Cuaderno de problemas
Estructuras de Datos.

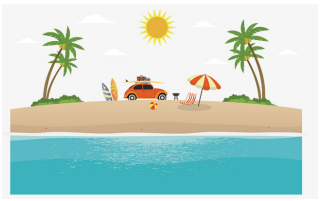
Prof. Isabel Pita

18 de febrero de 2021

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.

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 almacenar 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.
- 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.
- *Constructores.* El lenguaje proporciona un constructor por defecto sin parámetros. Este constructor no inicializa los atributos de la clase. Por lo tanto, es conveniente definir nuestros propios constructores.
- Podemos definir constructores que reciban parámetros. Estos constructores facilitan la creación de objetos de la clase, y permiten comprobar restricciones sobre los datos. 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.
- Si se realiza un constructor con parámetros, deja de tener efecto el constructor por defecto del lenguaje. En este caso, si se requiere un constructor sin parámetros, se debe implementar.
- 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*. 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 `numDiasEntreMeses` que se utiliza en la implementación de la función `diasQueFaltan` 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.
- 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. Observar que el constructor comprueba que la fecha es correcta, si permitimos que se modifiquen los atributos, 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. Los constructores devuelven un valor del tipo. En el problema, declaramos dos constructores, uno sin parámetros, ya que al declarar uno con parámetros queda sin efecto 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.

```
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 inicializar utilizando la *lista de inicialización*, tal como está hecho en el ejemplo anterior, o bien inicializarlos con instrucciones de asignación en el cuerpo de la función. 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.

```
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");
};
```

- 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.

```
// Funcion para calcular el numero de dias entre dos meses del mismo anyo.
// Se supone el mes de la segunda fecha siempre es posterior a la primera
// No se utiliza el anyo.
int numDiasEntreMeses(Fecha const& fAnterior, Fecha const& fPosterior) const {
    int totalDiasAnterior = (fAnterior.mes - 1) * 30 + fAnterior.dia;
    int totalDiasPosterior = (fPosterior.mes - 1) * 30 + fPosterior.dia;
    return totalDiasPosterior - totalDiasAnterior;
}
```

- 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.

```
inline std::ostream& operator<< (std::ostream & out, Fecha const& f) {
    std::cout << 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() const {
    std::cout << anyo << ' ' << mes << ' ' << dia << '\n';
}

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

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`:

```
#ifndef FECHA
#define FECHA

#include <iostream>
#include <array>
#include <stdexcept>


class Fecha {
public:
    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");
    };
    Fecha & operator=(Fecha const& other){ // operador de asignacion
        if (this != &other) {
            dia = other.dia; mes = other.mes; anyo = other.anyo;
        }
        return *this;
    }
    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).
    // Cero si la fecha del objeto es anterior a la del parametro.
int diasQueFaltan (Fecha const& fAnterior) const {
    if (*this <= fAnterior) throw std::invalid_argument("Ya pasaron");
    else if (this->anyo == fAnterior.anyo){
        if (this->mes == fAnterior.mes)  // mismo anyo y mismo mes
            return this->dia - fAnterior.dia;
        else  // mismo anyo distinto mes.
            return numDiasEntreMeses(fAnterior , *this);
    }
    else { // Distinto anyo
        // Completa este anyo hasta el final
        int suma = numDiasEntreMeses(fAnterior, {30,12,fAnterior.anyo});
        // Dias de los anyos completos
        suma += ((this->anyo - 1) - fAnterior.anyo) * 360;
        // Completa con el principio del anyo
        suma += numDiasEntreMeses({1,1,this->anyo}, *this);
        return suma;
    }
```

```
}

    // 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_ANYO = 360;
    const int DIAS_MES = 30;
    const int MESES_ANYO = 12;
    int dia, mes, anyo;


    // Funcion para calcular el numero de dias entre dos meses del mismo anyo.
    // Se supone el mes de la segunda fecha siempre es posterior a la primera
    // No se utiliza el anyo.
    int numDiasEntreMeses(Fecha const& fAnterior, Fecha const& fPosterior) const {
        int totalDiasAnterior = (fAnterior.mes - 1) * 30 + fAnterior.dia;
        int totalDiasPosterior = (fPosterior.mes - 1) * 30 + fPosterior.dia;
        return totalDiasPosterior - totalDiasAnterior;
    }



};

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
```

