

Introducción a la Programación Orientada a Objetos con C++

(Para aplicaciones de control)

Ignacio Alvarez García
Ing. De Sistemas y Automática
Universidad de Oviedo
Septiembre 2023



INDICE

1.	Introducción a la programación orientada a objetos con C++	1
1.1	De C a C++	1
1.2	Clases y objetos	5
1.2.1	Clases y objetos en C++	8
1.2.2	Utilizando clases y objetos	10
1.2.3	Un ejemplo sencillo: la clase std::string.....	11
1.2.4	Otras clases estándar	14
1.2.5	Otras clases no estándar	15
1.3	Ejercicios propuestos	17
1.3.1	Ejercicio resuelto (o casi)	17
1.3.2	Ejercicio medio resuelto.....	19
2.	Usando clases disponibles: Qt-SDK	21
2.1	Clases genéricas Qt-SDK.....	21
2.1.1	QString, QStringList y QTextStream.....	21
2.1.1.1	QByteArray vs QString	21
2.1.2	QVector.....	21
3.	Crear clases propias	23
4.	Usando clases disponibles para control: Arduino	23
4.1	Programa básico en Arduino.....	23
4.2	Clases y funciones más utilizadas en Arduino.....	24
4.2.1	Serial.....	24
4.2.2	Temporizaciones.....	25
4.2.3	Interacción con el exterior a través de pines.....	27



1. Introducción a la programación orientada a objetos con C++

La programación orientada a objetos (**OOP** de aquí en adelante) permite trasladar a los programas la forma de pensar que tenemos para la solución o descripción de problemas en general. Normalmente nos referimos a entidades (objetos), que solemos clasificar en grupos con características y funcionamiento comunes (clases), y de las cuales nos interesan ciertas características (propiedades) y su funcionamiento (métodos).

Además, al relacionar unos objetos con otros para diseñar sistemas complejos, solemos utilizar la encapsulación, esto es, permitimos que los objetos interactúen entre sí sólo a través de las características y funcionalidades expuestas al exterior, mientras que las características y funcionalidades internas de cada uno son ignoradas por el resto.

Las características básicas de la programación orientada a objetos se describen con ejemplos sencillos en:

<https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/>

El lenguaje C++ es una extensión a C para soportar la programación orientada a objetos (OOP). Lo que sigue no es un manual de C++, sino una muy breve introducción para ser capaz de manejar los objetos, propiedades y métodos de las librerías de clases más habituales, o para crear clases nuevas. Para una documentación más exhaustiva se sugiere la lectura de:

- “Aprenda C++ como si estuviera en 1º”, de la Universidad de Navarra (http://caminos.udc.es/info/asignaturas/grado_itop/503/Bibliografia/TextosPDF/AprendaCPPComoSiEstuvieraEnPrimero-V2_JGarciaDeJalon.pdf).
- Tutorials point: C++: <https://www.tutorialspoint.com/cplusplus/index.htm>
- Tutorial paso a paso C++: <https://www.w3schools.com/cpp/default.asp>

Para escribir código en C++ se pondrá la extensión ‘.cpp’ a los archivos de código fuente.

1.1 De C a C++

Un ejemplo sencillo nos puede ayudar a comprender las ventajas de la transición desde C hasta C++.

Supongamos que queremos un programa que realice la tarea de manejar fechas, y poder calcular día de la semana, días entre dos fechas, etc.

Para cada fecha a gestionar, necesitaremos 3 variables, todas ellas enteras: year, month, day.

Si queremos saber la edad de alguien, podríamos hacer lo siguiente:

- Solicitar por teclado la fecha de nacimiento (día, mes, año)
- Solicitar por teclado la fecha actual (día, mes, año)
- Calcular el número de días entre ambas fechas. Dividir el resultado entre 365 (empezamos de forma fácil) para obtener los años.

Una primera aproximación requerirá:

- Variables para todos los datos necesarios: fecha actual y fecha de nacimiento (día, mes, año para cada una de ellas), nº de días transcurridos, nº de años (edad).
- Una función que, dadas dos fechas, obtenga el nº de días.

El programa (realizado con un cálculo grosero, no nos interesa ahora ese detalle) sería algo como:



main.c

```
#include "date_functions.h"

int main() {
    int today_year,today_month,today_day; // 3 enteros representan la fecha de hoy
    int birth_year,birth_month,birth_day; //3 enteros representan la fecha de nacimiento
    int day_count; // representa la diferencia de días entre ambas fechas
    float age_years; // diferencia de años entre ambas fechas

    pedir al usuario todos los datos para today_xxx y birth_xxx

    day_count=DiffDays(birth_year,birth_month,birth_day,today_year,today_month,today_day);
    age_years=day_count/365.0;
    printf("Edad: %f anyos\n",age_years);
    return 0;
}
```

date_functions.h

```
int DiffDays(int date1_year,int date1_month, int date1_day,
             int date2_year,int date2_month, int date2_day);
```

date_functions.c

```
#include "date_functions.h"

int DiffDays(int date1_year,int date1_month, int date1_day,
             int date2_year,int date2_month, int date2_day)
{
    int days;
    // cálculo muy grosero, habría que afinar para años bisiestos y días de cada mes
    days=(date2_year-date1_year)*365 + (date2_month-date1_month)*30 + (date2_day-date1_day);
    return days;
}
```

Como se observa en el ejemplo anterior, tenemos que manejar muchas variables para tratar con las fechas. Si además añadimos la hora, minuto y segundo, tendríamos un montón de variables más y, por ejemplo, una función DiffSeconds () necesitaría un montón de argumentos.

Sin salirnos del lenguaje C, éste ofrece la herramienta que facilita la labor de agrupar variables formando una sola declaración: la estructura (struct). **Seguimos necesitando todas las variables**, pero las agrupamos en struct para facilitar su gestión por el programa. El resultado queda más simple, sobre todo desde el punto de vista de main:

main.c

```
#include "date_functions.h"

int main() {
    struct Date today,birth; // Una struct contiene todas las variables de la fecha
    int day_count; // representa la diferencia de días entre ambas fechas
    float age_years; // diferencia de años entre ambas fechas

    pedir al usuario todos los datos para today.xxx y birth.xxx

    day_count=DiffDays(birth,today);
    age_years=day_count/365.0;
    printf("Edad: %f anyos\n",age_years);
    return 0;
}
```



date_functions.h

```
struct Date {  
    int year,month,day; // Todas las variables de tipo struct Date tienen los 3 valores  
};  
  
int DiffDays(struct Date date1,struct Date date2);
```

date_functions.c

```
#include "date_functions.h"  
  
int DiffDays(struct Date date1,struct Date date2)  
{  
    int days;  
    // cálculo muy grosero, habría que afinar para años bisiestos y días de cada mes  
    days=(date2.year-date1.year)*365 + (date2.month-date1.month)*30 + (date2.day-date1.day);  
    return days;  
}
```

Dado que ahora resulta más cómodo manejar fechas, podemos añadir con pocas complicaciones otras funciones: por ejemplo, calcular el día de la semana para una fecha (1:lunes, ..., 7:domingo), añadir días a una fecha para obtener otra, y con ello, por ejemplo, calcular en qué fecha y día de la semana terminará un trabajo si me dan 40 días de plazo:

main.c

```
#include "date_functions.h"  
  
int main() {  
    struct Date today,endOfTask;  
    int dayCount=40; // representa la diferencia de días entre ambas fechas  
    int weekDay;  
  
    pedir al usuario todos los datos para today.xxx  
  
    endOfTask =AddDays(today,dayCount);  
    weekDay=DayOfWeek(endOfTask);  
  
    usar weekday  
  
    return 0;  
}
```

date_functions.h

```
struct Date {  
    int year,month,day; // Todas las variables de tipo struct Date tienen los 3 valores  
};  
...  
struct Date AddDays(struct Date date1,int days_to_add);  
int DayOfWeek(struct Date date);
```



date_functions.c

```
#include "date_functions.h"

struct Date AddDays(struct Date date1,int days_to_add)
{
    struct Date end;
    end.day=date.day+days_to_add;
    // cálculo muy grosero
    while (end.day >= 30) {
        end.day -= 30;
        end.month += 1;
        if (end.month > 12) {
            end.month -= 12;
            end.year += 1;
        }
    }
    return end;
}

int DayOfWeek(struct Date date) {
    int wd;
    wd= ...cálculo según algoritmo de Zeller ...
    return wd;
}
```

Una vez realizada esta tarea, nos damos cuenta de que, realmente, toda fecha lleva aparejado su día de la semana, por lo que la variable `weekDay` podría estar incluida en la estructura, y así todas las fechas la tendrían.

date_functions.h

```
struct Date {
    int year,month,day; // Todas las variables de tipo struct Date tienen los 3 valores
    int weekDay; // Añadido
};
...
struct Date AddDays(struct Date date1,int days_to_add);
```

Así que, si `main()` hace algo como esto, el resultado es incorrecto:

main.c

```
#include "date_functions.h"

int main() {
    struct Date today,endOfTask;
    int dayCount=40; // representa la diferencia de días entre ambas fechas

    pedir al usuario todos los datos para today.xxx

    endOfTask =AddDays(today,dayCount);
    endOfTask.weekDay =DayOfWeek(today); // Error: debería ser DayOfWeek(endOfTask)
    endOfTask.weekDay =4; // Error: debería ser DayOfWeek(endOfTask)

    usar endOfTask.weekDay
    ...

    return 0;
}
```

El campo `weekDay` de cada variable tipo "struct Date" debería protegerse de accesos inadecuados para evitar problemas, ya que siempre debería ser consistente con la fecha. Cualquier olvido o error haría que ese campo fuese incorrecto. Para ello, la programación orientada a objeto nos ofrece nuevos elementos: las clases. Una clase C++ es como una estructura "ampliada", donde se declaran no sólo las variables sino también las funciones que las manipulan. Variables y funciones pueden ser públicas o privadas.

