

SISTEMAS INFORMÁTICOS INDUSTRIALES

ACTIVIDAD 7 – Reforzando los conceptos de programación orientada a objetos.

Resumen

En esta actividad vamos a reforzar los conceptos de **encapsulamiento**, **herencia** y **polimorfismo**.

En la primera parte consiste en una guía paso a paso, para crear un proyecto en C++

En la segunda parte definiré un problema similar que tendréis que realizar por vuestra cuenta.

Primera parte: Problema a resolver

Un programa de ventanas en Qt que realice la gestión del personal de una institución docente.

Se desea implementar un programa de gestión de los miembros de una institución docente. Concretamente profesores y alumnos, de una forma simplificada enfocada al aprendizaje de los conceptos básicos de la programación orientada a objetos.

En el programa se podrá dar de alta profesores y alumnos, los cuales serán añadidos a una lista de personal (una lista común para todas las personas). Además, se podrá acceder a toda la información de una persona que se encuentre en dicha lista.

Definición de **Profesor**:

Un profesor estará definido por los siguientes datos:

- Nombre
- Apellidos
- Número de identificación (DNI/NIE)
- Departamento
- Ubicación del despacho
- Horario de tutorías

Definición de **Alumno**:

Un alumno estará definido por los siguientes datos:

- Nombre
- Apellidos
- Número de identificación (DNI/NIE)
- Curso
- Matrícula (lista de asignaturas matriculadas).

Tanto los alumnos como los profesores deben de poder generar una línea de texto que indique el número de identificación, los apellidos y el nombre y el rol (profesor o alumno)

Tanto los alumnos como los profesores deben de poder generar un texto multilínea con toda la información disponible.

Definiendo la jerarquía de clases para alumnos y profesores.

Podemos ver que las definiciones de alumno y profesor comparten una parte de su funcionamiento, por lo que es inteligente pensar que ambos pertenecerán al mismo árbol de clases

Debemos elegir si:

- Alumno hereda de Profesor
- Profesor hereda de Alumno
- Alumno y Profesor heredan de una clase base común.

Si analizamos los requerimientos de las definiciones de *profesor* y *alumno* podemos ver que ambos comparten tres propiedades *Nombre*, *Apellidos* y *Número de identificación*. Sin embargo, el resto de la información difiere bastante. De hecho, hay cosas de *profesor* que no aparecen en *alumno* y viceversa. Por lo que parece que no tiene sentido que uno herede del otro. Lo más idóneo en este caso es crear una clase base común, que podemos llamar **Persona** y dos clases **Profesor** y **Alumno** que deriven ambas de la clase común *Persona*.

La clase base **Persona**

La clase base *Persona* debe definir tanto aquellos datos que sean comunes para *profesor* y *alumno* como el comportamiento que sea común, aunque algunas de las funciones comunes sean diferentes para alumnos y profesores.

Debe definir las propiedades *Nombre*, *Apellidos* y *NúmeroDeIdentificación* y sus métodos de acceso. Pero también debe definir la función de *generar una línea de texto* y la de *generar un texto multilínea*. Estas dos funciones tendrán un comportamiento diferente dependiendo de si un objeto es de la clase *Persona*, *Alumno* o *Profesor*. Por lo que definiremos dichas funciones como virtuales en la clase base (clase *Persona*) con una implementación básica y redefiniremos su implementación en las clases derivadas (*Alumno* y *Profesor*).

Encapsulación de propiedades:

Vamos a resumir lo que implementaremos más adelante (en la sección *implementación paso a paso* veremos cómo llevamos esta implementación)

En la clase *Persona* vamos a definir tres propiedades de tipo *QString*: *nombre*, *apellidos* y *numID*; en la zona privada de la clase. Y en la zona pública añadiremos los métodos para acceder a dichas propiedades (los llamados *getters* y *setters*)

Además, vamos a definir de los métodos **virtuales**:

```
virtual QString Resumen();
```

```
virtual QString Descripción();
```

Recuerda que los métodos virtuales son métodos cuyo comportamiento puede modificarse en las clases derivadas.

La clase Profesor:

Profesor hereda de la clase *Persona* (esto significa que ya tendrá implementado el funcionamiento de los *getters* y los *setters* de dicha clase) y ya dispone de la definición de *Resumen* y *Descripcion*.

Hay que añadir tres propiedades en la zona privada:

- QString departamento
- QString despacho
- QVector<QString> tutorías

Despacho y departamento tendrán el *getter* y el *setter* habitual en la zona pública.

tutorías se ha definido como una lista en la que cada elemento corresponderá con uno de los horarios de tutorías del profesor. Como no es una variable Necesita un método para añadir un elemento a tutorías; un método para acceder a un elemento de tutorías; un método para eliminar un elemento y un método que nos diga cuantos elementos tiene la lista.

La clase Profesor deberá sobrescribir el comportamiento de los métodos virtuales *Resumen* y *Descripcion*. Para ello, veremos más adelante que usaremos la palabra reservada *override*:

```
QString Resumen() override;
```

```
QString descripcion() override;
```

La clase Alumno:

Alumno también hereda de la clase *Persona*, por lo que también hereda los *getters* y los *setters* de dicha clase, *Resumen* y *Descripcion*.

Hay que añadir dos propiedades en la zona privada:

- QString curso
- QVector<QString> asignaturas

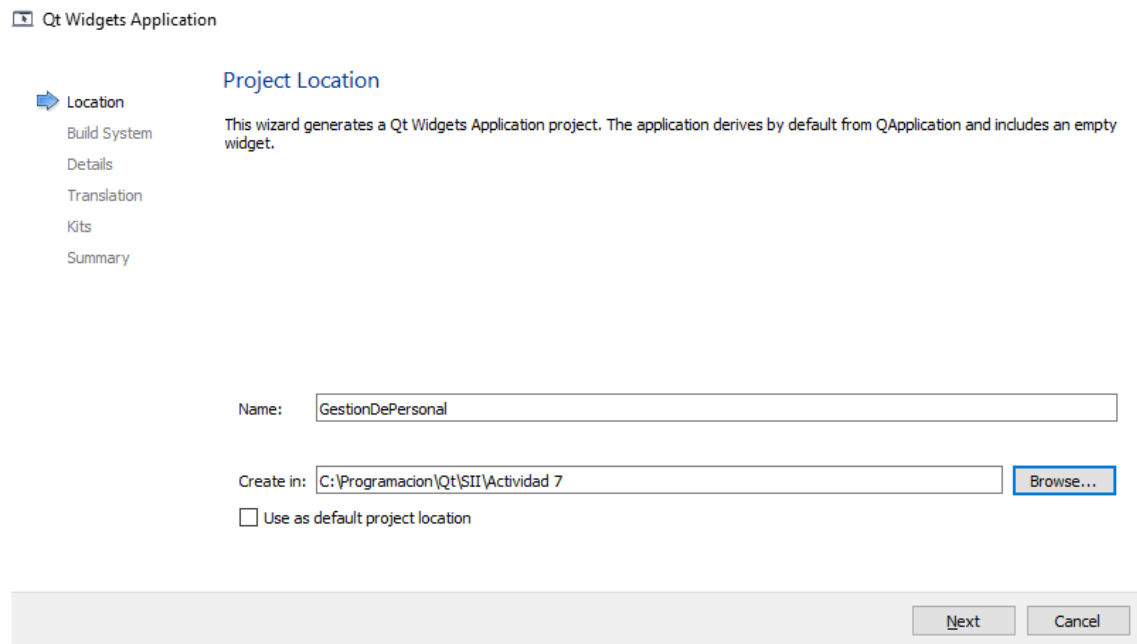
curso tendrá el *getter* y el *setter* habitual en la zona pública.

asignaturas se ha definido como una lista donde cada elemento corresponde con una asignatura. También necesita métodos para añadir un elemento; para acceder a un elemento; para eliminar un elemento y un método que nos diga cuantos elementos tiene la lista.

La clase *Alumno* también deberá sobrescribir el comportamiento de los métodos virtuales *Resumen* y *Descripcion*.

