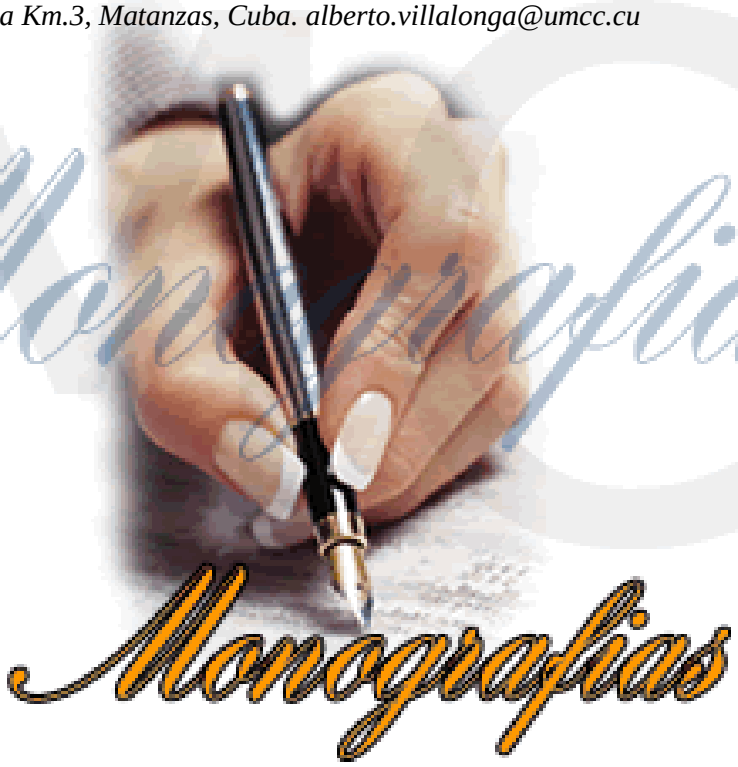
PROGRAMACIÓN MULTIHILOS EN C++ CON QT

Ing. Yarens Joaquín Cruz Hernández[1], MSc. Alberto Villalonga Jaén[2]

1. Universidad de Matanzas – Sede "Camilo Cienfuegos", Vía Blanca Km.3, Matanzas, Cuba. yarens.cruz@umcc.cu

1. Universidad de Matanzas – Sede "Camilo Cienfuegos", Vía Blanca Km.3, Matanzas, Cuba. alberto.villalonga@umcc.cu



CD de Monografías 2017

ISBN: XXX-XXX-XX-XXXX-X

Resumen

Con el presente trabajo se pretende iniciar al lector en la programación multihilos en C++ con *Qt*. Se realiza una diferenciación entre proceso e hilo y un breve análisis de la utilidad del paradigma multihilos, así su conveniencia en algunos casos. Se explica la utilización de varias clases que provee *Qt* para esta temática como son: *QThread, QMutex y QWaitCondition,* las cuales permiten la creación y sincronización de hilos. Además se explica en que consiste el patrón *Productor/Consumidor,* el cual es de gran utilidad cuando un hilo necesita datos generados por otro. Todos los aspectos mencionados son tratados mediante ejemplos sencillos, pero funcionales, que instruyen al lector en la gestión de hilos, por lo que este documento puede ser utilizado como guía práctica.

***Palabras claves**: Hilo de ejecución; Proceso; Sincronismo, Qt*

¿Qué es un proceso y qué es un hilo?

Cuando una aplicación se está ejecutando en una computadora normalmente en el procesador se lleva a cabo un proceso, el cual es una unidad de actividad que se caracteriza por la ejecución de la secuencia de instrucciones que constituyen el código de la aplicación, un estado actual y un grupo de recursos del sistema asociados (Stallings, 2012). En los sistemas operativos más modernos varias aplicaciones pueden estar ejecutándose a la vez, lo que significa que varios procesos se realizan en paralelo, pudiendo ser estos independientes o cooperativos. Los procesos independientes, como lo indica su nombre, son aquellos que no tienen relación entre sí, mientras que los cooperativos, se comunican mediante señales para sincronizarse. Sin importar el tipo de proceso que se esté llevando a cabo, los recursos del sistema asignados a cada proceso por el sistema operativo son independientes.

