DOCENCIA EN INFORMÁTICA INDUSTRIAL: LENGUAJES DE PROGRAMACIÓN

Rogelio Mazaeda Echevarría, Eusebio de la Fuente López, José L. González, Eduardo J. Moya de la Torre
Depto. ISA, Escuela de Ingenieros Industriales, UVA,
rogelio@cta.uva.es; {efuente, jossan, edumoy}@eii.uva.es

Resumen

En el presente artículo se discute la conveniencia de aplicar en la enseñanza de la asignatura de Informática Industrial del Grado en Electrónica Industrial y Automática de la Universidad de Valladolid, paradigmas de programación más abstractos a los que hasta ahora se venían utilizando. En este contexto se discuten las virtudes de la nueva versiones del estándar de C++v11 en lo que respecta a la programación concurrente, cuyo conocimiento es considerado un objetivo básico de la mencionada asignatura.

Palabras Clave: educación en control, sistemas controlados por computador, redes de computadores, arquitecturas concurrentes, sistemas distribuidos de control por computadores, sistemas de tiempo real.

1 INTRODUCCIÓN

Los computadores digitales, los programas informáticos y las redes constituyen los elementos técnicos fundamentales de la industria moderna. Esto es así, a todos los niveles de la fábrica: desde el más alto, perteneciente a la capa de negocios hasta el más bajo relacionado con los sensores y actuadores desplegados en el campo, pasando por supuesto por su utilización en los controladores industriales. Esto es cierto, además, para todo tipo de industrias: desde las dedicadas a la producción de elementos discretos y que utilizan fundamentalmente controladores y modelos del proceso a controlar de tipo secuencial y basado en eventos, hasta aquellas industrias, como las de proceso y petro-químicas, que deben lidiar, principalmente, con el control de variables continuas, y que apelan a conceptos derivados de la teoría de sistemas dinámicos y de la ingeniería de control [1].

En cualquier caso, las fronteras entre control discreto y continuo no son del todo nítidas: existen en las industrias continuas la necesidad de implementar procesos por lotes, o la necesidad de proveer mecanismos de seguridad, que requieren de la ejecución de secuencias lógicas y basadas en eventos, y por otra parte, en las industrias de fabricación discreta, también surge la necesidad de controlar variables continuas.

En todo caso, la informática industrial tal como aquí utilizamos el concepto, debe tratar con el mundo físico real y debe reaccionar al estado actual de este último, obtenido a través de diferentes tipos de sensores, de la manera adecuada y en el plazo de tiempo requerido utilizando los actuadores a su disposición. De manera que los temas como la programación concurrente y en tiempo real resultan elementos fundamentales de la disciplina. Pero también lo son aquellos otros que permiten aplicar la teoría de sistemas dinámicos a los sistemas digitales: (sistemas muestreados), o que permite modelar formalmente el funcionamiento basado en eventos tanto del sistema físico a controlar como del programa informático que implementa el controlador.

Por otra parte el uso de sistemas digitales distribuidos a través de redes informáticas es una realidad insoslayable y esto determina las existencias de complicaciones teóricas y prácticas importantes que deben ser tomadas en cuenta.

Pero los conceptos de sistemas informáticos reactivos antes descritos, no solo se utilizan en el contexto de la fábrica contemporánea. Existe otra dimensión, que no puede ser desdeñada, y es la del uso de los sistemas de control como elementos empotrados en el propio producto. Es más, la tendencia en este sentido es imparable: los productos "inteligentes" ya no están circunscritos a bienes de precios relativamente elevado como electrodomésticos o automóviles, sino que el abaratamiento del hardware y su miniaturización, unido a las capacidades de conexión utilizando redes inalámbricas, permiten la presencia ubicua de la necesaria capacidad de cómputo y de comunicación en los productos más simples y baratos. Este hecho, además, tiene repercusiones significativas en el diseño y funcionamiento previsible de la fábrica futura. Se considera que se está incubando lo que se cree sea la próxima revolución industrial, a la sombra de la cual, términos como internet de las cosas (IoT), sistemas cyber-físicos, fábricas inteligentes o industria 4.0, sirven para describir un entorno industrial mucho más dinámico y complejo, altamente distribuido y re-configurable, con profusión de datos redundantes y que deberá ser enfrentado, de una manera holística, analizando el sistema a controlar, el controlador

digital con su algoritmo de control y todo el resto de software interviniente (sistemas operativos de tiempo real, protocolos de red, etc) como un solo sistema híbrido de gran complejidad y extendido geográficamente [11,12].

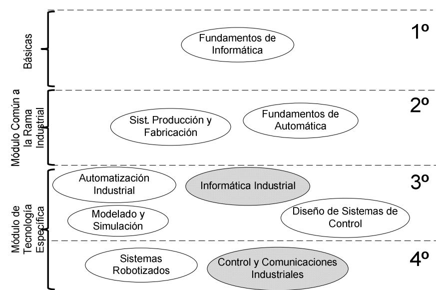

Fig. 1. La Informática Industrial en el marco de GEIA.

