UNIVERSIDAD CATÓLICA DE SANTA MARÍA
PROGRAMA PROFESIONAL DE INGENIERÍA DE SISTEMAS

PRÁCTICA N° 10:

Excepciones y Archivos en C++

I
OBJETIVOS

- Analizar los conceptos fundamentales de gestión de excepciones, manejo de archivos y serialización en POO, identificando sus características principales y su importancia en el desarrollo de aplicaciones.
- Aplicar los conocimientos adquiridos sobre gestión de excepciones, manejo de archivos y serialización en POO para resolver problemas y desarrollar soluciones eficientes y robustas.
- Evaluar críticamente el uso de la gestión de excepciones, manejo de archivos y serialización en el diseño y desarrollo de aplicaciones, considerando aspectos como la eficiencia, la seguridad y la escalabilidad.
- Diseñar y construir clases, métodos y estructuras de datos que faciliten la implementación y el manejo adecuado de excepciones, archivos y serialización en un programa.
- Utilizar herramientas de depuración y pruebas para detectar y corregir errores relacionados con la gestión de excepciones, el manejo de archivos y la serialización en POO.
- Integrar de manera eficiente y coherente la gestión de excepciones, el manejo de archivos y la serialización en el diseño y desarrollo de aplicaciones más complejas, considerando buenas prácticas y estándares de calidad.
- Valorar la importancia de una gestión adecuada de excepciones, manejo de archivos y serialización en POO en el desarrollo de aplicaciones confiables, seguras y de calidad.
- Demostrar responsabilidad y ética profesional al manejar excepciones, archivos y serialización, evitando prácticas que puedan comprometer la integridad de los datos o la estabilidad del sistema.
- Trabajar de manera colaborativa y cooperativa en equipos de desarrollo, compartiendo conocimientos y experiencias relacionadas con la gestión de excepciones, manejo de archivos y serialización en POO.

II
TEMAS A TRATAR

- Introducción.
- Manejo de Excepciones.
- Jerarquía de datos
- Resumen

III
MARCO TEORICO

1. INTRODUCCIÓN

En esta práctica presentaremos el manejo de excepciones. Una excepción es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre “excepción” implica que el problema ocurre con poca frecuencia; si la “regla” es que una instrucción generalmente se ejecuta en forma correcta, entonces la “excepción a la regla” es cuando ocurre un problema. El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que un programa continúe su ejecución como si no se hubiera encontrado un problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir programas tolerantes a fallas y robustos, que puedan tratar con los problemas que puedan surgir sin dejar de

ejecutarse, o que terminen de una manera no dañina.

La práctica empieza con una descripción general de los conceptos relacionados con el manejo de excepciones, y posteriormente se demuestran las técnicas básicas para el manejo de excepciones. Mostraremos estas técnicas mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando una función intenta realizar una división entre cero. Después hablaremos sobre ciertas cuestiones adicionales sobre el manejo de excepciones, como la forma en que se deben manejar las excepciones que ocurren en un constructor o destructor, y cómo manejar las excepciones que ocurren si el operador new falla al asignar memoria para un objeto. Concluiremos este capítulo presentando varias clases que proporciona la Biblioteca estándar de C++ para manejar excepciones.

El almacenamiento de datos en variables y arreglos es temporal. Los archivos se utilizan para la persistencia de los datos: datos de retención permanente. Las computadoras almacenan archivos en dispositivos de almacenamiento secundario como discos duros, CDs, DVDs, unidades f lash y cintas magnéticas. En este capítulo explicaremos cómo construir programas en C++ para crear, actualizar y procesar archivos de datos. Consideraremos los archivos secuenciales y los archivos de acceso aleatorio. Compararemos el procesamiento de archivos de datos con formato y el procesamiento de archivos de datos puros.

El problema de la seguridad es uno de los clásicos quebraderos de cabeza de la programación. Los diversos lenguajes han tenido siempre que lidiar con el mismo problema: ¿Qué hacer cuando se presenta una circunstancia verdaderamente imprevista? (por ejemplo un error). El asunto es especialmente importante si se trata de lenguajes para escribir programas de "Misión crítica"; digamos por ejemplo controlar los ordenadores de una central nuclear o de un sistema de control de tráfico aéreo.

Antes que nada, digamos que en el lenguaje de los programadores C++ estas "circunstancias imprevistas" reciben el nombre de excepciones, por lo que el sistema que implementa C++ para resolver estos problemas recibe el nombre de manejador de excepciones. Así pues, las excepciones son condiciones excepcionales que pueden ocurrir dentro del programa durante su ejecución. Por ejemplo, que ocurra una división por cero, se agote la memoria disponible, etc. que requieren recursos especiales para su control.

2. MANEJO DE EXCEPCIONES

En algunos programas complejos que realicen funciones críticas puede ser necesario controlar todos los detalles de su ejecución. Es relativamente normal que un programa falle durante su ejecución debido a que se haya realizado o intentado realizar una operación no permitida (por ejemplo, una división por cero), porque se haya excedido el rango de un vector, etc. Si en tiempo de ejecución se produce un error de éstos, se pueden adoptar varias estrategias.

Una consiste en no hacer nada, dejando que el ordenador emita un mensaje de error y finalizando la ejecución del programa bruscamente, con lo que la información referente al error producido es mínima. Otra posibilidad es obligar al programa a continuar con la ejecución arrastrando el error, lo que puede ser interesante en el caso de que se tenga la seguridad de que ese error no se va a propagar y que los resultados que se obtengan al final seguirán siendo fiables. Otra posibilidad es tratar de corregir el error y de seguir con la ejecución del programa.

C++ dispone de una herramienta adicional que permite terminar la ejecución del programa de una manera más ordenada, proporcionando la información que se requiera para detectar la fuente del error que se haya producido. Esta herramienta recibe el nombre de **mecanismo de excepciones**.

Consta básicamente de dos etapas: una inicial o de **lanzamiento** (`throw`) de la **excepción**, y otra de gestión o **captura** (**handle**) de la misma. **Lanzar** una **excepción** es señalar que se ha producido una determinada situación de error.

Para **lanzar una excepción** se coloca la porción de código que se desea controlar dentro de un bloque `try`. Si en un momento dado, dentro de ese bloque `try` se pasa por una instrucción `throw` consecuencia de un determinado error, la ejecución acude al **gestor de excepciones** correspondiente, que deberá haberse definido a continuación del bloque `try`. En este **gestor** se habrán definido las operaciones que se deban realizar en el caso de producirse un error de cada tipo, que pueden ser emitir mensajes, almacenar información en ficheros.

A continuación, se presenta un ejemplo muy sencillo de **manejo de excepciones**: supóngase

el caso de un programa para resolución de **ecuaciones de segundo grado** del tipo **$ax^2+bx+c=0$**. Se desean controlar dos tipos de posible error: que el coeficiente del término de segundo grado (**a**) sea **0**, y que la ecuación tenga soluciones imaginarias (**b2-4ac<0**).

