The Job Of Your Dreams

Hoy es tu primer día en GlobalBanksters United. Tu primera tarea es recrear un archivo perdido. Algo salió mal y un archivo fuente fue eliminado por error.

Tus compañeros desarrolladores te dan un montón de archivos. Al compilar tests.cpp, se revela que el archivo faltante es Account.cpp. Por suerte, el archivo de cabecera Account.hpp se guardó. También hay un archivo de registro. Tal vez puedas usarlo para entender cómo se implementó la clase Account.

Comienzas a recrear el archivo Account.cpp. En solo unos minutos, codificas algunas líneas de puro C++ impresionante. Después de un par de compilaciones fallidas, tu programa pasa las pruebas. Su salida coincide perfectamente con la guardada en el archivo de registro (excepto por las marcas de tiempo, que obviamente diferirán ya que las pruebas guardadas en el archivo de registro se ejecutaron antes de que fueras contratado).

Ejercicio: 02 Directorio de entrega: ex02 Archivos por entregar: Makefile, Account.cpp, Account.hpp, tests.cpp Funciones prohibidas: Ninguna

Account.hpp

Encabezado del archivo

#pragma once e #ifndef __ACCOUNT_H__ / #define __ACCOUNT_H__ son directivas de preprocesador que aseguran que el contenido del archivo solo se incluya una vez durante la compilación, evitando problemas de redefinición.

¿Qué hace exactamente #pragma once?

#pragma once es una directiva del preprocesador que le indica al compilador que incluya un archivo de cabecera solo una vez durante la compilación de un programa.

Pero ¿cómo logra esto?

- Marca de tiempo: El compilador suele asociar a cada archivo un "timestamp" o marca de tiempo que indica la última modificación del archivo. Cuando se incluye un archivo por primera vez, el compilador registra esta marca de tiempo.
- Comparación: Al encontrar una nueva directiva #pragma once en otro archivo, el compilador compara la marca de tiempo del archivo actual con la marca de tiempo que tiene registrada.
- Inclusión condicional: Si la marca de tiempo es diferente, el compilador asume que el archivo ha sido modificado y lo incluye de nuevo. Sin embargo, si la marca de tiempo es la misma, significa que el archivo ya ha sido incluido en esa unidad de traducción y el compilador lo omite.

¿Por qué "once"?

- El término "once" (una vez) se utiliza porque la intención de esta directiva es clara: garantizar que el contenido de un archivo de cabecera se incluya solo una vez en una unidad de traducción. Al utilizar la palabra "once", se enfatiza esta limitación de una sola inclusión.
- Ventajas de usar #pragma once:
- Simplicidad: Es más fácil de escribir y entender que las guardias de inclusión tradicionales.
- Eficiencia: Al evitar la relectura innecesaria de archivos, puede mejorar ligeramente el tiempo de compilación, especialmente en proyectos grandes.
- Menos código: Se requiere menos código en comparación con las guardias de inclusión tradicionales.

Desventajas y consideraciones:

- No estándar: Aunque es ampliamente soportada por los compiladores modernos, #pragma once no es parte del estándar C++. Esto significa que, en teoría, podría haber compiladores que no la reconozcan.
- Dependencia del compilador: El comportamiento exacto de #pragma once puede variar ligeramente entre diferentes compiladores.
- Limitaciones en sistemas de construcción complejos: En algunos sistemas de construcción muy complejos, #pragma once podría no funcionar como se espera.

En resumen:

#pragma once es una directiva conveniente y eficiente para evitar la inclusión múltiple de archivos de cabecera. Sin embargo, es importante ser consciente de sus limitaciones y utilizarla con precaución en entornos de desarrollo específicos.

¿Cuándo usar #pragma once?

En la mayoría de los casos, #pragma once es una excelente opción para evitar problemas de inclusión múltiple. Es especialmente útil en proyectos de tamaño mediano y grande, donde la gestión de dependencias entre archivos puede volverse compleja.

¿Cuándo usar guardias de inclusión tradicionales?

En algunos casos, como en proyectos altamente portables o en entornos donde no se puede garantizar la compatibilidad con #pragma once, es preferible utilizar guardias de inclusión tradicionales.

En conclusión:

#pragma once es una herramienta poderosa y útil para organizar y mantener el código C++. Al comprender cómo funciona y sus limitaciones, puedes tomar decisiones informadas sobre cuándo y cómo utilizarla en tus proyectos.

public:

La sección public contiene miembros y métodos que son accesibles desde fuera de la clase.

¿Qué significa public en C++?

En C++, public es un especificador de acceso que determina cómo se pueden acceder a los miembros (atributos y métodos) de una clase desde otras partes del programa. Cuando declaramos un miembro de una clase como public, estamos diciendo que:

- Es accesible desde cualquier parte del programa: Esto significa que cualquier código puede acceder a ese miembro, ya sea desde otra clase, una función global o incluso desde el main del programa.
- Forma parte de la interfaz pública de la clase: La interfaz pública de una clase define cómo se puede interactuar con los objetos de esa clase. Los miembros públicos son los "servicios" que la clase ofrece al mundo exterior.

¿Por qué usamos public?

- Compartir funcionalidad: Al hacer que los miembros sean públicos, permitimos que otras partes del programa utilicen la funcionalidad de nuestra clase.
- Crear interfaces claras: La parte pública de una clase define su interfaz, es decir, cómo se debe usar.
- Facilitar la reutilización de código: Las clases con interfaces públicas bien definidas son más fáciles de reutilizar en diferentes partes de un programa.

En resumen:

public es una palabra clave fundamental en C++ que nos permite controlar el acceso a los miembros de una clase. Al marcar un miembro como public, lo estamos haciendo accesible desde cualquier parte del programa, lo que lo convierte en parte de la interfaz pública de la clase.

Tipos y métodos estáticos:

typedef Account t;

Esta línea crea un alias para el nombre de la clase Account. Ahora, en lugar de escribir Account, puedes usar t para referirte a la clase. Por ejemplo:

```
t cuenta1(100);        // Esto crea una cuenta con un depósito inicial de 100
```

¿Qué es un typedef en C++?

typedef es una palabra clave en C++ que nos permite crear sinónimos o alias para tipos de datos existentes. Es decir, podemos asignar un nuevo nombre a un tipo de dato ya

definido. Esto puede ser útil por varias razones:

- Simplificar nombres largos: Si un tipo de dato tiene un nombre muy largo o complejo, podemos crear un alias más corto y fácil de recordar.
- Aumentar la legibilidad: Al utilizar alias significativos, podemos hacer que nuestro código sea más fácil de entender, ya que los nombres de los tipos reflejarán su propósito.
- Abstraer detalles de implementación: Podemos ocultar la implementación interna de un tipo de dato detrás de un alias, haciendo que nuestro código sea más modular y fácil de mantener.

¿Por qué usar typedef en este contexto?

- Simplificar la sintaxis: Al usar t en lugar de Account, el código se vuelve más conciso y fácil de leer, especialmente cuando se trabaja con tipos de datos complejos o nombres de clases largos.
- Aumentar la reutilización: Si necesitas crear muchos objetos de tipo Account, usar t puede ahorrarte tiempo y reducir la posibilidad de errores tipográficos.
- Mejorar la legibilidad: Si el nombre Account no es muy descriptivo o si quieres enfatizar un aspecto particular de la clase, puedes crear un alias más significativo. Por ejemplo, si Account representa una cuenta bancaria, podrías usar t como alias de BankAccount.