Dado que las funcionalidades `AddDays()` o `DayOfWeek()` siempre se aplican sobre una variable tipo fecha (ahora llamado objeto), el C++ nos permite incluir en la declaración de la estructura (ahora llamada clase) no sólo sus variables internas (campos) sino también las funciones que las manipulan (métodos). El resultado es un código más limpio y compacto.



Sin meternos en detalle, usando C++:

main.cpp

```
#include "date_functions.h"

int main() {
    Date today,endOfTask;
    int dayCount=40; // representa la diferencia de días entre ambas fechas
    char txt[12];

    pedir al usuario texto de fecha

    today.FromString(txt);
    endOfTask = today.AddDays(dayCount);

    usar endOfTask.GetDayOfWeek()

    return 0;
}
```

date_functions.h

```
class Date {
private:
    int dayOfWeek; // dayOfWeek no puede ser accedida más que desde la clase Date
public:
    int year,month,day;
    bool FromString(const char* txt);
    Date AddDays(int days_to_add) const;
    int GetDayOfWeek() const { return dayOfWeek; }
};
```

date_functions.cpp

```
#include "date_functions.h"

Date Date::AddDays(struct Date date1,int days_to_add)
{
    Date end;
    ...
    end.weekDay=DayOfWeek(); // Asegura que weekDay siempre es concordante con la fecha
    return end;
}

int Date::DayOfWeek() {
    int wd;
    wd= ...cálculo según algoritmo de Zeller ...
    return wd;
}
```

Yendo más allá, se debería proteger también (private) el contenido year, month, day, dado que no todas las combinaciones de valores enteros dan una fecha válida.

Si nos centramos en la función main(), apreciamos lo que conseguimos con C++ :

- Se simplifica el programa, parece más "directo": se usan objetos que se manipulan con funciones declaradas en sus clases.
- La complejidad se traslada a (queda encapsulada en) la(s) clase(s).
- La parte pública de la(s) clase(s) es la única que interesa al que las va a utilizar; la parte privada (variables y funciones) puede ser todo lo compleja que se necesite, pero el usuario de la clase no necesita los detalles.

1.2 Clases y objetos

El elemento fundamental de la OOP es la clase. Una clase es una definición de propiedades y métodos para un conjunto de entidades que comparten características comunes. La clase es un ente abstracto, simplemente consta de definiciones y operaciones que se pueden aplicar a los objetos de esa clase.



Una clase, en sí misma, sólo sirve para definir/organizar. Necesitamos elementos físicos para poder operar con ellos. Los elementos físicos que pertenecen a una clase son los objetos.

Las vacas son una clase: se definen por ser animales mamíferos, hembras, con 4 patas, vegetarianos, rumiantes, etc. etc. Entre otras cosas, las vacas pueden dar leche, para lo cual hay que hacer un procedimiento específico que es ordeñarlas. Pero no tenemos leche si no disponemos de objetos (instancias) de tipo vaca; la clase en sí misma define qué se puede hacer y cómo, pero para implementarlo necesitamos objetos de esa clase.

Además de lo anterior, la clase (y por tanto los objetos) tienen dos partes "claramente" diferenciadas: el interior (o parte privada), que debe existir pero no es accesible para otros objetos, y el exterior (interfaz, o parte pública), que sí es accesible para otros objetos. La vaca tiene un complejo sistema interno para producir la leche (parte privada), pero para ordeñarla sólo tenemos acceso a las ubres (parte pública).

Las anteriores definiciones pueden ser difíciles de asimilar, así que un ejemplo sencillo de la vida real para visualizar lo que es una clase y un objeto nos puede ayudar. Tomemos algún aparato de uso común, como por ejemplo una **TV**, un **microondas**, o un **botón pulsador**. Los 3 tipos de elementos son completamente diferentes en cuanto a las funcionalidades que nos ofrecen, independientemente de que puedan compartir algo de la tecnología interna.

Una TV sirve para ver programas emitidos remotamente. Tiene una determinada y compleja tecnología interna, pero como usuarios de la TV lo que nos interesa es su utilidad y el interfaz con el exterior: conexión de antena (para recibir las imágenes y el sonido), conexión de alimentación, pantalla (para ver las imágenes), altavoces (para escuchar el sonido), mando a distancia (para seleccionar), botones de pulsación, etc.

Múltiples fabricantes de TV tienen modelos distintos, pero todos ellos son para nosotros algo similar. No nos interesa cómo está hecha cada una para poder utilizarla.



Una TV (de forma genérica) es una **clase**. La clase define las características de los objetos de esa clase, su utilidad, y la forma de interactuar con ellos. La clase la forman una parte interna (la llamaremos **privada**, la ha desarrollado el fabricante; está/tiene que estar ahí para proveer la funcionalidad de TV, pero no nos interesa su detalle para usar la TV) y otra de interfaz (la llamaremos **pública**, es con lo que interactuamos y nos interesa como usuarios).

Tanto la parte pública como la privada contienen la declaración de dos tipos de componentes:

- Elementos físicos, necesarios para sostener la operatividad, que llamaremos **campos o propiedades o variables miembro**, pero que a su vez son objetos de clases específicas: públicos son el mando a distancia, los botones-pulsadores, la pantalla, etc.; privados son la fuente de alimentación, la electrónica para decodificar la señal de antena, etc.
- Estados o parámetros internos, que también llamaremos **campos o propiedades o variables miembro**: el número de canal seleccionado, el número máximo de canales disponibles, el estado encendido/apagado, serían algunos ejemplos.
- Procedimientos que gestionan la interacción entre elementos públicos y privados, que llamaremos **métodos o funciones miembro**: cuando se pulsa el botón [OFF] del mando a



distancia, el estado interno debe pasar de OFF a ON o viceversa, la fuente de alimentación se debe encender/apagar, la pantalla debe mostrar un mensaje, etc.; cuando se pulsa el botón [↓] del mando a distancia o del televisor, el número de canal se incrementa en 1 solo si el estado interno es ON y además no hemos rebasado el límite de canales; a continuación, se pasa el nuevo canal al decodificador para que muestre sus imágenes.

TV class definition:

Fields (physical elements and states):

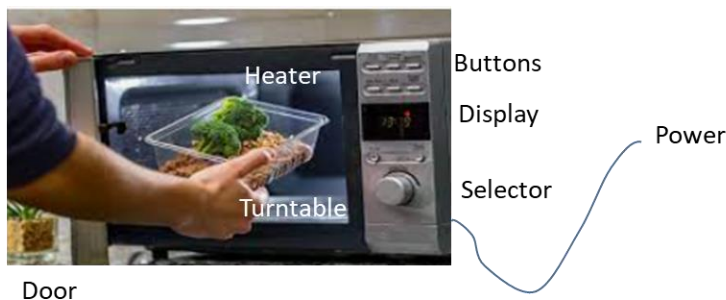
Channel number: integer
state On/OFF: boolean
Push button Up, Down, Enter: Push buttons
Remote: IR remote control

Methods (actions):

SwitchOnOff() →
if (state is ON) → state is OFF
else // state is OFF → state is ON
Pushbutton Down is pushed () →
if (state is ON) → increment channel
ShowImage() →
if (state is ON and channel is valid) →
decode channel from input data
show image in screen

