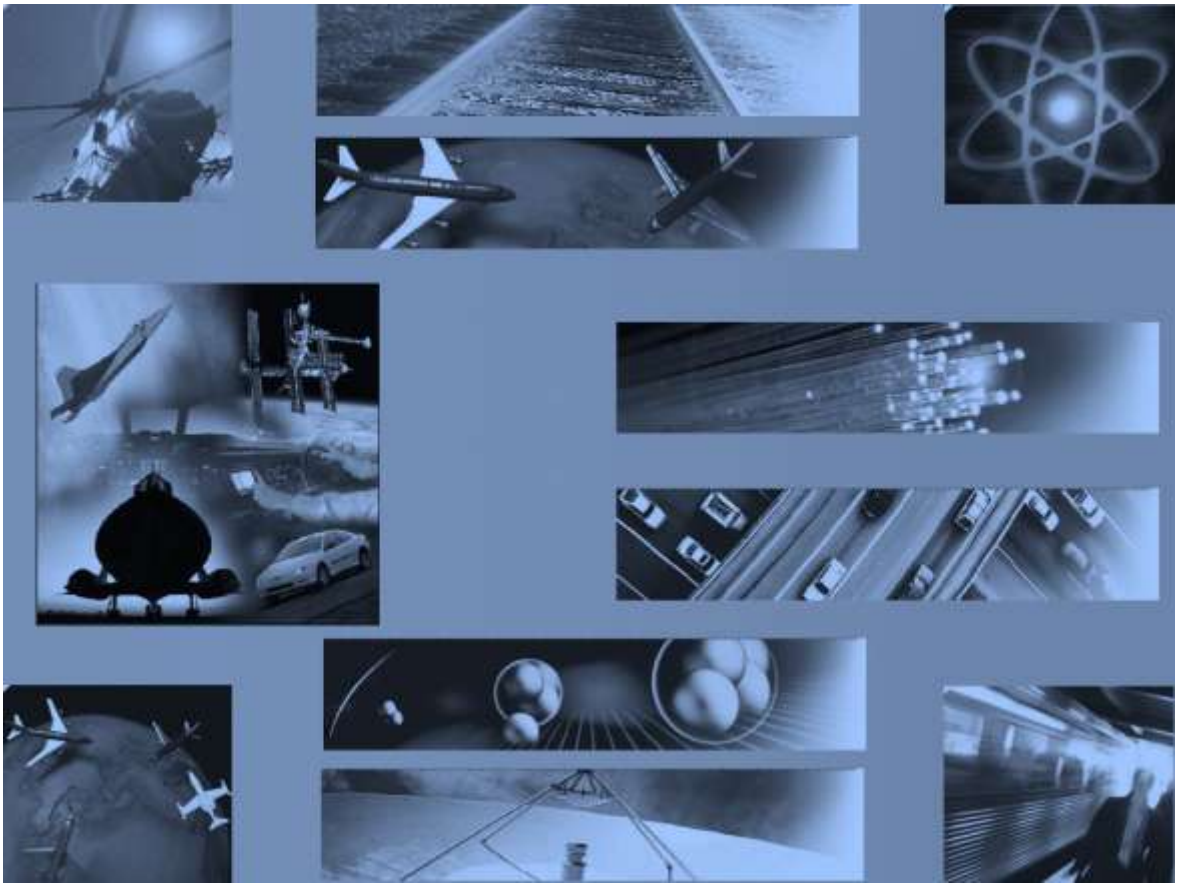
PRÁCTICAS DE TIEMPO REAL



GRADO DE INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA

C makes it easy to shoot yourself in the foot;

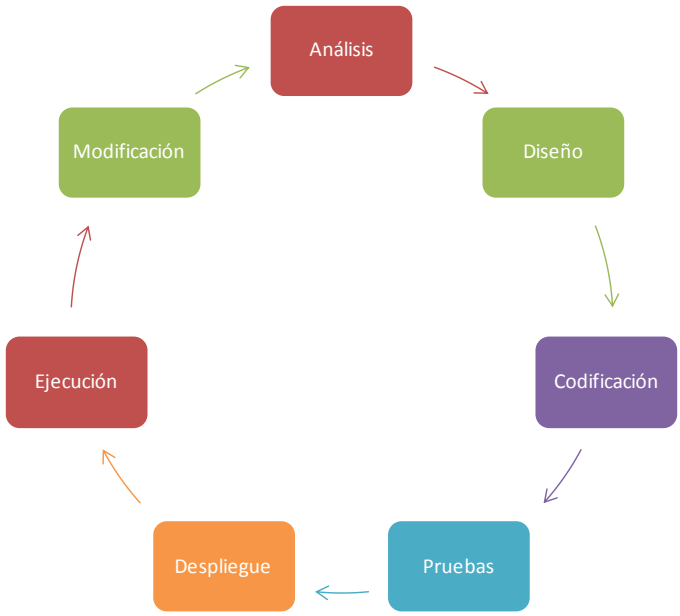
C++ makes it harder, but when you do, it blows your whole leg off

Bjarne Stroustrup (circa 1986)

INTRODUCCIÓN AL DESARROLLO DE PROGRAMAS DE ORDENADOR

DEFINICIÓN DEL PROBLEMA Y DISEÑO DE LA SOLUCIÓN

Desarrollar un programa es parte de un proceso que empieza por la definición del problema y el diseño de una solución. Estos dos pasos se abordan antes de comenzar a escribir una sola línea de código. Si el problema es grande y complicado será necesario aplicar técnicas que permitan descomponerlo en problemas más pequeños cuyas soluciones puedan integrarse para resolver el problema inicial. Pero aunque el problema se reduzca a plantear un pequeño algoritmo, sigue siendo necesario comprender el problema y solucionarlo antes de empezar a codificar la solución[1]. Además, en este paso aparece una complicación extra que a menudo es difícil de percibir por parte de los nuevos programadores: una cosa es que usted logre saber cómo se resuelve un problema, y otra cosa distinta es que esa forma que usted conoce y comprende sea fácilmente expresable en un programa. Ante cada problema usted debe llegar a un procedimiento que, además de conducir a la solución al problema, también sea expresable en un programa. Es decir, debe plantear la solución pensando y utilizando las abstracciones que proporciona el lenguaje de programación. Y esta no es una tarea trivial, sino que requiere oficio y práctica. La siguiente figura muestra las etapas habituales en el desarrollo de un programa de ordenador.