Account(int initial_deposit);

Este es el constructor de la clase. Se utiliza para crear un nuevo objeto Account con un depósito inicial.

¿Qué es un constructor?

Imagina que estás construyendo una casa. Necesitas ciertos materiales y pasos para que la casa esté lista para habitar, ¿verdad? En programación, un constructor es como el plano y los materiales que necesitas para "construir" un objeto. En este caso, estamos construyendo un objeto de tipo Account (cuenta bancaria).

¿Qué hace este constructor en particular?

- Account: Este es el nombre de la clase. Indica que estamos creando un nuevo objeto que pertenece a la clase Account.
- (): Los paréntesis después del nombre de la clase definen el constructor. Es aquí donde se colocan los parámetros que el constructor necesita para crear el objeto.
- int initial_deposit: Este es un parámetro de tipo entero (int). Representa el depósito inicial que se realizará en la cuenta cuando se cree.

En resumen:

Esta línea de código define un constructor para la clase Account. Este constructor espera recibir un valor entero que representa el depósito inicial. Cuando se crea un nuevo objeto de tipo Account, se llama a este constructor y se le pasa el valor del depósito inicial. Este valor se utilizará para inicializar el saldo de la cuenta recién creada.

Ejemplo:

```
Account miCuenta(1000); // Crea una nueva cuenta con un depósito inicial de 1000
```

En este ejemplo:

- miCuenta es el nombre que le damos al nuevo objeto de tipo Account.
- 1000 es el valor que se pasa al constructor, indicando que queremos que la cuenta tenga un saldo inicial de 1000.

¿Por qué es importante el constructor?

- Inicialización: Garantiza que los objetos se creen en un estado válido. En este caso, asegura que cada cuenta tenga un saldo inicial definido.
- Personalización: Permite crear objetos con diferentes características iniciales, en función de los parámetros que se le pasen al constructor.

En conclusión:

El constructor Account(int initial_deposit); es el encargado de crear nuevos objetos de tipo Account y de establecer su estado inicial. Es una parte fundamental de la clase Account y define cómo se crean las cuentas bancarias en este programa.

~Account(void);

Este es el destructor de la clase. Se ejecuta automáticamente cuando un objeto Account es destruido, liberando los recursos que pudiera estar utilizando.

- ~Account: Esta parte del nombre indica que se trata de un destructor. Los destructores en C++ son funciones especiales que se ejecutan automáticamente cuando un objeto de una clase deja de existir (por ejemplo, cuando sale de su ámbito o cuando se elimina explícitamente con delete). El nombre del destructor siempre coincide con el nombre de la clase, precedido por: ~.
- (void): Esto indica que el destructor no toma ningún argumento.

¿Qué hace el destructor?

La principal función de un destructor es realizar las tareas de limpieza necesarias antes de que un objeto sea destruido. En el contexto de una clase como Account, esto podría incluir:

- Liberar recursos: Si la clase Account hubiera asignado memoria dinámicamente (por ejemplo, para almacenar un historial de transacciones), el destructor sería el lugar adecuado para liberar esa memoria utilizando delete.
- Cerrar archivos: Si la clase estuviera trabajando con archivos, el destructor podría cerrar esos archivos para evitar fugas de recursos.
- Notificar a otros objetos: Si la clase Account estuviera relacionada con otros objetos, el destructor podría notificarles que la cuenta está siendo destruida.

Por qué es importante el destructor en la clase Account:

- Gestión de recursos: Garantiza que los recursos asignados a una cuenta sean liberados correctamente cuando la cuenta ya no se necesite, evitando así fugas de memoria.
- Consistencia de datos: Puede realizar operaciones de limpieza para mantener la integridad de los datos, como actualizar contadores globales o eliminar registros de la cuenta en una base de datos.
- Prevención de errores: Un destructor bien implementado puede ayudar a prevenir errores de programación y mejorar la estabilidad de la aplicación.

En resumen:

El destructor ~Account(void) es una función especial que se ejecuta automáticamente cuando un objeto de la clase Account es destruido. Su propósito es garantizar que los recursos asociados con ese objeto sean liberados de manera adecuada y que el estado del programa se mantenga consistente.

En conclusión:

Los destructores son una parte esencial de la gestión de la vida útil de los objetos en C++. Garantizan que los recursos sean liberados correctamente y que el programa se mantenga en un estado consistente. En el caso de la clase Account, el destructor ~Account(void) se encarga de realizar las tareas de limpieza necesarias cuando una cuenta ya no es necesaria.

Acceder a información estática:

¿Qué son los métodos estáticos y por qué los usamos en la clase Account?

Imagina la clase Account como un molde para crear objetos que representen cuentas bancarias individuales. Cada objeto creado a partir de este molde sería una cuenta específica con su propio saldo, número de cuenta, etc.

Los métodos estáticos, sin embargo, no están asociados a una cuenta en particular, sino a la clase Account en sí misma. Son como herramientas o funciones generales que nos permiten obtener información o realizar acciones relacionadas con todas las cuentas, sin necesidad de crear una instancia de una cuenta concreta.

¿Para qué sirven los métodos estáticos en este contexto?

En el caso de la clase Account, los métodos estáticos se utilizan para:

Obtener información global sobre todas las cuentas:

- getNbAccounts(): Nos dice cuántas cuentas han sido creadas en total.
- getTotalAmount(): Suma el saldo de todas las cuentas y nos da un total.
- getNbDeposits(): Cuenta el número total de depósitos realizados en todas las cuentas.
- getNbWithdrawals(): Cuenta el número total de retiros realizados en todas las cuentas.

Realizar operaciones que afectan a todas las cuentas:

- displayAccountsInfos(): Muestra un resumen de todas las cuentas, como el número total de cuentas, el saldo total, etc.

¿Por qué son estáticos y no pertenecen a una instancia?

- Información global: La información que proporcionan estos métodos es relevante para todas las cuentas, no solo para una en particular. Por ejemplo, el número total de cuentas es un valor que se comparte entre todas las instancias de la clase.
- Acceso sin instancia: No necesitamos crear una cuenta específica para saber cuántas cuentas hay en total o cuál es el saldo total depositado. Podemos llamar a estos métodos directamente desde la clase Account.
- Optimización: Al ser métodos estáticos, no necesitan acceder a los datos de una instancia en particular, lo que puede mejorar el rendimiento en algunas situaciones.

Métodos de instancia

¿Qué son los métodos de instancia?

En programación orientada a objetos, un método de instancia es una función que pertenece a un objeto específico de una clase. Esto significa que para llamar a un método de instancia, necesitas tener una instancia (un objeto) de esa clase. Los métodos de instancia operan sobre los datos (atributos) de ese objeto específico.