Para ello se realiza la llamada a la función de resolución **raíces** dentro de un bloque **try** y al final de éste se define el gestor **catch** que se activará si el programa pasa por alguna de las sentencias **throw** que están incluidas en la función **raíces**. En el caso de producirse alguno de los errores controlados se imprimirá en pantalla el mensaje correspondiente:

```
#include <iostream>
#include <math.h>
using namespace std;
void raices(const double a, const double b, const double c);
enum error{NO REALES, PRIMERO};
void main(void)
{
      try {
      raices(1.0, 2.0, 1.0); //dentro de raices() se
                  //lanza la excepción
         raices(2.0, 1.0, 2.0);
      }
      catch (error e)
   { // e es una variable enum de tipo error
         switch(e)
      {
            case NO REALES:
               cout << "No Reales" << endl;
               break;
            case PRIMERO:
               cout << "Primero Nulo" << endl;
               break;
         }
      }
}
void raices(const double a, const double b,
         const double c)
// throw(error);
{
      double disc, r1, r2;
      if (b*b<4*a*c)
         throw NO REALES; // se lanza un error
      if(a==0)
         throw PRIMERO;
      disc=sqrt(b*b-4*a*c);
      r1 = (-b-disc)/(2*a);
      r2 = (-b+disc)/(2*a);
      cout << "Las raíces son:" << r1
      <<" y " << r2 << endl;
}
```

Es importante señalar que los únicos errores que se pueden controlar son los que se han producido dentro del propio programa, conocidos como **errores síncronos**. Es imposible el manejo de errores debidos a un mal funcionamiento del sistema por cortes de luz, bloqueos del ordenador, etc.

A. MANEJO DE EXCEPCIONES EN C++

El manejo de excepciones C++ se basa en un mecanismo cuyo funcionamiento tiene tres etapas básicas:

- *Se intenta ejecutar un bloque de código y se decide qué hacer si se produce una circunstancia excepcional durante su ejecución.*
- *Se produce la circunstancia: se "**lanza**" una excepción (en caso contrario el programa sigue su curso normal).*
- *La ejecución del programa es desviada a un sitio específico donde la excepción es "**capturada**" y se decide que hacer al respecto.*

¿Pero que es eso de "**lanzar**" y "**capturar**" una excepción"? En general la frase se usa con un doble sentido: Por un lado es un mecanismo de salto que transfiere la ejecución desde un punto (que "**lanza**" la **excepción**) a otro dispuesto de antemano para tal fin (que "**captura**"

la **excepción**). A este último se le denomina **manejador** o "**handler**" de la **excepción**. Además del salto -como un **goto**-, en el punto de lanzamiento de la **excepción** se crea un **objeto**, a modo de mensajero, que es capturado por el "**handler**" (como una **función** que recibe un **argumento**). El **objeto** puede ser cualquiera, pero lo normal es que pertenezca a una clase especial definida al efecto, que contiene la información necesaria para que el receptor sepa qué ha pasado; cual es la naturaleza de la circunstancia excepcional que ha "lanzado" la excepción.

Para las tres etapas anteriores existen tres palabras clave específicas: **try**, **throw** y **catch**. El detalle del proceso es como sigue.

B. INTENTO (TRY).

En síntesis podemos decir que el programa se prepara para cierta acción, decimos que "lo intenta". Para ello se especifica un bloque de código cuya ejecución se va a intentar ("**try**-block") utilizando la palabra clave **try**.

```
try {    // bloque de código-intento
  ...
}
```

El juego consiste en indicar al programa que si existe un error durante el "intento", entonces debe lanzar una excepción y transferir el control de ejecución al punto donde exista un manejador de excepciones ("**handler**") que coincida con el tipo lanzado. Si no se produce ninguna excepción, el programa sigue su curso normal.

De lo dicho se deduce inmediatamente que se pueden lanzar excepciones de varios tipos y que pueden existir también receptores (manejadores) de varios tipos; incluso manejadores "universales", capaces de hacerse cargo de cualquier tipo de excepción. A la inversa, puede ocurrir que se lance una excepción para la que no existe manejador adecuado.

Así pues, **try** es una sentencia que en cierta forma es capaz de especificar el flujo de ejecución del programa. Un bloque-intento debe ser seguido inmediatamente por el bloque manejador de la excepción.

C. SE LANZA UNA EXCEPCIÓN (THROW).

Si se detecta una circunstancia excepcional dentro del bloque-intento, se lanza una excepción mediante la ejecución de una sentencia **throw**. Por ejemplo:

```
if (condicion) throw "overflow";
```

Es importante advertir que, salvo los casos en que la excepción es lanzada por las propias librerías C++ (como consecuencia de un error), estas no se lanzan espontáneamente. Es el programador el que debe utilizar una sentencia (generalmente condicional) para, en su caso, lanzar la excepción.

El lenguaje C++ especifica que todas las excepciones deben ser lanzadas desde el interior de un bloque-intento y permite que sean de cualquier tipo. Como se ha apuntado antes, generalmente son un objeto (instancia de una clase) que contiene información. Este objeto es creado y lanzado en el punto de la sentencia **throw** y capturado donde está la sentencia **catch**. El tipo de información contenido en el objeto es justamente el que nos gustaría tener para saber que tipo de error se ha producido. En este sentido puede pensarse en las excepciones como en una especie de correos que transportan información desde el punto del error hasta el sitio donde esta información puede ser analizada.

D. LA EXCEPCIÓN ES CAPTURADA EN UN PUNTO ESPECÍFICO DEL PROGRAMA (CATCH).

Esta parte del programa se denomina manejador ("**handler**"); se dice que el "**handler**" captura la excepción. El **handler** es un bloque de código diseñado para manejar la excepción precedida por la palabra **catch**. El lenguaje C++ requiere que exista al menos un manejador inmediatamente después de un bloque **try**. Es decir, se requiere el siguiente

esquema:

```
try {         // bloque de código que se intenta
  ...
}
catch (...) { // bloque manejador de posibles excepciones
  ...
}
...           // continua la ejecución normal
```

El "**handler**" es el sitio donde continua el programa en caso de que ocurra la circunstancia excepcional (generalmente un error) y donde se decide qué hacer. A este respecto, las estrategias pueden ser muy variadas (no es lo mismo el programa de control de un reactor nuclear que un humilde programa de contabilidad). En último extremo, en caso de errores absolutamente irrecuperables, la opción adoptada suele consistir en mostrar un mensaje explicando el error. Puede incluir el consabido "Avise al proveedor del programa" o bien generar un fichero texto (por ejemplo: error.txt) con la información pertinente, que se guarda en disco con objeto de que pueda ser posteriormente analizado y corregido en sucesivas versiones de la aplicación.

Llegados a este punto debemos recordar que, como veremos en los ejemplos, las excepciones generadas pueden ser de diverso tipo (según el tipo de error), y que también pueden existir diversos manejadores. De hecho, se debe incluir el manejador correspondiente a cada excepción que se pueda generar.