Por su parte, los hilos son las menores unidades de procesamiento que pueden ser planificadas por el sistema operativo y forman parte de un proceso. Cuando un proceso solo contiene un hilo se dice que el sistema es monohilo. Mientras que en los sistemas multihilos dentro de cada proceso pueden estar ejecutándose varios hilos a la vez. Los hilos asociados a un mismo proceso comparten recursos del sistema y memoria, por lo que necesitan estar sincronizados para que no interfieran unos con otros.

¿Por qué utilizar múltiples hilos?

Si se está ejecutando un proceso monohilo de una aplicación con interfaz visual y se lleva a cabo una operación de alto consumo de tiempo, la interfaz se queda congelada mientras se ejecuta la operación. Esto se debe a que el proceso tiene que terminar la operación antes de poder actualizar la interfaz. Con este sencillo ejemplo se puede ver la ventaja que representaría tener dos o más tareas ejecutándose a la vez en un mismo proceso, ya que

ISBN: XXX-XXX-XX-XXXX-X

mientras se estuviera llevando a cabo la operación de alto consumo de tiempo, la interfaz se mantendría activa.

La programación multihilos ha sido criticada porque puede provocar una disminución en el rendimiento en los sistemas con un solo procesador, debido a que este tiene que estar alternando constantemente entre los diferentes hilos de ejecución. Sin embargo, en los sistemas multiprocesador, los cuales cada día son más comunes, los hilos de ejecución pueden ser llevados a cabo en diferentes procesadores, lo que ayuda a dichos sistemas a trabajar más eficientemente por la forma en que fueron diseñados y logran un aumento considerable en el rendimiento (Blanchette and Summerfield, 2008).

Hilos y *Qt*

La distribución de los hilos y procesos entre los procesadores y el cambio entre hilos y procesos son llevados a cabo por el sistema operativo, haciendo que este tema sea muy dependiente de la plataforma en la que se esté trabajando (Thelin, 2007). Debido a que *Qt* enfatiza la programación independiente de plataformas, es una buena opción adentrarse en este tema utilizando *Qt*, ya que los programas que sean desarrollados en una plataforma no van a requerir ningún cambio para ejecutarse en otra, aunque sí deben ser implementados cuidadosamente. Por otra parte, *Qt* provee clases para trabajar con hilos y procesos, así como herramientas para hacerlos cooperar y compartir información, haciendo más simples estas tareas.

Creación y ejecución de hilos

Como ya se planteó anteriormente *Qt* contiene clases que facilitan el trabajo con hilos. Una de ellas es *QThread*, la cual será utilizada como clase base en el siguiente ejemplo para implementar una clase llamada *myThread* que nos permitirá crear nuevos hilos y ejecutar una tarea sencilla con estos. A continuación se muestra el código del archivo de cabecera de la clase *myThread*.

```
//mythread.h
#ifndef MYTHREAD_H
#define MYTHREAD_H
#include <QThread>
#include <QtDebug>

class myThread : public QThread
{
public:
    myThread(const QString &);
    void run();
    void stop();
private:
```


ISBN: XXX-XXX-XX-XXXX-X

```
    QString myText;
    bool stopThread;
};

#endif // MYTHREAD_H
```

Como se puede observar la clase cuenta con dos variables privadas, una llamada *myText* en la que se almacena un texto que se va a recibir mediante el constructor de la clase y otra de tipo booleano llamada *stopThread*, donde se almacenará el estado en el que se quiera poner al hilo, los cuales son: en ejecución o detenido. Además se definen los prototipos de las funciones *run()* y *stop()* que servirán para iniciar la ejecución del hilo y detenerla respectivamente. Seguidamente se muestra la implementación de la clase.

```
//mythread.cpp
#include "mythread.h"

myThread::myThread(const QString &text) : QThread()
{
    myText = text;
    stopThread = false;
}

void myThread::run()
{
    while(!stopThread)
    {
        qDebug() << myText;
        sleep(1);
    }
}

void myThread::stop()
{
    stopThread = true;
}
```