- 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);
        std::cout << fVacaciones.diasQueFaltan(fActual)<< '\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;
}
```

2. Elección del TAD apropiado para resolver un problema. El TAD set.

¿Ha llegado ya?

A Ester le gustan mucho las fiestas, reunirse con sus amigos y bailar toda la noche, pero a sus padres les preocupa que vuelva tarde, cuando ya está oscuro. Como no les apetece salir de madrugada a recogerla le han propuesto que organice las fiestas en casa. La casa es grande y puede invitar a todos los amigos que quiera.

La idea ha sido un éxito. Las fiestas en casa de Ester son famosas en el instituto hasta el punto que los chicos van invitando a sus parientes y amigos, de forma que muchas veces Ester ni siquiera conoce a los que se presentan en su casa. A Ester le encantan sus fiestas.

Sin embargo sus padres no están tan contentos. Les gusta ver feliz a su hija, pero cada vez que hay una fiesta están toda la noche levantándose. Los amigos nunca se presentan todos juntos a la hora que empieza la fiesta, sino que van llegando en grupos a lo largo de toda la noche. Llaman al teléfono automático y se identifican para que les abran la puerta. Con el ruido de la fiesta Ester nunca oye el timbre y son sus padres los que se levantan a abrir. Esto no puede seguir así.

Hoy por fin se les ha ocurrido la solución. Van a instalar un sistema de apertura automático. Cada vez que llega un grupo debe decir en el teléfono automático, en primer lugar el nombre de la persona que les invitó y a continuación el nombre de todos los que están en el grupo. Si esta persona ya está en la fiesta entonces la puerta se abrirá automáticamente, en caso contrario la puerta no se abre y el grupo se disuelve, cada uno a su casa. Hoy van a probar si el sistema funciona adecuadamente y por fin pueden dormir.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba comienza con el número de grupos que vendrán esta noche N, seguido del nombre del anfitrión de la fiesta. A continuación aparecen N líneas, cada una comienza con el nombre de la persona que les ha invitado, seguido del número de integrantes del grupo y a continuación los nombres de las personas del grupo. La entrada termina con un caso de prueba con cero grupos.

Todos los nombres son cadenas de caracteres sin blancos. Suponemos que todas las personas se llaman de forma distinta.

Salida

Para cada caso de prueba se escribe en primer lugar si cada uno de los grupos pudo entrar o no en una línea diferente. Si puede entrar se escribe *SI*, en caso contrario se escribe *NO*. A continuación se listan los nombres de todas las personas que finalmente entraron en la fiesta, uno por línea y por orden alfabético. El caso termina con una línea con 3 guiones.

Entrada de ejemplo

