Span

El ejercicio propuesto consiste en desarrollar una clase llamada Span que puede almacenar un máximo de N enteros, donde N es un valor de tipo unsigned int proporcionado al constructor. Esta clase debe incluir una función miembro addNumber() para añadir un único número al objeto Span. Si se intenta añadir un nuevo elemento cuando ya se han almacenado N elementos, la función debe lanzar una excepción.

Además, se deben implementar dos funciones miembro: shortestSpan() y longestSpan(). Estas funciones calcularán y devolverán, respectivamente, la diferencia mínima y máxima entre todos los números almacenados en el objeto Span. Si no hay números almacenados o solo hay uno, no se puede determinar ninguna diferencia, por lo que se debe lanzar una excepción en estos casos.

El propósito de aprendizaje de este ejercicio es familiarizarse con el uso de los contenedores y algoritmos de la Biblioteca Estándar de C++ (STL) en el contexto del estándar C++98. Se espera que el estudiante utilice estos componentes de manera efectiva para gestionar la colección de números y realizar las operaciones necesarias para calcular las diferencias solicitadas.

Para abordar este ejercicio en C++98, es recomendable seguir los siguientes pasos:

1. **Definición de la clase Span**: Declarar la clase con un constructor que acepte un parámetro de tipo unsigned int para establecer el tamaño máximo N de la colección de números. Este constructor inicializará los miembros de datos necesarios para almacenar los números y llevar un registro de la cantidad actual de elementos.
2. **Uso de contenedores de la STL**: Aunque el ejercicio permite resolver el problema sin utilizar los contenedores estándar, el objetivo es precisamente emplearlos. Por lo tanto, se recomienda utilizar un contenedor adecuado, como std::vector, para almacenar los números. Este contenedor dinámico permite gestionar automáticamente la memoria y proporciona funciones útiles para manipular los elementos.
3. **Implementación de addNumber()**: Esta función debe verificar si la cantidad actual de elementos ha alcanzado el máximo N. Si es así, lanzará una excepción para indicar que no se pueden añadir más números. De lo contrario, añadirá el nuevo número al contenedor.
4. **Implementación de shortestSpan() y longestSpan()**: Ambas funciones deben verificar si hay suficientes números almacenados (al menos dos) para calcular una diferencia. Si no es así, lanzarán una excepción. Para longestSpan(), se puede utilizar las funciones std::min_element y std::max_element de la STL para encontrar los valores mínimo y máximo en el contenedor, y luego calcular la diferencia entre ellos. Para shortestSpan(), una estrategia eficiente es ordenar una copia del contenedor utilizando std::sort y luego encontrar la menor diferencia entre elementos consecutivos.

5. **Manejo de excepciones**: Es importante definir clases de excepción personalizadas o utilizar las existentes en la STL para manejar los casos en los que no se puedan añadir más números o no se puedan calcular las diferencias debido a una cantidad insuficiente de elementos.
6. **Añadir múltiples números**: Para facilitar la inserción de una gran cantidad de números, se puede implementar una función miembro que acepte un rango de iteradores. Esto permite añadir múltiples elementos en una sola llamada, utilizando las facilidades que ofrecen los contenedores de la STL para manejar rangos de datos.

A continuación, se presentan algunos recursos en que pueden ser útiles para comprender y abordar con éxito este ejercicio:

1. **"Curso de C++ Avanzado. STL. Algoritmos de la STL"**: Este video ofrece una visión detallada sobre los algoritmos disponibles en la Standard Template Library (STL) de C++, incluyendo ejemplos prácticos de su uso.

   Enlace: https://www.youtube.com/watch?v=IebP5c4MABw

2. **"Manejo de excepciones en C++"**: En este video se explican los conceptos fundamentales del manejo de excepciones en C++, incluyendo el uso de try, catch y throw, así como la jerarquía de clases de excepciones.

   Enlace: https://www.youtube.com/watch?v=51QRwCOauHY

3. **"Primeros pasos con Standard Template Library de C++"**: Este recurso introduce la STL de C++, detallando sus características y proporcionando ejemplos prácticos para su comprensión.

   Enlace: https://www.youtube.com/watch?v=uBKzM-MBj18

4. **"Iteradores en C++ - STL"**: Este video aborda el uso de iteradores en la STL de C++, explicando su importancia y cómo se emplean para recorrer contenedores.

   Enlace: https://www.youtube.com/watch?v=3KSnP5FLf8M

5. **"Curso de C++ Avanzado. Uso de excepciones dentro de funciones."**: En este video se analiza cómo se manejan las excepciones dentro de las funciones en C++, incluyendo la propagación de excepciones y el "desenredado de pila".