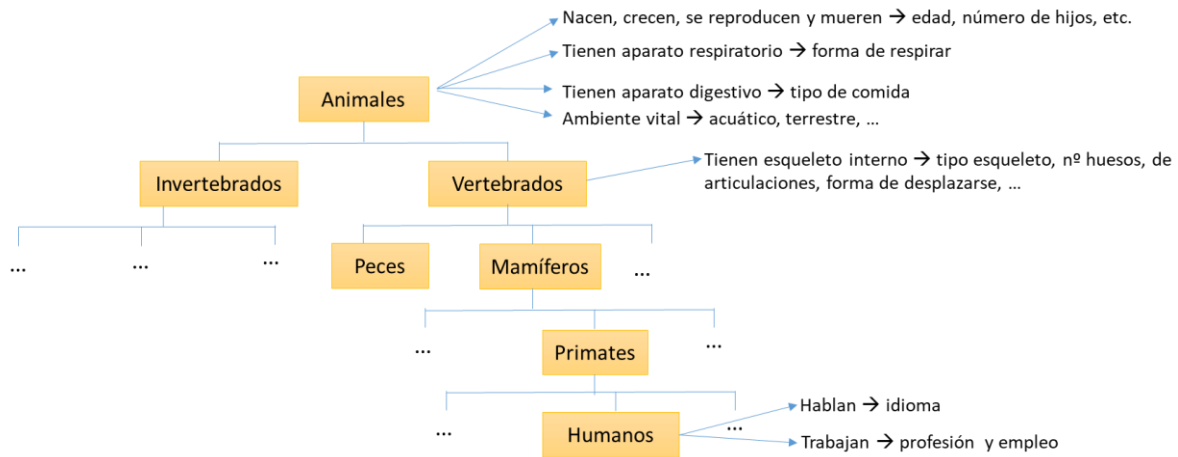
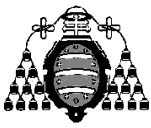
Pero para ver la TV no me sirve con la clase (la definición de sus campos y funciones miembro), necesito un objeto de esa clase: mi aparato de TV. Mi TV, la del vecino, la del bar, son objetos distintos (instancias) de la misma clase TV. Todos ellos tienen las mismas características y funcionalidades, pero su estado puede ser distinto (yo estoy viendo el canal A, y mi padre el canal B en otra TV; yo estoy pulsando un botón del mando en mi TV y el vecino no).

Un microondas, por su parte, es una clase con una utilidad diferente. Mediante unos botones y unas ruedecitas (potenciómetros) establecemos cuánto tiempo y con qué potencia queremos calentar algo. Hay muchos tipos de microondas, pero todos ellos pertenecen a la clase microondas, totalmente diferente de la clase TV. Eso sí, pueden compartir algunas cuestiones internas o de interfaz con la TV, pero su uso es distinto.



Nuevamente, la clase microondas define los elementos internos y de interfaz que permiten su funcionamiento, de los cuales como usuarios sólo estamos interesados en la parte pública: hay que enchufar, abrir la puerta, tocar ciertos botones, etc. La parte privada, interna, no nos interesa demasiado como usuarios, sólo le interesa al fabricante. Y, al igual que con la TV, necesitamos un objeto tipo microondas para poder calentar algo, y cada uno tendremos un objeto diferente, todos de la misma clase.

Por último, las clases se pueden definir de manera jerárquica, donde hay clases más generales (o abstractas) que definen campos y funcionalidades comunes, y otras más específicas (que se derivan y/o bifurcan) con campos y funcionalidades más específicos. Un ejemplo habitual lo tenemos en las clases de ciencias naturales, por ejemplo, para la clasificación de los animales:



El ser humano es un animal, vertebrado, mamífero, del orden de los primates.

Por ser animal, comparte muchas características y funcionalidades con otros animales; estas características y funcionalidades (alimentarse, reproducirse, morir, la edad, el sexo, etc.) se definen en la clase "animal", y se heredan y/o particularizan para los vertebrados, dentro de ellos para los mamíferos, etc.

Por ser vertebrado, comparte características y funcionalidades (tener esqueleto y articulaciones, moverse, etc.) con todos los vertebrados; éstas se definen en la clase "vertebrado" ya que no tienen sentido para todos los animales, y se heredan y/o particularizan para los mamíferos, ...

Según se va bajando en la jerarquía se añaden nuevas funcionalidades y propiedades, y/o se particularizan algunas. Por ejemplo, el idioma es una característica única de los humanos, pero moverse o tener piernas no.

Nuevamente, las clases sólo definen las propiedades y funcionalidades, los objetos son las instancias de la clase (tú, yo).

1.2.1 Clases y objetos en C++

Las clases y objetos en C++ "funcionan" en la programación como las clases y objetos de la vida real. Las clases agrupan definiciones comunes para un tipo de elementos, en este caso del programa, y los objetos son las instancias "físicas" de esos elementos.

Los programas deben declarar objetos de diferentes tipos, y hacerlos interactuar según unas determinadas reglas. Por ejemplo, para la gestión de un coche de juguete a control remoto necesitaríamos dos clases: ToyCar y RemoteControl. El programa declarará un objeto de cada clase, y hará que interactúen según la funcionalidad que deseemos.



De forma muy simplificada:

Las declaraciones de clases se realizan en archivos de cabecera (.h), que serán incluidos por otros archivos para obtener las definiciones. Es bastante común en C++ realizar un archivo de cabecera por cada clase, con el mismo nombre que ésta.



ToyCar.h

```
#ifndef _INC_TOYCAR_H // Esta línea, junto a la siguiente y la última, ...
#define _INC_TOYCAR_H // ... evitan la inclusión múltiple del mismo .h

#include "Motor.h" // Incluye otros archivos de cabecera con definiciones

class ToyCar
{
private: // Elementos necesarios internamente, pero no accesibles desde "el exterior"
// Campos miembro que determinan el estado y/o representan a partes de la clase
// (los definiremos más adelante)
;

public: // Elementos de interfaz con "el exterior"
// Funciones miembro que realizan determinadas operaciones
void GoForward(float speed_m_s);
void Stop( );
void TurnLeft(float degs,float degs_s);
float GetRemainingPowerTime_min() const;
}; ... // ¡¡¡ Atención !!!: terminar con punto y coma tras el }
#endif // _INC_TOYCAR_H
```

RemoteControl.h

```
#ifndef _INC_REMOTECONTROL_H // Esta línea, junto a la siguiente y la última, ...
#define _INC_REMOTECONTROL_H // ... evitan la inclusión múltiple del mismo .h

#include "Button.h" // Incluye otros archivos de cabecera con definiciones

// Valores devueltos por la función GetUserAction() de la clase RemoteControl
#define I_USER_ACTION_NONE -1
#define I_USER_ACTION_MOVE_FORWARD 1
#define I_USER_ACTION_TURN_LEFT 2
...

class RemoteControl
{
private: // Elementos necesarios internamente, pero no accesibles desde "el exterior"
// Campos miembro que determinan el estado y/o representan a partes de la clase
// (los definiremos más adelante)
;

public: // Elementos de interfaz con "el exterior"
// Funciones miembro que realizan determinadas operaciones
int GetUserAction();
}; // ¡¡¡ Atención !!!: terminar con punto y coma tras el }
#endif // _INC_REMOTECONTROL_H
```

En el programa principal se declaran objetos (variables) de esas clases, y se utiliza su parte pública:

main.cpp

```
#include "ToyCar.h"
#include "RemoteControl.h"

int main()
{
    ToyCar myCar; // Crea objeto myCar de tipo ToyCar
    RemoteControl myRemote; // Crea objeto myRemote de tipo RemoteControl

    while (myCar.GetRemainingPowerTime_min() > 5 )
    {
        int buttonPressed=myRemote.GetButtonPressed();
        if (buttonPressed==I_BUTTON_MOVE_FORWARD)
            myCar.GoForward(0.50);
        if (buttonPressed==I_BUTTON_TURN_LEFT)
            myCar.TurnLeft(2.0,1.0);
    }
    myCar.Stop();
    ...
}
```



En el ejemplo anterior se muestra únicamente la parte pública de las clases, y no la parte privada ni la implementación interna. El programa principal (función `main`) declara un objeto de cada clase, y utiliza su parte pública para realizar las interacciones requeridas para el funcionamiento deseado.

En el ejemplo anterior ya destacan algunas de las características de la OOP, que la hacen tan interesante. Fijémonos en `main` ():

- El programador de `main` () no necesita conocer los detalles de cómo funcionan internamente las clases que necesita. Simplemente declara objetos y utiliza su parte pública para que interactúen y conseguir su cometido. Por ejemplo, el cochecito podría estar realizado con tecnologías diversas (gasolina/eléctrico, giro por velocidad diferencial de dos ruedas motrices/giro mediante sistema de dirección de Ackermann, etc.), pero los detalles no son necesarios para `main` (). De la misma forma, el mando remoto podría estar compuesto por botones, joystick, acelerómetros, ... , pero de él sólo interesa si se está ordenando giro a izquierda o derecha, acelerar, etc.
- Los desarrolladores de las clases `ToyCar` y `RemoteControl` pueden tomar las decisiones que deseen sobre su implementación, siempre que mantengan las declaraciones de la parte pública.
- El desarrollo de las diferentes clases y del programa principal puede realizarse en paralelo, por diferentes desarrolladores, simplemente acordando al inicio la parte pública.
- Las clases son altamente reutilizables en otros programas.