```
3 Ester
Ester 2 Ana Miguel
Beatriz 3 Javier Olga Carmen
Miguel 1 Patricia
2 Gonzalo
Ester 2 Borja Manuel
Manuel 1 Fernando
0
```

2.1. Objetivos del problema

- Aprender a seleccionar el tipo de datos más adecuado para nuestro problema.
- Conocer el TAD conjunto: `set`
- Aprender a implementar clases genéricas.
- Uso de memoria dinámica en la implementación de una clase.
  - Reserva de memoria en la constructora
  - Liberar la memoria en el destructor
  - Constructor por copia
  - Operador de asignación.

2.2. Ideas generales.

- Para resolver el problema necesitamos consultar si una persona ha llegado ya a la fiesta. Seleccionamos un tipo de datos que nos ofrezca operaciones para insertar elementos y para consultar si un elemento ha sido insertado.
  - `vector`: No tiene una operación específica que nos permita buscar un elemento. Debemos utilizar una de las operaciones genéricas definidas en la libreria `algorithm`. Para que la operación de búsqueda sea eficiente (búsqueda binaria) los datos del vector deben estar ordenados, esto requiere ordenar los datos cada vez que se incorpora un grupo utilizando una función genérica de la librería, o realizar una inserción ordenada, operación que no nos proporciona el TAD. La complejidad en el caso peor del algoritmo que resuelve el problema utilizando el tipo `vector` está en el orden de $\mathcal{O}(k\, n \log n)$ siendo k el número de grupos y n el número de invitados a la fiesta, ya que los algoritmos de ordenación tienen complejidad $\mathcal{O}(n \log n)$ y debemos ordenar cada vez que llega un grupo.
  - `set`: Es el tipo adecuado. Tiene operaciones para añadir un elemento (`insert`) y para buscar un elemento (`find` y `count`). Ambas operaciones tienen una complejidad en el caso peor del orden de $\mathcal{O}(\log n)$ siendo n el número de invitados a la fiesta, por lo tanto la complejidad del algoritmo implementado utilizando este TAD es $\mathcal{O}(n \log n)$ siendo n el número de invitados a la fiesta, ya que debemos añadir a todos los invitados al conjunto.
  - `map`: este tipo está pensado para almacenar información con una estructura de *clave-valor*. Proporciona operaciones para añadir elementos y consultar si un elemento ya ha sido añadido y también permite resolver el problema con complejidad en el caso peor del orden de $\mathcal{O}(n \log n)$ siendo n el número de invitados. Sin embargo en nuestro problema sólo necesitamos guardar el nombre de la persona, sin ninguna información asociada a ella. Por lo tanto, se prefiere el tipo `set` que mantiene una información única. Debe seleccionarse el tipo más ajustado al problema..
- Para resolver el problema utilizaremos el tipo `set` de la librería STL. Este tipo puede implementarse de forma muy eficiente utilizando árboles binarios de búsqueda. En este problema mostraremos una posible implementación basada en vectores dinámicos para ilustrar los principales conceptos de la implementación de TADs genéricos y de TADs que utilizan memoria dinámica.
- Observamos que las operaciones ofrecidas por el TAD `set` no dependen del tipo de los datos que se almacenan en el conjunto. La implementación de un conjunto de enteros y de un conjunto de cadenas de caracteres difiere únicamente en los tipos utilizados para los parámetros y variables, siendo iguales las instrucciones en ambos casos. C++ nos permite definir clases y funciones con tipos genéricos, que son instanciados cuando se crea una instancia concreta de la clase o de la función. Para ello se utiliza un *template*.
- Si se utiliza memoria dinámica en la implementación de la clase debemos:
  - Reservar explícitamente la memoria que vamos a utilizar. Normalmente se realiza en el constructor.

- Liberar la memoria cuando se destruye el objeto. Es necesario definir un destructor que libere de forma explícita la memoria dinámica utilizada en el conjunto.
- Definir un constructor por copia. El constructor por copia por defecto, realiza una copia *superficial* del objeto. Si se utiliza el constructor copia por defecto, el resultado será que el objeto copia y el objeto copiado comparten la memoria dinámica. Como resultado los cambios que se realicen en uno de los objetos quedan también realizados en el otro. Este comportamiento no suele ser el deseado y para evitarlo hay que realizar el constructor por copia, que realice una *copia profunda* de todos los datos del objeto, incluyendo los guardados en la memoria dinámica. Este constructor se utiliza cuando:
  - Se inicializa un objeto en su declaración:

    ```
    set<Horas> miOtroSet = miSet;
    ```

  - Un objeto se pasa como parámetro por valor a una función. Normalmente debe evitarse pasar objetos por valor debido a que la copia que se realiza del objeto suele ser costosa en tiempo y espacio.
- Definir un operador de asignación. Al igual que ocurre con el constructor por copia, el operador de asignación proporcionado por defecto por el lenguaje hace una copia superficial de los atributos del objeto. Para realizar una *copia profunda* de los datos, que incluya la memoria dinámica es necesario definir un operador de asignación que realice la copia explícitamente.

2.3. Algunas cuestiones sobre implementación.

- *Cómo definir una clase genérica.*
  - Para indicar que una clase es genérica escribiremos `template <typename T>` delante de la cabecera.

    ```
    template <typename T>
    class set {
    public:
      ...
    private:
      ...
    };
    ```

    - Se puede utilizar indistintamente la palabra `typename` o la palabra `class`. La primera fue introducida como palabra reservada en el estandar ISO C++ para evitar ambigüedades en el lenguaje debidas al uso de la palabra `class`.
    - El identificador `T` se utiliza para referirnos al tipo en la implementación de la clase. Puede utilizarse cualquier identificador, aunque es común utilizar `T`.
  - Cuando las funciones se implementan fuera de la clase, se debe poner `template <typename T>` delante de cada una de ellas para indicar que pertenecen a una clase genérica.
- *Cómo declarar un array dinámico.*

  Los arrays dinámicos se declaran como punteros al tipo de datos que almacenarán. Cuando se reserva la memoria para el array se indica el número de elementos que tendrá para que el compilador reserve la memoria consecutiva donde almacenar los datos. Estos arrays requieren controlar explícitamente la *capacidad* que tienen para almacenar datos y el número de datos que tienen realmente guardados. Se ofrece también la posibilidad de ampliar la capacidad del array cuando no tiene espacio para almacenar más valores.