3. JERARQUÍA DE DATOS

Finalmente, todos los elementos de datos que procesan las computadoras digitales se reducen a combinaciones de ceros y unos. Esto ocurre debido a que es simple y económico construir dispositivos electrónicos que puedan asumir uno de dos estados estables: un estado representa 0 y el otro representa 1. Es increíble que las impresionantes funciones realizadas por las computadoras impliquen solamente las manipulaciones más fundamentales de 0s y 1s.

El elemento más pequeño de datos que soportan las computadoras se conoce como bit (abreviatura de “dígito binario”; un dígito que puede suponer uno de dos valores). Cada elemento de datos, o bit, puede asumir el valor 0 o el valor 1. Los circuitos de computadora realizan varias manipulaciones simples de bits, como examinar o establecer el valor de un bit, o invertir su valor (de 1 a 0 o de 0 a 1).

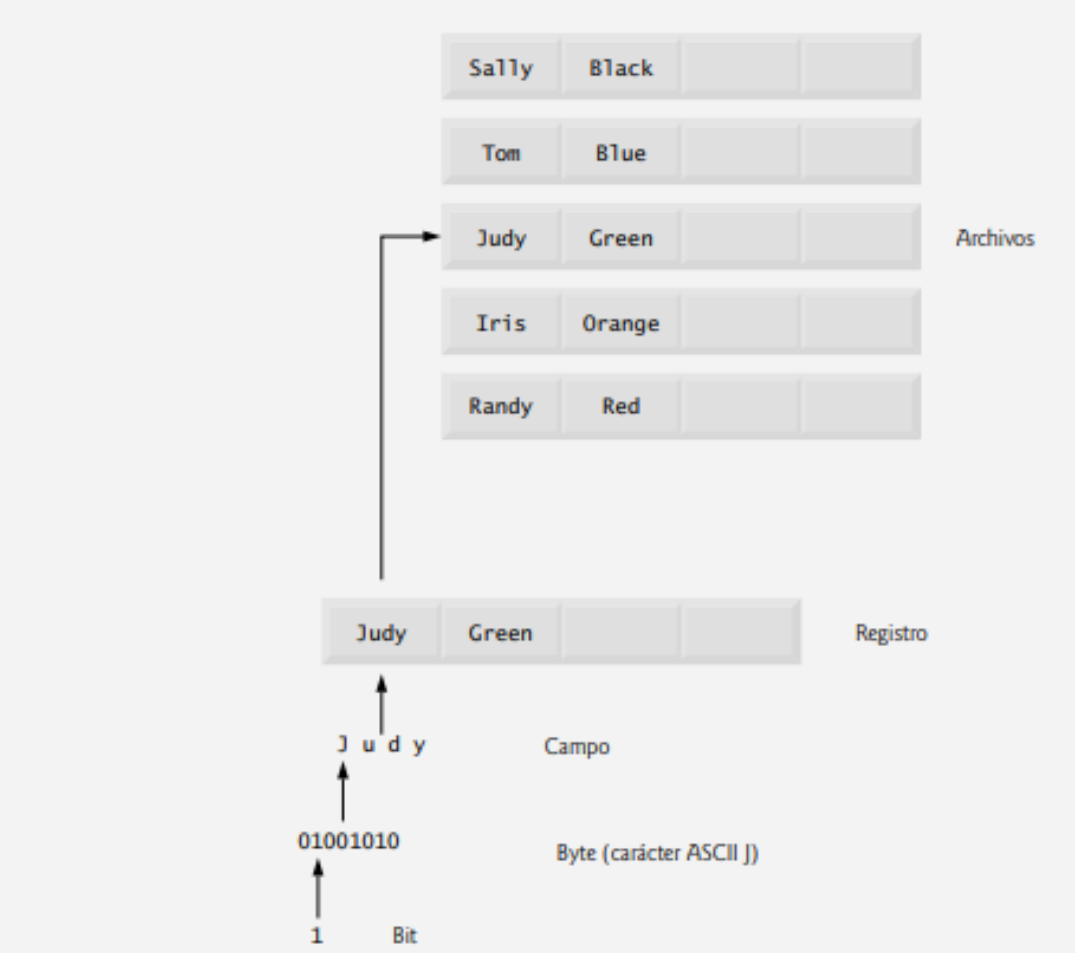
Es muy difícil para los programadores trabajar con datos en el formato de bits de bajo nivel. Es preferible trabajar con datos en formatos como dígitos decimales (0-9), letras (A-Z y a-z) y símbolos especiales (por ejemplo, $, @, %, &, * y muchos otros). Los dígitos, letras y símbolos especiales se conocen como caracteres. El conjunto de caracteres de una computadora es el conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos de esa computadora. Las computadoras pueden procesar solamente 1s y 0s, por lo que cada carácter en el conjunto de caracteres de una computadora se representa como un patrón de 1s y 0s. Los bytes están compuestos de ocho bits. Los programadores crean programas y elementos de datos con caracteres; las computadoras manipulan y procesan estos caracteres como patrones de bits. Por ejemplo, C++ proporciona el tipo de datos char. Cada char por lo general ocupa un byte. C++ también proporciona el tipo de datos wchar_t, que puede ocupar más de un byte (para soportar conjuntos de caracteres más grandes, como el conjunto de caracteres Unicode®; para obtener más información acerca de Unicode®, visite el sitio www.unicode.org).

Así como los caracteres están compuestos de bits, los campos están compuestos de caracteres. Un campo es un grupo de caracteres que transmiten cierto significado. Por ejemplo, un campo que consiste en letras mayúsculas y minúsculas puede representar el nombre de una persona.

Los elementos de datos que son procesados por las computadoras forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y complejos en estructura, a medida que progresamos de bits a caracteres, de caracteres a campos, hacia agregados de datos más grandes.

Generalmente, un registro (que puede ser representado como una clase en C++) está compuesto de varios campos (conocidos como miembros de datos en C++). Por ejemplo, en un sistema de nóminas el registro para un empleado específico podría incluir los siguientes campos:

1. Número de identificación del empleado
2. Nombre
3. Dirección
4. Sueldo por hora
5. Número de exenciones reclamadas
6. Ingresos desde inicio de año a la fecha
7. Monto de impuestos retenidos



Por lo tanto, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada campo está asociado al mismo empleado. Un archivo es un grupo de registros relacionados.1 El archivo de nómina de una compañía generalmente contiene un registro para cada empleado. Por ejemplo, un archivo de nómina para una pequeña compañía podría contener sólo 22 registros, mientras que un archivo de nómina para una compañía grande podría contener 100,000 registros. Es común para una compañía tener muchos archivos, algunos de ellos conteniendo miles de millones, o incluso billones de caracteres de información.

Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse cuando menos un campo en cada registro como clave de registro. Una clave de registro sirve para identificar que un registro pertenece a una persona o entidad específica, y es única en cada registro. En el registro de nómina que describimos anteriormente, por lo general se elegiría el número de identificación de empleado como clave de registro.