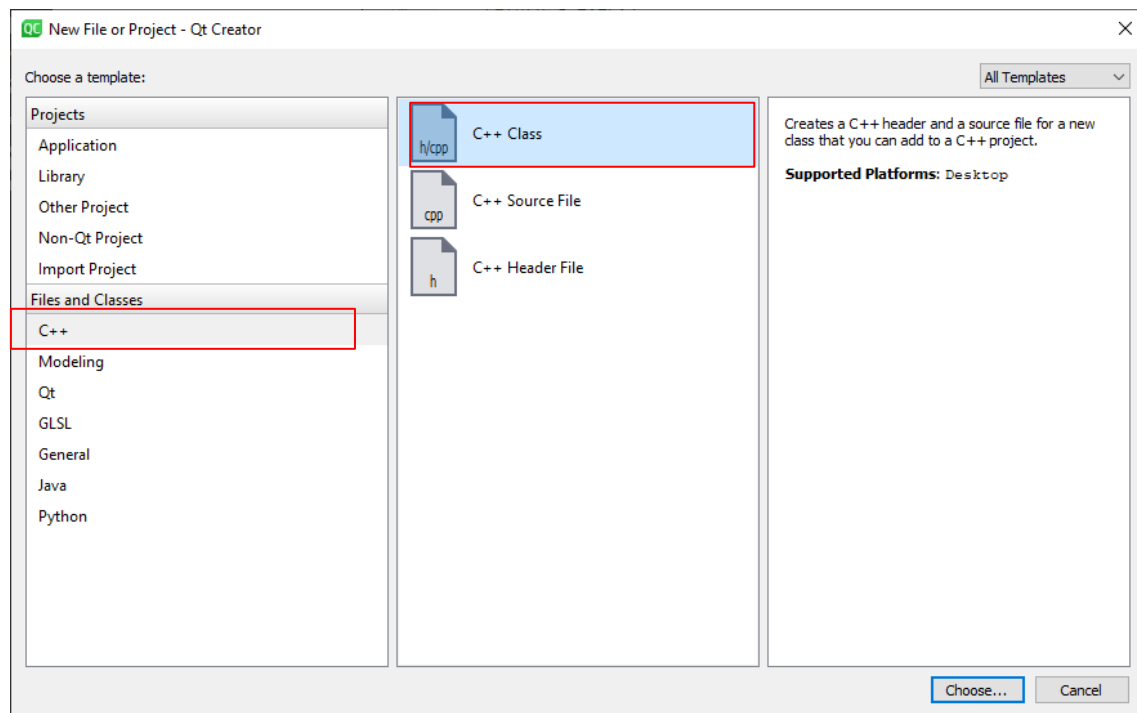
Implementación paso a paso

Crea un proyecto de tipo *Qt Widgets Application*



Crea una clase de C++ llamada **Persona**

File → New file or Project...



C++ Class

✕

➔ Details

Summary

Define Class

Class name:

Base class:

☐ Include QObject

☐ Include QWidget

☐ Include QMainWindow

☐ Include QDeclarativeItem - Qt Quick 1

☐ Include QQuickItem - Qt Quick 2

☐ Include QSharedData

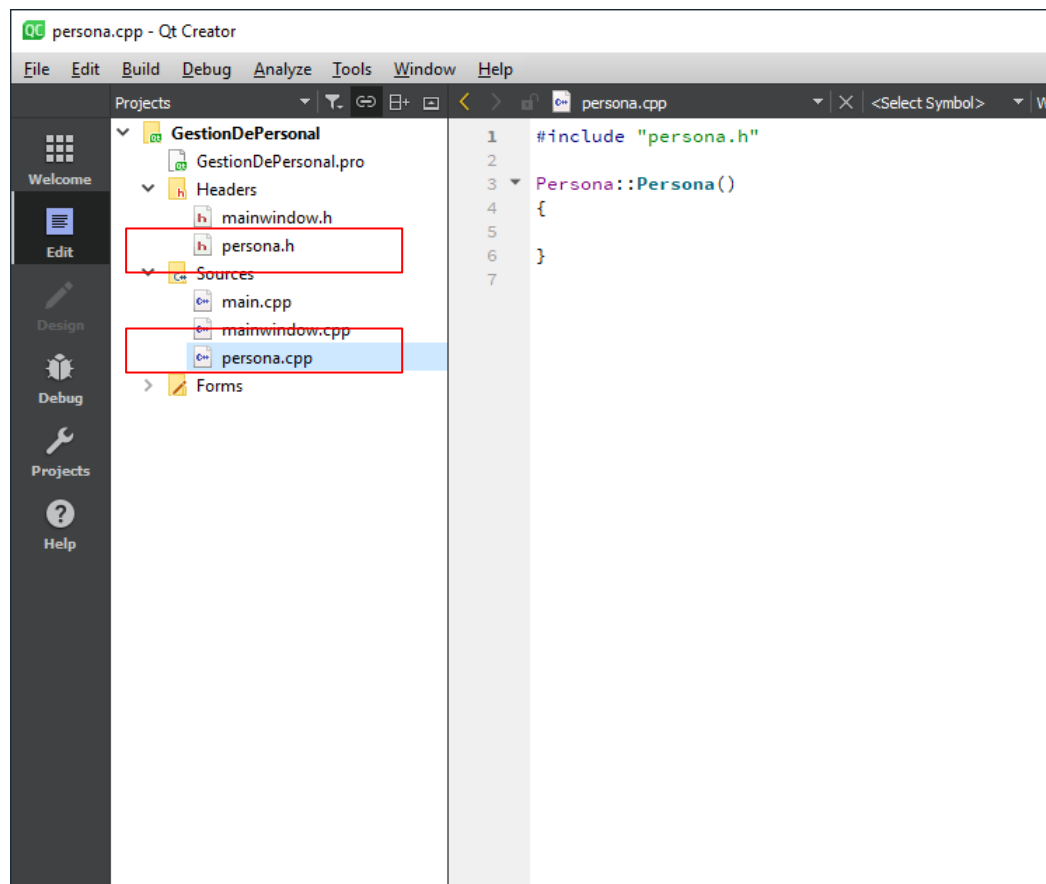
☐ Add Q_OBJECT

Header file:

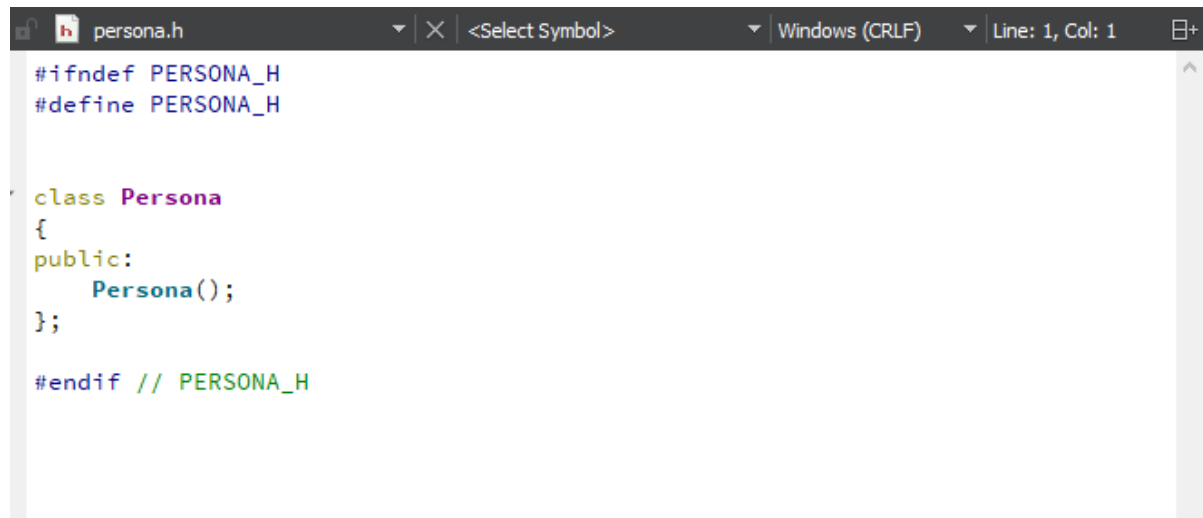
Source file:

Path:

Debe haber generado dos archivos *persona.h* y *persona.cpp*:



Veamos el contenido de *persona.h*:



```
#ifndef PERSONA_H
#define PERSONA_H

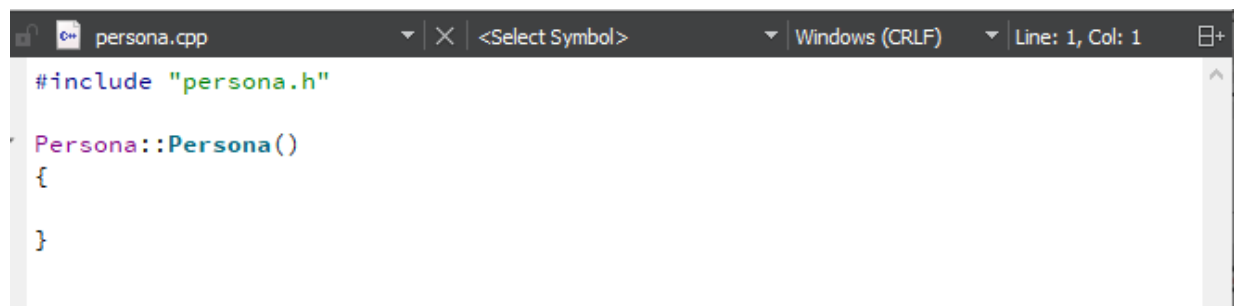
class Persona
{
public:
    Persona();
};

#endif // PERSONA_H
```

En los archivos “.h” se hace la definición de los datos que contendrá cada objeto de la clase y de las cabeceras de cada método pertenecientes a la misma. En el caso de la clase *Persona*, inicialmente solo está definida la declaración del método constructor por defecto *Persona()*

Recuerda que un constructor es un método que tiene exactamente el mismo nombre que la clase, en este caso *Persona*, y que no devuelve ningún tipo de dato. Es decir, delante de la declaración de los constructores no se indica ningún tipo de dato (ni void, ni int, ni nada de nada...)