1.2.2 Utilizando clases y objetos

Aunque podríamos ahora diseccionar el contenido de una clase, cómo se declaran sus componentes, etc., vamos a dejarlo de momento para ir por el camino más sencillo.

Al igual que con la TV o el microondas no se nos ocurre como primera opción diseñar y construir uno completo para poder utilizarlo, o para conocer su funcionamiento, sino que lo más cómodo inicialmente es buscar uno con las características deseadas y comenzar a manejarlo, así haremos con las clases y objetos C++. Solamente si no existe una clase de objetos con las características que queremos hacemos el esfuerzo de desarrollarlos, pero eso requiere más conocimientos que su simple uso.

Nuestra secuencia de decisión:

- Detectamos una necesidad: necesito calentar la comida.
- Buscamos alternativas que cumplan esa necesidad. Nuestra propia experiencia, las búsquedas en Internet, la lectura de libros, o las consultas a expertos nos ayudan a encontrar y comparar alternativas.
- Miramos características y seleccionamos la clase que mejor se adecúa a nuestra necesidad (podemos calentar la comida en la cocina de gas, en una hoguera, poniendo al sol, etc. pero el microondas satisface mejor nuestra necesidad).
- Obtenemos un objeto (comprado, regalado, prestado), o varios si son necesarios, de esa clase. Utilizamos ese(os) objeto(s) para nuestra necesidad, de acuerdo con las instrucciones operativas definidas por quien ha desarrollado la clase y fabricado el objeto.
- Si no existiese una clase que satisfaga completamente nuestras necesidades (necesitamos un microondas que abra la puerta de forma automática cuando el usuario silbe una determinada canción), tendremos que desarrollarla (desde cero, o mejor modificando/ampliando alguna existente) o pedir que la desarrollen para nosotros.

En programación (C++) muchos desarrolladores han realizado ya clases, y de la misma forma seleccionamos alguna si nos resulta de utilidad, y obtenemos objetos de la misma para nuestras



necesidades. Sólo pasaremos a conocer los detalles de las clases y objetos según nos sean necesarios, y desarrollaremos nuestras propias clases cuando estemos cómodos con las ya existentes.

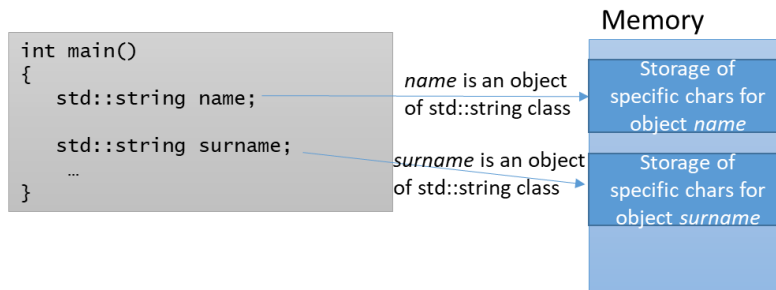
1.2.3 Un ejemplo sencillo: la clase `std::string`

Una clase sencilla ya existente que se puede utilizar rápidamente en programación se llama `std::string`.

Esta clase contiene (casi) todo lo necesario para manejar cadenas de caracteres (texto). Al igual que para la TV o el microondas, hay un montón de cuestiones internas que se necesitan a la hora de manipular texto en un programa. El texto está formado por tablas de caracteres, cuya longitud es muy variable, y para cuyo manejo se requiere asignación y liberación de memoria, bucles para asignación de cada carácter, búsqueda, comparación, pasar a mayúsculas o minúsculas, etc.

Cada objeto tipo `std::string` tiene las mismas funcionalidades potenciales, pero un contenido distinto. Mi nombre y mi apellido serían dos objetos de la misma clase `std::string` con contenidos diferentes.

`std::string` class → requires storage capacity for a varying length sequence of characters
→ requires functionalities to manipulate the contents (add, split, compare, get length, convert to uppercase, etc.)



El creador de la clase `std::string` ha definido una parte privada, inaccesible (como el interior del microondas), que contiene los datos y la funcionalidad internos, y una parte pública que es la que nos interesa como usuarios de esa clase. Sólo tendremos documentación de la parte pública, en este caso la podemos encontrar en <https://cplusplus.com/reference/string/string>.

A modo de ejemplo rápido, sin apenas conocer C++ ni estudiar detalladamente la documentación, podemos utilizarla para extraer el campo *nombre* de una secuencia de caracteres con el formato *apellidos, nombre*.

Al igual que ocurre con la TV o el microondas, los diseñadores de la clase (`std::string` o cualquier otra) han intentado que los objetos de la misma dispongan de toda la funcionalidad pública necesaria de la manera más sencilla posible. En la TV o el microondas se consigue con botones que indican los números, flechas arriba y abajo, etc., fáciles de comprender por cualquiera. En programación, el desarrollador de la clase inserta campos y funcionalidades miembro con nombres "sencillos" que aluden a lo que hacen. La mayoría de los desarrolladores no deja acceso directo a los campos internos, sino a través de funciones.

El **acceso a los elementos públicos se hace a nivel de objeto**, mediante el operador punto aplicado al objeto; para las funciones públicas, se pasarán argumentos y se obtendrán resultados según el contenido de dicho objeto.



ej1_step1_oop.cpp

```
#include <string> // Permite usar la clase std::string, definida en ese archivo
                // de cabecera
int main()
{
    std::string datos; // Objeto de la clase std::string (variable)
    int pos_coma; // Número entero
    std::string nombre; // Objeto de la clase std::string (otra variable, misma clase
                        // que datos, pero distintos contenidos)

    datos="Pi,Filemon"; // Asigna valor al objeto datos (operador público asignación)
    pos_coma =datos.find(","); // Busca coma (función pública find aplicada
                              // al objeto datos)
    if (pos_coma>=0) { // Si ha encontrado la coma
        nombre=datos.substr(pos_coma); // Asigna valor al objeto nombre (operador
                                       // público asignación) a partir del resultado
                                       // de la función pública subcadena (substr)
                                       // aplicada al objeto datos.
    }
    return 0;
}
```

Como es habitual con cualquier lenguaje de programación, las pruebas de los primeros programas se facilitan mucho con E/S por consola (pantalla de texto y teclado). C++ también tiene clases y objetos para esto.

En los ejemplos que siguen, para evitar duplicar todo el texto, resaltamos en negrita las partes que cambian o se añaden, y evitamos repetir el resto utilizando los puntos suspensivos.

El ejemplo 1, paso 2, muestra cómo escribir resultados en pantalla:

ej1_step2_oop.cpp

```
#include <string> // Permite ...
#include <iostream> // Permite usar las clases std::ostream y std::istream, y los
                  // objetos globales std::cout (clase std::ostream) y
                  // std::cin (clase std::istream)

int main()
{
    std::string datos;
    if (pos_coma>=0) {
        nombre=datos.substr(pos_coma+1); // ...

        std::cout << "El nombre es: " << nombre << "\n"; // Llama repetidamente al
                                                         // operador << del objeto
                                                         // std::cout para que escriba
                                                         // en consola con formato
    }
    else {
        std::cout << "Falta la coma en los datos\n";
    }
    std::cout.flush(); // Llama a la función flush() del objeto std::cout para
                       // asegurar que se efectúa ahora la operación de escritura

    return 0;
}
```

El paso 3 muestra como esperar texto por teclado y utilizarlo en el programa.



ej1_step3_oop.cpp

```
#include <string> // Permite ...
#include <iostream> // ...

int main()
{
    std::string datos;
    ...

    datos="Pi,Filemon";
    std::cout << "Introduce apellido, nombre: "; // Escribe en consola
    std::cin >> datos; // Llama al operador >> del objeto std::cin para que espere
    // texto por teclado y se lo asigne al objeto datos
    // Nota importante. Si el texto a leer pueden tener espacios en blanco,
    // sustituir la línea anterior por la siguiente (quitando el comentario):
    // std::getline(std::cin,datos);

    pos_coma =datos.find(","); // ...
    ...

    return 0;
}
```

Una vez vista la funcionalidad básica, veamos el detalle de la documentación de una de las funciones disponible en Internet, por ejemplo el método `find()` de la clase `std::string` (<https://cplusplus.com/reference/string/string/find/>)

Standard aplicable

Función pública, se aplica sobre un objeto de la clase

public member function

std::string::find

C++98 [C++11]

Hay 4 formas de invocar a la función (polimorfismo)

Primera forma:

Devuelve un valor de tipo `size_t` (entero)