En el código de la clase Account:

Los métodos de instancia son aquellos que no están marcados como static. Estos métodos operan sobre los datos de una cuenta individual. Por ejemplo:

- makeDeposit(int deposit);: Este método de instancia se utiliza para depositar una cierta cantidad de dinero en una cuenta específica. Cuando llamas a este método sobre un objeto Account, el depósito se realiza solo en esa cuenta particular.
- makeWithdrawal(int withdrawal);: Este método de instancia se utiliza para retirar una cierta cantidad de dinero de una cuenta específica.
- checkAmount(void) const;: Este método de instancia devuelve el saldo actual de una cuenta específica.
- displayStatus(void) const;: Este método de instancia muestra información detallada sobre el estado de una cuenta específica, como el saldo actual.

¿Por qué usamos métodos de instancia?

- Encapsulación: Los métodos de instancia permiten encapsular la lógica de manipulación de los datos de un objeto dentro de la clase. Esto significa que el estado interno de un objeto se mantiene privado y solo se puede modificar a través de los métodos públicos de la clase.
- Reutilización: Los métodos de instancia pueden ser reutilizados para realizar operaciones similares en diferentes objetos de la misma clase.
- Flexibilidad: Los métodos de instancia pueden tomar argumentos y devolver valores, lo que permite personalizar su comportamiento.

Ejemplo:

```
Account cuenta1(1000);      // Creamos una cuenta con 1000 euros
cuenta1.makeDeposit(500);          // Depositamos 500 euros en la cuenta1
int saldo = cuenta1.checkAmount();         // Obtenemos el saldo actual de la cuenta1
```

¿Cuál es la diferencia entre métodos de instancia y métodos estáticos?

Los métodos estáticos pertenecen a la clase en sí misma, no a una instancia específica. Se pueden llamar sin necesidad de crear un objeto de la clase. Los métodos estáticos suelen utilizarse para operaciones que no requieren acceder a los datos de un objeto en particular, como, por ejemplo, contar el número total de cuentas o calcular un valor global.

En resumen:

Los métodos de instancia son funciones que operan sobre los datos de un objeto específico de una clase. Son esenciales para encapsular la lógica de un objeto y proporcionar una interfaz clara para interactuar con él. En el caso de la clase Account, los métodos de instancia permiten realizar operaciones como depósitos, retiros y consultas de saldo sobre cuentas individuales.

Private:

Contiene miembros y métodos que solo son accesibles desde dentro de la clase.

Estas variables estáticas mantienen el estado compartido entre todas las instancias de Account.

- _nbAccounts: Número total de cuentas.
- _totalAmount: Saldo total de todas las cuentas.
- _totalNbDeposits: Número total de depósitos realizados en todas las cuentas.
- _totalNbWithdrawals: Número total de reintegros realizados en todas las cuentas.

Imagina una caja fuerte. Todo lo que está dentro de esa caja fuerte es privado, solo tú tienes acceso a su contenido. En programación, la sección private de una clase funciona de manera similar.

¿Qué contiene?

Atributos (variables) y métodos (funciones) que solo pueden ser utilizados dentro de la propia clase. Es decir, el código externo a la clase no puede acceder directamente a estos elementos.

¿Por qué es útil?

- Encapsulamiento: La sección private es fundamental para el encapsulamiento, un principio clave de la programación orientada a objetos. El encapsulamiento consiste en ocultar los detalles internos de una clase y exponer solo una interfaz pública (la parte public). Esto hace que el código sea más fácil de entender, mantener y reutilizar.
- Protección de datos: Al hacer los atributos privados, se evita que sean modificados accidentalmente desde fuera de la clase, lo que garantiza la integridad de los datos.
- Abstracción: La sección private permite crear abstracciones, es decir, modelos simplificados de objetos del mundo real. Por ejemplo, en una clase CuentaBancaria, el saldo podría ser un atributo privado, ya que no es necesario que el usuario sepa cómo se almacena internamente.

Ejemplo:

```
class CuentaBancaria {
private:
  double saldo; // Saldo de la cuenta

public:
  void depositar(double cantidad) {
    saldo += cantidad;
  }

  void retirar(double cantidad) {
    if (saldo >= cantidad) {
      saldo -= cantidad;
```

```
        } else {
            // Manejar error: fondos insuficientes
        }
    }
};
```

En este ejemplo:

- saldo es privado: solo los métodos de la clase CuentaBancaria pueden acceder y modificar el saldo.
- depositar y retirar son públicos: cualquier código puede llamar a estos métodos para realizar operaciones en la cuenta.

Diferencias entre public y private:

Característica	public	private
Accesibilidad	Accesible desde cualquier parte del programa.	Solo accesible desde dentro de la clase.
Propósito	Define la interfaz de la clase, es decir, como se puede interactuar con ella.	Implementa la lógica interna de la clase, ocultando los detalles de implementación.
Encapsulamiento	Forma parte la interfaz pública	Contribuye al encapsulamiento, ocultando los detalles de implementación.

Métodos estáticos

_displayTimestamp(): Método estático que muestra una marca de tiempo. Es privado, por lo que solo puede ser llamado desde dentro de la clase.

Variables de instancia

Estas variables mantienen el estado de una instancia individual de Account.

- _accountIndex: identificador de la cuenta.
- _amount: Saldo actual en la cuenta.
- _nbDeposits: Número de depósitos realizados en esta cuenta.
- _nbWithdrawals: Número de retiros realizados en esta cuenta.

Constructor privado

Account(void): Constructor por defecto privado, lo que significa que no se puede crear una instancia de Account sin un depósito inicial.

La última parte del código es un bloque de comentarios que contiene configuraciones específicas para el editor Vim y posiblemente otros editores que respetan estas configuraciones. Estas configuraciones ayudan a mantener un estilo de codificación consistente, como el uso de tabulaciones, el ancho de las líneas de texto y la columna de comentarios. También incluye la ruta del archivo para referencia rápida.

tests.cpp

#include <vector>:

- La biblioteca <vector> proporciona la plantilla para la clase std::vector, que es un contenedor de datos que permite almacenar una colección de elementos de cualquier tipo.
- Los vectores ofrecen una forma eficiente de manejar listas dinámicas de elementos, ya que pueden crecer o reducirse de tamaño según sea necesario.

#include <algorithm>:

- La biblioteca <algorithm> proporciona una amplia gama de algoritmos genéricos, como sort, find, copy, transform, entre otros.
- Estos algoritmos pueden ser utilizados para realizar operaciones sobre diferentes tipos de contenedores, como vectores, listas y arrays.
- En este código, se utiliza el algoritmo std::for_each, que aplica una función a cada elemento de un rango de elementos.

#include <functional>:

- La biblioteca <functional> proporciona soporte para objetos funcionales, como functores y adaptadores de funciones.
- En este código, se utiliza std::mem_fun_ref, que es un adaptador de función que permite pasar un método miembro de una clase como argumento a un algoritmo genérico como std::for_each.

#include "Account.hpp":

- Este archivo contiene la definición de la clase Account, que representa una cuenta bancaria.
- Al incluir este archivo, se le da al código acceso a la definición de la clase Account y sus miembros (atributos y métodos).

En resumen:

Estas directivas de inclusión proporcionan al código acceso a las herramientas y clases necesarias para:

- Crear y manipular vectores: Para almacenar las cuentas, los depósitos y los retiros.

- Utilizar algoritmos genéricos: Como std::for_each para aplicar operaciones a todos los elementos de un vector.
- Trabajar con métodos miembro de la clase Account: Para realizar operaciones sobre las cuentas bancarias.