Veamos el contenido inicial de *persona.cpp*



```
#include "persona.h"

Persona::Persona()
{
}
```

En los archivos “.cpp” es donde implementaremos el comportamiento de cada método de la clase.

En la primera línea vemos que se incluye el archivo *persona.h*. De esta forma el compilador sabe dónde está la definición de la clase *Persona*.

Después vemos la implementación del constructor *Persona()*

NOTA: Os recuerdo que estamos utilizando el operador de ámbito ::

En la definición `Persona::Persona()`

- La primera palabra indica el ámbito (la clase *Persona*). El ámbito se pone en el lado izquierdo del operador ::
- Al lado derecho del operador se indica el miembro de la clase que quiero usar, en este caso el constructor *Persona()*

Vamos a empezar por **la encapsulación de los datos** para la clase *Persona*. Hemos de definir los datos de dicha clase y los métodos para acceder y manipular dichos datos. Los datos los declararemos como privados. Para los métodos deberemos decidir en cada caso si son públicos, protegidos o privados.

En *persona.h* vamos a definir los siguientes datos:

```
#include <QString>

class Persona
{
private:
    QString nombre;
    QString apellidos;
    QString numID;
```

Como estamos usando `QString` deberemos añadir `#include<QString>` al principio del archivo.

Definimos las tres propiedades de tipo `QString` (*numID* podría ser el número del DNI, por lo que puede incluir dígitos y letras, por eso usamos el tipo `QString`).

Ahora vamos a declarar en la sección pública los métodos para acceder y modificar la variable *nombre*.

```
class Persona
{
private:
    QString nombre;
    QString apellidos;
    QString numID;

public:
    Persona();

    QString getNombre();
    void setNombre(QString s);
```

Si os fijáis hemos definido dos métodos. El primero lo usaremos para obtener el valor de la variable, y el segundo se usa para modificar el valor de la variable. A los métodos que se usan

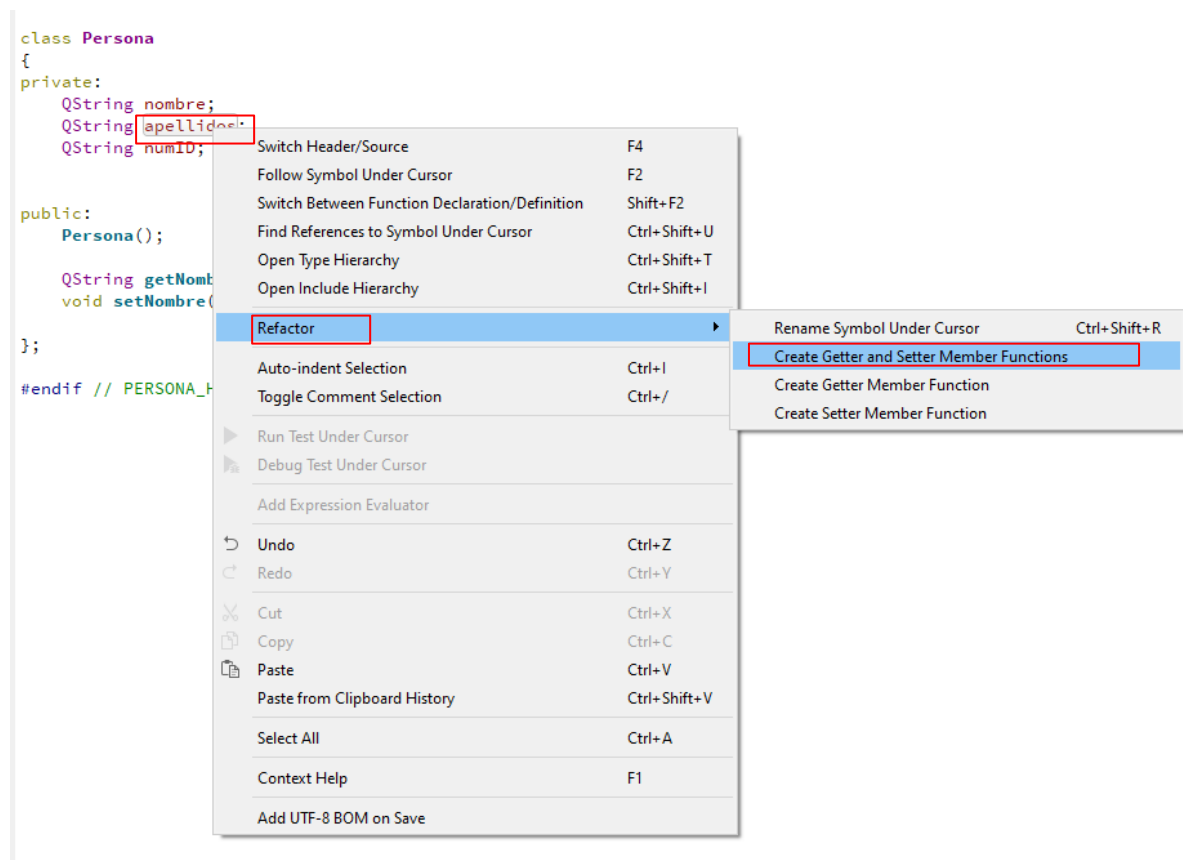
para obtener valores de datos encapsulados se los denomina **getters** y a los métodos que se usan para modificar dichos datos se denominan **setters**.

Ahora, en el archivo *persona.cpp* vamos a programar los dos métodos:

```
QString Persona::getNombre()  
{  
    return nombre;  
}  
  
void Persona::setNombre(QString s)  
{  
    nombre = s;  
}
```

Debido a que en muchas ocasiones los getters y los setters que tenemos que programar no tienen que hacer nada más que lo que vemos en el código anterior, Qt ofrece una forma rápida de generar el *getter* y el *setter* de una variable privada que hayamos declarado en una clase.

Pulsa con el botón derecho del ratón sobre la variable *apellidos* y en el menú contextual elige la opción *refactor* → *Create Getter and Setter Member Functions*



Observa ahora el contenido del archivo *persona.h* y verás que ha añadido:

```
QString getApellidos() const;  
void setApellidos(const QString &value);  
.
```


Ahora vamos a ver el contenido del archivo *persona.cpp* y verás que se ha añadido:

```
▼ QString Persona::getApellidos() const
{
    return apellidos;
}

▼ void Persona::setApellidos(const QString &value)
{
    apellidos = value;
}
```

No hemos tenido que programar esta parte... el propio editor de Qt nos ha hecho el trabajo.

El siguiente paso será encapsular *numID* pero, en este caso, el *getter* y el *setter* van a tener un nombre diferente al de la variable. En vez de *getNumID()* vamos a usar *getNumeroDeIdentificacion()*, por lo que lo vamos a programar manualmente.

persona.h

```
#ifndef PERSONA_H
#define PERSONA_H

#include <QString>

class Persona
{
private:
    QString nombre;
    QString apellidos;
    QString numID;

public:
    Persona();

    QString getNombre();
    void setNombre(QString s);

    QString getApellidos() const;
    void setApellidos(const QString &value);

    QString getNumeroDeIdentificacion();
    void setNumeroDeIdentificacion(QString s);
};

#endif // PERSONA_H
```

Persona.cpp

```
#include "persona.h"

QString Persona::getNombre()
{
    return nombre;
}

void Persona::setNombre(QString s)
{
    nombre = s;
}

QString Persona::getApellidos() const
{
    return apellidos;
}

void Persona::setApellidos(const QString &value)
{
    apellidos = value;
}

QString Persona::getNumeroDeIdentificacion()
{
    return numID;
}

void Persona::setNumeroDeIdentificacion(QString s)
{
    numID = s;
}

Persona::Persona()
{
    setNombre("");
    setApellidos("");
    setNumeroDeIdentificacion("");
}
```

Observa que el constructor ya no está vacío. En el constructor hemos asignado valores iniciales a las tres variables.