Existen muchas formas de organizar los registros en un archivo. Un tipo común de organización se conoce como archivo secuencial, en el cual los registros se almacenan en orden, con base en el campo que es la clave de registro. En un archivo de nómina, los registros generalmente se colocan en orden, con base en el número de identificación de empleado. El registro del primer empleado en el archivo contiene el número de identificación

de empleado más pequeño, y los registros subsiguientes contienen números cada vez mayores.

La mayoría de las empresas utilizan muchos archivos distintos para almacenar datos. Por ejemplo, una compañía podría tener archivos de nómina, de cuentas por cobrar (listas del dinero que deben los clientes), de cuentas por pagar (listas del dinero que se debe a los proveedores), archivos de inventarios (listas de información acerca de los artículos que maneja la empresa) y muchos otros tipos de archivos. A menudo, un grupo de archivos relacionados se almacena en una base de datos. A una colección de programas diseñada para crear y administrar bases de datos se le conoce como sistema de administración de bases de datos (DBMS).

4. ARCHIVOS Y FLUJOS

C++ considera a cada archivo como una secuencia de bytes. Cada archivo termina con un marcador de fin de archivo o con un número de bytes específico que se registra en una estructura de datos administrativa, mantenida por el sistema. Cuando se abre un archivo, se crea un objeto y se asocia un flujo a ese objeto. Ya vimos que los objetos cin, cout, cerr y clog se crean cuando se incluye <iostream>. Los f lujos asociados con estos objetos proporcionan canales de comunicación entre un programa y un archivo o dispositivo específico. Por ejemplo, el objeto cin (objeto f lujo de entrada estándar) permite a un programa introducir datos desde el teclado o desde otros dispositivos, el objeto cout (objeto flujo de salida estándar) permite a un programa enviar datos a la pantalla o a otros dispositivos, y los objetos cerr y clog (objetos flujo de error estándar) permiten a un programa enviar mensajes de error a la pantalla o a otros dispositivos.

A. CREACIÓN DE UN ARCHIVO SECUENCIAL

C++ no impone una estructura sobre un archivo. Por ende, en un archivo de C++ no existe un concepto tal como el de un “registro”. En consecuencia, el programador debe estructurar los archivos de manera que cumplan con los requerimientos de la aplicación. En el siguiente ejemplo veremos cómo se puede imponer una estructura de registro simple sobre un archivo.

El código siguiente crea un archivo secuencial que podría utilizarse en un archivo de cuentas por cobrar, para ayudar a administrar el dinero que deben los clientes de crédito a una empresa. Para cada cliente, el programa obtiene su número de cuenta, nombre y saldo (es decir, el monto que el cliente debe a la empresa por los bienes y servicios recibidos en el pasado).

```
// Creación de un archivo secuencial.
#include <iostream>
#include <string>
#include <fstream> // contains file stream processing types
#include <cstdlib> // prototipo de función de salida
using namespace std;

int main()
{
   // constructor ofstream apertura de archivo
   ofstream outClientFile{"clientes.txt", ios::out};

   // salir del programa si no puede crear el archivo
   if (!outClientFile) { // sobrecargar operador !
      cerr << " No se pudo abrir el archivo " << endl;
      exit(EXIT_FAILURE);
   }

   cout << " Ingrese la cuenta, el nombre y el saldo.\n"
      << " Ingrese el final del archivo para finalizar la entrada.\n? ";

   int account; // el número de cuenta
   string name; // el nombre del propietario de la cuenta
   double balance; // el saldo de la cuenta
```

```
   // lee cuenta, nombre y saldo de cin, luego lo coloca en el archivo
   while (cin >> account >> name >> balance) {
      outClientFile << Cuenta << ' ' << Nombre << ' ' << Saldo << endl;
      cout << "? ";
   }
}
```

Los datos que se obtienen para cada cliente constituyen un registro para ese cliente. El número de cuenta sirve como la clave de registro; es decir, el programa crea y da mantenimiento al archivo en orden por número de cuenta. Este programa supone que el usuario introduce los registros en orden por número de cuenta. En un sistema de cuentas por cobrar completo, sería conveniente contar con una herramienta para ordenar datos, para que el usuario pueda introducir los registros en cualquier orden; después los registros se ordenarían y escribirían en el archivo vamos a examinar este programa. Como se dijo antes, para abrir los archivos se crean objetos ifstream, ofstreamo fstream. En programa, el archivo se va a abrir para salida, por lo que se crea un objeto ofstream. Se pasan dos argumentos al constructor del objeto: el nombre de archivo y el modo de apertura de archivo. Para un objeto ofstream, el modo de apertura de archivo puede ser ios::out para enviar datos a un archivo, o ios::app para adjuntar datos al final de un archivo (sin modificar los datos que ya estén en el archivo). Los archivos existentes que se abren con el modo ios::out se truncan: se descartan todos los datos en el archivo. Si el archivo especificado no existe todavía, entonces el objeto ofstream crea el archivo, usando ese nombre de archivo.

Se crea un objeto ofstream llamado archivoClientesSalida, asociado con el archivo clientes.dat que se abre en modo de salida. Los argumentos "clientes.dat" e ios::out se pasan al constructor de ofstream, el cual abre el archivo (esto establece una “línea de comunicación” con el archivo). De manera predeterminada, los objetos ofstream se abren en modo de salida, por lo que se podría haber utilizado la instrucción alterna

```
ofstream archivoClientesSalida( "clientes.dat" );
```

Para abrir clientes.dat en modo de salida. En la tabla posterior se listan los modos de apertura de archivos.

Se puede crear un objeto ofstream sin necesidad de abrir un archivo específico; después se puede adjuntar un archivo al objeto. Por ejemplo, la instrucción

```
ofstream archivoClientesSalida;
```

crea un objeto ofstream llamado archivoClientesSalida. La función miembro open de ofstream abre un archivo y lo adjunta a un objeto ofstream existente, como se muestra a continuación:

```
archivoClientesSalida.open( "clientes.dat", ios::out );
```

Modo	Descripción
ios::app	Añade toda la salida al final del archivo.
ios::ate	Abre un archivo en modo de salida y se desplaza hasta el final del archivo (por lo general se utiliza para añadir datos a un archivo). Los datos se pueden escribir en cualquier parte del archivo.
ios::in	Abre un archivo en modo de entrada.
ios::out	Abre un archivo en modo de salida.

ios::trunc	Descarta el contenido del archivo, si es que existe (también es la acción predeterminada para ios::out).
ios::binary	Abre un archivo en modo de entrada o salida binaria (es decir, que no es texto).

