Cuaderno de problemas
Estructuras de Datos.

Prof. Isabel Pita

23 de enero de 2023

Índice

1. Definición de un Tipo Abstracto de Datos. ¿Cuánto queda para las vacaciones?

¿Cuánto queda para las vacaciones?

Las personas se dividen en dos grupos: las que vuelven de las vacaciones descansadas, felices, hablando del buen tiempo que hizo y las que vuelven estresadas, cansadas, contando toda una serie de accidentes que inevitablemente les pasan todos los años. Sin embargo, aun a pesar de estas diferencias, todos ellos siguen preguntándose ¿cuánto queda para las próximas vacaciones? Unos con ilusión y aire de esperanzada, los otros con horror y una mirada de pánico que intentan ocultar.

En este problema vamos a ayudarles a unos y otros a calcular cuántos días quedan para las próximas vacaciones. Hay que tener en cuenta que en nuestro pais todos los años tiene 12 meses, y todos los meses tienen 30 días.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de 6 números, los tres primeros representan la fecha actual en el formato: día, mes, año. Los tres siguientes representan la fecha en que comenzarán las próximas vacaciones.

Salida

Para cada caso de prueba se escribe en una línea el número de días que quedan para que comiencen las vacaciones. Si la fecha actual es mayor que la fecha dada de vacaciones se escribirá *Ya pasaron*. Si alguna de las dos fechas no es correcta se escribirá *Fecha invalida*.

Entrada de ejemplo

```
30 12 2017 1 1 2018
1 1 2018 30 12 2017
11 6 2018 1 8 2018
21 11 2015 4 3 2018
30 14 2015 2 4 2017
```

Salida de ejemplo

```
0
Ya pasaron
50
822
Fecha invalida
```

Autor: Isabel Pita

1.1. Objetivos del problema

- Aprender a definir un TAD e implementarlo mediante una clase en C++. Interfaz vs parte privada.
- Implementación de los constructores y del destructor de la clase.
- Implementación de métodos observadores y modificadores. Implementación *interna* vs implementación *externa* a la clase.
- Sobrecarga de operadores. Operadores de entrada/salida.
- Diferenciar entre un método de una clase y una función que recibe objetos de una clase como parámetros.
- Aprender a lanzar excepciones.
- Comparar diferentes implementaciones de un TAD.

1.2. Ideas generales.

- En el problema necesitamos realizar ciertas operaciones con fechas. Vamos a crear un TAD `Fecha` que nos permita manejar e implementar las fechas independientemente del programa que las vaya a utilizar.
- Para implementar el TAD `Fecha` utilizamos las clases de C++.
- La **declaración de una clase en C++** comienza con la palabra reservada `class` seguida del nombre de la clase. En una clase se diferencian tres partes:
  - La *parte pública* (*interfaz*), que va precedida de la palabra reservada `public`. En esta parte se declaran todas las operaciones que ofrece el TAD para su uso.
  - La *parte privada*, que va precedida de la palabra reservada `private`. En esta parte se declaran las variables necesarias para guardar la representación del TAD, y los métodos utilizados para la implementación de las operaciones públicas. Desde una función externa a la clase no se puede acceder ni a las variables ni a los métodos declarados en la parte privada.
  - La *parte protegida*, que va precedida de la palabra reservada `protected`. Semejante a la parte privada, permite acceder a los datos a las clases que la heredan.
- Realizamos una primera implementación de la clase en la que representamos una fecha mediante tres enteros que representan el día, el mes y el año. Para ello declaramos en la parte privada de la clase tres *atributos* de tipo entero. Además para poder manejar fechas necesitamos el número de días que tiene cada mes, el número de meses del año y el número de días que tiene un año. Guardamos estas cantidades en constantes de tipo entero.
- Realizamos una segunda implementación de la clase en la que representamos una fecha mediante un valor entero que representa el número de días transcurridos desde una determinada fecha, por ejemplo el 1 de enero de año 1. En la parte privada tendremos un atributo de tipo entero y las tres constantes que guardan el número de días que tiene cada mes, el número de meses del año y el número de días que tiene un año. Es interesante observar en la implementación que se proporciona al final del problema, como algunas operaciones son mas sencillas de implementar con esta representación que con la otra y en cambio otras son más complicadas. En este caso el orden de complejidad de los métodos es el mismo con las dos implementaciónes, pero esto no es cierto en general, pudiendo variar el orden de complejidad de unas representaciones a otras. Se debe seleccionar aquella representación que nos permite obtener mejor orden de complejidad en las operaciones que más se utilizan del tipo.
- **Constructores.** Los constructores se utilizan para declarar objetos de la clase. Se nombran con el identificador de la clase y no devuelven ningún valor.