En esta clase, como no tenemos ninguna variable definida como puntero no es necesario declarar el destructor.

Si miramos en la definición del problema, podemos ver que la clase *Persona* debe definir los métodos virtuales *Resumen()* y *Descripción()*.

Repasa la práctica 6 para recordar lo que es un método virtual, pero, básicamente, un método virtual es un método que será modificado por las clases derivadas. En nuestro caso, los métodos *Resumen* y *Descripción* serán modificados en las clases derivadas *Profesor* y *Alumno*.

El método *resumen* devolverá los apellidos, el nombre y el número de identificación en una única línea de texto.

El método *descripcion* devolverá los mismos datos, pero en una cadena de caracteres con varias líneas (el carácter de salto de línea es '\n').

En el archivo *persona.h* declararemos los dos métodos así:

```
class Persona
{
private:
    QString nombre;
    QString apellidos;
    QString numID;

public:
    Persona();

    QString getNombre();
    void setNombre(QString s);

    QString getApellidos() const;
    void setApellidos(const QString &value);

    QString getNumeroDeIdentificacion();
    void setNumeroDeIdentificacion(QString s);

    virtual QString Resumen();
    virtual QString Descripcion();
};
```

Y programaremos dichos métodos en el archivo *persona.cpp*

```
QString Persona::Resumen()
{
    QString res = getApellidos() + ", " + getNombre() + " (" + getNumeroDeIdentificacion() + ")";

    return res;
}

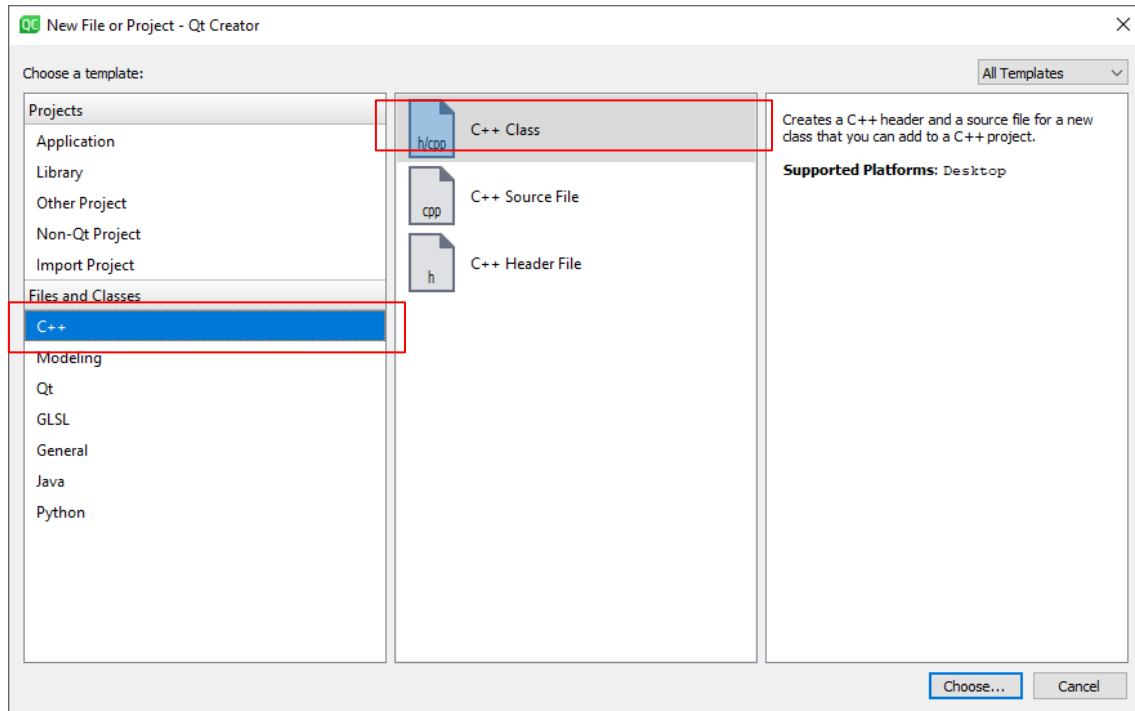
QString Persona::Descripcion()
{
    QString res;

    res = getNumeroDeIdentificacion() + "\n";
    res += getApellidos() + ", " + getNombre() + "\n";
    return res;
}
```

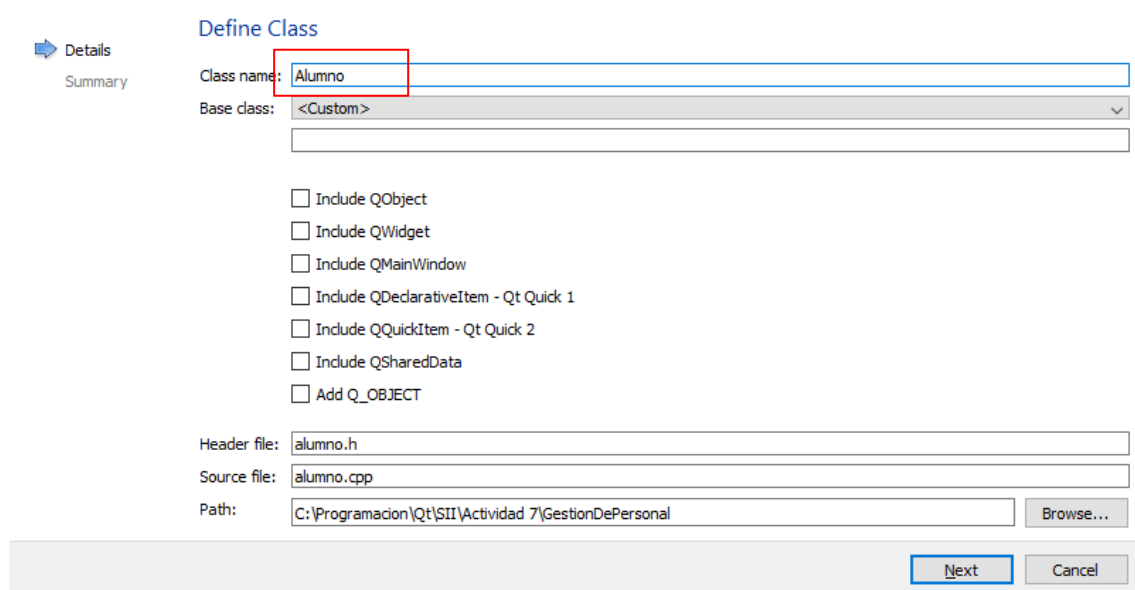
Clase Alumno

Vamos a crear una nueva clase llamada *Alumno* y vamos a definirla como una clase derivada de *Persona*.

Recuerda: Menú File → New File or Project...



C++ Class



Ahora, tendremos creados los archivos *alumno.h* y *alumno.cpp*

Lo primero que vamos a hacer es indicar que *Alumno* hereda de la clase *Persona*, por lo que en la definición de la clase *Alumno* tenemos que indicar que su clase base es la clase *Persona*

```
#ifndef ALUMNO_H
#define ALUMNO_H

#include "persona.h"

class Alumno : public Persona
{
public:
    Alumno();
};

#endif // ALUMNO_H
```

Observa que hemos añadido el “include” de la clase *Persona*, y en la línea que empieza por “class” hemos declarado la herencia con “: public *Persona*”

Como ya hemos visto en la práctica 6, al heredar la clase *Persona* con el modificador *public* estamos heredando todos los miembros de la clase *Persona* con exactamente el mismo tipo de acceso. Lo que es público en *Persona* será público en *Alumno*, lo que es protegido en *Persona* será protegido en *Alumno*. Pero ojo, lo que es privado en *Persona* no será accesible en *Alumno*. Lo que significa que para acceder a *nombre*, *apellidos* y *numID* hemos de usar sus *getters* y *setters* que sí son públicos.

A parte del nombre, los apellidos y del DNI/NIE, que hemos heredado de la clase base, la clase *Alumno* debe tener dos datos adicionales. El curso al que está matriculado y la lista de las asignaturas a las que está matriculado. Además debe cambiar el funcionamiento de los métodos virtuales heredados *resumen()* y *descripcion()*.

Lo primero que vamos a hacer es **añadir la propiedad *curso*** de tipo *QString*, y vamos a encapsularla de la misma forma que hicimos con *nombre*, *apellidos*, etc.