En la fase de análisis se determina qué queremos haga nuestro programa. En el diseño se da una solución arquitectónica tomando decisiones de cómo queremos sea la solución. En la fase de codificación trasladamos la arquitectura junto con otras decisiones de diseño a una representación en un lenguaje de programación, que puede ser ejecutada en un computador tras obtenerse el código ejecutable como resultado de la compilación del programa. Antes de tener la versión definitiva, es imprescindible probar que el programa funciona según lo previsto. Cuando se tienen todas las garantías, el programa puede ser puesto en producción. Tras cierto tiempo, es probable se detecten errores, inconsistencias o sea necesario modificar el programa por cambios impuestos.

[1] Cuando el problema está muy acotado, por ejemplo un algoritmo para solucionar un problema muy concreto, la solución puede expresarse directamente en el lenguaje de programación. Esta forma de actuar deja de ser viable en cuanto el problema tiene una entidad que hace necesaria su descomposición en problemas parciales, lo cual ocurre sorprendentemente pronto.

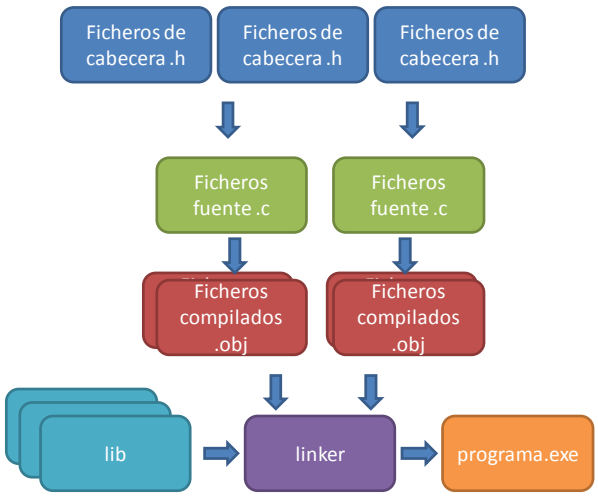
CODIFICACIÓN

Una vez resuelto el problema se pasa a codificar la solución, creando un archivo o un conjunto de archivos que contengan el código fuente. Para ello se necesita un programa *editor* que permita escribir y guardar el programa. Durante la codificación es muy habitual incorporar código disponible en bibliotecas software, que puede haber sido desarrollado por el propio programador o por terceros. La reutilización es un aspecto fundamental del desarrollo software. Los lenguajes de programación establecen mecanismos para importar y utilizar dichas librerías aunque no estén disponibles los archivos con su código fuente. Es importante, aunque no necesario, seguir unas "buenas prácticas" de programación a la hora de escribir el código. Las primeras "buenas prácticas" que debe aprender son:

1. *Utilizar nombres significativos para variables y funciones*. Hilando más fino, se recomienda utilizar sustantivos para los nombres de variables y verbos para las funciones.
2. *Documentar adecuadamente el código*. No se trata de documentar hasta la última declaración de variable, pero sí aquellos detalles que ayudan a entender por qué se ha escrito ese código. La documentación es esencial para entender o modificar código fuente.

COMPILACIÓN Y ENSAMBLADO DEL PROGRAMA

El código fuente no puede ser ejecutado directamente por ningún computador: es necesario traducirlo a código máquina. Para ello se necesita un *compilador* o un *intérprete*. Existen lenguajes de programación, como C, en que el código fuente es traducido por un *compilador* a código máquina y se genera un archivo ejecutable. Otros lenguajes, como Java, se compilan a un lenguaje intermedio (que en el caso de Java se denomina *bytecode*) que posteriormente es interpretado y ejecutado por un programa especial que se denomina *intérprete*. Además, los programas habitualmente no se encuentran en un único archivo, sino distribuidos en muchos, que deben ensamblarse para dar lugar a un programa ejecutable. La obtención de un programa ejecutable implica, por un lado, la traducción de los archivos en código fuente a archivos con código objeto y, por otro, el ensamblado de los archivos con código objeto. El proceso de generación de un ejecutable se muestra en la siguiente figura:



Como puede observarse, el proceso comienza con la *compilación* del código C (ficheros con extensión .c en la figura), que a su vez utilizan código de bibliotecas ya desarrolladas (ficheros con extensión .h en la figura). Como puede verse, un programa puede (y suele) reutilizar gran cantidad de código, y suele estar formado por varios ficheros de código fuente, ya que así se permite y facilita la

organización del código. Tras el proceso de compilación se genera un fichero de código objeto (fichero con extensión .obj en la figura) que contiene la traducción del código C de los ficheros .c a lenguaje máquina, que es el que entiende el computador. Pero el proceso no termina aquí, ya que hay un siguiente paso, el *enlazado*, en el que un programa enlazador coge todos los fichero .o y les añade el código de las bibliotecas (archivo .lib, suministrado por el propio compilador) que utiliza el código fuente y genera finalmente el ejecutable. Resumiendo, el proceso de generación de un ejecutable a partir del código fuente consiste fundamentalmente en dos pasos:

- El *compilador* genera el código objeto a partir de cada fichero de código fuente, sin tener en cuenta el uso de bibliotecas externas.
- El *enlazador* coge todos los ficheros con código objeto, resuelve las dependencias de bibliotecas externas, y genera finalmente el código ejecutable.

EJECUCIÓN, PRUEBA Y DEPURACIÓN DEL PROGRAMA

Dependiendo del lenguaje y de la plataforma de ejecución, para ejecutar un programa puede bastar con escribir su nombre como un comando en una consola o hacer doble clic en un icono o puede ser necesario utilizar algún programa auxiliar que lance al nuestro. En cualquier caso, que el programa sea capaz de ejecutarse no significa que realice correctamente la tarea que tiene encomendada. En programación es muy difícil acertar a la primera. Normalmente es necesario ejecutar el programa una y otra vez hasta conseguir detectar y corregir todos sus fallos. Existen técnicas para probar los programas de forma exhaustiva y utilidades (*depuradores*) que ayudan a detectar y corregir los errores de programación. Los errores de programación se clasifican en:

- **Errores de compilación** (sintácticos). Detectados por el compilador cuando el código fuente no se ajusta a lenguaje C. Ejemplos: falta de paréntesis, llaves, punto y coma al final de cada línea, utilización de palabras reservadas como identificadores, etc.
- **Errores en tiempo de ejecución** (semánticos). El programa compila pero no resuelve correctamente el problema, su comportamiento no se ajusta al esperado, el programa "se cuelga", etc. Estos errores son generalmente difíciles de detectar, ya que el compilador no ayuda a detectarlos. Hay que recurrir al uso de técnicas de depuración, como la impresión de trazas del funcionamiento del programa y al uso de programas depuradores.

