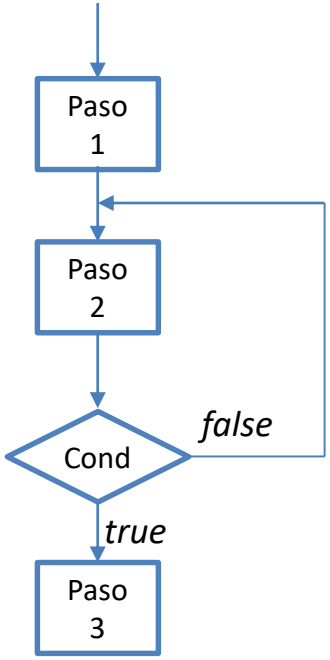
Programación orientada a eventos en C++ con Qt5

Ignacio Alvarez – Septiembre 2018

Programación lineal vs programación orientada a eventos

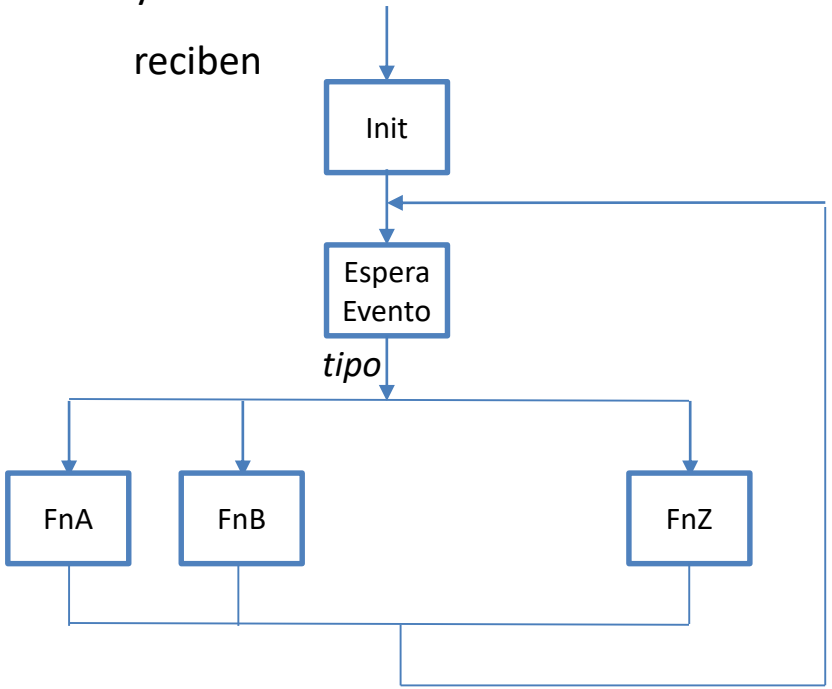
Programación Lineal

1) El programador decide el orden de los acontecimientos
2) Algunos acontecimientos pueden ser tratados como interrupciones



Programación Orientada a eventos

1) Los eventos tienen un orden impredecible
2) El programador define qué función debe tratar cada evento (Init).
3) El programa simplemente espera eventos y llama a la función adecuada cuando se reciben



Programación orientada a eventos en Qt5

Bucle de eventos

1) La función exec() de la clase QCoreApplication ejecuta el bucle de eventos
2) Se debe crear un objeto de la clase QCoreApplication (u otra derivada) en main() y llamar a su función exec().

```
#include <QCoreApplication>

int main(int argc, char *argv[])
{
    // Otras variables e inicializaciones
    QCoreApplication a(argc, argv);

    return a.exec();
}
```

Clases que emiten y sirven eventos

1) Todas las clases derivadas de QObject pueden emitir y recibir eventos
2) Un evento se envía mediante una función tipo **signal**
3) Un evento se sirve mediante una función tipo **slot**

```
#ifndef MICLASE_H
#define MICLASE_H

#include <QObject>

class MiClase : public QObject
{
    Q_OBJECT
public:
    explicit MiClase(QObject *parent = 0);
signals:
     // Aquí funciones para signals
public slots:
     // Aquí funciones para slots
};

#endif // MICLASE_H
```

Interacción signal/slot

Cualquier clase derivada (directa o indirectamente) de QObject puede manejar el mecanismo signal/slot para interactuar con las clases de usuario:

- Una señal (signal) es un evento que un objeto genera para ser atendido por otros.
- Un receptor (slot) es una función de que se ejecuta cuando sucede el evento.
- Es necesario conectar cada señal de interés con su receptor correspondiente.
- Tanto signal como slot pueden tener parámetros, pero devuelven void
- La clase QCoreApplication está ejecutando el bucle de espera por eventos, para sí misma o para cualquier otro QObject. Cuando se produce ese evento, llama al receptor que se ha conectado.