- El lenguaje proporciona un constructor por defecto sin parámetros. Este constructor inicializa los atributos de la clase al valor por defecto del tipo que tengan. Suele ser conveniente definir nuestros propios constructores lo que nos permite dar los valores que deseemos a los atributos. Debemos tener en cuenta que si realizamos nuestros propios constructores, el constructor por defecto deja de estar definido.
- Pueden realizarse tantos constructores como se quiera. El compilador los diferencia por el número y tipo de los parámetros, igual que ocurre con la sobrecarga de funciones. Los constructores facilitan la creación de objetos y permiten comprobar restricciones sobre los atributos. Si se implementa algún constructor, es conveniente implementar también uno sin parámetros, ya que el constructor por defecto deja de estar disponible.
- En este problema, realizamos un constructor que reciba el día, mes y año de la fecha que queramos construir. En el constructor comprobaremos que los datos dados en los parámetros son correctos, y en caso contrario lanzaremos una excepción para indicar el error.
- **Destructor**. Se utiliza para devolver la memoria utilizada por un objeto al sistema.
- En una clase solo se puede declarar un destructor. El lenguaje proporciona uno por defecto al igual que ocurre con el constructor.
- Definiremos nuestro propio destructor para la clase cuando debamos devolver explícitamente recursos utilizados por el objeto, por ejemplo, liberar memoria dinámica o cerrar algún fichero.
- **Operaciones del TAD**. Las operaciones se implementan por medio de métodos declarados en la parte pública de la clase. La implementación puede utilizar funciones auxiliares que no son operaciones del TAD y que se implementarán en la parte privada de la clase.
- La decisión de si un método es público o privado se realiza en base al uso que se quiera hacer del TAD. Si el método es interesante para manejar el tipo de datos, entonces lo declararemos como operación del TAD en la parte pública. Por el contrario, si el método resuelve un detalle concreto de la implementación de una operación pública, entonces lo declararemos privado. En la clase `Fecha`, el método `desdePrimeroAnyo` que se utiliza en la implementación del operador resta es privado, ya que se considera que no es interesante para el TAD contar con esta operación.
- La declaración de las operaciones tanto públicas como privadas se realiza dentro de la clase. La implementación puede realizarse al tiempo que se declaran en la clase, o posteriormente fuera de la clase.
- Si las operaciones se corresponden con algún operador se implementarán utilizando sobrecarga de operadores.
- Los operadores de inserción (escritura de datos en el buffer) y extracción (lectura de datos del buffer) utilizados para leer y escribir datos son funciones externas a la clase. Cuando la función es externa a la clase, debe recibir el objeto como parámetro. Estas funciones deben declararse `inline`.
- Para **declarar un objeto** escribimos el nombre de la clase seguida del nombre del objeto y de los argumentos con los que queremos llamar al constructor.
- Para **llamar a un método** se utiliza el nombre de un objeto de la clase, el operador punto y el nombre del método con sus parámetros.
- **Tratamiento de errores.** Cuando se produzca un error en la implementación de un método, lanzaremos una *excepción* que podrá ser tratada por la función que utiliza la clase. Para el tratamiento de las excepciones utilizamos la librería `stdexcept` que nos permite diferenciar los siguientes errores:
  - `domain_error`: las funciones no están definidas para esos valores, por ejemplo si intentamos dividir por cero.
  - `invalid_argument`: los argumentos no son correctos.
  - `out_of_range`: para los accesos fuera de rango en vectores y arrays.
  - `length_error`: cuando se reserve memoria y no haya más disponible.

- Las excepciones las trataremos con una instrucción `try...catch...`.
  - En el bloque `try` se implementan las instrucciones a ejecutar para resolver el problema.
  - Si en algún momento de la ejecución se produce un error, la ejecución continúa con el bloque `catch` correspondiente al tipo de excepción producido.
  - Se pueden implementar varios bloques `catch` uno para cada excepción para la que queramos un comportamiento específico.
  - Existe una instrucción `catch` que permite tratar cualquier tipo de excepción.
  - Pueden realizare varias instrucciones `try...catch...` en una función siempre que cada bloque `try` tenga sus correspondientes bloques `catch`. También pueden anidarse los bloque.
  - También puede haber instrucciones del programa que no estén dentro del bloque `try`. En caso de que estas instrucciones lancen una excepción, ésta no será tratada por esta función, pero se propagará a la función que la llamó. Si esta función está preparada para capturar la excepción, se tratará aquí, y sino seguirá propagándose por las funciones que hicieron las llamadas hasta que encuentre una que la trate o llegue a la función principal.