La corrección de un error semántico puede dar lugar a replantearse el problema desde el principio, de ahí que en la figura 1 se pueda regresar desde la prueba del programa a cualquier paso anterior. Los ciclos de la figura 1 revelan una característica esencial del proceso de desarrollo software: es un **proceso iterativo e incremental** que incluye varios ciclos en cada uno de los cuales se resuelven los problemas encontrados en el ciclo anterior, se refina la solución y se incorpora nueva funcionalidad. Cada fase produce sus propios artefactos que sirven para abordar la siguiente fase y que son revisados y actualizados en cada iteración.

Por último, unas reflexiones finales. Cuide la calidad del software, cómo escribe el código fuente (preste especial cuidado a la indentación del mismo y cumpla las reglas de nombrado: utilice verbos en minúsculas para las funciones y sustantivos para variables y tipos de datos, éstos últimos en mayúsculas) y la documentación del mismo, ya que se estima que el 60-70% del tiempo un programador está consultando o modificando código hecho por uno mismo u otra persona. Aprenderá técnicas para realizar programas más grandes, pero tendrá que mejorarlas con práctica y aprender

nuevas, como especificación de requisitos, diseño arquitectónico, diseño de pruebas del software, etc. Es decir, es necesario realizar un diseño detallado del software, sus planos, antes de construirlo.

Debe también mejorar sus habilidades depurando código, es decir, buscando un fallo semántico (es decir, el código compila pero no hace lo que tiene que hacer). Existen dos formas fundamentales depurar código:

1. Compilación para el depurador (opción –g) y con las optimizaciones que no afectan al depurados (-Og). Luego utilizar un depurador, ya sea el integrado en el propio IDE o gdb.
2. Ejecución del programa con sentencias de impresión adicionales que muestren por pantalla el valor de algunas variables (traza de ejecución). Esta estrategia es rápida y sencilla, pero no escala cuando el fallo que estamos buscando aparece al procesar grandes estructuras de datos o programas grandes.

En ambos casos es fundamental tener una idea de por dónde (en qué sección de código) se puede encontrar el error. Y cuando está depurando espere encontrarse casi "cualquier cosa". A veces cometemos errores porque desconocemos de verdad qué está haciendo nuestro código y hacemos asunciones erróneas.

Las aserciones son también una posibilidad muy útil en ambos casos, puesto que se activan cuando falla alguna condición y no imprimen valores constantemente. Además, se pueden eliminar del código de producción fácilmente. Existen dos tipos de aserciones: en tiempo de ejecución (función `assert`) y en tiempo de compilación (`static_assert`). Las aserciones están definidos en la librería `<cassert>`, aunque en general se recomienda definir nuestro propio sistema de aserciones que, por ejemplo, lance una excepción en lugar de terminar el programa si la aserción no se cumple. La ejecución de las aserciones se controla mediante el símbolo del preprocesador NDEBUG (con `#define`).

Por último existen otros programas que nos ayudan a analizar el código desde otros puntos de vista: rendimiento (*profilers*, como **gprof**), gestión de memoria (**valgrind**), analizador del código fuente (**clint**), pruebas de rendimiento (*unit testing*, **Catch**). Los grandes proyectos organizan el código en librerías, estáticas o dinámicas (las famosas *dlls*), utilizan software para gestionar la compilación (como **CMake**) y por supuestos sistemas de control de versiones (como GIT) y de documentación (como Doxygen). Estas herramientas no solo son propias del mundo de C++ (algunas sí), sino que se utilizan en otros muchos lenguajes de programación. Una inversión en conocimiento si le gusta el mundo de la programación y quiere dedicarse a él profesionalmente. Lea los capítulos 26 y 27 del libro recomendado para profundizar en el tema de depuración y pruebas unitarias.

PRÁCTICA 1: PROGRAMACIÓN BASADA EN OBJETOS CON C++

STRINGS Y COLECCIONES (2 SESIONES)

La programación basada en objetos consiste en la utilización de clases definidas por terceros, generalmente la librería estándar de C++, pero no considera la definición de clases, solo la instanciación de objetos y su utilización. Como tal, es un paso previo a la orientación a objetos: antes de definir una clase (e instanciar y utilizar luego objetos de dicha clase) es conveniente saber utilizar objetos.

1.1. IMPRIMIR POR SEPARADO LAS PALABRAS DE UNA LÍNEA DE TEXTO

Se pide que haga un programa que lea una línea de texto e imprima cada palabra en su propia línea por consola. Primero piense cómo se puede detectar una palabra y el final del stream (¿constante EOF?, '\n', u otro?). Puede utilizar las funciones `operator<<, getline` de la librería `<string>`. Pruebe varias alternativas, como por ejemplo procesamiento carácter a carácter. Amplíe el programa para que no imprima los signos de puntuación habituales: , . ; ¿ ? ¡ ! : etc. (utilice la función `ispunct`). Pruebe a crear un fichero de texto con las líneas de prueba y a ejecutar desde la consola "*`programa.exe < entrada.txt`*" (redirección) para que sea más cómodo.

1.2. ALMACENAR, ORDENAR E IMPRIMIR LAS PALABRAS DE UNA LÍNEA DE TEXTO

Mejore la versión anterior de forma que ahora almacene cada palabra en un vector de string, y luego imprímalas por consola, una palabra por línea, pero esta vez ordenadas alfabéticamente (una versión de mayor a menor y otra de menor a mayor). Utilice la función `sort` definida en la librería `<algorithm>`. ¿Qué sucede con las palabras en mayúsculas y minúsculas?. Prueba a introducir la frase: "Programación de Tiempo Real o programación de tiempo real" y compare resultados. ¿Cómo puede solucionarlo?. Intente redireccionar la entrada de datos como antes.

