Mutated abomination

El ejercicio propuesto consiste en crear una clase llamada MutantStack que extienda la funcionalidad de std::stack en C++98, permitiendo la iteración sobre sus elementos mediante iteradores. Aunque std::stack es una estructura de datos útil para gestionar pilas, carece de la capacidad de ser recorrida con iteradores, lo que limita su flexibilidad en ciertas operaciones. El objetivo de este ejercicio es subsanar esta limitación, proporcionando una comprensión más profunda de los contenedores estándar y los iteradores en C++.

Propósito de aprendizaje:

- **Comprender la implementación de contenedores adaptadores:** std::stack es un adaptador de contenedor que utiliza otro contenedor subyacente (por defecto, std::deque) para gestionar sus elementos. Al desarrollar MutantStack, se profundiza en cómo funcionan estos adaptadores y cómo se pueden extender.
- **Familiarizarse con iteradores:** Los iteradores son fundamentales en la manipulación de secuencias en C++. Implementar iteradores en MutantStack refuerza la comprensión de su funcionamiento y uso.
- **Uso de plantillas y herencia:** La creación de MutantStack implica el uso de plantillas para manejar diferentes tipos de datos y la herencia para extender la funcionalidad de std::stack.

Enfoque para abordar el ejercicio en C++98:

1. **Herencia de std::stack:** MutantStack debe heredar de std::stack para reutilizar sus funcionalidades básicas. En C++98, la herencia se declara especificando la clase base en la definición de la nueva clase.
2. **Definición de iteradores:** Para hacer que MutantStack sea iterable, es necesario definir tipos de iteradores que permitan recorrer los elementos. Dado que std::stack no proporciona iteradores directamente, se puede acceder al contenedor subyacente (como std::deque) y utilizar sus iteradores.
3. **Implementación de métodos begin() y end():** Estos métodos deben devolver iteradores al inicio y al final de la estructura de datos, respectivamente. Se implementan accediendo a los métodos begin() y end() del contenedor subyacente.
4. **Uso de plantillas:** Para que MutantStack sea genérico y pueda manejar cualquier tipo de dato, se debe implementar como una plantilla. En C++98, las plantillas se declaran utilizando la sintaxis template<typename T>.
5. **Gestión de archivos de cabecera y de implementación:** Según las reglas del módulo, las declaraciones de las plantillas deben estar en archivos de cabecera (.h o .hpp). Opcionalmente, las implementaciones pueden estar en archivos .tpp, que luego se incluyen en el archivo de cabecera correspondiente.

Recursos recomendados:

Para profundizar en los conceptos clave y facilitar la implementación del ejercicio, se sugieren los siguientes videos en español:

Entiendo que los recursos previamente proporcionados no están disponibles en internet. A continuación, te facilito una nueva selección de videos en español que están actualmente accesibles y que te ayudarán a comprender los conceptos clave necesarios para abordar el ejercicio propuesto:

1. **"Curso C++ No Tan Básico. Clases contenedor"**

   Este video ofrece una visión detallada sobre las clases contenedor en C++, incluyendo su funcionalidad y cómo se implementan.

   Ver video

2. **"Curso de C++ Moderno. STL. La Standard Template Library"**

   En este recurso se introduce la STL (Standard Template Library) de C++, explicando la relación entre contenedores, algoritmos e iteradores, y proporcionando ejemplos prácticos de su uso.

   Ver video

3. **"Iteradores en C++ - STL"**

   Este video se centra en el uso de iteradores en C++, explicando qué son, cómo se declaran y cómo se utilizan para recorrer contenedores de la STL.

   Ver video

4. **"Explicación y ejemplos de plantillas en C++"**

   Aquí se aborda el concepto de plantillas en C++, fundamentales para crear clases y funciones genéricas, con ejemplos prácticos que ilustran su implementación.

   Ver video

5. **"Curso de C++ Avanzado. Plantillas de Clase."**

   Este video forma parte de un curso avanzado de C++ y se enfoca en las plantillas de clase, mostrando cómo crearlas y utilizarlas en diferentes contextos.

   Ver video

Estos recursos están diseñados para proporcionarte una comprensión sólida de los contenedores, iteradores y plantillas en C++, elementos esenciales para implementar la clase MutantStack según lo solicitado en el ejercicio. Proporcionan una base sólida para comprender y abordar con éxito el ejercicio propuesto, reforzando los conceptos clave necesarios para su implementación en C++98.