La función no modificará los contenidos del objeto para el que se invoca

Requiere dos argumentos:

- El 1º es una referencia a un objeto de tipo `string`
- El 2º es un entero (opcional, porque tiene un valor por defecto que se usará si no se pasa argumento)

Ejemplos primera forma:

```
std::string datos;
std::string coma=",";
int pos_coma;

pos_coma=datos.find(coma,0); // Busca 1ª aparición de la coma ','
// empezando desde el índice 0
pos_coma=datos.find(coma); // Id. al anterior (2º arg por defecto = 0)
pos_coma=datos.find(coma,pos_coma+1); // Busca siguiente aparición de la coma ','
// ya que empieza desde el índice siguiente
// al último encontrado
```

En la documentación de esta función se observa:

- Hay dos posibles estándares aplicables: C++98 y C++11. 98 y 11 son años (1998 y 2011). El compilador que utilizemos puede adherirse a uno o varios estándares (98, 11, 14, 17, 20 y 23). Un estándar posterior acoge todo lo anterior (a veces hay partes eliminadas - deprecated) y añade nuevas funcionalidades.
- La función es pública: se puede invocar para cualquier objeto de esa clase desde cualquier parte del código, utilizando el operador punto.
- Hay 4 posibles invocaciones de la función (polimorfismo). En este caso, todas ellas devuelven un valor de tipo entero (`size_t`), pero admiten distintos formatos de argumentos; el compilador comprobará automáticamente cuál de ellas se está invocando a partir de los argumentos. Esta función, en sus 4 formas, es de tipo **const**, lo que indica que cuando se ejecute no se alterará el contenido del objeto sobre el que se aplica (el `string` sigue conteniendo el mismo texto al finalizar la función).
- Centrándonos en la primera forma polimórfica, requiere dos argumentos:
 - o Un `std::string` de nombre `str`, que es la cadena a buscar. Puesto que es un objeto, se suele acompañar de `const` y `&` para pasarlo como referencia constante, lo que evita las operaciones de construcción y destrucción (se verán luego).



- Un entero tipo `size_t` de nombre *pos*, que puede tener un valor por defecto (cero en este caso). Indica que este 2º argumento es opcional, y si el llamador no lo incluye, dentro de la función tomará el valor cero.

1.2.4 Otras clases estándar

Además de `std::string`, que nos ayuda a manipular texto (cadenas de caracteres), o `std::ostream`, que nos ayuda a gestionar salida hacia dispositivos, el estándar C++ dispone de otras muchas clases ya realizadas y que pueden ser utilizadas por el desarrollador. Se pueden consultar en <https://en.cppreference.com/w/cpp/header>.

Un ejemplo muy clásico (para personas con algún conocimiento en matemáticas avanzadas) es la clase `std::complex`. El computador sabe cómo tratar con números enteros (`int` y otros) o reales (`float` y otros) para realizar cálculos, pero en ocasiones necesitamos manejar complejos. Un número complejo no es más que una pareja de números reales, uno de los cuales representa la parte real y otro la imaginaria, y en el cual las operaciones se hacen de una manera especial:

$$x = a + b*j \quad (\text{donde } j = \sqrt{-1} \text{ es la unidad imaginaria})$$
$$y = c + d*j$$

Por definición:

$$z = x*y = (a*c - b*d) + (a*d + b*c)*j$$

La clase `std::complex` permite manejar objetos de tipo complejo para nuestros cálculos con sencillez, gestionando el almacenamiento de la parte real e imaginaria y proporcionando funciones y operadores para sumar, multiplicar, obtener módulo y argumento, exponenciar, ...

La clase `std::complex`, a diferencia de `std::string`, es una clase con plantilla (templated). Esto quiere decir que se añade a la clase `<tipo_de_datos>` para que pueda operar con diferentes datos internos: podemos hacer objetos de tipo `std::complex<float>` o `std::complex<double>` según la precisión que necesitemos.

En el ejemplo siguiente se declaran tres objetos de tipo complejo (*x,y,z*), utilizando la precisión `float`. *x* se inicializa a $5+2*j$, *y* se inicializa a $7-4*j$, *z* no se inicializa. Una función miembro especial en la clase `std::complex`, llamada constructor, permite la inicialización con valores específicos (la veremos más adelante).

ej2_step1_oop.cpp

```
#include <string>
#include <iostream>
#include <complex>

int main()
{
    std::complex<float> x(5,2.0), y(7,-4.0), z;

    z=x*y; // Usa el operador producto definido en la clase std::complex
    std::cout << "z = " << z << "\n"; // Escribe en consola el valor de z
    return 0;
}
```

Otra clase muy utilizada también basada en plantillas (templated) es `std::vector<tipo_de_datos>`. Esta clase permite crear objetos que sean vectores (tablas) de cualquier otro tipo de objeto, incluidos los tipos básicos de datos. Por ejemplo, podemos crear un vector de `float` para almacenar datos reales con



la lluvia de cada día la última semana, y utilizar sus funcionalidades para desplazar la tabla y actualizarla cada vez que tengamos un nuevo dato:

ej3_step1_oop.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<float> rain_week; // Vector de números reales (inicialmente, tamaño 0)
    rain_week.reserve(7); // Opcional: tamaño máximo del vector, reserva memoria

    while (1) // Bucle sin fin (1 es distinto de 0, por tanto siempre verdadero)
    {
        float rain_today; // Dato de hoy
        std::cout << "Introduce lluvia de hoy: "; // Escribe en consola
        std::cin >> rain_today; // Pide por teclado
        if (rain_week.size() >= 7) { // Si hay más de 7 datos (más de 1 semana) ...
            rain_week.pop_back(); // ... quito el último que ya no necesito
            rain_week.insert(rain_week.begin(), rain_today); // Añade nuevo dato al ppio del vector
        }
        std::cout << "Lluvia de los últimos 7 días: ";
        for (int i=0; i<rain_week.size(); i++) { // Se necesita un bucle para escribir cada valor
            std::cout << rain_week[i] << " "; // Con el operador [] se accede a cada elemento
        } // del vector como un array "normal"

        // La función std::max_element() devuelve un iterador que nos permite acceder al element
        // de la table con el valor máximo. El iterador es "como un puntero", por lo que se usa
        // * para acceder a su contenido
        std::cout << " -> Max=" << *std::max_element(rain_week.begin(), rain_week.end()) << "\n";
        std::cout.flush();
    }

    return 0;
}
```

El resultado del programa (en amarillo los datos introducidos por teclado):

```
Introduce lluvia de hoy: 7.5↵
Lluvia de los últimos 7 días: 7.5 --> Maximo = 7.5
Introduce lluvia de hoy: 0↵
Lluvia de los últimos 7 días: 0 7.5 --> Maximo = 7.5
Introduce lluvia de hoy: 2↵
Lluvia de los últimos 7 días: 2 0 7.5 --> Maximo = 7.5
Introduce lluvia de hoy: 9↵
Lluvia de los últimos 7 días: 9 2 0 7.5 --> Maximo = 9
Introduce lluvia de hoy: 8↵
Lluvia de los últimos 7 días: 8 9 2 0 7.5 --> Maximo = 9
Introduce lluvia de hoy: 3↵
Lluvia de los últimos 7 días: 3 8 9 2 0 7.5 --> Maximo = 9
Introduce lluvia de hoy: 6↵
Lluvia de los últimos 7 días: 6 3 8 9 2 0 7.5 --> Maximo = 9
Introduce lluvia de hoy: 2.2↵
Lluvia de los últimos 7 días: 2.2 6 3 8 9 2 0 --> Maximo = 9
Introduce lluvia de hoy: 0.8↵
Lluvia de los últimos 7 días: 0.8 2.2 6 3 8 9 2 --> Maximo = 9
Introduce lluvia de hoy: 5↵
Lluvia de los últimos 7 días: 5 0.8 2.2 6 3 8 9 --> Maximo = 9
Introduce lluvia de hoy:
```

1.2.5 Otras clases no estándar

El estándar C++ provee un buen número de clases y funciones auxiliares, pero sólo las más genéricas: le faltan un buen montón de utilidades. Otros desarrolladores han generado librerías de clases que se pueden utilizar en nuestros programas (de forma gratuita o como establezca el acuerdo de licencia). Como no forman parte del estándar, tendremos que descargar y en su caso instalarlas antes de poder utilizarlas, y comprobar si están disponibles para el equipo destino.

Ejemplos de librerías genéricas usuales son:



- boost (https://www.boost.org/doc/libs/1_82_0/): montones de clases para casi todo, gestión de texto, tiempo, imágenes, archivos y directorios, cálculos matemáticos con vectores, matrices, etc., cálculos geométricos, bases de datos, etc. Algunas se van incorporando al estándar C++ por su calidad y popularidad.
- Qt-SDK (<https://doc.qt.io/qt-6/reference-overview.html>) : montones de clases para casi todo, como el anterior, pero con una orientación especial a programas que deben responder a eventos, con interfaz gráfico, comunicaciones, etc.
- dlib (<https://dlib.net>) : clases muy variadas, incluyendo para machine-learning
- POCO (<https://pocoproject.org/>): menos clases, más orientadas a equipos pequeños (IOT).
- Arduino libraries (<https://www.arduino.cc/reference/en/libraries/>): específicas para Arduino y compatibles.

Ejemplos de librerías específicas para un tipo de problema determinado son:

- opencv (<https://opencv.org>): visión artificial
- eigen (<https://eigen.tuxfamily.org/>): álgebra lineal (matrices, etc.)
- Openssl : criptografía
- ...

Un listado de las librerías C++ más usuales en : <https://cpp.libhunt.com/projects>



1.3 Ejercicios propuestos

1.3.1 Ejercicio resuelto (o casi)

Realizar un programa que, dado un texto, lo divida en palabras (separadas por espacio) y cuente el número de ellas.

Solución:

- Necesitamos un objeto tipo `std::string` para el texto de entrada (*text*). Solicitaremos su contenido al usuario a través del teclado, para lo que usamos los objetos `std::cout` y `std::cin`.
- Necesitamos un objeto tipo `std::vector<std::string>` (*words*) para tener un vector de elementos, cada uno de los cuales será una de las palabras. Inicialmente estará vacío.
- Necesitamos un entero para contar palabras (*count_words*). Bueno, realmente este entero es prescindible ya que el número de palabras se obtiene en cualquier momento de la longitud del vector *words*.
- Necesitamos valores enteros auxiliares para ir buscando los espacios en blanco: *cur_pos*, *next_pos*
- Tras solicitar el texto, hay que realizar un bucle en que vamos buscando el siguiente espacio en blanco en el texto original. Cuando localizamos el siguiente espacio, tenemos una nueva palabra que añadimos al vector, y saltamos todos los espacios consecutivos a ese porque no contendrían nuevas palabras.

En forma de pseudo-código (lógica básica de lo que pretendemos, sin todas las restricciones del lenguaje formal):