1.3. UTILIZACIÓN DE FICHEROS

Modifique el programa 1.3 para que pueda leer el texto de un fichero y guardar el resultado en fichero. Este nuevo programa debe solicitar los nombres de los ficheros de entrada y salida y comprobar que son diferentes antes de proceder. Pruebe con el fichero del Quijote. La librería `<fstream>` proporciona las clases `ifstream`, que representa un fichero de lectura, y `ofstream`, que representa un fichero de salida. Ambas clases proporcionan funciones para abrir y cerrar ficheros, `operator<<` y `operator>>, getline`, etc. Consulte la documentación.

1.4. HISTOGRAMA DE LAS PALABRAS DEL QUIJOTE

Amplíe el programa anterior para crear un histograma de las palabras del Quijote mediante un `map<string, int>`. Muestre las dos palabras más repetidas y la cantidad de veces que se repiten (*de: 123, y: 104*). ¿Qué palabras acumulan el 20% de las apariciones? (*de: 123, y: 104, que: 88, a: 44, el: 40*).

1.5. HISTOGRAMA POR LONGITUD DE PALABRA

Amplíe el programa anterior para que ahora se cree un histograma por cada longitud de palabra. Es decir, habrá un histograma para almacenar el número de repeticiones de las palabras de 1 letra, las de 2 letras, etc. Suponga que la palabra más grande es de 15 letras. Utilice un vector de mapas: `vector<map<string,int>>`. Solución: *104 y, 123 de, 88 que, 13 como, 6 todos, 14 había, 5 amadís, 3 pareció, 13 caballero, 2 añadidura, 2 pensamiento, 3 pensamientos, 2 significativo, 1 encantamientos, 1 desentrañarles.*

PRÁCTICA 2: PROGRAMACIÓN ORIENTADA A OBJETOS CON C++

SIMULADOR ELECTRÓNICA DIGITAL (3 SESIONES)

La programación orientada a objetos consiste en la definición de clases y la instanciación y utilización de objetos de dichas clases, junto con la aplicación de composición, herencia y polimorfismo. La orientación a objetos es muy potente y requiere tiempo para poder controlarla. Como no se dispone de tanto tiempo como sería deseable, en esta práctica se proporciona el diseño de un sencillo simulador de electrónica digital, básicamente del tipo puertas lógicas, y se piden algunas extensiones que requieren entender algunas partes del código proporcionado (o todas para que la experiencia sea completa) .

Antes de empezar con el ejercicio del simulador y a modo de introducción a la programación orientada a objetos se propone el siguiente ejercicio:

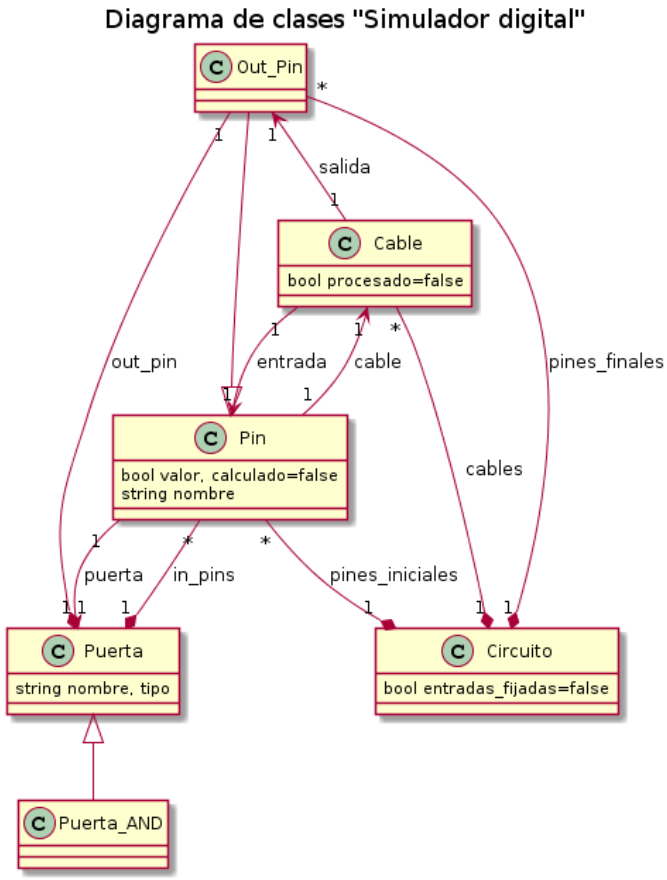
2.1. COMPLETAR LA DEFINICIÓN DEL "TIPO COMPLEJO"

Se pide completar la definición del tipo complejo presentado en las transparencias de clase con la siguiente funcionalidad:

- Representar el número complejo mediante módulo y argumento. Ojo, ¡tiene que mantener ambas representaciones siempre coherentes!.
- Modificar el constructor para poder crear un número complejo de cualquiera de las dos formas. Esto crea un problema, ya que no se pueden añadir dos constructores que aceptan parámetros del mismo tipo: y es que la pareja "parte real e imaginaria" y "módulo y argumento" son, en ambos casos, números reales. Añada un tercer parámetro al constructor de un tipo enumerado con valores **`reIm`**, **`modArg`** para diferenciar el tipo de construcción que se quiere realizar.
- Método para calcular el conjugado de un número complejo, así como para obtener (getter) cada valor por separado (método **`getParteReal`**). Cree también una pareja de métodos para fijar de manera conjunta una de las representaciones (**`setBinomio`**, **`setPolar`**). Ojo, ¡tiene que mantener ambas representaciones siempre coherentes!.
- Operadores para realizar la resta, multiplicación y división de números complejos. Suma y resta se realizan fácilmente utilizando la representación como binomio (parte real, parte imaginaria), mientras que multiplicación y división utilizan la forma polar (módulo y argumento). Utilice como ejemplo la definición de **`operator+`** proporcionada.

2.2. SIMULADOR ELECTRÓNICA DIGITAL

A continuación se muestra un diagrama de clases de UML[2]. Este diagrama muestra las clases que forman un programa, sus atributos y las relaciones de composición y herencia que existen entre ellas. Este diagrama permite entender "de un vistazo" la estructura general de una aplicación. El significado de los elementos que aparecen en el diagrama es:



- La flecha ⇽ indica relación de **herencia**. Por ejemplo, la clase **`Puerta_AND`** hereda de **`Puerta`**.
- La línea con rombo ◆ indica relación de **composición**, que tiene un nombre, mientras que los números en ambos extremos de la línea indican la cardinalidad de la relación. Por ejemplo, la clase **`Circuito`** contiene una relación con **`Out_Pin`** que se llama **`pines_finales`**, y la cardinalidad indica que 1 objeto de la clase **`Circuito`** contiene muchos (*) **`Out_Pin`**. Y la dirección de navegación es del rombo a la clase: desde un circuito accedo a un out_pin, pero no viceversa.
- La línea normal indica **asociación**, que es una forma débil de composición. Aplican las mismas reglas que se acaban de describir. No existe diferencia a la hora de implementar asociación y composición, salvo el hecho de que generalmente se entiende que en las relaciones de composición, la clase que tiene el rombo se encarga también de crear y gestionar los objetos de las clases que contiene, mientras que en la asociación los objetos se crean de manera independiente y se modela que guardan una relación entre ellos.