Nota: He considerado que *curso* sea de tipo *QString* pensando que el curso podría tener símbolos distintos a los dígitos, por ejemplo: “1ºA”. Pero podéis declararlo como *int*

Veamos primero el archivo *alumno.h*

```
#ifndef ALUMNO_H
#define ALUMNO_H

#include "persona.h"
#include <QString>
#include <QList>

class Alumno : public Persona
{
    QString curso;

public:
    Alumno();

    QString getCurso() const;
    void setCurso(const QString &value);
};

#endif // ALUMNO_H
```

Y ahora el archivo *alumno.cpp*

```
#include "alumno.h"

QString Alumno::getCurso() const
{
    return curso;
}

void Alumno::setCurso(const QString &value)
{
    curso = value;
}

Alumno::Alumno()
{
}
```

Ahora vamos a **añadir la lista de asignatura**, al ser una lista, no vamos a utilizar el *getter* y el *setter* habitual. En su lugar vamos a añadir los cuatro métodos más básicos para la gestión de listas:

- Añadir un elemento: al final de la lista en nuestro caso
- Leer un elemento: usaremos un índice para indicar qué elemento queremos leer
- Borrar un elemento: usaremos un índice para indicar que elemento queremos borrar
- Leer el tamaño de la lista: nos dirá el número de elementos de la lista

Vamos a utilizar la clase `QList<T>` de Qt para declarar nuestra lista de asignaturas (donde T lo podemos sustituir por cualquier tipo de datos). Lo haremos de la siguiente manera:

Archivo *alumno.h*

```
#ifndef ALUMNO_H
#define ALUMNO_H

#include "persona.h"
#include <QString>
#include <QList>

class Alumno : public Persona
{
    QString curso;

    QList<QString> listaAsignaturas;
```

Vamos a declarar los cuatro métodos:

```
public:
    Alumno();

    QString getCurso() const;
    void setCurso(const QString &value);

    void appendAsignatura(QString asig);
    QString getAsignatura(int i);
    void deleteAsignatura(int i);
    int countAsignatura();
};
```

Ahora vamos a programar dichos métodos en el archivo *alumno.cpp*

```
void Alumno::appendAsignatura(QString asig)
{
    listaAsignaturas.append(asig);
}

QString Alumno::getAsignatura(int i)
{
    if (i >= 0 && i < countAsignatura())
    {
        return listaAsignaturas[i];
    }
    else
    {
        return "";
    }
}

void Alumno::deleteAsignatura(int i)
{
    if (i >= 0 && i < countAsignatura())
    {
        listaAsignaturas.removeAt(i);
    }
}

int Alumno::countAsignatura()
{
    return listaAsignaturas.count();
}
```

Vamos a **reescribir** ahora el funcionamiento del método **resumen()** que hemos heredado de *Persona*

Os recuerdo que podemos modificar la programación de un método virtual de la clase base en la clase derivada, utilizando un algoritmo distinto.

En nuestro ejemplo la clase base es *Persona* y la clase derivada es *Alumno*. El método virtual que queremos cambiar es *Resumen()* por lo que habrá dos versiones de *Resumen*:

```
Persona::Resumen()
Alumno::Resumen()
```

Cuando se invoque el método *Resumen()*, los objetos de la clase *Persona* invocarán *Persona::Resumen()*. Los objetos de la clase *Alumno* invocarán *Alumno::Resumen()*

El cambio será mínimo. El método *resumen* de la clase *Persona* nos devuelve una línea con: Apellidos, Nombre (DNI) ... (por ejemplo: "Martínez González, Juan Carlos (12345678K)")

En la clase *Alumno* queremos que el resumen tenga la siguiente forma:

"Alumno: Apellidos, Nombre (DNI)"

Por lo que hemos de añadir el texto "Alumno: " al principio de la línea.

Lo primero es declarar el método en el archivo *alumno.h*

```
1  #ifndef ALUMNO_H
2  #define ALUMNO_H
3
4  #include "persona.h"
5  #include <QString>
6  #include <QList>
7
8  class Alumno : public Persona
9  {
10     QString curso;
11
12     QList<QString> listaAsignaturas;
13
14 public:
15     Alumno();
16
17     QString getCurso() const;
18     void setCurso(const QString &value);
19
20     void appendAsignatura(QString asig);
21     QString getAsignatura(int i);
22     void deleteAsignatura(int i);
23     int countAsignatura();
24
25     QString Resumen() override;
26 };
27
28 #endif // ALUMNO_H
29
```

Observa que hemos indicado con la palabra "override" que queremos modificar el método virtual *Resumen*.

Vamos a programar el nuevo método *Resumen* en el archivo *alumno.cpp*

```
QString Alumno::Resumen()
{
    QString resumenPersona = Persona::Resumen();

    return "Alumno: " + resumenPersona;
}
```

Vamos a centrarnos en la primera línea del cuerpo del método:

```
QString resumenPersona = Persona::Resumen();
```

Os recuerdo que el *resumen* de la clase *alumno* lo que tiene que hacer es añadir el texto “Alumno: “ al principio de la línea, por lo que, lo que estamos haciendo es invocar la versión del método *Resumen()* de la clase *Persona*. Es decir, *Persona::Resumen()*

resumenPersona, será por lo tanto el texto “Apellidos, Nombre (DNI)”

Observa que la siguiente línea del cuerpo de *Alumno::Resumen()* añade el texto “Alumno: “ delante:

```
return “Alumno: “ + resumenPersona;
```

Recuerda que, cuando en la clase derivada decidimos cambiar el comportamiento de un método virtual que hemos heredado de la clase base, la nueva versión del método es totalmente distinta a la versión anterior.

Si queremos que el método haga lo mismo que hacía y alguna cosa más podemos invocar la versión de la clase base usando:

```
CBase::método(parámetros);
```

En nuestro ejemplo invocamos *Persona::Resumen()* dentro del método *Alumno::Resumen()*

El segundo método virtual que tenemos que **reescribir** es ***Descripción()***

Añadimos, en *alumno.h*

```
QString getAsignatura(int i);  
void deleteAsignatura(int i);  
int countAsignatura();  
  
QString Resumen() override;  
QString Descripción() override;  
};  
  
#endif // ALUMNO_H
```

Ahora vamos a programarlo en *alumno.cpp*.

Esta vez vamos a reescribir el método completo, ya que los cambios son mayores. (No vamos a invocar a *Persona::Descripción()*)

```

▼ QString Alumno::Descripcion()
{
    QString res;

    res = "Alumno: \n";
    res += "DNI/NIE: " + getNumeroDeIdentificacion() + "\n";
    res += "Apellidos, Nombre: " + getApellidos() + ", " + getNombre() + "\n";
    res += "Curso: " + getCurso() + "\n";
    res += "Asignaturas matriculadas: \n";

    ▼
    for (int i = 0; i < countAsignatura(); i++)
    {
        res += "    - " + getAsignatura(i) + "\n";
    }

    return res;
}

```

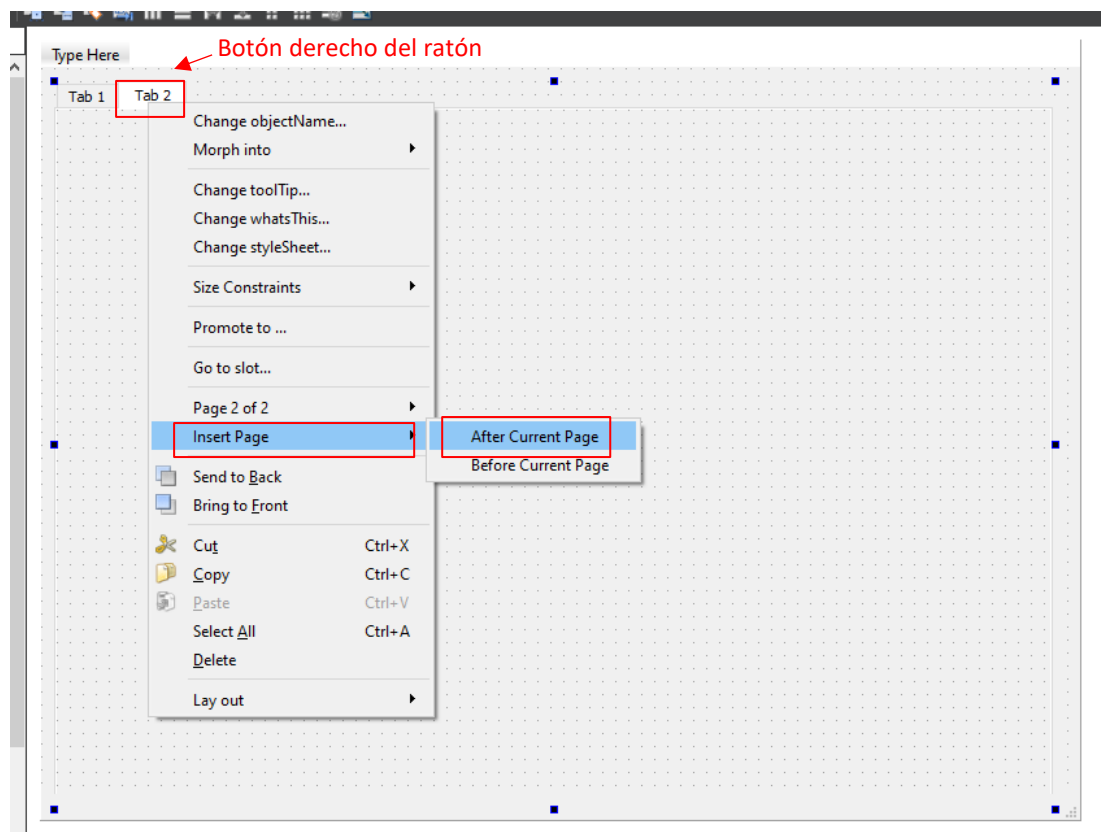
Observa que, como las asignaturas son una lista, hemos de recorrerla mediante un bucle para añadirlas todas a la descripción.