Después de crear un objeto ofstream y tratar de abrirlo, el programa prueba si la operación de apertura tuvo éxito. La instrucción if utiliza la función miembro operator! sobrecargada de ios para determinar si la operación open tuvo éxito. La condición devuelve un valor true si se establece el bit failbit o el bit badbit para el f lujo en la operación open. Ciertos posibles errores son: tratar de abrir un archivo no existente en modo de lectura, tratar de abrir un archivo en modo de lectura o escritura sin permiso, y abrir un archivo en modo de escritura cuando no hay espacio disponible en disco. Si la condición indica un intento fallido de abrir el archivo, se imprime el mensaje de error "No se pudo abrir el archivo", y en se invoca a la función exit para terminar el programa. El argumento para exit se devuelve al entorno desde el que se invocó el programa. El argumento 0 indica que el programa terminó en forma normal; cualquier otro valor indica que el programa terminó debido a un error. El entorno de llamada (probablemente el sistema operativo) utiliza el valor devuelto por exit para responder al error en forma apropiada. Otra función miembro de operador sobrecargada de ios (operator void*) convierte el flujo en un apuntador, para que se pueda evaluar como 0 (es decir, el apuntador nulo) o un número distinto de cero (es decir, cualquier otro valor de apuntador). Cuando se usa el valor de un apuntador como una condición, C++ convierte un apuntador nulo en el valor bool false y un apuntador no nulo en el valor bool true. Si se establecieron los bits failbit o badbit para el flujo, se devuelve 0 (false). La condición en la instrucción while invoca a la función miembro operator void* en cin de manera implícita. La condición permanece como true, siempre y cuando no se haya establecido el bit failbit o el bit badbit para cin. Al introducir el indicador de fi n de archivo se establece el bit failbit para cin. La función operator void * se puede utilizar para probar un objeto de entrada para el fi n de archivo, en vez de llamar a la función miembro eof de manera explícita en el objeto de entrada. Si se abrió el archivo con éxito, el programa empieza a procesar los datos. Se pide al usuario que introduzca varios campos para cada registro, o el indicador de fin de archivo cuando esté completa la entrada de datos. Se extrae cada conjunto de datos y se determina si se introdujo el fin de archivo. Al encontrar el fin de archivo, o cuando se introducen datos incorrectos, operator void * devuelve el apuntador nulo (que se convierte en el valor bool false) y la instrucción while termina. El usuario introduce el fin de archivo para informar al programa que ya no procese más datos. El indicador de fin de archivo se establece cuando el usuario introduce la combinación de teclas de fin de archivo. La instrucción while itera hasta que se establece el indicador de fin de archivo.

Se escribe un conjunto de datos en el archivo clientes.dat, usando el operador de inserción de flujo << y el objeto archivoClientesSalida asociado con el archivo al principio del programa. Los datos se pueden recuperar mediante un programa diseñado para leer el archivo. Observe que, debido a que el archivo creado es simplemente un archivo de texto, puede verse en cualquier editor de texto. Una vez que el usuario introduce el indicador de fin de archivo, main termina. Esto invoca de manera implícita a la función destructor del objeto archivoClientesSalida, que cierra el archivo clientes.dat. También podemos cerrar el objeto ofstream de manera explícita, usando la función miembro close en la instrucción

```
archivoClientesSalida.close();
```

En la ejecución de ejemplo para el programa, el usuario introduce información para cinco cuentas y después indica que la entrada de datos está completa, al introducir el fin de archivo (se muestra ^Z para Microsoft Windows). Esta ventana de diálogo no muestra cómo aparecen los registros de datos en el archivo. Para verificar que el programa haya creado el archivo con éxito, en la siguiente sección se muestra cómo crear un programa que lea este archivo e imprima su contenido.

B. CÓMO LEER DATOS DE UN ARCHIVO SECUENCIAL

Los archivos almacenan datos de manera que éstos se puedan obtener para procesarlos cuando sea necesario. En la sección anterior demostramos cómo crear un archivo para acceso secuencial. En esta sección veremos cómo leer datos secuencialmente desde un archivo. En el código siguiente se leen registros del archivo de datos clientes.dat que

creamos usando el programa anterior, y se muestra el contenido de estos registros. Al crear un objeto ifstream, se abre un archivo en modo de entrada.

El constructor de ifstream puede recibir el nombre del archivo y el modo de apertura del mismo como argumentos. Se crea un objeto ifstream llamado archivoClientesEntrada, y se asocia con el archivo clientes.dat.

Los argumentos entre paréntesis se pasan a la función constructor de ifstream, la cual abre el archivo y establece una "línea de comunicación" con el mismo.

```
// Cómo leer e imprimir un archivo secuencial.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <fstream> // flujo de archivo
using std::ifstream; // flujo de archivo de entrada

#include <iomanip>
using std::setw;
using std::setprecision;

#include <string>
using std::string;

#include <cstdlib>
using std::exit; // prototipo de la función exit

void imprimirLinea(int, const string, double); // prototipo

int main()
{
// el constructor de ifstream abre el archivo
ifstream archivoClientesEntrada("clientes.dat", ios::in);

// sale del programa si ifstream no pudo abrir el archivo
if (!archivoClientesEntrada)
{
   cerr << "No se pudo abrir el archivo" << endl;
   exit(1);
} // fin de if

int cuenta;
char nombre[30];
double saldo;

cout << left << setw(10) << "Cuenta" << setw(13)
   << "Nombre" << "Saldo" << endl << fixed << showpoint;

// muestra cada registro en el archivo
while (archivoClientesEntrada >> cuenta >> nombre >> saldo)
   imprimirLinea(cuenta, nombre, saldo);

return 0; // el destructor de ifstream cierra el archivo
} // fin de main

 // muestra un solo registro del archivo
void imprimirLinea(int cuenta, const string nombre, double saldo)
```

```
{
cout << left << setw(10) << cuenta << setw(13) << nombre
   << setw(7) << setprecision(2) << right << saldo << endl;
} // fin de la función imprimirLine
```

C. ARCHIVOS DE ACCESO ALEATORIO

Hasta ahora hemos visto cómo crear archivos secuenciales y buscar en ellos para localizar información. Los archivos secuenciales son inapropiados para las aplicaciones de acceso instantáneo, en las que un registro específico se debe localizar inmediatamente. Las aplicaciones comunes de acceso instantáneo son los sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, cajeros automáticos y otros tipos de sistemas de procesamiento de transacciones que requieran acceso rápido a datos específicos. Un banco podría tener cientos de miles (o incluso millones) de otros clientes, y a pesar de ello, cuando un cliente utiliza un cajero automático, el programa comprueba la cuenta de ese cliente en unos cuantos segundos o menos, para ver si tiene suficientes fondos. Este tipo de acceso instantáneo es posible mediante los archivos de acceso aleatorio. Los registros individuales de un archivo de acceso aleatorio se pueden utilizar de manera directa (y rápida), sin tener que buscar en otros registros. Como hemos dicho, C++ no impone una estructura sobre un archivo. Por lo tanto, la aplicación que desee utilizar archivos de acceso aleatorio debe crearlos. Se puede utilizar una variedad de técnicas. Tal vez el método más sencillo es requerir que todos los registros en un archivo sean de la misma longitud fi ja. Al utilizar registros de longitud fija y con el mismo tamaño, es más fácil para el programa calcular (como una función del tamaño de registro y la clave de registro) la ubicación exacta de cualquier registro relativo al inicio del archivo. Pronto veremos cómo esto facilita el acceso inmediato a registros específicos, incluso en archivos extensos. Luego se ilustra la forma en que C++ ve a un archivo de acceso aleatorio, compuesto de registros de longitud fija (en este caso, cada registro tiene 100 bytes de longitud). Un archivo de acceso aleatorio es como un ferrocarril con muchos carros del mismo tamaño; algunos están vacíos y otros llenos. Se pueden insertar datos en un archivo de acceso aleatorio sin destruir otros datos en el archivo. Los datos almacenados con anterioridad también se pueden actualizar o eliminar, sin necesidad de volver a escribir el archivo completo.