Este diseño representa una de las posibles soluciones al problema planteado, pero no la única. Por ejemplo, puede sorprender que no exista una relación entre **`Circuito`** y **`Puerta`**. Pero no ha hecho falta ya que he podido resolver el problema utilizando los objetos de la clase **`Cable`**. Aunque esto no quiere decir que para realizar una futura extensión no es necesario tener una relación entre ambas clases. Ni por supuesto que el diseño propuesto sea el mejor. Seguro que se pueden realizar diseños con más o menos clases. Recuerde: el problema de diseño es siempre un problema abierto, con múltiples soluciones posibles. Es parte de la gracia ☺.

[2] El *Lenguaje de Modelado Unificado* (UML en inglés) es un lenguaje gráfico que proporciona varios diagramas que permiten modelar el diseño del software desde distintos puntos de vista: requisitos, diseño de alto nivel (diagrama de despliegue, componentes y clases) y diseño de detalle (diagrama de objetos, máquinas de estados, actividades, secuencia, etc.). Es ampliamente utilizado y conocido en el mundo de diseño software.

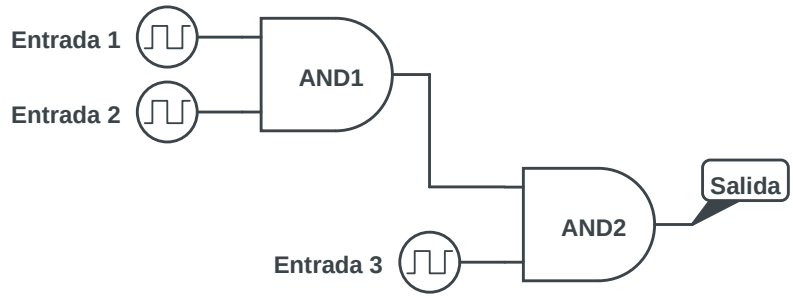
El propósito de las clases mostradas en el diagrama de clases es el siguiente:

- La clase **Puerta** se utiliza como clase base para crear el resto de puertas lógicas a partir de ésta y define bastante funcionalidad común a todas las posibles clases hijas. Las nuevas puertas lógicas hacen uso de esta funcionalidad y solo tienen que definir un constructor adecuado y sobrescribir la función *virtual* **actualizar()**, tal y como se muestra en la clase **Puerta_AND**. El destructor de la clase se define como *virtual* porque es una clase base de la cual van a heredar otras muchas.
- Las clases **Pin** y **Out_Pin** representan los pines de entrada y salida de una puerta lógica. Solo contienen los datos necesarios y una función para imprimir (sobrecarga de la función **operator<<**). Observe cómo se instancian los objetos de estas clases desde el constructor de **Puerta**. Es una implementación de la relación de composición entre ambas clases.
- La clase **Cable** representa la unión entre un pin de entrada (instancia de la clase **Pin**) y uno de salida (instancia de la clase **Out_Pin**). La clase **Puerta** se ha modelado para tener un solo pin de salida pero múltiples de entrada. Esta clase define de nuevo solo datos y el constructor necesario para su utilización, aunque se asegura que la conexión se realiza entre un pin de entrada y otro de salida, y no por ejemplo entre dos pines de entrada o dos de salida (que no tendría sentido). Es decir, el diseño de la clase evita una posible fuente de errores a la hora de crear el circuito que será luego simulado.
- La clase **Circuito** es la más complicada de todas, ya que no solo contiene los elementos que definen el circuito: los pines de entrada al circuito (cuyos valores son fijados por el usuario mediante la función miembro **fijar_entradas())**, los pines de salida del circuito (cuyos valores se calculan mediante la función miembro **calcular()** y se imprimen en consola mediante **mostrar_salidas()**), y los cables que definen las conexiones entre los pines de las puertas que forman el circuito (creados mediante la función **add_cable_entre(),** que a su vez utiliza **std::emplace_back** para crear el objeto de tipo **Cable** directamente en su vector). Hay algunas funciones extras, para imprimir todos los cables que definen el circuito, y constructores, pero son fáciles de entender. La función importante es **calcular(),** que solo puede ser invocada si previamente se ha invocado **fijar_entradas().** Vea en esta función la utilización de **std::initializer_list** para pasar varios valores como entradas. La función **calcular()** básicamente recorre la lista de **Cables** del circuito, copiando los valores de un extremo a otro e invocando la función **actualizar()** sobre la **Puerta** que contiene el valor del pin recién copiado. Si dicha **Puerta** tiene ya valores en todas las entradas actualiza la salida. Y así continúa hasta resolver el circuito. Claro, esto asume que el circuito está bien creado y que, por ejemplo, no se ha dejado "al aire" alguno de los pines de entrada de una puerta. Aunque para evitar que en este caso la computación no terminara nunca, se ha limitado el número máximo de iteraciones que se realizan para resolver el **Circuito**.

Observe como en casi todas las clases se han eliminado el *constructor de copia* y el *operador de asignación por copia*, mientras que se han añadido el *constructor de movimiento* y el *operador de copia por movimiento*. De esta forma se evita la copia de objetos a partir de otros (que en este caso se ha decidido que no tiene sentido), pero sí se permite que dichos objetos se muevan. Esto último es necesario por ejemplo en objetos que se utilizan en colecciones (vector, mapa, etc.), ya que la implementación de la colección suele copiar/mover objetos cuando realiza determinadas operaciones (por ejemplo, cuando añadimos nuevos elementos).