```
Pedir text por teclado
Desde cur_pos = 0 y mientras cur_pos es válido {
    next_pos = posición del siguiente espacio en blanco de text (a partir de
cur_pos)
    Si next_pos es válido (encontró un nuevo espacio) {
        Hay una nueva palabra que empieza en cur_pos y termina en next_pos →
añadir al vector words e incrementar contador count_words
        Incrementar next_pos mientras el carácter que está en next_pos sea el
espacio
        (para no tener en cuenta varios espacios consecutivos)
        cur_pos=next_pos (para la siguiente iteración del bucle)
    }
}
Terminado: escribir resultados en pantalla
```



propuesto_1.cpp

```
#include <string>
#include <iostream>
#include <vector>

int main()
{
    std::string text;
    std::vector<std::string> words;

    std::cout << "Introduzca texto con espacios: ";
    std::getline(std::cin, text);

    int count_words;
    int cur_pos, next_pos;
    for (cur_pos=0, count_words=0; cur_pos<=text.length(); cur_pos=next_pos)
    {
        next_pos=text.find(' ', cur_pos);
        if (next_pos>=0)
        {
            words.push_back(text.substr(cur_pos, next_pos-cur_pos));
            while (text[next_pos]==' ') {
                next_pos++;
            }
            count_words++;
        }
    }

    std::cout << "He encontrado " << count_words << " palabras\n";
    // count_words se podría haber sustituido por words.size()
    std::cout << "Las palabras son:\n";
    for (int i=0; i<words.size(); i++) {
        std::cout << i+1 << " -> " << words[i] << "\n";
    }

    std::cout.flush();
    return 0;
}
```

El resultado obtenido es:

```
Introduzca texto con espacios: Esto es una prueba para contar palabras.
He encontrado 6 palabras
Las palabras son:
1 -> Esto
2 -> es
3 -> una
4 -> prueba
5 -> para
6 -> contar
```

Mejoras propuestas:

- Como se ve, el resultado anterior no es del todo correcto (falta la última palabra). Intente corregir el programa para solucionarlo.
- Dividir un texto en palabras a partir del espacio en blanco es algo que puede ser interesante en múltiples situaciones, y sólo depende de un valor de entrada para producir los resultados. Estaría mejor en una función y no en main(). Modificar el código para hacerlo:

main.c

```
#include <iostream>
#include "MyStringFns.h"

int main()
{
    std::string text;
    std::vector<std::string> words;

    std::cout << "Introd txt: ";
    std::getline(std::cin, text);

    words=FnDividewords(text, ' ');
    std::cout << "Hay" << words.size() << " palabras\n";
    std::cout << "Las palabras son:\n";
    for (int i=0; i<words.size(); i++) {
        std::cout << i+1 << "->" << words[i] << "\n";
    }

    std::cout.flush();
}
```

MyStringFns.h

```
#ifndef _INC_MYSTRING_FNS
#define _INC_MYSTRING_FNS

#include <string>
#include <vector>

std::vector<std::string> FnDividewords(const
std::string& text, char divider=' ');

#endif // _INC_MYSTRING_FNS
```

MyStringFns.cpp

```
#include "MyStringFns.h"
std::vector<std::string> FnDividewords(const
std::string& text, char divider)
{
    std::vector<std::string> piezas;
```



```
return 0;
}
```

```
int cur_pos,next_pos;
for (...) {
    Mismo que antes, pero ahora usando pieces
    en lugar de text
}
return pieces;
}
```

1.3.2 Ejercicio medio resuelto

Un texto en formato xml sirve para definir contenidos categorizados. Está compuesto por el texto deseado empotrado entre un tag de comienzo con la forma <tag> y otro de cierre con la forma </tag>, donde tag es el nombre del elemento que se quiere almacenar. En el texto puede haber otros tag, dando lugar a una estructura jerárquica en forma de árbol. Se utiliza mucho en la comunicación de informaciones entre ordenadores; por ejemplo las páginas html tienen este formato (https://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic_document):

El servidor entrega este texto

→ El navegador lo interpreta y dibuja formateado

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

My First Heading

My first paragraph.

Se desea un programa que lea un texto xml de un archivo de texto, y extraiga del mismo el tag deseado. Para el caso anterior, suponiendo que el texto xml se encuentra en el archivo myPage.html, se busca un resultado como:

```
Introduzca nombre de archivo: myPage.html ↵
Introduzca tag a extraer: body ↵
El contenido de <body> es:

<h1>My First Heading</h1>

<p>My first paragraph.</p>

Introduzca subtag a extraer de body: p ↵
El contenido de <p> es: My first paragraph.
```

Solución:

- Necesitamos un objeto tipo `std::string` para el nombre del archivo (*filename*). Solicitaremos su contenido al usuario a través del teclado, para lo que usamos los objetos `std::cout` y `std::cin`.
- Necesitamos un objeto tipo `std::ifstream` (*file*) para leer el contenido del archivo. Para leer todo el texto de un `std::ifstream`, ver <https://www.tutorialspoint.com/what-is-the-best-way-to-read-an-entire-file-into-a-std-string-in-cplusplus> (ojo, en este ejemplo han eliminado la necesidad de indicar explícitamente `std::` delante de `string`, `ifstream` y `ostream` mediante la línea que comienza por `using`; particularmente no lo recomiendo, pero es posible).
- Necesitamos varios objetos tipo `std::string` para almacenar: el tag que se busca, el texto que se ha encontrado.
- Deberíamos realizar una función, ya que se utilizará el mismo código de forma similar varias veces, y será utilizable en otros casos:

```
std::string GetXmlTagContents(const std::string& all_xml_text,const std::string& tag)
{
    std::string tag_contents;
    ...
    return tag_contents;
}
```



- ```
}
```
- La función debe, por este orden:
    - Componer un texto con el tag de comienzo, y otro con el de final; se puede hacer con el operador + que aplicado a dos `std::string` devuelve un `std::string` que los concatena:

```
std::string open_tag("<"+tag+">");
std::string close_tag("</"+tag+">");
```
    - Buscar el tag de comienzo en el texto, y el de final a partir del de comienzo (usar `find`). Con ambos ya se puede usar la función `substr( )` aplicada al objeto `all_xml_text` para obtener el contenido intermedio.



## 2. Usando clases disponibles: Qt-SDK

Si se programa en entorno Qt, tenemos una gran cantidad de clases disponibles para la mayoría de necesidades, bajo el nombre común Qt-SDK, que incluye:

- Clases genéricas, similares a las de std pero mucho más completas: cadenas de caracteres (QString, QByteArray, QStringList), vectores (QVector), fecha y hora (QDate, QTime, QDateTime), archivos y directorios (QFile, QFileInfo, QDir), geometría 2D (QPoint, QPointF, QLine, QLineF, QRect, QRectF, QSize, QSizeF), etc.
- Clases auxiliares específicas más completas: QStringTokenizer, QDomStreamReader, QDomDocument, QUrl, ...
- Clases para programación orientada a eventos, derivadas de QObject y utilizando el paradigma de signals y slots.
- Clases para entorno gráfico de usuario (gui y widgets): ventanas, cuadros de diálogo, botones, bitmaps (QWidget, QDialog, QPushButton, QPixmap, ...)
- Clases para comunicaciones: QSerialPort, QTcpSocket, QTcpServer, QWebSocket, ...
- etc.

### 2.1 Clases genéricas Qt-SDK

Las clases genéricas Qt-SDK (o Qt-API) son más completas que sus equivalentes std. A modo de ejemplo:

#### 2.1.1 QString, QStringList y QTextStream

QString es una alternativa a std::string, pero con muchas más opciones. QTextStream es también mejor que std::cin y std::cout. Por ejemplo, el ejercicio resuelto 1.3.1 se haría mucho más fácil con QString (función split), QStringList (vector de strings) y QTextStream:

propuesto\_1\_revisitedQt.cpp

```
#include <QString>
#include <QStringList>
#include <QTextStream>

int main()
{
 QString text;
 QStringList words;
 QTextStream qtConsoleIn(stdin),qtConsoleOut(stdout);

 qtConsoleOut << "Introduzca texto con espacios: "; qtConsoleOut.flush();
 text=qtConsoleIn.readLine();
 words=text.split(" ", Qt::SkipEmptyParts);

 qtConsoleOut << "He encontrado " << words.count() << " palabras\n";
 for (int i=0;i<words.count();i++)
 qtConsoleOut << i+1 << " -> " << words[i] << "\n";

 qtConsoleOut.flush();
 return 0;
}
```

##### 2.1.1.1 QByteArray vs QString

##### 2.1.2 QVector

QVector es una alternativa a std::string, pero con muchas más opciones. También utiliza templates. Revisando el ejemplo 3, step 1, podemos usar las funciones prepend() o removeLast():



ej3\_step1\_oop.cpp

```
#include <QTextStream>
#include <QVector>

int main()
{
 QVector<float> rain_week; // Vector de números reales (inicialmente, tamaño 0)
 QTextStream qtConsoleIn(stdin),qtConsoleOut(stdout);

 rain_week.reserve(7);
 while (1)
 {
 float rain_today; // Dato de hoy
 qtConsoleOut << "Introduce lluvia de hoy: "; qtConsoleOut.flush();
 qtConsoleIn >> rain_today; // Pide por teclado
 if (rain_week.size() >= 7) { // Si hay más de 7 datos (más de 1 semana) ...
 rain_week.removeLast(); // ... quito el ultimo que ya no necesito
 rain_week.prepend(rain_today); // Añade nuevo dato al ppio del vector
 }
 qtConsoleOut << "Lluvia de los ultimos 7 dias: ";
 for (int i=0;i<rain_week.size();i++) {
 qtConsoleOut << rain_week[i] << " ";
 }

 qtConsoleOut << "\nMax=" << *std::max_element(rain_week.begin(),rain_week.end()) << "\n";
 qtConsoleOut.flush();
 }

 return 0;
}
```





### 3. Crear clases propias

El programador puede crear nuevas clases que se acomoden a sus intereses, si no encuentra ninguna que le satisfaga.

### 4. Usando clases disponibles para control: Arduino

Existen numerosas clases disponibles para nuestros programas de control. Un entorno donde es fácil obtener y utilizar algunas de ellas es Arduino IDE (<https://www.arduino.cc/reference/en/libraries/>), por lo que nos vamos a centrar en él para los primeros pasos, aunque lamentablemente estas librerías tienen una documentación muy pobre y la mayoría están pensadas para desarrolladores aficionados y no profesionales.

Para programar en entorno Arduino, es necesario (o más bien aconsejable) el entorno [Arduino IDE](#) y una tarjeta compatible (Arduino Uno, otras versiones basadas en Arduino, ESP32, etc.), además de los sensores y accionamientos necesarios para intercambiar informaciones con el mundo real. Existe una herramienta de simulación sencilla para manejar de forma básica la programación en este entorno, sin necesidad de los elementos físicos, que es [TinkerCad Circuits](#).

Usando cualquiera de las dos versiones, podemos realizar pequeños programas de control con librerías de clases ya disponibles. A continuación mostramos algunos ejemplos, que pueden ayudar a familiarizarse con el uso de clases.

Programar para Arduino tiene algunas pequeñas diferencias con el C++ estándar. Una de ellas es que no existe `main()`, o, mejor dicho, está oculto al programador y realiza el siguiente trabajo:

```
Install_path/cores/arduino/main.cpp (no accesible al programador)
int main(void)
{
 init();
 initVariant();
#ifdef USBCON
 USBDevice.attach();
#endif
 setup(); // User supplied function
 for (;;) {
 loop(); // User supplied function
 if (serialEventRun) serialEventRun();
 }
 return 0;
}
```

El hecho de que no exista `main()` a disposición del programador hace que se tenga que distribuir el desarrollo en 2 funciones principales:

- ❑ `setup()`: se llama una única vez, para inicializar todos los elementos
- ❑ `loop()`: se llama repetidamente a la velocidad máxima de ejecución del equipo

Debido a esta configuración, todas las variables que deban mantener su valor entre llamadas a `setup()` y sucesivas a `loop()` deben ser globales (en una aplicación "normal" se declararían locales en `main` y se pasarían como argumentos a las funciones correspondientes). Esto no nos debe despistar, cuando realicemos funcionalidades genéricas debemos evitar en lo posible estas variables globales, y seguir realizando funciones y clases que utilicen argumentos y valores devueltos.

#### 4.1 Programa básico en Arduino

Un programa básico en Arduino debe ser similar al siguiente:



#### ej4\_step\_0.ino

```
#include ... // Archivos de cabecera necesarios

// Declaración de variables globales: mantienen su valor durante todo el programa
// y son accesibles desde todas las funciones
// Usar solamente las imprescindibles
tipo1 _vbleGlobal1;
clase2 _objetoGlobal2;
...

// Declaración de funciones
tipo3 MyFn(tipo1 arg1,int arg2, ...)
{
 tipo3 result;
 ...
 result = ...;
 return result;
}

void setup() {
 // put your setup code here, to run once:
 _vbleGlobal1 = valor inicial;
 _objetoGlobal2.Init(argumentos);
 ...
}

void loop() {
 // put your main code here, to run repeatedly:
 tipo3 vbleLocal3; // vbles locales: se crean nuevas en cada loop()
 clase4 objetoLocal4; // y se destruyen cada vez que finaliza

 vbleLocal3 = MyFn(_vbleGlobal1,47);
 ...
}
```

En el ejemplo anterior, dado que la invocación a MyFn() desde loop() utiliza el valor de \_vbleGlobal1 como argumento, puede haber la tendencia a programar no usando este argumento, de forma que MyFn() utilice directamente la variable global. **Aunque es posible, no es aconsejable.** Si una función sólo depende de argumentos para producir resultados, es mucho más conveniente que utilice argumentos y no directamente variables globales.

## 4.2 Clases y funciones más utilizadas en Arduino

Las clases estándar C++ (std) no están disponibles por defecto en Arduino (sí lo están, sin embargo, en algunas variantes "compatibles" como ESP32). Algunas de las clases que podemos utilizar para nuestro código, específicas de Arduino y compatibles:

- Clase Serial: para comunicaciones con otros dispositivos (muy habitualmente con consola)
- delay(), delayMicroseconds(), micros(), millis() : temporizaciones
- pinMode(), digitalRead(), digitalWrite(), analogRead(), analogWrite(): E/S pines
- Clase Wire para comunicaciones I2C, y clase SPI para comunicaciones SPI

### 4.2.1 Serial

Un microcontrolador como Arduino no tiene una consola por defecto (teclado+pantalla). Lo más parecido es la comunicación serie (habitualmente a través de un convertor USB-Serie) que permite que nuestro programa se comunique (enviando y recibiendo texto) con un programa en el ordenador al que está conectado.

La clase Serial de Arduino facilita esta tarea, ya que provee de toda la funcionalidad para conectar el equipo a través de un (o varios) puerto serie.

Habitualmente usamos de la manera siguiente:



ej4\_step\_1.ino