Ejemplo: QTimer → permite generar una señal llamada `timeout()` de forma periódica, útil para temporizaciones. Para usarlo, añadiremos en nuestro código:

- Un objeto de tipo QTimer: `QTimer temporizador;`
- Una función receptora en nuestra clase (tipo slot): `void OnTemporizador();`
- Una conexión entre la señal de temporizador y nuestra función:
  ```
  connect(&temporizador,SIGNAL(timeout()),this,SLOT(OnTemporizador()));
  ```
- Una indicación al objeto temporizador para que genere el evento:
  ```
  temporizador.start(1000); // Genera el evento timeout() cada 1000 ms
  ```
- A partir de ese momento, cada vez que se produzca el evento se ejecutará nuestra función.

Ejemplo temporizador en Qt5

Crear aplicación tipo Qt Console

main.cpp

```
#include <QCoreApplication>
#include <QTimer>
#include "MiClase.h"

int main(int argc, char *argv[])
{
    QTimer tempo;
    MiClase obj;

    QObject::connect(&tempo,SIGNAL(timeout()),
                     &obj,SLOT(OnTimer()));

    QCoreApplication a(argc, argv);

    tempo.start(2000);
    return a.exec();
}
```

MiClase.h

```
#ifndef MICLASE_H
...

class MiClase : public QObject
{
    Q_OBJECT
private:
    int counter;
...
public slots:
    void OnTimer();
};

#endif // MICLASE_H
```

MiClase.cpp

```
#include "MiClase.h"
#include <QDebug>
MiClase::MiClase(QObject *parent) : QObject(pa
{
    counter=0;
}

void MiClase::OnTimer()
{
    counter++;
    qDebug() << "TEMPORIZADOR " << counter;
}
```

Interacción signal/slot

Cualquier clase derivada (directa o indirectamente) de QObject puede generar señales mediante emit(), que serán servidas por los objetos conectados a esas señales

MiClase.h

```
#ifndef MICLASE_H
#define MICLASE_H

#include <QObject>

class MiClase : public QObject
{
    Q_OBJECT
private:
    int counter;
public:
    explicit MiClase(QObject *parent = 0);

signals:
    void Alarma(QString txt);
public slots:
    void OnTimer();
};

#endif // MICLASE_H
```

MiClase.cpp

```
#include "MiClase.h"
#include <QDebug>

MiClase::MiClase(QObject *parent) : QObject(parer
{
    counter=0;
}

void MiClase::OnTimer()
{
    counter++;
    qDebug() << "TEMPORIZADOR" << counter;
    if (counter % 5 == 0)
    {
        QString txt=QString("Ha saltado %1"
         " veces el temporizador").arg(counter);
        emit Alarma(txt);
    }
}
```

Interacción signal/slot

Cualquier clase derivada (directa o indirectamente) de QObject puede generar señales mediante emit(), que serán servidas por los objetos conectados a esas señales

Alarma.h

```
#ifndef ALARMA_H
#define ALARMA_H

#include <QObject>
#include <QString>

class Alarma : public QObject
{
    Q_OBJECT
public:
    explicit Alarma(QObject *parent = 0);

signals:

public slots:
    void OnAlarm(QString txt);
};

#endif // ALARMA_H
```

Alarma.cpp

```
#include "Alarma.h"
#include <QDebug>

Alarma::Alarma(QObject *parent) : QObject(parent)
{

}

void Alarma::OnAlarm(QString txt)
{
    qDebug() << "ALARM: " << txt;
}
```

Interacción signal/slot

Cualquier clase derivada (directa o indirectamente) de QObject puede generar señales mediante emit(), que serán servidas por los objetos conectados a esas señales

main.cpp

```
#include <QCoreApplication>
#include <QTimer>
#include "MiClase.h"
#include "Alarma.h"

int main(int argc, char *argv[])
{
    QTimer tempo;
    MiClase obj;
    Alarma alarm;

    QObject::connect(&tempo,SIGNAL(timeout()),&obj,SLOT(OnTimer()));
    QObject::connect(&obj,SIGNAL(Alarma(QString)),&alarm,SLOT(OnAlarm(QString)));

    QCoreApplication a(argc, argv);

    tempo.start(2000);
    return a.exec();
}
```

Ejercicio

Ejercicio propuesto:

Realizar un programa Qt Console para gestión de eventos de dos temporizadores. Si se reciben los eventos de ambos temporizadores con un tiempo menor a 100ms, se debe emitir una alarma (eventos sincronizados) que sea servida por la clase Alarma.

- Se necesitan 2 QTimer en main(), cada uno de ellos con diferente tiempo (ej. 2000 y 3000), y una sola variable MiClase.
- Cada QTimer se conecta a un slot diferente de la variable de tipo MiClase
- Cada slot de MiClase almacena en una variable de tipo QDateTime (miembros privados de MiClase) el instante en que se ha recibido el último evento. Si el nuevo evento tiene una diferencia de tiempo con el más reciente del otro slot menor a 100ms, se emite la alarma.
- Ver ayuda de QDateTime para: saber tiempo actual, calcular diferencia entre 2 tiempos.