2.2.1 SIMULACIÓN DE CIRCUITO SENCILLO

Como ejemplo se va a utilizar un circuito muy sencillo: dos puertas AND conectadas entre sí, tres entradas de usuario y una salida, tal y como muestra la siguiente figura:



En este caso, solo cuando las tres entradas son **`true`** el valor de salida es **`true`** también. Con las clases que se proporciona, cree un nuevo fichero con la función *main* en la que simularlo utilizando el siguiente código (faltan añadir los **`#includes`** necesarios):

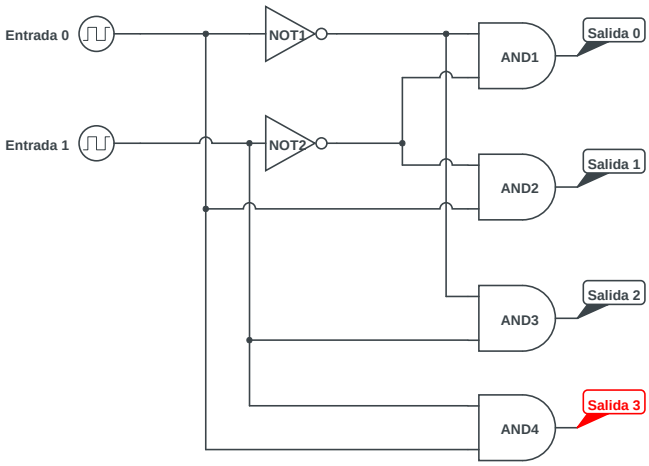
CÓDIGO DEL CIRCUITO INICIAL DE PRUEBAS

```
circuito::Puerta_AND p1 {"P1"};
circuito::Puerta_AND p2 {"P2"};

circuito::Circuito circ {3,1};

circ.add_cable_entre (p1.get_in_pin(0), circ.get_pin_inicial(0));
circ.add_cable_entre (p1.get_in_pin(1), circ.get_pin_inicial(1));
circ.add_cable_entre (p2.get_in_pin(0), p1.get_out_pin());
circ.add_cable_entre (p2.get_in_pin(1), circ.get_pin_inicial(2));
circ.add_cable_entre (circ.get_pin_final(0), p2.get_out_pin());

circ.fijar_entradas ({true, true, true});
circ.calcular ();
circ.mostrar_salidas ();
```

Una vez añadido el código, compruebe que efectivamente la salida del circuito es solo **`true`** cuando las tres entradas son **`true`**. Tómese su tiempo para revisar también el código suministrado.

2.2.2 SIMULACIÓN DE CIRCUITO DECODIFICADOR DE 2 ENTRADAS

El siguiente objetivo consiste en simular el circuito de un decodificador de entradas, implementado como se muestra a continuación:



E0	E1	SALIDA ACTIVA
F	F	Salida 0
F	T	Salida 2
T	F	Salida 1
T	T	Salida 3

Ahora se pide que amplíe la funcionalidad con la puerta NOT. Utilice la definición de la puerta AND como modelo. El equivalente de este circuito se muestra en el siguiente código, que debe completar convenientemente.

PARTE DEL CÓDIGO CIRCUITO DECODIFICADOR

```
circuito::Puerta_AND p_a0 {"P0"};
circuito::Puerta_AND p_a1 {"P1"};
circuito::Puerta_NOT p_n0 {"P0"};
// TO DO: Crear las puertas que faltan siguiendo el esquema propuesto

circuito::Circuito circ {2,4};

// Entrada 0
circ.add_cable_entre(p_n0.get_in_pin(0), circ.get_pin_inicial(0));
circ.add_cable_entre(p_a3.get_in_pin(1), circ.get_pin_inicial(0));
circ.add_cable_entre(p_a1.get_in_pin(0), circ.get_pin_inicial(0));
// Entrada 1
circ.add_cable_entre(p_n1.get_in_pin(0), circ.get_pin_inicial(1));
circ.add_cable_entre(p_a3.get_in_pin(0), circ.get_pin_inicial(1));
circ.add_cable_entre(p_a2.get_in_pin(0), circ.get_pin_inicial(1));
// Salida puerta NOT 0
circ.add_cable_entre(p_a0.get_in_pin(0), p_n0.get_out_pin());
circ.add_cable_entre(p_a2.get_in_pin(1), p_n0.get_out_pin());
// TO DO: completar la Salida puerta NOT 1
// ??????????????

// Conexión con las salidas
circ.add_cable_entre(circ.get_pin_final(0), p_a0.get_out_pin());
circ.add_cable_entre(circ.get_pin_final(1), p_a1.get_out_pin());
```

```
circ.add_cable_entre(circ.get_pin_final(2), p_a2.get_out_pin());
circ.add_cable_entre(circ.get_pin_final(3), p_a3.get_out_pin());
circ.fijar_entradas({false, true});
// ojo, el primer valor se corresponde a A0 y el segundo a A1!!!
circ.calcular();
circ.mostrar_salidas();
```

Y no se olvide de comprobar que el circuito funciona correctamente. Es decir, que se activa la puerta de salida correspondiente a la entrada fijada, y solo ésa.

2.2.3 CÁLCULO DE TIEMPOS DE PROPAGACIÓN

Amplíe la funcionalidad del programa para tener en cuenta los tiempos de propagación de las señales lógicas y los tiempos de cambio de las puertas. Las puertas lógicas no cambian el valor del pin de salida instantáneamente cuando cambian los valores de entrada sino que tienen un pequeño retardo del orden de nanosegundos en actualizar la salida. Si consulta algún *datasheet* comercial verá que estos tiempos están en torno a los 50 ns. Aunque son tiempos muy pequeños, en los circuitos lógicos tiene las siguientes repercusiones:

- Que las señales lógicas que llegan a una puerta estén muy desfasadas entre sí, debido por ejemplo a que una de ellas ha pasado por más puertas lógicas que la otra. Esto se conoce como *glitch*.
- Estos retardos marcan la frecuencia máxima a la que puede operar el circuito.

Tiene que añadir una propiedad tiempo a la clase **`Pin`**, **`Puerta`** y **`Cable`**, para tener en cuenta los retardos introducidos. Y ampliar los constructores para permitir que el usuario pueda especificar dichos retardos. Los pines no introducen retardo, pero es necesario que puedan almacenar el valor del tiempo con que llegó la señal a la puerta, ya que es tanto ésta como el cable los que introducen el retardo. En el caso de la puerta hay dos retardos: el tiempo del cambio de nivel alto a bajo y el de bajo a alto. Si la salida no cambia la puerta no introduce retardo. Modifique también las funciones que imprimen valores (**`Pin`**, **`Puerta`**, **`Cable`**) para mostrar también los retardos. Le ayudará a depurar el programa.