El constructor de la clase toma como parámetro un texto el cual es almacenado en la variable *myText* y se asigna el valor inicial false a la variable *stopThread*. La clase base *QThread* representa un hilo que lleva a cabo las tareas implementadas en un método llamado *run()*. Al redefinir este método en la clase *myThread* se logra que al iniciarse la ejecución del hilo la tarea que se lleve a cabo sea la que se encuentre implementada dentro del método *run()* redefinido. En el ejemplo, al ejecutarse el método *run()*, si el valor de *stopThread* es falso se entra en un ciclo del cual no se sale hasta que haya cambiado el valor de *stopThread* a verdadero. Dentro del ciclo se muestra el texto almacenado en la variable myText y se llama al método *sleep(int)*, el cual recibe como parámetro un entero que representa la cantidad mínima de segundos que el hilo va a esperar para continuar su ejecución. Por último, se define un método *stop()*, el cual va a permitir cambiar el estado de la variable *stopThread* a verdadero, lo que hará que la ejecución del hilo salga del ciclo del método *run()*. A

ISBN: XXX-XXX-XX-XXXX-X

continuación se muestra un código que sirve para comprobar el funcionamiento de la clase *myThread*.

```
//main.cpp
#include <QCoreApplication>
#include <mythread.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    myThread thread1("textoUno"), thread2("textoDos");

    thread1.start();
    thread2.start();

    cout << "HILOS EJECUTANDOSE. PRESIONE CUALQUIER TECLA PARA
DETENERLOS!!!" << endl;
    system("pause");

    thread1.stop();
    thread2.stop();

    thread1.wait();
    thread2.wait();

    return a.exec();
}
```

Primeramente se crean dos hilos llamados *thread1* y *thread2*, a los cuales se les asignan los textos "*textoUno*" y "*textoDos*" respectivamente. A continuación se ejecuta el método *start()* de cada hilo el cual da inicio a su ejecución, en este caso se llevará a cabo el código implementado en el método *run()*. Una vez que se haya iniciado la ejecución de los hilos se muestra un mensaje en el que se informa que para detener los hilos se debe presionar cualquier tecla e inmediatamente se pone al método *main()* en pausa. Al salir de la pausa se invoca al método *stop()* de cada hilo el cual modifica el valor de la variable *stopThread*, con el objetivo de finalizar la ejecución de los hilos. Por último se invoca al método *wait()* de cada hilo, esto es necesario ya que dicho método espera a que el hilo termine su ejecución para retornar el flujo de programa al método *main()*. Es un error asumir que al modificar el valor de la variable *stopThread* inmediatamente se va a detener el hilo, ya que este puede estar ejecutando el método *sleep(int)*, por lo que hay que esperar mediante *wait()*, si no existe la posibilidad de que el programa termine su ejecución antes que los hilos. A continuación se muestra la salida producida por el programa.