El código

A continuación se ofrece una explicación detallada, clara y completa de lo que hace el código y cómo lo hace:

1. Archivo: MutantStack.hpp

a) Inclusiones y macros

- **Inclusiones de cabeceras:**
- Se incluyen las cabeceras necesarias para usar funciones de la biblioteca estándar:
    - <cstdlib>: para funciones como rand().
    - <iostream>: para la entrada/salida estándar.
    - <list>: aunque en este archivo no se usa directamente, se incluye para poder compararlo en el test.
    - <stack>: para usar la clase std::stack sobre la que se basará nuestra clase.
- **Definición de macros de color:**

Se definen los siguientes macros para facilitar la impresión de mensajes coloreados en la terminal:

- BLUE, GREEN, RED: para textos en azul, verde y rojo, respectivamente.
- RESET: para reiniciar el color después de imprimir.

Esto se utiliza, por ejemplo, para identificar la llamada a constructores o destructores, haciendo más visual la salida.

- **Uso de directivas using:**

using std::cout; using std::endl;

Esto permite usar cout y endl sin tener que escribir std:: cada vez.

b) Declaración de la clase plantilla MutantStack

- **Plantilla (template):**

La clase se define como plantilla con template <typename T>, lo que permite que MutantStack trabaje con cualquier tipo de dato (por ejemplo, int, float, etc.).

- **Herencia de std::stack<T>:**

class MutantStack : public std::stack<T>

La clase MutantStack hereda de std::stack<T>. Esto significa que hereda todas las funcionalidades básicas de una pila (stack), como push(), pop(), top(), y size().

- **Definición del tipo iterador:**

typedef typename std::stack<T>::container_type::iterator iterator;

Aquí se utiliza el typedef para definir el tipo iterator de MutantStack.

- **Concepto clave:** std::stack es un contenedor adaptador que utiliza internamente otro contenedor (por defecto, std::deque<T>). Este contenedor interno se llama c (que es un miembro protegido) y tiene sus propios iteradores.
- Al definir iterator de esta forma, se "expone" la capacidad iterativa del contenedor subyacente, permitiendo recorrer la pila como si fuera un contenedor "iterable" (en inglés, *iterable container*).

Declaración de métodos:

- **Constructores y Destructor:**

  Se declaran:

  - Constructor por defecto.
  - Constructor de copia.
  - Operador de asignación (copy assignment operator).
  - Destructor.

  Estos métodos incluyen mensajes de salida para indicar cuándo se llaman, utilizando los colores definidos.

Métodos begin() y end():

Se declaran dos métodos que devuelven iteradores:

- iterator begin(); retorna el iterador al inicio del contenedor subyacente.
- iterator end(); retorna el iterador al final del contenedor subyacente.

Esto es lo que permite "iterar" sobre la pila, algo que la clase original std::stack no permite.

c) Implementación de los métodos de MutantStack

- **Constructor por defecto:**

```
template <typename T>
MutantStack<T>::MutantStack() {
  cout << "MutantStack " << GREEN << "constructor" << RESET " called" << endl;
};
```

Imprime un mensaje indicando la construcción del objeto.

- **Constructor de copia:**

```
template <typename T>
MutantStack<T>::MutantStack(const MutantStack &src) {
  *this = src;
  cout << "MutantStack " << GREEN << "copy constructor" << RESET " called" << endl;
};
```

Copia el contenido de otro MutantStack utilizando el operador de asignación y muestra un mensaje.

- **Operador de asignación:**

```
template <typename T>
MutantStack<T> &MutantStack<T>::operator=(const MutantStack &src) {
  cout << "MutantStack " << GREEN << "copy assignment operator" << RESET " called" << endl;
  std::stack<int>::operator=(src);  // Llama al operador= de la clase base std::stack
  return *this;
};
```

  - **Nota importante:** Se llama al operador de asignación de std::stack<int>; lo habitual sería usar std::stack<T>::operator=(src) para que funcione con cualquier tipo T.
  - El método copia los elementos de src en el objeto actual y devuelve una referencia a *this.
- **Métodos begin() y end():**

```
template <typename T>
typename MutantStack<T>::iterator MutantStack<T>::begin() {
  return this->c.begin();
};

template <typename T>
typename MutantStack<T>::iterator MutantStack<T>::end() {
  return this->c.end();
```

```
};
```

Aquí se accede al miembro protegido c (el contenedor subyacente de std::stack) y se devuelven sus iteradores de inicio y fin.

Concepto clave: La utilización de this->c es posible porque c es un miembro protegido de std::stack, permitiendo a las clases derivadas acceder a él.

- **Destructor:**

```
template <typename T>
MutantStack<T>::~MutantStack() {
   cout << "MutantStack " << RED << "destructor" << RESET " called" << endl;
};
```

Imprime un mensaje indicando que se está destruyendo el objeto.

2. Archivo: main.cpp

Se proporcionan dos variantes de main: una propia (con una función de test genérica y su especialización para std::list) y otra comentada (la que se ofrece en la descripción del ejercicio).

a) Función plantilla testContainer

Esta función se define para probar el comportamiento de contenedores que tienen funciones similares a las de una pila.