1.3. Algunas cuestiones sobre implementación.

- Las clases y sus operaciones las implementaremos en un fichero `.h`.
- **Declaración de una clase.** La parte privada y la parte pública se pueden escribir en cualquier orden. También se puede, aunque en general no se recomienda, ir alternando las diferentes partes, apareciendo estas varias veces. Si la parte privada se escribe en primer lugar no es necesario poner la palabra reservada `private`, si se declara después de la parte pública es obligatorio escribirlo. La clase debe terminar con un punto y coma.

```
class Fecha {
public:
  ....
private:
  ....
};
```

o bien:

```
class Fecha {
  ...... // Parte privada
public:
  ....
};
```

- La **declaración de los atributos** de la clase es semejante a la declaración de constantes y variables. Si se realiza en la parte privada, para poder consultar o modificar su valor será necesario un método público que acceda al dato. Se recomienda hacerlo así para evitar errores de modificaciones indebidas de los datos. Observad que el constructor comprueba que la fecha dada en los parámetros es correcta. Si permitimos que se modifiquen los atributos libremente, se podrían asignar valores que no fuesen correctos y no se detectaría:

```
const int DIAS_MES = 30;
const int MESES_ANYO = 12;
const int DIAS_ANYO = DIAS_MES * MESES_ANYO;
int dia, mes, anyo;
```

  Los valores no constantes los inicializaremos en el constructor.

- Los **constructores** reciben el mismo nombre que la clase y no se declara tipo de retorno, devuelven siempre un valor del tipo. En este problema, declaramos dos constructores, uno sin parámetros, ya que al declarar uno con parámetros se anula el constructor por defecto y otro que nos permite inicializar la fecha con valores de día, mes y año. En el constructor comprobamos que los datos introducidos son correctos y en caso de no serlo lanzamos una excepción.

- Los atributos se pueden inicializar utilizando la *lista de inicialización*. El uso de listas de inicialización es más eficiente cuando los atributos tienen inicializaciones complejas, además es necesario cuando se quiere inicializar un atributo constante o cuando un atributo pertenece a una clase que no tiene un constructor sin parámetros, o para inicializar un atributo de clase vector a un cierto tamaño.

```
Fecha(): dia(1), mes(1), anyo(0){};
Fecha(int d, int m, int a): dia(d), mes(m), anyo(a){
    if (m <= 0 || m > MESES_ANYO)
        throw std::invalid_argument("Fecha invalida");
    else if (d <= 0 || d > DIAS_MES)
        throw std::invalid_argument("Fecha invalida");
};
```

- Los atributos se pueden también inicializar con instrucciones de asignación en el cuerpo de la función. Esta forma de inicializar los atributos no permite crear un vector con un tamaño dado. Se deberá utilizar la función `resize`, por lo que si hay vectores entre los atributos es mejor utilizar la lista de inicialización.

```
Fecha(int d, int m, int a) {
  dia = d; mes = m; anyo = a;
    if (m <= 0 || m > MESES_ANYO)
        throw std::invalid_argument("Fecha invalida");
    else if (d <= 0 || d > DIAS_MES)
        throw std::invalid_argument("Fecha invalida");
};
```

- La palabra reservada `this` se refiere a un puntero a la propia clase. En las implementaciones de los métodos de la clase es opcional. En las implementaciones que se realicen en clases que heredan de una dada, es obligatorio para referirnos a atributos de la clase de la que se hereda. Si queremos referirnos a un atributo o método de la clase utilizaremos el operador `->`. Si queremos referirnos al contenido de la clase utilizaremos el operador `*`.

```
Fecha(int d, int m, int a) {
  dia = d; mes = m; anyo = a;
    ...
};
```

o bien

```
Fecha(int d, int m, int a) {
  this->dia = d; this->mes = m; this->anyo = a;
...
};
```

- **Declaración de las operaciones en la parte pública de la clase e implementación en la propia clase**. Las tres primeras funciones se utilizan para poder acceder a los valores de los atributos desde fuera de la clase. La última función devuelve el número de días entre dos fechas.

```
int get_dia() const {return dia;};
int get_mes() const {return mes;};
int get_anyo() const {return anyo;}
int diasQueFaltan const (Fecha const& f);
```

Las funciones se declaran `const` para indicar que no modifican los atributos de la clase.