ISBN: XXX-XXX-XX-XXXX-X

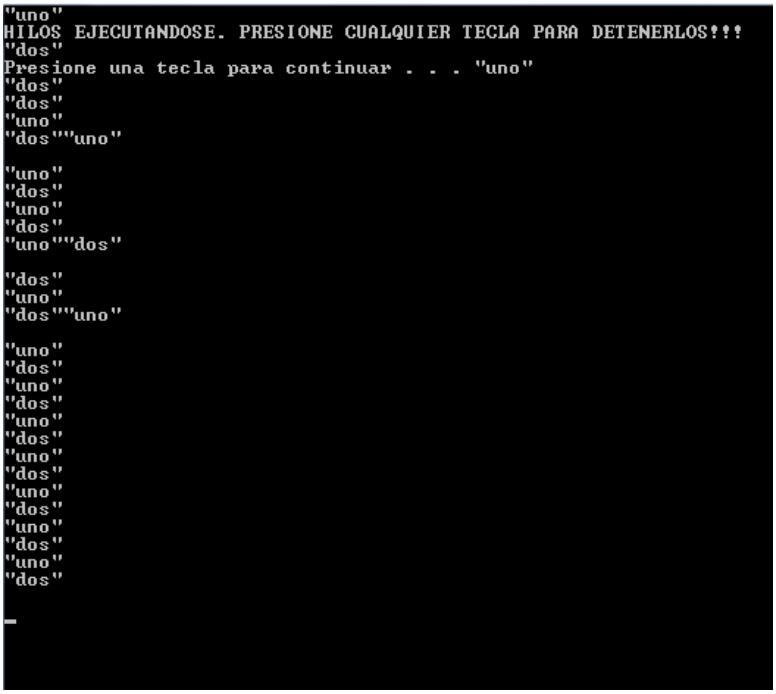


Figura 1. Salida del programa

Como se puede observar una vez que el método *main()* entra en la pausa se siguen mostrando los textos de los hilos, precisamente este era el resultado esperado, ya que la ejecución de los hilos es independiente. Se sabe que el sistema llegó a la pausa cuando se muestra el mensaje "*Presione una tecla para continuar . . .*". También es notable que los textos pueden alternar su orden de aparición o repetirse, esto es debido a la duración del método *sleep(int)* que no tiene un tiempo exacto, sino que tiene como valor mínimo el argumento pasado. Al presionarse una tecla se sale de la pausa en el método *main()* y se finaliza la ejecución de la manera descrita previamente.

Sincronización de hilos

En ocasiones es necesario que uno o más hilos actúen en consecuencia de lo que ocurra en otro hilo. Por ejemplo, un hilo puede necesitar para su ejecución el resultado de otro hilo. En este caso hay que esperar a que el segundo hilo termine su ejecución para tener un dato válido para trabajar. Otro caso es cuando los hilos van a utilizar un recurso común, para poder utilizarlo hay que asegurar que ningún otro hilo está haciendo uso del recurso en ese momento. La técnica utilizada para poder realizar estas acciones se llama sincronización (Thelin, 2007).

Para llevar a cabo la sincronización se puede utilizar una estructura llamada *mutex*, la cual tiene la capacidad de ser bloqueada y desbloqueada. Cuando el *mutex* está bloqueado permite

ISBN: XXX-XXX-XX-XXXX-X

que solamente el hilo que lo bloqueó tenga acceso a la porción de código protegida por él (Molkentin, 2007). De igual manera, solamente el hilo que bloqueó al *mutex* puede desbloquearlo. Las operaciones de bloqueo y desbloqueo son atómicas, lo que quiere decir que aunque pueden constar de varios pasos son tratadas como operaciones indivisibles que no pueden ser interrumpidas durante su ejecución. Esto es muy importante ya para bloquear el *mutex* se deben llevar a cabo dos pasos, el primero es comprobar si está previamente bloqueado y el segundo, si no está bloqueado, bloquearlo. Si las operaciones no fueran atómicas, dos hilos que intentaran bloquear el *mutex* a la vez podrían comprobar el estado al unísono y obtener que está desbloqueado e intentar el bloqueo. Esto provocaría que solo un hilo boqueara el *mutex* pero que los dos actuaran como si hubieran sido el causante del bloqueo.