Para que pueda probar su programa, utilice los valores de retardos para la puerta AND (t_hl=50, t_lh=100) y para la NOT (t_hl=30, t_lh=20). Con estos valores, los circuitos anteriores producen la siguiente salida:

PRIMER CIRCUITO				
Entrada			Salida	
E0	E1	E2	S	t
T	T	T	T	200
T	T	F	F	100
F	X	X	F	0
X	F	X	F	0

DECODIFICADOR DE 2 ENTRADAS			
Entradas		Salidas	
E0	E1	Salida activa	t
F	F	S0	120
F	T	S2	120
T	F	S1	120
T	T	S3	100

2.2.4 OTRAS POSIBLES AMPLIACIONES

Para terminar, se plantean las siguientes mejoras:

- Fijar el valor inicial de la salida de la puerta al crearla.
- Añadir una puerta NAND.
- Comprobar si alguna puerta sufre un desfase mayor entre pines que el dado por el usuario.
- Realizar una simulación pudiendo modificar los valores de las entradas.
- Calcular la frecuencia máxima de cambio de las entradas del circuito simulando todas las posibles entradas y buscando la más desfavorable de todas.

PRÁCTICA 3: PROGRAMACIÓN CONCURRENTE EN C++11 (2 SESIONES)

P3.1 CRIBA DE ERASTÓTENES PARALELA

Es un método para calcular todos los números primos que existen en un rango y que consiste en crear un vector del tamaño del rango. El contenido del vector indica lo que conocemos del número: 'd' (desconocido), 'p' (es primo) y 'c' (es compuesto). Inicialmente todas las posiciones del vector se inicializan a 'd', y se descartan las dos primeras posiciones (asumimos que 0 y 1 son primos). El algoritmo consiste en recorrer el vector desde la posición 2, marcando como primo el primer índice cuyo contenido sea 'd'. A continuación se marcan como compuesto todos sus múltiplos, con cuidado de no sobrepasar el rango inicial. La entrada de la Wikipedia tiene una animación del funcionamiento de la criba. Utilice la clase **std::array** (definida en **<array>**) para almacenar los datos de la criba. Utilice un rango grande (por ejemplo, hasta 1000000), ejecute el programa varias veces y mida los tiempos de ejecución con el siguiente código:

```
#include <chrono>
auto tiempo_ini = std::chrono::steady_clock::now();
//AQUÍ AÑADO EL ALGORITMO CUYO TIEMPO DE EJECUCIÓN QUIERO MEDIR
auto tiempo_fin = std::chrono::steady_clock::now();
std::cout << "\n\nTiempo transcurrido: " <<
std::chrono::duration_cast<std::chrono::microseconds>(tiempo_fin-tiempo_ini).count() <<
" us\n";
// duration_cast permite también convertir el tiempo medido en ms, s, etc. Consulte doc
```

Los 8 últimos primos del rango (para comprobar que efectivamente ha resuelto bien el ejercicio) son los siguientes: `999907, 999917, 999931, 999953, 999959, 999961, 999979, 999983`

Cree varios hilos que vayan recorriendo y rellenando el vector de la criba. No pasa nada si se modifica un mismo elemento varias veces.

Utilice el tipo **std::atomic_char** definido en **<atomic>** como tipo del vector del criba. De esta manera se asegura de que dos o más hilos no procesarán el mismo valor primo. Mida de nuevo los tiempos de ejecución.

3.2. SUMA DE MATRICES PARALELA

Haga un programa que rellene con valores aleatorios (utilice <random>) dos matrices 1000x1000 y luego obtenga el resultado de la suma. Utilice una versión con un solo hilo, otra con dos hilos (suma filas pares/impares) y otra con tres hilos. Calcule e imprima los tiempos de cómputo. Defina las matrices con arrays: **`double m1[1000][1000];`**

3.3. CARRERA DE CABALLOS

Se pide un programa que simule una carrera de caballos en un hipódromo de 3km con hilos. Cada hilo duerme un tiempo aleatorio (según una distribución uniforme), despierta y avanza otro valor aleatorio (según una distribución normal). El que primero recorra la distancia hasta la meta gana la carrera. Utilice la librería **<random>** para simular los valores aleatorios y la función

`std::this_thread::sleep_for(XXX)`, definida en **`<thread>`**, para retrasar la ejecución de un hilo. Tenga en cuenta que también necesitará la librería **`<chrono>`** para especificar tiempos.

Defina una única función para modelar el comportamiento de los caballos (hilos), que acepta los parámetros que necesita para funcionar: un identificador para el caballo (distintos entre sí) y los parámetros de las distribuciones de probabilidad. Defina también una clase **`Hipodromo`** :

```
class Hipodromo {
   // Atributos
public:
   Hipodromo(double longitud_m);
   void registrar_nombre (const std::string &n);
   // Devuelve 'true' si el caballo llega a la meta
   bool avanzar (const std::string &n, double longitud);
   std::string get_ganador ();
};
```