Creación de la interfaz de usuario

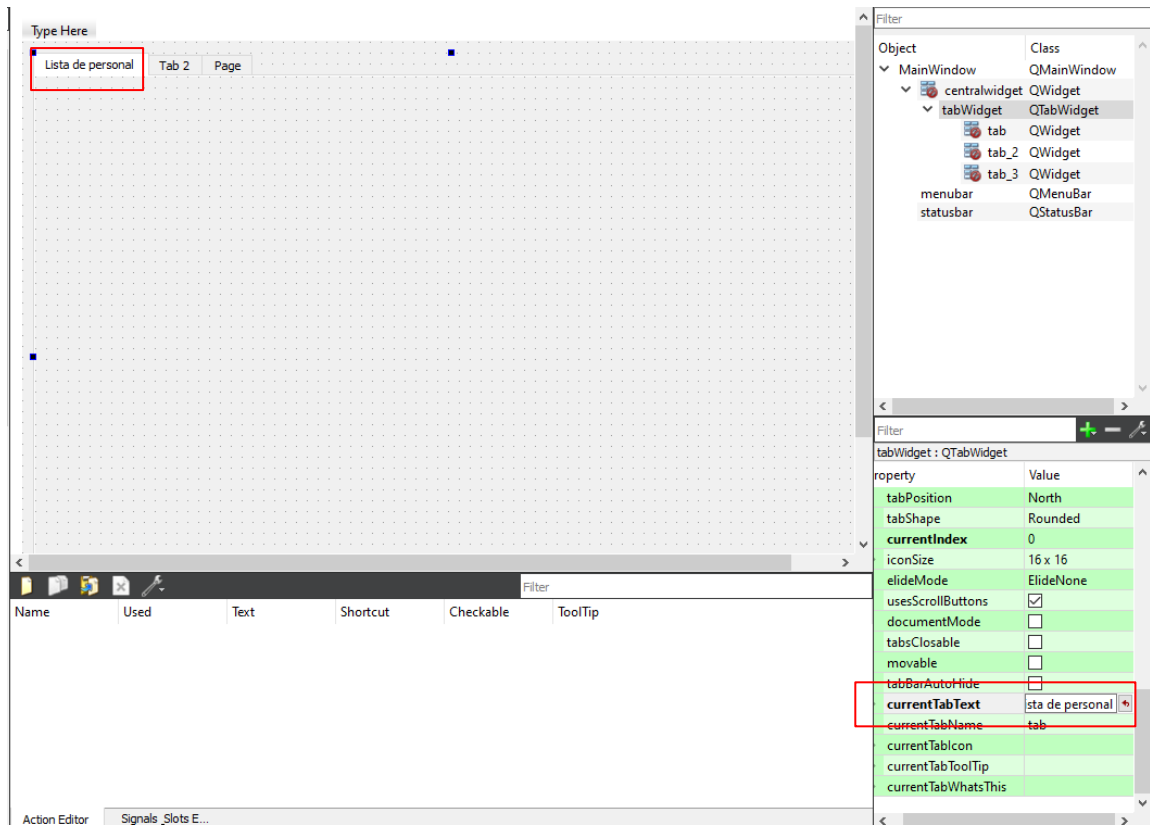
Antes de programar la clase Profesor vamos a crear una interfaz de usuario para probar la clase *Alumno* que acabamos de programar:

Abre **mainwindow.ui** en el editor y añade un Widget del tipo “Tab Widget”

El *Tab Widget* por defecto tiene dos pestañas, pero nosotros queremos que tenga tres. Así que, vamos a añadir una tercera pestaña de la siguiente forma:



Vamos a cambiar el nombre de cada una de las pestañas. Primero tienes que pulsar sobre la pestaña que quieres modificar, y en propiedades modificar la propiedad *currentTabText*



Cambia así los nombres de las tres pestañas por:

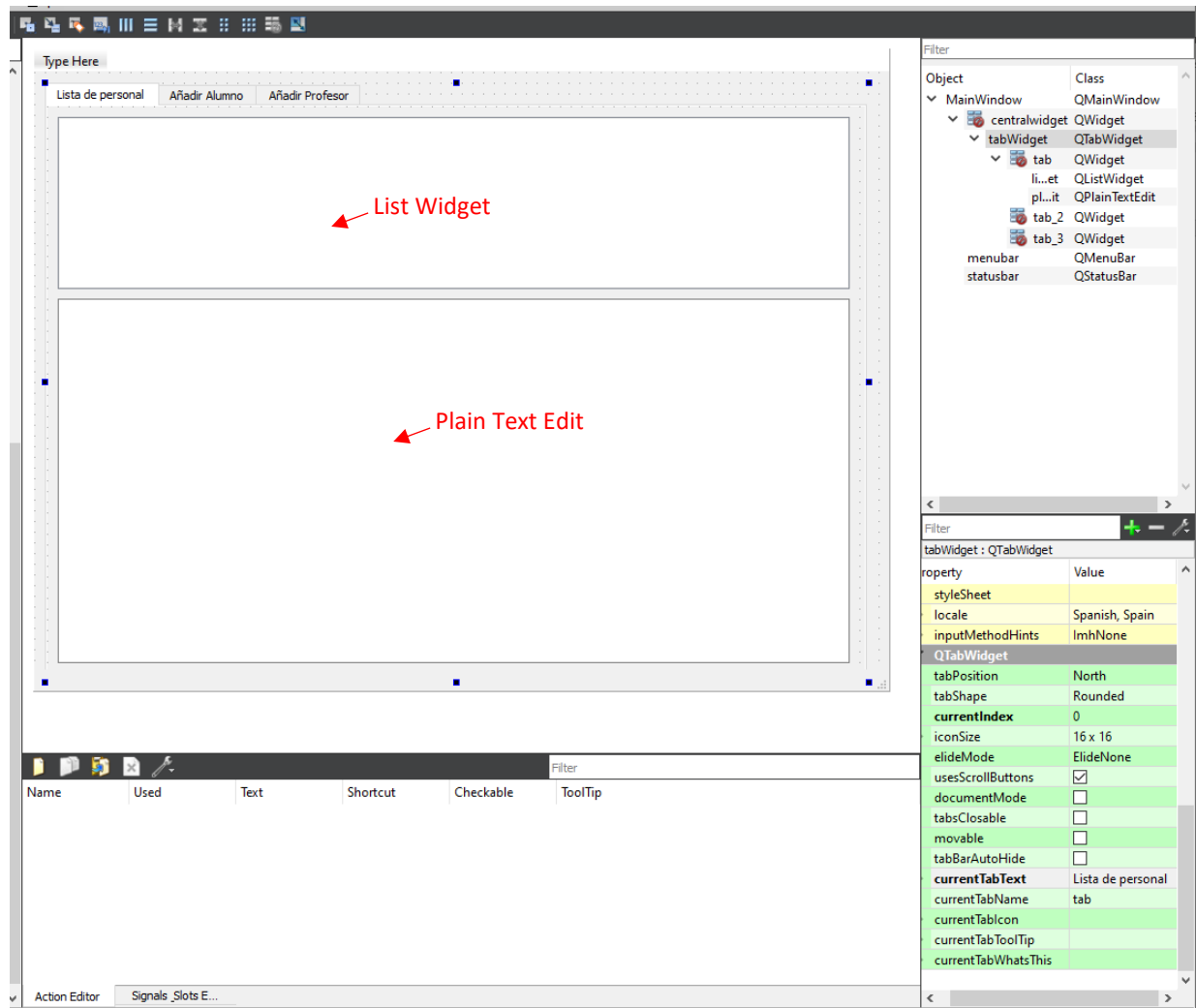
- Lista de personal
- Añadir Alumno
- Añadir Profesor

Ahora vamos a trabajar con las dos primeras pestañas.