La realización de este paradigma de la industria 4.0 constituirá un esfuerzo muy grande, y de carácter multidisciplinar. Los autores, sin embargo, tienen la convicción de que la disciplina de la informática industrial tendrá un papel central en ese escenario. Por otra parte, con casi total seguridad, muchos de los profesionales de la automatización que tendrán que enfrentar retos de esta envergadura, ya están o estarán en poco tiempo en nuestras aulas. Esta certeza, le da un sentido de urgencia a la necesidad de plantearnos continuamente la mejor manera de llevar los contenidos de esta disciplina a los alumnos.
Los contenidos de relevancia que pueden concebirse bajo el rótulo de Informática Industrial son prácticamente inabarcables y cambian muy rápidamente. La elección de que incluir y que dejar fuera, no resulta, por tanto, trivial y debe ser continuamente revisada.
El presente artículo se organiza del siguiente modo. En la sección 2 se plantea el contexto en que los contenidos de la Informática Industrial se imparten en el Grado de Electrónica Industrial y Automática (GEIA) en la Universidad de Valladolid (UVA), para posteriormente, en la sección 3 discutir la estrategia que los autores consideran más apropiada y que pasa por elevar el nivel de abstracción con el que se imparten los contenidos, en la sección 4 se defiende el uso de las nuevas versiones del C++ como lenguaje de elección para llevar a cabo la tarea antes descrita, decisión ésta que viene además reforzada por la reciente incorporación a dicho lenguaje de soporte directo para la programación concurrente. Finalmente en la sección 5 se ofrecen algunas ideas sobre la práctica docente en el marco de la asignatura Informática Industrial que se imparten en el Grado de Electrónica Industrial y Automática (GEIA) en la UVA, para luego ofrecer algunas consideraciones a modo de conclusión.

2 LA INFORMATICA INDUSTRIAL EN EL GRADO DE ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA DE LA UVA

La elección de los contenidos a impartir relacionados con la informática industrial debe plantearse el encontrar el balance adecuado entre los elementos teóricos de valor más permanente y los avances recientes. En un artículo anterior [7] se identificaron los temas fundamentales a tratar según se describe en la tabla 1. En ese artículo se describió como existen básicamente dos asignaturas en el Grado de electrónica Industrial y Automática (GEIA) de la UVA, que son las de Informática Industrial (II) de tercer año y Control y Comunicaciones Industriales (CCI) de cuarto curso, que están directamente vinculadas con los temas de la informática en la industria. A partir de considerar los descriptores de las mencionadas asignaturas y teniendo en cuenta el contexto de la titulación, representado simplificadamente en la figura 1, se determinó el reparto de temas a tratar entre II Y CCI que se muestra en la tabla 2.

Tabla 1. Temas fundamentales

T	Descripción
1	Contexto Industrial: El alcance de la II. La pirámide ISA-95. Diferentes tipos de plantas industriales. Problemas discretos, continuos e híbridos. El controlador como sistema reactivo.
2	Modelado Formal de Sistemas Discretos: redes de Petri, máquinas de estado finitas, *grafcets*...
3	Sistemas muestreados: teorema de Shannon, ecuaciones en diferencia, transformada z, filtros anti-alias, jitter.
4	Concurrencia: Arquitectura de ordenador y soporte del S.O y del lenguaje en relación a la concurrencia. Procesos e hilos. Modelos de concurrencia. Sincronización y comunicación entre tareas.
5	Sistemas de Tiempo Real: Discusión de alternativas y algoritmos de planificación de tareas. Determinismo.
6	Tolerancia a fallos y Seguridad
7	Comunicaciones Industriales y Sistemas Distribuidos
8	Sistemas de Control y Supervisión Industriales: Control digital directo, autómatas programables, sistemas DCS y SCADA. Ejemplos de aplicación.

Tabla 2. Reparto de temas por asignatura.

A	Temas enfatizados
II	Temas introductorios (1). El tema 4, de la programación concurrente, será el tema central de esta asignatura. Elementos del tema 6 (tolerancia a fallos), y 2 (Modelado Formal) serán tratados. El tema 8 en relación al control de sistemas de

	fabricación y ejemplos simples del tipo del como control todo-nada de sistemas continuos.
CCI	Temas centrales: el 5 y el 7. Se profundizará en el Tema 3 y en el Tema 8 con controladores para procesos continuos. Las prácticas de laboratorio incluirán además el control de maquetas físicas.

En el presente artículo, se continúa la reflexión de comenzada en la mencionada contribución, ahora concentrándonos en la mejor manera de impartir los contenidos concretos de la asignatura de II.