  ```
  private:
      size_t contador;
      size_t capacidad;
      T * datos; // sin repeticiones
      void amplia();
  ```

- *Cómo reservar memoria dinámica en el constructor.*

  Utilizaremos el operador `new` para reservar la memoria.

```
template <typename T>
set<T>::set() : contador(0), capacidad(8), datos(new T[capacidad]) {}
```

- *Cómo liberar memoria dinámica en el destructor.*

  El destructor (sólo puede haber uno), tiene el mismo nombre que la clase precedido del símbolo ∼. Como ocurre con el constructor no devuelve ningún valor. El destructor se ejecuta cuando un objeto sale del ámbito en el que fue declarado. No se puede llamar explícitamente a esta operación en el código.

```
template <typename T>
set<T>::~set() { libera(); }
```

- *Implementación del constructor por copia.*

  El constructor por copia es un constructor y por lo tanto se llama igual que la clase y no se declara tipo de retorno de la función. La implementación hace uso de una función privada `copia`. En esta función se realiza una copia explícita de cada uno de los datos del array.

```
template <typename T>
set<T>::set(set<T> const& other) {
    copia(other);
}

template <typename T>
void set<T>::copia(set<T> const& other) {
    capacidad = other.capacidad;
    contador = other.contador;
    datos = new T[capacidad];
    for (size_t i = 0; i < contador; ++i)
        datos[i] = other.datos[i];
}
```

- *Implementación del operador de asignación.*

  El operador sólo realiza la asignación si el objeto asignado es diferente del objeto al que lo asignamos. Para comprobarlo compara las direcciones de memoria de ambos objetos. Al realizar la asignación es necesario liberar la memoria dinámica del objeto al que le estamos asignando el valor. Por último utilizamos la misma función `copia` que se utiliza en el constructor por copia. Observar que se devuelve el objeto, no su dirección de memoria.

```
template <typename T>
set<T> & set<T>::operator=(set<T> const& other) {
    if (this != &other) {
        libera();
        copia(other);
    }
    return *this;
}

template <typename T>
void set<T>::libera() { delete[] datos; }
```

2.4. Implementación en C++.

Implementación de un TAD `set` (Obtenida de las transparencias del profesor Alberto Verdejo).

La implementación se realiza en un fichero `.h`. Esta implementación se muestra únicamente a efectos didácticos para explicar las clases genéricas y el uso de memoria dinámica en clases. No se pretende que la implementación sea completa ni eficiente.

```cpp
#ifndef conjunto_h
#define conjunto_h
#include <cstddef> // size_t
#include <stdexcept> // std::domain_error
#include <utility> // std::moveclass set

template <typename T>
class set {
public:
    set(); // constructor
    set(set<T> const& other); // constructor por copia
    set<T> & operator=(set<T> const& other); // operador de asignacion
    ~set(); // destructor
    void insert(T e);
    bool contains(T e) const;
    void erase(T e);
    bool empty() const;
    size_t size() const;
private:
    size_t contador;
    size_t capacidad;
    T * datos; // sin repeticiones
    void amplia();
    void libera();
    void copia(set<T> const& other);
};

// Constructor
template <typename T>
set<T>::set() : contador(0), capacidad(8), datos(new T[capacidad]) {}