- **Declaración de métodos privados para usarlos en la implementación de las operaciones públicas**. Estos métodos se declaran en la parte privada de la clase. Al igual que las operaciones pueden implementarse en la clase, o fuera de esta.

```
// Calcula el numero de dias transcurridos desde principio de anyo.
int desdePrimeroAnyo() const {
    return (this->mes - 1) * 30 + this->dia;
}
```

- Si implementamos las funciones fuera de la clase, deben ir precedidas del nombre de la clase.

```
class Fecha {
  ...
};

int Fecha::diasQueFaltan const (Fecha const& f) {
  ...
}
```

En ED, para simplificar el código, realizaremos las implementaciones dentro de la clase siempre que sea posible.

  - Cuando la implementación sea excesivamente larga y sea conveniente realizarla fuera de la clase, la implementaremos en el propio fichero .h detrás de la implementación de la clase.
  - Las funciones que no pertenezcan a la clase, pero se implementen en el fichero `.h` deben declararse `inline` para evitar que se produzca un error de compilación. Algunos compiladores compilan las funciones implementadas en los ficheros `.h`, esto produce que al compilar un fichero en el que se incluye el fichero `.h` se vuelva a intentar compilar la misma función produciéndose un error de función repetida. Al declarar la función `inline` el código de la función se incorpora al código del programa en las instrucciones en que se llama a la función durante la fase de preprocesado del compilador. Esto evita la compilación separada de la función.

En un proyecto *serio* la implementación de las funciones se realiza en un fichero con extensión `.cpp`, dejando sólo la interfaz en el fichero `.h`.

- Las operaciones que se implementen con sobrecarga de operadores las realizaremos siguiendo las mismas pautas que los otros métodos:

```
bool operator< (Fecha const& f) const{
    if (anyo < f.anyo) return true;
    else if (anyo > f.anyo) return false;
    else if (mes < f.mes) return true;
    else if (mes > f.mes) return false;
    else if (dia < f.dia) return true;
    else return false;
}

bool operator== (Fecha const& f) const {
    return anyo == f.anyo && mes == f.mes && dia == f.dia;
}
```

- Lectura y escritura de objetos de una clase. La sobrecarga de los operadores de extracción e inserción para realizar la lectura y escritura de datos no pertenecen a la clase. Observad que se utiliza la palabra reservada `inline` para que el preprocesador sustituya las llamadas a la función por el propio código.

```
inline std::ostream& operator<< (std::ostream & out, Fecha const& f) {
    out << f.get_anyo() << ' ' << f.get_mes() << ' ' << f.get_dia() << '\n';
    return out;
}
```

Si no se cuenta con métodos que accedan a los atributos privados, se puede implementar un método `print` en la clase y llamar a este método desde el operador.

```
void Fecha::print(std::ostream & out) const {
    out << anyo << ' ' << mes << ' ' << dia << '\n';
}

inline std::ostream& operator<< (std::ostream & out, Fecha const& f) {
    f.print(out);
    return out;
}
```

La implementación del extractor requiere la sobrecarga del operador de asignación de la clase, que debe declararse dentro de la clase.

```
Fecha & Fecha::operator=(Fecha const& other){ // operador de asignacion
    if (this != &other) {
        dia = other.dia; mes = other.mes; anyo = other.anyo;
    }
    return *this;
}


inline std::istream& operator>> (std::istream & in, Fecha & f) {
    int a,m, d;
    in >> a >> m >> d;
    f = Fecha(a,m,d);
    return in;
}
```

- Declaración y uso de una clase:

```
    int d,m,a;
    // Fecha actual
    std::cin >> d >> m >> a;
    Fecha fActual(d,m,a);
    // Fecha cuando empiezan las vacaciones
    std::cin >> d >> m >> a;
    Fecha fVacaciones(d,m,a);
    std::cout << "Quedan " << fVacaciones.diasQueFaltan(fActual);
    std::cout << " para las vacaciones\n";
```

- Para lanzar una excepción se utiliza la instrucción `throw`, seguida del tipo de excepción y entre paréntesis el mensaje que se quiere mostrar al usuario:

```
    Fecha(int d, int m, int a): dia(d), mes(m), anyo(a){
        if (m <= 0 || m > MESES_ANYO)
            throw std::invalid_argument("Fecha invalida");
        else if (d <= 0 || d > DIAS_MES)
            throw std::invalid_argument("Fecha invalida");
    };
```

- Para tratar una excepción producida en un método se utiliza la instrucción `try ...catch...`. Las instrucciones del programa se escriben en el bloque `try`. Cuando se produce un error pasan a ejecutarse las instrucciones del bloque `catch` correspondiente a la excepción lanzada.