Qt provee una clase para trabajar con el *mutex* llamada *QMutex*. El método para bloquear se llama *lock()* y el método para desbloquear se llama *unlock()*. En el ejemplo anterior se vio como el mensaje de un hilo se podía repetir antes de que se mostrara el mensaje del otro hilo debido a la duración del método *sleep(int)*. A continuación se va a modificar dicho ejemplo para ordenar la aparición de estos textos alternadamente con el uso de un *mutex*.

```
//mythread.h
#ifndef MYTHREAD_H
#define MYTHREAD_H
#include <QThread>
#include <QtDebug>
#include <QMutex>

class myThread : public QThread
{
public:
    myThread(const QString &);
    void run();
    void stop();
private:
    QString myText;
    bool stopThread;
    QMutex mutex;
};

#endif // MYTHREAD_H
```

En el fichero *mythread.h* el único cambio que se realizó fue la declaración de la variable *mutex* de tipo *QMutex*. A continuación se muestra el fichero *mythread.cpp* modificado.

```
//mythread.cpp
#include "mythread.h"

myThread::myThread(const QString &text) : QThread()
{
    myText = text;
```

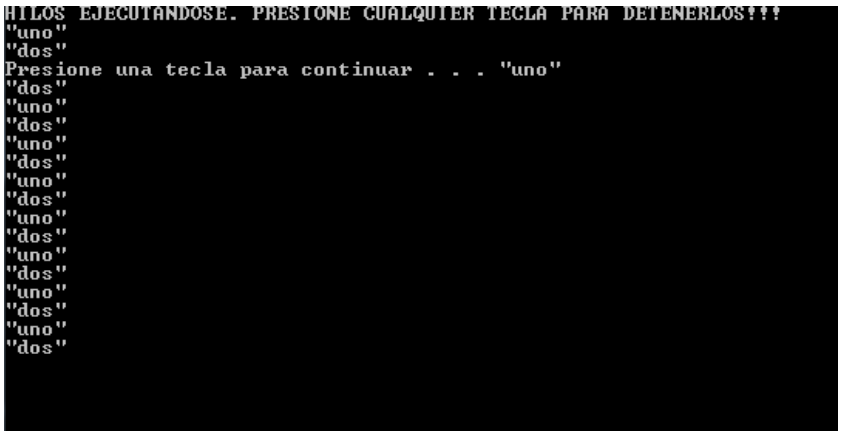
ISBN: XXX-XXX-XX-XXXX-X

```
    stopThread = false;
}

void myThread::run()
{
    while(!stopThread)
    {
        mutex.lock();
        if(stopThread)
        {
            mutex.unlock();
            return;
        }
        qDebug() << myText;
        sleep(1);
        mutex.unlock();
    }
}

void myThread::stop()
{
    stopThread = true;
}
```

La mayor parte del código se mantiene igual, el único método con cambios es *run()*. Antes de mostrar el texto se llama al método *lock()* para que solo el hilo que realice el bloqueo en ese momento muestre su texto y haga la espera de al menos 1 segundo, luego se llama al método *unlock()* para que los otros hilos puedan ejecutar esa parte del código, es decir, que si un hilo diferente llega a la llamada del método *lock()* cuando el *mutex* está bloqueado, tiene que espera a que se desbloquee para continuar su ejecución. De esta manera aseguramos que se muestren los textos alternadamente. En este caso es necesario incluir la sentencia *if*, ya que la función *main()* puede llamar al método *stop()* mientras se estuviera esperando para realizar el bloqueo y el texto se mostraría una vez más antes de detener la ejecución del hilo. A continuación se muestra la salida del programa.



ISBN: XXX-XXX-XX-XXXX-X

Figura 2. Salida del código

Como se puede ver, los textos son mostrados alternadamente como se esperaba. Aunque en este caso se mostró primero el texto "*uno*", no podemos asegurar que esto siempre va a suceder así, ya que el segundo hilo pudiera inicializarse primero incluso cuando el método *start()* del primer hilo sea invocado antes. Para ver una forma de solucionar este problema referirse a (Thelin, 2007).

El patrón Productor/Consumidor

Como ya se había planteado antes, hay ocasiones en las que un hilo necesita datos proporcionados por otro hilo, lo que requiere de un sincronismo entre los dos hilos. La forma de lograr este sincronismo es mediante el patrón productor/consumidor. Dicho patrón se basa en que un hilo productor genera la información y la va almacenando en una cola, de la cual el hilo consumidor puede extraerla (Molkentin, 2007).