Este código C++ demuestra el uso de la clase Account (se asume que está definida en el archivo "Account.hpp") para simular el comportamiento de una serie de cuentas bancarias.

1. Inicialización:

   Se crean tres vectores: accounts para almacenar objetos de la clase Account, deposits para almacenar cantidades a depositar y withdrawals para almacenar cantidades a retirar.

   Un vector es una estructura de datos muy utilizada en C++ que nos permite almacenar una colección de elementos del mismo tipo. Imagina un vector como una lista ordenada de cajas, donde cada caja contiene un elemento.

   Características principales de un vector:

   - Ordenado: Los elementos se almacenan en un orden específico, lo que significa que puedes acceder a ellos por su índice (la posición que ocupan en la lista).
   - Dinámico: Un vector puede crecer o encogerse a medida que agregas o eliminas elementos. No necesitas especificar de antemano el tamaño exacto que tendrá.
   - Acceso aleatorio: Puedes acceder a cualquier elemento del vector de forma directa utilizando su índice.
   - Homogéneo: Todos los elementos de un vector deben ser del mismo tipo (por ejemplo, todos enteros, todas cadenas de texto, etc.).

   ¿Para qué se utilizan los vectores?

   Los vectores son extremadamente útiles para:

   - Almacenar colecciones de datos: Por ejemplo, una lista de nombres, una lista de números, una lista de objetos de una clase, etc.
   - Realizar operaciones sobre conjuntos de datos: Puedes recorrer todos los elementos de un vector, buscar elementos específicos, ordenar los elementos, etc.
   - Pasar múltiples valores a funciones: Puedes pasar un vector completo como argumento a una función para procesar varios elementos a la vez.

   En el código que nos ocupa, los vectores se utilizan para:

   - accounts: Almacena objetos de la clase Account, representando cada una de las cuentas bancarias.
   - deposits y withdrawals: Almacenan las cantidades a depositar y retirar, respectivamente, para cada cuenta.

Ventajas de usar vectores:

- Facilidad de uso: La biblioteca estándar de C++ proporciona muchas funciones y métodos para trabajar con vectores, lo que facilita su manejo.
- Flexibilidad: Los vectores se adaptan dinámicamente a la cantidad de datos que necesitas almacenar.
- Eficiencia: En general, los vectores ofrecen un buen rendimiento en términos de tiempo de acceso y modificación de elementos.

En resumen:

Los vectores son una herramienta fundamental en C++ para trabajar con colecciones de datos. Su versatilidad y facilidad de uso los convierten en una estructura de datos muy popular en la programación.

typedef std::vector<Account::t> accounts_t;

- typedef: Esta palabra clave en C++ se utiliza para crear un alias o sinónimo para un tipo de datos existente.
- std::vector<Account::t>: Representa un tipo de dato: un vector (una colección ordenada) de elementos del tipo Account::t.
- Account::t es un alias para la clase Account (definido en el archivo "Account.hpp"). Esto significa que std::vector<Account::t> es un vector de objetos de la clase Account.
- accounts_t: Este es el nombre del alias que estamos creando. A partir de este momento, en lugar de escribir std::vector<Account::t>, podemos usar simplemente accounts_t. Esto hace que el código sea más legible y conciso.

typedef std::vector<int> ints_t;

Análogamente, se crea un alias llamado ints_t para representar un std::vector<int>, es decir, un vector de números enteros.

typedef std::pair<accounts_t::iterator, ints_t::iterator> acc_int_t;

- std::pair es un tipo de datos de la biblioteca estándar de C++ que permite empaquetar dos valores diferentes en una sola entidad. Los valores pueden ser de distinto tipo.
- accounts_t::iterator es un tipo de dato que representa un iterador para el vector accounts_t. Un iterador es como un puntero que permite recorrer los elementos de un contenedor (en este caso, el vector).
- ints_t::iterator es un tipo de dato que representa un iterador para el vector ints_t.
- std::pair<accounts_t::iterator, ints_t::iterator> crea un par donde el primer elemento es un iterador de accounts_t y el segundo elemento es un iterador de ints_t.
- acc_int_t es el alias para este tipo de par, lo que facilita su uso posterior en el código.

En resumen:

Estas líneas de código utilizan la palabra clave typedef para crear alias para tipos de datos más complejos, lo que hace que el código sea más legible y fácil de mantener. Estos alias simplifican la declaración y el uso de vectores de objetos Account, vectores de enteros y pares de iteradores, mejorando la claridad y reduciendo la repetición de código.

Cada vector se inicializa con un conjunto de valores predefinidos.

Declaración de arreglos constantes:

int const amounts[] = { 42, 54, 957, 432, 1234, 0, 754, 16576 };:

- Se declara un arreglo constante de enteros llamado amounts.
- const indica que los valores del arreglo no pueden ser modificados después de su inicialización.
- El arreglo se inicializa con los valores: 42, 54, 957, 432, 1234, 0, 754, 16576.
- int const d[] = { 5, 765, 564, 2, 87, 23, 9, 20 }; y int const w[] = { 321, 34, 657, 4, 76, 275, 657, 7654 };:
- Se declaran de manera similar otros dos arreglos constantes de enteros llamados d (para depósitos) y w (para reintegros), cada uno con sus respectivos valores.

Cálculo del tamaño de los arreglos:

size_t const amounts_size(sizeof(amounts) / sizeof(int));:

- sizeof(amounts) calcula el tamaño en bytes del arreglo amounts completo.
- sizeof(int) calcula el tamaño en bytes de un único elemento entero. La palabra clave sizeof en C++ es un operador que se utiliza para determinar el tamaño en bytes de un tipo de dato o de una variable.

  Sintaxis:

  - sizeof(type) donde type es un tipo de dato, como int, float, double, etc.
  - sizeof(variable) donde variable es una variable de cualquier tipo.

  Propósito:

  - sizeof devuelve el tamaño en bytes del tipo de dato o de la variable especificada.
  - Es útil para determinar cuánto espacio en memoria ocupa un tipo de dato o una variable.

- La división sizeof(amounts) / sizeof(int) determina el número de elementos en el arreglo amounts.

- size_t es un tipo entero sin signo utilizado para representar tamaños.
- size_t const d_size(sizeof(d) / sizeof(int)); y size_t const w_size(sizeof(w) / sizeof(int));:
- Se calculan de manera similar los tamaños de los arreglos d y w.

Creación de vectores:

accounts_t accounts(amounts, amounts + amounts_size);:

- Se crea un vector de objetos Account llamado accounts.
- accounts_t es el alias creado anteriormente para std::vector<Account::t>.
- El vector accounts se inicializa con los valores del arreglo amounts.
- amounts y amounts + amounts_size representan el inicio y el final del arreglo amounts, respectivamente.
- ints_t deposits(d, d + d_size); y ints_t deposits(w, w + w_size);:
- Se crean de manera similar los vectores deposits e withdrawals a partir de los arreglos d y w, respectivamente.

Obtención de iteradores:

accounts_t::iterator acc_begin = accounts.begin();:

- accounts.begin() devuelve un iterador que apunta al primer elemento del vector accounts.