3 ESTRATEGIA DOCENTE ANTE LA ASIGNATURA DE INFORMÁTICA INDUSTRIAL.

La idea central que da coherencia al esfuerzo realizado es la de encontrar el grado de abstracción adecuado que resuelva el compromiso entre el nivel de detalle necesario y la posibilidad de tener los conocimientos y recursos imprescindibles para abordar temas de mayor envergadura y más relacionados con el control industrial. La figura 2 trata de ilustrar el sentido del balance buscado. ¿Se debe comenzar utilizando lenguajes procedurales como el C y librerías cómo POSIX para acceder a los recursos de concurrencia y tiempo real del sistema operativo? Es una opción razonable, y es la que históricamente se ha seguido y que resulta compatible además con el hecho de que los alumnos de ingeniería han sido expuestos al lenguaje C, en primer año del grado, en la asignatura básica de **Fundamentos de Informática**. Está opción es quizá la más flexible y general y la más estrechamente ligada al *hardware*. Es la opción tradicional para la programación, a relativamente alto nivel, de los sistemas empotrados. La desventaja, desde luego, reside en ese propio nivel de generalidad. La curva de aprendizaje que va desde la creación de las estructuras de datos necesarias, el dominio de las primitivas de POSIX para la creación de procesos o hilos de ejecución concurrente, y su soporte de tiempo real, hasta llegar a aplicaciones que tengan un contenido realista desde el punto de vista del control y la automatización, es muy empinada. Resulta muy difícil, en el tiempo limitado de que se dispone durante el curso, el pasar de utilizar ejemplos generales, y muy sencillos. En el otro extremo del espectro, estarían los sistemas configurables te tipo de sistemas de control distribuidos (DCS) o de supervisión (SCADAS) que son específicos de la automatización. Si se adoptara este nivel de abstracción, se obviarían todos los detalles concernientes a la programación aunque se brinde algún nivel de configuración más o menos complejo. Esta alternativa también constituye una posibilidad viable, que pone al graduado en disposición de ser insertado en el mercado laboral inmediatamente y quizá sea la ideal para muchas titulaciones. Los autores de este comunicado son de la opinión, sin embargo, que los graduados de GEIA deben retener la capacidad de poder programar y no de ser meros configuradores o integradores de sistemas ya desarrollados. En caso de optar por la última opción, el alumno de automática quedaría al margen, como observador más o menos informado, del importante proceso de renovación de la industria antes descrito en la introducción.

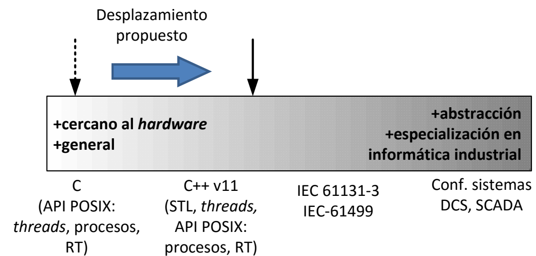

Figura 2. Nuevo nivel de abstracción propuesto.

La solución que se viene adoptando en los dos últimos años en la asignatura II, intenta colocarse a medio camino de ambos extremos. Este posicionamiento de compromiso consideramos que pasa por la adopción del lenguaje orientado a objetos C++. Esto nos permite trabajar a un nivel de abstracción mayor [8], sin perder las ventajas de poder, a conveniencia, bajar a nivel del *hardware* [2] para mayor eficiencia o para implementar las soluciones empotradas. La librería estándar de C++ provee directamente toda una colección y jerarquía de clases, como por ejemplo las clases genéricas de tipo contenedor como vectores, listas, mapas, entre otras, que liberan al programador de la necesidad de crear una infraestructura que es siempre necesaria en cualquier aplicación de cierta envergadura. Trabajar a este nivel permitiría al alumno concentrarse en el objetivo final de la aplicación. De esta forma, quedaría suficiente tiempo disponible, como para superar la etapa de realizar pequeños ejemplos de poca entidad, y pasar directamente a aplicaciones de una mayor envergadura y más ligadas al objetivo de crear sistemas reactivos capaces de lidiar como el mundo real industrial. Esta decisión resulta reforzada a partir del hecho de que los estándares recientes del lenguaje C++ soportan el uso nativo de la concurrencia.

4 LA CONCURRENCIA EN C++

La programación de ordenadores es una tarea especialmente exigente. Se requiere una gran atención a los detalles: el dominio, sin ambigüedades, de la sintaxis y la semántica de un lenguaje formal para dar las instrucciones precisas al ordenador pero también se requiere de la adquisición de otras competencias relacionadas con la incorporación de

un paradigma de programación apropiado, la adopción de buenas prácticas y en general, el aprendizaje de patrones y formas preferidas de abordar los problemas reales. Por otra parte, esta complejidad inherente a la programación en general, se multiplica cuando se trata de implementar aplicaciones informáticas concurrentes. Ahora la aplicación se concibe como compuesta por varias tareas, que pueden ser ejecutadas, la mayor parte del tiempo, en hilos de ejecución separados, sin especificar ninguna relación temporal especial entre ellas, excepto en los casos en que se requiera el intercambio de datos entre las mismas o su sincronización por cualquier otro motivo.