// Destructor
template <typename T>
set<T>::~set() { libera(); }

// Op. privada que libera la memoria dinamica
template <typename T>
void set<T>::libera() { delete[] datos; }

// constructor por copia
template <typename T>
set<T>::set(set<T> const& other) {
    copia(other);
}

// operador de asignacion
template <typename T>
set<T> & set<T>::operator=(set<T> const& other) {
    if (this != &other) {
        libera();
        copia(other);
    }
    return *this;
}

// Op. privada que copia el contenido de un vector
template <typename T>
void set<T>::copia(set<T> const& other) {
    capacidad = other.capacidad;
    contador = other.contador;
    datos = new T[capacidad];
    for (size_t i = 0; i < contador; ++i)
```

```
        datos[i] = other.datos[i];
}

// Operacion para anyadir elementos
template <typename T>
void set<T>::insert(T e) {
    if (!contains(e)) {
        if (contador == capacidad)
            amplia();
        datos[contador] = e;
        ++contador;
    }
}

// Op. privada que amplia la memoria del vector
template <typename T>
void set<T>::amplia() {
    T * nuevos = new T[2*capacidad];
    for (size_t i = 0; i < capacidad; ++i)
        nuevos[i] = std::move(datos[i]);
    delete[] datos;
    datos = nuevos;
    capacidad *= 2;
}


// Operacion que comprueba si un elemento pertenece al conjunto
template <typename T>
bool set<T>::contains(T e) const {
    size_t i = 0;
    while (i < contador && datos[i] != e)
        ++i;
    return i < contador;
}

// Operacion que elimina un elemento del conjuto
// Lanza una excepcion si el elemento no esta
template <typename T>
void set<T>::erase(T e) {
    size_t i = 0;
    while (i < contador && datos[i] != e)
        ++i;
    if (i < contador) {
        datos[i] = move(datos[contador-1]);
        --contador;
    } else
        throw std::domain_error("El elemento no esta");
}

// Comprueba si el conjunto esta vacio
template <typename T>
bool set<T>::empty() const {
    return contador == 0;
}

// Obtiene el numero de elementos del conjunto
template <typename T>
size_t set<T>::size() const {
    return contador;
}

#endif // conjunto_h
```

Solución al problema planteado utilizando la clase `set` de la STL.

```
#include <iostream>
#include <fstream>
#include <set>

bool resuelveCaso()
{
    // Lectura de datos
    int numGrupos;
    std::cin >> numGrupos;
    if (numGrupos == 0) return false;
    std::string anfitrion;
    std::cin >> anfitrion;
    // Anyade al anfitrion al conjunto
    std::set<std::string> s;
    s.insert(anfitrion);
    // Lee cada uno de los grupo. Si el conocido esta en el
    // grupo anyade a los miembros al grupo, en caso contrario
    // debe leer los nombres de los integrantes del grupo antes
    // de poder tratar al siguiente grupo
    for (int i = 0; i < numGrupos; ++i) {
        std::string conocido; int numMiembros;
        std::cin >> conocido >> numMiembros;
        if (s.find(conocido) == s.end()) {
            for (int j = 0; j < numMiembros; ++j) {
                std::string aux; std::cin >> aux;
            }
            std::cout << "NO\n";
        }
        else {
            std::cout << "SI\n";
            for (int j = 0; j < numMiembros; ++j) {
                std::string aux; std::cin >> aux; s.insert(aux);
            }
        }
    }
    for (std::string const& i : s)
        std::cout << i << '\n';
    std::cout << "---\n";
    return true;
}

int main() {
    // Para la entrada por fichero.
 #ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif
    while (resuelveCaso()){ } //Resolvemos todos los casos

    // Para restablecer entrada.
#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

3. Objetos función y comparadores

Buscando alojamiento

Este fin de semana queremos hacer una excursión por el norte de España. Hemos estado seleccionando todos los alojamientos que nos han parecido adecuados y ahora tenemos que decidir a cual de ellos iremos. El problema es que unos ofrecen mejor precio, pero están situados mas lejos de nuestro destino. Algunos tienen una puntuación muy buena, pero son un poco caros. No nos importa pagar un poco más si el hotel es mejor, o estar un poco más lejos, pero....!Es tan díficil elegir!.