Al ser un recurso compartido, la cola debe estar protegida por un *mutex*. Puede darse el caso en el que el productor genere datos más rápido de lo que el consumidor los extrae de la cola, con lo que la cola se llenaría. Cuando esto ocurra, el productor debe entrar en un estado de suspensión o espera hasta que haya espacio en la cola. También pudiera ocurrir que el consumidor extrajera los datos de la cola más rápido de lo que el productor fuera capaz de generarlos, lo que haría que la cola se quedara vacía. En este caso es el consumidor el que debe entrar en un estado de espera hasta que haya datos disponibles nuevamente.

A continuación se desarrolla un ejemplo en el que se implementa el patrón descrito anteriormente. La clase *QQueue* de *Qt* implementa la cola y entre otros métodos cuenta con el método *enqueue()*, el cual coloca la información que recibe como parámetro al final de la cola y el método *dequeue()*, el cual extrae el dato más antiguo que se encuentre en la cola. Primero se muestra el archivo cabecera *producer.h* de la clase que representa al consumidor.

```
//producer.h
#ifndef PRODUCER_H
#define PRODUCER_H
#include <QThread>
#include <QtDebug>
#include <QQueue>
#include <QMutex>
#include <QWaitCondition>

class Producer : public QThread
{
public:
    Producer(QQueue<int> *, int, QMutex *, QWaitCondition *,
QWaitCondition *);
protected:
    void produceMessage();
```


ISBN: XXX-XXX-XX-XXXX-X

```
    void run();
private:
    QQueue<int> *queue;
    int length;
    QMutex *mutex;
    QWaitCondition *queueNotFull;
    QWaitCondition *queueNotEmpty;
};

#endif // PRODUCER_H
```

La clase además del constructor cuenta con el método *run()*, siendo este un aspecto que ya se trató anteriormente. También cuenta con el método *produceMessage()* con el cual se produce un nuevo dato y se inserta en la cola, en caso de estar llena se muestra un mensaje y se espera a que tenga espacio disponible. Por otra parte, se definen las variables necesarias para poder desarrollar el ejemplo. Estas son punteros ya que las variables realmente se encuentran en el método *main()*. Las variables del tipo *QWaitCondition* sirven como condicionales para la sincronización de hilos. A continuación se muestra el fichero *producer.cpp*.

```
//producer.cpp
#include "producer.h"

Producer::Producer(QQueue<int> *q, int l, QMutex *m,
QWaitCondition *qnf, QWaitCondition *qne)
{
    queue = q;
    length = l;
    mutex = m;
    queueNotFull = qnf;
    queueNotEmpty = qne;
}

void Producer::produceMessage()
{
    qDebug() << "Produciendo ...";
    mutex->lock();
    if(queue->size() == length)
    {
        qDebug() << "La cola esta llena, en espera de espacio
disponible ...";
        queueNotFull->wait(mutex);
    }
    queue->enqueue((rand() % 100) + 1);
    queueNotEmpty->wakeAll();
    mutex->unlock();
}

void Producer::run()
{
```

ISBN: XXX-XXX-XX-XXXX-X

```
    forever
    {
        produceMessage();
        msleep((rand() % 3000) + 1);
    }
}
```

En el constructor se asignan los datos a las variables. El método *produceMessage()* primeramente muestra el mensaje "*Produciendo ...*", seguidamente bloquea el *mutex* que protege las acciones sobre la cola. Si la cola está llena, se muestra un mensaje que lo indica y la condicional que espera por el espacio en la cola se activa. El método *wait()* de la clase *QWaitCondition* desbloquea el *mutex* momentáneamente para que pueda ser utilizado por otros hilos hasta que se cumplan las condiciones para salir de la espera, momento en el cual se vuelve a bloquear el *mutex* para continuar la ejecución. Si no estuviera llena la cola o cuando se salga de la condicional de espera, se genera un nuevo dato, el cual es insertado en la cola. Se desactiva la posible condicional que estuviera en espera por ausencia de datos en la cola y se desbloquea el *mutex*. Por otra parte, el método *run()* se encarga de siempre invocar al método *produceMessage()* cada un intervalo de tiempo aleatorio. A continuación se muestra el archivo de cabecera de la clase *Consumer*.

```
//consumer.h
#ifndef CONSUMER_H
#define CONSUMER_H
#include <QThread>
#include <QtDebug>
#include <QQueue>
#include <QMutex>
#include <QWaitCondition>

class Consumer : public QThread
{
public:
    Consumer(QQueue<int> *, int, QMutex *, QWaitCondition *,
QWaitCondition *);
protected:
    int consumeMessage();
    void run();
private:
    QQueue<int> *queue;
    int length;
    QMutex *mutex;
    QWaitCondition *queueNotFull;
    QWaitCondition *queueNotEmpty;
};

#endif // CONSUMER_H
```