A pesar de las dificultades de la programación concurrente, ahora parece claro que es el camino a seguir. Se conoce que existen dos motivaciones básicas para intentar una solución concurrente: la primera es la de la separación de funciones y la segunda es el aprovechamiento del paralelismo físico real que actualmente existe en los ordenadores modernos. La separación de funciones ha sido quizá el factor que motivó el uso preferente de la concurrencia en los sistemas reactivos que deben lidiar, a través de sensores y actuadores, con el mundo real. Cada tarea informática se ocuparía de determinado proceso, bien delimitado, del mundo real y de alguna manera aproximaría mediante la *simultaneidad* simulada por el cambio de contexto entre los diferentes procesos o hilos de ejecución, la simultaneidad real de los sistemas físicos. En un ordenador con una sola CPU, el uso de tareas concurrentes implicaría siempre una degradación de la capacidad bruta de procesamiento, debido al tiempo, no productivo, invertido en el mencionado proceso de cambio de contexto. Sin embargo tendría como ventaja fundamental el hecho de que la *latencia*, o tiempo de reacción ante un estímulo externo, puede disminuir notablemente, lo que unido al hecho de que las tareas concurrentes pudieran ser planificadas, mediante un sistema operativo de tiempo real, para cumplir determinados plazos impuestos por las necesidades de la planta a controlar, hacen de la concurrencia la elección natural para las aplicaciones que nos ocupan. Ahora, sin embargo, abundan los ordenadores con varios núcleos o CPUs, de manera que la ejecución realmente simultánea de tareas paralelas es una posibilidad común, de manera que también en este segundo sentido, la elección del paradigma de la programación concurrente parece ser el camino apropiado.

Desde luego la adopción de la concurrencia está plagada de dificultades. Las tareas típicamente tienen que intercambiar datos o sincronizarse en función de determinado evento y en este caso, el orden relativo de ejecución de ciertas sentencias en las diferentes tareas, recobra su importancia. Los retos ahora, sin embargo, son significativos: la modificación de datos compartidos puede crear las conocidas condiciones de carrera (*race conditions*), en la que la consistencia de los mismos puede quedar comprometida. Existen mecanismos para lidiar con estos problemas como son los semáforos, **mutex**, secciones críticas, pero su utilización debe ser diseñada adecuadamente: un uso indiscriminado deteriora notablemente la velocidad de ejecución global o puede incluso impedir completamente el avance del programa, si se da la situación anómala que se conoce como *deadlock*.

La posibilidad de aprovechar del paralelismo real de los modernos CPU de múltiples núcleos es un avance importante, pero no resulta ajeno a los problemas antes mencionados. Es conocida, por ejemplo, la llamada ley de Amdahl [10], que nos explica, que aún en el caso ideal, el factor de mejora en la velocidad (P) que debe esperarse en un ordenador con N núcleos que comparten una fracción f_s de código que debe ser serializado se deteriora rápidamente según se muestra en (1). Además de la anterior dificultad, el uso de múltiple memorias cachés, para cada núcleo y la existencia de optimizaciones a nivel del compilador, complican la programación de manera notable.

$$P = \frac{1}{f_s + \frac{(1 - f_s)}{N}} \qquad (1)$$

El lenguaje C++ cuenta ya con una historia de casi 30 años. Surgió como una extensión del conocido lenguaje C, capaz de implementar los conceptos de la programación Orientada a Objetos (OO). En los últimos años el C++ ha experimentado un cambio muy importante y muestra una gran vitalidad. Existen dos recientes estándares ya aprobados, la llamada versión v11 y la v14 y se encuentra en discusión muy avanzada el nuevo estándar versión v17.

La versión v11 del estándar de C++, constituyó el cambio más notable y está ahora incorporado por defecto en la mayoría de los compiladores incluido el **gcc**, libremente disponible. El C++ v11 es un lenguaje multi-paradigma, que conserva la compatibilidad con un lenguaje procedural como el C, mantiene y profundiza sus características de OO e incluye nuevas características típicamente relacionadas con la programación funcional. Entre las características de programación que introduce o profundiza lo que ahora se denomina C++ moderno, se tienen:

- Una mejor librería estándar. Extendida y más robusta. Soporte, por ejemplo, para tabla *hash*.
- Bucles más sencillos basados en rango. Ideales para recorrer de forma muy intuitiva las clases contenedoras de la biblioteca STL.
- A las tradicionales formas típicas de pasar parámetros por valor, punteros (*) y referencias (&), se añade ahora el paso por

movimiento (&&) (*move semantics*) que permite, entre otras cosas, pasar a las funciones valores literales y objetos temporales de forma muy eficiente, cuando previamente antes solo podían pasarse por valor.