   Enlace: https://www.youtube.com/watch?v=ogWzE92ReZ4

Estos recursos proporcionan una base sólida para comprender los conceptos clave necesarios para implementar la clase Span de manera efectiva utilizando las características del estándar C++98 y la Biblioteca Estándar de C++.

El código

Explicación detallada del código Span en C++98

El código implementa una clase Span que gestiona una colección de números enteros y proporciona métodos para agregar números y calcular la distancia mínima y máxima entre los valores almacenados. Vamos a desglosar cada aspecto de su funcionamiento.

1. ¿Qué hace este código?

La clase Span permite:

- **Almacenar un conjunto de números enteros hasta un máximo de N elementos**.
- **Agregar números individualmente (addNumber) o en bloque (addNumbers)**.
- **Añadir números aleatorios (addRamNumber)**.
- **Calcular la diferencia mínima (shortestSpan) y máxima (longestSpan) entre los números almacenados**.
- **Imprimir la lista de números (printNumbers)**.
- **Lanzar excepciones en caso de errores como desbordamiento o falta de elementos suficientes**.

Además, la clase maneja **excepciones personalizadas** para situaciones específicas:

- **MaxSizeException** → Ocurre cuando se intenta añadir más números de los permitidos.
- **notEnoughNumbersException** → Ocurre cuando no hay suficientes números almacenados para calcular una diferencia.

El código sigue el **estándar C++98**, por lo que no usa características modernas como std::move, std::initializer_list o auto.

2. Desglose del código

Vamos a analizar cada parte del código para entender qué hace y cómo se implementa en **C++98**.

2.1. Encabezado Span.hpp

Este archivo define la clase Span, sus atributos y métodos.

Librerías incluidas

```
#include <algorithm>  // Para std::sort, std::min_element, std::max_element...
#include <cmath>      // Para std::abs
#include <exception>  // Para std::exception (manejo de excepciones)
#include <iostream>   // Para std::cout y std::endl
#include <stdexcept>  // Para std::out_of_range, std::overflow_error...
#include <stdint.h>   // Para uint32_t (tipos de tamaño fijo)
#include <vector>     // Para std::vector (contenedor dinámico de enteros)
```

- Se usa **std::vector<int>** como estructura de almacenamiento.
- #pragma once evita la inclusión múltiple del archivo.

Definición de la clase Span

```
class Span {
  public:
    Span(uint32_t n);  // Constructor
    Span(Span const &src);  // Constructor de copia
    Span &operator=(Span const &src);  // Operador de asignación
    ~Span();  // Destructor
```

- Se define el **constructor** para inicializar el objeto.
- Se implementa el **constructor de copia** y el **operador de asignación** para manejar correctamente copias de objetos Span (evita problemas de memoria y corrupción de datos).
- El **destructor** es trivial, ya que std::vector gestiona su propia memoria.

Métodos principales

```
void addNumber(int n);  // Agregar un solo número
void addNumbers(std::vector<int>::iterator begin, std::vector<int>::iterator end);  // Agregar varios números con iteradores
void addRamNumber(uint32_t amount);  // Agregar números aleatorios
int shortestSpan();  // Calcular la menor diferencia entre los números almacenados
int longestSpan();  // Calcular la mayor diferencia entre los números almacenados
void printNumbers();  // Imprimir todos los números almacenados
```

- **addNumber(int n)** → Añade un número a la lista (_numbers).
- **addNumbers(iterator begin, iterator end)** → Agrega un rango de números desde iteradores.
- **addRamNumber(uint32_t amount)** → Genera números aleatorios y los añade a la lista.
- **shortestSpan() y longestSpan()** → Calculan la menor y mayor distancia entre valores almacenados.
- **printNumbers()** → Muestra la lista de números almacenados.

Excepciones personalizadas

```
class notEnoughNumbersException : public std::exception {
  public:
    const char *what() const throw();  // Excepción cuando hay pocos números
};

class MaxSizeException : public std::exception {
  public:
    const char *what() const throw();  // Excepción cuando se excede el tamaño máximo
};
```

- **Heredan de std::exception** para definir errores personalizados.
- **what() devuelve un mensaje de error** cuando la excepción es lanzada.

Atributos privados

```
private:
  uint32_t _maxSize;  // Tamaño máximo del Span
  std::vector<int> _numbers;  // Almacena los números
  Span();  // Constructor por defecto (privado, evita creación sin parámetros)
```

- _maxSize establece el **límite máximo de números**.
- _numbers almacena los valores en un **vector dinámico**.
- El constructor **por defecto está deshabilitado** para forzar la inicialización con un tamaño.