ISBN: XXX-XXX-XX-XXXX-X

Este archivo de cabecera es bastante similar al de la clase *Producer* solo que contiene un método *consumeMessage()* en vez de *produceMessage()*. Seguidamente se muestra el fichero *consumer.cpp*.

```
//consumer.cpp
#include "consumer.h"

Consumer::Consumer(QQueue<int> *q, int l, QMutex *m,
QWaitCondition *qnf, QWaitCondition *qne)
{
    queue = q;
    length = l;
    mutex = m;
    queueNotFull = qnf;
    queueNotEmpty = qne;
}

int Consumer::consumeMessage()
{
    qDebug() << "Consumiendo ...";
    mutex->lock();
    if(queue->isEmpty())
    {
        qDebug() << "La cola esta vacia, en espera de datos
disponibles ...";
        queueNotEmpty->wait(mutex);
    }
    int val = queue->dequeue();
    queueNotFull->wakeAll();
    mutex->unlock();
    return val;
}

void Consumer::run()
{
    forever
    {
        qDebug() << consumeMessage();
        msleep((rand() % 4000) + 1);
    }
}
```

El método *consumeMessage()* comienza mostrando un mensaje y bloqueando el *mutex*. Si la cola está vacía muestra un mensaje indicándolo y activa la condicional que espera a que hayan datos en la cola. Si no estuviera vacía la cola o cuando se salga de la condicional de espera se extrae el primer valor de la cola. Se desactiva la posible condicional que estuviera en espera por espacio en la cola, se desbloquea el *mutex* y se devuelve el dato extraído de la cola. El método *run()* se encarga de siempre mostrar el dato obtenido mediante *consumeMessage()* cada un intervalo de tiempo aleatorio. A continuación se muestra el fichero *main.cpp* en el cual se hace uso de las clases implementadas anteriormente.

ISBN: XXX-XXX-XX-XXXX-X

```
//main.cpp
#include <QCoreApplication>
#include <consumer.h>
#include <producer.h>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QQueue<int> myQueue;
    int myLength = 5;
    QMutex myMutex;
    QWaitCondition myQueueNotFull;
    QWaitCondition myQueueNotEmpty;

    Producer producer(&myQueue, myLength, &myMutex,
&myQueueNotFull, &myQueueNotEmpty);
    Consumer consumer(&myQueue, myLength, &myMutex,
&myQueueNotFull, &myQueueNotEmpty);
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();

    return a.exec();
}
```

En este fichero primero se definen las variables necesarias para el ejemplo. La longitud de la cola fue definida a 5 valores; en una aplicación real serían muchos más. A continuación se inicia la ejecución del productor y del consumidor, los cuales se ejecutarán hasta que se cierre la aplicación. Seguidamente se muestra una salida producida por el código anterior.