  La función miembro .begin() es una función de los contenedores de la biblioteca estándar de C++ (como std::vector, std::list, std::map, etc.) que devuelve un iterador al primer elemento del contenedor. Vamos a desglosar su uso y propósito.

  Sintaxis:

  - container.begin(), donde container es una instancia de un contenedor de la biblioteca estándar de C++.

  Propósito:

  - .begin() devuelve un iterador que apunta al primer elemento del contenedor.
  - Este iterador se puede utilizar para recorrer el contenedor desde el principio hasta el final.

  Tipo de retorno:

  - El tipo de retorno de .begin() es un iterador. Para contenedores como std::vector, el tipo de iterador es std::vector<T>::iterator, donde T es el tipo de los elementos almacenados en el vector.

- acc_begin almacena este iterador.

accounts_t::iterator acc_end = accounts.end();:

- accounts.end() devuelve un iterador que apunta al elemento "más allá del final" del vector accounts.

  La función miembro .end() es una función como .begin(), forma parte de los contenedores de la biblioteca estándar de C++. En este caso, devuelve un iterador al elemento que sigue al último elemento del contenedor. Vamos a desglosar su uso y propósito.

  Sintaxis:

  - container.end(), donde container es una instancia de un contenedor de la biblioteca estándar de C++.

  Propósito:

  - .end() devuelve un iterador que apunta al elemento que sigue al último elemento del contenedor.
  - Este iterador no apunta a un elemento válido del contenedor, pero se utiliza como un marcador para indicar el final del contenedor.
  - Es comúnmente utilizado en bucles para determinar cuándo se ha alcanzado el final del contenedor.

  Tipo de retorno:

  - El tipo de retorno de .end() es un iterador. Para contenedores como std::vector, el tipo de iterador es std::vector<T>::iterator, donde T es el tipo de los elementos almacenados en el vector.

- acc_end almacena este iterador.

De manera similar, se obtienen los iteradores dep_begin, dep_end, wit_begin y wit_end para los vectores deposits y withdrawals.

En resumen, este fragmento de código:

- Define e inicializa tres arreglos constantes de enteros: amounts, d y w.
- Calcula el tamaño de cada arreglo.
- Crea tres vectores: accounts (de objetos Account), deposits y withdrawals, utilizando los arreglos constantes como fuente de datos.
- Obtiene iteradores para el inicio y el final de cada vector, lo que será útil para recorrerlos posteriormente.

2. Información inicial:

Account::displayAccountsInfos();: Se llama a un método estático de la clase Account para mostrar información general sobre todas las cuentas existentes (número total de cuentas, cantidad total de dinero, etc.).

std::for_each(...): Se utiliza la función std::for_each para iterar sobre cada cuenta en el vector accounts y llamar al método displayStatus() de cada cuenta para mostrar su estado inicial.

En concreto, std::for_each es una función de la biblioteca estándar de C++ , que se encuentra en el encabezado <algorithm>, y que se utiliza para aplicar una función a cada elemento de un rango especificado.

Sintaxis: std::for_each(first, last, func);

- first: Un iterador al primer elemento del rango.
- last: Un iterador al elemento que sigue al último elemento del rango.
- func: Una función o un objeto funcional que se aplicará a cada elemento del rango.

3. Realizar depósitos:

Se utiliza un bucle for para iterar simultáneamente sobre los vectores accounts y deposits.

En cada iteración, se realiza un depósito en la cuenta correspondiente utilizando el método makeDeposit() de la clase Account y la cantidad de depósito correspondiente del vector deposits.

- acc_int_t it(acc_begin, dep_begin); crea una variable it que es un par de iteradores. Este par de iteradores se inicializa con acc_begin y dep_begin, que son iteradores al primer elemento de los contenedores de cuentas y depósitos, respectivamente.
- La función miembro .second proviene de la clase plantilla std::pair de la biblioteca estándar de C++. Recordemos que, std::pair es una plantilla de clase que permite almacenar dos valores como un solo objeto. Los valores pueden ser de diferentes tipos. Los miembros de std::pair son first y second, que representan el primer y el segundo valor del par, respectivamente.
- La línea ++(it.first), ++(it.second) se utiliza para avanzar ambos iteradores (it.first y it.second) al siguiente elemento en sus respectivos contenedores en cada iteración del bucle for. Esto permite recorrer simultáneamente dos contenedores diferentes, realizando operaciones paralelas en los elementos correspondientes de cada contenedor.
  - it.first:
    - it.first es el primer iterador del par, que apunta a un elemento en el contenedor de cuentas.
  - *(it.first):
    - *(it.first) desreferencia el iterador it.first, obteniendo una referencia al objeto Account al que apunta el iterador.
  - it.second:
    - it.second es el segundo iterador del par, que apunta a un elemento en el contenedor de depósitos.
  - *(it.second):

- *(it.second) desreferencia el iterador it.second, obteniendo el valor del depósito al que apunta el iterador.

- (*(it.first)).makeDeposit(*(it.second)):
  - (*(it.first)).makeDeposit(*(it.second)) llama al método en el objeto Account makeDeposit referenciado por *(it.first), pasando el valor *(it.second) como argumento.

  Esto realiza un depósito en la cuenta apuntada por it.first usando el valor apuntado por it.second.

Básicamente, este fragmento de código realiza depósitos en múltiples cuentas utilizando dos contenedores: uno para las cuentas (accounts_t) y otro para los depósitos (ints_t). El bucle for itera sobre ambos contenedores simultáneamente, utilizando un par de iteradores (acc_int_t). En cada iteración, se llama al método makeDeposit en la cuenta actual, pasando el valor del depósito correspondiente. El bucle continúa hasta que se alcanzan los finales de ambos contenedores.

4. Mostrar información después de los depósitos:

   Se vuelve a llamar a Account::displayAccountsInfos() para mostrar la información actualizada sobre todas las cuentas.

   Se vuelve a utilizar std::for_each para mostrar el estado de cada cuenta individualmente.

5. Realizar reintegros:

   Se utiliza otro bucle for para iterar simultáneamente sobre los vectores accounts y withdrawals.

   En cada iteración, se intenta realizar un retiro en la cuenta correspondiente utilizando el método makeWithdrawal() de la clase Account y la cantidad a retirar del vector withdrawals.

6. Mostrar información final:

   Se llama nuevamente a Account::displayAccountsInfos() para mostrar la información final sobre todas las cuentas después de los retiros.

   Se utiliza std::for_each para mostrar el estado final de cada cuenta individualmente.

En esencia, este código:

- Crea un conjunto de cuentas bancarias.
- Realiza una serie de depósitos en estas cuentas.
- Realiza una serie de retiros en estas cuentas.
- Muestra información sobre las cuentas antes y después de las operaciones.

Account.cpp

Incluir el archivo de cabecera Account.hpp:

#include "Account.hpp" Esta línea incluye el archivo de cabecera "Account.hpp". "Account.hpp" contiene la definición de la clase Account. Incluir este archivo permite utilizar la clase Account y sus miembros en el archivo Account.cpp

Incluir la biblioteca <ctime>:

#include <ctime> Esta línea incluye la biblioteca estándar de C++ <ctime>. La biblioteca <ctime> proporciona funciones para manipular y obtener información sobre el tiempo y la fecha. Se utiliza para obtener y mostrar marcas de tiempo.