Al final hemos decidido ordenar todos los alojamientos por la mayor puntuación, por la menor distancia, y por el menor precio. Si varios hoteles tienen la misma puntuación, distancia, o precio, los ordenaremos por orden alfabético del nombre del establecimiento. Mirando las tres listas veremos por cual nos decidimos.

Requisitos de implementación. Los datos de entrada se almacenarán en un vector que se ordenará siguiendo los tres criterios indicados. Una vez ordenado por un criterio se mostrará el resultado antes de realizar la siguiente ordenación.

Se utilizará como criterio principal la puntuación, y para ello se sobrecargará el operador `>`. Para realizar el orden por precio y distancia se utilizarán *objetos función*.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso comienza con una línea en que se indica el número de alojamientos, n, que se han conseguido. En las n líneas siguientes se muestra la descripción de cada hotel. Primero el nombre, una cadena de caracteres sin blancos, después la puntuación obtenida, la distancia al punto de destino y el precio.

La puntuación, distancia y precio son datos enteros mayores que cero.

Salida

Para cada caso de prueba se escribe primero la lista de los hoteles ordenados por puntuación, después la lista ordenada por distancia y por último la lista ordenada por precio.

Entrada de ejemplo

```
5
aaa 7 5 150
bbb 9 4 200
ccc 6 4 100
ddd 9 3 300
eee 7 7 100
4
fffff 4 5 200
tt 10 8 300
uuuu 8 3 160
ddd 8 4 200
```

Salida de ejemplo

```
bbb ddd aaa eee ccc
ddd bbb ccc aaa eee
ccc eee aaa bbb ddd
tt ddd uuuu fffff
uuuu ddd fffff tt
uuuu ddd fffff tt
```

3.1. Objetivos del problema

- Aprender a declarar objetos función
- Uso de los objetos función como comparadores de las funciones de la librería.

3.2. Ideas generales.

- El comparador por defecto de las funciones de la librería STL es el operador menor estricto (`<`).
- En muchos problemas necesitamos utilizar un comparador diferente, bien para ordenar una colección de mayor a menor, o bien para ordenarla por un criterio distinto del considerado para el tipo por defecto.
- El lenguaje proporciona la sobrecarga de operadores y los objetos función para resolver estos problemas. La sobrecarga de operadores debe utilizarse para definir el orden lógico de los elementos del TAD. En ningún caso debe usarse para definir un orden concreto para un problema concreto.
- Los objetos función permiten definir cualquier función.

3.3. Algunas cuestiones sobre implementación.

- Se pueden sobrecargar como métodos de la clase todos los operadores aritméticos y relacionales (`+`, `-`, `*`, `++`, `--`, `/`,`%`, `<`, `<=`, `>`, `>=`, `==`, `!=`). Se puede (y debe cuando hay uso de memoria dinámica) sobrecargar el operador de asignación `=`. Tambien se sobrecargan el operador de elemento de matriz `[]` y el operador función `()`. (Consultar todos los operadores que se pueden sobrecargar en ...)
- Los operadores de asignación `=`, elemento de matriz `[]`, invocación de función `()`, y selector indirecto de miembro `->` sólo pueden ser sobrecargados como funciones miembro de una clase.
- Los operadores de extracción e inserción de flujo no pueden sobrecargarse como funciones miembro de una clase. Si se implementan en el fichero `.h` deben declararse `inline` (problema **??**).
- *Sobrecarga de los operadores menor y mayor.* Cuando se puede, los operadores se deben sobrecargar como métodos de la clase. Los operadores binarios reciben un único parámetro, ya que el primer parámetro es pasado por el programa como `this`.

```
bool operator< (info const& i) const {
    return  (this->puntuacion < i.puntuacion) ||
            (this->puntuacion == i.puntuacion && this->nombre < i.nombre);
}

bool operator> (info const& i) const {
    return  (this->puntuacion > i.puntuacion) ||
            (this->puntuacion == i.puntuacion && this->nombre < i.nombre);
}
```