2.2. Implementación en Span.cpp

Este archivo contiene la implementación de los métodos definidos en Span.hpp.

Constructor y métodos de copia

```
Span::Span(uint32_t n) : _maxSize(n), _numbers() {};  // Inicializa con tamaño `n`

Span::Span(const Span &src) { *this = src; }  // Constructor de copia

Span &Span::operator=(const Span &src) {  // Asignación por copia
   this->_maxSize = src._maxSize;
   this->_numbers = src._numbers;
   return *this;
}
```

- Se usa la **lista de inicialización** en el constructor.
- En el **constructor de copia**, se usa operator= para evitar duplicación de código.

Agregar números

```
void Span::addNumber(int n) {
   if (this->_numbers.size() >= this->_maxSize)
      throw Span::MaxSizeException();
   this->_numbers.push_back(n);
}
```

- Si se excede el tamaño máximo, lanza una **excepción**.
- Si hay espacio, usa push_back() para agregar el número.

```
void Span::addNumbers(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
   if (this->_numbers.size() + std::distance(begin, end) > this->_maxSize)
      throw Span::MaxSizeException();
   this->_numbers.insert(this->_numbers.end(), begin, end);
}
```

- Se usa std::distance(begin, end) para calcular la cantidad de elementos.
- insert() añade todos los valores de golpe.

Generar números aleatorios

```
void Span::addRamNumber(uint32_t amount) {
   srand(time(NULL));
   try {
```

```
    for (uint32_t i = 0; i < amount; i++)
      this->addNumber(rand());
  } catch (std::exception &e) {
    cout << RED << e.what() << RESET << endl;
  }
}
```

- **srand(time(NULL))** inicializa la semilla para números aleatorios.
- **Se usa rand()** para generar valores aleatorios.
- **Se maneja cualquier excepción** lanzada por addNumber().

Calcular distancias mínimas y máximas

```
int Span::shortestSpan() {
  if (_numbers.size() < 2)
    throw notEnoughNumbersException();
  std::vector<int> vec = _numbers;
  std::sort(vec.begin(), vec.end());
  int min = vec[1] - vec[0];
  for (size_t i = 1; i < vec.size(); i++)
    min = std::min(min, vec[i] - vec[i - 1]);
  return min;
}
```

- Se ordenan los números para facilitar el cálculo.
- Se compara la diferencia entre elementos consecutivos.

```
int Span::longestSpan() {
  if (_numbers.size() < 2)
    throw notEnoughNumbersException();
  return *std::max_element(_numbers.begin(), _numbers.end()) -
*std::min_element(_numbers.begin(), _numbers.end());
}
```

- Se usa **std::min_element y std::max_element** para hallar el mínimo y el máximo.

Conclusión

La clase Span gestiona eficientemente una colección de números enteros y calcula sus diferencias mínima y máxima. Hace un uso correcto de **STL (std::vector, std::sort, std::min_element, std::max_element)** y **manejo de excepciones** en C++98.

3. Explicación del archivo main.cpp

El archivo main.cpp se encarga de probar la clase Span con diferentes escenarios. Hay dos versiones: una propia y otra proporcionada en el enunciado.

3.1. main.cpp (propio)

```
int	main() {
  Span sp = Span(10000);  // Crear un Span con tamaño máximo de 10000

  std::vector<int> numbers;  // Crear un vector para almacenar números
  for (int i = 0; i < 10000; i++)
    numbers.push_back(i);  // Agregar números del 0 al 9999

  sp.addNumbers(numbers.begin(), numbers.end());  // Agregar todos los números al objeto Span

  std::cout << "Shortest span: " << sp.shortestSpan() << std::endl;  // Imprimir la menor distancia
  std::cout << "Longest span: " << sp.longestSpan() << std::endl;  // Imprimir la mayor distancia

  return 0;
}
```

¿Qué hace esta versión de main.cpp?

1. **Crea un objeto Span con un tamaño máximo de 10000.**
2. **Llena un std::vector<int> con números desde 0 hasta 9999.**
3. **Agrega todos estos números al objeto Span de golpe usando addNumbers().**
4. **Calcula e imprime la diferencia mínima (shortestSpan()) y máxima (longestSpan()).**

Resultados esperados

Dado que los números se agregan en orden secuencial (0, 1, 2, ..., 9999):

- La **diferencia mínima** (shortestSpan()) será 1, ya que cada número tiene como vecino el siguiente número consecutivo.
- La **diferencia máxima** (longestSpan()) será 9999 - 0 = 9999.