- Inicialización universal: brinda un formato común para inicializar variables de tipos de datos primitivos, objetos y colecciones de objetos.
- Soporte en la biblioteca estándar para diferentes tipos de punteros inteligentes (*smart pointers*).
- Funciones *lambdas*, concepto provenientes de la programación funcional y que entre otras muchas cosas, permite trabajar de forma compacta y eficiente con los algoritmos aplicados a clases contenedoras.
- Mejoras en la eficiencia del mecanismo de programación genérica mediante patrones (*templates*). Introducción de *templates* con número variable de argumentos (*variadic templates*).
- La programación genérica y la sobrecarga de operadores ha hecho uso siempre de la llamada inferencia de datos. Esta capacidad es ahora provechada utilizándola palabra reservada **auto** para hacer código más legible en aquellos contextos en que el tipo de datos al que se hace referencia puede ser deducido de forma no ambigua.

Pero la característica quizá más notable, al menos en lo que respecta a nuestro caso, es que ahora se soporta de forma nativa, la posibilidad de realizar programas concurrentes dentro del lenguaje de forma portable y sin recurrir a librerías externas, que como es el caso de POSIX, resultan dependientes del sistema operativo. Desde luego, el C++v11 no es el primer lenguaje que permite realizar esto: lenguajes anteriores como ADA o el propio JAVA tienen soporte para la concurrencia. El C++ sin embargo, ha tenido, desde siempre, una mayor popularidad entre los desarrolladores de sistemas, por ser un lenguaje más vinculado al *hardware* del ordenador, y que en general resulta en códigos con una menor huella en memoria y mayor velocidad de ejecución.

Para dar soporte a la concurrencia [10], el C++ v11 define un nuevo modelo de memoria con soporte para la multiprogramación, incluye nuevas primitivas que permiten el trabajo con diferentes hilos de ejecución o *threads,* que pueden compartir memoria, y que son ahora descritos a partir de una clase (**std::thread**) en la librería estándar. También, como es de rigor, se brindan recursos para el control de acceso a datos compartidos a través de objetos de tipo exclusión mutua (**std::mutex)** para evitar las condiciones de carrera. La sincronización entre *threads* se logra con mecanismos clásicos que van desde la unión de varios hilos, implementado como un método de la clase ***std::thread*** (**std::thread.join()**), al uso de variables de condición (**std::condition_variable**) que, utilizados en combinación con objetos tipo ***mutex*** permite a un determinado hilo de ejecución esperar de forma segura y sin consumir recursos del ordenador, por la notificación de eventos provenientes de otro. La espera de estos eventos, pueden estar limitadas en tiempo, o puede que el evento mismo sea un evento relacionado con el transcurso de determinado lapso, en ambos casos serían útiles las clases definidas en el fichero de cabecera **<chronos>**.

También se dispone del concepto de promesas (**std::promises<>**) y futuros (**std::futures<>**), ambas clases genéricas, que resultan útiles al implementar, de forma asíncrona la relación entre diferentes hilos de ejecución estableciendo, por ejemplo, relaciones de tipo productor/consumidor.

El estándar ofrece una gran cantidad de clases y funciones útiles, que resulta imposible comentar en este rápido recuento. Por ejemplo, se dispone de la función libre **std::lock()** que permite adquirir varios objetos de exclusión mutua de formar simultánea y segura para evitar el fenómeno del *deadlock*. También se podría conocer, si estuviera disponible, el grado de concurrencia real o de núcleos de CPU existentes, mediante la función **std::thread:: hardware_concurrency()**, información útil que permite adoptar, en tiempo de ejecución, la solución más eficaz.

```
#include <iostream>
#include <thread>
#include <vector>

void ImprimeVec(std::vector<int> && v)
{
    for (auto elem: v )
        std::cout << elem << std::endl;
}

int main()
{
    std::vector<int> vec {1,5,7,8,9,10,11,45,-12};

    std::thread t(ImprimeVec,std::move(vec));
    t.join();
    return 0;
}
```

Fig.3 Ejemplo de hilo en C++v11.

En la fig. 3 se ofrece un ejemplo muy simple que trata de la creación de un hilo desde la función principal, a que se le pasa como parámetro un vector, con el objetivo de que sea impreso por la pantalla. Obsérvese que el vector se pasa al hilo con el nuevo modo (***move***: movimiento), muy eficiente. Una vez creado el hilo, la función principal espera por la finalización del *thread* utilizando el método ***join*** del objeto que lo representa. Nótense detalles típicos del C++ moderno como el uso la **inicialización universal** al crear el vector y el del bucle ***for*** de rango a la hora de recórrelo.