Incluir la biblioteca <iomanip>:

#include <iomanip> Esta línea incluye la biblioteca estándar de C++ <iomanip>. La biblioteca <iomanip> proporciona manipuladores de flujo que permiten formatear la salida de datos, como establecer la precisión de los números decimales, ajustar el ancho de los campos, etc. Se utiliza para formatear la salida de datos en el código.

Incluir la biblioteca <iostream>:

#include <iostream> Esta línea incluye la biblioteca estándar de C++ <iostream>. La biblioteca <iostream> proporciona objetos de flujo de entrada y salida, como std::cin y std::cout, que se utilizan para leer y escribir datos en la consola. Se utiliza para mostrar mensajes y datos en la consola.

En C++, la sentencia using se utiliza para simplificar el uso de nombres de espacios de nombres (namespaces).

Tras las inclusiones de las bibliotecas, encontramos sentencias using, utilizadas en el código como:

```
using std::cin;
using std::cout;
using std::endl;
```

¿Qué es un espacio de nombres?

- Un espacio de nombres es un mecanismo para organizar el código y evitar conflictos de nombres.
- Permite agrupar elementos relacionados (como clases, funciones, variables) bajo un nombre común.
- Por ejemplo, std es el espacio de nombres estándar de la biblioteca C++. Contiene muchas clases, funciones y objetos predefinidos, como cout, cin, vector, string, etc.

¿Qué hace using std::cin;?

- Esta línea de código utiliza la directiva using para importar el nombre cin directamente desde el espacio de nombres std.
- A partir de este punto, puedes usar cin directamente en tu código sin tener que escribir std::cin cada vez.

Ejemplos:

Sin using:

```
std::cout << "Hola, mundo!" << std::endl;
```

Con using:

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

cout << "Hola, mundo!" << endl;
```

¿Qué hace using std::cout; y using std::endl;?

- Estas líneas importan los nombres cout y endl directamente desde el espacio de nombres std.
- cout es un objeto de la clase ostream que se utiliza para enviar datos a la salida estándar (generalmente la consola).
- endl es un manipulador de flujo que inserta un salto de línea en la salida y luego vacía el búfer de salida.

Ventajas de usar using:

- Mejora la legibilidad: Hace que el código sea más limpio y fácil de leer al evitar la repetición de std::.
- Simplifica el código: Reduce la cantidad de código que tienes que escribir.

Consideraciones:

Aunque using puede mejorar la legibilidad, es importante usarlo con moderación. Importar todo un espacio de nombres con using namespace std; puede generar conflictos de nombres si hay otros elementos en tu código que tienen los mismos nombres que los elementos del espacio de nombres std.

Inicialización de variables.

int Account::_nbAccounts;: Esta línea declara un atributo estático de tipo entero llamado _nbAccounts dentro de la clase Account.

- Atributo estático: Un atributo estático es compartido por todos los objetos de la clase. Es decir, existe una única copia de este atributo para toda la clase, no una copia diferente para cada objeto individual.
- _nbAccounts: El nombre del atributo, que por convención suele comenzar con un guion bajo (_) para indicar que es un miembro privado.

= 0;: Esta parte inicializa el valor del atributo _nbAccounts a 0. Esto significa que al principio no hay ninguna cuenta creada.

int Account::_totalAmount = 0; Declara otro atributo estático de tipo entero llamado _totalAmount.

- Este atributo se utiliza para almacenar la suma total del dinero depositado en todas las cuentas creadas.
- Se inicializa con 0, ya que al principio no hay dinero depositado.

int Account::_totalNbDeposits = 0; Declara un atributo estático de tipo entero llamado _totalNbDeposits.

- Este atributo se utiliza para contar el número total de depósitos realizados en todas las cuentas.
- Se inicializa con 0, ya que al principio no se han realizado depósitos.

int Account::_totalNbWithdrawals = 0; Declara un atributo estático de tipo entero llamado _totalNbWithdrawals.

- Este atributo se utiliza para contar el número total de retiros realizados en todas las cuentas.
- Se inicializa con 0, ya que al principio no se han realizado retiros.

Importancia de los atributos estáticos:

- Información global: Permiten almacenar información que es compartida por todos los objetos de la clase.
- Conteo de objetos: Se pueden utilizar para contar el número de objetos creados de una clase.
- Estadísticas: Se pueden utilizar para recopilar estadísticas sobre el uso de la clase.

En resumen:

Estas líneas declaran cuatro atributos estáticos dentro de la clase Account. Estos atributos se utilizan para mantener un registro de información global sobre todas las cuentas creadas, como el número total de cuentas, el monto total depositado, el número total de depósitos y el número total de retiros.

Las siguientes líneas de código muestran la implementación de cuatro métodos estáticos de la clase Account.

- int Account::getNbAccounts()

  - int Account::: Indica que se trata de una función miembro de la clase Account y que devuelve un valor entero (int).
  - getNbAccounts(): Este es el nombre de la función. Es un método estático, lo que significa que pertenece a la clase en sí misma y no a una instancia específica de la clase.
  - return _nbAccounts;: Esta línea devuelve el valor del atributo privado _nbAccounts.

- int Account::getTotalAmount()

  - int Account::: Indica que se trata de una función miembro de la clase Account que devuelve un valor entero.
  - getTotalAmount(): Este es el nombre de la función. Es un método estático que devuelve el monto total de dinero depositado en todas las cuentas.
  - return _totalAmount;: Esta línea devuelve el valor del atributo privado _totalAmount.

- int Account::getNbDeposits()

  - int Account::: Indica que se trata de una función miembro de la clase Account que devuelve un valor entero.
  - getNbDeposits(): Este es el nombre de la función. Es un método estático que devuelve el número total de depósitos realizados en todas las cuentas.
  - return _totalNbDeposits;: Esta línea devuelve el valor del atributo privado _totalNbDeposits.

- int Account::getNbWithdrawals()

  - int Account::: Indica que se trata de una función miembro de la clase Account que devuelve un valor entero.
  - getNbWithdrawals(): Este es el nombre de la función. Es un método estático que devuelve el número total de reintegros realizados en todas las cuentas.
  - return _totalNbWithdrawals;: Esta línea devuelve el valor del atributo privado _totalNbWithdrawals.

- En resumen:

  Estos cuatro métodos son métodos estáticos de la clase Account que permiten obtener información global sobre todas las cuentas creadas:

  - getNbAccounts(): Obtiene el número total de cuentas creadas.
  - getTotalAmount(): Obtiene el monto total de dinero depositado en todas las cuentas.
  - getNbDeposits(): Obtiene el número total de depósitos realizados en todas las cuentas.
  - getNbWithdrawals(): Obtiene el número total de reintegros realizados en todas las cuentas.

Estos métodos son útiles para obtener estadísticas generales sobre el estado de las cuentas en el sistema.

Nota:

- Estos métodos asumen, como es el caso, que la clase Account tiene atributos privados para almacenar la información correspondiente como: _nbAccounts, _totalAmount, _totalNbDeposits y _totalNbWithdrawals.

- Los métodos estáticos no operan sobre una instancia específica de la clase, sino sobre la clase en sí misma.

void Account::_displayTimestamp(void):

1. void: Indica que la función no devuelve ningún valor.
2. Account::_displayTimestamp: Este es el nombre de la función. El carácter _ al principio del nombre indica que esta función es un método privado de la clase Account. Los métodos privados solo pueden ser llamados desde dentro de la propia clase.
3. (void): Indica que la función no recibe ningún argumento.

Obtención de la hora actual:

- time_t timestamp;: Declara una variable de tipo time_t llamada timestamp. time_t es un tipo de dato definido en la biblioteca estándar de C++ para representar el tiempo.
- struct tm *localTime;: Declara un puntero a una estructura tm llamada localTime. La estructura tm se utiliza para representar la hora en una forma más legible (año, mes, día, hora, minutos, segundos).
- std::time(&timestamp);: Llama a la función std::time() para obtener la hora actual en segundos desde la Época Unix (1 de enero de 1970) y almacena el resultado en la variable timestamp.
- localTime = localtime(&timestamp);: Convierte la hora en segundos almacenada en timestamp a una estructura tm que representa la hora local. El resultado se almacena en el puntero localTime.

Impresión de la marca de tiempo:

- cout << "[";: Imprime un corchete izquierdo "[" en la salida estándar.
- cout << localTime->tm_year + 1900;: Imprime el año. localTime->tm_year devuelve el número de años desde 1900.
- cout << std::setw(2) << std::setfill('0') << localTime->tm_mon + 1;:
- Imprime el mes.
- std::setw(2) establece el ancho mínimo del campo de salida a 2 caracteres.
- std::setfill('0') especifica que los espacios vacíos se rellenarán con ceros.
- localTime->tm_mon devuelve el mes (0-11), por lo que se suma 1 para obtener el mes en el rango 1-12.
- cout << std::setw(2) << std::setfill('0') << localTime->tm_mday;:
- Imprime el día del mes con formato de dos dígitos.

- cout << "_";: Imprime un guión bajo "_".
- cout << std::setw(2) << std::setfill('0') << localTime->tm_hour;: Imprime la hora con formato de dos dígitos.
- cout << std::setw(2) << std::setfill('0') << localTime->tm_min;: Imprime los minutos con formato de dos dígitos.
- cout << std::setw(2) << std::setfill('0') << localTime->tm_sec;: Imprime los segundos con formato de dos dígitos.
- cout << "] ";: Imprime un corchete derecho "]" y un espacio.

En resumen:

Esta función obtiene la hora actual del sistema, la formatea en un formato específico (AAAA-MM-DD_HH:MM:SS) y la imprime en la salida estándar. Esta función se utiliza para registrar los eventos o acciones dentro de la clase Account.

Nota: Dado que esta función es privada (_displayTimestamp), solo puede ser llamada desde otros métodos de la clase Account.

void Account::displayAccountsInfos()

El método Account::displayAccountsInfos() es un método público de la clase Account que se encarga de mostrar información general sobre todas las cuentas creadas.

1. Llama al método privado _displayTimestamp() para imprimir la marca de tiempo actual.
2. Imprime en la consola el número total de cuentas, el monto total de dinero depositado, el número total de depósitos y el número total de retiros, utilizando los métodos estáticos correspondientes para obtener estos valores.

Este método proporciona una forma conveniente de obtener una visión general del estado de todas las cuentas en la aplicación.

Account::Account(int initial_deposit)

Este código implementa el **constructor de la clase Account**. Cuando se crea un nuevo objeto Account, el constructor:

1. Incrementa el contador de cuentas totales.
2. Asigna un índice único a la nueva cuenta.
3. Inicializa el saldo de la cuenta con el depósito inicial.
4. Inicializa los contadores de depósitos y retiros.
5. Actualiza el monto total depositado en todas las cuentas.
6. Imprime un mensaje en la consola indicando la creación de la cuenta.

Este constructor es crucial para el correcto funcionamiento de la clase Account, ya que establece el estado inicial de cada objeto de la clase al momento de su creación.

¿Qué hace this?

- Referencia al objeto actual: Dentro de los métodos de una clase, this es un puntero constante (const this) que apunta al objeto sobre el que se está invocando el método.

- Acceso a miembros: Se utiliza this-> para acceder a los miembros (atributos y métodos) de la propia clase, especialmente cuando hay conflictos de nombres.

En resumen: this es un puntero especial que permite a los métodos de una clase acceder y manipular los miembros de la propia clase. Es esencial para resolver ambigüedades y acceder correctamente a los atributos y métodos del objeto actual. Aunque no siempre es estrictamente necesario, su uso mejora la claridad y la mantenibilidad del código.