- *Uso de los operadores sobrecargados en las funciones de la librería.* Por defecto las funciones de la librería que admiten un parámetro *comparador*: `sort`, `merge`, `upper_bound`, `lower_bound` ... utilizan el operador menor del tipo. Si se quiere utilizar el operador mayor se debe llamar al objeto función `greater` de la librería `functional`:

```
std::sort(v.begin(), v.end(), std::greater<info>());
```

  Las funciones de la librería exigen que el operador sea estricto (menor estricto o mayor estricto). Si la implementación es menor o igual, o mayor o igual el comportamiento de la función no será el esperado.
- *Definición de objetos función.* Se define una clase que redefine el operador función. El comportamiento del operador se define en el cuerpo de la función. El operador se debe declarar público.

```
class ordDistancia {
public:
    bool operator()(info const& i1, info const& i2) {
        return (i1.get_distancia() < i2.get_distancia()) ||
        (i1.get_distancia() == i2.get_distancia() && i1.get_nombre() < i2.get_nombre()) ;
    }
};
```

- *Uso de un objeto función*:

```
std::sort(v.begin(), v.end(), ordDistancia());
```

Si la clase que define la función pertenece a otra clase, la llamada incluye el nombre de ésta:

```
std::sort(v.begin(), v.end(), info::ordDistancia());
```

3.4. Implementación en C++.

Fichero .h

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

class info {
    std::string nombre;
    int puntuacion, distancia, precio;
public:
    info(){}
    info(std::string nombre, int punt, int dist, int precio) : nombre(nombre), puntuacion(punt),
    std::string get_nombre()const {return nombre;}
    int get_puntuacion() const {return puntuacion;}
    int get_distancia()const {return distancia;}
    int get_precio()const {return precio;}

    // Para ordenar por puntuacion de menor a mayor.
    // A igualdad de puntuacion orden alfabetico
    bool operator< (info const& i) const {
        return  (this->puntuacion < i.puntuacion) ||
                (this->puntuacion == i.puntuacion && this->nombre < i.nombre);
    }

    // Para ordenar por puntuacion de mayor a menor
    bool operator> (info const& i) const {
        return  (this->puntuacion > i.puntuacion) ||
                (this->puntuacion == i.puntuacion && this->nombre < i.nombre);
    }

  // Sobrecarga del operador funcion para ordenar por distancia
  class ordDistancia {
  public:
    bool operator()(info const& i1, info const& i2) {
        return (i1.get_distancia() < i2.get_distancia()) ||
        (i1.get_distancia() == i2.get_distancia() && i1.get_nombre() < i2.get_nombre()) ;
    }
  };

  // Sobrecarga del operador funcion para ordenar por precio
  class ordPrecio {
  public:
    bool operator()(info const& i1, info const& i2) {
        return (i1.get_precio() < i2.get_precio()) ||
        (i1.get_precio() == i2.get_precio() && i1.get_nombre() < i2.get_nombre());
```

```
    }
  };
};

inline std::istream & operator>> (std::istream & flujo, info& hotel) {
    std::string auxNombre; int auxPuntuacion, auxDistancia, auxPrecio;
    flujo >> auxNombre >> auxPuntuacion >> auxDistancia >> auxPrecio;
    hotel = info(auxNombre,auxPuntuacion,auxDistancia,auxPrecio);
    return flujo;
}

inline std::ostream& operator<< (std::ostream & flujo, std::vector<info> const& v) {
    for (info hotel : v) flujo << hotel.get_nombre() << ' ';
    flujo << '\n';
    return flujo;
}
```

Fichero .cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <functional>
#include "info.h"

bool resuelveCaso() {
    // lectura de los datos
    int numHoteles;
    std::cin >> numHoteles;
    if (!std::cin) return false;
    std::vector<info> v(numHoteles);
    for (int i = 0; i < numHoteles; ++i) {
        std::cin >> v[i];
    }
    // ordenar por primer criterio
    std::sort(v.begin(), v.end(), std::greater<info>());
    std::cout << v;
    // ordenar por segundo criterio
    std::sort(v.begin(), v.end(), info::ordDistancia());
    std::cout << v;
    // ordenar por tercer criterio
    std::sort(v.begin(), v.end(), info::ordPrecio());
    std::cout << v;
    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.txt
#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;
}
```