  Cuando se están leyendo datos en el juez, no deben ejecutarse funciones que puedan lanzar excepciones antes de terminar de leer todos los datos del caso, en caso contrario debemos terminar de leer los datos del caso en el bloque `catch` correspondiente a la excepción para que no se mezclen los datos de un caso con los de otro en la entrada de datos.

```
try{
    int d1,m1,a1;
    std::cout << "Escriba la fecha actual\n";
    std::cin >> d1 >> m1 >> a1;
    int d2,m2,a2;
    std::cout << "Escriba cuando empiezan las vacaciones\n";
    std::cin >> d2 >> m2 >> a2;
    Fecha fActual(d1,m1,a1);
    Fecha fVacaciones(d2,m2,a2);
    std::cout << "Quedan " << fVacaciones.distancia(fActual) << " para las vacaciones\n";
}
catch (std::invalid_argument & ia) {
    std::cout << ia.what() << '\n';
}
```

```
catch (std::domain_error & de) {
    std::cout << de.what() << '\n';
}
catch (...) {
   std::cout << "No es ninguna de las excepciones anteriores\n";
}
```

El método `what` definido en las excepciones de la librería permite mostrar el mensaje que se indica en la instrucción `throw`.

Si utilizamos el juez en lugar de pedir los datos al usuario como en el ejemplo anterior, podemos leer los datos antes de entrar en la instrucción `try ...catch...`

```
int d1,m1,a1,d2,m2,a2;
std::cin >> d1;
if (!std::cin) return false;
std::cin >> m1 >> a1;
std::cin >> d2 >> m2 >> a2;
try {
    Fecha fActual(d1,m1,a1);
    Fecha fVacaciones(d2,m2,a2);
    std::cout << fVacaciones.diasQueFaltan(fActual)<< '\n';
}
catch (std::invalid_argument & ia) {
    std::cout << ia.what() << '\n';
}
```

A continuación se muestra una forma de implementar la lectura de los datos un tanto artificial, en la cual se lee la primera fecha dentro de la instrucción `try ...catch...`, con el fin de mostrar como se pueden anidar estas instrucciones.

```
int d1,m1,a1,d2,m2,a2;
std::cin >> d1;
if (!std::cin) return false;
try {
  std::cin >> m1 >> a1;
  Fecha fActual;
  fActual = {d1,m1,a1};
  std::cin >> d2 >> m2 >> a2;
  try {
    Fecha fVacaciones(d2,m2,a2);
    std::cout << fVacaciones.diasQueFaltan(fActual)<< '\n';
  }
  catch (std::invalid_argument & ia) {
    std::cout << ia.what() << '\n';
  }
}
catch (std::invalid_argument & ia) {
  std::cout << ia.what() << '\n';
  std::cin >> d2 >> m2 >> a2;
}
```