Account::~Account()

Este es un destructor de la clase Account. Un destructor es un método especial que se llama automáticamente cuando un objeto de esa clase deja de existir (por ejemplo, cuando sale de su ámbito o cuando se elimina explícitamente).

Su función es registrar la destrucción de la cuenta y mostrar información relevante sobre ella en ese momento. Esto puede ser útil para depurar el código, realizar un seguimiento de las cuentas y comprender el ciclo de vida de los objetos de tipo Account.

Un objeto puede ser destruido por varias razones:

- Al finalizar el bloque de código donde fue creado: Si un objeto se crea dentro de una función, por ejemplo, será destruido automáticamente cuando se salga de esa función.
- Al eliminar explícitamente un objeto con delete: Si se ha creado un objeto dinámicamente utilizando new, se debe eliminar explícitamente con delete para liberar la memoria.
- Cuando un objeto sale de su alcance: Si un objeto se declara dentro de un bloque de código (por ejemplo, un bucle o una condición), será destruido al salir de ese bloque.

¿Por qué es importante este destructor?

- Liberación de recursos: Si la clase Account hubiera asignado memoria dinámicamente (por ejemplo, para almacenar el historial de transacciones), el destructor sería el lugar adecuado para liberar esa memoria utilizando delete. Esto evita fugas de memoria.
- Registro: El mensaje que se imprime en la consola puede ser útil para depurar el programa o para llevar un registro de las operaciones realizadas.
- Cierre de conexiones: Si la clase Account manejara conexiones a una base de datos o a otros servicios, el destructor podría cerrar esas conexiones para liberar recursos.

void Account::makeDeposit(int deposit)

La función makeDeposit de la clase Account realiza las siguientes acciones:

1. Imprime la fecha y hora actual.
2. Imprime información sobre la cuenta antes del depósito.
3. Suma la cantidad a depositar al saldo actual de la cuenta.
4. Incrementa el contador de depósitos de la cuenta actual.
5. Incrementa el monto total depositado en todas las cuentas.
6. Incrementa el contador total de depósitos realizados en todas las cuentas.
7. Imprime información sobre el depósito realizado y el nuevo estado de la cuenta.

Esta función encapsula la lógica de realizar un depósito en una cuenta bancaria, manteniendo el estado de la cuenta y registrando la actividad.

int Account::checkAmount(void) const

int Account::: Indica que estamos definiendo una función llamada checkAmount dentro de la clase Account.

¿Qué hace esta función en resumen?

La función checkAmount verifica si el saldo de una cuenta bancaria es negativo. Si lo es, devuelve 1 para indicar un error o un saldo insuficiente; de lo contrario, devuelve 0 para indicar que el saldo es válido.

¿Por qué se utiliza?

La función checkAmount es una herramienta fundamental para garantizar la integridad de los datos en una aplicación bancaria. Al verificar el saldo de una cuenta antes de realizar cualquier operación, se evita que se produzcan errores y se asegura que las transacciones se realicen de manera segura.

¿Por qué const al final?

Esta palabra clave es crucial. Indica que la función no modificará el estado del objeto Account sobre el que se invoca. En otras palabras, la función es de "solo lectura". Se garantiza así que la función no modificará el objeto: El modificador const al final de la declaración de la función asegura que la función checkAmount no modificará ningún miembro de datos del objeto Account. Esto es importante por varias razones:

- Seguridad: Evita cambios accidentales en el estado del objeto, lo que podría llevar a errores difíciles de depurar.
- Claridad: Indica claramente a otros programadores que la función es de "solo lectura".
- Eficiencia: En algunos casos, el compilador puede realizar optimizaciones adicionales en funciones marcadas como const.

bool Account::makeWithdrawal(int withdrawal)

Este método makeWithdrawal de la clase Account intenta realizar un reintegro de dinero de la cuenta. Verifica si hay fondos suficientes (checkAmount) y actualiza el saldo de la cuenta, el

número de reintegros realizados y el monto total de dinero retirado de todas las cuentas. También proporciona información sobre el estado de la operación (éxito o rechazo) a través de mensajes en la consola y el valor de retorno.

void Account::displayStatus(void) const

La función displayStatus es responsable de mostrar en la consola la información relevante de una cuenta bancaria. Esta información incluye:

Un timestamp (fecha y hora) para indicar cuándo se generó la información.
El índice de la cuenta (un identificador único).
El saldo actual de la cuenta.
El número total de depósitos realizados.
El número total de retiros realizados.

¿Por qué es útil esta función?

Esta función es fundamental para cualquier sistema de gestión de cuentas bancarias, ya que permite:

Visualizar el estado de una cuenta: Los usuarios pueden ver rápidamente información clave sobre sus cuentas.
Depurar y diagnosticar problemas: Los desarrolladores pueden utilizar esta función para verificar el estado de las cuentas durante el desarrollo y la depuración.
Generar informes: La información proporcionada por esta función puede utilizarse para generar informes sobre las transacciones de las cuentas.

En resumen:

La función displayStatus es una herramienta esencial para la clase Account. Proporciona una forma sencilla y clara de mostrar la información más relevante de una cuenta bancaria, lo que facilita la gestión y el seguimiento de las transacciones.

EN RESUMEN:

Este código define la clase Account en C++, la cual representa una cuenta bancaria.

Inclusión de archivos de cabecera:

- #include "Account.hpp": Incluye el archivo de cabecera Account.hpp (que se asume que contiene la declaración de la clase Account).
- #include <ctime>: Incluye la cabecera ctime, necesaria para trabajar con fechas y horas.
- #include <iomanip>: Incluye la cabecera iomanip, necesaria para manipular la salida de datos (como el ancho de campo y el relleno).
- #include <iostream>: Incluye la cabecera iostream, necesaria para trabajar con la entrada y salida estándar (como cin y cout).

Directivas using:

- using std::cin;, using std::cout;, using std::endl;: Estas directivas permiten usar los elementos cin, cout y endl directamente sin necesidad de escribir std:: antes de cada uno.

Variables estáticas:

- int Account::_nbAccounts = 0;: Variable estática que cuenta el número total de cuentas creadas.
- int Account::_totalAmount = 0;: Variable estática que almacena el monto total de dinero depositado en todas las cuentas.
- int Account::_totalNbDeposits = 0;: Variable estática que cuenta el número total de depósitos realizados en todas las cuentas.
- int Account::_totalNbWithdrawals = 0;: Variable estática que cuenta el número total de retiros realizados en todas las cuentas.

Métodos privados:

- int Account::getNbAccounts(), int Account::getTotalAmount(), int Account::getNbDeposits(), int Account::getNbWithdrawals(): Estos métodos privados proporcionan acceso de solo lectura a las variables estáticas, permitiendo a otros métodos de la clase obtener esta información.
- void Account::_displayTimestamp(void): Este método privado se encarga de obtener la hora actual del sistema y mostrarla en un formato específico (AAAA-MM-DD_HH:MM:SS).

Constructor:

- Account::Account(int initial_deposit): Este es el constructor de la clase.
- Incrementa el contador de cuentas (_nbAccounts).
- Asigna un índice único a cada cuenta (_accountIndex).
- Inicializa el saldo de la cuenta con el depósito inicial.
- Inicializa el número de depósitos y retiros.
- Actualiza el monto total y el número total de depósitos.
- Muestra un mensaje en la consola indicando que se ha creado una nueva cuenta.

Destructor:

- Account::~Account(): Este es el destructor de la clase.
- Se ejecuta automáticamente cuando un objeto de la clase Account es destruido.
- Muestra un mensaje en la consola indicando que la cuenta se ha cerrado.

Métodos públicos:

- void Account::makeDeposit(int deposit): Realiza un depósito en la cuenta.
- Actualiza el saldo, el número de depósitos y el monto total.

- Muestra un mensaje en la consola con los detalles de la operación.
- bool Account::makeWithdrawal(int withdrawal): Realiza un retiro de la cuenta.
- Verifica si hay fondos suficientes para realizar el retiro.
- Si hay fondos suficientes, actualiza el saldo, el número de retiros y el monto total.
- Muestra un mensaje en la consola con los detalles de la operación.
- Devuelve true si el retiro se realizó correctamente, false en caso contrario.
- int Account::checkAmount(void) const: Verifica si el saldo de la cuenta es negativo. Devuelve 1 si el saldo es negativo, 0 en caso contrario.
- void Account::displayStatus(void) const: Muestra el estado actual de la cuenta, incluyendo su índice, saldo, número de depósitos y número de retiros.

En resumen:

Este código define una clase Account que representa una cuenta bancaria. La clase incluye métodos para realizar depósitos y retiros, verificar el saldo, obtener información sobre todas las cuentas y mostrar el estado de una cuenta específica. Además, utiliza variables estáticas para mantener un registro de información global sobre todas las cuentas creadas.

Este código cumple con las restricciones establecidas.

- No utiliza contenedores de la STL: No se utilizan vectores (vector), listas (list), mapas (map), ni otras estructuras de datos de la STL.
- No utiliza algoritmos de la STL: No se incluye el encabezado <algorithm> ni se utilizan funciones como sort, find, etc.
- No utiliza printf, alloc o free: Estas funciones están explícitamente prohibidas.
- No utiliza using namespace std; ni friend: Se cumplen estas restricciones.
- Utiliza la biblioteca estándar C++: Se utiliza std::time, std::ctime, std::setw, std::setfill, std::cout, std::endl de la biblioteca estándar C++, lo cual está permitido.

El código utiliza principalmente:

- Variables miembro: Para almacenar información de cada cuenta (saldo, número de depósitos, etc.).
- Métodos miembro: Para realizar operaciones sobre las cuentas (depósitos, retiros, etc.).
- Operadores de flujo (<<) para la salida estándar (cout).
- La biblioteca estándar C++ para manejo de tiempo (ctime) y manipulación de salida (iomanip).

Por lo tanto, este código cumple con todas las restricciones establecidas y es válido según las reglas del ejercicio.