En las siguientes secciones explicaremos cómo crear un archivo de acceso aleatorio, introducir datos en el archivo, leer los datos tanto en forma secuencial como aleatoria, actualizar los datos y eliminar datos que ya no sean necesarios.

D. CREACIÓN DE UN ARCHIVO DE ACCESO ALEATORIO

La función miembro write de ostream envía un número fijo de bytes, empezando en una ubicación específica en memoria al flujo especificado. Cuando el flujo se asocia con un archivo, la función write escribe los datos en la ubicación en el archivo especificado mediante el apuntador “colocar” de posición del archivo. La función miembro read de istream recibe un número fijo de bytes del flujo especificado y los coloca en un área en memoria, que empieza en una dirección especificada. Si el flujo se asocia con un archivo, la función read introduce bytes en la ubicación en el archivo especificado por el apuntador “obtener” de posición del archivo.

Escritura de bytes mediante la función miembro write de ostream

Al escribir el entero número en un archivo, en vez de usar la instrucción

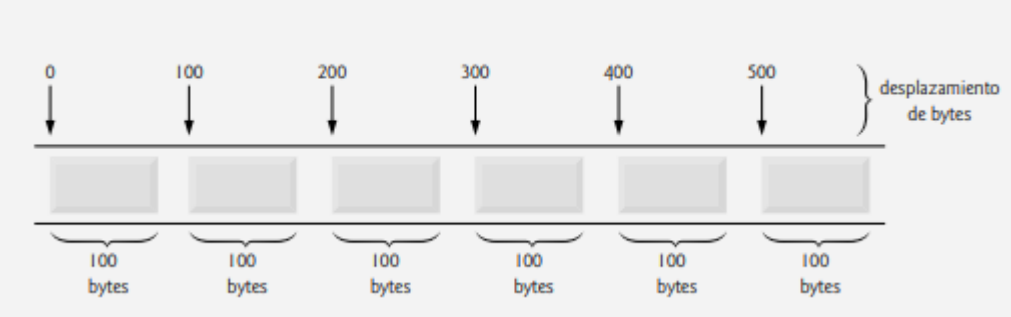
```
archivoSalida << numero;
```

Que para un entero de cuatro bytes podría imprimir desde un dígito hasta 11 (10 dígitos más un signo, cada uno de los cuales requiere un solo byte de almacenamiento), podemos usar la instrucción

```
archivoSalida.write(reinterpret_cast<const char*>(&numero), sizeof(numero));
```

Que siempre escribe la versión binaria de los cuatro bytes del entero (en un equipo con

enteros de cuatro bytes). La función write trata a su primer argumento como un grupo de bytes, al ver el objeto en memoria como un const char *, el cual es un apuntador a un byte. Empezando desde esa ubicación, la función write envía como salida el número de bytes especificados por su segundo argumento: un entero de tipo size_t. Como veremos, la función read de istream se puede utilizar después para leer los cuatro bytes y colocarlos de vuelta en la variable entera número.




```
// Definición de la clase DatosCliente.
#ifndef DATOSCLIENTE_H
#define DATOSCLIENTE_H

#include <string>
using std::string;

class DatosCliente
{
public:
// constructor predeterminado de DatosCliente
DatosCliente(int = 0, string = "", string = "", double = 0.0);

// funciones de acceso para numeroCuenta
void establecerNumeroCuenta(int);
int obtenerNumeroCuenta() const;

// funciones de acceso para apellidoPaterno
void establecerApellidoPaterno(string);
string obtenerApellidoPaterno() const;

// funciones de acceso para primerNombre
void establecerPrimerNombre(string);
string obtenerPrimerNombre() const;

// funciones de acceso para el saldo
void establecerSaldo(double);
double obtenerSaldo() const;
private:
int numeroCuenta;
char apellidoPaterno[15];
char primerNombre[10];
double saldo;
}; // fin de la clase DatosCliente

#endif
```

Los objetos de la clase string no tienen tamaño uniforme, sino que utilizan la memoria asignada en forma dinámica para dar cabida a las cadenas de varias longitudes. Este programa debe mantener registros de longitud fija, por lo que la clase DatosCliente almacena el primer nombre y el apellido del cliente en arreglos char de longitud fija. Las funciones miembro establecerApellidoPaterno y establecerPrimerNombre copian los caracteres de un objeto string en el arreglo char correspondiente. Considere la función

establecerApellidoPaterno. En la línea 40 se inicializa el apuntador const char * valorApellidoPaterno con el resultado de una llamada a la función miembro string llamada datos, la cual devuelve un arreglo que contiene los caracteres del objeto string. [Nota: no se garantiza que este arreglo tenga terminación nula]. Se invoca a la función miembro string llamada size para obtener la longitud de cadenaApellidoPaterno. Se asegura que longitud sea menor de 15 caracteres, y después se copian longitud caracteres de valorApellidoPaterno al arreglo char llamado apellidoPaterno. La función miembro establecerPrimerNombre realiza los mismos pasos para el primer nombre.

```
// La clase DatosCliente almacena la información de crédito del cliente.
#include <string>
using std::string;

#include "DatosCliente.h"

// constructor predeterminado de DatosCliente
DatosCliente::DatosCliente(int valorNumeroCuenta,
string valorApellidoPaterno, string valorPrimerNombre, double valorSaldo)
{
establecerNumeroCuenta(valorNumeroCuenta);
establecerApellidoPaterno(valorApellidoPaterno);
establecerPrimerNombre(valorPrimerNombre);
establecerSaldo(valorSaldo);
} // fin del constructor de DatosCliente

// obtiene el valor del número de cuenta
int DatosCliente::obtenerNumeroCuenta() const
{
return numeroCuenta;
} // fin de la función obtenerNumeroCuenta

// establece el valor del número de cuenta
void DatosCliente::establecerNumeroCuenta(int valorNumeroCuenta)
{
numeroCuenta = valorNumeroCuenta; // debe validar
} // fin de la función establecerNumeroCuenta

// obtiene el valor del apellido paterno
string DatosCliente::obtenerApellidoPaterno() const
{
return apellidoPaterno;
} // fin de la función obtenerApellidoPaterno

// establece el valor del apellido paterno
void DatosCliente::establecerApellidoPaterno(string cadenaApellidoPaterno)
{
// copia a lo más 15 caracteres de la cadena a apellidoPaterno
const char* valorApellidoPaterno = cadenaApellidoPaterno.data();
int longitud = cadenaApellidoPaterno.size();
longitud = (longitud < 15 ? longitud : 14);
strncpy(apellidoPaterno, valorApellidoPaterno, longitud);
apellidoPaterno[longitud] = '\0'; // adjunta un carácter nulo a
apellidoPaterno
} // fin de la función establecerApellidoPaterno

// obtiene el valor del primer nombre
string DatosCliente::obtenerPrimerNombre() const
{
return primerNombre;
} // fin de la función obtenerPrimerNombre

// establece el valor del primer nombre
void DatosCliente::establecerPrimerNombre(string cadenaPrimerNombre)
```

```
{
// copia a lo más 10 caracteres de la cadena a primerNombre
const char* valorPrimerNombre = cadenaPrimerNombre.data();
int longitud = cadenaPrimerNombre.size();
longitud = (longitud < 10 ? longitud : 9);
strncpy(primerNombre, valorPrimerNombre, longitud);
primerNombre[longitud] = '\0'; // adjunta un carácter nulo a primerNombre
} // fin de la función establecerPrimerNombre
// obtiene el valor del saldo
double DatosCliente::obtenerSaldo() const
{
return saldo;
} // fin de la función obtenerSaldo

// establece el valor del saldo
void DatosCliente::establecerSaldo(double valorSaldo)
{
saldo = valorSaldo;
} // fin de la función establecerSaldo
```