Empecemos por la **pestaña “Lista de personal”**

Vamos a añadir dos componentes:

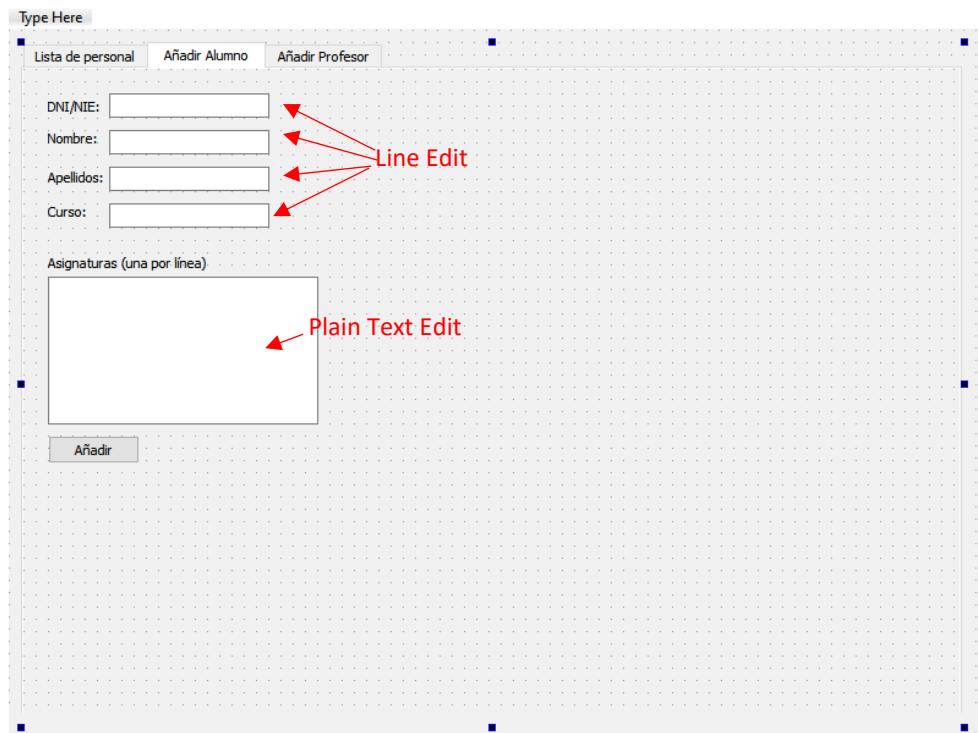
- List Widget
- Plain Text Edit



Los nombres que le he asignado a los dos elementos son:

- listWidget_personal
- plainTextEdit_detalle

En la **pestaña *Añadir Alumno*** vamos a diseñar la siguiente interfaz:



El nombre que le he asignado a cada Widget de entrada es:

- lineEdit_NI
- lineEdit_Nombre
- lineEdit_Apellidos
- lineEdit_Curso
- plainTextEdit_Asignaturas

Y el nombre del botón:

- pushButton_AnyadirAlumno

Vamos a preparar la clase MainWindow para que pueda mantener una lista de las personas creadas (*alumnos*, por ahora).

En *mainwindow.h* añadimos lo siguiente:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QList>
#include "persona.h"
#include "alumno.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

    QList<Persona *> listaDePersonal;

    void mostrarLista();
};
#endif // MAINWINDOW_H

```

Como podéis ver, hemos añadido los *includes* necesarios en la parte de arriba. Además hemos añadido una lista de punteros a *Persona* y un método privado llamado *mostrarLista()* que usaremos para rellenar *listWidget_personal* de la pestaña *Lista de personal* con la información de la *listaDePersonal*

En *mainwindow.cpp* implementamos el método *mostrarLista()*

```

void MainWindow::mostrarLista()
{
    ui->listWidget_personal->clear();

    for (int i = 0; i < listaDePersonal.count(); i++)
    {
        ui->listWidget_personal->addItem(listaDePersonal[i]->Resumen());
    }
}

```

Observa que lo que vamos a añadir a la *listWidget_personal* es el resumen de cada elemento de la *listaDePersonal*.

Para la **pestaña “Añadir alumno”**, vamos a crear un SLOT de Qt para el botón añadir que hay en la parte de abajo:

Declaramos el slot en *mainwindow.h*:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

    QList<Persona *> listaDePersonal;

    void mostrarLista();

public slots:
    void anyadirAlumno();
};
```

Y vamos a conectar la señal `pressed()` del botón al slot, añadiendo la siguiente línea en el constructor de *MainWindow*:

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pushButton_AnyadirAlumno, SIGNAL(pressed()), this, SLOT(anyadirAlumno()));
}
```


Ahora, vamos a implementar el SLOT *anyadirAlumno()*:

```
void MainWindow::anyadirAlumno()
{
    Alumno *alumno = new Alumno;

    alumno->setNumeroDeIdentificacion(ui->lineEdit_NI->text());
    alumno->setNombre(ui->lineEdit_Nombre->text());
    alumno->setApellidos(ui->lineEdit_Apellidos->text());
    alumno->setCurso(ui->lineEdit_Curso->text());

    QString asignaturas = ui->plainTextEdit_Asignaturas->toPlainText();

    /*
     * La siguiente línea del código (método split) divide el texto guardado
     * en "asignaturas" en líneas de texto.
     * Cada línea de texto encontrada la guarda en un QString, y guarda todas las
     * líneas encontradas en una lista de tipo QStringList
     */
    QStringList lineas = asignaturas.split(QRegExp("[\\r\\n]"),QString::SkipEmptyParts);

    /*
     * Cada línea encontrada es una asignatura, por lo que las recorro con un bucle
     * y las añado a alumno
     */
    for (int i = 0; i < lineas.count(); i++)
    {
        alumno->appendAsignatura(lineas[i]);
    }

    listaDePersonal.append(alumno);

    mostrarLista();

    ui->tabWidget->setCurrentIndex(0);
}
```

La parte principal de este método consiste en crear el objeto de la clase *Alumno* y rellenar todos sus datos (usando los *setters*)

Las últimas tres líneas, lo que hacen es, añadir el alumno a la lista, mostrar la lista en *listWidget_personal* (invocando *mostrarLista()*) y la última línea hace que se muestre la primera pestaña, para que podamos ver que se ha añadido el alumno a la lista.

En lo que se refiere a los alumnos, sólo nos queda hacer que cuando pulsemos sobre un nombre en la lista de alumnos de la primera pestaña, abajo nos muestre los detalles.

En *mainwindow.h* añadimos el nuevo SLOT al cual le vamos a conectar la señal *currentRowChanged(int)*. Como la señal tiene un parámetro entero en el que se nos indica el número de fila que hemos seleccionado, al nuevo SLOT le voy a poner un parámetro del mismo tipo, para recibir dicho dato.

```
public slots:
    void anyadirAlumno();

    void mostrarDetalles(int fila);

};
```

Vamos a implementarlo en *mainwindow.cpp*:

```
void MainWindow::mostrarDetalles(int i)
{
    if (i < 0 || i >= listaDePersonal.count())
    {
        ui->plainTextEdit_detalles->clear();
    }
    else
    {
        ui->plainTextEdit_detalles->setPlainText(listaDePersonal[i]->Descripcion());
    }
}
```

Solo nos queda añadir el *connect* en el constructor:

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pushButton_AnyadirAlumno, SIGNAL(pressed()), this, SLOT(anyadirAlumno()));
    connect(ui->listWidget_personal, SIGNAL(currentRowChanged(int)), this, SLOT(mostrarDetalles(int)));
}
```

Ya podéis probar el proyecto.

Ya puedes probar a añadir alumnos a la lista de personal y puedes seleccionarlos en la lista para ver los detalles de cada uno.

Observa que la lista de personal la hemos declarado como una lista de objetos de la clase *persona* (es decir, del tipo `QList<Persona *>`). El polimorfismo nos permite añadir a esta lista objetos de clases derivadas de *Persona*. En este caso añadimos objetos de la clase *Alumno*.

Clase Profesor

Vamos a crear una clase llamada *Profesor*, de la misma forma que hemos creado la clase *Alumno*. La clase *Profesor* debe heredar de la clase *Persona*.

Tenemos que añadir las propiedades: departamento, despacho y una lista para los horarios de tutorías, y hemos de crear los métodos *getter*, los métodos *setter* y los métodos para la gestión de la lista de tutorías.

Además, hemos de redefinir los métodos virtuales *resumen* y *descripcion*.