```
#include "Arduino.h"

int _counter;

void setup() {
 Serial.begin(115200);
 _counter=0;
 Serial.print("Starting in setup\n");
}

void loop() {
 _counter=_counter+1;
 Serial.print("Running loop. Counter = ");
 Serial.println(_counter);
 delay(1000); // Ver apartado siguiente
}
```

A destacar del programa anterior:

- En muchos ejemplos no aparece la línea `#include "Arduino.h"`. Esto es porque Arduino IDE "entiende" que siempre querremos este include, y lo añade por defecto para el archivo .ino.
- Como se ha indicado anteriormente, las variables que deben mantener su valor entre llamadas a `setup()` o `loop()` deben ser globales. Es el caso de `_counter`.
- Hace falta un objeto de clase `Serial` para acceder a un puerto serie, sin embargo no lo hemos declarado. Esto es porque Arduino IDE "entiende" que siempre lo querremos, y declara uno por defecto (que es global y además se llama igual que la clase).
- En `setup()` llamamos a la función `begin()` del objeto (que no la clase) `Serial`, para inicializar la comunicación; hay que indicar la velocidad en baudios (que debe ser la misma que la del otro extremo).
- Las funciones `print()` y `println()` de `Serial` son polimórficas, aceptan multitud de tipos de argumento (texto, entero, ...). `println()` hace lo mismo que `print()` pero añade un salto de línea al final. También se puede conseguir ese salto con `\n` en el texto.
- `loop()` se invoca automáticamente a la máxima velocidad posible.
- `delay()` se describe a continuación. Sirve para retardar la ejecución durante un tiempo, ya que la ejecución es muy rápida (decenas de millones de instrucciones por segundo). No se ejecutan instrucciones durante este tiempo.

Más información de la clase `Serial` en <https://www.luisllamas.es/arduino-puerto-serie/>

## 4.2.2 Temporizaciones

La librería de Arduino dispone de las funciones (no clases) siguientes para realizar temporizaciones y calcular tiempos. El hecho de disponer de clases no obliga a que todo el código recaiga en ellas: si consideramos que una porción de código no tiene la orientación a clases/objetos, realizamos funciones.

|                                                            |                                                                                                                                                                                                                                                                                                                           |
|------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void delay(unsigned long time_ms);</code>            | Retarda la ejecución de la siguiente instrucción durante el tiempo indicado, expresado en ms. <b>¡¡¡ Intentar evitar en lo posible !!!</b><br><a href="https://www.arduino.cc/reference/en/language/functions/time/delay/">https://www.arduino.cc/reference/en/language/functions/time/delay/</a>                         |
| <code>void delayMicroseconds(unsigned int time_us);</code> | Retarda la ejecución de la siguiente instrucción durante el tiempo indicado, expresado en ms. <b>¡¡¡ Intentar evitar en lo posible !!!</b><br><a href="https://www.arduino.cc/reference/en/language/functions/time/delaymicroseconds/">https://www.arduino.cc/reference/en/language/functions/time/delaymicroseconds/</a> |
| <code>unsigned long millis();</code>                       | Devuelve el valor en milisegundos desde que comenzó la ejecución. Se produce overflow (regreso a 0) en algo menos de 50 días (50 días=4.320.000.000 ms, máx unsigned long = $2^{32}-1= 4.294.967.295$ )                                                                                                                   |
| <code>unsigned long micros();</code>                       | Devuelve el valor en milisegundos desde que comenzó la ejecución. Se produce overflow (regreso a 0) en algo más de 70 minutos (70                                                                                                                                                                                         |



minutos=4.200.000.000 ms, máx unsigned long =  $2^{32}-1= 4.294.967.295$ )

Un "buen" programador debe evitar las funciones delay, dado que evitan que se pueda ejecutar ningún otro código mientras dura la espera. En su lugar, es buena práctica utilizar millis() (o micros) y un valor que recuerde la última activación.

#### ej4\_step\_2.ino

```
#include "Arduino.h"

int _counter;
unsigned long _last_time;

void setup() {
 Serial.begin(115200);
 _counter=0;
 _last_time=millis();
 Serial.print("Starting in setup\n");
}

void loop() {
 unsigned long cur_time=millis();

 if (cur_time - _last_time > 1000) {
 _last_time=cur_time;
 _counter=_counter+1;
 Serial.print("Running loop. Counter = ");
 Serial.println(_counter);
 }
}
```

Dado que esta utilidad es muy práctica, y nos permite comprobar el vencimiento de temporizaciones variadas, podemos embeberla en una clase para que sea más cómoda de utilizar en múltiples casos y proyectos:

#### ej4\_step\_3.ino

```
#include "Arduino.h"
#include "MyDelayCheck.h"

int _counter1,_counter2;
MyDelayCheck _check1sec,_check2sec;

void setup() {
 Serial.begin(115200);
 _counter1=_counter2=0;
 _check1sec.begin(1000);
 _check2sec.begin(2000);
 Serial.print("Starting in setup\n");
}

void loop() {
 if (_check1sec.check()) {
 _counter1=_counter1+1;
 Serial.print("Running loop. Counter1 = ");
 Serial.println(_counter1);
 }
 if (_check2sec.check()) {
 _counter2=_counter2+1;
 Serial.print("Running loop. Counter2 = ");
 Serial.println(_counter2);
 }
}
```



#### MyDelayCheck.h

```
class MyDelayCheck
{
private:
 unsigned long last_time_ms=0;
 unsigned long delay_ms=0;
public:
 void begin(unsigned long i_delay_ms);
 bool check();
};
```

#### MyDelayCheck.cpp

```
#include "MyDelayCheck.h"
#include "Arduino.h"

void MyDelayCheck::begin(unsigned long i_delay_ms)
{
 last_time_ms =millis();
 delay_ms=i_delay_ms;
}

bool MyDelayCheck::check()
{
 if (last_time_ms==0) // No inicializado con begin()
 return false;

 unsigned long cur_time_ms=millis();
 if (cur_time_ms - last_time_ms >= delay_ms) {
 last_time_ms=cur_time_ms;
 return true;
 }
 return false;
};
```

### 4.2.3 Interacción con el exterior a través de pines

Para este caso la librería de Arduino proporciona por defecto funciones en lugar de clases. Según el modelo hardware, algunos pines serán válidos para unas operaciones de E/S y otros para otras, aunque muchos son seleccionables.

Un pin de entrada permite recibir información del mundo exterior. Puede ser de tipo todo/nada (o también llamada entrada digital) o de tipo analógico.

Un pin de salida permite generar acciones sobre el mundo exterior. Puede ser de tipo todo/nada (o también llamada salida digital) o de tipo analógico.

Para usar un pin en alguno de estos modos, lo primero es declarar su sentido. Se suele hacer en la función `setup()` o en funciones `begin()` dentro de clases:

```
void pinMode(uint8_t pinNumber, uint8_t mode);
```

Una vez declarado el sentido, según se necesite (y según acepte el pin):

- Si se ha declarado de entrada (INPUT), se pueden utilizar las funciones:
  - `int digitalRead(uint8_t pinNumber)` si está conectado a una entrada digital. En este caso, se puede usar también modo `INPUT_PULLUP` para habilitar (si está disponible en el hardware) una resistencia que pondrá valor 1 si el pin está flotante (sin conexión).
  - `uint16_t analogRead(uint8_t pinNumber)` si está conectado a una entrada analógica.
- Si se ha declarado de salida (OUTPUT), se pueden utilizar las funciones:
  - `void digitalWrite(uint8_t pinNumber, uint8_t value)` si está conectado a una salida digital.



- `void analogWrite(uint8_t pinNumber,int value)` si está conectado a una salida analógica o PWM.

#### 4.2.4 Comunicaciones I2C y SPI

Para I2C se usa habitualmente la clase **Wire**. Los Arduino y compatibles suelen tener pines específicos para esta comunicación. Más información en:

<https://www.luisllamas.es/arduino-i2c/>

<https://programarfacil.com/blog/arduino-blog/comunicacion-i2c-con-arduino>

Para SPI se usa habitualmente la clase SPI. Los Arduino y compatibles suelen tener pines específicos para esta comunicación. Más información en:

<https://www.luisllamas.es/arduino-spi/>

<https://programarfacil.com/blog/arduino-blog/spi-con-arduino-bmp280/>

#### 4.2.5 Control de motores

Para controlar la velocidad o posición de un conjunto mecánico accionado por un motor existen diversas librerías, dependiendo del tipo de motor utilizado:

- ❑ Paso a paso (stepper)
- ❑ Servo motor
- ❑ Motor DC con encóder

Información sobre los tipos de motores y sus usos en:

<https://www.luisllamas.es/guia-seleccion-motores-para-arduino/>

<https://www.luisllamas.es/tipos-motores-rotativos-proyectos-arduino>

Para control de motores se suele utilizar la librería Servo, que permite el accionamiento de motores variados (stepper, DC) con drivers de diferentes modelos. OJO: es una librería bastante limitada ya que no permite la gestión de los parámetros del control, pero es útil para casos sencillos.