- **Versión general (para MutantStack y contenedores similares):**

```
template <typename Container>
void testContainer(Container &container) {
   container.push(5);            // Agrega el número 5
   container.push(17);           // Agrega el número 17
   std::cout << "Top element: " << container.top() << std::endl;  // Muestra el elemento en la cima (17)
   container.pop();              // Quita el elemento superior
   std::cout << "Size: " << container.size() << std::endl;       // Muestra el tamaño (1)
   container.push(3);
   container.push(5);
   container.push(737);
   container.push(0);
```

```
  typename Container::iterator it = container.begin();  // Obtiene el iterador al inicio
  typename Container::iterator ite = container.end();    // Obtiene el iterador al final
  ++it;  // Avanza el iterador
  --it;  // Retrocede el iterador
  while (it != ite) {
    std::cout << *it << std::endl;  // Imprime el valor apuntado por el iterador
    ++it;
  }
}
```

Puntos clave:

- Se utilizan funciones propias de una pila (push(), pop(), top(), size()) y, gracias a la extensión de MutantStack, también se pueden usar los métodos begin() y end() para iterar sobre sus elementos.
- La iteración se realiza de forma tradicional: se incrementa el iterador hasta alcanzar el final del contenedor.

Especialización para std::list<int>:

```
template <>
void testContainer(std::list<int> &container) {
  container.push_back(5);
  container.push_back(17);
  std::cout << "Top element: " << container.back() << std::endl;  // En una lista, usamos back()
para obtener el último elemento
  container.pop_back();
  std::cout << "Size: " << container.size() << std::endl;
  container.push_back(3);
  container.push_back(5);
  container.push_back(737);
  container.push_back(0);

  std::list<int>::iterator it = container.begin();
  std::list<int>::iterator ite = container.end();
  ++it;
  --it;
  while (it != ite) {
    std::cout << *it << std::endl;
    ++it;
```

```
    }
}
```

Diferencias y similitudes:

- Se utilizan funciones específicas de std::list (por ejemplo, push_back(), pop_back(), back()).
- El propósito es poder comparar la salida y el comportamiento con el de MutantStack, asegurando que ambos contenedores se comporten de forma similar cuando se recorre su contenido.

b) Función main()

En la función main(), se realizan las siguientes acciones:

1. **Prueba con MutantStack:**

```
cout << BLUE << "Testing MutantStack with stack" << RESET << endl;
MutantStack<int> mstack;
testContainer(mstack);
```

   a. Se crea una instancia de MutantStack<int>.
   b. Se llama a testContainer pasándole esta instancia, lo que ejecuta la serie de operaciones (push, pop, iteración) y muestra los resultados en consola.

2. **Prueba con std::list<int>:**

```
cout << BLUE << "Testing MutantStack with list" << RESET << endl;
std::list<int> list;
testContainer(list);
```

   a. Se crea un objeto std::list<int> y se prueba de la misma manera con la versión especializada de testContainer.
   b. La idea es que, si se sustituye MutantStack por std::list, ambos contenedores producirán salidas equivalentes (en cuanto a la secuencia de elementos), demostrando que MutantStack ha adquirido la capacidad de ser iterado, similar a otros contenedores de la STL.

3. **Comentarios adicionales:**
   a. Al final del código se incluye, de forma comentada, un ejemplo de main() proporcionado en la descripción del ejercicio. Este ejemplo realiza operaciones similares pero únicamente con MutantStack, para probar que su comportamiento es análogo al de un contenedor estándar iterable.

Resumen y Propósito del Código

- **Qué hace el código:**

  El código extiende la funcionalidad de la clase std::stack creando una clase llamada MutantStack que, además de ofrecer todas las funciones propias de una pila, permite iterar sobre sus elementos mediante la definición de los métodos begin() y end(). Esto se logra aprovechando el contenedor subyacente (accesible como this->c).

- **Propósito de aprendizaje:**

  - **Comprender la herencia en C++:** Se extiende una clase estándar para añadir funcionalidad.
  - **Utilización de plantillas:** Permite que la clase trabaje con cualquier tipo de dato.
  - **Exposición de la capacidad iterativa:** Al definir un tipo iterator y métodos begin()/end(), se enseña cómo "sacar" iteradores de contenedores que originalmente no los proporcionan.
  - **Comparación entre contenedores:** Se usa MutantStack y std::list para demostrar que, con las modificaciones adecuadas, se puede lograr un comportamiento similar entre diferentes estructuras de datos de la STL.

Este código es un excelente ejemplo de cómo ampliar las capacidades de los contenedores estándar y aprovechar la flexibilidad de la STL en C++, incluso bajo el estándar C++98.