En el programa, se crea un objeto ofstream para el archivo credito.dat. El segundo argumento para el constructor (ios::out | ios::binary) indica que vamos a abrir el archivo para salida en modo binario, lo cual es requerido si debemos escribir registros de longitud fija. Se escribe el objeto clienteEnBlanco en el archivo credito.dat asociado con el objeto ofstream llamado creditoSalida. Recuerde que el operador sizeof devuelve el tamaño en bytes del objeto contenido entre paréntesis (vea el capítulo 8). El primer argumento para la función write debe ser de tipo const char *. Sin embargo, el tipo de datos de &clienteEnBlanco es DatosCliente *. Para convertir &clienteEnBlanco en const char *, se utiliza el operador de conversión reinterpret_cast, por lo que la llamada a write se compila sin generar un error de compilación.

```
// Creación de un archivo de acceso aleatorio.
#include <iostream>
using std::cerr;
using std::endl;
using std::ios;

#include <fstream>
using std::ofstream;

#include <cstdlib>
using std::exit; // prototipo de la función exit

#include "DatosCliente.h" // definición de la clase DatosCliente

int main()
{
ofstream creditoSalida("credito.dat", ios::out | ios::binary);

// sale del programa si ofstream no pudo abrir el archivo
if (!creditoSalida)
{
   cerr << "No se pudo abrir el archivo." << endl;
   exit(1);
} // fin de if

DatosCliente clienteEnBlanco; // el constructor pone en ceros cada miembro de
datos

// escribe 100 registros en blanco en el archivo
for (int i = 0; i < 100; i++)
   creditoSalida.write(reinterpret_cast<const char*>(&clienteEnBlanco),
      sizeof(DatosCliente));
```

```
return 0;
} // fin de main
```

5. RESUMEN

Una excepción es un suceso de error dentro de la ejecución de un programa no muy frecuente, para poder hacer uso de la gestión de excepciones tengo aquí sí más más nos intriga a la instrucción try…cach para procesar y prevenir la ocurrencia de errores que desencadenen la terminación abrupta de la ejecución de un programa, los mecanismos de implementación de la gestión decepciones nos permite tomar medidas sobre la ocurrencia de errores evitando que el programa termine de forma inesperada dando la oportunidad que se pueda recuperar de dicha ocurrencia y a la vez le permite al programador tomar medidas para poder resolver los problemas presentados por la ocurrencia del error; dentro del bloque try se colocan las instrucciones que podrían generar excepciones, mientras que en el bloque catch se colocan la respuesta a la ocurrencia de dicha excepción, de manera programada podemos utilizar la instrucción throw para el lanzamiento de decepciones.

el procesamiento de archivos se lleva a cabo a través de flujos que son utilizados para la inserción de información dentro de archivos en disco y el proceso inverso de la extracción de los datos almacenados. se consideran dos tipos de archivos diferentes los de acceso secuencial y los de acceso aleatorio, ambos tipos tienen sus propias características especiales; los archivos de acceso secuencial procesan 1 a 1 los registros son las líneas almacenadas o que van a ir grabando, mientras que los archivos de acceso aleatorio se basan en una longitud de registro fija para utilizando la función seekp() saltar de registro en registro de forma aleatoria; los procesos que se llevan a cabo con el manejo de archivos responden a los mismos procesos que se hace con las funciones de la librería STDIO, a diferencia del anterior ahora se utilizan flujos para los cuales utilizamos los operadores de ingreso y salida de flujo sobre objetos de los tipos ofstream e ifstream, al igual que en las versiones anteriores estamos haciendo uso de modificadores de acceso para los archivos los cuales nos van a permitir abrir los archivos para un tipo de operación específica haciendo diferencia entre creación de archivos y escritura lectura y agregación de datos.

IV

(La práctica tiene una duración de 4 horas) ACTIVIDADES

ENUNCIADO

Diseñar un programa de gestión de inventario para una tienda de productos electrónicos. El programa debe permitir realizar las siguientes operaciones:

1. Registrar un nuevo producto en el inventario, ingresando su nombre, precio y cantidad disponible.
2. Actualizar la información de un producto existente en el inventario, como su precio o cantidad.
3. Realizar una venta de un producto, reduciendo la cantidad disponible en el inventario.
4. Generar un informe de inventario que muestre todos los productos registrados, su precio y cantidad disponible.

El programa debe utilizar archivos secuenciales para almacenar la información del inventario, permitiendo agregar nuevos productos, actualizar la información existente y realizar consultas sobre el inventario. Además, se debe implementar el acceso aleatorio a los registros del archivo para permitir la búsqueda eficiente de productos por su nombre u otra característica.

El programa también debe incluir el manejo de excepciones para garantizar la integridad de los datos y evitar errores en la ejecución. Se deben manejar posibles excepciones, como la entrada de datos incorrectos, acceso a archivos no encontrados o intentos de venta de productos con una cantidad disponible insuficiente.

1. EXPERIENCIA DE PRÁCTICA N° 01: DISEÑO DEL PROGRAMA

1. : Definir la estructura de datos necesaria para almacenar la información del inventario, como una clase "Producto" con atributos como nombre, precio y cantidad.

2. Diseñar las funciones o métodos necesarios para realizar las operaciones mencionadas anteriormente.
3. Agregar un nuevo producto, actualizar la información existente, realizar ventas y generar informes.