1.4. Implementación en C++.

- Implementación de la clase `Fecha` en el fichero `.h`. El código se protege de inclusiones múltiples con una cláusula `ifndef...endif`:
- **Primera representación:**

```
#ifndef FECHA
#define FECHA

#include <iostream>
```

```cpp
#include <array>
#include <stdexcept>


class Fecha {
public:
    // Constructores
    Fecha (){}
    Fecha(int d, int m, int a): dia(d), mes(m), anyo(a){
        if (m <= 0 || m > MESES_ANYO) throw std::invalid_argument("Fecha invalida");
        else if (d <= 0 || d > DIAS_MES)
            throw std::invalid_argument("Fecha invalida");
    }
    // operador de asignacion
    Fecha & operator=(Fecha const& other){
        if (this != &other) {
            dia = other.dia; mes = other.mes; anyo = other.anyo;
        }
        return *this;
    }
    // consultoras
    int get_dia() const {return dia;}
    int get_mes() const {return mes;}
    int get_anyo() const {return anyo;}

    // Numero de dias transcurridos desde la fecha del parametro (anterior)
    // hasta la fecha del objeto (posterior).
    // -1 si la fecha del objeto es anterior a la del parametro.
    int operator- (Fecha const& fAnt) const {
        if (*this <= fAnt) return -1;
        else
          return (this->anyo-fAnt.anyo)*DIAS_ANYO+this->desdePAnyo()-fAnt.desdePAnyo()-1;
    }

    // operadores de comparacion
    bool operator< (Fecha const& f) const{
        if (anyo < f.anyo) return true;
        else if (anyo > f.anyo) return false;
        else if (mes < f.mes) return true;
        else if (mes > f.mes) return false;
        else if (dia < f.dia) return true;
        else return false;
    }

    bool operator== (Fecha const& f) const {
        return anyo == f.anyo && mes == f.mes && dia == f.dia;
    }

    bool operator<= (Fecha const& f) const {
        return *this < f || *this == f;
    }

    bool operator> (Fecha const& f) const {
        return !(*this <= f);
    }

    bool operator>= (Fecha const& f) const {
        return !(*this < f);
    }

private:
    const int DIAS_MES = 30;
```

```
    const int MESES_ANYO = 12;
    const int DIAS_ANYO = DIAS_MES * MESES_ANYO;
    int dia, mes, anyo;


    // Calcula el numero de dias transcurridos desde principio de anyo.
    int desdePAnyo() const {
        return (this->mes - 1) * 30 + this->dia;
    }
};

inline std::ostream& operator<< (std::ostream & out, Fecha const& f) {
    out << f.get_anyo() << ' ' << f.get_mes() << ' ' << f.get_dia() << '\n';
    return out;
}

inline std::istream& operator>> (std::istream & in, Fecha & f) {
    int a,m, d;
    in >> a >> m >> d;
    f = Fecha(a,m,d);
    return in;
}

#endif
```

- **Segunda representación:**

```
class Fecha {
public:
    // Constructores
    Fecha (){}
    Fecha(int d, int m, int a): dias((a-1)*DIAS_ANYO + (m-1)*DIAS_MES+(d-1)){
        if (m <= 0 || m > MESES_ANYO) throw std::invalid_argument("Fecha invalida");
        else if (d <= 0 || d > DIAS_MES)
            throw std::invalid_argument("Fecha invalida");
    }
    // operador de asignacion
    Fecha & operator=(Fecha const& other){
        if (this != &other) {
            dias = other.dias;
        }
        return *this;
    }
    // consultoras
    int get_dia() const {return (dias%DIAS_ANYO)%DIAS_MES+1;}
    int get_mes() const {return (dias%DIAS_ANYO)/DIAS_MES+1;}
    int get_anyo() const {return dias/DIAS_ANYO+1;}

    // Numero de dias transcurridos desde la fecha del parametro (anterior)
    // hasta la fecha del objeto (posterior).
    // -1 si la fecha del objeto es anterior a la del parametro.
    int operator- (Fecha const& fAnt) const {
        if (*this <= fAnt) return -1;
        else return this->dias - fAnt.dias-1;
    }

    // operadores de comparacion
    bool operator< (Fecha const& f) const{
        return this->dias < f.dias;
    }

    bool operator== (Fecha const& f) const {
```

```
        return this->dias == f.dias;
    }

    bool operator<= (Fecha const& f) const {
        return *this < f || *this == f;
    }

    bool operator> (Fecha const& f) const {
        return !(*this <= f);
    }

    bool operator>= (Fecha const& f) const {
        return !(*this < f);
    }

private:
    const int DIAS_MES = 30;
    const int MESES_ANYO = 12;
    const int DIAS_ANYO = DIAS_MES * MESES_ANYO;
    int dias;
};
```

- **Implementación del código que resuelve el problema en un fichero** `.cpp`

```
#include <iostream>
#include <fstream>
#include "Fecha.h"

bool resuelveCaso()
{
    int d1,m1,a1,d2,m2,a2;
    std::cin >> d1;
    if (!std::cin) return false;
    std::cin >> m1 >> a1;
    std::cin >> d2 >> m2 >> a2;
    try {
        Fecha fActual(d1,m1,a1);
        Fecha fVacaciones(d2,m2,a2);
        int n = fVacaciones -fActual;
        if (n == -1) std::cout << "Ya pasaron\n";
        else std::cout << n << '\n';
    }
    catch (std::invalid_argument & ia) {
        std::cout << ia.what() << '\n';
    }
    return true;
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif
    return 0;
}
```