Si se pretendiera trabajar la concurrencia a un nivel más ligado al *hardware*, implementado soluciones denominadas “libre de cerrojos” (*lock-free programming*) mucho más difíciles de diseñar pero

con el potencial de ser más eficientes, se ofrecen también una serie de nuevas operaciones atómicas.
El *coste de la abstracción* en términos de velocidad o de la huella en memoria siempre ha sido una preocupación sobre todo en el caso de la comunidad dedicada al desarrollo de sistemas empotrados. La buena noticia es que se han hecho mejoras en este sentido y existen reportes de que actualmente este coste es mínimo [5]. Los compiladores cruzados y en particular los enlazadores, disponibles para plataformas embebidas de diferentes capacidades, desde tarjetas potentes dotadas de sistema operativo hasta microcontroladores más simples, permiten controlar de forma precisa los compromisos entre velocidad y tamaño de memoria. Existen opciones de los enlazadores, por ejemplo, para desactivar elementos que se puedan considerar innecesarios en las plataformas más simples, como por ejemplo el código relacionados con información dinámica sobre los tipos de datos. También, como es obvio, se permite definir con precisión las zonas específicas de memoria donde los diferentes segmentos del programa (código, *stack*, variables globales, etc) van a ser cargados en la plataforma objetivo. Existe, además, por ejemplo la forma de informar a **new** la dirección de memoria donde se requiere la creación de memoria dinámica, para evitar la fragmentación del montículo (*heap*) hecho que, en plataformas, tan elementales, sería insostenible. Una preocupación que surge con fuerza en estas aplicaciones empotradas está relacionada con ser conscientes de las optimizaciones de código que realizan los compiladores modernos, que puede alterar de forma significativa el flujo que el programador prevé en una lectura lineal del código fuente. Por ejemplo, es habitual en este tipo de aplicaciones el utilizar interrupciones de hardware y las correspondientes subrutinas de atención a las interrupciones (ISR) como una forma legítima de interactuar con el mundo exterior. La ISR podría modificar determinada variable, para informar al bucle principal de control de, por ejemplo, la disponibilidad de un nuevo dato. Pero como el compilador no "conoce" este hecho, puede que el chequeo de dicha variables haya sido "optimizado" sacándolo fuera del bucle. Siendo conscientes de este problema, el programador deberá informar al compilador del hecho de que la variable puede cambiar sin previo aviso, definiéndola con el modificador *volatile*.
En cualquier caso, a pesar de los cuidados que una programación a tan bajo nivel merece, se puede asegurar que el C++ es, cada vez más una opción viable también para las soluciones empotradas.

5 ENFOQUE DOCENTE

El estudio de la programación concurrente en C++, ya sea de forma nativa al lenguaje, mediante hilos o través de procesos, en este último caso accediendo a las funcionalidades del sistema operativo mediante una librería basada en POSIX, constituirá el centro de la asignatura de II.
El primer problema que se ha de vencer es el de los pre-requisitos. En su recorrido académico el alumno ha sido expuesto solamente al curso introductorio de **Fundamentos de Informática** en el que se da exclusivamente el lenguaje C. Actualmente existe un cierto consenso entre los especialistas, relativo a que la forma tradicionalmente usada de enseñar primero C para luego introducir C++ como una extensión del primero, no es la variante idónea. En nuestro caso sin embargo, ese camino de la exposición inicial al C ya ha sido cubierto previamente, de manera que se pueden dar los conceptos de principales de C++ como una mejora sobre sus contrapartes en C. Así por ejemplo, se pueden introducir, tan pronto como sea posible, el uso de clases como *string* y colecciones como los vectores genéricos de manera muy rápida, sin necesidad de explicar previamente como diseñar una clase en C++. Este tipo de introducción "suave" a las ventajas del nuevo lenguaje es la que se propone para la asignatura de II y podría ser llevada a cabo perfectamente en las primeras sesiones de laboratorio frente al ordenador. Una lista de los conceptos fundamentales del C++ y el orden aproximado de su introducción sería la siguiente:

1. Nuevos tipos de datos como ***bool***, variables de tipo s***tring***, manejo de entrada y salida a través de la consola (clases ***cin***, ***cout***) y ficheros (clases ***istream***, ***ostream***). La clase vector. Breve relato teórico del concepto de clases y objetos como extensión de la estructura (**struct**) de C.
2. Funciones libres en C++. Sobrecarga de funciones. Nueva forma de **paso por referencia** y semántica del **paso por movimiento**.
3. Creación de clases sencillas. Sobrecarga de operadores. Concepto de herencia. Clases virtuales puras. El polimorfismo.
4. Los patrones o *templates*. Recorrido por otras clases contenedoras: listas, mapas, etc.
5. Tratamiento estructurado de excepciones.
6. Reserva dinámica de memoria: ***new*** and ***delete***.

