta đã tạo được class quản lý việc thay đổi thiết đặt ngôn ngữ hiển thị cho file .qml thì trong class đó ta nên tạo thêm 1 string\_id có tên là **empty\_id với nội dung translate là rỗng,** Khi đó, ta muốn thêm 1 string\_id bất kì bên file .qml thì ta chỉ cần cộng thêm empty\_id này, khi đó việc thay đổi thiết đặt ngôn ngữ sẽ thay đổi cả những key\_id mới thêm vào này

Buổi 13 Dbus and Share memory

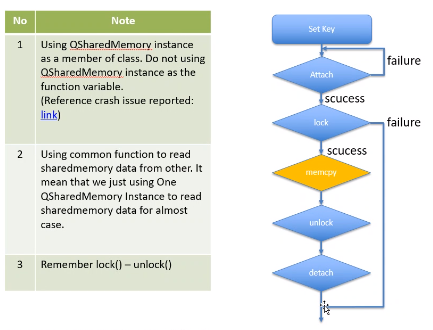
Share memory (ShareMemory.png)

Hạn chế:

Nằm trên RAM, nên chỉ tồn tại trong thời gian chạy, bị mất khi tắt thiết bị

Dung lượng vùng nhớ hạn chế

Cách sử dụng share memory



Ghi data vào share memory

* Set key : (key này như 1 pathFile (key này không được trùng nhau))
* Create : Lần đầu sử dụng key (app mà tạo ra phân dùng bộ nhớ chung ) thì ta phải create để tạo 1 phân vùng trên RAM với kích thước cho trước.
* Lock:
* Write: Sử dụng 1 con trỏ buffer để trỏ tới phân vùng nhớ đã cấp phát, Khi đó ta sẽ ghi vào con trỏ buffer này
* Unlock

Đọc data từ share memory

* Set key : (key này như 1 pathFile (key này không được trùng nhau)). Key này phải trùng với key mà app trên đã tạo ra phân dùng share memory.
* Lock:
* read: Sử dụng 1 con trỏ buffer để trỏ tới phân vùng nhớ đã cấp phát, Khi đó ta sẽ ghi vào con trỏ buffer này
* Unlock

Dbus

Có 2 dạng là system bus và session bus: System bus sẽ bắn ra các signal tới tất cả user. Còn session bus chỉ bắn các signal tới các app mà user hiện đang có quyền sử dụng

Có phần giống với connection giữa signal và slot

Cách hoạt động:

1 app sẽ gửi 1 signal xuống hệ điều hành, hệ điều hành sẽ broadcast cái signal đó lên, Khi đó, tất các các app khác đang connect tới signal này sẽ nhận được tín hiệu

Cần phải configuare ở file .pro bằng cách thêm : QT += dbus

Có 2 cách sử dụng dbus:

**broadcast signal:** Cách này 1 app sẽ bắn 1 signal xuống hệ điều hành để nó bắn signal đó tới các app khác, cách này xử lý rời rạc, nếu có nhiều signal thì khó quản lý và maintain. Cách này xử dụng với 1 signal thì đơn giản, không cần quan tâm tới file xml gì cả.

// 1. Create a dbus signal

QDBusMessage msg QdbusMessage::**createSignal**(

“/home/appInterface”, //object path

“home.appInterface”, // service interface

“requestChangeVolume” //signal name

);

// 2. Appendinput argument

Msg << m\_currentVolume;

//3.broad cast signal using session bus

**QDBusConecttion::sessionBus().send(msg);**

App khác muốn phải conect signal trước khi app1 broadcast signal

**QDBusConnection::sessionBuss().connect**(

“/home/appInterface”, //object path

“home.appInterface”, // service interface

“requestChangeVolume”, //signal name

This, SLOT(onReceiverResquestChangeVolume(int)

);

Gom các signal và method vào 1 file .xml để dễ quản lý

Với app mà có file xml để quản lý method và signal cần share thì trong file .pro cần cấu hình thêm:

appsxml.files += ./xml/appmanager.xml

DBUS\_ADAPTORS += appsxml

Sau khi đã có file xml và cấu hình file .pro ta cần buld app để tạo ra các file .h cần thiết để include vào để sử dụng

Buổi 14 Multi Thread

Vd 1: QtConcurrent.run(….), mỗi lần chạy sẽ tạo ra thread mới

QthreadPool …. (Giới hạn số thread tạo ra)

Vd2 : Tạo 1 class kế thừa lại Qthread và overload lại hàm run

Vd3: tương tự vd2 nhưng mà xử lý trường hợp khi có nhiều quá trình xử lý đang trong hàng chờ của thread.

Xây dựng màn hình

Các nguyên tắc xây dựng màn hình

* Tạo file qml riêng cho các đối tượng bị lặp lại
* Suy nghĩ hướng đối tượng
* Khả năng mở rộng
* Dễ đọc, kiến trúc rõ ràng
* Code qml nhẹ nhất có thể
* Kích thước các màn hình động

Đánh giá yêu cầu:

* Count số lượng màn hình theo tính năng (user view)
* Ước lượng số lượng màn hình theo source code (dev view)
* Liệt kê các componet qml cần sử dụng
* Điều hướng qua lại giữa các màn hình

Cách mà thêm 1 màn hình vào 1 dự án đã đang chạy.

* Cách mà 1 màn hình trước đó được hiển thị (load screen)
* Cách mà từ 1 màn hình chuyển qua 1 màn hình khác (chuyển screen)

Progress:

* Tạo màn hình chính trước, tạo màn hình phụ sau
* Dummy các data: ví dụ như dựa vào input từ người dùng tạo 5 rectangle. Nhưng chưa lấy được dữ liệu từ người dùng thì ta đưa data dummy là 5 vào để test tính năng tạo 1 số lượng rectangle có sẵn
* Dựng các component từ chính đến chi tiết

UI -> test on pc -> test on device -> commit review (UI: User Interface: Giao diện người dụng)

UX: User Experience: trải nghiệm người dùng

Tối ưu trong thiết kế màn hình: 9 key để tối ưu:

* Divide and Rule: Tối giản hóa các item lồng vào nhau
* Sử dụng C++ thay cho JavaScript: tránh sử dụng js
* Sử dụng multi thread trong việc load các model
* Sử dụng anchors, hạn chế sử dụng, x, y
* Sử dụng loader để tạo component động
* Tối ưu hóa lists
* Không nên sử dụng toán tử cộng khi nối chuỗi
* Sử dụng đúng loại tên property vd: string , url, phải rõ ràng