ISBN: XXX-XXX-XX-XXXX-X

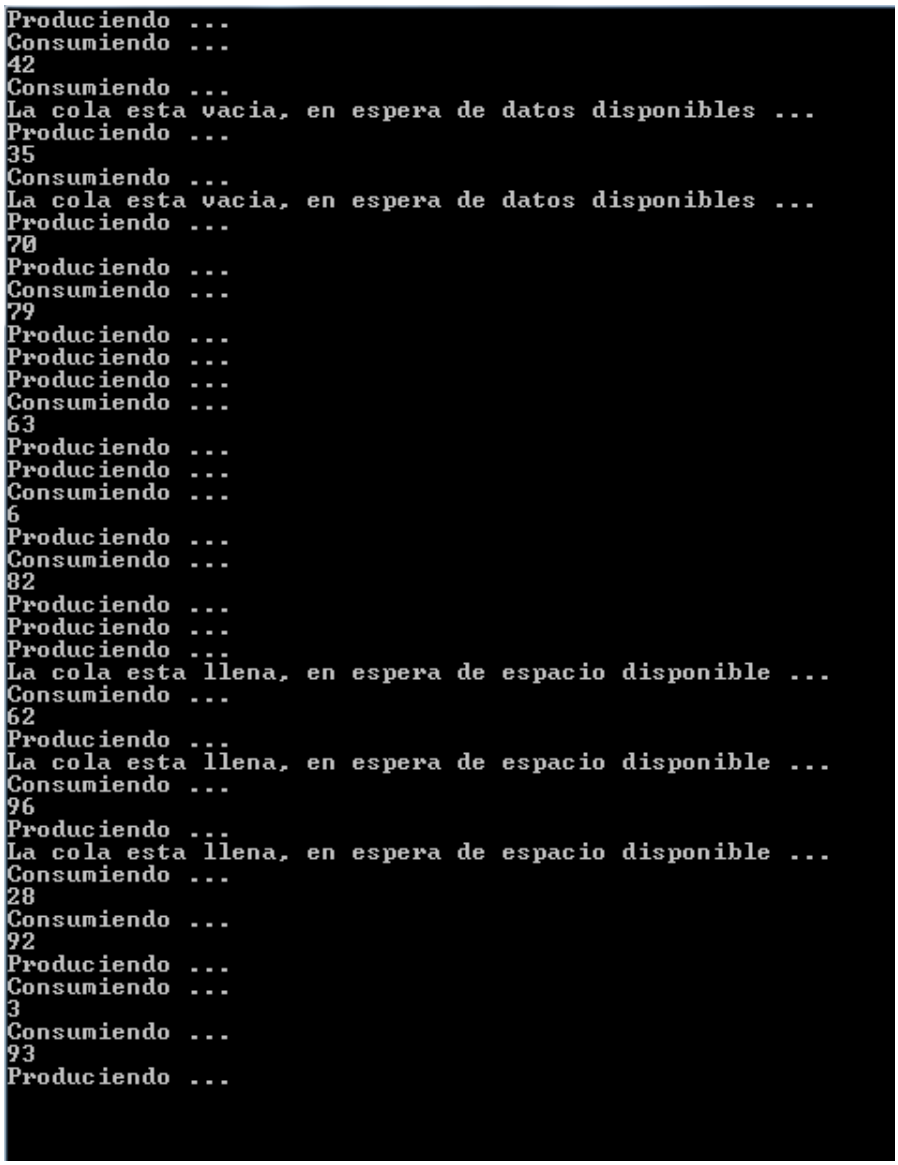


Figura 3. Salida del programa productor/consumidor

Los resultados son los esperados. Cuando se consume más rápido de lo que se produce la cola se queda vacía y hay que esperar a que se generen más datos, por el contrario cuando se produce más rápido de lo que se consume la cola se llena y hay que esperar a que haya espacio disponible.

ISBN: XXX-XXX-XX-XXXX-X

Bibliografía

BLANCHETTE, J. and SUMMERFIELD, M. *C++ GUI Programming with Qt 4* (Second Edition), Prentice Hall. 2008.

MOLKENTIN, D. *The Book of Qt 4,* Open Source Press, 2007.

STALLINGS, W. *Sistemas Operativos* (5ta Edición), Prentice Hall. 2012.

THELIN, J. *Foundations of Qt Development, Apress, 2007.*

ISBN: XXX-XXX-XX-XXXX-X