Simultáneamente, a esta exposición del C++ frente al ordenador, en las clases de pizarra, se podría ir cubriendo un terreno importante que incluye: generalidades sobre la pirámide de control, sistemas ciberfísicos e industria 4.0, introducción a sistemas de tiempo real, y sistemas empotrados. Y ya con un carácter más detallado: programación concurrente mediante procesos e hilos, soporte a nivel de sistema operativo (POSIX) y a nivel del propio lenguaje (las descritas novedades del C++v11). Métodos de comunicación y sincronización: semáforos,

monitores, memoria compartida, mensajes, etc. Así como ejemplos de arquitecturas clásicas, preferiblemente vinculadas al contexto industrial. La parte teórica podría incluir temas relacionadas con la programación de sistemas industriales [6] basados en los estándares IEC-61131 e IEC 61499, sistemas SCADAS industriales y elementos de seguridad.
Una vez transcurrido algo menos que un tercio del curso, el alumno típicamente habrá adquirido las habilidades básicas de C++ necesarias y podrá comenzar a realizar proyectos, de una envergadura moderada y motivados en el control industrial, utilizando los recursos nativos del lenguaje C++ y los *threads* o los procesos con bibliotecas a nivel del sistema operativo del tipo de POSIX. El estudio de los procesos nunca puede ser abandonado. En ocasiones, el grado de seguridad extra que estos representan, con espacios de memoria separados que estimulan el uso de mecanismos de comunicación más seguros como el paso de mensajes, compensan con creces cualquier latencia añadida debido al cambio de contexto. Pero además, la concurrencia basada en procesos formarían la base de los sistemas distribuidos que podrían ser un tema de la siguiente asignatura (CCI), donde la ejecución de programas, ciertamente concurrentes, pero ejecutándose en ordenadores diferentes conectados a través de redes informáticas, impone no ya el uso de espacios de memoria separados sino, posiblemente, la utilización de ordenadores con arquitecturas y sistemas operativos diferentes, con la consiguiente necesidad de realizar la conversión del formato de los datos (*marshalling*).

5.1 USO DE BUENAS PRÁCTICAS DE PROGRAMACIÓN EN C++ MODERNO

En el listado de los conceptos fundamentales de C++ a explorar frente al ordenador, la colocación de la reserva dinámica en último lugar habrá quizá sorprendido a algún lector. Es importante recalcar que el uso indiscriminado de los punteros tradicionales y de los operadores ***new*** y ***delete*** está completamente desaconsejado en la programación de aplicaciones convencionales, aunque evidentemente son utilizados profusamente en la creación de las librerías. Es obvio que la capacidad de redimensionamiento de las colecciones de la STL incluyendo la clase ***string***, son posibles debido al uso, tras bambalinas, de la reserva dinámica de memoria. Es conocido, sin embargo, que el uso apropiado de estos conceptos, requiere de un cuidado extremo y puede dar lugar a fugas de memoria, que eventualmente provocarían el fallo del programa, cuestión esta que se agravaría en presencia de excepciones. La solución considerada como correcta es la que se resume en una idea muy fructífera, conocida por las siglas en inglés RAII (*resource adquisition is initialization:* la adquisición del recurso es la inicialización) y que es la base de los conocidos *smart-pointers*. La idea es conceptualmente simple: se trata de encapsular el proceso de obtención del recurso (la memoria desde el montículo en el caso de los punteros inteligentes) en el constructor de la clase creada para este fin y de liberar el recurso en el destructor. El recurso se adquiere cuando se *instancia* el objeto de la clase envoltorio de forma local y se destruye automáticamente cuando el objeto creado abandona el alcance de la función en la que fue declarado. Este esquema funciona correctamente aún en presencia de excepciones. El patrón RAII puede utilizarse en circunstancias aparentemente muy diferentes, que involucren la adquisición y devolución ordenada de algún recurso, por ejemplo: la apertura y posterior cierre de la conexión con una base datos, o la ejecución de un **thread** y posterior espera por su finalización con el método ***join***.
En el código mostrado en la fig. 4 se ejemplifica el uso de la misma idea a la hora de adquirir y liberar un ***mutex*** para la protección de determinado objeto compartido. Aquí el objeto ***rec*** de la clase ***Recurso***, es compartido entre dos hilos de ejecución, el representado por la función ***main*** y el correspondiente al hilo ***t.*** La clase tiene como datos privados un vector y un objeto tipo ***mutex*** (***m***) destinado a proteger el acceso compartido al primero. La función ejecutada por el hilo ***t***, "despierta" cada 5 segundos, e intenta imprimir el vector por la consola, utilizando para ello el método ***imprime()*** del objeto ***r,*** que le ha sido pasado por referencia en el momento de la creación del hilo. Por su parte, en la función ***main*** se le pide un factor positivo al usuario, con el objetivo de multiplicar todo el vector compartido, por dicho número. El diseño del ejemplo espera que la operación de multiplicación ocurra simultáneamente para todo el vector. Por ello, la condición de carrera entre las operaciones de multiplicación y salida por pantalla son evitadas mediante el ***mutex m***. La exclusión mutua se podría haber abordado, utilizando directamente los métodos ***lock()*** y ***unlock()*** de ***m***; pero el estándar brinda y recomienda el uso de la clase genérica ***std::lock_guard***, que implementa el patrón **RAII** cada vez que se crea en las llamadas a los métodos de la clase compartida que acceden al vector. Luego se tienen otros detalles menores en el ejemplo que se puede seguir sin dificultad en el código, como el uso del miembro público **salir** de **r** para comunicar al hilo la intención del usuario de abandonar el programa, o el uso de funciones de la librería definida en <***chrono***> para especificar tiempos, entre otros.
Nótese que en el ejemplo, el diseño del esquema de exclusión mutua tiene una granularidad muy gruesa, que puede tener un impacto importante en el tiempo de ejecución y la latencia del programa para vectores muy grandes. Este tipo de compromiso entre la seguridad de la aplicación multiprogramada y sus