2. EXPERIENCIA DE PRÁCTICA N° 02: IMPLEMENTACIÓN DE LA GESTIÓN DE ARCHIVOS SECUENCIALES

1. Utilizar las funciones de lectura y escritura de archivos secuenciales para almacenar y recuperar la información del inventario.
2. Agregar un nuevo producto, se debe escribir el registro en el archivo.
3. Actualizar o vender un producto, se deben realizar las modificaciones correspondientes en el archivo.

3. EXPERIENCIA DE PRÁCTICA N° 03: IMPLEMENTACIÓN DEL ACCESO ALEATORIO

1. Crear un archivo que contenga los registros de productos que deseas buscar.
2. Puedes utilizar un formato de archivo adecuado, como CSV (valores separados por comas) o JSON (notación de objetos de JavaScript), para almacenar la información de cada producto.
3. Para facilitar la búsqueda eficiente, puedes utilizar índices o estructuras adicionales en tu archivo de registros.
4. Puedes crear un índice que almacene los nombres de los productos y las ubicaciones de los registros correspondientes en el archivo principal. Esto permitirá ubicar rápidamente el registro deseado a través del índice en lugar de recorrer todo el archivo.
5. Desarrolla funciones o métodos que permitan acceder al archivo utilizando la técnica de acceso aleatorio.
6. Estas funciones deben utilizar los índices o estructuras adicionales para buscar los registros de manera eficiente.
7. Puedes implementar una función que reciba el nombre de un producto como parámetro y utilice el índice para encontrar su ubicación en el archivo principal.

4. EXPERIENCIA DE PRÁCTICA N° 04: IMPLEMENTACIÓN DE GESTIÓN DE EXCEPCIONES

1. Investiga y comprende los conceptos básicos de la gestión de excepciones en programación orientada a objetos.
2. Aprende sobre los bloques try-catch y cómo se utilizan para capturar y manejar excepciones en C++.
3. Define el objetivo de tu programa de práctica, que puede ser resolver un problema específico o simular una situación que requiera el manejo de excepciones.
4. Diseña las clases y métodos necesarios para implementar el programa.
5. Identifica las posibles excepciones que pueden ocurrir en el programa y planifica cómo manejarlas.
6. Escribe el código en C++ siguiendo el diseño que has creado.
7. Utiliza los bloques try-catch para capturar y manejar las excepciones en los lugares apropiados del código.
8. Implementa el código necesario para manejar las excepciones de manera adecuada, como mostrar mensajes de error o tomar acciones correctivas.
9. Ejecuta el programa y realiza pruebas para asegurarte de que el manejo de excepciones funciona correctamente.
10. Introduce casos de prueba que generen las excepciones esperadas y verifica que sean manejadas adecuadamente.
11. Utiliza herramientas de depuración para identificar y corregir posibles errores en tu código.
12. Analiza tu código y busca oportunidades de mejora.
13. Considera la posibilidad de utilizar jerarquías de excepciones para clasificar y manejar diferentes tipos de excepciones de manera más eficiente.
    Realiza cambios en tu código para optimizar su legibilidad, rendimiento y mantenibilidad.

V

EJERCICIOS

1. Escribe un programa en C++ que solicite al usuario el nombre de un archivo de texto. Luego,

lee el contenido del archivo y muestra su contenido por pantalla. Si el archivo no existe, el programa debe mostrar un mensaje de error adecuado.

2. Crea una función en C++ llamada "divide" que acepte dos números enteros como parámetros. La función debe dividir el primer número por el segundo número y devolver el resultado. Sin embargo, si el segundo número es igual a cero, la función debe lanzar una excepción de tipo "std::runtime_error" con un mensaje indicando que no se puede dividir por cero. Luego, en el programa principal, llama a la función "divide" con diferentes valores y maneja las excepciones adecuadamente.
3. Implementa una clase en C++ llamada "Persona" con los siguientes atributos: nombre (string), edad (int) y dirección (string). Añade los métodos necesarios para establecer y obtener los valores de los atributos. Luego, utiliza una biblioteca de serialización en C++ (como "Boost.Serialization" o "Cereal") para serializar un objeto de la clase "Persona" en un archivo binario. A continuación, deserializa el objeto desde el archivo y muestra sus atributos por pantalla.

VI
CUESTIONARIO

1. ¿Qué es un error de ejecución?
2. ¿Qué son las excepciones en el desarrollo de software?
3. Menciona tres ejemplos comunes de situaciones excepcionales que pueden ocurrir durante la ejecución de un programa.
4. ¿Cuál es la diferencia entre excepciones comprobadas y excepciones no comprobadas en Java?
5. Proporciona tres ejemplos de excepciones comprobadas en Java.
6. Enumera tres ejemplos de excepciones no comprobadas en Java.
7. ¿Cuál es la relación entre las excepciones no comprobadas y las excepciones RuntimeException en Java?
8. ¿Qué significa manejar una excepción en Java? ¿Cuáles son las opciones para manejar excepciones?
9. ¿Qué es la serialización en programación orientada a objetos?
10. ¿Cuál es el propósito de la serialización en Java?
11. ¿Qué interfaz se utiliza en Java para lograr la serialización de objetos?
12. Menciona dos métodos de la interfaz Serializable en Java.
13. ¿Qué es un archivo secuencial en el contexto de manejo de archivos en Java?
14. ¿Cuáles son las operaciones básicas que se pueden realizar con archivos secuenciales en Java?
15. ¿Qué es un archivo de acceso aleatorio y cómo difiere de un archivo secuencial?
16. Menciona dos operaciones que se pueden realizar con archivos de acceso aleatorio en Java.
17. ¿Cuál es la ventaja de utilizar archivos de acceso aleatorio en lugar de archivos secuenciales?
18. ¿Qué clases y métodos se utilizan en Java para trabajar con flujos de datos de entrada y salida?
19. ¿Cuál es la diferencia entre flujos de bytes y flujos de caracteres en Java?
20. ¿Qué clase se utiliza para leer datos de un archivo en Java?
21. ¿Cuál es la diferencia entre FileReader y FileInputStream en Java?
22. ¿Qué clase se utiliza para escribir datos en un archivo en Java?
23. ¿Cuál es la diferencia entre FileWriter y FileOutputStream en Java?
24. ¿Qué es la accesibilidad de archivos PDF y por qué es importante?
25. ¿Qué herramienta se puede utilizar para crear y verificar la accesibilidad de archivos PDF?
26. ¿Cuál es el propósito principal de los flujos y archivos en Java?

VII
BIBLIOGRAFIA Y REFERENCIAS

- Deitel, Paul J., Deitel, Harvey M., "Cómo Programar en C++", 6ta Edición, Ed. Pearson Educación, México 2009.
- Stroustrup, Bjarne, "El Lenguaje de Programación C++", 3ra Edición, Adisson Pearson Educación S.A., Madrid 2002.
- Eckel, Bruce, "Thinking in C++", 2da Edición, Prentice Hall, 2000.