Este clase está encargada de controlar la carrera. Para ello, primero se tienen que registrar los caballos. Una vez la carrera comienza, cada caballo utiliza la función avanzar para recorrer un tramo del hipódromo. Una vez todos los caballos han terminado la carrera, el programa principal pregunta al hipódromo cuál es el caballo ganador. Un ejemplo de ejecución es el siguiente:

```
[Caballo '2'] ha recorrido 151.431555 metros
[Caballo '3'] ha recorrido 157.565124 metros
[Caballo '5'] ha recorrido 146.997255 metros
[Caballo '4'] ha recorrido 158.210302 metros
[Caballo '1'] ha recorrido 133.815653 metros
[Caballo '2'] ha recorrido 208.321527 metros
[Caballo '5'] ha recorrido 171.372775 metros
[Caballo '3'] ha recorrido 136.935994 metros
[Caballo '1'] ha recorrido 107.229596 metros
[Caballo '4'] ha recorrido 154.436871 metros
[Caballo '2'] ha recorrido 125.335133 metros
[Caballo '5'] ha recorrido 132.690533 metros
[Caballo '4'] ha recorrido 121.960589 metros
[Caballo '3'] ha recorrido 157.966570 metros
[Caballo '1'] ha recorrido 169.723486 metros
[Caballo '2'] ha recorrido 136.075108 metros
[Caballo '4'] ha recorrido 116.998911 metros
[Caballo '5'] ha recorrido 148.148844 metros
[Caballo '3'] ha recorrido 156.045766 metros
[Caballo '1'] ha recorrido 176.219034 metros
[Caballo '2'] ha recorrido 169.254356 metros
[Caballo '5'] ha recorrido 121.948946 metros
[Caballo '4'] ha recorrido 167.475451 metros
[Caballo '1'] ha recorrido 163.710310 metros
[Caballo '3'] ha recorrido 141.772390 metros
[Caballo '2'] ha recorrido 138.687069 metros
[Caballo '4'] ha recorrido 145.038606 metros
[Caballo '5'] ha recorrido 153.745093 metros
[Caballo '1'] ha recorrido 144.615695 metros
[Caballo '3'] ha recorrido 144.181311 metros
[Caballo '2'] ha recorrido 125.192851 metros
2 HA LLEGADO EN POSICION 1
[Caballo '2'] ha terminado
[Caballo '1'] ha recorrido 132.757358 metros
1 HA LLEGADO EN POSICION 2
[Caballo '1'] ha terminado
[Caballo '4'] ha recorrido 110.934847 metros
[Caballo '5'] ha recorrido 165.826965 metros
 HA LLEGADO EN POSICION 3
[Caballo '5'] ha terminado
[Caballo '3'] ha recorrido 136.828264 metros
3 HA LLEGADO EN POSICION 4
[Caballo '3'] ha terminado
[Caballo '4'] ha recorrido 159.462947 metros
4 HA LLEGADO EN POSICION 5
[Caballo '4'] ha terminado
--> Ha ganado 2
```

¡¿Cómo lo he hecho?!

Algunos consejos:

- La salida por consola no está sincronizada por defecto, así que si no utiliza un **`mutex`** los mensajes saldrán entremezclados (busque en las transparencias).
- Puede utilizar un **`mutex`** para sincronizar el punto en que todos los caballos comienzan la carrera (es decir, a utilizar la función **`avanzar`** de **`Hipodromo`**). Para ello, la función principal adquiere el **`mutex`** nada más comenzar y lo suelta una vez que ha creado todos los hilos. Y cada hilo, antes de utilizar avanzar, intenta coger el **`mutex`** y lo libera a continuación.
- Asegúrese de crear un único objeto de la clase **`Hipodromo`**. Si no, ¡cada caballo correrá solo!.
- Puede utilizar un **`mapa`** para que el **`Hipodromo`** almacene la distancia recorrida por cada caballo. En cualquier caso, recuerde proteger este recurso frente al acceso concurrente.

- Puede almacenar los hilos que representan los caballos en un **`vector`**, y utilizar **`emplace_back`** o **`push_back`** para crearlos.

3.4. 1 PRODUCTOR / 1 CONSUMIDOR

Haga un programa en el que un hilo "*productor*" genera un valor aleatorio según una distribución de Poisson (consulte la documentación de **`<random>`**), lo imprime por consola y lo almacena en una variable compartida con otro hilo, y luego duerme un valor aleatorio (según una distribución normal). Un segundo hilo "*consumidor*" debe leer dicho valor, imprimirlo por consola y dormir otro valor aleatorio (según distribución normal). El productor no produce un nuevo valor (se bloquea) hasta que el consumidor lo ha recuperado, y el consumidor no consume dos veces el mismo valor (se bloquea). Utilice **`<mutex>`** y **`<condition_variable>`**. Para la variable de condición tiene que utilizar un unique_lock, que puede ser creado al principio de cada hilo de la siguiente manera para que no intente coger el mutex:

```
std::unique_lock<std::mutex> ul {barrera, std::defer_lock};
```

Mejora: implemente un buffer compartido (**`std::vector`**) para que el productor pueda almacenar varios valores antes de que el consumidor los consuma. Tiene que proteger todos los accesos a la variable compartida. Utilice **`push_back`** para añadir elementos al final del vector y una combinación **`front`** y **`erase`** para recuperar y eliminar el primer elemento añadido

3.5. 1 PRODUCTOR / 4 CONSUMIDORES

Amplíe el programa anterior para que funciones con 4 consumidores. De esta manera, el productor no generará un nuevo número hasta que los 4 consumidores lo hayan consumido, y cada consumidor se bloqueará si intenta leer dos veces el mismo valor. Necesitará una forma de identificar a los consumidores (por ejemplo, **`std::this_thread::get_id()`** o mediante un número) para saber quién ha consumido ya su valor. Y si utiliza alguna estructura de datos para almacenar valores, recuerde protegerla frente acceso concurrente con un mutex. Utilice como base el programa anterior.

Mejora: implemente un buffer compartido para que el productor pueda almacenar varios valores antes de que los consumidores los consuman. Utilice como base el programa anterior.

3.6. ENSAMBLADO

Se pide realizar un programa concurrente que simule el siguiente sistema de ensamblado. Una pieza está formada por tres partes: A, B, C. Hay tres trabajadores (hilos) alrededor de una mesa que montan la pieza (con suministro infinito), y un proveedor (hilo) que coloca sobre la mesa dos de las partes, seleccionadas de forma aleatoria (según una distribución uniforme), y tardando un tiempo aleatorio (según una distribución normal). En cuanto las dos partes están colocadas, el trabajador que tiene la que falta se pone a montar la pieza, tarea en la que tarda un tiempo aleatorio (elija distribución). Tras terminar la pieza, el trabajador la deja y la apunta en su cuenta para cobrarla al final del día. El proveedor coloca dos nuevas partes de forma aleatoria y el proceso comienza de nuevo. En un día se pueden hacer 30 piezas. Al final del día se cobra cada pieza a 10€. Calcule la ganancia de cada trabajador.

Mejora 1: en cuanto un trabajador se pone a montar una pieza, el proveedor coloca dos partes de nuevo sobre la mesa, con la esperanza de que alguno de los trabajadores que no está montando pueda comenzar inmediatamente, y de esta forma ahorrar tiempo.

Mejora 2: sobre la mejora 1, haga que el sueldo dependa también del tiempo total que se tarda en montar las 30 piezas. Cuanto menos tiempo, más dinero.

Mejora 3: sobre la mejora 1, haga que los trabajadores trabajen hasta que pase un determinado tiempo en lugar de hasta que monten 30 piezas. Calcule el sueldo al final de la simulación.