prestaciones no puede ser resuelto de forma general. Es por esta buena razón que las clases de contenedores STL no han sido diseñada para ser seguras en estas situaciones. De todas formas, la discusión en el aula de estas alternativas, sin pretender dar soluciones definitivas, pero realizadas a un nivel de abstracción alto, sería el objetivo docente a perseguir.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>

class Recurso
{
public:
    bool salir=false;

    void mult(float factor)
    {
        std::lock_guard<std::mutex> guard(m);
        for (auto &elem: vec )
            elem*=factor;
    }
    void imprime()
    {
        std::lock_guard<std::mutex> guard(m);
        for (auto elem: vec )
            std::cout << elem << " ";
        std::cout << std::endl;
    }
private:
    std::mutex m;
    std::vector<float> vec {1,5,7,8,9,10,11,45,-12};
};
void ImprimeVec(Recurso &r)
{
    while (!r.salir)
    {
        std::this_thread::sleep_for(std::chrono::seconds(5));
        r.imprime();
    }
}
int main()
{
    Recurso rec;
    std::thread t(ImprimeVec,std::ref(rec));
    do
    {
        float f;

        std::cout << "Entre factor (negativo salir)" << std::endl;
        std::cin >>  f;
        if (f>=0)
            rec.mult(f);
        else
            rec.salir=true;
    }
    while (!rec.salir);
    t.join();
    return 0;
}
```

Fig. 4. Ejemplo RAII con **mutex** en C++v11.

El patrón RAII, es un ejemplo importante y general, y nos sirve aquí para resaltar la importancia de discutir en la docencia buenas prácticas y patrones de programación que han probado su utilidad en múltiples escenarios [3, 4, 9]

CONCLUSIONES

Se propone una actualización de los contenidos de la asignatura para situarlos a medio camino en el espectro (fig. 1) y que va desde soluciones de bajo nivel de abstracción y muy generales a otras muy específicas de control, creemos que es una decisión estratégicamente adecuada y que puede ser generalizable.
La experiencia con la introducción del C++ a partir del C, en estos dos primeros años ha sido muy exitosa. Se debe abandonar la pretensión de dar una cobertura completa del lenguaje, y si los elementos fundamentales, para animar al estudiante a consultar la bibliografía y ponerlos en disposición de explorar, por ejemplo, las ventajas de la extensa funcionalidad de la biblioteca estándar.
Los autores esperan que la introducción de los elementos de concurrencia dentro del C++ moderno, de una mayor coherencia a la asignatura. Pretendemos, como trabajo futuro inmediato, ofrecer una estructura de clases similar para encapsular el acceso a los procesos POSIX bajo UNIX.

Agradecimientos

Este artículo ha sido realizado en el marco del Proyecto de Innovación Docente PID14-15N39 de la UVA.

Referencias

[1] K. J. Aström and P. R. Kumar, "Control: A perspective," *Automatica*, vol. 50, no. 1, pp. 3–43, 2014.

[2] M. Barr, *Programming Embedded Systems in C and C++*, no. January. 1999.

[3] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Eng. Pract.*, vol. 21, no. 11, pp. 1608–1619, 2012.

[4] B. P. Douglass and D. Ph, *Real-Time Design Patterns*. 1998.

[5] C. Kormanyos, *Real-Time C++. Efficient Object-Oriented and Template Microcontroller Programming*. Springer, 2013.

[6] K. H. John and M. Tiegelkamp, "IEC 61131-3: Programming industrial automation systems: Concepts and programming languages, requirements for programming systems, decision-making aids," *IEC 61131-3 Program. Ind. Autom. Syst. Concepts Program. Lang. Requir. Program. Syst. Decis. Aids*, pp. 1–390, 2010.

[7] R. Mazaeda, E. de la Fuente, J.L. Sánchez, E. Moya, Sobre la Docencia en la Informática Industrial. *XXXVI Jornadas de Automática*. Bilbao, 2015.

[8] E. S. Roberts, *Programming Abstractions in C++*. 2012.

[9] R. Sanz and J. Zalewski, "Pattern-based control systems engineering," *Control Syst. Mag. IEEE*, vol. 23, no. 3, pp. 43–60, 2003.

[10] A. Williams, *C++ Concurrency in Action. Practical Multithreading*. Manning, 2012.

[11] L. Da Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Trans. Ind. Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[12] A. Walter, S. Karnouskos, and P. Leita, "Computers in Industry Industrial automation based on cyber-physical systems technologies : Prototype implementations and challenges," 2015.