Aquí podéis ver como quedaría el fichero *profesor.h*:

```
#ifndef PROFESOR_H
#define PROFESOR_H

#include "persona.h"
#include <QString>
#include <QList>

class Profesor : public Persona
{
    QString departamento;
    QString despacho;
    QList<QString> listaTutorias;

public:
    Profesor();

    QString getDepartamento() const;
    void setDepartamento(const QString &value);

    QString getDespacho() const;
    void setDespacho(const QString &value);

    void appendTutoria(QString asig);
    QString getTutoria(int i);
    void deleteTutoria(int i);
    int countTutorias();

    QString Resumen() override;
    QString Descripcion() override;
};

#endif // PROFESOR_H
```

La implementación de estos métodos se hará de forma similar a como se ha hecho en la clase Alumno:

Getters y setters de *departamento* y de *despacho*:

```
QString Profesor::getDepartamento() const
{
    return departamento;
}

void Profesor::setDepartamento(const QString &value)
{
    departamento = value;
}

QString Profesor::getDespacho() const
{
    return despacho;
}

void Profesor::setDespacho(const QString &value)
{
    despacho = value;
}
```

Métodos para la gestión de las tutorías:

```
void Profesor::appendTutoria(QString asig)
{
    listaTutorias.append(asig);
}

QString Profesor::getTutoria(int i)
{
    if (i >= 0 && i < countTutorias())
    {
        return listaTutorias[i];
    }
    else
    {
        return "";
    }
}

void Profesor::deleteTutoria(int i)
{
    if (i >= 0 && i < countTutorias())
    {
        listaTutorias.removeAt(i);
    }
}

int Profesor::countTutorias()
{
    return listaTutorias.count();
}
```

La nueva implementación de los métodos *Resumen* y *Descripcion*:

```
QString Profesor::Resumen()
{
    QString resumenPersona = Persona::Resumen();

    return "Profesor: " + resumenPersona;
}

QString Profesor::Descripcion()
{
    QString res;

    res = "Profesor: \n";
    res += "DNI/NIE: " + getNumeroDeIdentificacion() + "\n";
    res += "Apellidos, Nombre: " + getApellidos() + ", " + getNombre() + "\n";
    res += "Departamento: " + getDepartamento() + "\n";
    res += "Despacho: " + getDespacho() + "\n";
    res += "Horario de tutorías: \n";

    for (int i = 0; i < countTutorias(); i++)
    {
        res += "    - " + getTutoria(i) + "\n";
    }

    return res;
}
```

Ahora vamos a modificar la interfaz de usuario:

En la pestaña **Añadir Profesor** diseñaremos una interfaz de usuario similar a esta:

The screenshot shows a Qt Designer window titled 'Type Here'. Inside, there are three tabs: 'Lista de personal', 'Añadir Alumno', and 'Añadir Profesor'. The 'Añadir Profesor' tab is selected. The form contains the following elements:

- Five text input fields stacked vertically, each with a label to its left: 'DNI/NIE:', 'Nombre:', 'Apellidos:', 'Departamento:', and 'Depacho:'.
- A text area labeled 'Horario de tutorías (uno por línea)'.
- A 'Añadir' button located below the text area.

El nombre que le he asignado a cada Widget de entrada es:

- `lineEdit_NI_Prof`
- `lineEdit_Nombre_Prof`
- `lineEdit_Apellidos_Prof`
- `lineEdit_Departamento_Prof`
- `lineEdit_Despacho_Prof`
- `plainTextEdit_Tutorias_Prof`

Y el nombre del botón:

- `pushButton_AnyadirProfesor`

Vamos a programar el funcionamiento del botón, para lo que hemos de crear un SLOT nuevo en la clase *MainWindow*

En *MainWindow.h*, en la sección de *public slots*:

```
public slots:
    void anyadirAlumno();
    void anyadirProfesor();

    void mostrarDetalles(int fila);

};
```

En *MainWindow.cpp*, en el constructor añadimos el *connect*:

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pushButton_AnyadirAlumno, SIGNAL(pressed()), this, SLOT(anyadirAlumno()));
    connect(ui->pushButton_AnyadirProfesor, SIGNAL(pressed()), this, SLOT(anyadirProfesor()));
    connect(ui->listWidget_personal, SIGNAL(currentRowChanged(int)), this, SLOT(mostrarDetalles(int)));
}
```

Y en el mismo archivo implementamos el nuevo SLOT:

```
void MainWindow::anyadirProfesor()
{
    Profesor *profesor = new Profesor;

    profesor->setNumeroDeIdentificacion(ui->lineEdit_NI_Prof->text());
    profesor->setNombre(ui->lineEdit_Nombre_Prof->text());
    profesor->setApellidos(ui->lineEdit_Apellidos_Prof->text());
    profesor->setDepartamento(ui->lineEdit_Departamento_Prof->text());
    profesor->setDespacho(ui->lineEdit_Despacho_prof->text());

    QString tutorias = ui->plainTextEdit_Tutorias_Prof->toPlainText();

    /*Lo separamos por líneas*/
    QStringList lineas = tutorias.split(QRegExp("[\\r\\n]"),QString::SkipEmptyParts);

    for (int i = 0; i < lineas.count(); i++)
    {
        profesor->appendTutoria(lineas[i]);
    }

    /* Insertamos el profesor al principio de la lista de personal: */
    listaDePersonal.prepend(profesor);

    mostrarLista();

    ui->tabWidget->setCurrentIndex(0);
}
```

Ya puedes probar el programa. Prueba a añadir profesores y alumnos a la lista. En la lista principal selecciónalos para ver los detalles de cada Persona. Verás que para cada persona se muestra la información correctamente.

El **polimorfismo** me ha permitido crear una única lista de personas con `QList<Persona *>`, donde he añadido tanto profesores como alumnos. Os recuerdo que se puede asignar una clase derivada a una variable de la clase base.

Voy a intentar explicar ahora, cómo funcionan los métodos virtuales junto al polimorfismo. Cuando tenemos un método virtual dentro de un árbol de clases, las clases derivadas pueden modificar la implementación de dicho método. Tal y como hemos hecho con `Resumen()` y `Descripcion()`. Esto significa que tenemos varias versiones de dicho método. En nuestro programa, para el método `Resumen()`, tenemos tres versiones: `Persona::Resumen()`, `Alumno::Resumen()`, `Profesor::Resumen()`. En una variable de tipo **Persona *** (puntero a persona) podemos asignar tanto objetos de tipo `Persona`, como objetos de tipo `Alumno` o `Profesor`, gracias al polimorfismo. Voy a continuar con el ejemplo del método `Resumen()` de nuestro programa. Si una variable de tipo **Persona*** ejecuta el método `Resumen()` (por ejemplo, si `p1` es de tipo `Persona *` : `p1->Resumen()`). C++, en tiempo de ejecución del programa, comprueba a qué clase pertenece el objeto apuntado por la variable y selecciona el método virtual correcto. En nuestro ejemplo, si el objeto apuntado por `p1` es de la clase `Alumno` ejecutará el método `Alumno::Resumen()`; si el objeto apuntado por `p1` es de la clase `Profesor` ejecutará el método `Profesor::Resumen()`; y si es de la clase `Persona` ejecutará la versión del método resumen de la clase `Persona`, `Persona::Resumen()`.

En nuestro programa, si nos fijamos en el método: `void MainWindow::mostrarDetalles(int i)`, Vemos que invocamos `listaDePersonal[i]->Descripcion()`, para obtener la descripción de una persona. Como descripción es virtual, el polimorfismo más los métodos virtuales hacen que C++ ejecute siempre la versión de `Descripcion()` correcta para el objeto que ocupe la posición `i` de la lista (según sea `Profesor` o `Alumno`).

Si todo funciona correctamente ya puedes pasar a realizar el segundo ejercicio en el siguiente apartado.

Proyecto 2: Programa de gestión de DNIs y NIEs.

En este proyecto, se desea implementar un programa para gestionar la información contenida en los carnés de identidad DNI y NIE. Sólo añadiremos la información que hay en el anverso de dichos carnés.

Aquí tenéis los modelos:



Debéis identificar aquellos aspectos que son comunes a ambos documentos y crear una jerarquía de clases.

En dichas clases debéis incluir una función que nos diga si el número de DNI o NIE es correcto.

- El número del DNI es correcto si tiene 8 dígitos y una letra
- El número del NIE comienza con una letra, que puede ser la X, la Y o la Z, seguido de 7 dígitos y una letra al final.
- No tenéis que comprobar si la letra final es la correcta, sólo que esté presente.

Implementar un programa similar al de gestión de personal que hemos hecho anteriormente. El nuevo programa gestionará los documentos de identidad.

No tenéis que generar gráficamente los carnés. Solo hay que gestionar la información que hay en los carnés (nombre, apellidos, número, etc.)