3.2. main.cpp (proporcionado en el enunciado)

```
int main() {
  Span sp = Span(5);  // Crear un objeto Span con un tamaño máximo de 5

  sp.addNumber(6);
  sp.addNumber(3);
  sp.addNumber(17);
  sp.addNumber(9);
  sp.addNumber(11);

  std::cout << sp.shortestSpan() << std::endl;  // Imprimir la menor distancia
  std::cout << sp.longestSpan() << std::endl;  // Imprimir la mayor distancia

  return 0;
}
```

¿Qué hace esta versión de main.cpp?

1. **Crea un objeto Span con un tamaño máximo de 5.**
2. **Agrega manualmente los números 6, 3, 17, 9 y 11.**
3. **Calcula e imprime la distancia mínima (shortestSpan()) y máxima (longestSpan()).**

Resultados esperados

Los números ingresados son: **[6, 3, 17, 9, 11]**. Al ordenarlos: **[3, 6, 9, 11, 17]**.

- **Diferencia mínima (shortestSpan()):** 6 - 3 = 3
- **Diferencia máxima (longestSpan()):** 17 - 3 = 14

4. Análisis de eficiencia

La eficiencia de los métodos depende de sus operaciones internas:

4.1. shortestSpan()

- **Ordena los números usando std::sort(), que tiene complejidad O(N log N).**
- **Recorre la lista en O(N) para encontrar la diferencia mínima.**
- **Complejidad total: O(N log N).**

4.2. longestSpan()

- **Encuentra el mínimo y el máximo con std::min_element() y std::max_element(), ambos en O(N).**
- **Complejidad total: O(N).**

4.3. addNumber()

- **Añadir un número con push_back() tiene complejidad O(1).**
- **Lanzar una excepción si el vector está lleno también es O(1).**
- **Complejidad total: O(1).**

4.4. addNumbers()

- **Usa std::distance(begin, end) que tiene complejidad O(1).**
- **insert() tiene complejidad O(K), donde K es la cantidad de números a insertar.**
- **Complejidad total: O(K).**

5. Excepciones y manejo de errores

El código maneja errores con excepciones para evitar problemas en tiempo de ejecución.

5.1. Excepción MaxSizeException

Se lanza si se intenta agregar más números de los permitidos:

```
void Span::addNumber(int n) {
  if (this->_numbers.size() >= this->_maxSize)
    throw Span::MaxSizeException();
  this->_numbers.push_back(n);
```

```
}
```

Mensaje de error

```
const char *Span::MaxSizeException::what() const throw() {
  return "Max size reached";
}
```

5.2. Excepción notEnoughNumbersException

Se lanza si se intenta calcular shortestSpan() o longestSpan() sin suficientes elementos:

```
if (_numbers.size() < 2)
  throw notEnoughNumbersException();
```

Mensaje de error

```
const char *Span::notEnoughNumbersException::what() const throw() {
  return "Not enough numbers in the list";
}
```

6. Posibles mejoras

6.1. Optimizar shortestSpan()

- En lugar de copiar y ordenar std::vector, se podría usar std::set<int>, que mantiene los números ordenados automáticamente. Esto reduciría la complejidad de O(N log N) a O(N) para encontrar la diferencia mínima.

```
int Span::shortestSpan() {
  if (_numbers.size() < 2)
    throw notEnoughNumbersException();

  std::set<int> sortedSet(_numbers.begin(), _numbers.end());
  int minSpan = *std::next(sortedSet.begin()) - *sortedSet.begin();

  for (std::set<int>::iterator it = sortedSet.begin(); it != --sortedSet.end(); ++it)
```

```
        minSpan = std::min(minSpan, *std::next(it) - *it);

    return minSpan;
}
```

6.2. Agregar soporte para std::multiset

Un std::multiset<int> permitiría almacenar números duplicados de manera eficiente y mejorar la búsqueda de la distancia mínima.

6.3. Implementar findKthSmallestSpan(int k)

Un nuevo método que retorne la k-ésima diferencia más pequeña entre elementos podría ser útil en algunos escenarios.

7. Conclusión

✅ **La clase Span proporciona una manera eficiente de almacenar y analizar diferencias entre números enteros.**

✅ **Utiliza STL (std::vector, std::sort, std::min_element, std::max_element).**

✅ **Maneja excepciones correctamente (MaxSizeException, notEnoughNumbersException).**

✅ **Es eficiente en operaciones básicas, aunque shortestSpan() puede optimizarse con std::set.**

En resumen, el código está bien estructurado y cumple su propósito.